DECOMPOSITION METHODS FOR INTEGER LINEAR PROGRAMMING

by

Matthew Galati

Presented to the Graduate and Research Committee
of Lehigh University
in Candidacy for the Degree of
Doctor of Philosophy

in

Industrial and Systems Engineering

Lehigh University

November 2009

Approved and recommended for acceptance as a dissertation in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

____
Date

____
Dr. Theodore K. Ralphs
Dissertation Advisor

____
Accepted Date

Committee:

____
Dr. Theodore K. Ralphs, Chairman

____
Dr. Francisco Barahona

____
Dr. Joseph C. Hartman

____
Dr. Jeffrey T. Linderoth

Acknowledgments

I owe my deepest gratitude to my advisor, Theodore Ralphs, for his encouragement, guidance, and support throughout this entire process. From our first meeting many years ago to our final editing days, he has shown a great deal of enthusiasm for the topic and patience, allowing me the time to fully develop an understanding of the subject. I would also like to thank my thesis committee, Francisco Barahona, Joseph Hartman, and Jeffrey Linderoth, for graciously giving so much of their time in support of completing this work.

This thesis would not have been possible without the amazing support and understanding I received from management at SAS Institute. In particular, special thanks to Manoj Chari, Trevor Kearney, and Radhika Kulkarni. The flexibility of the working arrangement allowed for the time to complete this work while still pursuing my career goals. The experience gained at SAS during these past few years has greatly benefited the development of this research.

Throughout this journey I have had the pleasure of working with excellent colleagues from all over the globe. Each and every one of them has helped in my understanding of the subject. In particular I would like to point out a few people that helped me along the way. Going all the way back to my undergraduate days in the Math Department at Stetson University, my mentors Erich Friedman and Margie Hale inspired me to love and appreciate the field of Mathematics. I cherish those days more than any other in my life and looking back, I owe a great deal of gratitude to Erich and Margie for guiding me towards this path. From my very first days at Lehigh University, Joe Hartman, Jeff Linderoth, Ted Ralphs, George Wilson, and Bob Storer have been great friends, mentors, and colleagues. Despite the fact that I dominated Joe, Jeff, and Bob on the basketball court and softball field, and Ted in the weightroom, they never held this against me and continued with

their encouragement and support throughout the years.

So many other people are to thank during my years at Stetson, Lehigh, IBM, and SAS. In particular: Jeff Fay, Rita Frey, Kathy Rambo, Ashutosh Mahajan, Mustafa Kılınç, Menal Güzelsoy, João Gonçalves, Mark Booth, John Forrest, Selçuk Avcı, Gardner Pomper, Alper Uygur, Ivan Oliveira, and Yan Xu. An extra special thanks goes to Rob Pratt, who inspired many of the ideas in this thesis and also helped edit the final draft. I have spent countless hours working together with Rob on projects at SAS and have benefited greatly from his expertise and ability to envision models and algorithms in innovative ways with the end-goal of providing value to our clients.

Finally, and perhaps most important, this thesis would not have been possible without the unquestioning support, sacrifice, and patience of my family and friends. Thanks in particular to Rose Galati (Mom), Victor Galati (Dad), Chris Galati (Brother), Pauline Magliacano (Grandma), Ruby *Red Dress* Galati (Dog), and Jessica Nash for your love and encouragement. You always helped me to put everything in perspective, allowing me to focus on those things in life that really matter. I love you all very much.

Contents

Acknowledgments **iii**

Contents **iv**

List of Tables **viii**

List of Figures **ix**

Abstract **1**

1 Introduction **3**

1.1 Background Definitions and Notation 6

1.2 The Principle of Decomposition 8

1.3 Computational Software for Decomposition Methods 14

1.4 Contributions 16

1.5 Outline of the Thesis 18

2 Decomposition Methods **20**

2.1 Traditional Decomposition Methods 20

2.1.1 Cutting-Plane Method 21

2.1.2 Dantzig-Wolfe Method 26

2.1.3 Lagrangian Method 35

2.2 Integrated Decomposition Methods 38

2.2.1 Price-and-Cut 39
2.2.2 Relax-and-Cut 51
2.3 Decompose-and-Cut 53
2.3.1 The Template Paradigm and Structured Separation 54
2.3.2 Separation Using an Inner Approximation 62
2.3.3 Decomposition Cuts 66

3 Algorithmic Details **69**
3.1 Branching for Inner Methods 70
3.2 Relaxation Separability 75
3.2.1 Identical Subproblems 77
3.2.2 Price-and-Branch 80
3.3 Nested Pricing 81
3.4 Initial Columns 84
3.5 Standard MILP Cutting Planes for Inner Methods 85
3.6 Compression of Master LP and Object Pools 86
3.7 Solver Choice for Master Problem 86

4 DIP Software **89**
4.1 Design 91
4.1.1 The Application Interface 92
4.1.2 The Algorithm Interface 94
4.1.3 Interface with ALPS 95
4.2 Interface with CGL 96
4.3 Creating an Application 97
4.3.1 Small Integer Program 97
4.3.2 Generalized Assignment Problem 100
4.3.3 Traveling Salesman Problem 103
4.4 Other Examples 103

5 Applications and Computational Results **106**

5.1 Multi-Choice Multi-Dimensional Knapsack 107

5.1.1 Results on Integrated Methods 109

5.1.2 Results using Nested Pricing 112

5.1.3 Comparison of Master Solver 113

5.2 ATM Cash Management Problem 119

5.2.1 Mixed Integer Nonlinear Programming Formulation 119

5.2.2 Mixed Integer Linear Programming Approximation. 122

5.2.3 Results 127

5.3 Automated Decomposition for Block Angular MILP 128

6 Future Research **133**

A Detailed Tables of Results **136**

Bibliography **140**

List of Tables

4.1 COIN-OR Projects used by DIP 92

4.2 Basic Classes in DIP Interfaces 92

4.3 COIN/DIP Applications 105

5.1 MMKP: CPX10.2 vs CPM/PC/DC (Summary Table) 110

5.2 MMKP: PC vs PC Nested with MC2KP and MMKP (Summary Table) 114

5.3 ATM: CPX11 vs PC/PC+ (Summary Table) 130

5.4 MILPBlock Retail: CPX11 vs PC (Summary Table) 132

A.1 MMKP: CPX10.2 vs CPM (Detailed Table) 137

A.2 MMKP: PC vs DC (Detailed Table) 138

A.3 MMKP: PC-M2 vs PC-MM (Detailed Table) 139

List of Figures

1.1 Polyhedra (Example 1: SILP) 11

2.1 Outline of the cutting-plane method 22

2.2 Cutting-plane method (Example 1: SILP) 25

2.3 Finding violated inequalities in the cutting-plane method (Example 3a: TSP) . . . 26

2.4 Outline of the Dantzig-Wolfe method 28

2.5 Dantzig-Wolfe method (Example 1: SILP) 31

2.6 Dantzig-Wolfe method (Example 3a: TSP) 32

2.7 The relationship of $\mathcal{P}' \cap \mathcal{Q}''$, conv $(\mathcal{S}(\hat{u}_{\text{DW}}, \hat{\alpha}_{\text{DW}})) \cap \mathcal{Q}''$, and the face F. 36

2.8 Outline of the Lagrangian method 37

2.9 Outline of the price-and-cut method 40

2.10 Price-and-cut method (Example 1: SILP) 42

2.11 Price-and-cut method (Example 3a: TSP) 44

2.12 Finding violated inequalities in price-and-cut (Example 3b: TSP) 46

2.13 Solving the cutting subproblem with the aid of a decomposition 47

2.14 Using the optimal decomposition to find violated inequalities in price-and-cut (Example 3a: TSP) 49

2.15 Using the optimal decomposition to find violated inequalities in price-and-cut (Example 3b: TSP) 50

2.16 Outline of the relax-and-cut method 52

2.17 Example of a decomposition into b-matchings and k-DCTs 61

2.18 Example of decomposition VRP/k-TSP 62
2.19 Separation in the decompose-and-cut method 64
2.20 Outline of the decomposition method for decompose-and-cut 65
2.21 Decompose-and-cut 66

3.1 Branching in the Dantzig-Wolfe method (Example 1: SILP) 72

4.1 Inheritance Diagram for `DecompAlgo` 96

5.1 MMKP: CPX10.2 vs CPM/PC/DC (Performance Profile) 111
5.2 MMKP: PC vs PC Nested with MC2KP and MMKP (Performance Profile) 115
5.3 MMKP: CPX10.2 vs CPM/PC/DC/PC-M2/PC-MM (Performance Profile) 115
5.4 MMKP: CPX10.2 vs CPM/PC/DC/PC-M2/PC-MM (Stacked Bar Chart) 116
5.5 MMKP: Comparison of Primal vs Dual Simplex for Master LP solver (PC/DC) . . 117
5.6 MMKP: Comparison of Primal vs Dual Simplex for Master LP solver (PC-M2/PC-MM) 118
5.7 ATM: CPX11 vs PC/PC+ (Performance Profiles) 129
5.8 ATM: CPX11 vs PC/PC+ (Stacked Bar Chart) 131

Abstract

In this research, we present a theoretical and computational framework for using the principle of decomposition to solve mixed integer linear programs (MILP). We focus on the common threads among three traditional methods for generating approximations to the convex hull of feasible solutions to a MILP. These include a method employing an outer approximation, the cutting plane method, as well as two related methods employing inner approximations, the Dantzig-Wolfe method and the Lagrangian method. We then extend these traditional methods by allowing for the use of both outer and inner approximation simultaneously. This lends itself to the development of two bounding methods that generate even stronger bounds, known as price-and-cut and relax-and-cut.

We introduce a relatively unknown integrated method, called *decompose-and-cut*, which was originally inspired by the fact that separation of structured solutions is frequently easier than separation of arbitrary real vectors. We present its use in the standard cutting-plane method and introduce a class of cutting-planes called *decomposition cuts*. These cuts serve to break the template paradigm by using information from an implicitly defined polyhedron, similar to what can be done in price-and-cut.

Next, we focus some attention on the implementation of branch-and-price-and-cut methods based on Dantzig-Wolfe decomposition. We describe a number of algorithmic details discovered during the development of DIP. We then present some applications developed in DIP and provide some computational results on the effectiveness of some of these ideas.

We describe DIP (**D**ecomposition for **I**nteger **P**rogramming), a new open-source software framework which provides the algorithmic shell for implementation of these methods. DIP has been designed with the goal of providing a user with the ability to easily utilize various traditional and

integrated decomposition methods while requiring only the provision of minimal problem-specific algorithmic components. We provide numerous examples to help solidify the understanding of how a user would interface with the framework.

To demonstrate the effectiveness of these ideas in practice, we describe details of applications written in support of work done while employed at SAS Institute. For each, we present computational results showing the effectiveness of the framework in practice. We present the Multi-Choice Multi-Dimensional Knapsack Problem, an important subproblem used in the algorithms present in SAS Marketing Optimization, which attempts to improve the return-on-investment for marketing campaign offers. We introduce an application from the banking industry for ATM cash management that we worked on for the Center of Excellence in Operations Research at SAS Institute. We model the problem as a mixed integer nonlinear program and create an application in DIP, to solve an approximating MILP. Finally, we present another application developed in DIP, called MILPBlock, which provides a black-box framework for using these integrated methods on generic MILPs that have some block angular structure. The ability to develop a software framework that can handle these methods in an application-independent manner relies on the conceptual framework proposed. DIP is the first of its kind in this respect and should greatly break down the barriers of entry into developing solvers based on these methods. We present some computational results using MILPBlock on a model presented to us from SAS Retail Optimization.

Chapter 1

Introduction

Within the field of mathematical programming, discrete optimization has become the focus of a vast body of research and development due to the increasing number of industries that are now using it to model the decision analysis for their most complex systems. Mixed integer linear programming problems involve minimizing (or maximizing) the value of some linear function over a polyhedral feasible region subject to integrality restrictions on some of the variables. More formally, a *mixed integer linear program* (MILP) can be defined as

$$\min_{x \in \mathbb{R}^n} \left\{ c^\top x \mid Ax \geq b, x_i \in \mathbb{Z} \; \forall i \in I \right\}, \tag{1.1}$$

for a given cost vector $c \in \mathbb{Q}^n$, where $A \in \mathbb{Q}^{m \times n}$ is the constraint matrix, $b \in \mathbb{Q}^m$ is the right hand side, and $I \subseteq \{1, \ldots, n\}$ is the index set of variables that are restricted to integer values. Two important special cases are when $I = \{1, \ldots, n\}$, which we refer to as a *(pure) integer linear program* (ILP) and when $I = \emptyset$, which we refer to as a *linear program* (LP).

Solving an MILP is known to be an $\mathcal{NP}$-hard problem in general [34]. However, due to recent breakthroughs in both the underlying theory and available computer implementations, discrete optimization is now a viable tool for optimizing some of the most complex systems. We are just now starting to understand the impact that discrete optimization can have in helping organizations optimize the efficiency of their processes. In the past two decades, MILP has seen widespread adoption in a large and diverse array of industries, including: logistics, finance, medical research, engineering

design, retail, and many others.

In the following paragraphs, we attempt to put into context the direction of our research. For this purpose, we assume the reader has a working knowledge of the theory and practice of integer programming. In Sections 1.1 and 1.2, we present a more formal treatment of the relevant definitions and notation. For an in-depth treatment of the theory of integer programming, we direct the reader to the works of Schrijver [83], Nemhauser and Wolsey [70], and Wolsey [93].

One of the most successful algorithms developed to date for solving MILPs is the *branch-and-bound* method [50]. Branch and bound is a divide-and-conquer approach that reduces the original problem to a series of smaller subproblems and then recursively solves each subproblem. This dissertation focuses on the development of a theoretical and computational framework for computing strong bounds to help improve the performance of branch-and-bound methods. Most bounding procedures for MILPs are based on the iterative construction and improvement of polyhedral approximations of $\mathcal{P}$, the *convex hull* of feasible solutions. Solving an optimization problem over such a polyhedral approximation, provided it fully contains $\mathcal{P}$, produces a bound that can be used to drive a branch-and-bound algorithm. The effectiveness of the bounding procedure depends largely on how well $\mathcal{P}$ can be approximated. The most straightforward approximation is the *continuous approximation*, consisting simply of the linear constraints present in the original formulation. The bound resulting from this approximation is frequently too weak to be effective, however. In such cases, it can be improved by dynamically generating additional polyhedral information that can be used to augment the approximation.

Traditional dynamic procedures for augmenting the continuous approximation can be grouped roughly into two categories. *Cutting plane methods* improve the approximation by dynamically generating half-spaces that contain $\mathcal{P}$ but not the continuous approximation, i.e., valid inequalities. These half-spaces are then intersected with the current approximation, thereby improving it. With this approach, valid inequalities are generated by solution of an associated *separation problem*. Generally, the addition of each valid inequality reduces the hypervolume of the approximating polyhedron, resulting in a potentially improved bound. Because they dynamically generate part of the description of the final approximating polyhedron as the intersection of half-spaces (an *outer*

representation), we refer to cutting plane methods as *outer approximation methods*. Traditional *column-generation methods*, on the other hand, improve the approximation by dynamically generating the extreme points of a polyhedron containing $\mathcal{P}$, which is again intersected with the continuous approximation, as in the cutting plane method, to yield a final approximating polyhedron. In this case, each successive extreme point is generated by solution of an associated *optimization problem* and at each step, the hypervolume of the approximating polyhedron is increased. Because decomposition methods dynamically generate part of the description of the approximating polyhedron as the convex hull of a finite set (an *inner representation*), we refer to these methods as *inner approximation methods*.

Both inner and outer methods work roughly by alternating between a procedure for computing solution and bound information (the *master problem*) and a procedure for augmenting the current approximation (the *subproblem*). The two approaches, however, differ in important ways. Outer methods require that the master problem produce "primal" solution information, which then becomes the input to the subproblem, a *separation problem*. Inner methods require "dual" solution information, which is then used as the input to the subproblem, an *optimization problem*. In this sense, the two approaches can be seen as "dual" to one another. A more important difference, however, is that the valid inequalities generated by an inner method can be valid with respect to *any* polyhedron containing $\mathcal{P}$ (see Section 2.3.1), whereas the extreme points generated by an inner method must come from a single polyhedron, or some *restriction* of that polyhedron (see Section 3.3). Procedures for generating new valid inequalities can also take advantage of knowledge of previously generated valid inequalities to further improve the approximation, whereas with inner methods, such "backward-looking" procedures do not appear to be possible. Finally, the separation procedures used in the cutting plane method can be heuristic in nature as long as it can be proven that the resulting half-spaces do actually contain $\mathcal{P}$. Although heuristic methods can be employed in solving the optimization problems required of an inner method, valid bounds are obtained only when using exact optimization for some valid relaxation. On the whole, outer methods have proven to be more flexible and powerful, and this is reflected in their position as the approach of choice for solving most MILPs.

As we show, however, inner methods do still have an important role to play. Although inner and outer methods have traditionally been considered separate and distinct, it is possible, in principle, to integrate them in a straightforward way. By doing so, we obtain bounds at least as good as those yielded by either approach alone. In such an integrated method, one alternates between a master problem that produces both primal and dual information, and either one of two subproblems, one an optimization problem and the other a separation problem. This may result in significant synergy between the subproblems, as information generated by solving the optimization subproblem can be used to generate cutting planes and vice versa.

The theoretical framework tying together these different bounding methods only starts to scratch the surface. The computational nuances of standard approaches to MILP, like branch-and-cut, are just beginning to be better understood. Although much of the theory on these standard methods has been known for decades [70], real performance improvements are just starting to be realized [14]. Column-generation methods, traditional and integrated, are even less understood. The basic theory has also been around for quite some time [10]. However, computational success stories have been limited to a small number of industries. In addition, the ability to apply these methods has relied heavily on application-specific techniques. In this research, we attempt to generalize many of the algorithmic enhancements under one umbrella framework that does not depend on the structure of a specific application.

1.1 Background Definitions and Notation

For ease of exposition, we consider only pure integer linear programs with bounded, nonempty feasible regions throughout this thesis, although the methods presented herein can be extended to more general settings. For the remainder of the paper, we consider an ILP whose feasible set consists of the integer vectors contained in the polyhedron $\mathcal{Q} = \{x \in \mathbb{R}^n \mid Ax \geq b\}$, where $A \in \mathbb{Q}^{m \times n}$ and $b \in \mathbb{Q}^m$. Let $\mathcal{F} = \mathcal{Q} \cap \mathbb{Z}^n$ be the feasible set and let $\mathcal{P}$ be the convex hull of $\mathcal{F}$. The canonical *optimization problem* for $\mathcal{P}$ is that of determining

$$z_{\text{IP}} = \min_{x \in \mathbb{Z}^n} \left\{ c^\top x \mid Ax \geq b \right\} = \min_{x \in \mathcal{F}} \left\{ c^\top x \right\} = \min_{x \in \mathcal{P}} \left\{ c^\top x \right\} \tag{1.2}$$

for a given cost vector $c \in \mathbb{Q}^n$, where $z_{\text{IP}} = \infty$ if $\mathcal{F}$ is empty. We refer to the optimization over some polyhedron $\mathcal{P}$ for a given cost vector c as OPT$(\mathcal{P}, c)$. In what follows, we also consider the equivalent decision version of this problem, which is to determine, for a given upper bound U, whether there is a member of $\mathcal{P}$ with objective function value strictly better than U. We denote by OPT$(\mathcal{P}, c, U)$ a subroutine for solving this decision problem. The subroutine is assumed to return either the empty set, or a set of one *or more* (depending on the situation) members of $\mathcal{P}$ with objective value better than U.

A related problem is the *separation problem* for $\mathcal{P}$, which is typically already stated as a decision problem. Given $x \in \mathbb{R}^n$, the problem of separating x from $\mathcal{P}$ is that of deciding whether $x \in \mathcal{P}$ and if not, determining $a \in \mathbb{R}^n$ and $\beta \in \mathbb{R}$ such that $a^\top y \geq \beta \; \forall y \in \mathcal{P}$ but $a^\top x < \beta$. A pair $(a, \beta) \in \mathbb{R}^{n+1}$ such that $a^\top y \geq \beta \; \forall y \in \mathcal{P}$ is a *valid inequality* for $\mathcal{P}$ and is said to be *violated* by $x \in \mathbb{R}^n$ if $a^\top x < \beta$. We denote by SEP$(\mathcal{P}, x)$ a subroutine that separates an arbitrary vector $x \in \mathbb{R}^n$ from polyhedron $\mathcal{P}$, returning either the empty set or a set of one or more violated valid inequalities. The inequalities returned from this subroutine are referred to as *cuts*. Note that the optimization form of the separation problem is that of finding the inequality *most violated* by a point $y \notin \mathcal{P}$ and is equivalent to the decision form stated here.

A closely related problem is the *facet identification problem*, which restricts the generated inequalities to only those that are *facet-defining* for $\mathcal{P}$. In [39], it was shown that the facet identification problem for $\mathcal{P}$ is polynomially equivalent to the optimization problem for $\mathcal{P}$ (in the worst-case sense). However, a theme that arises in what follows is that the complexity of optimization and separation can vary significantly if either the input or the output must have known structure. If the solution to an optimization problem is required to be integer, the problem generally becomes much harder to solve. On the other hand, if the input vector to a separation problem is an integral vector, then the separation problem frequently becomes much *easier* to solve in the worst case. From the dual point of view, if the input cost vector of an optimization problem has known structure, such as integrality of certain components; this may make the problem easier. Requiring the output of the separation problem to have known structure is known as the *template paradigm* and may also make the separation problem easier, but such a requirement is essentially equivalent to enlarging $\mathcal{P}$.

These concepts are discussed in more detail in Section 2.3.1.

1.2 The Principle of Decomposition

We now formalize some of the notions described in the introduction. Implementing a branch-and-bound algorithm for solving an ILP requires a procedure that will generate a lower bound as close as possible to the optimal value z_{IP}. The most commonly used method of bounding is to solve the linear programming relaxation obtained by removing the integrality requirement from the ILP formulation. The *LP bound* is given by

$$z_{\text{LP}} = \min_{x \in \mathbb{R}^n} \left\{ c^\top x \mid Ax \geq b \right\} = \min_{x \in \mathcal{Q}} \left\{ c^\top x \right\} \tag{1.3}$$

and is obtained by solving a linear program with the original objective function c over the polyhedron $\mathcal{Q}$. It is clear that $z_{\text{LP}} \leq z_{\text{IP}}$ since $\mathcal{P} \subseteq \mathcal{Q}$. This LP relaxation is usually much easier to solve than the original ILP, but z_{LP} may be arbitrarily far away from z_{IP} in general, so we need to consider more effective procedures.

In most cases, the description of $\mathcal{Q}$ is small enough that it can be represented explicitly and the bound computed using a standard linear programming algorithm. To improve the LP bound, decomposition methods construct a second approximating polyhedron that can be intersected with $\mathcal{Q}$ to form a better approximation. Unlike $\mathcal{Q}$, this second polyhedron usually has a description of exponential size, and we must generate portions of its description dynamically. Such a dynamic procedure is the basis both for cutting plane methods [21, 73] , which generate an outer approximation, and for column-generation methods, such as the Dantzig-Wolfe method [23] and the Lagrangian method [31, 11], which generate inner approximations.

For the remainder of this section, we consider the relaxation of (1.2) defined by

$$\min_{x \in \mathbb{Z}^n} \left\{ c^\top x \mid A'x \geq b' \right\} = \min_{x \in \mathcal{F}'} \left\{ c^\top x \right\} = \min_{x \in \mathcal{P}'} \left\{ c^\top x \right\}, \tag{1.4}$$

where $\mathcal{F} \subset \mathcal{F}' = \{x \in \mathbb{Z}^n \mid A'x \geq b'\}$ for some $A' \in \mathbb{Q}^{m' \times n}, b' \in \mathbb{Q}^{m'}$ and $\mathcal{P}'$ is the convex

hull of $\mathcal{F}'$. Along with $\mathcal{P}'$ is associated a set of *side constraints* $[A'', b''] \in \mathbb{Q}^{m'' \times (n+1)}$ such that $\mathcal{Q} = \{x \in \mathbb{R}^n \mid A'x \geq b', A''x \geq b''\}$. We denote by $\mathcal{Q}'$ the polyhedron described by the inequalities $[A', b']$ and by $\mathcal{Q}''$ the polyhedron described by the inequalities $[A'', b'']$. Thus, $\mathcal{Q} = \mathcal{Q}' \cap \mathcal{Q}''$ and $\mathcal{F} = \{x \in \mathbb{Z}^n \mid x \in \mathcal{P}' \cap \mathcal{Q}''\}$. We often refer to $\mathcal{Q}'$ as the *relaxed polyhedron*. For the decomposition to be effective, we must have that $\mathcal{P}' \cap \mathcal{Q}'' \subset \mathcal{Q}$, so that the bound obtained by optimizing over $\mathcal{P}' \cap \mathcal{Q}''$ is at least as good as the LP bound and strictly better for some objective functions. The description of $\mathcal{Q}''$ must also be *compact* so that we can construct it explicitly. Finally, we assume that there exists an *effective* algorithm for optimizing over $\mathcal{P}'$ and thereby, for separating arbitrary real vectors from $\mathcal{P}'$. We are deliberately using the term *effective* here to denote an algorithm that has an acceptable average-case running time, since this is more relevant than worst-case behavior in our computational framework. Note, throughout this research, we are assuming that the efficiency of the algorithm used for solving OPT$(\mathcal{P}', c)$ is not affected by the structure of the cost vector c.

Traditional decomposition methods can all be viewed as techniques for iteratively computing the bound

$$z_{\mathrm{D}} = \min_{x \in \mathcal{P}'} \left\{ c^\top x \mid A''x \geq b'' \right\} = \min_{x \in \mathcal{F}' \cap \mathcal{Q}''} \left\{ c^\top x \right\} = \min_{x \in \mathcal{P}' \cap \mathcal{Q}''} \left\{ c^\top x \right\}. \tag{1.5}$$

In Section 2.1, we review the cutting plane method, the Dantzig-Wolfe method, and the Lagrangian method, all classical approaches that can be used to compute this bound. This common perspective motivates Section 2.2, where we consider more advanced decomposition methods called *integrated decomposition methods*, in which both inner *and* outer approximation techniques are used in tandem. To illustrate the effect of applying the decomposition principle, we now introduce three simple examples that we build on throughout the thesis. The first is a simple generic ILP that we refer to as SILP (small integer linear program).

Example 1: SILP Let the following be the formulation of a given ILP:

$\min\ x_1,$

$$7\,x_1 - x_2 \geq 13, \quad (1.6) \qquad -\,x_1 - x_2 \geq -8, \quad (1.13)$$

$$x_2 \geq 1, \quad (1.7) \qquad -0.4\,x_1 + x_2 \geq 0.3, \quad (1.15)$$

$$-\,x_1 + x_2 \geq -3, \quad (1.8) \qquad x_1 + x_2 \geq 4.5, \quad (1.17)$$

$$-4\,x_1 - x_2 \geq -27, \quad (1.9) \qquad 3\,x_1 + x_2 \geq 9.5, \quad (1.19)$$

$$-\,x_2 \geq -5, \quad (1.10) \qquad 0.25\,x_1 - x_2 \geq -3, \quad (1.21)$$

$$0.2\,x_1 - x_2 \geq -4, \quad (1.11) \qquad x \in \mathbb{Z}^2. \quad (1.23)$$

In this example, we let

$$\begin{aligned}
\mathcal{P} &= \text{conv}\left\{x \in \mathbb{R}^2 \mid x \text{ satisfies } (1.6) - (1.22)\right\}, \\
\mathcal{Q}' &= \left\{x \in \mathbb{R}^2 \mid x \text{ satisfies } (1.6) - (1.11)\right\}, \\
\mathcal{Q}'' &= \left\{x \in \mathbb{R}^2 \mid x \text{ satisfies } (1.12) - (1.20)\right\}, \text{ and} \\
\mathcal{P}' &= \text{conv}(\mathcal{Q}' \cap \mathbb{Z}^2).
\end{aligned}$$

In Figure 1.1(a), we show the associated polyhedra, where the set of feasible solutions $\mathcal{F} = \mathcal{Q}' \cap \mathcal{Q}'' \cap \mathbb{Z}^2 = \mathcal{P}' \cap \mathcal{Q}'' \cap \mathbb{Z}^2$ and $\mathcal{P} = \text{conv}(\mathcal{F})$. Figure 1.1(b) depicts the continuous approximation $\mathcal{Q}' \cap \mathcal{Q}''$, while Figure 1.1(c) shows the improved approximation $\mathcal{P}' \cap \mathcal{Q}''$. For the objective function in this example, optimization over $\mathcal{P}' \cap \mathcal{Q}''$ leads to an improvement over the LP bound obtained by optimization over $\mathcal{Q}$. ■

In our second example, we consider the well-known *Generalized Assignment Problem* (GAP) [59]. The GAP, which is in the complexity class $\mathcal{NP}$-hard, has some interesting relaxations that we use to illustrate some of the ideas discussed throughout this work.

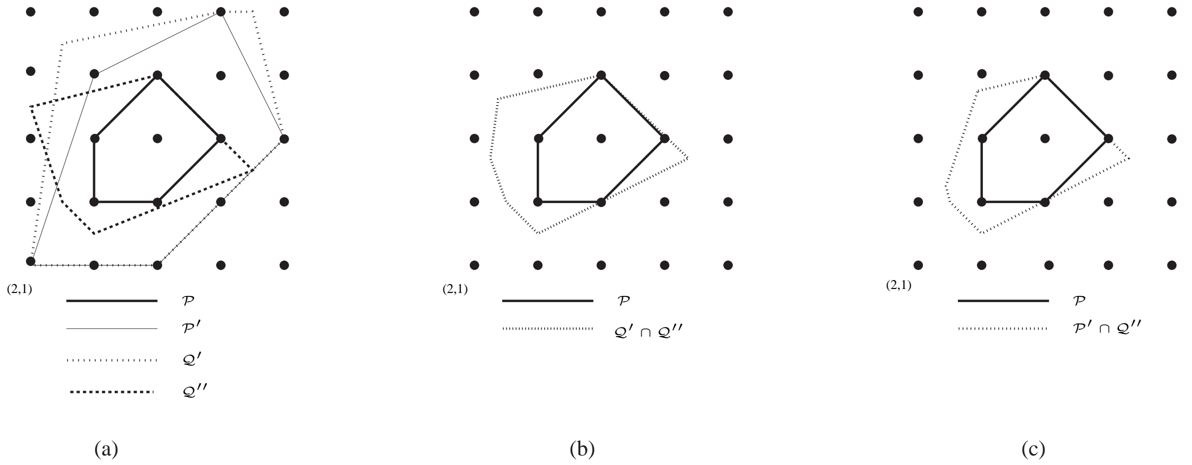


Figure 1.1: Polyhedra (Example 1: SILP)

Example 2: GAP The Generalized Assignment Problem (GAP) is that of finding a minimum cost *assignment* of n tasks to m machines such that each task is assigned to precisely one machine subject to capacity restrictions on the machines. With each possible assignment, we associate a binary variable x_{ij}, which, if set to 1, indicates that machine i is assigned to task j. For ease of notation, let us define two index sets $M = \{1, \ldots, m\}$ and $N = \{1, \ldots, n\}$. Then an ILP formulation of GAP is as follows:

$$\min \sum_{i \in M} \sum_{j \in N} c_{ij} x_{ij} ,$$

$$\sum_{j \in N} w_{ij} x_{ij} \leq b_i \ \forall i \in M, \tag{1.24}$$

$$\sum_{i \in M} x_{ij} = 1 \ \forall j \in N, \tag{1.25}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i, j \in M \times N. \tag{1.26}$$

In this formulation, equations (1.25) ensure that each task is assigned to exactly one machine. Inequalities (1.24) ensure that for each machine, the capacity restrictions are met.

One possible decomposition of GAP is to let the relaxation polyhedron be defined by the assignment constraints as follows:

$$\mathcal{P}' = \text{conv}\left\{x_{ij} \in \mathbb{R}^+ \; \forall i,j \in M \times N \mid x \text{ satisfies (1.25) and (1.26)}\right\},$$
$$\mathcal{Q}'' = \left\{x_{ij} \in \mathbb{R}^+ \; \forall i,j \in M \times N \mid x \text{ satisfies (1.24)}\right\}.$$

Unfortunately, for this decomposition, the polytope $\mathcal{P}'$ has the *integrality property*, which means that every extremal solution to its continuous relaxation is integral. In this case, the decomposition bound is no better than the standard continuous relaxation; i.e., $z_{\text{D}} = z_{\text{LP}}$. Therefore, if our goal is to generate tighter bounds, this is not a good choice for a relaxation. However, if we instead choose the capacity constraints as our relaxation, we get the following:

$$\mathcal{P}' = \text{conv}\left\{x_{ij} \in \mathbb{R}^+ \; \forall i,j \in M \times N \mid x \text{ satisfies (1.24) and (1.26)}\right\},$$
$$\mathcal{Q}'' = \left\{x_{ij} \in \mathbb{R}^+ \; \forall i,j \in M \times N \mid x \text{ satisfies (1.25)}\right\}.$$

In this case, the relaxation is a set of independent knapsack problems, which do not have the integrality property and are separable. Since there are many efficient algorithms for solving the knapsack problem, this is a good choice for the subproblem, as each one can be solved independently. ■

In our third example, we consider the classical *Traveling Salesman Problem* (TSP), a well-known combinatorial optimization problem [4]. The TSP, which is also in the complexity class $\mathcal{NP}$-hard, lends itself well to the application of the principle of decomposition, as the standard formulation contains an exponential number of constraints and has a number of well-solved combinatorial relaxations.

Example 3a: TSP The Traveling Salesman Problem is that of finding a minimum cost *tour* in an undirected graph G with vertex set $V = \{0, \ldots, n-1\}$ and edge set E. We assume without loss of generality that G is complete. A tour is a connected subgraph for which each node has degree two. The TSP is then to find such a subgraph of minimum cost, where the cost is the sum

of the costs of the edges comprising the subgraph. With each edge $e \in E$, we therefore associate a binary variable x_e, indicating whether edge e is part of the subgraph, and a cost $c_e \in \mathbb{R}$. Let $\delta(S) = \{\{i,j\} \in E \mid i \in S, j \notin S\}$, $E(S:T) = \{\{i,j\} \mid i \in S, j \in T\}$, $E(S) = E(S:S)$ and $x(F) = \sum_{e \in F} x_e$. Then an ILP formulation of the TSP is as follows:

$$
\begin{aligned}
\min \sum_{e \in E} & c_e x_e, \\
& x(\delta(\{i\})) = 2 \quad \forall i \in V, && (1.27) \\
& x(E(S)) \leq |S| - 1 \quad \forall S \subset V,\ 3 \leq |S| \leq n-1, && (1.28) \\
& 0 \leq x_e \leq 1 \quad \forall e \in E, && (1.29) \\
& x_e \in \mathbb{Z} \quad \forall e \in E. && (1.30)
\end{aligned}
$$

The convex hull of the *TSP polytope* is then

$$
\mathcal{P} = \text{conv}\left\{ x \in \mathbb{R}^E \mid x \text{ satisfies } (1.27) - (1.30) \right\}.
$$

The equations (1.27) are the *degree constraints*, which ensure that each vertex has degree two in the subgraph, while the inequalities (1.28) are known as the *subtour elimination constraints* (SECs) and enforce connectivity. Since there are an exponential number of SECs, it is impossible to explicitly construct the LP relaxation of TSP for large graphs. Following the pioneering work of Held and Karp [42], however, we can apply the principle of decomposition by employing the well-known *Minimum* 1-*Tree Problem*, a combinatorial relaxation of TSP.

A 1-tree is a tree spanning $V \setminus \{0\}$ plus two edges incident to vertex 0. A 1-tree is hence a subgraph containing exactly one cycle through vertex 0. The Minimum 1-Tree Problem is to find a

1-tree of minimum cost and can thus be formulated as follows:

$$\min \sum_{e \in E} c_e x_e,$$

$$x(\delta(\{0\})) = 2, \tag{1.31}$$

$$x(E(V)) = |V|, \tag{1.32}$$

$$x(E(S)) \leq |S| - 1 \ \forall S \subset V \setminus \{0\}, 3 \leq |S| \leq |V| - 1, \tag{1.33}$$

$$x_e \in \{0, 1\} \qquad \forall e \in E. \tag{1.34}$$

A minimum cost 1-tree can be obtained easily as the union of a minimum cost spanning tree of $V \setminus \{0\}$ plus two cheapest edges incident to vertex 0. For this example, we thus let

$$\mathcal{P}' = \text{conv}\left\{x \in \mathbb{R}^E \mid x \text{ satisfies } (1.31) - (1.34)\right\}$$

be the 1-*tree polytope*, while the degree and bound constraints comprise the polytope

$$\mathcal{Q}'' = \left\{x \in \mathbb{R}^E \mid x \text{ satisfies } (1.27) \text{ and } (1.29)\right\}.$$

The set of feasible solutions to TSP is then $\mathcal{F} = \mathcal{P}' \cap \mathcal{Q}'' \cap \mathbb{Z}^E$, the integer points in the intersection of these two polytopes. ■

1.3 Computational Software for Decomposition Methods

Sometime around the late 1980s the recognition of mixed integer programming models as an important paradigm for solving real business problems had encouraged a number of commercial software vendors towards a large investment in tackling bigger and harder MILPs. The computational strides made on methods for solving generic MILPs throughout the 1990s were incredible [13]. Despite this, there are still many classes of important MILPs that are extremely difficult for today's best solvers. Exploiting the special structure of certain models has long been an active field of research.

In the early 1990s, several research groups recognized the potential of abstracting the general

branch-and-cut framework in the form of a software framework with user *hooks* for adding problem-specific routines. This led to the development of several popular frameworks, for example, MINTO [68], MIPO [6], bc-opt [20], SIP [61], ABACUS [46], and SYMPHONY [79]. The majority of these frameworks are focused on providing an infrastructure for implementing branch-and-bound algorithms in which the user could provide their own specific methods for customizing both the branching and the bounding operations. In the 1990s, most of the work using these frameworks focused on problem-specific cutting planes that were incorporated into the framework to produce a branch-and-cut algorithm.

At the same time, column-generation methods were also gaining some popularity. Of the list above, the only frameworks that provided some facility for branch-and-price were MINTO, ABACUS, and SYMPHONY. In all cases, the end-goal was to automate the most common elements of the branch-and-cut (or price) algorithm, allowing the user to focus on the problem-specific hooks. In addition, some of the frameworks (SYMPHONY, for example) were designed in a generic manner to allow complete flexibility for the user to override just about every algorithmic function. This added to the wide array of problem types and methods that could be implemented within the frameworks. Much less common in these frameworks was support for integrated methods like branch-and-price-and-cut. Although there is some early mention of these ideas, there are very few implementations discussed in the literature that use any of these frameworks.

Around 1993, a research group headed by Ralphs and Ladányi at Cornell University, produced what was then known as COMPSys (Combinatorial Optimization Multi-Processing System). After several revisions to enable broader functionality, COMPSys became SYMPHONY (Single- or Multi-Process Optimization over Networks) [81]. SYMPHONY was originally written in C and provided a fully generic branch-and-cut framework where the nodes of the branch-and-bound tree could be processed in parallel in either distributed or shared memory architectures. SYMPHONY also provided limited functionality for branch-and-price. A version of SYMPHONY written in C++, called COIN/BCP was later produced at IBM as part of the COIN-OR (Computational Optimization INfrastructure for Operations Research) project [53]. In contrast to SYMPHONY, COIN/BCP is more focused on providing extended functionality for branch-and-price-and-cut.

Although column-generation methods are abundant in the literature for tackling difficult MILPs, the computational improvements are almost always based on problem-specific techniques. In many cases, theoretical generalizations of these ideas have long been known, but a treatment of the algorithmic implications has not. Consequently, the software frameworks for use in this area have remained inherently flexible, leaving it up to the users to implement the details of each method as it pertains to their specific application. With this flexibility comes a great deal of power but also a burden on the user to implement and reimplement the various algorithmic components in the context of their specific application.

In this research, we propose a theoretical framework that ties together various algorithmic approaches related to decomposition methods. From this foundation, we develop a new open-source C++ software framework, called DIP (**D**ecomposition for **I**nteger **P**rogramming). DIP is designed with the goal of providing a user with the ability to easily utilize various traditional and integrated decomposition methods while requiring only the provision of minimal problem-specific algorithmic components. With DIP, the majority of the algorithmic structure is provided as part of the framework, making it easy to compare various algorithms directly and determine which option is the best for a given problem setting. In addition, DIP is extensible—each algorithmic component *can* be overridden by the user, if they so wish, in order to develop sophisticated variants of these methods.

1.4 Contributions

In this section we summarize the contributions of this body of research.

- *A conceptual framework tying together numerous decomposition-based methods for generating approximations of the convex hull of feasible solutions.*

We draw connections among various decomposition-based methods used in the context of integer linear programming. These include outer approximation methods, like the cutting plane method, and inner approximation methods, like the Dantzig-Wolfe method and the Lagrangian method. We then extend these connections to encompass integrated methods, which generate tighter approximations by combining elements from more than one method simultaneously.

- *Development of a framework for implementing the integrated method called* decompose-and-cut*, an associated class of cutting planes called* decomposition cuts*, and the concept of* structured separation.

We introduce an extension of the well-known template paradigm, called *structured separation*, inspired by the fact that separation of structured solutions is frequently easier than separation of arbitrary real vectors. We also introduce a relatively new class of decomposition-based algorithms called *decompose-and-cut*. We present its use in the standard cutting plane method for structured separation, introduce a class of cutting planes called *decomposition cuts*, and provide supporting computational evidence of its effectiveness.

- *Descriptions of numerous implementation considerations for branch-and-price-and-cut, including an introduction to a relatively unknown idea of using nested polytopes for generating inner approximations.*

We introduce several techniques that can help overall performance when using integrated methods embedded in a branch-and-bound framework. We introduce an extension to the idea of *price-and-branch* and discuss the benefits of using nested polytopes when generating inner approximations. We provide computational comparisons of some of these techniques as they apply to the related methods.

- *DIP, an extensible open-source software framework for implementing decomposition-based methods with minimal user burden.*

We have created a new project as part of COIN-OR, called DIP (**D**ecomposition for **I**nteger **P**rogramming). This project includes a C++ software framework, which implements the majority of methods described in the thesis. With the framework, we provide numerous examples to show how a user would interact with the software to develop their own application based on these methods.

- *MILPBlock, a DIP application and generic black-box solver for* block diagonal *MILPs that fully automates the branch-and-price-and-cut algorithm with no additional user input.*

Along with the DIP project, we have created an application called MILPBlock. MILPBlock provides a black-box solver, based on these decomposition-based methods, for generic MILPs that have some block-angular structure.

- *Computational results using DIP on three real-world applications coming from the marketing, banking, and retail industries.*

Finally, we introduce a few applications developed using DIP and associated computational results coming from various industries.

1.5 Outline of the Thesis

In this section we outline the remaining chapters of the thesis. In Chapter 2, we present the overall theoretical framework for decomposition methods. In Section 2.1, we expand on the principle of decomposition and its application to integer linear programming in a traditional setting. This includes a review of three related algorithmic approaches: the cutting plane method, the Dantzig-Wolfe method, and the Lagrangian method. Each of these methods relies on finding an approximation of the convex hull of feasible solutions to the original problem. This is accomplished by intersecting one polytope, which has an explicit, compact representation, with another polytope, which has exponential size and is represented implicitly through the solution of some auxiliary subproblem. We frame these methods under one common thread in order to facilitate the presentation of the more advanced integrated algorithms. In Section 2.2, we extend the traditional framework to show how the cutting plane method can be integrated with either the Dantzig-Wolfe method or the Lagrangian method to yield improved bounds. In these integrated methods we now allow simultaneous generation of two polytopes both of exponential size. In Section 2.3, we discuss the solution of the separation subproblem and introduce an extension of the well-known template paradigm, called *structured separation*, inspired by the fact that separation of structured solutions is frequently easier than separation of arbitrary real vectors. We also introduce a relatively new class of decomposition-based algorithms called *decompose-and-cut*. We present its use in the standard cutting plane method for structured separation and introduce a class of cutting planes called *decomposition cuts*. These cuts

serve to break the template paradigm by using information from the implicitly defined polyhedron as in the case of the Dantzig-Wolfe method.

In Chapter 3, we focus attention on the implementation of branch-and-price-and-cut methods based on Dantzig-Wolfe decomposition. We describe a number of algorithmic details discovered during the development of DIP. Later, in Chapter 5, we present some applications developed in DIP and provide some computational results on the effectiveness of some of these ideas.

In Chapter 4, we describe DIP, a new open-source software framework, which follows the conceptual framework described in Chapter 2. We provide numerous examples to help solidify the understanding of how a user would interface with the framework.

In Chapter 5, we introduce a few applications developed using DIP and associated computational results referring back to some of the implementation details discussed in Chapter 3. In Section 5.1, we present the Multi-Choice Multi-Dimensional Knapsack Problem, which is an important subproblem used in the algorithms present in SAS Marketing Optimization, which attempts to improve the ROI for marketing campaign offers. In Section 5.2, we introduce an application from the banking industry for ATM cash management, which we worked on for the Center of Excellence in Operations Research at SAS Institute. We model the problem as a mixed integer nonlinear program and create an application in DIP, called ATM, to solve an approximating MILP using the aforementioned integrated decomposition methods. We discuss the ease of development using DIP as well as computational results, which show the effectiveness of the algorithmic approach. Then, in Section 5.3, we present another application developed in DIP, called MILPBlock, which provides a black-box framework for using these integrated methods on generic MILPs that have some block-angular structure. The ability to develop a software framework that can handle these methods in an application-independent manner relies on the conceptual framework proposed in the first few chapters. DIP is the first of its kind in this respect and should greatly break down the barriers of entry into developing solvers based on these methods. We present some computational results using MILPBlock on a model presented to us from SAS Retail Optimization. Finally, in Chapter 6, we conclude with a discussion of proposed future research.

Chapter 2

Decomposition Methods

Decomposition methods, in our context, are defined as methods capitalizing on knowledge of relaxations, for generating bounds by iterative construction of polyhedral approximations of the convex hull of feasible solutions to some MILP. In this chapter, we present two major categories of methods. The first category, called *traditional methods*, considers the intersection of a polyhedron having a compact description with one that is generated implicitly by solving an auxiliary problem. This is done because the second polyhedron has a description that is of exponential size and therefore cannot be efficiently defined explicitly. Traditional methods are further broken down into *outer methods*, like the cutting-plane method, and *inner methods*, like the Dantzig-Wolfe method and the Lagrangian method. The second category, called *integrated methods*, allows for both polyhedra to have exponential size. This lends itself to algorithms that allow the integration of both inner and outer methods simultaneously.

2.1 Traditional Decomposition Methods

In the following section, we review three classical approaches that take advantage of implicit generation of a polyhedral approximation. By viewing all of these methods within the same conceptual framework, we are able to draw several connections among the methods, helping to ease the transition into integrated methods in the following section.

2.1.1 Cutting-Plane Method

Using the cutting-plane method, the bound $z_{\text{D}} = \min_{x \in \mathcal{P}' \cap \mathcal{Q}''} \left\{c^\top x\right\}$ can be obtained dynamically by generating the *relevant* portions of an outer description of $\mathcal{P}'$. Let $[D, d]$ denote the set of facet-defining inequalities of $\mathcal{P}'$, so that

$$\mathcal{P}' = \{x \in \mathbb{R}^n \mid Dx \geq d\}. \tag{2.1}$$

Then the cutting-plane formulation for the problem of calculating z_{D} can be written as

$$z_{\text{CP}} = \min_{x \in \mathcal{Q}''} \left\{c^\top x \mid Dx \geq d\right\}. \tag{2.2}$$

This is a linear program, but since the set $[D, d]$ of valid inequalities is potentially of exponential size, we dynamically generate them by solving a separation problem. An outline of the method is presented in Figure 2.1.

In Step 2, the master problem is a linear program whose feasible region is the current outer approximation $\mathcal{P}_O^t$, defined by a set of initial valid inequalities plus those generated dynamically in Step 3. Solving the master problem in iteration t, we generate the relaxed (primal) solution x_{CP}^t and a valid lower bound. In the figure, the initial set of inequalities is taken to be those of $\mathcal{Q}''$, since it is assumed that the facet-defining inequalities for $\mathcal{P}'$, which dominate those of $\mathcal{Q}'$, can be generated dynamically. In practice, however, this initial set may be chosen to include those of $\mathcal{Q}'$ or some other polyhedron, on an empirical basis.

In Step 3, we solve the subproblem, which is to try to generate a set of *improving* valid inequalities, i.e., valid inequalities that improve the bound when added to the current approximation. This step is usually accomplished by applying one of the many known techniques for separating x_{CP}^t from $\mathcal{P}$. It is well known that violation of x_{CP}^t is a necessary condition for an inequality to be improving, and hence, we generally use this condition to judge the potential effectiveness of generated valid inequalities. However, this condition is not sufficient and unless the inequality separates the entire optimal face of $\mathcal{P}_O^t$, it will not actually be improving. Because we want to refer to these results later in the paper, we state them formally as theorem and corollary without proof. See [85] for a thorough

Cutting-Plane Method

Input: An instance OPT $(\mathcal{P}, c)$.
Output: A lower bound z_{CP} on the optimal solution value for the instance, and $\hat{x}_{\text{CP}} \in \mathbb{R}^n$ such that $z_{\text{CP}} = c^\top \hat{x}_{\text{CP}}$.

1. **Initialize:** Construct an initial outer approximation

$$\mathcal{P}_O^0 = \left\{ x \in \mathbb{R}^n \mid D^0 x \geq d^0 \right\} \supseteq \mathcal{P}, \tag{2.3}$$

   where $D^0 = A''$ and $d^0 = b''$, and set $t \leftarrow 0$.

2. **Master Problem:** Solve the linear program

$$z_{\text{CP}}^t = \min_{x \in \mathbb{R}^n} \left\{ c^\top x \mid D^t x \geq d^t \right\} \tag{2.4}$$

   to obtain the optimal value $z_{\text{CP}}^t = \min_{x \in \mathcal{P}_O^t} \left\{ c^\top x \right\} \leq z_{\text{IP}}$ and optimal primal solution x_{CP}^t.

3. **Subproblem:** Call the subroutine SEP $\left(\mathcal{P}, x_{\text{CP}}^t\right)$ to generate a set $[\tilde{D}, \tilde{d}]$ of potentially *improving* valid inequalities for $\mathcal{P}$, violated by x_{CP}^t.

4. **Update:** If violated inequalities were found in Step 3, set $[D^{t+1}, d^{t+1}] \leftarrow \left[\begin{smallmatrix} D^t & d^t \\ \tilde{D} & \tilde{d} \end{smallmatrix} \right]$ to form a new outer approximation

$$\mathcal{P}_O^{t+1} = \left\{ x \in \mathbb{R}^n \mid D^{t+1} x \leq d^{t+1} \right\} \supseteq \mathcal{P}, \tag{2.5}$$

   and set $t \leftarrow t + 1$. Go to Step 2.

5. If no violated inequalities were found, output $z_{\text{CP}} = z_{\text{CP}}^t \leq z_{\text{IP}}$ and $\hat{x}_{\text{CP}} = x_{\text{CP}}^t$.

Figure 2.1: Outline of the cutting-plane method

treatment of the theory of linear programming that leads to this result.

Theorem 2.1 ([77]) *Let F be the face of optimal solutions to an LP over a nonempty, bounded polyhedron X with objective function vector c. Then (a, β) is an improving inequality for X with respect to c, i.e.,*

$$\min \left\{ c^\top x \;\middle|\; x \in X, a^\top x \geq \beta \right\} > \min \left\{ c^\top x \mid x \in X \right\}, \tag{2.6}$$

if and only if $a^\top y < \beta$ for all $y \in F$.

Corollary 2.2 ([77]) *If (a, β) is an improving inequality for X with respect to c, then $a^\top \hat{x} < \beta$, where $\hat{x}$ is any optimal solution to the linear program over X with objective function vector c.*

Even in the case when the optimal face cannot be separated in its entirety, the augmented cutting-plane LP must have a different optimal solution, which in turn may be used to generate more potential improving inequalities. Since the condition of Theorem 2.1 is difficult to verify, one typically terminates the bounding procedure when increases resulting from additional inequalities become "too small."

If violated inequalities are found in Step 3, then the approximation is improved and the algorithm continues. By assumption, OPT $(\mathcal{P}, c)$ cannot be solved effectively, which means in turn that SEP $(\mathcal{P}, x)$ is also, in general, assumed to be difficult. Therefore, it is typical to look at the separation problem over some larger polyhedron containing $\mathcal{P}$, such as $\mathcal{P}'$.

To better understand this, we must first introduce the *template paradigm* as it applies to the cutting-plane method [2]. A set $F \subset \mathcal{P}$ is called a *face* if there exists a valid inequality (a, β) for $\mathcal{P}$ such that $F = \left\{x \in \mathcal{P} \;\middle|\; a^\top x = \beta\right\}$. A face of $\mathcal{P}$ is a *facet* of $\mathcal{P}$ if $dim\,(F) = dim\,(\mathcal{P}) - 1$. Clearly, when looking for tight approximation of $\mathcal{P}$, we ultimately want to generate those facets of $\mathcal{P}$ in the direction of the cost vector.

Instead of considering all valid inequalities at once, the template paradigm considers various (finite) subsets of valid inequalities, called *classes*, whose coefficients conform to the structure of a given *template*. The separation problem for a class of inequalities is then that of determining whether a given real vector lies in the polyhedron described by all inequalities in the class, and if not, determining an inequality from the class that is violated by the vector. In many cases, it is possible to solve the separation problem for a given class of inequalities valid for the polyhedron $\mathcal{P}$ effectively, though the general separation problem for $\mathcal{P}$ is difficult. Consider some finite class C of valid inequalities. The set of points satisfying all members of C is a polyhedron $\mathcal{C}$, called the *closure* with respect to C. Let us denote the separation problem for some class C of inequalities for a given vector x over the polyhedron $\mathcal{C}$ as SEP $(\mathcal{C}, x)$.

It is often the case that, for some class C of facet-defining inequalities, SEP $(\mathcal{C}, x)$ is also difficult to solve for arbitrary x. However, if we restrict our attention to points x that have some structure, we

can often solve the separation problem for class C effectively. This is a major point of emphasis in the discussion of integrated decomposition methods. In the classical implementation of the cutting-plane method, the augmentation of the constraint set is done by finding a separating hyperplane for some solution to a linear program. In general, this solution is not guaranteed to have any structure. One of the advantages of using decomposition methods is that we partition the problem such that the solutions often have a nice combinatorial structure, which we can exploit in the separation problem. We explore this idea further in Section 2.3.1.

If we start with the continuous approximation $\mathcal{P}_O^0 = \mathcal{Q}''$ and generate only facet-defining inequalities of $\mathcal{P}'$ in Step 3, then the procedure described here terminates in a finite number of steps with the bound $z_{\text{CP}} = z_{\text{D}}$ (see [70]). Since $\mathcal{P}_O^t \supseteq \mathcal{P}' \cap \mathcal{Q}'' \supseteq \mathcal{P}$, each step yields an approximation for $\mathcal{P}$, along with a valid bound. In Step 3, we are permitted to generate any valid inequality for $\mathcal{P}$, however, not just those that are facet-defining for $\mathcal{P}'$. In theory, this means that the cutting-plane method can be used to compute the bound z_{IP} exactly. However, in practice, this is usually not possible.

To illustrate the cutting-plane method, we show how it could be applied to generate the bound z_{D} for the ILPs in Examples 1 (SILP) and 3 (TSP). Since we are discussing the computation of the bound z_{D}, we only generate facet-defining inequalities for $\mathcal{P}'$ in these examples. We discuss more general scenarios later in the paper.

Example 1: SILP (Continued) We define the initial outer approximation to be $\mathcal{P}_O^0 = \mathcal{Q}' \cap \mathcal{Q}'' = \left\{x \in \mathbb{R}^2 \mid x \text{ satisfies } (1.6) - (1.20)\right\}$, the continuous approximation.

Iteration 0: Solving the master problem over $\mathcal{P}_O^0$, we find an optimal primal solution $x_{\text{CP}}^0 = (2.25, 2.75)$ with bound $z_{\text{CP}}^0 = 2.25$, as shown in Figure 2.2(a). We then call the subroutine SEP $\left(\mathcal{P}', x_{\text{CP}}^0\right)$, generating facet-defining inequalities of $\mathcal{P}'$ that are violated by x_{CP}^0. One such facet-defining inequality, $3x_1 - x_2 \geq 5$, is pictured in Figure 2.2(a). We add this inequality to form a new outer approximation $\mathcal{P}_O^1$.

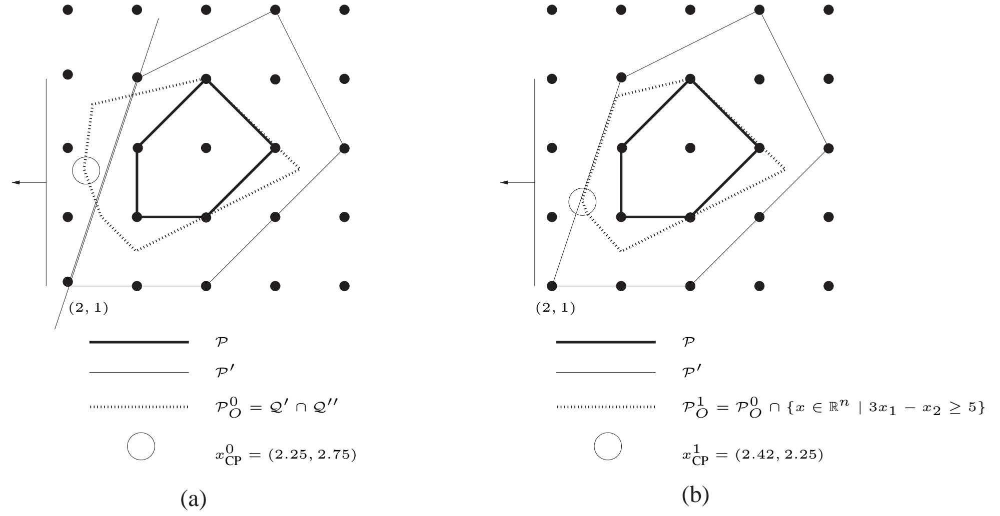


Figure 2.2: Cutting-plane method (Example 1: SILP)

Iteration 1: We again solve the master problem, this time over $\mathcal{P}^1_O$, to find an optimal primal solution $x^1_{\text{CP}} = (2.42, 2.25)$ and bound $z^1_{\text{CP}} = 2.42$, as shown in Figure 2.2(b). We then call the subroutine SEP $\left(\mathcal{P}', x^1_{\text{CP}}\right)$. However, as illustrated in Figure 2.2(b), there are no more facet-defining inequalities violated by x^1_{CP}. In fact, further improvement in the bound would necessitate the addition of valid inequalities violated by points in $\mathcal{P}'$. Since we are only generating facets of $\mathcal{P}'$ in this example, the method terminates with bound $z_{\text{CP}} = 2.42 = z_{\text{D}}$. ■

We now consider the use of the cutting-plane method for generating the bound z_{D} for the TSP example. Once again, we only generate facet-defining inequalities for $\mathcal{P}'$, the 1-tree polytope.

Example 3a: TSP (Continued) We define the initial outer approximation to be comprised of the degree constraints and the bound constraints, so that

$$\mathcal{P}^0_O = \mathcal{Q}'' = \left\{x \in \mathbb{R}^E \mid x \text{ satisfies (1.27) and (1.29)}\right\}.$$

To calculate z_{CP}, we must dynamically generate violated facet-defining inequalities of the 1-tree polytope $\mathcal{P}'$ defined earlier. Specifically, we want to find violated subtour elimination constraints (SECs). Given a vector $\hat{x} \in \mathbb{R}^E$ satisfying (1.27) and (1.29), the problem of finding an inequality of

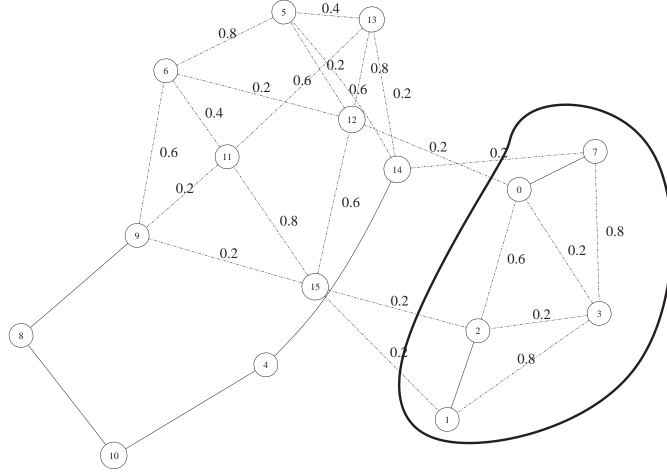


Figure 2.3: Finding violated inequalities in the cutting-plane method (Example 3a: TSP)

the form (1.33) violated by $\hat{x}$ is equivalent to the well-known minimum cut problem, which can be nominally solved in $O\left(|E||V| + |V|^2 \log |V|\right)$ time [67]. We can use this approach to implement Step 3 of the cutting-plane method and hence compute the bound z_{CP} effectively. As an example, consider the vector $\hat{x}$ pictured graphically in Figure 2.3, obtained in Step 2 of the cutting-plane method. In the figure, only edges e for which $\hat{x}_e > 0$ are shown. Each edge e is labeled with the value $\hat{x}_e$, except for edges e with $\hat{x}_e = 1$. The circled set $S = \{0, 1, 2, 3, 7\}$ of vertices defines an SEC violated by $\hat{x}$, since $\hat{x}\left(E\left(S\right)\right) = 4.6 > 4.0 = |S| - 1$. ■

2.1.2 Dantzig-Wolfe Method

In the Dantzig-Wolfe method, the bound z_{D} can be obtained by dynamically generating the *relevant* portions of an inner description of $\mathcal{P}'$ and intersecting it with $\mathcal{Q}''$. Consider Minkowski's Representation Theorem, which states that every bounded polyhedron is finitely generated by its extreme points and extreme rays [70]. Since we assume the feasible region for the problem of interest is bounded, we can focus solely on extreme points. Let $\mathcal{E} \subseteq \mathcal{F}'$ be the set of extreme points of $\mathcal{P}'$, so that

$$\mathcal{P}' = \left\{ x \in \mathbb{R}^n \;\middle|\; x = \sum_{s \in \mathcal{E}} s\lambda_s, \sum_{s \in \mathcal{E}} \lambda_s = 1, \lambda_s \geq 0 \;\forall s \in \mathcal{E} \right\}. \tag{2.7}$$

That is, every point $x \in \mathcal{P}'$ can be written as a convex combination of extreme points of $\mathcal{P}'$. In a slight abuse of notation, we let $s \in \mathcal{E}$ define an extreme point, a vector in $\mathbb{R}^n$, but also to serve as an index for the set of weights λ. Then the Dantzig-Wolfe formulation for computing the bound z_{D} is

$$\begin{aligned} z_{\mathrm{DW}} &= \min_{x \in \mathbb{R}^n} \left\{ c^\top x \mid A''x \geq b'', x \in \mathcal{P}' \right\}, && (2.8) \\ &= \min_{x \in \mathbb{R}^n} \left\{ c^\top x \;\middle|\; A''x \geq b'', x = \sum_{s \in \mathcal{E}} s\lambda_s, \sum_{s \in \mathcal{E}} \lambda_s = 1, \lambda_s \geq 0 \; \forall s \in \mathcal{E} \right\}. && (2.9) \end{aligned}$$

By substituting out the original variables, this formulation can be rewritten in the more familiar form

$$z_{\mathrm{DW}} = \min_{\lambda \in \mathbb{R}_+^{\mathcal{E}}} \left\{ c^\top \left(\sum_{s \in \mathcal{E}} s\lambda_s \right) \;\middle|\; A'' \left(\sum_{s \in \mathcal{E}} s\lambda_s \right) \geq b'', \sum_{s \in \mathcal{E}} \lambda_s = 1 \right\}. \tag{2.10}$$

This is a linear program, but since the set $\mathcal{E}$ of extreme points is potentially of exponential size, we dynamically generate those that are relevant by solving an optimization problem over $\mathcal{P}'$. The reformulation in (2.10) is commonly referred to as the *extended formulation* because of the large number of columns. In contrast, the *original* formulation (2.2) used in the cutting plane method is referred to as the *compact* formulation. An outline of the Dantzig-Wolfe method is presented in Figure 2.4.

In Step 1, we need to initialize the set of extreme points to obtain the first approximation. Note that there is no obvious candidate for the approximation $\mathcal{P}_I^0$. This approximation can be obtained by generating an initial subset $\mathcal{E}^0 \subseteq \mathcal{E}$ either randomly or by some other appropriate method. This concept is further explored in Section 3.4.

In Step 2, we solve the master problem, which is a restricted linear program obtained by substituting $\mathcal{E}^t$ for $\mathcal{E}$ in (2.10). Solving this results in a primal solution λ_{DW}^t, and a dual solution consisting of the dual multipliers u_{DW}^t on the constraints corresponding to $[A'', b'']$ and the multiplier α_{DW}^t on the convexity constraint. The dual solution is needed to generate the improving columns in Step 3. In each iteration, we are generating an inner approximation, $\mathcal{P}_I^t \subseteq \mathcal{P}'$, the convex hull of $\mathcal{E}^t$. Thus $\mathcal{P}_I^t \cap \mathcal{Q}''$ may or may not contain $\mathcal{P}$, and the value $\bar{z}_{\mathrm{DW}}^t$ returned from the master problem in Step 2 provides an *upper* bound on z_{DW}. Nonetheless, it is easy to show (see Section 2.1.3) that an optimal solution to the subproblem solved in Step 3 yields a valid lower bound. In particular, if $\tilde{s}$ is

Dantzig-Wolfe Method

Input: An instance OPT $(\mathcal{P}, c)$.
Output: A lower bound z_{DW} on the optimal solution value for the instance, a primal solution $\hat{\lambda}_{\text{DW}} \in \mathbb{R}^{\mathcal{E}}$, and a dual solution $(\hat{u}_{\text{DW}}, \hat{\alpha}_{\text{DW}}) \in \mathbb{R}^{m''+1}$.

1. **Initialize:** Construct an initial inner approximation

$$\mathcal{P}_I^0 = \left\{ \sum_{s \in \mathcal{E}^0} s\lambda_s \;\middle|\; \sum_{s \in \mathcal{E}^0} \lambda_s = 1, \lambda_s \geq 0 \; \forall s \in \mathcal{E}^0, \lambda_s = 0 \; \forall s \in \mathcal{E} \setminus \mathcal{E}^0 \right\} \subseteq \mathcal{P}' \quad (2.11)$$

   from an initial set $\mathcal{E}^0$ of extreme points of $\mathcal{P}'$ and set $t \leftarrow 0$.

2. **Master Problem:** Solve the Dantzig-Wolfe reformulation

$$\bar{z}_{\text{DW}}^t = \min_{\lambda \in \mathbb{R}_+^{\mathcal{E}}} \left\{ c^\top \left(\sum_{s \in \mathcal{E}} s\lambda_s \right) \;\middle|\; A'' \left(\sum_{s \in \mathcal{E}} s\lambda_s \right) \geq b'', \sum_{s \in \mathcal{E}} \lambda_s = 1, \lambda_s = 0 \; \forall s \in \mathcal{E} \setminus \mathcal{E}^t \right\} \quad (2.12)$$

   to obtain the optimal value $\bar{z}_{\text{DW}}^t = \min_{\mathcal{P}_I^t \cap \mathcal{Q}''} c^\top x \geq z_{\text{DW}}$, an optimal primal solution $\lambda_{\text{DW}}^t \in \mathbb{R}_+^{\mathcal{E}}$, and an optimal dual solution $\left(u_{\text{DW}}^t, \alpha_{\text{DW}}^t\right) \in \mathbb{R}^{m''+1}$.

3. **Subproblem:** Call the subroutine OPT $\left(\mathcal{P}', c^\top - \left(u_{\text{DW}}^t\right)^\top A'', \alpha_{\text{DW}}^t\right)$, generating a set $\tilde{\mathcal{E}}$ of *improving* members of $\mathcal{E}$ with negative reduced cost, where the reduced cost of $s \in \mathcal{E}$ is

$$rc\,(s) = \left(c^\top - \left(u_{\text{DW}}^t\right)^\top A''\right) s - \alpha_{\text{DW}}^t. \quad (2.13)$$

   If $\tilde{s} \in \tilde{\mathcal{E}}$ is a member of $\mathcal{E}$ with smallest reduced cost, then $\underline{z}_{\text{DW}}^t = rc\,(\tilde{s}) + \alpha_{\text{DW}}^t + \left(u_{\text{DW}}^t\right)^\top b'' \leq z_{\text{DW}}$ provides a valid lower bound.

4. **Update:** If $\tilde{\mathcal{E}} \neq \emptyset$, set $\mathcal{E}^{t+1} \leftarrow \mathcal{E}^t \cup \tilde{\mathcal{E}}$ to form the new inner approximation

$$\mathcal{P}_I^{t+1} = \left\{ \sum_{s \in \mathcal{E}^{t+1}} s\lambda_s \;\middle|\; \sum_{s \in \mathcal{E}^{t+1}} \lambda_s = 1, \lambda_s \geq 0 \; \forall s \in \mathcal{E}^{t+1}, \lambda_s = 0 \; \forall s \in \mathcal{E} \setminus \mathcal{E}^{t+1} \right\} \subseteq \mathcal{P}', \quad (2.14)$$

   and set $t \leftarrow t + 1$. Go to Step 2.

5. If $\tilde{\mathcal{E}} = \emptyset$, output the bound $z_{\text{DW}} = \bar{z}_{\text{DW}}^t = \underline{z}_{\text{DW}}^t$, $\hat{\lambda}_{\text{DW}} = \lambda_{\text{DW}}^t$, and $(\hat{u}_{\text{DW}}, \hat{\alpha}_{\text{DW}}) = \left(u_{\text{DW}}^t, \alpha_{\text{DW}}^t\right)$.

Figure 2.4: Outline of the Dantzig-Wolfe method

a member of $\mathcal{E}$ with the smallest reduced cost in Step 3, then

$$\underline{z}^t_{\text{DW}} = z^t_{\text{DW}} + rc\left(\tilde{s}\right) = c^\top \tilde{s} + \left(u^t_{\text{DW}}\right)^\top \left(b'' - A''\tilde{s}\right) \tag{2.15}$$

is a valid lower bound. It should be noted that any valid lower bound returned from OPT$(\mathcal{P}', c^\top - (u^t_{\text{DW}})^\top A'', \alpha^t_{\text{DW}})$ can serve to generate a valid lower bound for z_{DW}. That is, if $\underline{rc}\left(\tilde{s}\right) \leq rc\left(\tilde{s}\right)$, then $\underline{z}^t_{\text{DW}} = z^t_{\text{DW}} + \underline{rc}\left(\tilde{s}\right) \leq z_{\text{DW}}$, is also a valid lower bound.

In Step 3, we search for *improving* members of $\mathcal{E}$, where, as in the previous section, this means members that when added to $\mathcal{E}^t$ yield an improved bound. It is less clear here, however, which bound we would like to improve, $\bar{z}^t_{\text{DW}}$ or $\underline{z}^t_{\text{DW}}$. A necessary condition for improving $\bar{z}^t_{\text{DW}}$ is the generation of a column with negative reduced cost. In fact, if one considers (2.15), it is clear that this condition is also necessary for improvement of $\underline{z}^t_{\text{DW}}$. If any such members exist, we add them to $\mathcal{E}^t$ and iterate.

In practice, the algorithmic flow in Figure 2.4 is done in two *phases*. Given a set of initial columns, there is no guarantee that (2.12) will be feasible. To deal with this, we solve a slightly different problem in *Phase 1*. We add slack variables $y \in \mathbb{R}^{m''}_+$ to each constraint in the master problem, set the cost vector c to zero, and instead minimize the slacks. This gives the following restricted master problem:

$$\bar{z}^t_{\text{DW}} = \min_{\lambda \in \mathbb{R}^{\mathcal{E}}_+, y \in \mathbb{R}^{m''}_+} \left\{ 1^\top y \;\middle|\; A''\left(\sum_{s\in\mathcal{E}} s\lambda_s\right) + y \geq b'', \sum_{s\in\mathcal{E}} \lambda_s = 1, \lambda_s = 0 \;\forall s \in \mathcal{E} \setminus \mathcal{E}^t \right\}, \tag{2.16}$$

which replaces (2.12) while in the first phase. In this manner, the same algorithmic flow applies, with the standard reduced cost replaced by

$$rc\left(s\right) = -\left(u^t_{\text{DW}}\right)^\top A''s - \alpha^t_{\text{DW}}. \tag{2.17}$$

The influence in this phase is towards feasibility rather than optimality. Once a feasible solution has been found, i.e., $\hat{y} = 0$, we can remove the slack variables and move on to *Phase 2*, as described in Figure 2.4.

An area that deserves some deeper investigation is the relationship between the solution obtained by solving the reformulation (2.12) and the solution that would be obtained by solving an LP directly over $\mathcal{P}_I^t \cap \mathcal{Q}''$ with the objective function c. Consider the primal optimal solution λ_{DW}^t, which we refer to as an *optimal decomposition*. If we combine the members of $\mathcal{E}^t$ using λ_{DW}^t to obtain an *optimal fractional solution*

$$x_{\text{DW}}^t = \sum_{s \in \mathcal{E}^t} s \left(\lambda_{\text{DW}}^t\right)_s, \tag{2.18}$$

then we see that $\bar{z}_{\text{DW}}^t = c^\top x_{\text{DW}}^t$. In fact, $x_{\text{DW}}^t \in \mathcal{P}_I^t \cap \mathcal{Q}''$ is an optimal solution to the linear program solved directly over $\mathcal{P}_I^t \cap \mathcal{Q}''$ with objective function c.

The optimal fractional solution plays an important role in the integrated methods to be introduced later. To illustrate the Dantzig-Wolfe method and the role of the optimal fractional solution in the method, we show how to apply it to generate the bound z_{D} for the ILP of Example 1.

Example 1 : SILP (Continued) For the purposes of illustration, we begin with a randomly generated initial set $\mathcal{E}_0 = \{(4,1), (5,5)\}$ of points. Taking their convex hull, we form the initial inner approximation $\mathcal{P}_I^0 = \text{conv}\left(\mathcal{E}^0\right)$, as illustrated in Figure 2.5(a).

Iteration 0: Solving the master problem with inner polyhedron $\mathcal{P}_I^0$, we obtain an optimal primal solution $\left(\lambda_{\text{DW}}^0\right)_{(4,1)} = 0.75$, $\left(\lambda_{\text{DW}}^0\right)_{(5,5)} = 0.25$, $x_{\text{DW}}^0 = (4.25, 2)$, and bound $\bar{z}_{\text{DW}}^0 = 4.25$. Since constraint (1.14) is binding at x_{DW}^0, the only nonzero component of u_{DW}^0 is $\left(u_{\text{DW}}^0\right)_{(1.14)} = 0.28$, while the dual variable associated with the convexity constraint has value $\alpha_{\text{DW}}^0 = 4.17$. All other dual variables have value zero. Next, we search for an extreme point of $\mathcal{P}'$ with negative reduced cost, by solving the subproblem OPT $\left(\mathcal{P}', c^\top - \left(u_{\text{DW}}^t\right)^\top A'', \alpha_{\text{DW}}^0\right)$. From Figure 2.5(a), we see that $\tilde{s} = (2,1)$. This gives a valid lower bound $\underline{z}_{\text{DW}}^0 = 2.03$. We add the corresponding column to the restricted master and set $\mathcal{E}^1 = \mathcal{E}^0 \cup \{(2,1)\}$.

Iteration 1: The next iteration is depicted in Figure 2.5(b). First, we solve the master problem with inner polyhedron $\mathcal{P}_I^1 = \text{conv}\left(\mathcal{E}^1\right)$ to obtain $\left(\lambda_{\text{DW}}^1\right)_{(5,5)} = 0.21$, $\left(\lambda_{\text{DW}}^1\right)_{(2,1)} = 0.79$, $x_{\text{DW}}^1 = (2.64, 1.86)$, and bound and $\bar{z}_{\text{DW}}^1 = 2.64$. This also provides the dual solution $\left(u_{\text{DW}}^1\right)_{(1.16)} = 0.43$

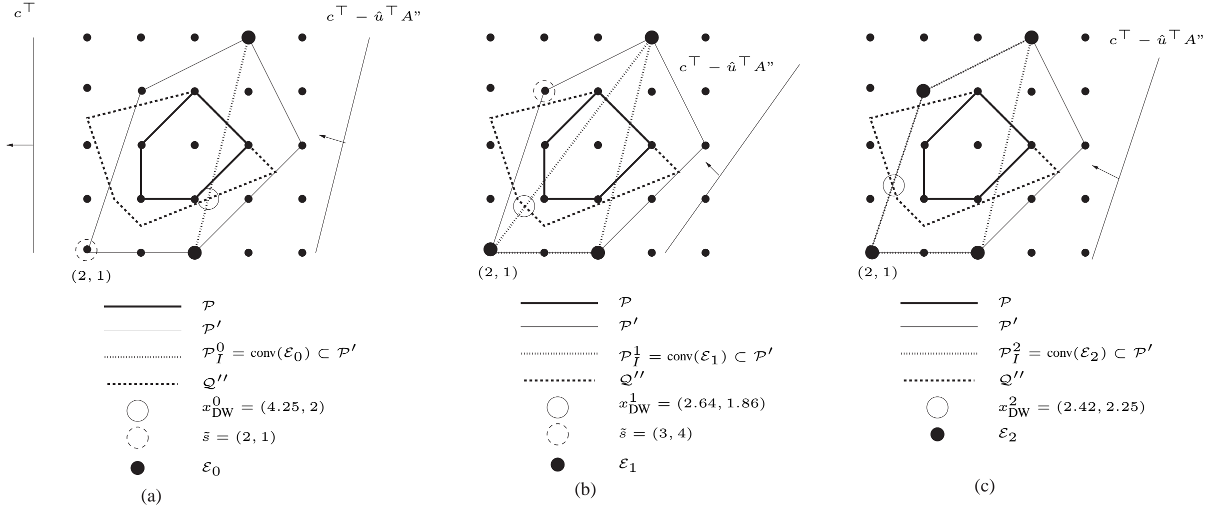


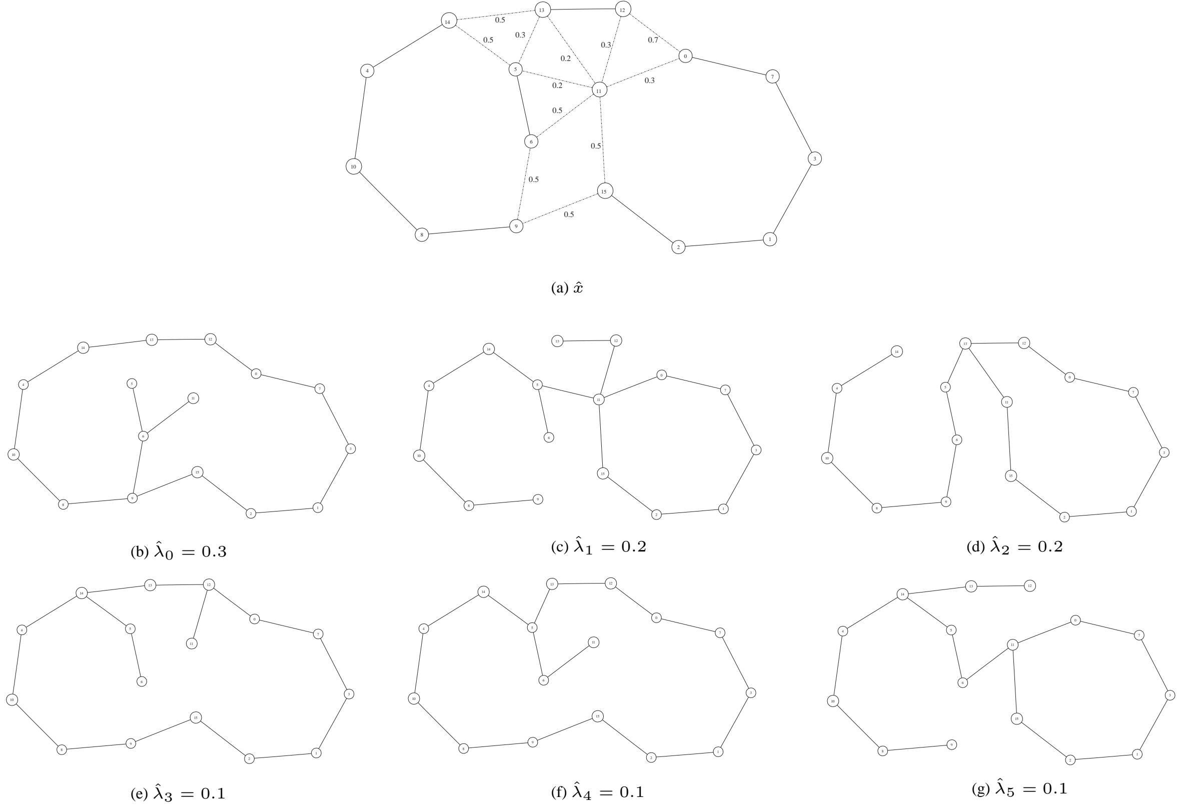
Figure 2.5: Dantzig-Wolfe method (Example 1: SILP)

and $\alpha^1_{\text{DW}} = 0.71$ (all other dual values are zero). Solving OPT $\left(\mathcal{P}', c^\top - u^1_{\text{DW}} A'', \alpha^1_{\text{DW}}\right)$, we obtain $\tilde{s} = (3, 4)$, and $\underline{z}^1_{\text{DW}} = 1.93$. We add the corresponding column to the restricted master and set $\mathcal{E}^2 = \mathcal{E}^1 \cup \{(3, 4)\}$.

Iteration 2: The final iteration is depicted in Figure 2.5(c). Solving the master problem once more with inner polyhedron $\mathcal{P}^2_I = \text{conv}\left(\mathcal{E}^2\right)$, we obtain $\left(\lambda^2_{\text{DW}}\right)_{(2,1)} = 0.58$ and $\left(\lambda^2_{\text{DW}}\right)_{(3,4)} = 0.42$, $x^2_{\text{DW}} = (2.42, 2.25)$, and bound $\bar{z}^2_{\text{DW}} = 2.42$. This also provides the dual solution $\left(u^2_{\text{DW}}\right)_{(1.18)} = 0.17$ and $\alpha^2_{\text{DW}} = 0.83$. Solving OPT $\left(\mathcal{P}', c^\top - u^2_{\text{DW}} A'', \alpha^2_{\text{DW}}\right)$, we conclude that $\tilde{\mathcal{E}} = \emptyset$. We therefore terminate with the bound $z_{\text{DW}} = 2.42 = z_{\text{D}}$. ■

As a further brief illustration, we return to the TSP example introduced earlier.

Example 3a: TSP (Continued) As we noted earlier, the Minimum 1-Tree Problem can be solved by computing a minimum cost spanning tree on vertices $V \setminus \{0\}$, and then adding two cheapest edges incident to vertex 0. This can be done in $O\left(|E| \log |V|\right)$ time using standard algorithms. In applying the Dantzig-Wolfe method to compute z_{D} using the decomposition described earlier, the subproblem to be solved in Step 3 is a Minimum 1-Tree Problem. Because we can solve this problem



(a) $\hat{x}$

(b) $\hat{\lambda}_0 = 0.3$ (c) $\hat{\lambda}_1 = 0.2$ (d) $\hat{\lambda}_2 = 0.2$

(e) $\hat{\lambda}_3 = 0.1$ (f) $\hat{\lambda}_4 = 0.1$ (g) $\hat{\lambda}_5 = 0.1$

Figure 2.6: Dantzig-Wolfe method (Example 3a: TSP)

effectively, we can apply the Dantzig-Wolfe method in this case. As an example of the result of solving the Dantzig-Wolfe master problem (2.12), Figure 2.6 depicts an optimal fractional solution (a) to a Dantzig-Wolfe master LP and the six extreme points 2.6(b-g) of the 1-tree polyhedron $\mathcal{P}'$, with nonzero weight comprising an optimal decomposition. ■

Now consider the set $\mathcal{S}(u, \alpha)$, defined as

$$\mathcal{S}(u, \alpha) = \left\{ s \in \mathcal{E} \;\middle|\; \left(c^\top - u^\top A''\right) s = \alpha \right\}, \tag{2.19}$$

where $u \in \mathbb{R}^{m''}$ and $\alpha \in \mathbb{R}$. The set $\mathcal{S}\left(u^t_{\mathrm{DW}}, \alpha^t_{\mathrm{DW}}\right)$ is the set of members of $\mathcal{E}$ with reduced cost zero at optimality for (2.12) in iteration t. It follows that $\operatorname{conv}\left(\mathcal{S}\left(u^t_{\mathrm{DW}}, \alpha^t_{\mathrm{DW}}\right)\right)$ is in fact the face of optimal solutions to the linear program solved over $\mathcal{P}^t_I$ with objective function $c^\top - u^\top A''$. This

line of reasoning culminates in the following theorem tying together the set $\mathcal{S}\left(u^t_{\text{DW}}, \alpha^t_{\text{DW}}\right)$ defined above, the vector x^t_{DW}, and the optimal face of solutions to the LP over the polyhedron $\mathcal{P}^t_I \cap \mathcal{Q}''$.

Theorem 2.3 ([77]) *conv* $\left(\mathcal{S}\left(u^t_{DW}, \alpha^t_{DW}\right)\right)$ *is a face of* $\mathcal{P}^t_I$ *and contains* x^t_{DW}.

Proof We first show that conv $\left(\mathcal{S}\left(u^t_{\text{DW}}, \alpha^t_{\text{DW}}\right)\right)$ is a face of $\mathcal{P}^t_I$. Observe that

$$\left(c^\top - \left(u^t_{\text{DW}}\right)^\top A'', \alpha^t_{\text{DW}}\right)$$

defines a valid inequality for $\mathcal{P}^t_I$ since α^t_{DW} is the optimal value for the problem of minimizing over $\mathcal{P}^t_I$ with objective function $c^\top - \left(u^t_{\text{DW}}\right)^\top A''$. Thus, the set

$$G = \left\{x \in \mathcal{P}^t_I \;\middle|\; \left(c^\top - \left(u^t_{\text{DW}}\right)^\top A''\right) x = \alpha^t_{\text{DW}}\right\}, \tag{2.20}$$

is a face of $\mathcal{P}^t_I$ that contains $\mathcal{S}\left(u^t_{\text{DW}}, \alpha^t_{\text{DW}}\right)$. We show that conv $\left(\mathcal{S}\left(u^t_{\text{DW}}, \alpha^t_{\text{DW}}\right)\right) = G$. Since G is convex and contains $\mathcal{S}\left(u^t_{\text{DW}}, \alpha^t_{\text{DW}}\right)$, it also contains conv $\left(\mathcal{S}\left(u^t_{\text{DW}}, \alpha^t_{\text{DW}}\right)\right)$, so we just need to show that conv $\left(\mathcal{S}\left(u^t_{\text{DW}}, \alpha^t_{\text{DW}}\right)\right)$ contains G. We do so by observing that the extreme points of G are elements of $\mathcal{S}\left(u^t_{\text{DW}}, \alpha^t_{\text{DW}}\right)$. By construction, all extreme points of $\mathcal{P}^t_I$ are members of $\mathcal{E}$, and the extreme points of G are also extreme points of $\mathcal{P}^t_I$. Therefore, the extreme points of G must be members of $\mathcal{E}$ and contained in $\mathcal{S}\left(u^t_{\text{DW}}, \alpha^t_{\text{DW}}\right)$. The claim follows, and conv $\left(\mathcal{S}\left(u^t_{\text{DW}}, \alpha^t_{\text{DW}}\right)\right)$ is a face of $\mathcal{P}^t_I$. The fact that $x^t_{\text{DW}} \in$ conv $\left(\mathcal{S}\left(u^t_{\text{DW}}, \alpha^t_{\text{DW}}\right)\right)$ follows from the fact that x^t_{DW} is a convex combination of members of $\mathcal{S}\left(u^t_{\text{DW}}, \alpha^t_{\text{DW}}\right)$. ■

An important consequence of Theorem 2.3 is that the face of optimal solutions to the LP over the polyhedron $\mathcal{P}^t_I \cap \mathcal{Q}''$ is actually contained in conv $\left(\mathcal{S}\left(u^t_{\text{DW}}, \alpha^t_{\text{DW}}\right)\right) \cap \mathcal{Q}''$, as stated in the following corollary.

Corollary 2.4 ([77]) *If* F *is the face of optimal solutions to the linear program solved directly over* $\mathcal{P}^t_I \cap \mathcal{Q}''$ *with objective function vector* c*, then* $F \subseteq$ *conv* $\left(\mathcal{S}\left(u^t_{DW}, \alpha^t_{DW}\right)\right) \cap \mathcal{Q}''$.

Proof Let $\hat{x} \in F$ be given. Then we have that $\hat{x} \in \mathcal{P}^t_I \cap \mathcal{Q}''$ by definition, and

$$c^\top \hat{x} = \alpha^t_{\text{DW}} + \left(u^t_{\text{DW}}\right)^\top b'' = \alpha^t_{\text{DW}} + \left(u^t_{\text{DW}}\right)^\top A'' \hat{x}, \tag{2.21}$$

where the first equality in this chain is a consequence of strong duality and the last is a consequence of complementary slackness. Hence, it follows that $\left(c^\top - \left(u^t_{\text{DW}}\right)^\top A''\right)\hat{x} = \alpha^t_{\text{DW}}$, and the result is proven. ■

Hence, each iteration of the method produces not only the primal solution $x^t_{\text{DW}} \in \mathcal{P}^t_I \cap \mathcal{Q}''$ but also a dual solution $\left(u^t_{\text{DW}}, \alpha^t_{\text{DW}}\right)$ that defines a face conv $\left(\mathcal{S}\left(u^t_{\text{DW}}, \alpha^t_{\text{DW}}\right)\right)$ of $\mathcal{P}^t_I$ that contains the entire optimal face of solutions to the LP solved directly over $\mathcal{P}^t_I \cap \mathcal{Q}''$ with the original objective function vector c.

When no column with negative reduced cost exists, the two bounds must be equal to z_{D} and we stop, outputting both the primal solution $\hat{\lambda}_{\text{DW}}$ and the dual solution $(\hat{u}_{\text{DW}}, \hat{\alpha}_{\text{DW}})$. It follows from the results proven above that in the final iteration, any column of (2.12) with reduced cost zero must in fact have a cost of $\hat{\alpha}_{\text{DW}} = z_{\text{D}} - \hat{u}^\top_{\text{DW}} b''$ when evaluated with respect to the modified objective function $c^\top - \hat{u}^\top_{\text{DW}} A''$. In the final iteration, we can therefore strengthen the statement of Theorem 2.3, as follows.

Theorem 2.5 ([77]) *conv $(\mathcal{S}(\hat{u}_{DW}, \hat{\alpha}_{DW}))$ is a face of $\mathcal{P}'$ and contains $\hat{x}_{DW}$.*

The proof follows along the same lines as Theorem 2.3. As before, we can also state the following important corollary.

Corollary 2.6 ([77]) *If F is the face of optimal solutions to the linear program solved directly over $\mathcal{P}' \cap \mathcal{Q}''$ with objective function vector c, then $F \subseteq$ conv $(\mathcal{S}(\hat{u}_{DW}, \hat{\alpha}_{DW})) \cap \mathcal{Q}''$.*

Thus, conv $(\mathcal{S}(\hat{u}_{\text{DW}}, \hat{\alpha}_{\text{DW}}))$ is actually a face of $\mathcal{P}'$ that contains $\hat{x}_{\text{DW}}$ and the entire face of optimal solutions to the LP solved over $\mathcal{P}' \cap \mathcal{Q}''$ with objective function c. This fact provides strong intuition regarding the connection between the Dantzig-Wolfe method and the cutting-plane method and allows us to regard Dantzig-Wolfe decomposition as either a procedure for producing the bound $z_{\text{D}} = c^\top \hat{x}_{\text{DW}}$ from primal solution information or the bound $z_{\text{D}} = c^\top \hat{s} + \hat{u}^\top_{\text{DW}}(b'' - A''\hat{s})$, where $\hat{s}$ is any member of $\mathcal{S}(\hat{u}_{\text{DW}}, \hat{\alpha}_{\text{DW}})$, from dual solution information. This fact is important in the next section, as well as later when we discuss integrated methods.

The exact relationship among $\mathcal{S}(\hat{u}_{\text{DW}}, \hat{\alpha}_{\text{DW}})$, the polyhedron $\mathcal{P}' \cap \mathcal{Q}''$, and the face F of optimal solutions to an LP solved over $\mathcal{P}' \cap \mathcal{Q}''$ can vary for different polyhedra and even for different

objective functions. Figure 2.7 shows the polyhedra of Example 1 with three different objective functions indicated. The convex hull of $\mathcal{S}(\hat{u}_{\text{DW}}, \hat{\alpha}_{\text{DW}})$ is typically a proper face of $\mathcal{P}'$, but it is possible for $\hat{x}_{\text{DW}}$ to be an inner point of $\mathcal{P}'$, in which case we have the following result.

Theorem 2.7 ([77]) *If $\hat{x}_{DW}$ is an inner point of $\mathcal{P}'$, then conv $(\mathcal{S}(\hat{u}_{DW}, \hat{\alpha}_{DW})) = \mathcal{P}'$.*

Proof We prove the contrapositive. Suppose conv $(\mathcal{S}(\hat{u}_{\text{DW}}, \hat{\alpha}_{\text{DW}}))$ is a proper face of $\mathcal{P}'$. Then there exists a facet-defining valid inequality $(a, \beta) \in \mathbb{R}^{n+1}$ such that conv $(\mathcal{S}(\hat{u}_{\text{DW}}, \hat{\alpha}_{\text{DW}})) \subseteq \left\{x \in \mathbb{R}^n \mid a^\top x = \beta\right\}$. By Theorem 2.5, $\hat{x}_{\text{DW}} \in$ conv $(\mathcal{S}(\hat{u}_{\text{DW}}, \hat{\alpha}_{\text{DW}}))$ and $\hat{x}_{\text{DW}}$ therefore cannot satisfy the definition of an inner point.

In this case, illustrated graphically in Figure 2.7(a) with the polyhedra from Example 1, $z_{\text{DW}} = z_{LP}$ and Dantzig-Wolfe decomposition does not improve the bound. All columns of the Dantzig-Wolfe LP have reduced cost zero, and any member of $\mathcal{E}$ can be given positive weight in an optimal decomposition. A necessary condition for an optimal fractional solution to be an inner point of $\mathcal{P}'$ is that the dual value of the convexity constraint in an optimal solution to the Dantzig-Wolfe LP be zero. This condition indicates that the chosen relaxation may be too weak.

A second case of potential interest is when $F =$ conv $(\mathcal{S}(\hat{u}_{\text{DW}}, \hat{\alpha}_{\text{DW}})) \cap \mathcal{Q}''$, illustrated graphically in Figure 2.7(b). In this case, all constraints of the Dantzig-Wolfe LP *other than* the convexity constraint must have dual value zero, since removing them does not change the optimal solution value. This condition can be detected by examining the objective function values of the members of $\mathcal{E}$ with positive weight in the optimal decomposition. If they are all identical, any such member that is contained in $\mathcal{Q}''$ (if one exists) must be optimal for the original ILP, since it is feasible and has objective function value equal to z_{IP}. The more typical case, in which F is a proper subset of conv $(\mathcal{S}(\hat{u}_{\text{DW}}, \hat{\alpha}_{\text{DW}})) \cap \mathcal{Q}''$, is shown in Figure 2.7(c).

2.1.3 Lagrangian Method

The Lagrangian method [31, 11] is a general approach for computing z_{D} that is closely related to the Dantzig-Wolfe method but is focused primarily on producing dual solution information. The Lagrangian method can be viewed as a method for producing a particular face of $\mathcal{P}'$, as in the

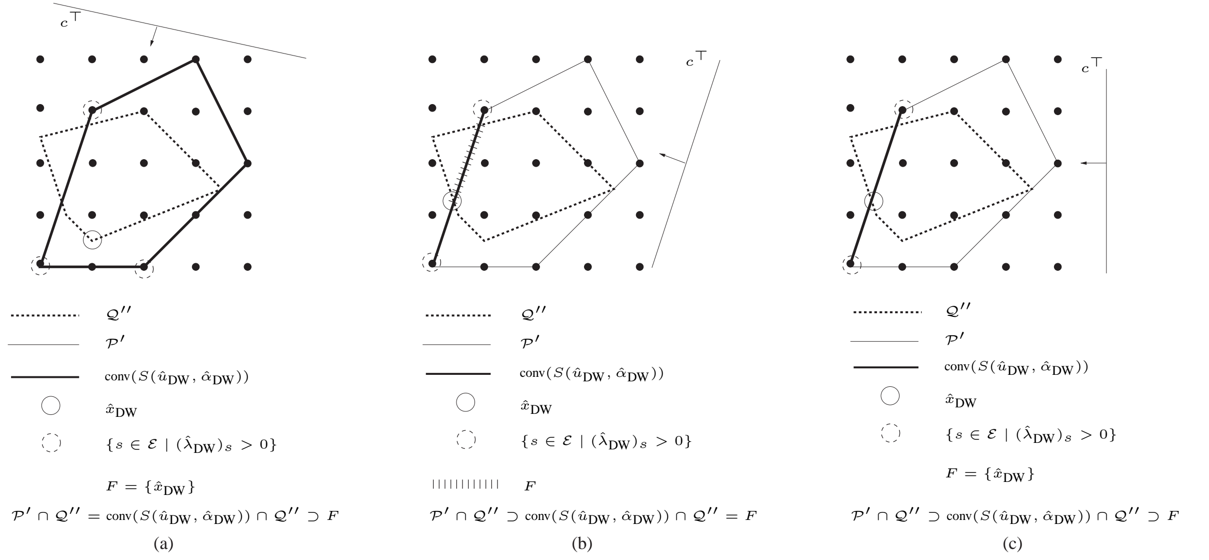


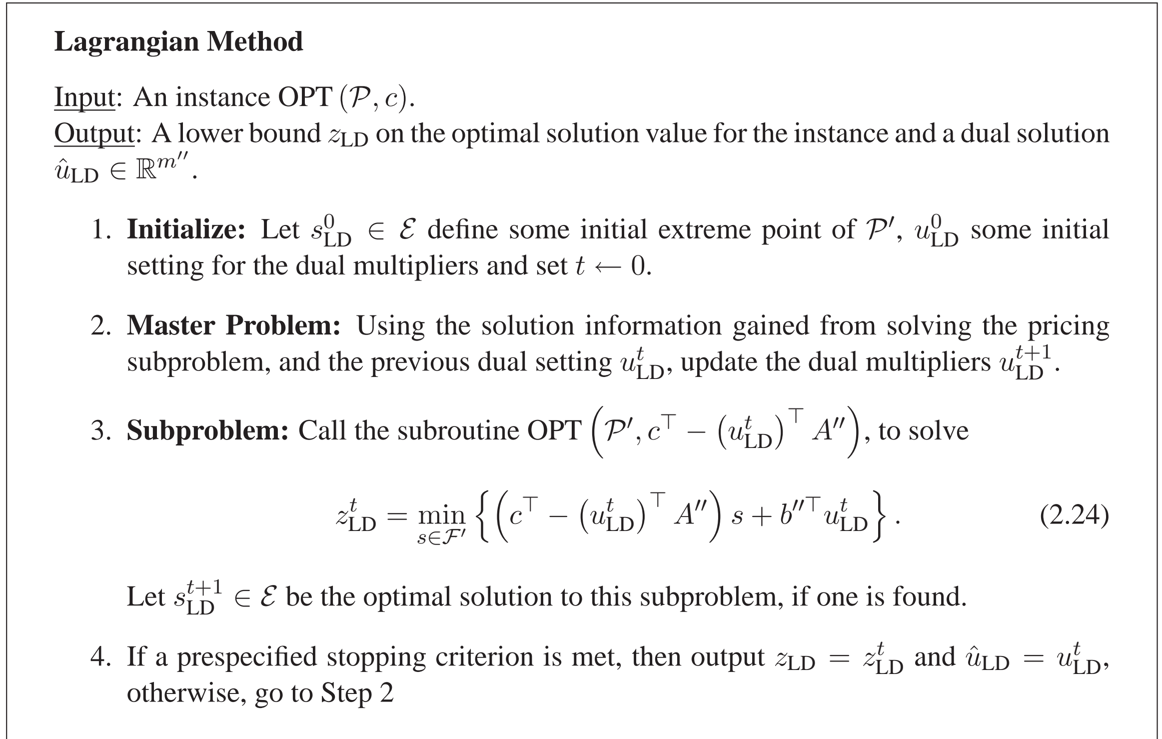
Figure 2.7: The relationship of $\mathcal{P}' \cap \mathcal{Q}''$, conv $(\mathcal{S}\ (\hat{u}_{\text{DW}}, \hat{\alpha}_{\text{DW}})) \cap \mathcal{Q}''$, and the face F.

Dantzig-Wolfe method, but no explicit approximation of $\mathcal{P}'$ is maintained. Although there are implementations of the Lagrangian method that *do* produce approximate primal solution information similar to the solution information that the Dantzig-Wolfe method produces (see Section 2.1.2), our viewpoint is that the main difference between the Dantzig-Wolfe method and the Lagrangian method is the type of solution information they produce. This distinction is important when we discuss integrated methods in Section 2.2. When exact primal solution information is not required, faster algorithms for determining the dual solution are possible. By employing a Lagrangian framework instead of a Dantzig-Wolfe framework, we can take advantage of this fact.

For a given vector $u \in \mathbb{R}_+^{m''}$, the *Lagrangian relaxation* of (1.2) is given by

$$z_{\text{LR}}(u) = \min_{s \in \mathcal{F}'} \left\{ c^\top s + u^\top \left(b'' - A''s \right) \right\}. \tag{2.22}$$

It is easily shown that $z_{\text{LR}}(u)$ is a lower bound on z_{IP} for any $u \geq 0$. The elements of the vector u are called *Lagrange multipliers* or *dual multipliers* with respect to the rows of $[A'', b'']$. Note that (2.22) is the same subproblem solved in the Dantzig-Wolfe method to generate the most negative



Lagrangian Method

Input: An instance OPT $(\mathcal{P}, c)$.
Output: A lower bound z_{LD} on the optimal solution value for the instance and a dual solution $\hat{u}_{\text{LD}} \in \mathbb{R}^{m''}$.

1. **Initialize:** Let $s_{\text{LD}}^{0} \in \mathcal{E}$ define some initial extreme point of $\mathcal{P}'$, u_{LD}^{0} some initial setting for the dual multipliers and set $t \leftarrow 0$.

2. **Master Problem:** Using the solution information gained from solving the pricing subproblem, and the previous dual setting u_{LD}^{t}, update the dual multipliers u_{LD}^{t+1}.

3. **Subproblem:** Call the subroutine OPT $\left(\mathcal{P}', c^{\top} - \left(u_{\text{LD}}^{t}\right)^{\top} A''\right)$, to solve

$$z_{\text{LD}}^{t} = \min_{s \in \mathcal{F}'} \left\{ \left(c^{\top} - \left(u_{\text{LD}}^{t}\right)^{\top} A''\right) s + b''^{\top} u_{\text{LD}}^{t} \right\}. \tag{2.24}$$

Let $s_{\text{LD}}^{t+1} \in \mathcal{E}$ be the optimal solution to this subproblem, if one is found.

4. If a prespecified stopping criterion is met, then output $z_{\text{LD}} = z_{\text{LD}}^{t}$ and $\hat{u}_{\text{LD}} = u_{\text{LD}}^{t}$, otherwise, go to Step 2

Figure 2.8: Outline of the Lagrangian method

reduced cost column. The problem

$$z_{\text{LD}} = \max_{u \in \mathbb{R}_{+}^{m''}} \{z_{\text{LR}}(u)\} \tag{2.23}$$

of maximizing this bound over all choices of dual multipliers is a dual to (1.2) called the *Lagrangian dual* and also provides a lower bound z_{LD}, which we call the *LD bound*. A vector $\hat{u}$ of multipliers that yield the largest bound are called *optimal (dual) multipliers*.

It is easy to see that $z_{\text{LR}}(u)$ is a piecewise-linear concave function and can be maximized by any number of methods for non-differentiable optimization, including the well-known *subgradient algorithm*. For a complete treatment of the various methods for solving this problem, see [41]. In Figure 2.8, we give an outline of the steps involved in the Lagrangian method. As in Dantzig-Wolfe, the main loop involves updating the dual solution and then generating an *improving* member of $\mathcal{E}$ by solving a subproblem. Unlike the Dantzig-Wolfe method, there is no approximation and hence no update step, but the method can nonetheless be viewed in the same frame of reference.

To more clearly see the connection to the Dantzig-Wolfe method, consider the dual of the Dantzig-Wolfe LP (2.10),

$$z_{\mathrm{DW}} = \max_{\alpha \in \mathbb{R}, u \in \mathbb{R}^{m''}_{+}} \left\{ \alpha + b''^{\top} u \;\middle|\; \alpha \leq \left(c^{\top} - u^{\top} A'' \right) s \;\forall s \in \mathcal{E} \right\}. \tag{2.25}$$

Letting $\eta = \alpha + b''^{\top} u$ and rewriting, we see that

$$z_{\mathrm{DW}} = \max_{\eta \in \mathbb{R}, u \in \mathbb{R}^{m''}_{+}} \left\{ \eta \;\middle|\; \eta \leq \left(c^{\top} - u^{\top} A'' \right) s + b''^{\top} u \;\forall s \in \mathcal{E} \right\} \tag{2.26}$$

$$= \max_{\eta \in \mathbb{R}, u \in \mathbb{R}^{m''}_{+}} \left\{ \min_{s \in \mathcal{E}} \left\{ \left(c^{\top} - u^{\top} A'' \right) s + b''^{\top} u \right\} \right\} = z_{\mathrm{LD}}. \tag{2.27}$$

Thus, we have that $z_{\mathrm{LD}} = z_{\mathrm{DW}}$ and that (2.23) is another formulation for the problem of calculating z_{D}. It is also interesting to observe that the set $\mathcal{S}\left(u^t_{\mathrm{LD}}, z^t_{\mathrm{LD}} - b''^{\top} u^t_{\mathrm{LD}}\right)$ is the set of alternative optimal solutions to the subproblem solved at iteration t in Step 3. The following theorem is a counterpart to Theorem 2.5 that follows from this observation.

Theorem 2.8 ([77]) *conv* $\left(\mathcal{S}\left(\hat{u}_{LD}, z_{LD} - b''^{\top} \hat{u}_{LD}\right)\right)$ *is a face of* $\mathcal{P}'$*. Also, if* F *is the face of optimal solutions to the linear program solved directly over* $\mathcal{P}' \cap \mathcal{Q}''$ *with objective function vector* c*, then* $F \subseteq$ *conv* $\left(\mathcal{S}\left(\hat{u}_{LD}, z_{LD} - b''^{\top} \hat{u}_{LD}\right)\right) \cap \mathcal{Q}''$.

Again, the proof is similar to that of Theorem 2.5. This shows that while the Lagrangian method does not maintain an explicit approximation, it does produce a face of $\mathcal{P}'$ containing the optimal face of solutions to the linear program solved over the approximation $\mathcal{P}' \cap \mathcal{Q}''$.

2.2 Integrated Decomposition Methods

In Section 2.1, we demonstrated that traditional decomposition approaches can be viewed as utilizing dynamically generated polyhedral information to improve the LP bound by building either an inner or an outer approximation of an implicitly defined polyhedron that approximates $\mathcal{P}$. The choice between inner and outer methods is largely an empirical one, but recent computational research has favored outer methods. In what follows, we discuss two methods for integrating inner and

outer methods. In Section 2.3 we introduce a third, relatively unknown, integrated method. Conceptually, these methods are not difficult to understand and can result in bounds that are improved over those achieved by either approach alone.

While traditional decomposition approaches build either an inner *or* an outer approximation, *integrated decomposition methods* build both an inner *and* an outer approximation. These methods follow the same basic loop as traditional decomposition methods, except that the master problem is required to generate both primal *and* dual solution information, and the subproblem can be either a separation problem *or* an optimization problem. The first two techniques we describe integrate the cutting plane method with either the Dantzig-Wolfe method or the Lagrangian method. The third technique, described in Section 2.3, is a cutting plane method that uses an inner approximation to perform separation.

2.2.1 Price-and-Cut

The integration of the cutting plane method with the Dantzig-Wolfe method results in a procedure that alternates between a subproblem that attempts to generate improving columns (the *pricing* subproblem) and a subproblem that attempts to generate improving valid inequalities (the *cutting* subproblem). Hence, we call the resulting method *price-and-cut*. When employed in a branch and bound framework, the overall technique is called *branch-and-price-and-cut*. This method has already been studied previously by a number of authors [10, 88, 47, 9, 86] and more recently by Arãgao and Uchoa [25].

As in the Dantzig-Wolfe method, the bound produced by price-and-cut can be thought of as resulting from the intersection of two approximating polyhedra. However, the Dantzig-Wolfe method required one of these, $\mathcal{Q}''$, to have a short description. With integrated methods, both polyhedra can have descriptions of exponential size. Hence, price-and-cut allows partial descriptions of both an inner polyhedron $\mathcal{P}_I$ *and* an outer polyhedron $\mathcal{P}_O$ to be generated dynamically. To optimize over the intersection of $\mathcal{P}_I$ and $\mathcal{P}_O$, we use a Dantzig-Wolfe reformulation as in (2.10), except that the $[A'', b'']$ is replaced by a matrix that changes dynamically. The outline of this method is shown in Figure 2.9.

Price-and-Cut Method

Input: An instance OPT$(\mathcal{P}, c)$.
Output: A lower bound z_{PC} on the optimal solution value for the instance, a primal solution $\hat{x}_{\text{PC}} \in \mathbb{R}^n$, an optimal decomposition $\hat{\lambda}_{\text{PC}} \in \mathbb{R}^{\mathcal{E}}$, a dual solution $(\hat{u}_{\text{PC}}, \hat{\alpha}_{\text{PC}}) \in \mathbb{R}^{m^t+1}$, and the inequalities $[D_{\text{PC}}, d_{\text{PC}}] \in \mathbb{R}^{m^t \times (n+1)}$.

1. **Initialize:** Construct an initial inner approximation

$$\mathcal{P}_I^0 = \left\{ \sum_{s \in \mathcal{E}^0} s\lambda_s \;\middle|\; \sum_{s \in \mathcal{E}^0} \lambda_s = 1, \lambda_s \geq 0 \; \forall s \in \mathcal{E}^0, \lambda_s = 0 \; \forall s \in \mathcal{E} \setminus \mathcal{E}^0 \right\} \subseteq \mathcal{P}' \quad (2.28)$$

from an initial set $\mathcal{E}^0$ of extreme points of $\mathcal{P}'$ *and* an initial outer approximation

$$\mathcal{P}_O^0 = \left\{ x \in \mathbb{R}^n \;\middle|\; D^0 x \geq d^0 \right\} \supseteq \mathcal{P}, \quad (2.29)$$

where $D^0 = A''$ and $d^0 = b''$, and set $t \leftarrow 0$, $m^0 = m''$.

2. **Master Problem:** Solve the Dantzig-Wolfe reformulation

$$\bar{z}_{\text{PC}}^t = \min_{\lambda \in \mathbb{R}_+^{\mathcal{E}}} \left\{ c^\top \left(\sum_{s \in \mathcal{E}} s\lambda_s \right) \;\middle|\; D^t \left(\sum_{s \in \mathcal{E}} s\lambda_s \right) \geq d^t, \sum_{s \in \mathcal{E}} \lambda_s = 1, \lambda_s = 0 \; \forall s \in \mathcal{E} \setminus \mathcal{E}^t \right\} \quad (2.30)$$

of the LP over the polyhedron $\mathcal{P}_I^t \cap \mathcal{P}_O^t$ to obtain the optimal value $\bar{z}_{\text{PC}}^t$, an optimal primal solution $\lambda_{\text{PC}}^t \in \mathbb{R}^{\mathcal{E}}$, an optimal fractional solution $x_{\text{PC}}^t = \sum_{s \in \mathcal{E}} s \left(\lambda_{\text{PC}}^t \right)_s$, and an optimal dual solution $\left(u_{\text{PC}}^t, \alpha_{\text{PC}}^t \right) \in \mathbb{R}^{m^t+1}$.

3. Do either (a) or (b).

   (a) **Pricing Subproblem and Update:** Call the subroutine OPT$(\mathcal{P}', c^\top - (u_{\text{PC}}^t)^\top D^t, \alpha_{\text{PC}}^t)$, generating a set $\tilde{\mathcal{E}}$ of *improving* members of $\mathcal{E}$ with negative reduced cost (defined in Figure 2.4). If $\tilde{\mathcal{E}} \neq \emptyset$, set $\mathcal{E}^{t+1} \leftarrow \mathcal{E}^t \cup \tilde{\mathcal{E}}$ to form a new inner approximation $\mathcal{P}_I^{t+1}$. If $\tilde{s} \in \mathcal{E}$ is a member of $\mathcal{E}$ with smallest reduced cost, then $\underline{z}_{\text{PC}}^t = rc(\tilde{s}) + \alpha_{\text{PC}}^t + \left(d^t \right)^\top u_{\text{PC}}^t$ provides a valid lower bound. Set $[D^{t+1}, d^{t+1}] \leftarrow [D^t, d^t]$, $\mathcal{P}_O^{t+1} \leftarrow \mathcal{P}_O^t$, $m^{t+1} \leftarrow m^t$, $t \leftarrow t+1$, and go to Step 2.

   (b) **Cutting Subproblem and Update:** Call the subroutine SEP$\left(\mathcal{P}, x_{\text{PC}}^t \right)$ to generate a set of *improving* valid inequalities $[\tilde{D}, \tilde{d}] \in \mathbb{R}^{\tilde{m} \times n+1}$ for $\mathcal{P}$, violated by x_{PC}^t. If violated inequalities were found, set $[D^{t+1}, d^{t+1}] \leftarrow \left[\begin{smallmatrix} D^t & d^t \\ \tilde{D} & \tilde{d} \end{smallmatrix} \right]$ to form a new outer approximation $\mathcal{P}_O^{t+1}$. Set $m^{t+1} \leftarrow m^t + \tilde{m}$, $\mathcal{E}^{t+1} \leftarrow \mathcal{E}^t$, $\mathcal{P}_I^{t+1} \leftarrow \mathcal{P}_I^t$, $t \leftarrow t+1$, and go to Step 2.

4. If $\tilde{\mathcal{E}} = \emptyset$ and no valid inequalities were found, output the bound $z_{\text{PC}} = \bar{z}_{\text{PC}}^t = \underline{z}_{\text{PC}}^t = c^\top x_{\text{PC}}^t$, $\hat{x}_{\text{PC}} = x_{\text{PC}}^t$, $\hat{\lambda}_{\text{PC}} = \lambda_{\text{PC}}^t$, $(\hat{u}_{\text{PC}}, \hat{\alpha}_{\text{PC}}) = \left(u_{\text{PC}}^t, \alpha_{\text{PC}}^t \right)$, and $[D_{\text{PC}}, d_{\text{PC}}] = [D^t, d^t]$.

Figure 2.9: Outline of the price-and-cut method

In examining the steps of this generalized method, the most interesting question that arises is how methods for generating improving columns and valid inequalities translate to this new dynamic setting. Potentially troublesome is the fact that column-generation results in a reduction of the bound $\bar{z}^t_{\text{PC}}$ produced by (2.30), while generation of valid inequalities is aimed at increasing it. Recall again, however, that while it is the bound $\bar{z}^t_{\text{PC}}$ that is directly produced by solving (2.30), it is the bound $\underline{z}^t_{\text{PC}}$ obtained by solving the pricing subproblem that one might claim is more relevant to our goal, and this bound can potentially be improved by generation of either valid inequalities or columns.

Improving columns can be generated in much the same way as they were in the Dantzig-Wolfe method. To generate new columns, we simply look for those with negative reduced cost, where reduced cost is defined to be the usual LP reduced cost with respect to the current reformulation. Having a negative reduced cost is still a necessary condition for a column to be improving. However, it is less clear how to generate improving valid inequalities. Consider an optimal fractional solution x^t_{PC} obtained by combining the members of $\mathcal{E}$ according to weights yielded by the optimal decomposition λ^t_{PC} in iteration t. Following a line of reasoning similar to that followed in analyzing the results of the Dantzig-Wolfe method, we can conclude that x^t_{PC} is in fact an optimal solution to an LP solved directly over $\mathcal{P}^t_I \cap \mathcal{P}^t_O$ with objective function vector c and that therefore, it follows from Theorem 2.2 that any improving inequality must be violated by x^t_{PC}. It thus seems sensible to consider separating x^t_{PC} from $\mathcal{P}$. This is the approach taken in the method of Figure 2.9. To demonstrate how the price-and-cut method works, we return to Example 1.

Example 1: SILP (Continued) We pick up the example at the last iteration of the Dantzig-Wolfe method and show how the bound can be further improved by dynamically generating valid inequalities.

Iteration 0. Solving the master problem with $\mathcal{E}^0 = \{(4,1),(5,5),(2,1),(3,4)\}$ and the initial inner approximation $\mathcal{P}^0_I = \text{conv}(\mathcal{E}^0)$ yields $(\lambda^0_{\text{PC}})_{(2,1)} = 0.58$ and $(\lambda^0_{\text{PC}})_{(3,4)} = 0.42$, $x^0_{\text{PC}} = (2.42, 2.25)$, and bound $\underline{z}^0_{\text{PC}} = \bar{z}^0_{\text{PC}} = 2.42$. Next, we solve the cutting subproblem SEP($\mathcal{P}, x^0_{\text{PC}}$), generating facet-defining inequalities of $\mathcal{P}$ that are violated by x^0_{PC}. One such facet-defining inequality, $x_1 \geq 3$, is illustrated in Figure 2.10(a). We add this inequality to the current set $D^0 = [A'', b'']$ to

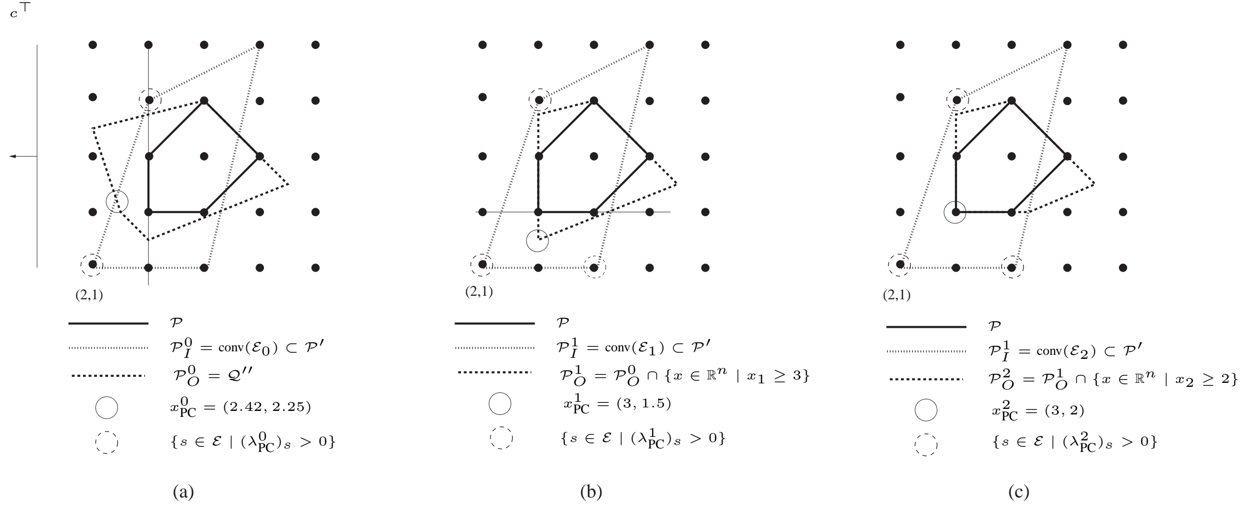


Figure 2.10: Price-and-cut method (Example 1: SILP)

form a new outer approximation $\mathcal{P}_O^1$, defined by the set D^1.

Iteration 1. Solving the new master problem, we obtain an optimal primal solution $(\lambda_{\mathrm{PC}}^1)_{(4,1)} = 0.42$, $(\lambda_{\mathrm{PC}}^1)_{(2,1)} = 0.42$, $(\lambda_{\mathrm{PC}}^1)_{(3,4)} = 0.17$, $x_{\mathrm{PC}}^1 = (3, 1.5)$, bound $\bar{z}_{\mathrm{PC}}^1 = 3$, as well as an optimal dual solution $\left(u_{\mathrm{PC}}^1, \alpha_{\mathrm{PC}}^1\right)$. Next, we consider the pricing subproblem. Since x_{PC}^1 is in the interior of $\mathcal{P}'$, every extreme point of $\mathcal{P}'$ has reduced cost 0 by Theorem 2.7. Therefore, there are no negative reduced cost columns, and we switch again to the cutting subproblem SEP $\left(\mathcal{P}, x_{\mathrm{PC}}^1\right)$. As illustrated in Figure 2.10(b), we find another facet-defining inequality of $\mathcal{P}$ violated by x_{PC}^1, $x_2 \geq 2$. We then add this inequality to form D^2 and further tighten the outer approximation, now $\mathcal{P}_O^2$.

Iteration 2. In the final iteration, we solve the master problem again to obtain $(\lambda_{\mathrm{PC}}^2)_{(4,1)} = 0.33$, $(\lambda_{\mathrm{PC}}^2)_{(2,1)} = 0.33$, $(\lambda_{\mathrm{PC}}^2)_{(3,4)} = 0.33$, $x_{\mathrm{PC}}^2 = (3, 2)$, bound $\bar{z}_{\mathrm{PC}}^2 = 3$. Now, since the primal solution is integral and is contained in $\mathcal{P}' \cap \mathcal{Q}''$, we know that $z_{\mathrm{PC}} = \bar{z}_{\mathrm{PC}}^2 = z_{\mathrm{IP}}$ and we terminate. ■

Let us now return to the TSP example to further explore the use of the price-and-cut method.

Example 3a: TSP (Continued) As described earlier, application of the Dantzig-Wolfe method along with the 1-tree relaxation for the TSP allows us to compute the bound z_{D} obtained by optimizing over the intersection of the 1-tree polyhedron (the inner polyhedron) with the polyhedron $\mathcal{Q}''$ (the outer polyhedron) defined by constraints (1.27) and (1.29). With price-and-cut, we can further improve the bound by allowing both the inner and the outer polyhedra to have large descriptions. For this purpose, let us now introduce the well-known *comb inequalities* [37, 38], which we generate to improve our outer approximation. A comb is defined by a set $H \subset V$, called the *handle*, and sets $T_1, T_2, \ldots, T_k \subset V$, called the *teeth*, which satisfy

$$H \cap T_i \neq \emptyset \;\text{ for } i = 1, \ldots, k,$$

$$T_i \setminus H \neq \emptyset \;\text{ for } i = 1, \ldots, k,$$

$$T_i \cap T_j = \emptyset \text{ for } 1 \leq i < j \leq k,$$

for some odd $k \geq 3$. Then, for $|V| \geq 6$ the comb inequality,

$$x(E(H)) + \sum_{i=1}^{k} x(E(T_i)) \leq |H| + \sum_{i=1}^{k} (|T_i| - 1) - \lceil k/2 \rceil \tag{2.31}$$

is valid and facet-defining for the TSP. Let the comb polyhedron be defined by constraints (1.27), (1.29), and (2.31).

There are no known efficient algorithms for solving the general facet identification problem for the comb polyhedron. To overcome this difficulty, one approach is to focus on comb inequalities with special forms. One subset of the comb inequalities, known as the *blossom inequalities*, is obtained by restricting the teeth to have exactly two members. The facet identification for the polyhedron comprised of the blossom inequalities and constraints (1.27) and (1.29) can be solved in polynomial time, a fact we return to shortly. Another approach is to use heuristic algorithms not guaranteed to find a violated comb inequality when one exists (see [3] for a survey). These heuristic algorithms could be applied in price-and-cut as part of the cutting subproblem in Step 3b to improve the outer approximation.

In Figure 2.6 of Section 2.1.2, we showed an optimal fractional solution $\hat{x}$ that resulted from

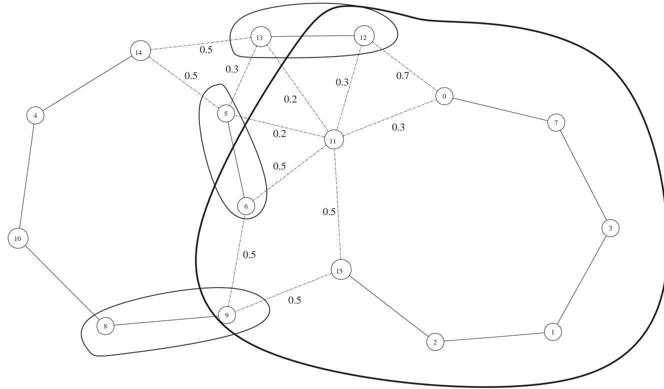


Figure 2.11: Price-and-cut method (Example 3a: TSP)

the solution of a Dantzig-Wolfe master problem and the corresponding optimal decomposition, consisting of six 1-trees. In Figure 2.11, we show the sets $H = \{0, 1, 2, 3, 6, 7, 9, 11, 12, 15\}, T_1 = \{5, 6\}, T_2 = \{8, 9\}$, and $T_3 = \{12, 13\}$ forming a comb that is violated by this fractional solution, since

$$\hat{x}(E(H)) + \sum_{i=1}^{k} \hat{x}(E(T_i)) = 11.3 > 11 = |H| + \sum_{i=1}^{k}(|T_i| - 1) - \lceil k/2 \rceil.$$

Such a violated comb inequality, if found, could be added to the description of the outer polyhedron to improve on the bound z_{D}. This shows the additional power of price-and-cut over the Dantzig-Wolfe method. Of course, it should be noted that it is also possible to generate such inequalities in the standard cutting plane method and to achieve the same bound improvement. ■

The choice of relaxation has a great deal of effect on the empirical behavior of decomposition algorithms. In Example 3a, we employed an inner polyhedron with integer extreme points. With such a polyhedron, the integrality constraints of the inner polyhedron have no effect and $z_{\text{D}} = z_{\text{LP}}$. In Example 3b, we consider a relaxation for which the bound z_{D} may be strictly improved over z_{LP} by employing an inner polyhedron that is not integral.

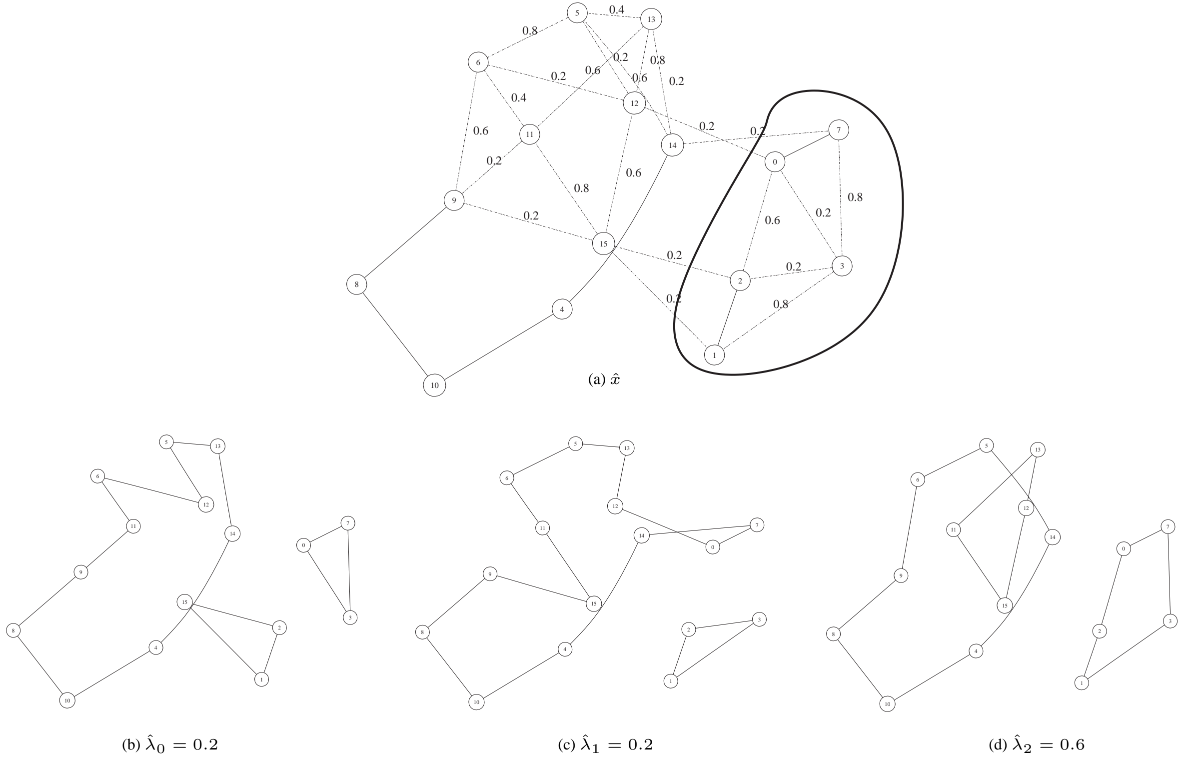
Example 3b: TSP Let G be a graph as defined in Example 3a for the TSP. A *2-matching* is a subgraph in which every vertex has degree two. Every TSP tour is hence a 2-matching. The *Minimum 2-Matching Problem* is a relaxation of TSP whose feasible region is described by the degree

(1.27), bound (1.29), and integrality (1.30) constraints of the TSP. Interestingly, the 2-matching polyhedron, which is implicitly defined to be the convex hull of the feasible region just described, can also be described by replacing the integrality constraints (1.30) with the blossom inequalities. Just as the SEC constraints provide an almost complete description of the 1-tree polyhedron, the blossom inequalities (plus degree and bound) constraints provide a complete description of the 2-matching polyhedron. Therefore, we could use this polyhedron as an outer approximation to the TSP polyhedron. In [65], Müller-Hannemann and Schwartz present several polynomial algorithms for optimizing over the 2-matching polyhedron. We can therefore also use the 2-matching relaxation in the context of price-and-cut to generate an inner approximation of the TSP polyhedron. Using integrated methods, it would then be possible to simultaneously build up an outer approximation of the TSP polyhedron consisting of the SECs (1.28). Note that this simply reverses the roles of the two polyhedra from Example 3a and thus would yield the same bound.

Figure 2.12 shows an optimal fractional solution arising from the solution of the master problem and the 2-matchings with positive weight in a corresponding optimal decomposition. Given this fractional subgraph, we could employ the separation algorithm discussed in Example 3a of Section 2.1.1 to generate the violated subtour $S = \{0, 1, 2, 3, 7\}$. ■

Another approach to generating improving inequalities in price-and-cut is to try to take advantage of the information contained in the optimal decomposition to aid in the separation procedure. This information, though computed by solving (2.30), is typically ignored. Consider the fractional solution x^t_{PC} generated in iteration t of the method in Figure 2.9. The optimal decomposition λ^t_{PC} for the master problem in iteration t provides a decomposition of x^t_{PC} into a convex combination of members of $\mathcal{E}$. We refer to elements of $\mathcal{E}$ that have a positive weight in this combination as *members of the decomposition*. The following theorem shows how such a decomposition can be used to derive an alternate necessary condition for an inequality to be improving. Because we apply this theorem in a more general context later in the paper, we state it in a general form.

Theorem 2.9 ([77]) *If $\hat{x} \in \mathbb{R}^n$ violates the inequality $(a, \beta) \in \mathbb{R}^{(n+1)}$ and $\hat{\lambda} \in \mathbb{R}^{\mathcal{E}}_{+}$ is such that $\sum_{s \in \mathcal{E}} \hat{\lambda}_s = 1$ and $\hat{x} = \sum_{s \in \mathcal{E}} s\hat{\lambda}_s$, then there must exist an $s \in \mathcal{E}$ with $\hat{\lambda}_s > 0$ such that s also*



(a) $\hat{x}$

(b) $\hat{\lambda}_0 = 0.2$ (c) $\hat{\lambda}_1 = 0.2$ (d) $\hat{\lambda}_2 = 0.6$

Figure 2.12: Finding violated inequalities in price-and-cut (Example 3b: TSP)

violates the inequality (a, β) *.*

Proof Let $\hat{x} \in \mathbb{R}^n$ and $(a, \beta) \in \mathbb{R}^{(n+1)}$ be given such that $a^\top \hat{x} < \beta$. Also, let $\hat{\lambda} \in \mathbb{R}^{\mathcal{E}}_{+}$ be given such that $\sum_{s \in \mathcal{E}} \hat{\lambda}_s = 1$ and $\hat{x} = \sum_{s \in \mathcal{E}} s\hat{\lambda}_s$. Suppose that $a^\top s \geq \beta$ for all $s \in \mathcal{E}$ with $\hat{\lambda}_s > 0$. Since $\sum_{s \in \mathcal{E}} \hat{\lambda}_s = 1$, we have $a^\top(\sum_{s \in \mathcal{E}} s\hat{\lambda}_s) \geq \beta$. Hence, $a^\top \hat{x} = a^\top(\sum_{s \in \mathcal{E}} s\hat{\lambda}_s) \geq \beta$, which is a contradiction. ■

In other words, an inequality can be improving only if it is violated by at least one member of the decomposition. If $\mathcal{I}$ is the set of all improving inequalities in iteration t, then the following corollary is a direct consequence of Theorem 2.9.

Corollary 2.10 ([77]) $\mathcal{I} \subseteq \mathcal{V} = \{(a, \beta) \in \mathbb{R}^{(n+1)} : a^\top s < \beta$ *for some* $s \in \mathcal{E}$ *such that* $(\lambda^t_{PC})_s > 0\}$.

The importance of these results is that in many cases, it is easier to separate members of $\mathcal{F}'$ from $\mathcal{P}$ than to separate arbitrary real vectors. There are a number of well-known polyhedra for which the

Separation using a Decomposition

Input: A decomposition $\lambda \in \mathbb{R}^{\mathcal{E}}$ of $\hat{x} \in \mathbb{R}^n$.
Output: A set $[D, d]$ of potentially improving inequalities.

1. Form the set $\mathcal{D} = \{s \in \mathcal{E} \mid \lambda_s > 0\}$.
2. For each $s \in \mathcal{D}$, call the subroutine SEP$(\mathcal{P}, s)$ to obtain a set $[\tilde{D}, \tilde{d}]$ of violated inequalities.
3. Let $[D, d]$ be composed of the inequalities found in Step 2 that are also violated by $\hat{x}$, so that $D\hat{x} < d$.
4. Return $[D, d]$ as the set of potentially improving inequalities.

Figure 2.13: Solving the cutting subproblem with the aid of a decomposition

problem of separating an arbitrary real vector is difficult, but the problem of separating a solution to a given relaxation is easy. This concept is formalized in Section 2.3.1 along with some more examples. In Figure 2.13, we propose a new separation procedure that can be embedded in price-and-cut that takes advantage of this fact. The procedure takes as input an arbitrary real vector $\hat{x}$ that has been previously decomposed into a convex combination of vectors with known structure. In price-and-cut, the arbitrary real vector x^t_{PC} is decomposed into a convex combination of members of $\mathcal{E}$ by solving the master problem (2.30). Rather than separating x^t_{PC} directly, the procedure consists of separating each one of the members of the decomposition in turn, then checking each inequality found for violation against x^t_{PC}.

The running time of this procedure depends in part on the cardinality of the decomposition. Carathéodory's Theorem assures us that there exists a decomposition with at most $\dim(\mathcal{P}^t_I) + 1$ members. Unfortunately, even if we limit our search to a particular known class of valid inequalities, the number of such inequalities violated by each member of $\mathcal{D}$ in Step 2 may be extremely large, and these inequalities may not be violated by x^t_{PC} (such an inequality cannot be improving). Unless we enumerate *every* inequality in the set $\mathcal{V}$ from Corollary 2.10, either implicitly or explicitly, the procedure does not guarantee that an improving inequality is found, even if one exists. In cases where it is possible to examine the set $\mathcal{V}$ in polynomial time, the worst-case complexity of the entire procedure is polynomially equivalent to that of optimizing over $\mathcal{P}'$. Obviously, it is unlikely that

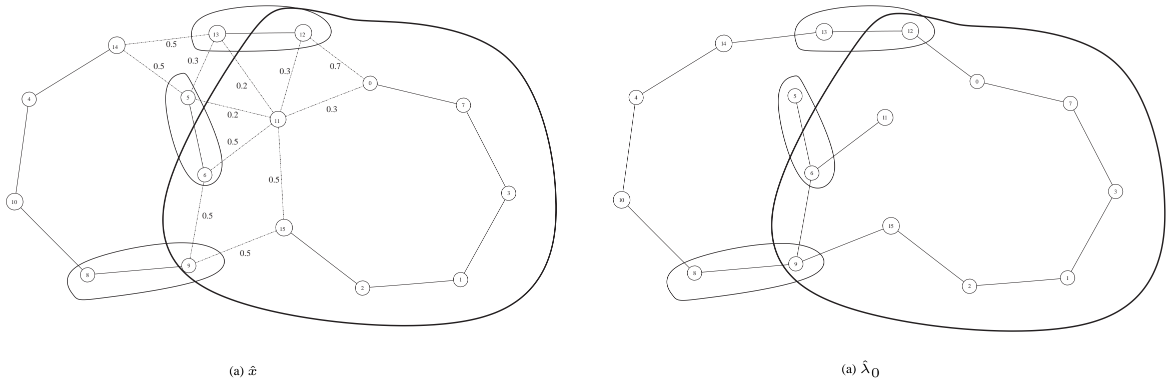
the set $\mathcal{V}$ can be examined in polynomial time in situations when separating x^t_{PC} is itself an $\mathcal{NP}$-complete problem. In such cases, the procedure to select inequalities that are likely to be violated by x^t_{PC} in Step 2 is necessarily a problem-dependent heuristic. The effectiveness of such heuristics can be improved in a number of ways, some of which are discussed in [80].

Note that members of the decomposition in iteration t must belong to the set $\mathcal{S}(u^t_{\text{PC}}, \alpha^t_{\text{PC}})$, as defined by (2.19). It follows that the convex hull of the decomposition is a subset of conv $\left(\mathcal{S}(u^t_{\text{PC}}, \alpha^t_{\text{PC}})\right)$ that contains x^t_{PC} and can be thought of as a surrogate for the face of optimal solutions to an LP solved directly over $\mathcal{P}^t_I \cap \mathcal{P}^t_O$ with objective function vector c. Combining this corollary with Theorem 2.1, we conclude that separation of $\mathcal{S}\left(u^t_{\text{PC}}, \alpha^t_{\text{PC}}\right)$ from $\mathcal{P}$ is a sufficient condition for an inequality to be improving. Although this sufficient condition is difficult to verify in practice, it does provide additional motivation for the method described in Figure 2.13.

Example 1: SILP (Continued) Returning to the cutting subproblem in iteration 0 of the price-and-cut method, we have a decomposition $x^0_{\text{PC}} = (2.42, 2.25) = 0.58(2,1) + 0.42(3,4)$, as depicted in Figure 2.10(a). Now, instead of trying to solve the subproblem SEP($\mathcal{P}, x^0_{\text{PC}}$), we instead solve SEP($\mathcal{P}, s$), for each $s \in \mathcal{D} = \{(2,1),(3,4)\}$. In this case, when solving the separation problem for $s = (2,1)$, we find the same facet-defining inequality of $\mathcal{P}$ as we did by separating x^0_{PC} directly.

Similarly, in iteration 1, we have a decomposition of $x^2_{\text{PC}} = (3, 1.5)$ into a convex combination of $\mathcal{D} = \{(4,1),(2,1),(3,4)\}$. Clearly, solving the separation problem for either $(2,1)$ or $(4,1)$ produces the same facet-defining inequality as with the original method. ■

Example 3a: TSP (Continued) Returning again to Example 3a, recall the optimal fractional solution and the corresponding optimal decomposition arising during solution of the TSP by the Dantzig-Wolfe method in Figure 2.6. Figure 2.11 shows a comb inequality violated by this fractional solution. By Theorem 2.9, at least one of the members of the optimal decomposition shown in Figure 2.6 must also violate this inequality. In fact, the member with index 0, also shown in Figure 2.14, is the only such member. Note that the violation is easy to discern from the structure of this integral solution. Let $\hat{x} \in \{0,1\}^E$ be the incidence vector of a 1-tree. Consider a subset H of V whose



(a) $\hat{x}$ (a) $\hat{\lambda}_0$

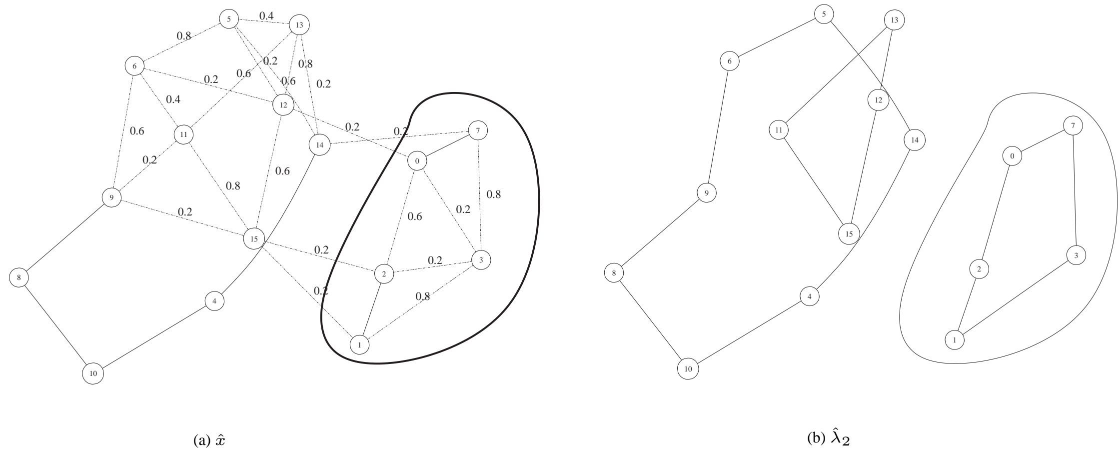
Figure 2.14: Using the optimal decomposition to find violated inequalities in price-and-cut (Example 3a: TSP)

induced subgraph in the 1-tree is a path with edge set P. Consider also an odd set O of edges of the 1-tree of cardinality at least 3 and disjoint from P, such that each edge has one endpoint in H and one endpoint in $V \setminus H$. Taking the set H to be the handle and the endpoints of each member of O to be the teeth, it is easy to verify that the corresponding comb inequality is violated by the 1-tree, since

$$\hat{x}(E(H)) + \sum_{i=1}^{k} \hat{x}(E(T_i)) = |H| - 1 + \sum_{i=1}^{k} (|T_i| - 1) > |H| + \sum_{i=1}^{k} (|T_i| - 1) - \lceil k/2 \rceil.$$

Hence, searching for such configurations in the members of the decomposition, as suggested in the procedure of Figure 2.13, may lead to the discovery of comb inequalities violated by the optimal fractional solution. In this case, such a configuration does in fact lead to discovery of the previously indicated comb inequality. Note that we have restricted ourselves in the above discussion to the generation of blossom inequalities. The teeth, as well as the handles, can have more general forms that may lead to the discovery of more general forms of violated combs. ■

Example 3b: TSP (Continued) Returning now to Example 3a, recall the optimal fractional solution and the corresponding optimal decomposition, consisting of the 2-matchings shown in Figure 2.12. Previously, we produced a set of vertices defining a SEC violated by the fractional point by using a minimum cut algorithm with the optimal fractional solution as input. Now, let us consider



(a) $\hat{x}$ (b) $\hat{\lambda}_2$

Figure 2.15: Using the optimal decomposition to find violated inequalities in price-and-cut (Example 3b: TSP)

applying the procedure of Figure 2.13 by examining the members of the decomposition in order to discover inequalities violated by the optimal fractional solution. Let $\hat{x} \in \{0,1\}^E$ be the incidence vector of a 2-matching. If the corresponding subgraph does not form a tour, then it must be disconnected. The vertices corresponding to any connected component thus define a violated SEC. By determining the connected components of each member of the decomposition, it is easy to find violated SECs. In fact, for any 2-matching, every component of the 2-matching forms a SEC that is violated by exactly 1. For the 2-matching corresponding to $\hat{s}$, we have $\hat{x}(E(S)) = |S| > |S| - 1$. Figure 2.15(b) shows the third member of the decomposition along with a violated SEC defined by one of its components. This same SEC is also violated by the optimal fractional solution. ■

There are many variants of the price-and-cut method shown in Figure 2.9. Most significant is the choice of which subproblem to execute during Step 3. It is easy to envision a number of heuristic rules for deciding this. For example, one obvious rule is to continue generating columns until no more are available and then switch to valid inequalities for one iteration, then generate columns again until none are available. This can be seen as performing a "complete" dual solution update before generating valid inequalities. Further variants can be obtained by not insisting on a "complete" dual update before solving the pricing problem [36, 17]. This rule could easily be inverted to

generate valid inequalities until no more are available and then generate columns. A hybrid rule in which some sort of alternation occurs is a third option. The choice among these options is primarily empirical.

2.2.2 Relax-and-Cut

Just as with the Dantzig-Wolfe method, the Lagrangian method of Figure 2.8 can be integrated with the cutting plane method to yield a procedure several authors have recently termed *relax-and-cut* [26, 62, 56]. This is done in much the same fashion as in price-and-cut, with a choice in each iteration between solving a pricing subproblem and a cutting subproblem. In each iteration that the cutting subproblem is solved, the generated valid inequalities are added to the description of the outer polyhedron, which is explicitly maintained as the algorithm proceeds. As with the traditional Lagrangian method, no explicit inner polyhedron is maintained, but the algorithm can again be seen as one that computes a face of the implicitly defined inner polyhedron that contains the optimal face of solutions to a linear program solved over the intersection of the two polyhedra. When employed within a branch and bound framework, we call the overall method *branch-and-relax-and-cut*.

An outline of the relax-and-cut method is shown in Figure 2.16. The question again arises as to how to ensure that the inequalities being generated in the cutting subproblem are improving. In the case of the Lagrangian method, this is a much more difficult issue since we cannot assume the availability of the same primal solution information available within price-and-cut. Furthermore, we cannot verify the condition of Corollary 2.2, which is the best available necessary condition for an inequality to be improving. Nevertheless, *some* primal solution information is always available in the form of the solution s^t_{RC} to the last pricing subproblem that was solved. Intuitively, separating s^t_{RC} makes sense since the infeasibilities present in s^t_{RC} may possibly be removed through the addition of valid inequalities violated by s^t_{RC}.

As with both the cutting plane and price-and-cut methods, the difficulty is that the valid inequalities generated by separating s^t_{RC} from $\mathcal{P}$ may not be improving, as Guignard first observed in [40]. To deepen understanding of the potential effectiveness of the valid inequalities generated, we further examine the relationship between s^t_{RC} and x^t_{PC} by recalling again the results from Section 2.1.2.

Relax-and-Cut Method

Input: An instance OPT$(\mathcal{P}, c)$.
Output: A lower bound z_{RC} on the optimal solution value for the instance and a dual solution $\hat{u}_{\text{RC}} \in \mathbb{R}^{m^t}$.

1. **Initialize:** Let $s^0_{\text{RC}} \in \mathcal{E}$ define some initial extreme point of $\mathcal{P}'$ and construct an initial outer approximation
$$\mathcal{P}^0_O = \left\{ x \in \mathbb{R}^n \;\middle|\; D^0 x \geq d^0 \right\} \supseteq \mathcal{P}, \tag{2.32}$$
where $D^0 = A''$ and $d^0 = b''$. Let $u^0_{\text{RC}} \in \mathbb{R}^{m''}$ be some initial set of dual multipliers associated with the constraints $\left[D^0, d^0\right]$. Set $t \leftarrow 0$ and $m^t = m''$.

2. **Master Problem:** Using the solution information gained from solving the pricing subproblem, and the previous dual solution u^t_{RC}, update the dual solution (if the pricing problem was just solved) or initialize the new dual multipliers (if the cutting subproblem was just solved) to obtain $u^{t+1}_{\text{RC}} \in \mathbb{R}^{m^t}$.

3. Do either (a) or (b).

   (a) **Pricing Subproblem:** Call the subroutine OPT $\left(\mathcal{P}', c - \left(u^t_{\text{RC}}\right)^\top D^t\right)$ to obtain
   $$z^t_{\text{RC}} = \min_{s \in \mathcal{F}'} \left\{ \left(c^\top - \left(u^t_{\text{RC}}\right) D^t\right) s + d^t \left(u^t_{\text{RC}}\right) \right\}. \tag{2.33}$$
   Let $s^{t+1}_{\text{RC}} \in \mathcal{E}$ be the optimal solution to this subproblem. Set $\left[D^{t+1}, d^{t+1}\right] \leftarrow \left[D^t, d^t\right]$, $\mathcal{P}^{t+1}_O \leftarrow \mathcal{P}^t_O$, $m^{t+1} \leftarrow m^t$, $t \leftarrow t+1$, and go to Step 2.

   (b) **Cutting Subproblem**: Call the subroutine SEP $\left(\mathcal{P}, s^t_{\text{RC}}\right)$ to generate a set of *improving* valid inequalities $\left[\tilde{D}, \tilde{d}\right] \in \mathbb{R}^{\tilde{m} \times n+1}$ for $\mathcal{P}$, violated by s^t_{RC}. If violated inequalities were found, set $\left[D^{t+1}, d^{t+1}\right] \leftarrow \left[\begin{smallmatrix} D^t & d^t \\ \tilde{D} & \tilde{d} \end{smallmatrix}\right]$ to form a new outer approximation $\mathcal{P}^{t+1}_O$. Set $m^{t+1} \leftarrow m^t + \tilde{m}$, $s^{t+1}_{\text{RC}} \leftarrow s^t_{\text{RC}}$, $t \leftarrow t+1$, and go to Step 2.

4. If a pre-specified stopping criterion is met, then output $z_{\text{RC}} = z^t_{\text{RC}}$ and $\hat{u}_{\text{RC}} = u^t_{\text{RC}}$.

5. Otherwise, go to Step 2.

Figure 2.16: Outline of the relax-and-cut method

Consider the set $\mathcal{S}(u^t_{\text{RC}}, z^t_{\text{RC}})$, where z^t_{RC} is obtained by solving the pricing subproblem (2.33) from Figure 2.16 and the set $\mathcal{S}(\cdot,\cdot)$ is as defined in (2.19). In each iteration where the pricing subproblem is solved, s^{t+1}_{RC} is a member of $\mathcal{S}(u^t_{\text{RC}}, z^t_{\text{RC}})$. In fact, $\mathcal{S}(u^t_{\text{RC}}, z^t_{\text{RC}})$ is exactly the set of alternative solutions to this pricing subproblem. In price-and-cut, a number of members of this set are available, one of which must be violated in order for a given inequality to be improving. This yields a verifiable necessary condition for a generated inequality to be improving. Relax-and-cut, in its most straightforward incarnation, produces one member of this set. Even if improving inequalities exist, it is possible that none of them are violated by the member of $\mathcal{S}(u^t_{\text{RC}}, z^t_{\text{RC}})$ so produced, especially if it would have had a small weight in the optimal decomposition produced by the corresponding iteration of price-and-cut.

It is important to note that by keeping track of the solutions to the Lagrangian subproblem that are produced while solving the Lagrangian dual, one can approximate the optimal decomposition and the optimal fractional solution produced by solving (2.30). This is the approach taken by the volume algorithm [7] and a number of other subgradient-based methods. As in price-and-cut, when this fractional solution is an inner point of $\mathcal{P}'$, all members of $\mathcal{F}'$ are alternative optimal solutions to the pricing subproblem and the bound is not improved over what the cutting plane method alone would produce. In this case, solving the cutting subproblem to obtain additional inequalities is unlikely to yield further improvement.

As with price-and-cut, there are again many variants of the algorithm shown in Figure 2.16, depending on the choice of subproblem to execute at each step. One such variant is to alternate between each of the subproblems, first solving one and then the other [56]. In this case, the Lagrangian dual is not solved to optimality before solving the cutting subproblem. Alternatively, another approach is to solve the Lagrangian dual all the way to optimality before generating valid inequalities. Again, the choice is primarily empirical.

2.3 Decompose-and-Cut

In Section 2.2.1, we introduced the idea of using the decomposition to aid in separation in the context of price-and-cut. In this chapter we extend this idea to the traditional cutting-plane method

and introduce a relatively unknown decomposition-based algorithm we refer to as *decompose-and-cut*.

First, we review the well-known *template paradigm* for separation and introduce a concept called *structured separation*. Then, we describe a separation algorithm called *decompose-and-cut* that is closely related to the integrated decomposition methods we have already described, which utilizes several of the concepts introduced earlier. From this, we will introduce a class of cutting planes that we have termed *decomposition cuts*.

2.3.1 The Template Paradigm and Structured Separation

The ability to generate valid inequalities for $\mathcal{P}$ violated by a given real vector is a crucial step in many of the methods discussed in this paper. Ideally, we would like to be able to solve the general facet identification problem for $\mathcal{P}$, allowing us to generate a violated valid inequality whenever one exists. This is clearly not practical in most cases, since the complexity of this problem is the same as that of solving the original ILP. In practice, the subproblem SEP($\mathcal{P}, x^t_{\text{CP}}$) in Step 3 of the cutting plane method pictured in Figure 2.1 is usually solved by dividing the valid inequalities for $\mathcal{P}$ into *template classes* with known structure. Procedures are then designed and executed for identifying violated members of each class individually.

A template class (or simply *class*) of valid inequalities for $\mathcal{P}$ is a set of related valid inequalities that describes a polyhedron containing $\mathcal{P}$, so we can identify each class with its associated polyhedron. The *template paradigm* was introduced in Section 2.1.1. In Example 3b, we described two well-known classes of valid inequalities for the TSP, the *subtour elimination constraints* and the *comb inequalities*. Both classes have an identifiable coefficient structure and describe polyhedra containing $\mathcal{P}$.

Consider a polyhedron $\mathcal{C}$ that is the closure with respect to a class of inequalities valid for $\mathcal{P}$. The separation problem for the class $\mathcal{C}$ of valid inequalities for $\mathcal{P}$ is defined to be the facet identification problem over the polyhedron $\mathcal{C}$. In other words, the separation problem for a class of valid inequalities depends on the form of the inequality and is independent of the polyhedron $\mathcal{P}$. It follows that the worst-case running time for solving the separation problem is also independent of

$\mathcal{P}$. In particular, the separation problem for a particular class of inequalities may be much easier to solve than the general facet identification problem for $\mathcal{P}$. Therefore, in practice, the separation problem is usually attempted over "easy" classes first, and more difficult classes are attempted only when needed. This was shown in the context of TSP in Section 2.1. The facet identification problem for the polyhedron described by all SECs, as well as variable bound constraints, is equivalent to the minimum cut problem (an optimization problem) and hence is polynomially solvable [4], whereas the facet identification problem for the convex hull of feasible solutions to the TSP is an $\mathcal{NP}$-complete problem in general. In general, the intersection of the polyhedra associated with the classes of inequalities for which the separation problem can be reasonably solved is not equal to $\mathcal{P}$.

It is also frequently the case that when applying a sequence of separation routines for progressively more difficult classes of inequalities, routines for the more difficult classes assume implicitly that the solution to be separated satisfies all inequalities of the the easier classes. In the case of the TSP, for instance, any solution passed to the subroutine for separating the comb inequalities is generally assumed to satisfy the degree and subtour elimination constraints. This assumption can allow the separation algorithms for subsequent classes to be implemented more efficiently.

In many cases, the complexity of the separation problem is also affected by the structure of the real vector being separated. In Section 2.2.1, we informally introduced the notion that a solution vector with known structure may be easier to separate from a given polyhedron than an arbitrary one and illustrated this phenomenon for TSP in Examples 3a and 3b. This is a concept called *structured separation* that arises quite frequently in the solution of combinatorial optimization problems where the original formulation is of exponential size. When using the cutting plane method to solve the LP relaxation of the TSP, for example, as described in Example 3a, we must generate the SECs dynamically. It is thus possible that the intermediate solutions are integer-valued but nonetheless infeasible because they violate some SEC that is not present in the current approximation. When the current solution is optimal, however, it is easy to determine whether it violates a SEC by simply examining the connected components of the underlying support graph, as described earlier. This process can be done in $O(|V| + |E|)$ time. For an arbitrary real vector, the separation problem for SECs, the

minimum cut problem, is more difficult, taking $O\left(|E||V| + |V|^2 \log |V|\right)$ time. To further illustrate this point, let us now introduce another classical example from combinatorial optimization, the *Vehicle Routing Problem* (VRP).

Example 4: VRP The *Vehicle Routing Problem* (VRP) was introduced by Dantzig and Ramser [24]. In this $\mathcal{NP}$-hard optimization problem, a fleet of k vehicles with uniform capacity C must service known customer demands for a single commodity from a common depot at minimum cost. Let $V = \{1, \ldots, |V|\}$ index the set of customers and let the depot have index 0. Associated with each customer $i \in V$ is a demand d_i. The cost of travel from customer i to j is denoted c_{ij} and we assume that $c_{ij} = c_{ji} > 0$ if $i \neq j$ and $c_{ii} = 0$.

By constructing an associated complete undirected graph G with vertex set $N = V \cup \{0\}$ and edge set E, we can formulate the VRP as an integer program. A *route* is a set of vertices $R = \{i_1, i_2, \ldots, i_m\}$ such that the members of R are distinct. The edge set of R is $E_R = \{\{i_j, i_{j+1}\} \mid j \in 0, \ldots, m\}$, where $i_0 = i_{m+1} = 0$. A feasible solution is then any subset of E that is the union of the edge sets of k disjoint routes $R_i, i \in \{1, \ldots, k\}]$, each of which satisfies the capacity restriction, i.e., $\sum_{j \in R_i} d_j \leq C$, $\forall i \in \{1, \ldots, k\}$. Each route corresponds to a set of customers serviced by one of the k vehicles. To simplify the presentation, let us define some additional notation.

By associating a variable with each edge in the graph, we obtain the following formulation of

this ILP [51]:

$$\min \sum_{e \in E} c_e x_e,$$

$$x(\delta(\{0\})) = 2k, \tag{2.34}$$

$$x(\delta(\{v\})) = 2 \qquad \forall v \in V, \tag{2.35}$$

$$x(\delta(S)) \geq 2b(S) \; \forall S \subseteq V, \; |S| > 1, \tag{2.36}$$

$$x_e \in \{0,1\} \qquad \forall e \in E(V), \tag{2.37}$$

$$x_e \in \{0,1,2\} \qquad \forall e \in \delta(\{0\}). \tag{2.38}$$

Here, $b(S)$ represents a lower bound on the number of vehicles required to service the set S of customers. Equations (2.34) ensure that there are exactly k vehicles, each departing from and returning to the depot, while equations (2.35) require that each customer must be serviced by exactly one vehicle. Inequalities (2.36), known as the *generalized subtour elimination constraints* (GSECs), can be viewed as a generalization of the subtour elimination constraints from TSP, and enforce connectivity of the solution, as well as ensuring that no route has total demand exceeding capacity C. For ease of computation, we can define $b(S) = \lceil (\sum_{i \in S} d_i)/C \rceil$, a trivial lower bound on the number of vehicles required to service the set S of customers.

The set of feasible solutions to the VRP is

$$\mathcal{F} = \{x \in \mathbb{R}^E \mid x \text{ satisfies } (2.34) - (2.38)\}$$

and we call $\mathcal{P} = \text{conv}(\mathcal{F})$ the *VRP polyhedron*. Many classes of valid inequalities for the VRP polyhedron have been reported in the literature (see [66] for a survey). Significant effort has been devoted to developing efficient algorithms for separating an arbitrary fractional point using these classes of inequalities (see [58] for recent results).

We concentrate here on the separation of GSECs. The separation problem for GSECs was shown to be $\mathcal{NP}$-complete by Harche and Rinaldi (see [5]), even when $b(S)$ is taken to be $\lceil (\sum_{i \in S} d_i)/C \rceil$.

In [58], Lysgaard, et al. review heuristic procedures for generating violated GSECs. Although GSECs are part of the formulation presented above, there are exponentially many of them, so we generate them dynamically. We discuss three relaxations of the VRP: the *Multiple Traveling Salesman Problem*, the *Perfect b-Matching Problem*, and the *Minimum Degree-Constrained k-Tree Problem*. For each of these alternatives, violation of GSECs by solutions to the relaxation can be easily discerned.

Perfect b-Matching Problem. With respect to the graph G, the *Perfect b-Matching Problem* is to find a minimum-weight subgraph of G such that $x(\delta(v)) = b_v \; \forall v \in V$. This problem can be formulated by dropping the GSECs from the VRP formulation, resulting in the feasible set

$$\mathcal{F}' = \{x \in \mathbb{R}^E \mid x \text{ satisfies } (2.34), (2.35), (2.37), (2.38)\}.$$

In [65], Müller-Hannemann and Schwartz present several fast polynomial algorithms for solving b-Matching. The polyhedron $\mathcal{P}_O$ consists of the GSECs (2.36) in this case.

In [64], Miller uses the b-matching relaxation to solve the VRP by branch-and-relax-and-cut. He suggests generating GSECS violated by b-matchings as follows. Consider a member s of $\mathcal{F}'$ and its support graph G_s (a b-matching). If G_s is disconnected, then each component immediately induces a violated GSEC. On the other hand, if G_s is connected, we first remove the edges incident to the depot vertex and find the connected components, which comprise the routes described earlier. To identify a violated GSEC, we compute the total demand of each route, checking whether it exceeds capacity. If not, the solution is feasible for the original ILP and does not violate any GSECs. If so, the set S of customers on any route whose total demand exceeds capacity induces a violated GSEC. This separation routine runs in $O(|V|+|E|)$ time and can be used in any of the integrated decomposition methods previously described. Figure 2.17(a) shows an example vector that could arise during execution of either price and cut or decompose-and-cut, along with a decomposition into a convex combination of two b-matchings, shown in Figures 2.17(b) and 2.17(c). In this example, the capacity $C = 35$, and by inspection we find a violated GSEC in the second b-matching (c) with S equal to the marked component. This inequality is also violated by the optimal fractional solution, since

$\hat{x}(\delta(S)) = 3.0 < 4.0 = 2b(S)$.

Minimum Degree-Constrained k-Tree Problem. A *k-tree* is defined as a spanning subgraph of G that has $|V| + k$ edges (recall that G has $|V| + 1$ vertices). A *degree-constrained k-tree* (k-DCT), as defined by Fisher in [32], is a k-tree with degree $2k$ at vertex 0. The *Minimum k-DCT Problem* is that of finding a minimum-cost k-DCT, where the cost of a k-DCT is the sum of the costs on the edges present in the k-DCT. Fisher [32] introduced this relaxation as part of a Lagrangian relaxation-based algorithm for solving the VRP.

The k-DCT polyhedron is obtained by first adding the redundant constraint

$$x(E) = |V| + k, \tag{2.39}$$

then deleting the degree constraints (2.35), and finally, relaxing the capacity to $C = \sum_{i \in S} d_i$. Relaxing the capacity constraints gives $b(S) = 1$ for all $S \subseteq V$ and replaces constraints (2.36) with

$$\sum_{e \in \delta(S)} x_e \geq 2, \forall S \subseteq V, \ |S| > 1. \tag{2.40}$$

The feasible region of the Minimum k-DCT Problem is then

$$\mathcal{F}' = \{x \in \mathbb{R}^E \mid x \text{ satisfies } (2.34), (2.37), (2.39), (2.40)\}.$$

This time, the polyhedron $\mathcal{P}_O$ is comprised of the constraints (2.35) and the GSECs (2.36). Since the constraints (2.35) can be represented explicitly, we focus again on generation of violated GSECs. In [92], Wei and Yu give a polynomial algorithm for solving the Minimum k-DCT Problem that runs in $O(|V|^2 \log |V|)$ time. In [63], Martinhon et al. study the use of the k-DCT relaxation for the VRP in the context of branch-relax-and-cut. Again, consider separating a member s of $\mathcal{F}'$ from the polyhedron defined by all GSECS. It is easy to see that for GSECs, an algorithm identical to that described above can be applied. Figure 2.17(a) also shows a vector that could arise during the execution of either the price-and-cut or decompose-and-cut algorithms, along with a decomposition into a convex combination of four k-DCTs, shown in Figures 2.17(d) through 2.17(g). Removing

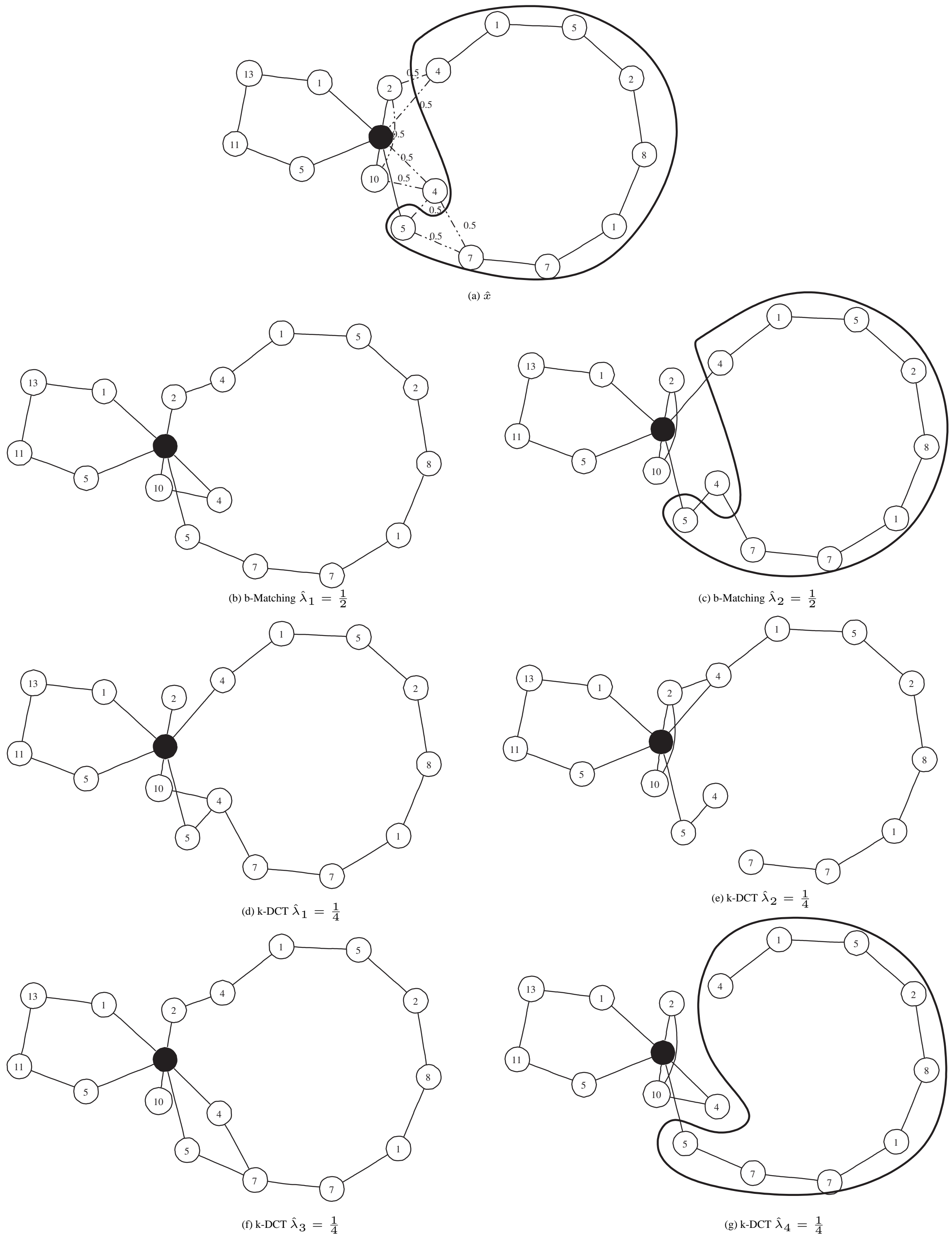
the depot edges and checking each component's demand, we easily identify the violated GSEC shown in Figure 2.17(g).

Multiple Traveling Salesman Problem. The *Multiple Traveling Salesman Problem* (k-TSP) is an uncapacitated version of the VRP obtained by adding the degree constraints to the k-DCT polyhedron. The feasible region of the k-TSP is

$$\mathcal{F}' = \{x \in \mathbb{R}^E \mid x \text{ satisfies } (2.34), (2.35), (2.37), (2.38), (2.40)\}.$$

Although the k-TSP is an $\mathcal{NP}$-hard optimization problem, small instances can be solved effectively by transformation into an equivalent TSP obtained by adjoining to the graph $k-1$ additional copies of vertex 0 and its incident edges. In this case, the polyhedron $\mathcal{P}_O$ is again comprised solely of the GSECs (2.36). In [80], Ralphs et al. report on an implementation of branch-and-cut using the k-TSP relaxation to aid in separation.

Consider separating a member s of $\mathcal{F}'$ from the polyhedron defined by all GSECs. We first construct the subgraph corresponding to s (a k-TSP) with all edges incident to the depot vertex removed. We then find the connected components, which comprise the routes described earlier. In order to identify a GSEC that violates s, we simply compute the total demand of each route, checking whether it exceeds capacity. If not, the solution is feasible for the original ILP and does not violate any GSECs. If so, the set S of customers in any route whose total demand exceeds capacity induces a violated GSEC. In this manner, we can separate a given member of $\mathcal{F}'$ from $\mathcal{P}$ with GSECs in $O(n)$ time. This separation routine can be used to generate GSECs with any of the dynamic decomposition methods previously described. Figure 2.18 gives an example of a fractional solution (a) to the VRP decomposed into three k-TSP tours (b,c,d). In this example, the capacity $C = 6000$. So, by inspection we can see that the top component of the second k-TSP tour (b) induces a violated GSEC. Adding this constraint also cuts off the original fractional solution. In [80], Ralphs et al. reports on the use of this decomposition algorithm for separation of GSECs. This work was the first known attempt at using the decomposition to aid in finding violated cuts and served as the original motivation for this thesis. ■



(a) $\hat{x}$

(b) b-Matching $\hat{\lambda}_1 = \frac{1}{2}$

(c) b-Matching $\hat{\lambda}_2 = \frac{1}{2}$

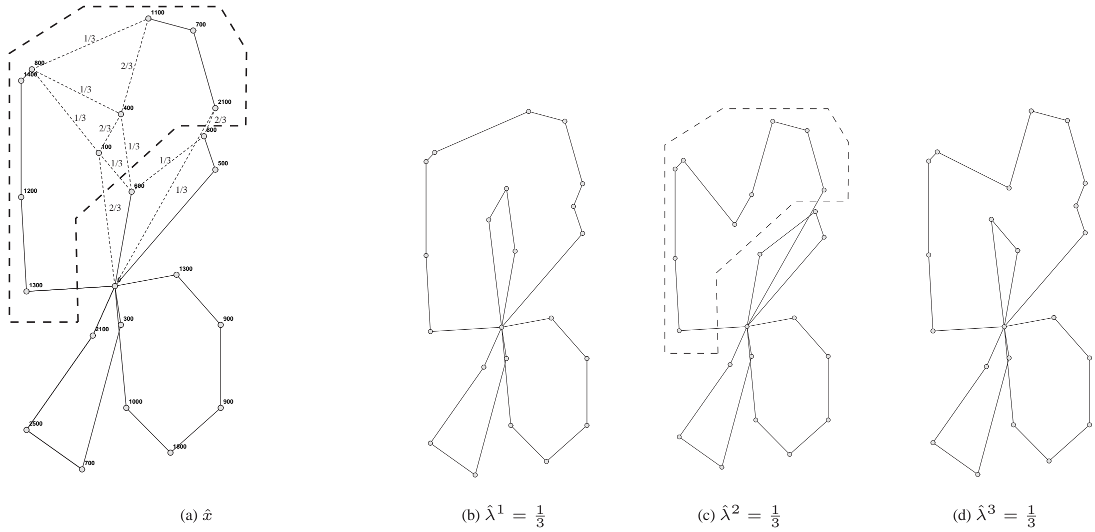
(d) k-DCT $\hat{\lambda}_1 = \frac{1}{4}$

(e) k-DCT $\hat{\lambda}_2 = \frac{1}{4}$

(f) k-DCT $\hat{\lambda}_3 = \frac{1}{4}$

(g) k-DCT $\hat{\lambda}_4 = \frac{1}{4}$

Figure 2.17: Example of a decomposition into b-matchings and k-DCTs



(a) $\hat{x}$ (b) $\hat{\lambda}^1 = \frac{1}{3}$ (c) $\hat{\lambda}^2 = \frac{1}{3}$ (d) $\hat{\lambda}^3 = \frac{1}{3}$

Figure 2.18: Example of decomposition VRP/k-TSP

In our framework, the concept of structured separation is combined with the template paradigm in specifying template classes of inequalities for which separation of integral solutions is much easier, in a complexity sense, than separation of arbitrary real vectors over that same class. That is, for some given relaxation, we consider separating solutions that are known to be integral, in particular, members of $\mathcal{F}'$. We now examine a separation paradigm called *decompose-and-cut* that can take advantage of our ability to easily separate solutions with structure.

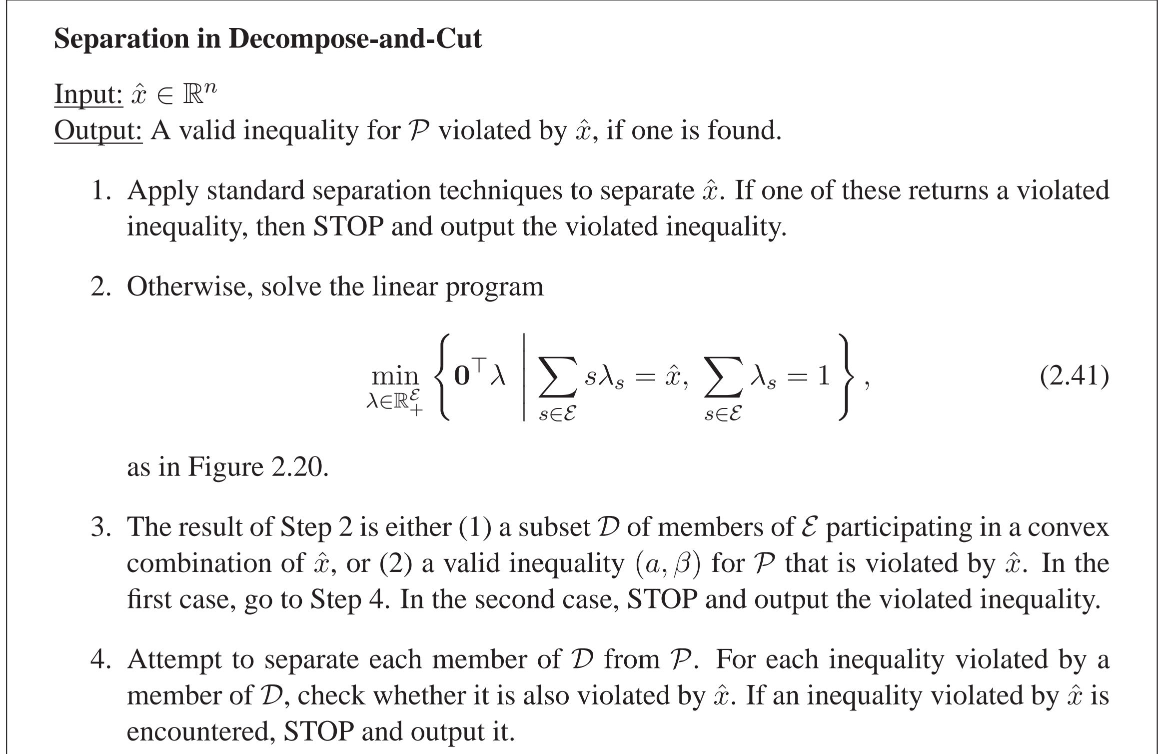
2.3.2 Separation Using an Inner Approximation

The use of an inner approximation to aid in separation, as is described in the procedure of Figure 2.13, is easy to extend to a traditional branch-and-cut framework using a technique we call *decompose-and-cut*, originally proposed by Ralphs in [78] and further developed in [48] and [80]. Suppose that we are given an optimal fractional solution x^t_{CP} obtained during iteration t of the cutting plane method. By first *decomposing* x^t_{CP} (i.e., expressing x^t_{CP} as a convex combination of members of $\mathcal{E} \subseteq \mathcal{F}'$) and then separating each member of this decomposition from $\mathcal{P}$ in the fashion described in Figure 2.13, we may be able to find valid inequalities for $\mathcal{P}$ that are violated by x^t_{CP}. The difficult step is finding the decomposition of x^t_{CP}. This can be accomplished by solving a linear

program whose columns are the members of $\mathcal{E}$, as described in Figure 2.19. This linear program is reminiscent of (2.10) and in fact can be solved using an analogous column-generation scheme, as described in Figure 2.20. This scheme can be seen as the "inverse" of the method described in Section 2.2.1, since it begins with the fractional solution x^t_{CP} and tries to compute a decomposition, instead of the other way around. By the equivalence of optimization and facet identification, we can conclude that the problem of finding a decomposition of x^t_{CP} is polynomially equivalent to that of optimizing over $\mathcal{P}'$.

It should be noted here, that, on average, the master problem (2.41) can be solved much more efficiently than the linear program in a typical Dantzig-Wolfe method. The reason is twofold. First, this linear program is just a feasibility problem. The costs involved are only used to drive out the artificial variables. So, there is no concept here of *best solution*, we are simply looking for any solution (a decomposition), or proof that one does not exist. The second reason is algebraic. Because the right hand side of constraint (2.43) is the current fractional point $\hat{x}$, we only need to consider the support of that vector, i.e., where $\hat{x}_i \neq 0$. This is due to the fact that for every component i where $\hat{x}_i = 0$, constraint (2.43) requires that for any column selected, $\lambda_s > 0$, that column's component must also be zero ($s_i = 0$). This can also be optionally enforced in the subproblem if the solver can handle such restrictions. In addition, for the case where the original space has only binary variables, we know that for each component i, $s_i \in \{0, 1\}$. Therefore, for each component i where $\hat{x}_i = 1$, constraint (2.43) is redundant because of the presence of (2.44). The presolver for the linear programming solver can easily remove these constraints, thereby reducing the computational cost at each iteration of the master solve.

Once the decomposition is found, it can be used as before to locate a violated valid inequality. In Figure 2.21(a) we show an example of this based on the small integer program defined in Example 1. In this case, the fractional point x_{CP} is decomposed into two extreme points of $\mathcal{P}'$. We then separate each of these points using valid inequalities for $\mathcal{P}$. As mentioned in Theorem 2.9, if one of these inequalities separates x_{CP}, then it must also separate one of the extreme points in its decomposition. This case is shown in Figure 2.21(a). Hence, we can perform heuristic separation on the fractional point at a possibly lower computational cost since separation of the extreme points is typically much



Separation in Decompose-and-Cut

Input: $\hat{x} \in \mathbb{R}^n$
Output: A valid inequality for $\mathcal{P}$ violated by $\hat{x}$, if one is found.

1. Apply standard separation techniques to separate $\hat{x}$. If one of these returns a violated inequality, then STOP and output the violated inequality.
2. Otherwise, solve the linear program
$$\min_{\lambda \in \mathbb{R}_+^{\mathcal{E}}} \left\{ \mathbf{0}^\top \lambda \;\middle|\; \sum_{s \in \mathcal{E}} s \lambda_s = \hat{x},\ \sum_{s \in \mathcal{E}} \lambda_s = 1 \right\}, \tag{2.41}$$
as in Figure 2.20.
3. The result of Step 2 is either (1) a subset $\mathcal{D}$ of members of $\mathcal{E}$ participating in a convex combination of $\hat{x}$, or (2) a valid inequality (a, β) for $\mathcal{P}$ that is violated by $\hat{x}$. In the first case, go to Step 4. In the second case, STOP and output the violated inequality.
4. Attempt to separate each member of $\mathcal{D}$ from $\mathcal{P}$. For each inequality violated by a member of $\mathcal{D}$, check whether it is also violated by $\hat{x}$. If an inequality violated by $\hat{x}$ is encountered, STOP and output it.

Figure 2.19: Separation in the decompose-and-cut method

easier than separation of the fractional points.

In contrast to the algorithm proposed in Figure 2.13 for price-and-cut, it is possible that $x^t_{\text{CP}} \notin \mathcal{P}'$. This scenario is depicted in Figure 2.21(b). This could occur in the context of the standard cutting plane method, if exact separation methods for $\mathcal{P}'$ are too expensive to apply consistently. In this case, it is obviously not possible to find a decomposition in Step 2 of Figure 2.19. The proof of infeasibility for the linear program (2.41), however, provides an inequality separating x^t_{CP} from $\mathcal{P}'$ at no additional expense. This comes directly from the well-known Farkas Lemma. We refer to this class of cuts as *decomposition cuts*. Hence, even if we fail to find a decomposition, we still find an inequality valid for $\mathcal{P}$ and violated by x^t_{CP}. This idea was originally suggested in [78] and was further developed in [48] in the context of the Vehicle Routing Problem (VRP). A similar concept was also discovered and developed independently by Applegate, et al. [2] for use with TSP. They termed these cuts *non-template cuts*, which has a nice symmetry with the template idea we discussed in the last section. In the following section, we generalize this approach to our

Decomposition Method in Decompose-and-Cut

Input: $\hat{x} \in \mathbb{R}^n$
Output: Either (1) a valid inequality for $\mathcal{P}$ violated by $\hat{x}$; or (2) a subset $\mathcal{D}$ of $\mathcal{E}$ and a vector $\hat{\lambda} \in \mathbb{R}^{\mathcal{E}}_{+}$ such that $\sum_{s \in \mathcal{D}} \lambda_s s = \hat{x}$ and $\sum_{s \in \mathcal{D}} \lambda_s = 1$.

1. **Initialize**: Construct an initial inner approximation

$$\mathcal{P}_I^0 = \left\{ \sum_{s \in \mathcal{E}^0} s\lambda_s \;\middle|\; \sum_{s \in \mathcal{E}^0} \lambda_s = 1, \lambda_s \geq 0 \; \forall s \in \mathcal{E}^0, \lambda_s = 0 \; \forall s \in \mathcal{E} \setminus \mathcal{E}^0 \right\} \subseteq \mathcal{P}' \quad (2.42)$$

   from an initial set $\mathcal{E}^0$ of extreme points of $\mathcal{P}'$ and set $t \leftarrow 0$.

2. **Master Problem**: Solve the Dantzig-Wolfe reformulation

$$\begin{aligned} \min\ & x^+ + x^-, \\ & \sum_{s \in \mathcal{E}} s\lambda_s + x^+ - x^- = \hat{x}, && (2.43) \\ & \sum_{s \in \mathcal{E}} \lambda_s = 1, && (2.44) \\ & \lambda_s = 0 \quad \forall s \in \mathcal{E} \setminus \mathcal{E}^t, \\ & \lambda \in \mathbb{R}^{\mathcal{E}}_{+}, \;\; x^+, x^- \in \mathbb{R}^n_{+}. \end{aligned}$$

   to obtain the optimal value $\bar{z}^t_{\text{DC}} = x^+ + x^-$, an optimal primal solution $\lambda^t_{\text{DC}} \in \mathbb{R}^{\mathcal{E}}_{+}$, and an optimal dual solution $\left(u^t_{\text{DC}}, \alpha^t_{\text{DC}}\right) \in \mathbb{R}^{n+1}$. If $x^+ + x^- \leq 0$, then we have found a decomposition; let $\mathcal{D} = \{s \in \mathcal{E} \mid \lambda_s > 0\}$ and STOP.

3. **Subproblem**: Call the subroutine OPT $\left(\mathcal{P}', -u^t_{\text{DC}}, \alpha^t_{\text{DC}}\right)$, generating a set of $\tilde{\mathcal{E}}$ of *improving* members of $\mathcal{E}$ with negative reduced cost, where the reduced cost of $s \in \mathcal{E}$ is

$$rc(s) = -u^t_{\text{DC}} s - \alpha^t_{\text{DC}}. \quad (2.45)$$

4. **Update**: If $\tilde{\mathcal{E}} \neq \emptyset$, set $\mathcal{E}^{t+1} \leftarrow \mathcal{E}^t \cup \tilde{\mathcal{E}}$ to form the new inner approximation

$$\mathcal{P}_I^{t+1} = \left\{ \sum_{s \in \mathcal{E}^{t+1}} s\lambda_s \;\middle|\; \sum_{s \in \mathcal{E}^{t+1}} \lambda_s = 1, \lambda_s \geq 0 \; \forall s \in \mathcal{E}^{t+1}, \lambda_s = 0 \; \forall s \in \mathcal{E} \setminus \mathcal{E}^{t+1} \right\} \subseteq \mathcal{P}', \quad (2.46)$$

   set $t \leftarrow t + 1$, and go to Step 2.

5. If $\tilde{\mathcal{E}} = \emptyset$, we have failed to find a decomposition. The point $\hat{x} \notin \mathcal{P}'$, and the valid inequality $\left(-u^t_{\text{DC}}, \alpha^t_{\text{DC}}\right)$ is violated by $\hat{x}$. STOP.

Figure 2.20: Outline of the decomposition method for decompose-and-cut

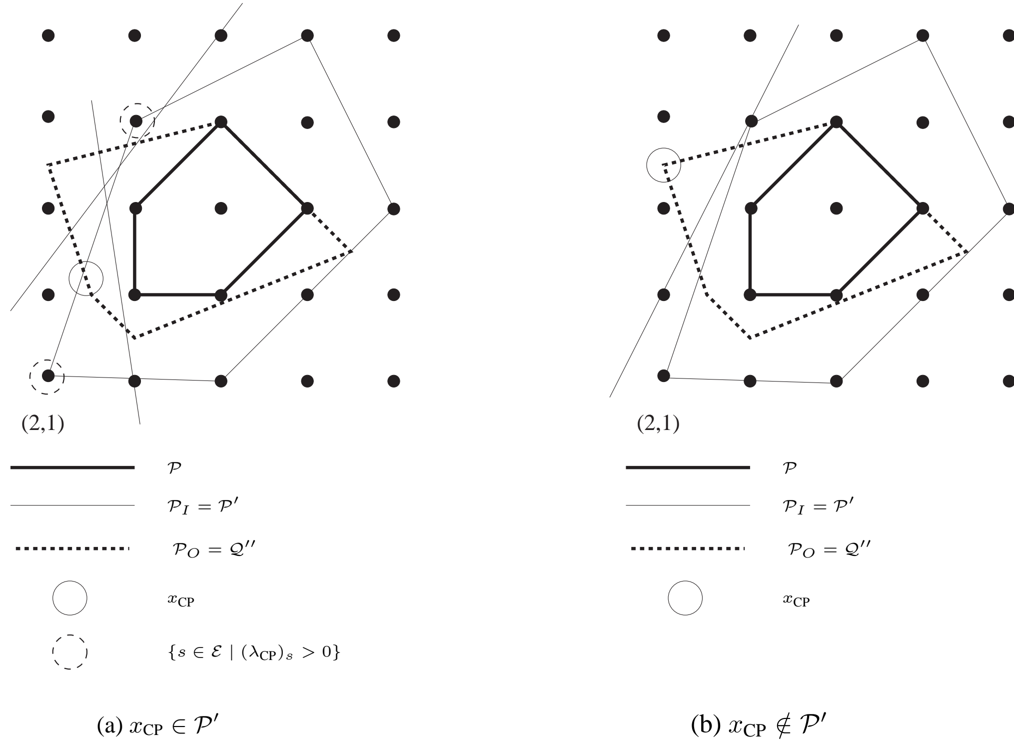


Figure 2.21: Decompose-and-cut

decomposition framework and recognize its potential as a method for generating cuts based on any number of relaxations.

2.3.3 Decomposition Cuts

Consider the case of the standard cutting plane method, in which we have a set of valid inequalities that form some outer approximation $\mathcal{P}^O$. For the same problem, consider an inner method with which we generate some inner approximation $\mathcal{P}^I$. As discussed earlier, we assume here that we have efficient methods for generating these approximations implicitly by solving SEP($\mathcal{P}^O, x$) or OPT($\mathcal{P}^I, c$). Let's also assume that $\left(\mathcal{P}^I \cap \mathcal{P}^O\right) \subset \mathcal{P}^O$.

The goal in the decompose-and-cut algorithm is to find a decomposition or prove that one does not exist. Therefore, we set up the model (2.41) as a feasibility problem. In this case, it is easier to implement the algorithm by introducing a set of slack variables x^+ and x^-, similar to what we did in *Phase 1* for the Dantzig-Wolfe method. This is shown in equation (2.43) in Figure 2.20.

In Section 3.4, we will discuss more of the practical details of this two-phase approach used in a practical implementation of the Dantzig-Wolfe method. In the Phase 1 model, the slack variables serve as bounds on the dual variables to drive the pricing mechanism for generating columns, which can attempt to improve feasibility. In the previous section, we have already considered the case when $\hat{x} \in \mathcal{P}'$. In this section we are are focusing on the case when $\hat{x} \notin \mathcal{P}'$. As mentioned earlier, the proof of this provides what we call a decomposition cut. This decomposition cut is conceptually quite simple. Each step of the Dantzig-Wolfe method is to generate extreme points of $\mathcal{P}'$ that have negative reduced costs. When we fail to find such a point, we have proven that the reduced cost vector must be greater than or equal to zero for all points in $\mathcal{P}'$. However, since $\hat{x} \notin \mathcal{P}'$, it is guaranteed that $\hat{x}$ violates the inequality defined by this vector. Following the notation of Figure 2.20, this means that

$$u^t_{\text{DC}} s + \alpha^t_{\text{DC}} \leq 0 \;\; \forall s \in \mathcal{P}' \text{ and}$$

$$u^t_{\text{DC}} \hat{x} + \alpha^t_{\text{DC}} > 0.$$

One of the key advantages of inner methods is the ability to generate bounds that are typically much tighter than what can typically be found using generic cutting planes. Since we often do not know the facets of the polytope $\mathcal{P}'$ or we cannot generate them efficiently, inner methods attack the problem with optimization, rather than separation. However, despite the potential from improved bounds, inner methods are often unsuccessful due to numerous other factors, such as convergence. That is, the expense of generating better bounds is not always worth the trouble.

Decomposition cuts, as described in this context, provide a possible way to get the *best of both worlds*. If we stay in the context of the direct cutting plane method, we can avoid some of the difficulties that come about from inner methods. However, we can still obtain the improved bounds by using these ideas. Each time the algorithm fails to produce a decomposition, the decomposition cut provides a violated inequality that is valid for the chosen relaxation $\mathcal{P}'$. In addition, it is possible for us to choose any number of relaxations when attempting to find the decomposition. This idea was first alluded to by Kopman [48] when solving VRP. In the most extreme case, he attempted to decompose the fractional point into a convex combination of extreme points of the original VRP

polytope by solving the column-generation subproblem heuristically. In this sense, he was assured to fail at finding a decomposition, but had a good chance of producing strong cuts. The computational evidence of such an approach was and remains somewhat of an open question. In Section 5.1, we provide evidence that it has some potential. The software framework presented in Chapter 4 provides an easy-to-use environment for experimenting with such ideas.

Chapter 3

Algorithmic Details

This chapter describes some of the issues and considerations when developing a solver based on the integrated methods discussed in the first two chapters. Specifically, we focus here on the implementation details related to price-and-cut when embedded in a branch-and-bound algorithm. Our primary computational focus, thus far, in this research, has been on *branch-and-price-and-cut*. Although the theoretical framework extends nicely to *branch-and-relax-and-cut*, the implementation details of this method are left for future research. From the abundance of literature discussing variants of integrated methods used to address real-world applications, it is clear that they are important in practice, but there has been relatively little work on the computational details of implementing such methods. In Chapter 4, we introduce a new open-source software framework called DIP (**D**ecomposition for **I**nteger **P**rogramming) that we hope can help to facilitate the in-depth study of these methods and the computational implications of their many algorithmic choices. The development of DIP and application to several problem classes helped us bridge the gap between theory and implementation. Here, we detail some of the issues and discoveries that arose during this development work.

Some of the issues we discuss in this chapter are relatively straightforward but worth mentioning for the benefit of future developers of these methods. In some cases, we provide a simple solution, while in other cases we can only provide some guidance, leaving the *best practice* as an open question. Much like other areas of optimization, the key to success in implementation of these

methods is to provide a large toolbox of techniques that can be used to attack any particular problem. It is rare to find a technique that helps in every instance. However, if it helps in some instance without hurting most others, it is worth including. The laborious part of developing optimization software is tuning the solver to find that set of techniques to turn on by default and those to leave as user options. In the following sections, we provide a starting point for such a study by introducing ideas that were useful for the applications we tried during our research.

3.1 Branching for Inner Methods

Thus far, we have focused on methods for finding strong bounds. The overall goal is to embed these bounding methods in a branching framework that divides the compact space into disjoint subproblems whose union covers the complete set of feasible solutions. Effective branching methods are a quite important element of the overall solution scheme and have been studied extensively for the case of branch-and-cut methods [52]. For branch-and-price methods, there has also been some recent research in this area [89]. Branching methods as applied to branch-and-price-and-cut have received much less attention in the literature. As is the case with most branch-and-price-and-cut research in the literature, most of the work on branching methods is application-specific. In this research, we take a very generic approach to branching, with the goal of keeping the framework both theoretically and computationally straightforward to implement.

In this section, we explain how a basic version of branching can be automated in the context of inner methods by reverting back to the compact space. We first explain the general concept and show an example of how it works in the context of branch-and-price-and-cut. We then briefly consider the same setup in the context of branch-and-relax-and-cut. Since Lagrangian methods do not, by default, guarantee a primal solution that can be mapped back to the compact space, there are some additional things to consider.

In order to develop branching methods for the overall decomposition framework, we return to the mapping (2.18) defined in Section 2.1.2. When integrating generation of valid inequalities with inner methods, this projection into the space of the compact formulation is precisely what allows us to add cuts to the extended formulation without any change in the complexity of the system. This

same idea can also be used to allow for a simple implementation of branching.

Branching for Price-and-Cut Consider the standard branching method used in the branch-and-cut algorithm. After solving the linear programming relaxation to obtain $\hat{x} \in \mathbb{R}^n$, we choose an integer-constrained variable x_i for which $\hat{x}_i \notin \mathbb{Z}$ and produce two disjoint subproblems by enforcing $x_i \leq \lfloor \hat{x}_i \rfloor$ in one subproblem and $x_i \geq \lceil \hat{x}_i \rceil$ in the other. In standard branch-and-cut, the branching constraints are enforced by simply updating the variable bounds in each subproblem. Now, consider the Dantzig-Wolfe reformulation presented in Section 2.1.2 and the price-and-cut algorithm defined in Section 2.2.1. In the compact formulation, let all variable bounds be considered explicitly as constraints by moving them into the set of side constraints $[A'', b'']$. This simple trick greatly simplifies the process of branching in the context of the Dantzig-Wolfe reformulation. Using the mapping (2.18), the variable bounds can now be written as a set of constraints in the extended formulation as $l_i \leq \left(\sum_{s \in \mathcal{E}} s\lambda_s\right)_i \leq u_i \forall i \in I$. After solving the master linear program to obtain $\hat{\lambda}$, we use the mapping to construct a solution in the original space. Then, as in standard branch-and-cut, we choose a variable x_i whose value is currently fractional $\left(\sum_{s \in \mathcal{E}} s\hat{\lambda}_s\right)_i = \hat{x}_i \notin \mathbb{Z}$ and produce two disjoint subproblems by enforcing $\left(\sum_{s \in \mathcal{E}} s\lambda_s\right)_i \leq \lfloor \hat{x}_i \rfloor$ in one subproblem and $\left(\sum_{s\lambda_s \in \mathcal{E}} s\lambda_s\right)_i \geq \lceil \hat{x}_i \rceil$ in the other. Since these are *branching constraints* rather than the standard *branching variables* we enforce them by adding them directly to $[A'', b'']$.

To illustrate this, we return to our first example, picking up from the last iteration of the Dantzig-Wolfe method as described in Section 2.1.2.

Example 1 : SILP (Continued) In the final iteration of the Dantzig-Wolfe method, we had constructed the inner approximation $\mathcal{P}_I^0 = \text{conv}\{(4,1),(5,5),(2,1),(3,4)\}$ and solved the restricted master to give the solution $(\lambda_{\text{DW}})_{(2,1)} = 0.58$ and $(\lambda_{\text{DW}})_{(3,4)} = 0.42$. For this example, let the superscript for the polyhedron represent the node number. Solving the subproblem, we showed that no more improving columns could be found and the current bound was optimal for the chosen relaxation. For the sake of illustration, let us assume that no violated cuts were found. Therefore, we have completed the calculation of the root node bound and now need to check to see whether the solution is in fact feasible to the original problem. Using the mapping (2.18), we compose the

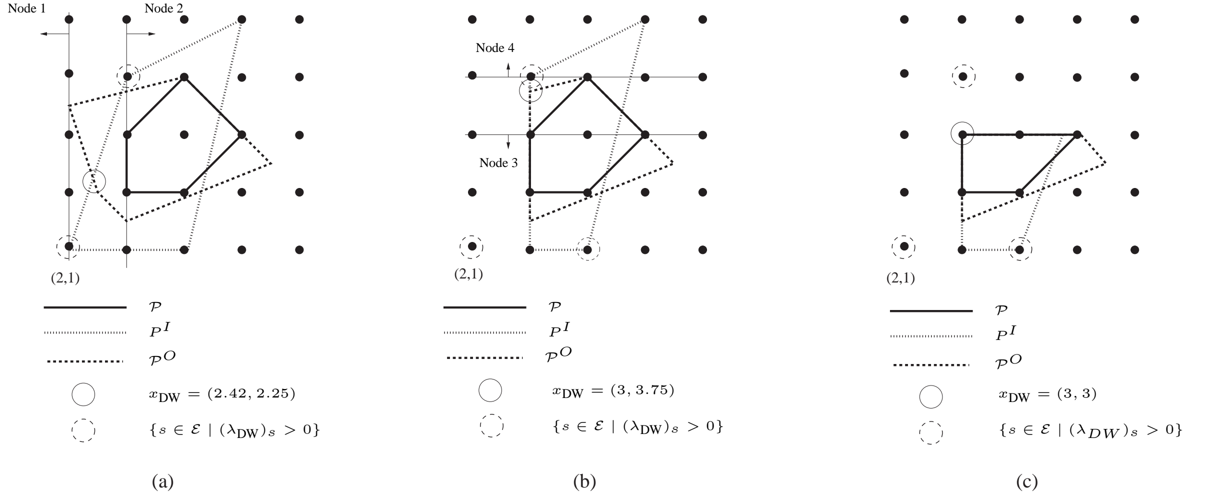


Figure 3.1: Branching in the Dantzig-Wolfe method (Example 1: SILP)

solution in the compact space as $x_{\text{DW}} = (2.42, 2.25)$. Since the solution is fractional, we branch on the most fractional variable x_0 by creating two new nodes:

$$\begin{aligned}
\text{Node 1: } \mathcal{P}_I^1 &= \text{conv } \left\{x \in \mathbb{R}^2 \mid x \text{ satisfies } (1.6) - (1.11), (1.22) \text{ and } x_0 \leq 2\right\}, \\
\mathcal{P}_O^1 &= \left\{x \in \mathbb{R}^2 \mid x \text{ satisfies } (1.12) - (1.20) \text{ and } x_0 \leq 2\right\}, \\
\text{Node 2: } \mathcal{P}_I^2 &= \text{conv } \left\{x \in \mathbb{R}^2 \mid x \text{ satisfies } (1.6) - (1.11), (1.22) \text{ and } x_0 \geq 3\right\}, \\
\mathcal{P}_O^2 &= \left\{x \in \mathbb{R}^2 \mid x \text{ satisfies } (1.12) - (1.20) \text{ and } x_0 \geq 3\right\}.
\end{aligned}$$

Notice here that we choose to include the branching constraint in both the master and the subproblem. Adding these constraints to the subproblem can improve convergence at the expense of increasing the computation time spent on each subproblem solve. The choice is purely empirical and can differ from application to application. This step is depicted in Figure 3.1(a).

We now use the mapping between the compact formulation and the extended formulation to produce the equivalent branching cuts in the master as follows:

$$\text{Node 1: } 4\,(\lambda_{\text{DW}})_{(4,1)} + 5\,(\lambda_{\text{DW}})_{(5,5)} + 2\,(\lambda_{\text{DW}})_{(2,1)} + 3\,(\lambda_{\text{DW}})_{(3,4)} \leq 2,$$

$$\text{Node 2: } 4\,(\lambda_{\text{DW}})_{(4,1)} + 5\,(\lambda_{\text{DW}})_{(5,5)} + 2\,(\lambda_{\text{DW}})_{(2,1)} + 3\,(\lambda_{\text{DW}})_{(3,4)} \geq 3.$$

We then solve the master problem for node 1 and immediately declare this node infeasible. We then move to node 2 and solve the master problem, which gives $(\lambda_{\text{DW}})_{(4,1)} = 0.04$, $(\lambda_{\text{DW}})_{(2,1)} = 0.04$, $(\lambda_{\text{DW}})_{(3,4)} = 0.92$, and $x_{\text{DW}} = (3, 3.75)$. This solution is depicted in Figure 3.1(b). Next, we solve the subproblem in order to generate new columns but find that no more exist that can improve the bound. Therefore, we are done with node 2 and have a new lower bound $z_{\text{DW}} = 3$. Since the solution is fractional, we once again branch, this time on x_1, creating two new nodes, as depicted in Figure 3.1(b):

$$\begin{aligned}
\text{Node 3: } \mathcal{P}_I^3 &= \text{conv}\left\{x \in \mathbb{R}^2 \mid x \text{ satisfies } (1.6) - (1.11), (1.22), x_0 \geq 3 \text{ and } x_1 \leq 3\right\}, \\
\mathcal{P}_O^3 &= \left\{x \in \mathbb{R}^2 \mid x \text{ satisfies } (1.12) - (1.20), x_0 \geq 3 \text{ and } x_1 \leq 3\right\}, \\
\text{Node 4: } \mathcal{P}_I^4 &= \text{conv}\left\{x \in \mathbb{R}^2 \mid x \text{ satisfies } (1.6) - (1.11), (1.22), x_0 \geq 3 \text{ and } x_1 \geq 4\right\}, \\
\mathcal{P}_O^4 &= \left\{x \in \mathbb{R}^2 \mid x \text{ satisfies } (1.12) - (1.20), x_0 \geq 3 \text{ and } x_1 \geq 4\right\}.
\end{aligned}$$

Using the mapping again gives the following branching cuts in the respective nodes:

$$\begin{aligned}
\text{Node 3: } & 4\,(\lambda_{\text{DW}})_{(4,1)} + 5\,(\lambda_{\text{DW}})_{(5,5)} + 2\,(\lambda_{\text{DW}})_{(2,1)} + 3\,(\lambda_{\text{DW}})_{(3,4)} \geq 3, \\
& 1\,(\lambda_{\text{DW}})_{(4,1)} + 5\,(\lambda_{\text{DW}})_{(5,5)} + 1\,(\lambda_{\text{DW}})_{(2,1)} + 4\,(\lambda_{\text{DW}})_{(3,4)} \leq 3, \\
\text{Node 4: } & 4\,(\lambda_{\text{DW}})_{(4,1)} + 5\,(\lambda_{\text{DW}})_{(5,5)} + 2\,(\lambda_{\text{DW}})_{(2,1)} + 3\,(\lambda_{\text{DW}})_{(3,4)} \geq 3, \\
& 1\,(\lambda_{\text{DW}})_{(4,1)} + 5\,(\lambda_{\text{DW}})_{(5,5)} + 1\,(\lambda_{\text{DW}})_{(2,1)} + 4\,(\lambda_{\text{DW}})_{(3,4)} \geq 4.
\end{aligned}$$

Next, we solve the master problem for node 3, which gives the solution $(\lambda_{\text{DW}})_{(4,1)} = 0.16$, $(\lambda_{\text{DW}})_{(2,1)} = 0.17$, $(\lambda_{\text{DW}})_{(3,4)} = 0.67$, and $x_{\text{DW}} = (3, 3)$. This solution is depicted in Figure 3.1(c). Since this solution is integer feasible, it now gives a global upper bound of 3. Since the global lower bound is also 3, we are done. ■

Branching for Relax-and-Cut We now address the additional considerations when using this idea in the context of relax-and-cut. As mentioned in Section 2.1.3, one way to solve the master problem in the Lagrangian Method is to use the subgradient method. Standard versions of the

subgradient method provide only a dual solution $(u_{\text{LR}}, \alpha_{\text{LR}})$, which is sufficient when using this method for bounding. As shown in Section 2.2.2, we can also apply cutting planes by separating the solution $s \in \mathcal{F}'$ to the Lagrangian subproblem. However, in order to use the branching framework we mention above, we need to be able to map back to the compact formulation so that we can construct dichotomies based on the bounds in the compact space. If this can be accomplished, then the rest of the machinery follows.

Like the case for branch-and-price, the majority of literature on the topic of branching in the context of the Lagrangian Method uses application-specific information. Some authors have suggested the following approach [27]. Let $\mathcal{B}$ define the set of extreme points that have been found in Step 2 of Figure 2.8. After the method has converged and the bound z_{LD} has been found, take the set of extreme points $\mathcal{B}$ and form the inner approximation $\mathcal{P}_I$ as we do in the context of the Dantzig-Wolfe method. That is,

$$\mathcal{P}_I = \left\{ x \in \mathbb{R}^n \;\middle|\; x = \sum_{s \in \mathcal{B}} s\lambda_s, \sum_{s \in \mathcal{B}} \lambda_s = 1, \lambda_s \geq 0 \; \forall s \in \mathcal{B} \right\}. \tag{3.1}$$

Now, construct a linear program analogous to the Dantzig-Wolfe restricted master formulation using this set of extreme points as the inner approximation as follows:

$$\min_{\lambda \in \mathbb{R}^{\mathcal{B}}_{+}} \left\{ c^\top \left(\sum_{s \in \mathcal{B}} s\lambda_s \right) \;\middle|\; A'' \left(\sum_{s \in \mathcal{B}} s\lambda_s \right) \geq b'', \sum_{s \in \mathcal{B}} \lambda_s = 1 \right\}. \tag{3.2}$$

Solve this linear program, giving an optimal primal solution $\hat{\lambda}$. Now, from this we can go back to our mapping (2.18) to project to the compact space and construct the point $\hat{x}$. This idea of using the generated set of extreme points to construct a primal solution has close ties to several other important areas of research, including *bundle methods* [18] and the *volume algorithm* [7].

Now, just like branching for price-and-cut, once the branching constraints have been constructed, we enforce them by adding them directly to $[A'', b'']$. For relax-and-cut, these constraints are immediately relaxed and new dual multipliers are added the Lagrangian master problem, as described in Steps 2 and 3(b) of the relax-and-cut procedure in Figure 2.16. As before, we have the choice whether to enforce the branching constraint in the master problem or the subproblem.

3.2 Relaxation Separability

One of the key motivators of the original Dantzig-Wolfe method [23] was the potential separability of the subproblem. It is often the case that once the side constraints $[A'', b'']$ are relaxed, the constraint system $[A', b']$ becomes separable. Assume that the constraints can be broken up into a set $K = \{1, \ldots, \kappa\}$ of independent blocks. This type of problem, often referred to as *block-diagonal*, has the following form:

$$\begin{array}{llllll} \min c_1^\top x_1 & + & c_2^\top x_2 & + & \ldots & + c_\kappa^\top x_\kappa, \\ A'_1 x_1 & & & & & \geq b_1, \\ & & A'_2 x_2 & & & \geq b_2, \\ & & & & \ddots & \geq \vdots \\ & & & & A'_\kappa x_\kappa & \geq b_\kappa, \\ A''_1 x_1 & + & A''_2 x_2 & + & \ldots \quad A''_\kappa x_\kappa & \geq b'', \\ x_1 \in \mathbb{Z}_+^{n_1}, & & x_2 \in \mathbb{Z}_+^{n_2}, & & \ldots, \quad x_\kappa \in \mathbb{Z}_+^{n_\kappa}. & \end{array}$$

As before, we choose $[A', b']$ as the relaxation from which we generate extreme points to approximate $\mathcal{P}'$ implicitly. From the structure of the above matrix, it should be clear that once $[A'', b'']$ is removed, the constraint system is separable, giving k independent subproblems. We can define a projection in which all the variables *not in* block k are fixed to 0 as follows:

$$\mathcal{P}'_k = \text{conv}\{x \in \mathbb{Z}^n \mid A'_k x \geq b'_k, x_i = 0 \; \forall i \in K \setminus \{k\}\}, \tag{3.3}$$

and let $\mathcal{E}^k$ represent the set of extreme points of $\mathcal{P}_k$. This means that, when solving the subproblem OPT($\mathcal{P}', c$), one can instead solve OPT($\mathcal{P}'_k, c$) for each k and return a set of extreme points as candidates for entering the master problem. If we then sum these projected extreme points, we are once again in the original space. This relationship can be seen easily by simply generalizing the

mapping (2.7) between the original compact formulation and the Dantzig-Wolfe reformulation as follows:

$$\mathcal{P}' = \left\{ x \in \mathbb{R}^n \;\middle|\; x = \sum_{k \in K} \sum_{s \in \mathcal{E}^k} s\lambda_s^k, \sum_{k \in K} \sum_{s \in \mathcal{E}^k} \lambda_s^k = 1 \; \forall k \in K, \lambda_s^k \geq 0 \; \forall k \in K, \; s \in \mathcal{E}^k \right\}. \tag{3.4}$$

Therefore, we can now write the Dantzig-Wolfe bound as follows:

$$z_{\text{DW}} = \min_{\lambda} \left\{ c^\top \left(\sum_{k \in K} \sum_{s \in \mathcal{E}^k} s\lambda_s^k \right) \;\middle|\; A'' \left(\sum_{k \in K} \sum_{s \in \mathcal{E}^k} s\lambda_s^k \right) \geq b'', \sum_{s \in \mathcal{E}^k} \lambda_s^k = 1 \; \forall k \in K \right\}. \tag{3.5}$$

The independence of the optimization subproblems lends itself nicely to parallel implementation. By processing the blocks simultaneously we can greatly improve the overall performance of these applications. This is an important area of future research that we address in Section 6.

Let us return to our second example, the *Generalized Assignment Problem.*

Example 2: GAP (continued) It should be clear that when choosing the capacity constraints (1.24) as the relaxation, we are left with a set of independent knapsack problems, one for each machine in M. Therefore, we can define the projected polytopes for each *block* as follows:

$$\mathcal{P}'_k = \text{conv}\ \{x_{ij} \in \mathbb{R}^+ \; \forall i, j \in M \times N | \; x \text{ satisfies (1.24) and (1.26)}, x_{ij} = 0 \; \forall i \in M \setminus \{k\}, j \in N\},$$

$$\mathcal{Q}'' = \{x_{ij} \in \mathbb{R}^+ \; \forall i, j \in M \times N | \; x \text{ satisfies (1.25)}\}.$$

For clarity, let us consider an example with $n = 3$ tasks and $m = 2$ machines. The formulation

looks as follows:

$$\begin{array}{lllllllll}
\min x_{11} & + x_{12} & + x_{13} & + x_{21} & + x_{22} & + x_{23}, & & & \\
x_{11} & + x_{12} & + x_{13} & & & & \leq b_1, & & (3.6) \\
 & & & x_{21} & + x_{22} & + x_{23} & \leq b_2, & & (3.7) \\
x_{11} & + & & + x_{21} & & & = 1, & & (3.8) \\
 & x_{12} & + & & x_{22} & & = 1, & & (3.9) \\
 & & x_{13} & + & & x_{23} & = 1, & & (3.10) \\
x_{11}, & x_{12}, & x_{13}, & x_{21}, & x_{22}, & x_{23} & \in \{0,1\}. & &
\end{array}$$

The polyhedron $\mathcal{Q}''$ is formed from constraints (3.8)-(3.10), while the polyhedron $\mathcal{P}'$ is formed from (3.6)-(3.7). From the structure of the matrix, it is clear that the two capacity constraints are separable, forming two independent blocks. ■

3.2.1 Identical Subproblems

One of the motivations for using inner methods based on Dantzig-Wolfe reformulation is to help eliminate *symmetry* present in the compact formulation. Symmetry occurs when a model admits large classes of equivalent solutions obtained by permutation of the variables [72]. To make this more concrete, let us revisit the *Vehicle Routing Problem* described in Example 4 in Section 2.3.1.

Example 4: VRP (continued) An alternative formulation for the VRP that is commonly used is to set up the problem as a multi-commodity network flow problem. Once again, consider a complete graph, this time a directed graph G with node set N and arc set A. Let $V = \{1, \ldots, n\}$ be the set of customers and let the depot be represented by vertex 0 (the starting depot) and vertex $n+1$ (the ending depot). Therefore, the node set $N = V \cup \{0, n+1\}$. Associated with each customer $i \in V$ is a demand d_i. The cost of travel from customer i to j is denoted c_{ij}. Let $M = \{1, \ldots, m\}$ be the fleet of vehicles, each having capacity C. Define a binary variable x_{ijk} for each arc $(i, j) \in A$ and

each vehicle k. If vehicle k traverses arc (i,j), then $x_{ijk} = 1$; otherwise, $x_{ijk} = 0$. Now we obtain the following alternative formulation for VRP:

$$\min \sum_{k\in M} \sum_{(i,j)\in A} c_{ij}x_{ijk},$$

$$\sum_{k\in M}\sum_{j\in N} x_{ijk} = 1 \quad \forall i \in V, \tag{3.11}$$

$$\sum_{k\in V}\sum_{j\in N} d_i x_{ijk} \leq C \quad \forall k \in M, \tag{3.12}$$

$$\sum_{j\in N} x_{0jk} = 1 \quad \forall k \in M, \tag{3.13}$$

$$\sum_{i\in N} x_{ihk} - \sum_{j\in N} x_{hjk} = 0 \quad \forall h \in V, k \in M, \tag{3.14}$$

$$\sum_{i\in N} x_{i,n+1,k} = 1 \quad \forall k \in M, \tag{3.15}$$

$$x_{ijk} \in \{0,1\} \qquad \forall (i,j) \in A, k \in M. \tag{3.16}$$

In this formulation, equations (3.11) ensure that each customer is visited exactly once. These are referred to as the *assignment constraints*. Equations (3.13)-(3.15) force each vehicle to leave the depot 0 and return to the depot $n+1$ while satisfying flow balance; i.e., after a vehicle arrives at a customer, it must then depart for another destination.

It should be clear that this formulation has a great deal of symmetry due to the fact that the vehicle fleet is homogeneous. In fact, the index given to a particular vehicle is an arbitrary label, and these can be permuted without affecting the model. For any given solution, it is clear that we can swap the vehicles assigned to a given pair of routes and obtain a new solution with the same cost. ■

The kind of symmetry described in the VRP example can negatively affect the performance of branch-and-bound algorithms. Breaking this symmetry through reformulation or algorithmic techniques is an important and active area of current research [72]. One popular way to do this is by using the Dantzig-Wolfe reformulation. As described in Section 3.2, we often choose a relaxation that is separable, which leads to the reformulation (3.4). A special case, called *identical subproblems*

occurs when $\mathcal{P}'_a \cong \mathcal{P}'_b$ for all pairs $a, b \in K \times K$. This occurs when the subproblems formed by each separable block are structurally identical. This is often the case for models where symmetry is inherent.

Since this is a special case of the block-diagonal structure, we can, of course, apply the aforementioned algorithms directly. However, this would suffer from the same symmetry issues present in the original formulation. To counteract this, we introduce a new variable y_s, for each extreme point $s \in \mathcal{E}$, that will represent an aggregation across all blocks, as follows:

$$\Lambda_s = \sum_{k \in K} s\lambda_s^k \; \forall s \in \mathcal{E}. \tag{3.17}$$

Now, substituting into the Dantzig-Wolfe formulation (2.10) we can now redefine the Dantzig-Wolfe bound in the aggregated space as follows:

$$z_{\text{DW}} = \min_{\Lambda} \left\{ c^\top (s\Lambda_s) \;\middle|\; A'' \left(\sum_{s \in \mathcal{E}} s\Lambda_s \right) \geq b'', \sum_{s \in \mathcal{E}} \Lambda_s^k = K \right\}. \tag{3.18}$$

To illustrate this case, let us return to the VRP example.

Example 4: VRP (Continued) A common decomposition for the multi-commodity flow formulation of VRP is to use the assignment constraints in the master formulation and use the capacitated flow structure as the relaxation, as follows:

$$\begin{aligned} \mathcal{P}'_k &= \text{conv } \{x_{ijk} \in \mathbb{Z}^{A \times M} \mid x \text{ satisfies } (3.12) - (3.16)\} \forall k \in M, \\ \mathcal{Q}'' &= \{x_{ijk} \in \mathbb{R}_+^{A \times M} \mid x \text{ satisfies } (3.11)\}. \end{aligned}$$

The subproblem defined by $\mathcal{P}'_k$ is known as the *Elementary Shortest Path Problem with Resource Constraints* (ESPPRC). Since the vehicle fleet is homogeneous, this leads to identical instances of the ESPPRC subproblem. We can now apply the aggregate reformulation (3.18), which will break the symmetry. Notice, in the case of VRP, the coefficients of the master reformulation are all in $\{0,1\}$. Because of this, the aggregate convexity constraint in 3.18 is redundant and can be dropped. This leads to the commonly referenced *set partitioning formulation* for VRP [35].

Because the special case of identical subproblems is encountered quite often in practice, we feel it is important to try and understand this case in the context of our framework. Unfortunately, the end result of the aggregation reformulation does not directly fit into our framework. This is because of our dependence on the mapping (2.18) between the compact space x and the extended space λ. Once the variable Λ has been introduced, the mapping between the two spaces is no longer one-to-one. This breaks our ability to perform cut generation and branching in the compact space in a straightforward way. For branch-and-price, a method for generic branching that deals with this special case is currently being investigated by Vanderbeck, et al. [90]. The idea is based on performing a disaggregation step to go from Λ_s to λ_s^k. Due to the fact that this mapping is not unique, this has to be done carefully, using lexicographic ordering of the columns. Once this is done, we can then apply the original mapping back to the compact space and proceed normally.

3.2.2 Price-and-Branch

A not very well-known but still quite effective technique to generate good solutions quickly when using Dantzig-Wolfe decomposition on integer programs is to enforce integrality when solving the restricted master problem (2.12). This can be done once when no new columns with negative reduced cost have been found. This idea, called *price-and-branch* was used by Barahona and Jensen to solve a plant location problem in [8]. Price-and-branch is, of course, a heuristic, since there is no guarantee that one can even find a feasible solution to this integer program. Moreover, the dual information from the continuous restricted master LP gives no information about whether or not additional entering columns are needed to prove optimality to the original integer program. Nonetheless, as shown in [8], this can be a very effective heuristic if one is producing good candidate columns during the pricing phase. In addition, this algorithm is attractive from the standpoint of simplicity in implementation.

In this research, we propose to extend the simple idea of price-and-branch to each node of the branch-and-bound tree. That is, in the context of Dantzig-Wolfe decomposition or price-and-cut, after completing the bounding routine for each node, we restrict the current set of variables λ to integer values, and solve the resulting integer program. From our computational experiments, we

have seen that the time to solve this integer program at each node is relatively small. The reason is twofold. First, the number of columns needed to obtain the Dantzig-Wolfe bound is generally small. Hence, the resulting integer program is fairly easy to solve. Second, we are careful about keeping the size of the master problem small by actively compressing its size by removing non-binding cuts and non-active columns. This idea is explored further in Section 3.6.

This technique of extending price-and-branch to each node of the tree is a cheap heuristic that often does a very good job of producing incumbents and results in a more efficient overall algorithm. In Sections 5.2 and 5.3, we show computational results of using this idea on two different applications.

3.3 Nested Pricing

In this section, we describe an idea that can help improve the overall performance of integrated methods when embedded in a branch-and-bound framework. Although much of the focus of this research is on the generation of strong lower bounds for z_{IP}, the performance of branch-and-bound algorithms also depends highly on the ability to find feasible solutions, yielding upper bounds. Moreover, in practice, users of optimization solvers often wish to find good feasible solutions in some fixed amount of time. We can tell if a feasible solution is of high quality by looking at the gap between the lower and upper bounds on z_{IP}.

As described in Chapter 2, integrated decomposition methods improve upon traditional methods by implicitly building an inner *and* an outer approximation. By intersecting these two polyhedra, one can find better approximations of the convex hull of the original problem. In Section 2.3.1, we discussed the template paradigm and the idea of a polytope defined by some class of valid inequalities. Generally, one thinks of the collection of valid inequalities as *the* outer approximation generating during the cutting plane method. However, one could also view the polytopes defined by each class of inequalities separately. Then, taking the intersection of all of these polytopes, one generates the outer approximation of interest. Computationally, it is well known that generating cuts from various classes of inequalities can be beneficial. Common intuition is that different classes of cuts contribute to describing different areas of the polyhedral region; or, in the context of the

application, different cuts correspond to different aspects of the constraint system.

In fact, many difficult integer programs arise as the combination of two (or more) integer programs defined over the same set of variables. For example, the classical Vehicle Routing Problem is often seen as the intersection between a routing problem (TSP) and a packing problem (bin-packing). Each problem, by itself, is relatively easy to solve, while the combination is often extremely difficult. This line of thinking is the motivating factor behind the idea presented below.

In every study we are aware of, inner approximation methods generate an approximation based on *one* fixed relaxation (or polytope). If one considers inner approximations simply as the dual of outer approximations, then this motivates us to consider generating extreme points from multiple polyhedra, similar to how facets are generated from multiple polyhedra in outer methods. In fact, any polyhedron $\mathcal{P}'_N \subseteq \mathcal{P}'$ is a candidate for inclusion in the pricing phase. For example, in Step 3 of the Dantzig-Wolfe method, when calling the subroutine

$$\text{OPT}\left(\mathcal{P}', c^\top - \left(u^t_{\text{DW}}\right)^\top A'', \alpha^t_{\text{DW}}\right),$$

we also call

$$\text{OPT}\left(\mathcal{P}'_N, c^\top - \left(u^t_{\text{DW}}\right)^\top A'', \alpha^t_{\text{DW}}\right),$$

producing another extreme point. It appears that this simple but intriguing idea has been overlooked until now. To illustrate this idea, let us again revisit the *Vehicle Routing Problem*.

Example 4: VRP (continued) Recall the formulation for VRP described in Section 2.3.1. Let's assume we choose the perfect b-matching relaxation for our decomposition. Then, the relaxed polyhedron $\mathcal{P}'$ is defined by the constraints (2.35), which enforce that each customer must be serviced by exactly one vehicle, and (2.34), which enforce that each of the k vehicles must depart and return to the depot exactly once. Now, let us also consider again the multiple traveling salesman relaxation. Feasible solutions to this relaxation also satisfy the same constraints as in the case of b-matching. In addition, solutions to k-TSP also satisfy the subtour elimination constraints (2.40). The two nested

relaxations are defined as follows:

$$\mathcal{P}^{\text{bMatch}} = \text{conv}\{x \in \mathbb{R}^E \mid x \text{ satisfies } (2.34), (2.35), (2.37), (2.38)\},$$
$$\mathcal{P}^{\text{kTSP}} = \text{conv}\{x \in \mathbb{R}^E \mid x \text{ satisfies } (2.34), (2.35), (2.37), (2.38), (2.40)\}.$$

where $\mathcal{P}^{\text{bMatch}} \supset \mathcal{P}^{\text{kTSP}}$. So, by choosing $\mathcal{P}' = \mathcal{P}^{\text{bMatch}}$, we can generate extreme points from polytopes. As usual, in order to attain valid bounds, we have to solve the optimization subproblem for $\mathcal{P}^{\text{bMatch}}$ exactly. Then, since $\mathcal{P}^{\text{kTSP}}$ is contained in $\mathcal{P}^{\text{bMatch}}$, we can solve the optimization subproblem for $\mathcal{P}^{\text{bMatch}}$ heuristically.

Let us now return to our motivating principle—using different classes of valid inequalities when generating cutting planes. Analogously, it would be beneficial if the extreme points used for the inner approximation had some diversity with respect to which parts of the feasible region are satisfied. To this point, let us now consider an additional relaxation to use in our nested pricing scheme for VRP. Recall that (2.36), the *generalized subtour elimination constraints* (GSECs), have exponential size. In the standard cutting plane algorithm for VRP, these GSECs would be generated dynamically as needed. Assume we have run several iterations of a cutting plane method and define $\mathcal{G}$ as the set of subsets of nodes that represent the GSECs that were found. Now, define the following class of inequalities, a subset of the full set of GSECs:

$$x(\delta(S)) \geq 2b(S) \; \forall S \in \mathcal{G}. \tag{3.19}$$

Now, combining these inequalities with (2.34) and (2.35), let us define a new relaxation

$$\mathcal{P}^{\text{bMatch+}} = \text{conv}\{x \in \mathbb{R}^E \mid x \text{ satisfies } (2.34), (2.35), (3.19), (2.37), (2.38)\},$$

which now incorporates some of the capacity restrictions from the original problem. Now, since $\mathcal{P}^{\text{bMatch}} \supset \mathcal{P}^{\text{bMatch+}}$, we can use all three polyhedra, $\mathcal{P}^{\text{bMatch}}$, $\mathcal{P}^{\text{bMatch+}}$, and $\mathcal{P}^{\text{kTSP}}$ when generating extreme points, thereby improving our diversity and increasing our chances of finding feasible solutions quickly. ■

One of the key drawbacks for methods like price-and-cut is that, in order to derive a valid bound, one must *price out* exactly over the chosen inner relaxation $\mathcal{P}'$. Although this is still true, with this idea, one can also generate points from tighter relaxations heuristically. Since these extreme points are in some sense *closer* to feasible for the original problem, the probability of finding upper bounds earlier in the search tree is higher. Typically pricing over more restrictive polytopes can be more expensive computationally. So, there is a tradeoff between the time spent calculating the extreme points and the improvement gained by better upper bounds on z_{IP}. In Section 5.1 we will present computational results to show the effectiveness of this idea.

3.4 Initial Columns

As discussed in Section 2.1.2, the generation of the starting set of extreme points for $\mathcal{P}_I^0$ is theoretically arbitrary. However, computationally, it can have a big impact on performance. For specific applications, the best way to seed the set of columns is to run a heuristic known for that application. Any feasible solution to the original problem can serve as a starting column, but for some applications, finding a feasible solution is non-trivial. Ideally, we would like to have some generic way to overcome this issue.

In the absence of any application-specific knowledge, we can still start this process by generating initial columns as solutions to the subproblem using any arbitrary cost vector. We can use this idea as the default way to generate initial columns. We make a call to the solver for the subproblem (an MILP solver by default) with a cost vector c equal to the original cost vector. This tends to give high-quality columns that can be used to start the process. Of course, starting off with many columns can provide a better chance of finding an initial feasible solution. So, in addition to solving OPT$(\mathcal{P}', c)$, we also solve the relaxation for several random perturbations of the original cost vector.

Another way to seed the master columns is to first run several steps of the Lagrangian method, collecting those extreme points found at each master step. This idea was proposed by Barahona, et al. in [8].

A third way to generate initial columns is to use the decompose-and-cut procedure described in

Section 2.3. The original idea of *decompose-and-cut* was to take advantage of the fact that, computationally, separating structured points is easier than separating unstructured points. This can serve as a heuristic for solving difficult separation problems in the context of the standard cutting plane method. As discussed in Section 2.3.3, an additional benefit of the method is the potential for finding these non-templated (or decomposition) cuts, which can help improve the bounds as it pushes the point into the space of the inner relaxation polyhedron while still in the context of traditional cutting plane methods. Another advantage of this idea is its potential to be used to seed the initial columns of a Dantzig-Wolfe formulation. At each step of the decompose-and-cut algorithm, we are generating extreme points of some relaxation. We can store each of these columns and use this to seed price-and-cut. In addition, if the decomposition is successful, this then guarantees us that we can start out price-and-cut with a feasible basis.

3.5 Standard MILP Cutting Planes for Inner Methods

Generic cutting planes have long been one of the most important factors in the success of branch-and-cut codes. In the past decade, there have been major advances in increasing the variety of valid inequalities cuts that can be generated and used to solve generic MILPs [13]. Up until now, the use of cutting planes with inner methods like price-and-cut has been limited to application-specific cuts. When application-specific cuts are known and provably facet-defining, it is common for these cuts to dominate generic classes in terms of strength. However, it is quite common in real-world applications that the user has no knowledge of any specialized cuts that can improve the model. Also, although a specific class of cut might be facet-defining, an efficient separation algorithm may not be known and one must rely on heuristics that give no guarantee of quality. In such cases, generic cuts may be extremely beneficial and necessary to solve a given problem. This fact has long been understood with respect to branch-and-cut codes, and all solvers include at least a basic implementation of separation algorithms for the most common classes of valid inequalities. However, until now, there have been no branch-and-price-and-cut frameworks that have successfully allowed for direct integration of generic cutting planes. In the framework we propose, this integration is possible by taking advantage of the same mapping between the compact and extended formulations that we

have mentioned earlier. We explore this idea further in Section 4.2, after introducing our software framework.

3.6 Compression of Master LP and Object Pools

A well-known idea in the context of branch-and-cut is to remove *non-binding cuts* when they have been deemed *ineffective*. This idea is described in [73] in the context of solving large-scale TSPs. A non-binding cut has an associated dual variable that currently has value zero. By removing this cut, the optimality of the current solution is unchanged. Therefore, removing the cut from the master LP can improve efficiency by reducing the size of the LP basis and decreasing the overall processing done by subsequent iterations. In [73], along with this idea of removing non-binding cuts, Padberg and Rinaldi also suggest the idea of using *cut pools* to store the discarded cuts for use later in the algorithm. Since these cuts could once again become violated, or could be useful in some other part of the tree, it makes sense to store them and simply check them for violation rather than regenerating them using an expensive separation routine.

An analogous idea can be applied in the context of branch-and-price-and-cut. A variable is called *inactive* if it is non-basic and has positive reduced cost. When a variable is deemed inactive, it may be removed without affecting the optimality of the current solution. Like removal of cuts, by compressing the size of the master LP we can improve the efficiency of the solution update step. In the same way we can store discarded cuts in an associated cut pool, we also store discarded variables in an associated variable pool. This can have an even bigger impact on performance since the generation of variables can often be relatively expensive. By simply checking the variable pool for columns with negative reduced cost we can save a good deal of processing time.

3.7 Solver Choice for Master Problem

Another important algorithmic choice in the implementation of a branch-and-price-and-cut method is how one solves the master problem (an LP). It is well-known that the *dual simplex* method generally performs best in the context of branch-and-cut. This is because upon adding valid inequalities

(rows) to the master LP, or adjusting bounds, the current solution remains dual feasible. Adding a cut to the (primal) master LP is equivalent to adding a column to its dual, while adjusting a variable bound is equivalent to changing the dual cost vector. Neither of these things affects the feasibility of the current dual solution. Because of this, the *dual simplex* method can skip its *first phase*, which attempts to find a dual feasible basis, and use the previous basis to *warm-start* the algorithm.

In the context of branch-and-price-and-cut, we may add either rows or columns or both at each iteration. Adding a column to the master LP is equivalent to adding a row to its dual. This row, because of how we select the columns to enter, will cause the current point to be dual infeasible. However, it is still primal feasible. For this reason, the *primal simplex* method makes more sense since its *first phase* is to find a primal feasible basis. So, it seems appropriate to use primal simplex after adding columns and dual simplex after adding cuts or adjusting bounds.

Of course, the overall algorithmic efficiency of the primal simplex method versus the dual simplex method is also a consideration in this decision. Despite being able to start with a feasible basis, the primal simplex is in general considered less efficient than dual simplex for solving many of the linear programs encountered in practice. So, there is a tradeoff to be considered that can only be studied empirically. In Chapter 5 we will present some computational results comparing these options.

A third option for solving the master LP is to use an interior point method (IPM). It is fairly well known that there are sometimes performance issues for inner methods like Dantzig-Wolfe, when using simplex-based methods for solving the master LP, due to convergence. The slow convergence of these methods has been attributed to the nature of the development of the optimal dual values [55]. Since simplex-based methods provide extremal points, we often encounter oscillation in the dual variables, which can greatly hurt performance. There have been several attempts at dealing with this issue by penalizing large movements in the dual solution from some *stability center*. For an excellent treatment of these methods, see [16]. A simpler approach to dealing with the stability issue is to use an interior point method to solve the master LP. Since IPMs give solutions in the interior of the feasible region as opposed to extremal solutions, the oscillations can be greatly reduced. Of course, like the choice of primal versus dual simplex, there are other important computational

considerations if choosing to use an interior point method over a simplex-based method. Although there is some recent work on *warm-starting* IPMs [95], it is not yet well understood how an IPM would perform in the context of a branch-and-price-and-cut framework. Integration of some of these stabilization ideas into our framework is an important area of future research.

Chapter 4

DIP Software

The theoretical and algorithmic framework that we propose in Chapter 2 lends itself nicely to a wide-ranging and flexible generic software framework. In this chapter, we introduce a new open-source C++ software framework called DIP (**D**ecomposition for **I**nteger **P**rogramming). DIP is a flexible generic software framework that provides extensible implementations of the various decomposition algorithms proposed in Chapter 2. The typical user of DIP will have access to the full suite of decomposition algorithms by providing methods that define application-specific components. In addition, the advanced user has the ability to override almost any algorithmic component, allowing them to develop their own variants of the aforementioned methods.

The software frameworks for use in the area of decomposition methods are primarily designed to be flexible, leaving it up to the user to implement various algorithms as they pertain to their specific application. With each variant of the chosen algorithm comes the burden of implementing the components specific to their application. DIP takes a much different approach, sacrificing some of the extensive flexibility of a framework like COIN/BCP in order to provide a great deal more automation for the user in order to reduce their burden. Due to this, the learning curve is much less steep using DIP than with a framework like COIN/BCP.

One of the key contributions of this research is the manifestation of the conceptual framework discussed in Chapter 2 in the form of DIP. By considering inner methods, such as Dantzig-Wolfe

decomposition and Lagrangian relaxation, in the same framework as classical cutting plane methods, we allow for seamless integration. The mapping (2.18) between the compact formulation and the extended formulation, which was discussed in Section 2.1.2, is key to the success of this design. This allows the framework to manage the integration of dynamically generated cuts and variables simultaneously. In addition, as discussed in Section 3.1, this also greatly simplifies the setup for branching, as the rules can be defined in the compact space. These two facts allow for a much simpler user interface than any of the previously mentioned frameworks. For example, in COIN/BCP, to implement a column generation algorithm scheme, the user must define the pricing subproblem, provide a solver for it, and then also provide a routine to produce a new column for inclusion in the reformulated master. The subproblem is generally expressed in the original compact space, so the user has the burden of providing the method for applying the mapping in both directions. The user must also define the objective for the subproblem by providing a function that computes the reduced cost vector from the master formulation's dual solution.

When simultaneously generating valid inequalities, things get even more complex. The translation from columns in the compact space to rows in the reformulated space must be done carefully, taking into account any cuts that have been added or deleted. The same applies to branching, for which a separate routine is required to enforce branching rules. In COIN/BCP, there are no built-in facilities for this. DIP does all of this in an automated fashion for the user. The user has simply to describe the model in the compact space and identify the relaxation (subproblem) that defines the decomposition.

If the user wishes to add methods for generating problem-specific valid inequalities, these methods can be defined in terms of the compact model. If a function to perform the separation in the compact space is provided, then DIP takes care of all of the accounting necessary for implementing an integrated method such as branch-and-price-and-cut. In addition, because of its design, one can even employ separation routines for generic classes of valid inequalities for MILP, such as those provided by COIN/CGL (Cut Generator Library) [54]. The same cuts that help improve the performance of outer methods, such as branch-and-cut, can be applied in combination with inner methods using DIP.

An overarching motivation behind the design of this framework is that a user can describe a model in the context of the compact space and with a simple option run any combination of the traditional or integrated methods discussed above. This means that with no additional user interaction, it is possible to compare cutting plane methods with price-and-cut or relax-and-cut in the same software framework. To our knowledge, this is the first time such a framework has been provided.

4.1 Design

DIP is a C++ library that relies on inheritance to define specific customizations. The user interface is divided into two separate classes, an *applications interface*, encapsulated in the class `DecompApp`, in which the user may provide implementations of problem-specific methods (e.g., solvers for the subproblems), and an *algorithms interface*, encapsulated in the class `DecompAlgo`, in which the user can modify DIP's internal algorithms, if desired. The `DecompAlgo` class provides implementations of all of the methods described in Chapter 2. As mentioned before, an important feature of DIP is that the problem is always presented, by the user, in the compact space, rather than in the space of a particular reformulation. The user may provide subroutines for separation and optimization in the original space without considering the underlying algorithmic method. DIP performs all of the necessary bookkeeping tasks, including automatic reformulation for price-and-cut constraint dualization for relax-and-cut, cut and variable pool management, and row and column expansion.

DIP has been released as part of the COIN-OR project. It is built on top of several other COIN-OR projects and relies on others for some of its default implementations. Since we will be mentioning several of these projects in the following sections, we include here a reference table of those projects that are used by DIP. One of those projects, ALPS (Abstract Library for Parallel Search), serves as the base foundation upon which DIP is built. ALPS is the search-handling layer of the COIN-OR High-Performance Parallel Search (CHiPPS) framework [94]. In ALPS, there is no assumption about the algorithm that the user wishes to implement, except that it is based on a tree search. Since our decomposition-based algorithms are focused on the resolution of strong bounds for use in a branch-and-bound framework, the ALPS framework was a perfect candidate to build upon. To facilitate this, DIP provides an interface object `AlpsDecompModel` that is derived from

Project Name	Description
ALPS	The Abstract Library for Parallel Search, an abstract base layer for implementations of various tree search algorithms
CBC	COIN-OR branch-and-cut, an LP-based branch-and-cut solver for MILP
CGL	Cut Generator Library, a library of cutting-plane generators
CLP	COIN-OR LP, a linear programming solver that includes simplex and interior point solvers
CoinUtils	Utilities, data structures, and linear algebra methods for COIN-OR projects
OSI	Open Solver Interface, a uniform API for calling embedded linear and mixed integer programming solvers

Table 4.1: COIN-OR Projects used by DIP

Class	Description
`DecompApp`	Interface for setting the model and application-specific methods
`DecompAlgo`	Interface for defining algorithmic components
`AlpsDecompModel`	Interface to the Alps tree search framework
`DecompConstraintSet`	Class for defining the model(s)
`DecompVar`	Class for storing an extreme point of $\mathcal{P}'$
`DecompCut`	Class for storing a valid inequality

Table 4.2: Basic Classes in DIP Interfaces

the base class `AlpsModel`.

In the following sections, we discuss the DIP interface design at a more abstract level. Then, in section 4.4, we provide several examples to make these concepts more concrete. In Table 4.1 we list the basic set of classes that a typical user would encounter when creating an application in DIP.

4.1.1 The Application Interface

The base class `DecompApp` provides an interface for the user to define the application-specific components of their model and algorithm. There are two main steps to creating an application. The first step is to define the model. The second step is to define any application-specific algorithmic components.

The user must first provide the relevant input data, as discussed in Section 1.2, i.e., the description of the polyhedron $\mathcal{Q}''$ and (optionally) an implicit description of $\mathcal{P}'$ or an explicit description

of the polyhedron $\mathcal{Q}'$, for the chosen relaxation. The framework expects the user to define these models in the form of the object type `DecompConstraintSet`. The API for this object is as one might expect, where the user must define the constraint matrix, row and column bounds, and variable types, such as continuous or integral. Once the model has been defined, the user must set these models using the following methods of the class `DecompApp`:

- `setModelObjective(double * c)`, for the original objective c, and
- `setModelCore(DecompConstraintSet * model)`, for $\mathcal{Q}''$.

Optionally, if the chosen relaxation $\mathcal{Q}'$ is to be defined explicitly as an MILP, the user would need to call the method `setModelRelax(DecompConstraintSet * model, int block)`. Notice that the block separable case for $\mathcal{Q}'$ is easily accommodated in the method that sets the relaxation. In the case where the extreme points of the relaxation are generated implicitly, such as in Example 3 (TSP), we do not have to explicitly define anything for the relaxation model.

Once the models have been defined, the user has the choice to override a number of different methods that will affect the algorithmic processing. Here we point out some of the most commonly used methods. For a full description, reference the doxygen API documentation provided at [33].

`DecompApp::solveRelaxed()` In this method, the user can provide an algorithm for solving the optimization subproblem OPT$(\mathcal{P}', c)$. In the case of an explicit definition of $\mathcal{Q}'$ as an MILP, the user does not have to override this method—DIP simply uses the built-in MILP solver (CBC) or another OSI-supported MILP solver that has been defined in the setup process. In the case of a problem like TSP or GAP, however, the user might know of specific algorithms for solving the subproblem that are more efficient than using a generic MILP solver. In GAP, for example, each block can be solved using specialized methods for the Binary Knapsack Problem. In this method, the user is given the cost vector to optimize. From this, the user must return a solution or set of solutions in the form of `DecompVar` objects. The solution s, an extreme point of $\mathcal{P}'$, is simply an assignment of the variables in the original space. The `DecompVar` object is simply a sparse vector that represents that solution.

`DecompApp::generateCuts()` In this method, the user can provide an algorithm for solving the separation subproblem SEP($\mathcal{P}', x$). DIP already has generators for most of the common classes of generic valid inequalities provided by CGL built in. For many applications, the user will have specific knowledge of methods for generating valid inequalities that can greatly enhance performance. For example, for the TSP, we have already mentioned subtour elimination constraints (SECs) and comb constraints. For SECs, one approach to solving the separation problem would be to set up the problem as the well-known *Minimum Cut Problem*. The method `generateCuts` provides the solution to the current relaxation, and the user must return a cut or set of cuts that separates this point. The cuts are returned in the form of `DecompCut` objects. Like `DecompVar`, this object is simply a sparse vector representing the coefficients for the cut plus the sense and right hand side. Also, like the variable object, the representation of the cut is in the original space. The user does not have to define the cut in the reformulated space for any particular algorithm. This fact greatly simplifies the burden on the user and allows them to leverage previous work on polyhedral methods.

`DecompApp::isUserFeasible()` This function allows the user to determine whether or not a given solution, $\hat{x}$, is feasible to the original model. In the case where the user gives an explicit definition of $\mathcal{Q}''$, this function is unnecessary, since all constraints are described and can be checked automatically. However, when a full description of $\mathcal{Q}''$ is not given a priori, the user will also have to provide this function to declare if a solution is feasible to the original system. In TSP, for example, when the chosen relaxation is 2-Matching, the SECs are too numerous to define explicitly, so the user must provide this function to tell whether or not a solution forms a tour.

4.1.2 The Algorithm Interface

At a high level, the main loop of the base algorithm provided in `DecompAlgo` follows the paradigm described earlier, alternating between solving a master problem to obtain solution information and solving a subproblem to generate new polyhedral information. Each of the methods described in this thesis has its own interface derived from `DecompAlgo`. For example, the base class for the price-and-cut method is `DecompAlgo::DecompAlgoPC`. In this manner, the user can override a

specific subroutine common to all methods (in `DecompAlgo`) or restrict it to a particular method.

Since the integrated methods described in Section 2.2 generalize the traditional methods from Section 2.1, we are able to cover the spectrum of aforementioned methods with the following four algorithmic classes.

Derived from `DecompAlgo`

- `DecompAlgoC` provides the cutting plane method,
- `DecompAlgoPC` provides the Dantzig-Wolfe method and price-and-cut,
- `DecompAlgoRC` provides the Lagrangian method and relax-and-cut.

Derived from `DecompAlgoPC`

- `DecompAlgoDC` provides the decompose-and-cut method.

Figure 4.1.2 depicts the inheritance diagram for the algorithmic classes. Recall that the core step of the decompose-and-cut method described in Figure 2.20 looks exactly like the steps of the Dantzig-Wolfe method described in Figure 2.4. For this reason, we have derived the decompose-and-cut object directly from the price-and-cut object, which greatly simplified its implementation.

The typical DIP user simply instantiates the `DecompAlgo` objects rather than deriving from them to produce a customized algorithm. The decomposition methods mentioned in Chapter 2 can all be applied to a user's application by simply passing the application object onto the algorithmic object. An example of this is shown in Section 4.3. However, there are many possible variants of the basic algorithms that can be explored by simply overriding certain components of the algorithms. For this reason, all of the methods in the base and derived classes are declared `virtual`, giving the user full flexibility to experiment with different derivations.

4.1.3 Interface with ALPS

Once the user has defined their applications interface `DecompApp` and their algorithmic interface `DecompAlgo`, they must pass this information to an `AlpsDecompModel`. This object is derived

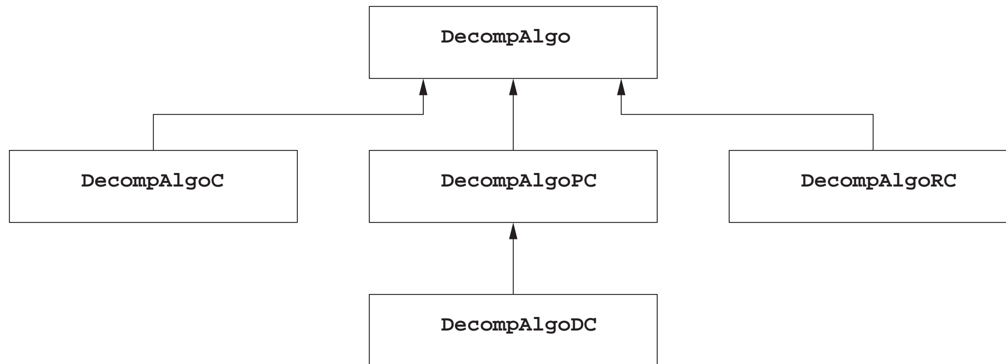


Figure 4.1: Inheritance Diagram for `DecompAlgo`

from the ALPS object, `AlpsModel`, which provides an interface to a basic tree search. It handles the basic operations of the outer loop of a classic branch and bound. That is, defining search tree nodes, calling the bounding methods, pruning based on infeasibility or quality, and farming off work based on node selection rules. In the context of this research, the interaction with ALPS is quite simple because we currently process the tree search in a serial environment. However, ALPS was originally designed to be deployed on distributed architectures, such as clusters or grids, and is targeted at large-scale computing platforms where the number of processors participating in the search can be very large. Using DIP with ALPS in a parallel environment has great potential and is one important area of future research discussed in Section 6.

4.2 Interface with CGL

By using COIN's Cut Generator Library project, DIP has access to all of the well-known separation algorithms for generic classes of valid inequalities typically employed in branch-and-cut code: Knapsack Cover, Flow Cover, Cliques, Mixed-Integer Rounding, and Gomory Mixed Integer. The design of any cut generator should follow the simple paradigm of producing an inequality valid for a given polyhedron that is violated by a given vector. Since DIP always projects back to the compact formulation, the polyhedra, $\mathcal{Q}''$ and (optionally) $\mathcal{Q}'$, are defined in the compact space. Obviously, the vector to be separated must also be given in the compact space. In this manner, implementation for the majority of cuts is relatively straightforward. This simple observation is a major contribution

of this work allowing, for the first time, for direct integration of standard cutting planes into an inner method like branch-and-price-and-cut.

The simple design above should be clear for any generic cutting plane that follows the template paradigm and is independent of the solution technique used for the calculation of $\hat{x}$. For cutting planes like Gomory Mixed Integer, the standard separation algorithm depends on the existence of a basis, resulting from having solved the compact formulation using the simplex method. In the context of branch-and-cut, the master problem *is* the compact formulation and, if solved using simplex, can easily integrate with separation algorithms for Gomory Mixed Integer cuts. In the case of branch-and-price-and-cut, however, the master problem is the extended formulation and we map back to the compact space to compute $\hat{x}$. This gives us a primal solution but not an associated basis. Conceptually, this can be overcome by running a *crossover* step to construct a basis from the primal solution $\hat{x}$. This idea of "crossing over" from a solution to a basic solution is common in the user of hybrid linear programming solvers that start out using an interior point method and then cross over to the simplex method []. Applying this idea in the context of DIP is another potential area of future research.

4.3 Creating an Application

To better understand how a user might interact with the framework, let us return to our examples.

4.3.1 Small Integer Program

The small integer linear program in Example 1 provides the simplest use of the software. In Listing 4.1 we show the driver program for the application SILP [1]. Lines 3–6 use the `UtilParameters` class, a simple utility object for reading parameters from a file or the command line. At line 9, the user declares a `SILP_DecompApp` object that is derived from a `DecompApp` object. This object contains all of the necessary member functions for defining the model and any user-specific overrides as described in Section 4.1. Lines 12–15 define the `DecompAlgo` of interest and take the

[1] For the sake of conserving space, in each of the listings in this chapter, we remove all of the standard error checking, exception handling, logging, and memory deallocation. Therefore, these listings should be considered *code snippets*. For a complete set of source code, please see [33].

```
1  int main(int argc, char ** argv){
2     //create the utility class for parsing parameters
3     UtilParameters utilParam(argc, argv);
4     bool doCut          = utilParam.GetSetting("doCut",       true);
5     bool doPriceCut     = utilParam.GetSetting("doPriceCut",  false);
6     bool doRelaxCut     = utilParam.GetSetting("doRelaxCut",  false);
7
8     //create the user application (a DecompApp)
9     SILP_DecompApp sip(utilParam);
10
11    //create the CPM/PC/RC algorithm objects (a DecompAlgo)
12    DecompAlgo    * algo = NULL;
13    if(doCut)       algo = new DecompAlgoC (&sip, &utilParam);
14    if(doPriceCut) algo = new DecompAlgoPC(&sip, &utilParam);
15    if(doRelaxCut) algo = new DecompAlgoRC(&sip, &utilParam);
16
17    //create the driver AlpsDecomp model
18    AlpsDecompModel alpsModel(utilParam, algo);
19
20    //solve
21    alpsModel.solve();
22 }
```

Listing 4.1: DIP main for SILP example

application and parameter objects as input. At this point, the two main objects (the application and the algorithm) have been constructed and DIP is ready to solve. The last step is to construct an `AlpsDecompModel` from the algorithm and parameter object, which is done at Line 18. Then, at Line 21, we call the solve method. The setup for the main program in this example is almost identical to that for any other application. The details of the implementation are contained in the framework itself and the user derivations.

In Listing 4.2, we take a closer look at what is happening when the user application is constructed at Line 9 of `main`. In this code, we are considered two different possible decompositions for Example 1. The original decomposition described in Section 1.2,

$$\begin{aligned} \mathcal{P}'_1 &= \{x \in \mathbb{Z}^2 \mid x \text{ satisfies } (1.6) - (1.11)\}, \\ \mathcal{Q}''_1 &= \{x \in \mathbb{R}^2 \mid x \text{ satisfies } (1.12) - (1.20)\}, \end{aligned}$$

```
1  void SILP_DecompApp::createModels(){
2     //construct the objective function
3     const int    numCols      = 2;
4     double       objective[2] = {0,1};
5     setModelObjective(objective);
6
7     //build matrix parts 1 and 2
8     const int numNzs1  = 10, numNz2   = 10;
9     const int numRows1 =  6, numRows2 = 5;
10    int        rowIndices1[numNzs1] = {0,0,1,2,2,3,3,4,5,5};
11    int        colIndices1[numNzs1] = {0,1,1,0,1,0,1,1,0,1};
12    double     elements1  [numNzs1] = { 7.0, -1.0,  1.0, -1.0,  1.0,
13                                       -4.0, -1.0, -1.0, 0.2, -1.0};
14    int        rowIndices2[numNzs2] = {0,0,1,1,2,2,3,3,4,4};
15    int        colIndices2[numNzs2] = {0,1,0,1,0,1,0,1,0,1};
16    double     elements2  [numNzs2] = { -1.0, -1.0, -0.4, 1.0,   1.0,
17                                       1.0 ,   3.0,  1.0, 0.25, -1.0};
18
19    m_modelPart1.M = new CoinPackedMatrix(false, rowIndices1, colIndices1,
20                                          elements1, numNzs1);
21    m_modelPart2.M = new CoinPackedMatrix(false, rowIndices2, colIndices2,
22                                          elements2, numNzs2);
23
24    //set the row upper and lower bounds of part 1
25    double   rowLB1[numRows1] = {13.0, 1.0, -3.0, -27.0, -5.0, -4.0};
26    std::fill_n(back_inserter(m_modelPart1.rowUB), numRows1, DecompInf);
27    std::copy  (rowLB1, rowLB1 + numRows1, back_inserter(m_modelPart1.rowLB));
28
29    //set the column upper and lower bounds of part 1
30    std::fill_n(back_inserter(m_modelPart1.colLB), numCols, 0.0);
31    std::fill_n(back_inserter(m_modelPart1.colUB), numCols, 6.0);
32
33    //set the integer variables for part 1
34    m_modelPart1.integerVars.push_back(0);
35    m_modelPart1.integerVars.push_back(1);
36
37    //set the row/col bounds and integer variables for part 2
38    <... snip ... >
39
40    switch(whichRelax){
41    case 1:
42      //set model 1
43      setModelRelax(&m_modelPart1);   //P'
44      setModelCore (&m_modelPart2);   //Q''
45      break;
46    case 2:
47      //set model 2
48      setModelRelax(&m_modelPart2);    //P'
49      setModelCore (&m_modelPart1);    //Q''
50      break;
51    }
52 }
```

Listing 4.2: DIP createModels for SILP example

as well as the opposite construction

$$\mathcal{P}_2' = \{x \in \mathbb{Z}^2 \mid x \text{ satisfies } (1.12) - (1.20)\},$$
$$\mathcal{Q}_2'' = \{x \in \mathbb{R}^2 \mid x \text{ satisfies } (1.6) - (1.11)\}.$$

The data members `m_modelPart1` and `m_modelPart2` are `DecompConstraintSet` objects. They are passed into the applications interface to define the relevant model depending on the decomposition the user is interested in solving. The construction of the model follows the typical steps for creating any MILP model in COIN. At lines 3–5 we define and set the objective function, which is the same for either decomposition. Next, at lines 8–22, we build two matrices using the triple index data structure input for a `CoinPackedMatrix`. A `CoinPackedMatrix` is the standard sparse matrix data structure provided with the CoinUtils project. There are many different ways to construct the input using this object. The interested reader is referred to the CoinUtils API documentation at [49]. Then, at lines 24–38, we define the column and row bounds, as well as a list of those variables that are integral. At this point, we are done defining the necessary model elements and we can simply assign the appropriate data member using the methods `setModelRelax` and `setModelCore`. As can be seen, switching between the choice of relaxations is quite easy.

At this point, the work to create the application SILP is complete. In this case, the methods for solving the subproblems are all built into the framework. The generation of variables is done using the built-in MILP solver (CBC), and the generation of valid inequalities is done by the built-in separation routines (CGL). Checking feasibility is as simple as checking that the complete model constraints and integrality requirements are met. This is also done automatically by the framework. In the following section we look at a slightly more advanced implementation for the Generalized Assignment Problem that requires using a few more of the interface methods defined by `DecompApp`.

4.3.2 Generalized Assignment Problem

Looking back at our second example, the Generalized Assignment Problem, we recall that when choosing the capacity constraints (1.24) as the relaxation, we are left with a set of independent knapsack problems, one for each machine in M.

```
1  void GAP_DecompApp::createModels(){
2
3     //get information about this problem instance
4     int          nTasks     = m_instance.getNTasks();    //n
5     int          nMachines  = m_instance.getNMachines(); //m
6     const int *  profit     = m_instance.getProfit();
7     int          nCols      = nTasks * nMachines;
8
9     //construct the objective function
10    m_objective = new double[nCols];
11    for(i = 0; i < nCols; i++)
12       m_objective[i] = profit[i];
13    setModelObjective(m_objective);
14
15    DecompConstraintSet * modelCore = new DecompConstraintSet();
16    createModelPartAP(modelCore);
17    setModelCore(modelCore);
18
19    for(i = 0; i < nMachines; i++){
20       DecompConstraintSet * modelRelax = new DecompConstraintSet();
21       createModelPartKP(modelRelax, i);
22       setModelRelax(modelRelax, i);
23    }
24 }
```

Listing 4.3: DIP createModels for GAP example

In Listing 4.3 we show how one defines these blocks when setting the model components. At lines 15–17 we create and set the model core $\mathcal{Q}''$ as the set of assignment constraints using the method `setModelCore`. Then, at lines 19–23, we create and set each block using the `setModelRelax` method. Notice the third argument of this method that allows the user to define the appropriate block. For each block defined, DIP knows how to handle the algorithmic accounting necessary for the associated reformulations.

Since each block defines an independent binary knapsack problem, we want to take advantage of this using one of the many well-known algorithms for solving this problem. We could explicitly define the knapsack problem as an MILP when defining the matrix $\mathcal{Q}'$. In that case, DIP would just call the built-in MILP solver when asked to generate new extreme points of $\mathcal{P}'$. Instead, we employ a public-domain code for solving the binary knapsack problem distributed by Pisinger at [74] that uses the *combo algorithm* described in [60]. The algorithm is based on dynamic programming. In Listing 4.4, we show the main elements of declaring a user-specified solver for the

```
1  DecompSolverStatus GAP_DecompApp::solveRelaxed(const int          whichBlock,
2                                                 const double     * costCoeff,
3                                                 DecompVarList & newVars){
4
5     DecompSolverStatus status = DecompSolStatNoSolution;
6     if(!m_appParam.UsePisinger)
7        return status;
8
9     vector<int>        solInd;
10    vector<double>     solEls;
11    double             varCost    = 0.0;
12    const double     * costCoeffB = costCoeff + getOffsetI(whichBlock);
13
14    status = m_knap[whichBlock]->solve(whichBlock,
15                                       costCoeffB,
16                                       solInd,
17                                       solEls,
18                                       varCost);
19
20    DecompVar * var = new DecompVar(solInd,
21                                    solEls,
22                                    varCost,
23                                    whichBlock);
24    newVars.push_back(var);
25    return status;
26 }
```

Listing 4.4: DIP solveRelaxed for GAP example

subproblem by derivation of the base function `solveRelaxed`. The framework's recourse is determined by the `DecompSolverStatus` status code returned by this method. In the case of `DecompSolStatNoSolution`, the framework attempts to solve the subproblem as a generic MILP, assuming the explicit construction of $\mathcal{Q}'$ was defined and set. The inputs to the method `solveRelaxed()` are as follows:

- `whichBlock` defines which block it is currently processing, and
- `costCoeff` defines the coefficients of the cost vector used in the subproblem.

In lines 14–17, we are calling out to the solver provided by Pisinger's code. This code solves the binary knapsack problem, minimizing the provided cost. It returns a sparse vector that represents the solution to that knapsack problem. Then, in lines 20–23, that vector is used to create a `DecompVar` object that is then returned in a list of extreme points to be considered as candidates for adding to the master formulation.

In the following section, we now look again at the Traveling Salesman Problem. For TSP, we also need to implement the `solveRelaxed` method so that we can take advantage of efficient algorithms for solving the 1-tree or 2-matching relaxations. In addition, TSP requires a user-defined separation algorithm for the subtour elimination constraints. We explore this next.

4.3.3 Traveling Salesman Problem

In Listing 4.5 we show the derivation of the base function `generateCuts` in the context of separation for subtour elimination constraints for TSP. The only input to this method is $\hat{x}$, which defines the current point to be separated, The output is a list of `DecompCut` objects that separate the current point. The details of the actual separation algorithm are contained in the call to the `generateCutsSubtour` method of the `TSP_Concorde` object at line 13. This method is a wrapper around Concorde's algorithm for finding the minimum cut [1]. It returns a vector of `ConcordeSubtourCuts` that are simply containers for a set of nodes in the graph that form a violated subtour. From these sets of nodes, we must formulate the SEC constraint in the form of a cut in the compact space of the edge variables defined for the TSP as in (1.28). This is done at line 17 in the constructor of the class `TSP_SubtourCut`, which is derived from the base class `DecompCut`. Since the `TSP_SubtourCut` *is a* `DecompCut` it can be returned in the list of `newCuts` for DIP to process.

4.4 Other Examples

In this section we briefly discuss some of the other example applications that are included in the COIN distribution for the sake of illustrating the various types of models that can be solved using DIP. In addition, in Chapter 4 we use some of these applications for an empirical analysis of certain algorithmic components. A list of the current applications included in the distribution appears in Table 4.4. The table lists the application name, a short description, the choice of relaxation $\mathcal{P}'$, and the form of the user input data. In addition, it shows what technology was used for the associated optimization and separation subproblems.

In the application AP3, we solve the *Three-Index Assignment Problem*, which is the problem of

```
1  int TSP_DecompApp::generateCuts(const double  * x,
2                                  DecompCutList & newCuts){
3
4     int                  nCuts            = 0;
5     UtilGraphLib      & graphLib          = m_tsp.m_graphLib;
6     TSP_Concorde      & tspConcorde       = m_tsp.m_concorde;
7     tspConcorde.buildSubGraph(graphLib.n_vertices,
8                               graphLib.n_edges, x);
9
10    vector<ConcordeSubtourCut> subtourCuts;
11    int c;
12    int n_vertices = graphLib.n_vertices;
13    int n_subtour  = tspConcorde.generateCutsSubtour(subtourCuts);
14    for(c = 0; c < n_subtour; c++){
15       vector<int>     & S        = subtourCuts[c].S;
16       vector<bool>    & inS      = subtourCuts[c].inS;
17       TSP_SubtourCut * sec_cut = new TSP_SubtourCut(inS, S);
18       newCuts.push_back(sec_cut);
19       nCuts++;
20    }
21    return nCuts;
22 }
```

Listing 4.5: DIP generateCuts for TSP example

finding a minimum-weight clique cover of a complete tripartite graph. This problem is a generalization of the well-known *Assignment Problem* (AP). In this application, we use the AP solver provided by Jonker [45] to solve the optimization subproblem. This code is based on a shortest augmenting path algorithm described in [44]. In addition, we implemented various separation routines for some facet-defining valid inequalities, which are discussed in [76].

As mentioned before, in the GAP application, we solve the optimization subproblem using the knapsack solver provided by Pisinger [74]. For the separation subproblem we employ CGL.

The application MAD attempts to solve the *Matrix Decomposition Problem* described in [15]. This problem attempts to decompose a matrix row-wise into a given number of blocks while satisfying capacity constraints on each block. The optimization subproblem, in this case, is the maximum node-weighted clique problem, for which we employ the public-domain solver Cliquer [71]. For the separation subproblem we again employ CGL.

In the TSP application, we generate valid inequalities by using methods provided by the software package Concorde [1]. As described in Examples 3a and 3b, we consider relaxations based on extreme points from the 1-tree polytope and the 2-matching polytope. In the latter case we simply

Table 4.3: COIN/DIP Applications

Application	**Description**	$\mathcal{P}'$	**OPT**(c)	**SEP**(x)	**Input**
AP3	3-index assignment	AP	Jonker	user	user
ATM	cash management (SAS COE)	MILP(s)	CBC	CGL	user
GAP	generalized assignment	KP(s)	Pisinger	CGL	user
MAD	matrix decomposition	MaxClique	Cliquer	CGL	user
MILP	random partition into A', A''	MILP	CBC	CGL	mps
MILPBlock	user-defined blocks for A'	MILP(s)	CBC	CGL	mps, block
MMKP	multi-dim/choice knapsack	MCKP	Pisinger	CGL	user
		MDKP	CBC	CGL	user
SILP	intro example, tiny IP	MILP	CBC	CGL	user
TSP	traveling salesman problem	1-Tree	Boost	Concorde	user
		2-Match	CBC	Concorde	user
VRP	vehicle routing problem	k-TSP	Concorde	CVRPSEP	user
		b-Match	CBC	CVRPSEP	user

use CBC to solve the optimization subproblem; in the former case we employ the graph algorithms included in the software package Boost/Graph [84].

In the VRP application, we also consider two relaxations mentioned in Section 2.3.1. They are the Multiple Traveling Salesman Problem (k-TSP) and the Perfect b-Matching Problem (b-Match). In the latter case we simply formulate the subproblem as an MILP and use CBC. In the former case we use an expanded graph to formulate k-TSP as a standard TSP and use the solvers in Concorde. For the separation routines, we use the software package CVRPSEP [57], which is described in [58].

In Chapter 5 we will expand in full detail on the remaining applications: ATM, MMKP, MILP, and MILPBlock.

Chapter 5

Applications and Computational Results

This chapter describes several applications written using DIP in support of work done at SAS Institute. For each, we present computational results referring back to some of the details considered in Chapter 3. As stated earlier, although the theory behind integrated decomposition methods has been around for some time, there have been very few studies on the implementation details. We expect this area of computational research to grow dramatically as solvers for more real-world applications are successfully implemented. We hope that DIP can help facilitate the study of these things.

In the first section, we present the Multi-Choice Multi-Dimensional Knapsack Problem (MMKP), an important subproblem used in the algorithms present in SAS Marketing Optimization (SAS/MO). SAS/MO attempts to improve the return-on-investment for marketing campaign offers. It does this by targeting higher response rates, improving channel effectiveness, and reducing spending. We look at a number of benchmark instances for MMKP to compare and contrast the different integrated methods and some associated options available in DIP.

In the second section, we introduce an application from the banking industry for ATM cash management that we worked on for the Center of Excellence in Operations Research at SAS Institute. We model the problem as a mixed integer nonlinear program and create an application in DIP to solve an approximating MILP. We show results using DIP's branch-and-price-and-cut method and compare it directly to the branch-and-cut algorithm in Cplex 10.2 [28].

In the third section, we present another application developed in DIP, called MILPBlock, which

provides a black-box solver, using integrated methods, to solve generic MILPs that have block-angular structure. We present computational results using MILPBlock on a model presented to us from SAS Retail Optimization. The model comes from a multi-tiered supply chain distribution problem.

5.1 Multi-Choice Multi-Dimensional Knapsack

The *Multi-Choice Multi-Dimensional Knapsack Problem* (MMKP) is an important problem that has attracted a great deal of interest in numerous industries. One well-documented context is for quality adaptation and admission control of interactive multimedia systems [19]. It has also been used for service-level agreement management in telecommunications networks [91]. At SAS Institute, MMKP has been recognized as an important subproblem in the algorithms used for SAS Marketing Optimization and, hence, motivated our study of this application using the ideas presented in this thesis.

Given a set of groups of items, the goal is to select the best item in each group so as to maximize the value, given some set of resource constraints. Let N define the set of groups, and for each group i, let L_i define the set of items in that group. Let M be the set of resource types and define r_{kij} to be the amount of consumption of resource type k for item j in group i. Define v_{ij} as the value of item j in group i, and b_k as the capacity of resource type k. With each possible selection, we associate a binary variable x_{ij}, which, if set to 1, indicates that item j from group i is selected. Then an ILP formulation of MMKP is as follows:

$$\max \sum_{i\in N}\sum_{j\in L_i} v_{ij}x_{ij},$$

$$\sum_{i\in N}\sum_{j\in L_i} r_{kij}x_{ij} \le b_k \quad \forall k \in M, \tag{5.1}$$

$$\sum_{j\in L_i} x_{ij} = 1 \quad \forall i \in N, \tag{5.2}$$

$$x_{ij} \in \{0,1\} \quad \forall i \in N, j \in L_i. \tag{5.3}$$

In this formulation, equations (5.2) ensure that exactly one item is selected from each group. Inequalities (5.1) enforce the capacity restrictions on each type of resource.

The relaxation that we focus on for MMKP is the well-known *Multi-Choice Knapsack Problem* (MCKP), which is simply an MMKP with only one resource type, i.e., $|M| = 1$. Let us choose one particular resource type $m \in M$ to define our MCKP relaxation. Now, we break out the resource constraints as follows:

$$\sum_{i \in N} \sum_{j \in L_i} r_{mij} x_{ij} \leq b_m, \tag{5.4}$$

$$\sum_{i \in N} \sum_{j \in L_i} r_{kij} x_{ij} \leq b_k \quad \forall k \in M \setminus \{m\}, \tag{5.5}$$

so that we can define the associated polyhedra for our decomposition. That is,

$$\begin{aligned}
\mathcal{P} &= \text{conv } \{x_{ij} \in \{0,1\} \; \forall i \in N, j \in L_i \mid x \text{ satisfies } (5.1),\ (5.2),\ (5.3)\}, \\
\mathcal{P}' &= \text{conv } \{x_{ij} \in \{0,1\} \; \forall i \in N, j \in L_i \mid x \text{ satisfies } (5.4),\ (5.2),\ (5.3)\}, \\
\mathcal{Q}'' &= \qquad \{x_{ij} \in [0,1] \; \forall i \in N, j \in L_i \mid x \text{ satisfies } (5.5)\}.
\end{aligned}$$

We developed an application in DIP to solve MMKP using the integrated methods discussed earlier. The chosen relaxation, MCKP, is studied extensively by Pisinger in [75]. For solving this relaxation, we employed his public-domain code called *mcknap* [74]. The algorithm for MCKP uses a sophisticated core-based branch-and-bound algorithm integrated with dynamic programming. For generation of valid inequalities, we used CGL, which includes the class of Knapsack Cover cuts that can be useful for solving MMKP due to the structure of constraints (5.1).

In the following sections, we present results on the use of DIP to solve this problem. We used a standard set of benchmarks that can be found in [43]. All comparisons were run on the *inferno* servers, which are part of the High Performance Computing cluster at Lehigh University. Each machine is running the CentOS (release 5), 64-bit x86_64 operating system and has a dual quad-core Xeon 1.8Ghz processor, 16GB of memory, and 4MB of cache. For a baseline comparison, we compare our results using DIP with the branch-and-cut algorithm provided by CPLEX 10.2 [28].

In each run we used a time limit of 600 seconds and focus on the best solution and gap provided within the limit.

5.1.1 Results on Integrated Methods

In this first experiment we compared the following variants of DIP against CPLEX 10.2: (DIP-CPM) branch-and-cut, using CGL cuts; (DIP-PC) branch-and-price-and-cut, using CGL cuts and *mcknap* to solve the relaxation MCKP; and, (DIP-DC): branch-and-cut, using CGL cuts and using decompose-and-cut (with the MCKP relaxation), for separation of decomposition cuts. We provide detailed results in Tables A.1 and A.2, in the Appendix.

Here, we provide a summary of results with Table 5.1. For each solver, we provide the time to solve (Time) and the percentage gap (Gap) [1]. In addition, Figure 5.1 shows the results in the form of a *performance profile* [29], which is a way to easily visualize the relative performance of algorithmic variants. Given some specified comparison metric, a performance profile gives the cumulative distribution function of the ratio of that metric for a particular algorithm to the best corresponding value obtained in any of the algorithms used. Since the majority of the MMKP instances are too difficult to solve to optimality, we use the percentage gap between the lower and upper bounds as our comparison metric.

It is not too surprising that the performance of DIP's branch-and-cut algorithm performs poorly as compared to CPLEX. There are many aspects of implementing a state-of-the-art branch-and-cut solver that are out of the scope of this research and therefore not yet included in DIP. The most important missing aspects include: a presolver, better branching strategies, and primal heuristics. A presolver is important to reduce and tighten the formulation before it reaches the main solution process. The reductions done by the presolver have numerous implications on the performance of subsequent cutting planes, branching decisions, and heuristics. Better branching strategies are also extremely important to overall performance. In DIP, we simply choose the branching variable (in the compact space) as that variable that is currently most fractional. When selecting a node from the active search tree for processing, we simply choose the node with the best relaxed objective.

[1] For the summary tables, Time=T means the solver hit the specified time limit, Gap=OPT means the solver declared the problem optimal, and Gap=∞ means the solver found no feasible solutions.

	CPX10.2		DIP-CPM		DIP-PC		DIP-DC	
Instance	**Time**	**Gap**	**Time**	**Gap**	**Time**	**Gap**	**Time**	**Gap**
I1	0.00	OPT	0.02	OPT	0.04	OPT	0.14	OPT
I10	T	0.05%	T	∞	T	11.86%	T	0.15%
I11	T	0.03%	T	∞	T	12.25%	T	0.14%
I12	T	0.01%	T	∞	T	7.93%	T	0.10%
I13	T	0.02%	T	∞	T	11.89%	T	0.12%
I2	0.01	OPT	0.01	OPT	0.05	OPT	0.05	OPT
I3	1.17	OPT	23.23	OPT	T	1.07%	T	0.75%
I4	15.71	OPT	T	∞	T	5.14%	T	0.77%
I5	0.01	0.01%	0.01	OPT	0.13	OPT	0.05	OPT
I6	0.14	OPT	0.07	OPT	T	0.28%	0.63	OPT
I7	T	0.08%	T	∞	T	14.32%	T	0.09%
I8	T	0.09%	T	∞	T	13.36%	T	0.20%
I9	T	0.06%	T	∞	T	10.71%	T	0.19%
INST01	T	0.43%	T	∞	T	9.99%	T	0.70%
INST02	T	0.09%	T	∞	T	7.39%	T	0.45%
INST03	T	0.38%	T	∞	T	3.83%	T	0.85%
INST04	T	0.34%	T	∞	T	7.48%	T	0.45%
INST05	T	0.18%	T	∞	T	10.23%	T	0.62%
INST06	T	0.21%	T	∞	T	9.82%	T	0.38%
INST07	T	0.36%	T	∞	T	15.75%	T	0.62%
INST08	T	0.25%	T	∞	T	11.55%	T	0.46%
INST09	T	0.21%	T	∞	T	15.24%	T	0.40%
INST11	T	0.22%	T	∞	T	7.96%	T	0.39%
INST12	T	0.18%	T	∞	T	7.90%	T	0.42%
INST13	T	0.08%	T	∞	T	2.97%	T	0.14%
INST14	T	0.05%	T	∞	T	3.89%	T	0.09%
INST15	T	0.04%	T	∞	T	3.43%	T	0.10%
INST16	T	0.06%	T	∞	T	2.19%	T	0.06%
INST17	T	0.03%	T	∞	T	2.09%	T	0.09%
INST18	T	0.03%	T	∞	T	4.43%	T	0.06%
INST19	T	0.03%	T	∞	T	3.13%	T	0.04%
INST20	T	0.03%	T	∞	T	3.05%	T	0.04%
Optimal		5		5		3		4
$\leq$ 1% Gap		32		5		4		32
$\leq$ 10% Gap		32		5		22		32

Table 5.1: MMKP: CPX10.2 vs CPM/PC/DC (Summary Table)

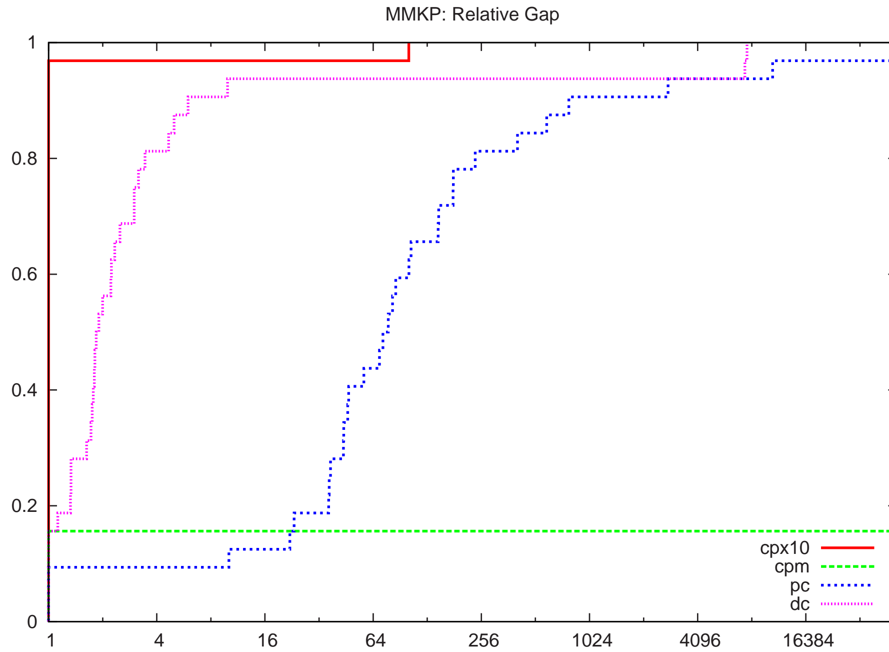


Figure 5.1: MMKP: CPX10.2 vs CPM/PC/DC (Performance Profile)

Perhaps the most essential missing piece is the lack of primal heuristics for generating feasible solutions and therefore good upper bounds early in the search tree. The ability to find good feasible solutions is extremely important to overall performance. CPLEX 10.2 currently employs numerous different primal heuristics.

Despite this, the performance of DIP's integrated methods relative to CPLEX is quite acceptable. The default branch-and-price-and-cut (using MCKP as the relaxation) performs fairly well, finding solutions within 10% of optimal in 22 of 32 cases. Relative to our implementation of branch-and-cut, which only found 5 of 32, this is very good. This example supports our claim that inner methods can be very useful when the polyhedron defined by the outer approximation is not *good enough*. Our outer approximation, in this case, is simply defined by the set of valid inequalities defined in CGL. Recall, from Section 4.2, that this includes the following classes: Knapsack Cover, Flow Cover, Cliques, Mixed-Integer Rounding, and Gomory Mixed Integer. CPLEX, of course, also has an implementation of the separation for each of these classes of cuts. In addition, there is another class of generic cutting planes, called *GUB Covers* [69] missing from CGL that might be putting DIP at

a disadvantage for this problem type. In fact, these cuts, which are strengthenings of the Knapsack Cover cuts, are generated from relaxations that have the form of MCKP. Clearly, since MCKP is an important substructure of MMKP, the lack of GUB Covers could be a major factor in performance.

Now, since we are using MCKP for the generation of an inner approximation, we are implicitly generating the same polyhedral approximation as CPLEX is when they generate GUB covers. This might partially explain why our integrated methods perform well compared to CPLEX and outperforms our direct cutting plane method. Moreover, the implementation of branch-and-cut that includes decomposition cuts, seems to be outperforming our branch-and-price-and-cut. In fact, is very close to the performance of CPLEX. Both CPLEX and DIP with decompose-and-cut find a solution within 1% of optimal in all 32 cases. CPLEX finds the optimal solution in 5 cases, while DIP does so in 4 cases. The dramatic improvement over standard branch-and-cut implies that, in these cases, the decomposition cuts are quite effective. In the next section, we look at how our idea of nested pricing, described in Section 3.3, can improve the performance of branch-and-price-and-cut on this problem.

5.1.2 Results using Nested Pricing

In order to test our ideas on using nested pricing, we now consider another relaxation that we call the *Multi-Choice 2-Dimensional Knapsack Problem* (MC2KP). For each $p \in M \setminus \{m\}$, define the MC2KP polyhedron as

$$\mathcal{P}_p^{MC2KP} = \mathcal{P}' \cap \text{conv} \left\{ x_{ij} \in \mathbb{R}^+ \ \forall i \in N, j \in L_i \;\middle|\; \sum_{i \in N} \sum_{j \in L_i} r_{pij} x_{ij} \leq b_p \right\}. \tag{5.6}$$

Since each of those polyhedra are contained in $\mathcal{P}'$, any (or all) of them are candidates for generating extreme points heuristically. Unfortunately, there are no known algorithms for solving MC2KP, so there will be an efficiency tradeoff to consider. To implement this in DIP, we construct the constraint matrix for each polyhedron directly and use the built-in MILP solver (CPLEX10.2) to solve the optimization subproblems. Since we have several polyhedra, we have many different possible strategies for choosing which polyhedra to use and how to set the limits on the solver

to return heuristic solutions. To keep things simple, we chose to solve every subproblem at each iteration, and we set the integrality gap for the MILP solver to 10%. Note, that this is a perfect opportunity to exploit parallelism, since the subproblems used to generate extreme points can all be solved independently and simultaneously. This is another area of future research that we consider in Chapter 6.

The comparison of the default branch-and-price-and-cut (DIP-PC) and a version using the nested polyhedra (DIP-PC-M2) is shown in Table 5.2 and with a performance profile in Figure 5.2. Using the default method, we are able to solve 22 out of 32 to within 10% gap, while using nested pricing, we can now solve 27 out of 32 within the gap. The performance profile also shows a clear improvement when using nested pricing.

With evidence that the nested pricing idea can be beneficial, we now push the idea further. In the same table and figure, we show the results of an experiment in which we use $\mathcal{P}$ itself as the nested polyhedron (DIP-PC-MM). That is, when solving the subproblem in the integrated method, we heuristically solve MMKP using the built-in MILP solver. The improvements above our default implementation were dramatic. Now, we are able to solve all cases to within 10% of optimality and 20 out of 32 cases to within 1%.

To summarize the results so far, we show, in Figure 5.3, all the experiments on the same performance profile. In addition, in Figure 5.4, we show a stacked bar chart that gives the percentage of instances solved to optimality, within 5% gap and within 10% gap respectively. As can be seen, our implementation of decompose-and-cut performs best relative to CPLEX. After that, the nested pricing that uses $\mathcal{P}$ to generate extreme points heuristically is next, followed by the other two variants of branch-and-price-and-cut. As expected, the cutting plane method is the wosrst performer.

5.1.3 Comparison of Master Solver

As discussed in Section 3.7, the choice of solver for the master problem can have an effect on performance. The natural choice is to use primal simplex after adding columns and dual simplex after adding rows. To test this, we ran each of the variants of our algorithms twice. In the first case, we used DIP's default settings, which uses primal simplex after adding columns (denoted PC-PS,

	DIP-PC		DIP-PC-M2		DIP-PC-MM	
Instance	**Time**	**Gap**	**Time**	**Gap**	**Time**	**Gap**
I1	0.04	OPT	0.16	OPT	0.08	OPT
I10	T	11.86%	T	6.99%	T	0.63%
I11	T	12.25%	T	11.15%	T	0.60%
I12	T	7.93%	T	11.41%	T	0.79%
I13	T	11.89%	T	13.65%	T	0.52%
I2	0.05	OPT	0.45	OPT	0.14	OPT
I3	T	1.07%	T	1.18%	T	1.10%
I4	T	5.14%	T	3.18%	T	1.23%
I5	0.13	OPT	0.14	OPT	0.07	OPT
I6	T	0.28%	483.53	OPT	T	0.25%
I7	T	14.32%	T	4.85%	T	0.97%
I8	T	13.36%	T	9.79%	T	0.67%
I9	T	10.71%	T	10.57%	T	0.73%
INST01	T	9.99%	T	5.97%	T	1.86%
INST02	T	7.39%	T	7.29%	T	1.74%
INST03	T	3.83%	T	11.93%	T	1.61%
INST04	T	7.48%	T	7.04%	T	1.56%
INST05	T	10.23%	T	8.84%	T	1.11%
INST06	T	9.82%	T	9.77%	T	1.39%
INST07	T	15.75%	T	8.78%	T	1.23%
INST08	T	11.55%	T	8.50%	T	1.37%
INST09	T	15.24%	T	8.48%	T	0.89%
INST11	T	7.96%	T	8.72%	T	1.13%
INST12	T	7.90%	T	6.72%	T	1.03%
INST13	T	2.97%	T	3.06%	T	0.76%
INST14	T	3.89%	T	3.67%	T	0.52%
INST15	T	3.43%	T	2.81%	T	0.78%
INST16	T	2.19%	T	3.01%	T	0.50%
INST17	T	2.09%	T	2.16%	T	0.39%
INST18	T	4.43%	T	2.60%	T	0.41%
INST19	T	3.13%	T	3.97%	T	0.46%
INST20	T	3.05%	T	4.06%	T	0.94%
Optimal		3		4		3
$\leq$ 1% Gap		4		4		20
$\leq$ 10% Gap		22		27		32

Table 5.2: MMKP: PC vs PC Nested with MC2KP and MMKP (Summary Table)

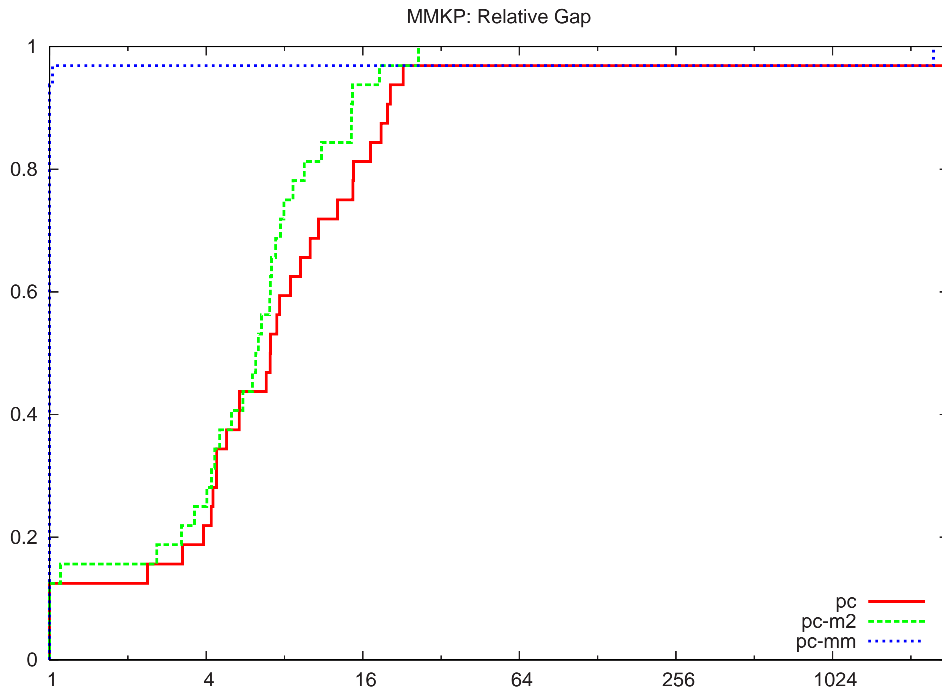


Figure 5.2: MMKP: PC vs PC Nested with MC2KP and MMKP (Performance Profile)

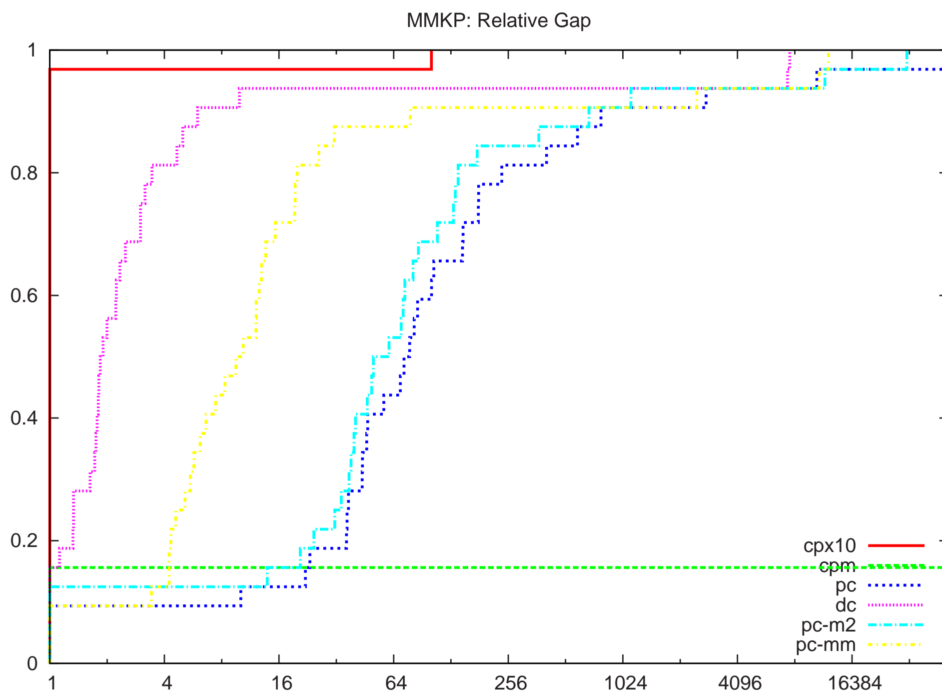


Figure 5.3: MMKP: CPX10.2 vs CPM/PC/DC/PC-M2/PC-MM (Performance Profile)

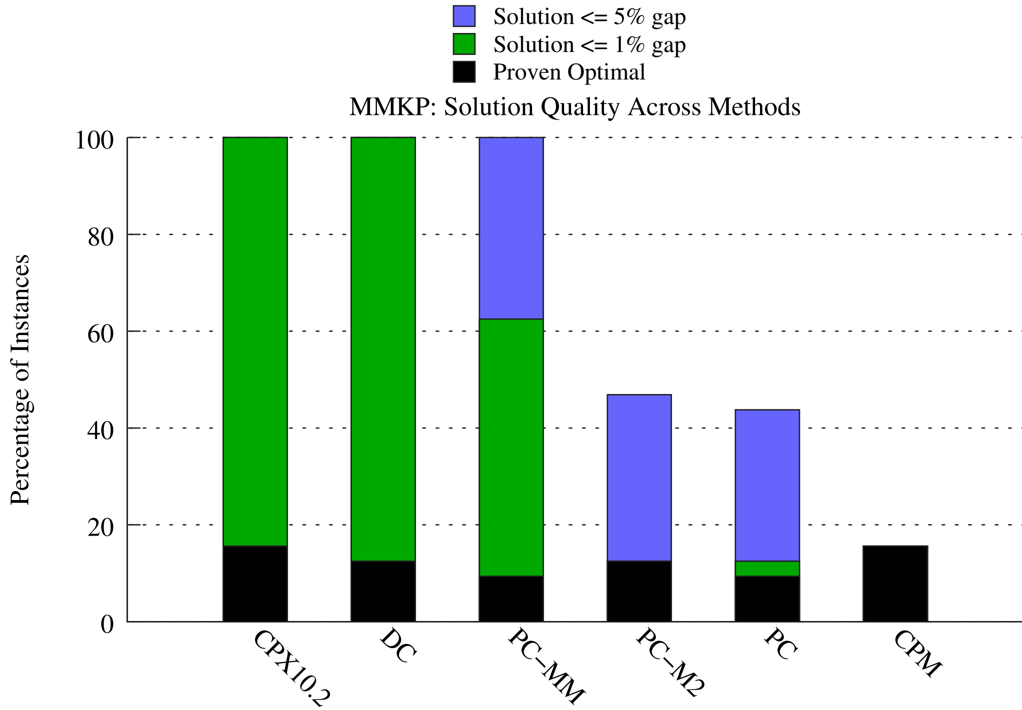


Figure 5.4: MMKP: CPX10.2 vs CPM/PC/DC/PC-M2/PC-MM (Stacked Bar Chart)

DC-PS, PC-M2-PS, and PC-MM-PS). In the second case, we used dual simplex at every iteration (denoted PC-DS, DC-DS, PC-M2-DS, and PC-MM-DS). The results from these experiments are shown in the four performance profiles in Figures 5.5 and 5.6. From the experiments, there is no clear winner, though dual simplex is slightly favored. This is a bit surprising given the fact that when using dual simplex after adding columns, the solver must run a first phase to generate a dual feasible basis, while primal simplex can start directly in the second phase. However, it has been documented that the dual simplex method, on average, performs somewhat better than primal simplex [12]. So, the benefit of the warm-start might be negated.

This experiment could be improved by also comparing the use of an interior point method for the master solver. Unfortunately, the OSI interface used in DIP does not currently support interaction with interior point methods. For this reason, we have left this exercise for future research.

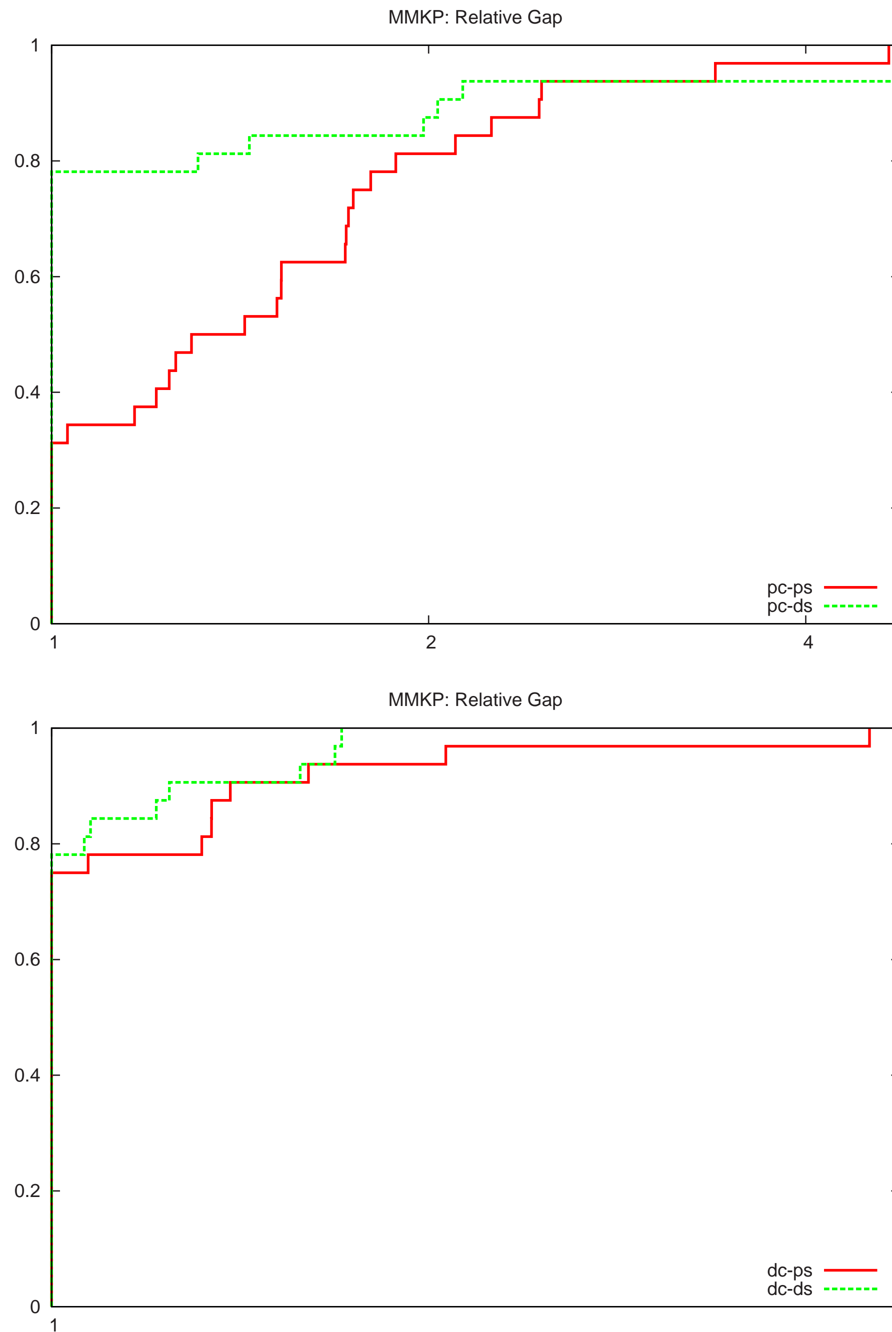


Figure 5.5: MMKP: Comparison of Primal vs Dual Simplex for Master LP solver (PC/DC)

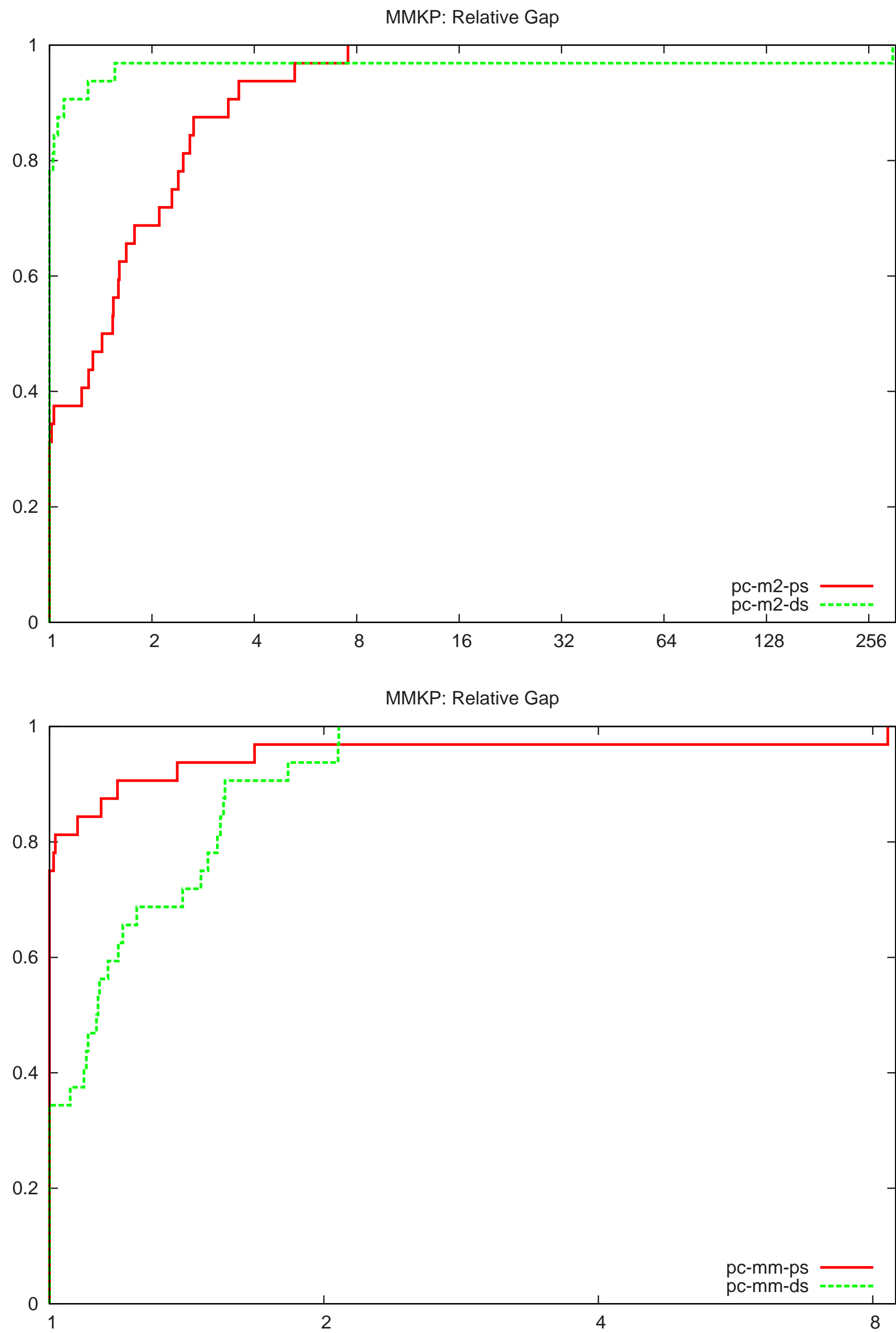


Figure 5.6: MMKP: Comparison of Primal vs Dual Simplex for Master LP solver (PC-M2/PC-MM)

5.2 ATM Cash Management Problem

The application described in this section was presented to the Center of Excellence in Operations Research Applications (OR COE) at SAS Institute. The model definitions and data have been obfuscated to protect the proprietary nature of the work. However, the general structure of the problem has been retained. The goal of this application is to determine a schedule for allocation of cash inventory at bank branches to service a preassigned subset of automated teller machines (ATMs). Given historical *training data* per day for each ATM we first define a polynomial fit for the predicted cash flow need. This is done using SAS forecasting tools to determine the expected total daily cash withdrawals and deposits at each branch. The modeling of this prediction depends on various seasonal factors, including the days of the week, the weeks of the month, holidays, typical salary disbursement days, location of the branches, and other demographic data. We then want to determine the multipliers that to minimize the mismatch based on predicted withdrawals. The amount of cash allocated to each day is subject to a budget constraint. In addition, there is a constraint for each ATM that limits the number of days the cash flow can be less than the predicted withdrawal. This scenario is referred to as a *cash-out*. Cash allocation plans are usually finalized at the beginning of the month, and any deviation from the plan is costly. In some cases, it may not even be feasible. So, the goal is to determine a policy for cash distribution that balances the inventory levels while satisfying the budget and customer dissatisfaction constraints. By keeping too much cash-on-hand for ATM fulfillment, the banks will incur investment opportunity loss. In addition, regulatory agencies in many nations enforce a minimum cash reserve ratio at branch banks. According to regulatory policy, the cash in ATMs or in transit, do not contribute towards this threshold.

5.2.1 Mixed Integer Nonlinear Programming Formulation

The most natural formulation for this model is in the form of a mixed integer nonlinear program (MINLP). Let A denote the set of ATMs and D denote the set of days used in the training data. The predictive model fit is defined by the following set of parameters: $(c^x_{ad}, c^y_{ad}, c^{xy}_{ad}, c^u_{ad})$ for each ATM a on each day d. Define variables (x_a, y_a, u_a) for each ATM that, when applied to the predictive model, give the estimated cash flow need per day, per ATM. In addition, define a surrogate variable

f_{ad} for each ATM on each day that defines the net cash flow minus withdrawals given by the fit. Let B_d define the budget per day, K_a define the limit on cash-outs per ATM, and w_{ad} define the historical withdrawals at a particular ATM, on a particular day. Then the following MINLP models this problem.

$$
\begin{aligned}
\min \ & \sum_{a \in A} \sum_{d \in D} |f_{ad}|, & & & \\
\text{s.t. } \ & c^x_{ad} x_a + c^y_{ad} y_a + c^{xy}_{ad} x_a y_a + c^u_{ad} u_a - w_{ad} &= \ & f_{ad} & \forall a \in A, d \in D, && (5.7)\\
& \sum_{a \in A} (f_{ad} + w_{ad}) &\leq \ & B_d & \forall d \in D, && (5.8)\\
& |\{d \in D \mid f_{ad} < 0\}| &\leq \ & K_a & \forall a \in A, && (5.9)\\
& x_a, y_a &\in \ & [0,1] & \forall a \in A, && (5.10)\\
& u_a &\geq \ & 0 & \forall a \in A, && (5.11)\\
& f_{ad} &\geq \ & -w_{ad} & \forall a \in A, d \in D. && (5.12)
\end{aligned}
$$

Inequalities (5.8) and (5.9) ensure that the solution satisfies the budget and cash-out constraints, respectively. Constraint (5.7) defines the surrogate variable f_{ad}, which gives the estimated net cash flow.

In order to put this model into a more standard form, we first must use some standard model reformulations to linearize the absolute value and the cash-out constraint (5.9).

Linearization of Absolute Value. A well-known reformulation for linearizing the absolute value of a variable is to introduce one variable for each *side* of the absolute value. The following system:

$$
\begin{aligned}
\min \ & |y|, \\
\text{s.t. } \ & Ay \leq b,
\end{aligned}
\qquad \text{is equivalent to} \qquad
\begin{aligned}
\min \ & y^+ + y^-, \\
\text{s.t. } \ & A(y^+ - y^-) \leq b, \\
& y^+, y^- \geq 0.
\end{aligned}
$$

Let f^+_{ad} and f^-_{ad} represent the positive and negative parts, respectively, of the net cash flow f_{ad}.

Then, we can rewrite the model, removing the absolute value, as the following:

$$
\begin{aligned}
\min\ & \sum_{a\in A}\sum_{d\in D}\left(f^+_{ad}+f^-_{ad}\right), & & & \\
\text{s.t.}\ & c^x_{ad}x_a + c^y_{ad}y_a + c^{xy}_{ad}x_a y_a + c^u_{ad}u_a - w_{ad} &=\ & f^+_{ad}-f^-_{ad} & \forall a\in A, d\in D,\\
& \sum_{a\in A}(f^+_{ad}-f^-_{ad}) + w_{ad} &\le\ & B_d & \forall d\in D,\\
& |\{d\in D \mid (f^+_{ad}-f^-_{ad})<0\}| &\le\ & K_a & \forall a\in A,\\
& x_a, y_a &\in\ & [0,1] & \forall a\in A,\\
& u_a &\ge\ & 0 & \forall a\in A,\\
& f^+_{ad} &\ge\ & 0 & \forall a\in A, d\in D,\\
& f^-_{ad} &\in\ & [0, w_{ad}] & \forall a\in A, d\in D.
\end{aligned}
$$

Modeling the Cash-Out Constraints In order to count the number of times a cash-out occurs, we need to introduce a binary variable to keep track of when this event occurs. Let v_{ad} be an indicator variable that takes value 1 when the net cash flow is negative. We can model the following implication $f^-_{ad} > 0 \Rightarrow v_{ad} = 1$, or its contrapositive $v_{ad} = 0 \Rightarrow f^-_{ad} \le 0$, by adding the constraint

$$f^-_{ad} \le w_{ad}v_{ad}\ \forall a\in A, d\in D.$$

Now, we can model the cash-out constraint simply by counting the number of days the net-cash flow is negative for each ATM, as follows:

$$\sum_{d\in D} v_{ad} \le K_a\ \forall a\in A.$$

The MINLP model can now be written as follows:

$$
\begin{aligned}
\min \; & \sum_{a \in A} \sum_{d \in D} \left(f_{ad}^{+} + f_{ad}^{-} \right), & & & \\
\text{s.t.} \; & c_{ad}^{x} x_a + c_{ad}^{y} y_a + c_{ad}^{xy} x_a y_a + c_{ad}^{u} u_a - w_{ad} &=& \; f_{ad}^{+} - f_{ad}^{-} & \forall a \in A, d \in D, \\
& \sum_{a \in A} (f_{ad}^{+} - f_{ad}^{-}) + w_{ad} &\leq& \; B_d & \forall d \in D, \\
& f_{ad}^{-} &\leq& \; w_{ad} v_{ad} & \forall a \in A, d \in D, \\
& \sum_{d \in D} v_{ad} &\leq& \; K_a & \forall a \in A, \\
& x_a, y_a &\in& \; [0, 1] & \forall a \in A, \\
& u_a &\geq& \; 0 & \forall a \in A, \\
& f_{ad}^{+} &\geq& \; 0 & \forall a \in A, d \in D, \\
& f_{ad}^{-} &\in& \; [0, w_{ad}] & \forall a \in A, d \in D, \\
& v_{ad} &\in& \; \{0, 1\} & \forall a \in A, d \in D.
\end{aligned}
$$

We tried using this model with several of the available MINLP solvers on the NEOS Server for Optimization [22]. However, we had little success solving anything but models of trivial size. We also solicited the help of several researchers doing computational work in MINLP, but thus far, none of the solvers have been able to successfully solve this problem. Presumably the difficulty comes from the non-convexity of the prediction function. Another approach is to formulate an approximation of the problem using mixed integer linear programming (MILP). We show this in the next section.

5.2.2 Mixed Integer Linear Programming Approximation.

Since the predictive model is a forecast, finding the optimal multipliers based on nondeterministic data is not of primary importance. Rather, we want to provide as good a solution as possible in a reasonable amount of time. So, using MILP to approximate the MINLP is perfectly acceptable.

In the original problem we have products of two continuous variables that are both bounded by 0 (lower bound) and 1 (upper bound). This allows us to create an approximate linear model using a few standard modeling reformulations.

Discretization of Continuous Variables. The first step is to discretize one of the continuous variables x_a. The goal is to transform the product $x_a y_a$ of a continuous variable with another continuous variable instead to a continuous variable with a binary variable. By doing this, we can linearize the product form.

We must assume some level of approximation by defining a binary variable for each possible setting of the continuous variable to be chosen from some discrete set. For example, if we let $n = 10$, then we allow x to be chosen from the set $\{0.1, 0.2, 0.3, ..., 1.0\}$. Let $T = \{1, 2, ..., n\}$ represent the possible steps and $c_t = t/n$. Then, we apply the following transformation to variable x_a:

$$\begin{aligned} \sum_{t \in T} c_t x_{at} &= x_a, \\ \sum_{t \in T} x_{at} &\leq 1, \\ x_{at} &\in \{0, 1\} \ \forall t \in T. \end{aligned}$$

The MINLP model can now be rewritten as the following:

$$
\begin{aligned}
\min \ & \sum_{a \in A} \sum_{d \in D} \left(f^+_{ad} + f^-_{ad} \right), \\
\text{s.t.} \ & c^x_{ad} \sum_{t \in T} c_t x_{at} + c^y_{ad} y_a + \\
& \quad c^{xy}_{ad} \sum_{t \in T} c_t x_{at} y_a + c^u_{ad} u_a - w_{ad} = f^+_{ad} - f^-_{ad} && \forall a \in A, d \in D, \\
& \sum_{t \in T} x_{at} \leq 1 && \forall a \in A, \\
& \sum_{a \in A} (f^+_{ad} - f^-_{ad}) + w_{ad} \leq B_d && \forall d \in D, \\
& f^-_{ad} \leq w_{ad} v_{ad} && \forall a \in A, d \in D, \\
& \sum_{d \in D} v_{ad} \leq K_a && \forall a \in A, \\
& y_a \in [0, 1] && \forall a \in A, \\
& u_a \geq 0 && \forall a \in A, \\
& f^+_{ad} \geq 0 && \forall a \in A, d \in D, \\
& f^-_{ad} \in [0, w_{ad}] && \forall a \in A, d \in D, \\
& v_{ad} \in \{0, 1\} && \forall a \in A, d \in D, \\
& x_{at} \in \{0, 1\} && \forall a \in A, t \in T.
\end{aligned}
$$

Linearization of Products. Now, we need to linearize the product of a bounded continuous variable and a binary. This can be accomplished by introducing another variable z, which serves as a surrogate for the product. In general, we know the following relationship:

$$
\begin{aligned}
z &= xy, \\
x &\in \{0,1\}, \\
y &\in [0,1],
\end{aligned}
\qquad \text{is equivalent to} \qquad
\begin{aligned}
z &\geq 0, \\
z &\leq x, \\
z &\leq y, \\
z &\geq x + y - 1, \\
x &\in \{0,1\}, \\
y &\in [0,1].
\end{aligned}
$$

Using this to replace each product form, we now can write the problem as an approximate MILP as

follows:

$$
\begin{aligned}
\min\ & \sum_{a\in A}\sum_{d\in D}\left(f^{+}_{ad}+f^{-}_{ad}\right) \\
\text{s.t.}\ & c^{x}_{ad}\sum_{t\in T}c_t x_{at}+c^{y}_{ad}y_a+ \\
& \quad c^{xy}_{ad}\sum_{t\in T}c_t z_{at}+c^{u}_{ad}u_a-w_{ad} = f^{+}_{ad}-f^{-}_{ad} && \forall a\in A, d\in D, && (5.13)\\
& \sum_{t\in T}x_{at} \le 1 && \forall a\in A, && (5.14)\\
& \sum_{a\in A}(f^{+}_{ad}-f^{-}_{ad})+w_{ad} \le B_d && \forall d\in D, && (5.15)\\
& f^{-}_{ad} \le w_{ad}v_{ad} && \forall a\in A, d\in D, && (5.16)\\
& \sum_{d\in D}v_{ad} \le K_a && \forall a\in A, && (5.17)\\
& z_{at} \le x_{at} && \forall a\in A, t\in T, && (5.18)\\
& z_{at} \le y_a && \forall a\in A, t\in T, && (5.19)\\
& z_{at} \ge x_{at}+y_a-1 && \forall a\in A, t\in T, && (5.20)\\
& z_{at} \ge 0 && \forall a\in A, t\in T, && (5.21)\\
& y_a \in [0,1] && \forall a\in A, && (5.22)\\
& u_a \ge 0 && \forall a\in A, && (5.23)\\
& f^{+}_{ad} \ge 0 && \forall a\in A, d\in D, && (5.24)\\
& f^{-}_{ad} \in [0,w_{ad}] && \forall a\in A, d\in D, && (5.25)\\
& v_{ad} \in [0,1] && \forall a\in A, d\in D, && (5.26)\\
& x_{at} \in [0,1] && \forall a\in A, t\in T, && (5.27)\\
& v_{ad} \in \mathbb{Z} && \forall a\in A, d\in D, && (5.28)\\
& x_{at} \in \mathbb{Z} && \forall a\in A, t\in T. && (5.29)
\end{aligned}
$$

5.2.3 Results

Since we had trouble solving the the MINLP model directly, we had no choice but to move to the approximate MILP formulation. Unfortunately, the size of the approximate MILP was much bigger than the associated MINLP. Due to the fact that state-of-the-art commercial MILP solvers can now handle many different large-scale models, we had confidence that this approach would be successful. Unfortunately, solving the problem directly using commercial solvers proved to be nearly impossible, as will be evident from our computational results.

Examining the structure of the MILP model, it is clear that the constraints can be easily decomposed by ATM. In fact, the only set of constraints involving decision variables across ATMs is the budget constraint (5.15). That is, if we relax constraint (5.15) we are left with independent blocks of constraints, one for each ATM. This fits the block angular structure that we have mentioned in previous chapters and therefore lends itself nicely to our decomposition methods. Using DIP, we built a new application, called ATM, that constructs this approximate MILP and defines the subproblems as follows:

$$\mathcal{P}'_k = \text{conv}\ \{(x, v, z, y, u, f^+, f^-) \mid (x, v, z, y, u, f^+, f^-) \text{ satisfies } (5.13), (5.14), (5.16) - (5.29)\},$$

$$\mathcal{Q}'' = \{(x, v, z, y, u, f^+, f^-) \mid (x, v, z, y, u, f^+, f^-) \text{ satisfies } (5.15), (5.21) - (5.27)\}.$$

In order to test the effectiveness of decomposition methods on this model, we generated a set of problem instances that have the same characteristics as those provided by the client. In order to do this, we took point estimates of all the relevant data parameters and generated random data using normal distributions around these estimates. From this, we then randomly perturbed the constraint requirements to ensure the problem had a feasible solution. We simulated several different sizes based on the number of ATMS and the number of days.

For these experiments, all comparisons were run on the *altair* server at Lehigh University. This machine is running the Redhat Enterprise Linux (release 5) 64-bit x86_64 operating system and has 8 quad-core Xeon 2.3Ghz processor, 128GB of memory, and 6MB of cache. In this case, we compared our results using DIP with the branch-and-cut algorithm provided by CPLEX 11. In each run we used a time limit of 3600 seconds and once again, focus on the best solution and gap

provided within the limit.

Table 5.3 gives a summary of the results for each problem instance. The first three columns denote the size of the problem and an instance id. For each size we generate five random instances. In columns 4-6 (CPX11) we show the results for CPLEX 11, including time to solve (Time), gap (Gap), and the number of nodes generated in the search tree (Nodes). In columns 7-9 (DIP-PC), we show the same results for the default branch-and-price-and-cut method in DIP. Since this problem has a block-diagonal form, we also used this experiment to show the effectiveness of using price-and-branch, as described in Section 3.2.2. These results are shown in columns 10-12 (DIP-PC+).

Once again, we display the comparative results in the form of performance profiles, in Figure 5.7, and a stacked bar chart in Figure 5.8. In Figure 5.7, we use the gap after one hour as the metric for the profile, as well as, the time it took to solve to optimality. It is clear from the results, that all three solvers can solve the majority of the small instances ($|A| = 5$) to optimality. For the medium instances ($|A| = 10, |D| = 50$), CPLEX has some trouble, while the two variants based on integrated methods still find the optimal solution fairly quickly. Looking at the case ($|A| = 10, |D| = 100$), CPLEX no longer finds any feasible solutions after one hour of processing. Interestingly, neither does DIP-PC. However, when using the price-and-branch technique, it is able to solve all 5 instances to within 2% gap. This gives some indication of how this heuristic can greatly enhance performance by producing additional incumbents during the branch-and-bound search.

5.3 Automated Decomposition for Block Angular MILP

One of the main goals in the development of DIP was to provide a more user-friendly environment for the development of application solvers based on decomposition methods. During our work with the SAS Center of Excellence, it became apparent that there was an abundance of difficult client MILPs that could not be solved with direct branch-and-cut methods, for which we would then reconsider using inner methods like Dantzig-Wolfe. The difficulty, of course, was that, in order to

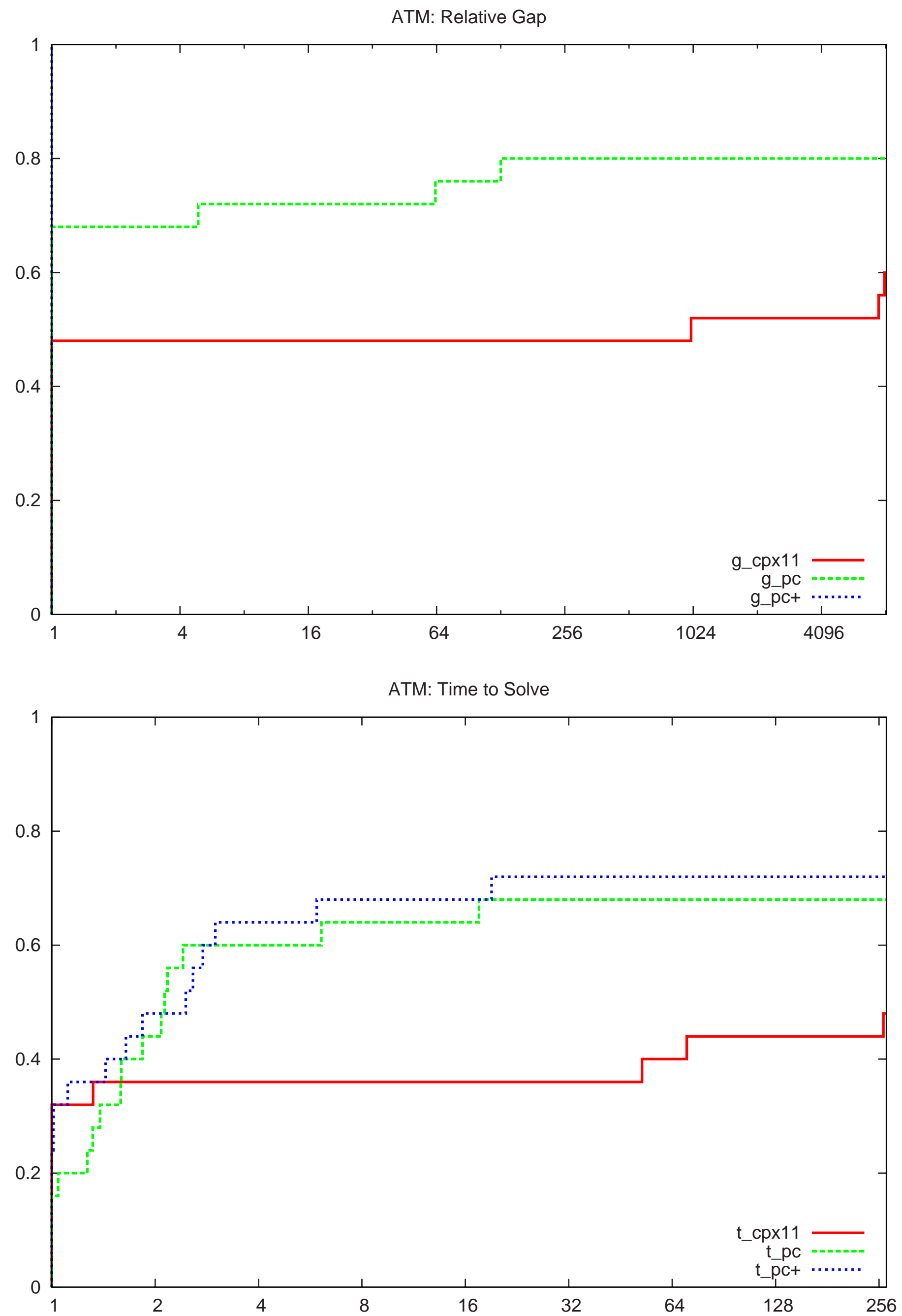


Figure 5.7: ATM: CPX11 vs PC/PC+ (Performance Profiles)

			CPX11			DIP-PC			DIP-PC+		
$\|A\|$	$\|D\|$	s	Time	Gap	Nodes	Time	Gap	Nodes	Time	Gap	Nodes
5	25	1	0.76	OPT	467	1.62	OPT	6	1.96	OPT	6
5	25	2	1.41	OPT	804	1.95	OPT	9	1.57	OPT	7
5	25	3	0.42	OPT	147	7.38	OPT	32	8.03	OPT	32
5	25	4	1.49	OPT	714	2.74	OPT	14	2.45	OPT	13
5	25	5	0.16	OPT	32	0.98	OPT	7	0.95	OPT	6
5	50	1	T	0.10	1264574	162.74	OPT	127	164.46	OPT	131
5	50	2	87.96	OPT	38341	183.28	OPT	273	263.24	OPT	275
5	50	3	8.09	OPT	3576	17.58	OPT	36	22.28	OPT	35
5	50	4	4.13	OPT	1317	3.13	OPT	3	3.17	OPT	3
5	50	5	57.55	OPT	32443	91.30	OPT	145	141.29	OPT	147
10	50	1	T	0.76	998624	297.65	OPT	301	234.47	OPT	156
10	50	2	1507.84	OPT	351879	28.84	OPT	29	52.99	OPT	29
10	50	3	T	0.81	667371	64.72	OPT	64	49.20	OPT	47
10	50	4	1319.00	OPT	433155	7.97	OPT	1	5.00	OPT	1
10	50	5	365.51	OPT	181013	12.49	OPT	3	5.18	OPT	3
10	100	1	T	∞	128155	T	∞	20590	T	0.11	13190
10	100	2	T	∞	116522	T	∞	60554	2437.43	OPT	135
10	100	3	T	∞	118617	T	∞	52902	T	0.20	40793
10	100	4	T	∞	108899	T	∞	47931	T	1.51	59477
10	100	5	T	∞	167617	T	∞	40283	T	0.38	26490
20	100	1	T	∞	93519	379.75	OPT	9	544.49	OPT	9
20	100	2	T	∞	68863	T	16.44	14240	T	0.26	25756
20	100	3	T	∞	95981	T	15.37	41495	T	0.12	3834
20	100	4	T	∞	81836	T	0.39	7554	T	0.08	7918
20	100	5	T	∞	101917	635.59	OPT	21	608.68	OPT	19
Optimal			12			17			18		
$\leq$ 1% Gap			15			18			25		
$\leq$ 10% Gap			15			18			25		

Table 5.3: ATM: CPX11 vs PC/PC+ (Summary Table)

try these methods, we had to create a different application for each client. Even though DIP greatly simplifies this task, the development time was still significant for experimentation on methods which may or may not even help. Often, the models we received had no well-known relaxation from which we could employ specialized techniques for solving the subproblem. However, we did find that, quite often, the models had a block-diagonal structure, leading to independent subproblems. This is not too surprising, since business problems are often modeled such that there are departmental policies that are then governed by some global constraint that couples the system. After enough engagements, we came to realize that, by utilizing DIP's built-in facilities, we could automate the entire process of using integrated methods in a generic manner. Given a model in standard MPS

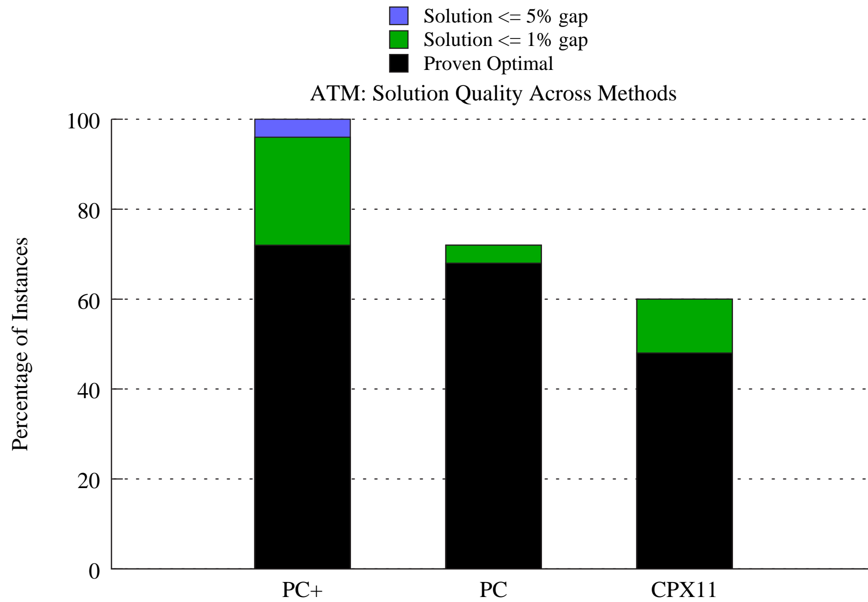


Figure 5.8: ATM: CPX11 vs PC/PC+ (Stacked Bar Chart)

(Mathematical Programming System) format, if the user simply defines which rows belong to which blocks, then DIP can process everything automatically with no user interaction.

To this end, we have developed a DIP application called MILPBlock, which accepts the model in MPS format, as input and a file that identifies the rows in each block. To our knowledge, this is the first time a *black-box* solver for integrated methods has been developed. There is currently work being done at INRIA (Institut National De Recherche En Informatique Et En Automatique) on a product called BapCod (A Generic Branch-and-Price Code) [87] that has the same general goal of providing a black-box implementation. It is our understanding that this framework does not include cut generation in its branch-and-price algorithm.

To test effectiveness (and ease-of-use) of the MILPBlock application, we partnered with the Retail Optimization group at SAS who are working on a product offering for a multi-tiered supply chain distribution problem. In early development, the retail group developed prototype optimization models using SAS's modeling language OPTMODEL [82]. From this, they were able to easily produce the necessary input for MILPBlock.

The following experiments were done on the same hardware we used for the ATM model. We

	CPX11			DIP-PC		
Instance	**Time**	**Gap**	**Nodes**	**Time**	**Gap**	**Nodes**
retail27	T	2.30%	2674921	3.18	OPT	1
retail31	T	0.49%	1434931	767.36	OPT	41
retail3	529.77	OPT	2632157	0.54	OPT	1
retail4	T	1.61%	1606911	116.55	OPT	1
retail6	1.12	OPT	803	264.59	OPT	303

Table 5.4: MILPBlock Retail: CPX11 vs PC (Summary Table)

were given 5 instances that covered a wide spectrum of supply chain configurations. We once again compared our results using DIP with the branch-and-cut algorithm provided by Cplex 11, and a time limit of 3600 seconds. Table 5.4 gives a summary of the results for each problem instance. In columns 2-5 (CPX11), we show the results for Cplex 11, including time to solve (Time), gap (Gap), and the number of nodes generated in the search tree (Nodes). In columns 6-8 (DIP-PC), we show the same results for the default branch-and-price-and-cut method in DIP. While Cplex struggles to solve some of the instances, DIP solves all cases within the specified time limit.

MILPBlock is the first step in creating a black-box implementation of integrated decomposition methods for MILP. The next step is to attempt to embed some automatic recognition of the block diagonal structure, so the user can simply input an MPS file directly. There is a great deal of research in the linear algebra community on detecting this structure [30]. In addition, there are several public-domain software packages that could potentially be integrated into DIP, making MILPBlock a fully automated solver.

Chapter 6

Future Research

In this final chapter, we briefly discuss some potential areas for future research related to decomposition methods. Specifically, we mention some of the missing pieces in the DIP implementation of our framework, as well as ideas for improving overall performance.

Branch-and-Relax-and-Cut Although the conceptual framework included the area of relax-and-cut, we have focused most of our computational study on branch-and-price-and-cut. A basic implementation of relax-and-cut has already been included in DIP. However, we have not yet implemented the ideas discussed in Section 3.1 to integrate relax-and-cut in the branch-and-bound framework, which will allow us to do branch-and-relax-and-cut. Adding this feature will open the door for a great deal more computational experimentation and will complete the mapping between the conceptual framework and the software framework.

Convergence Issues and Stabilization Issues with the convergence of the Dantzig-Wolfe method have been well documented [55]. As discussed in Section 3.7, there have been several papers on using *stability centers* to control the oscillation of the dual solutions and improve convergence. Many authors have stated that this can make a very big difference in overall performance. Conceptually, this should be possible to add to DIP and we hope to investigate this in the near future. Along the same lines, as mentioned in Section 3.7, the use of an interior point method when solving the master problem might also improve convergence by reducing some of the extreme jumps in the dual

solutions when using simplex-based methods. Integration of an interior point solver into DIP is currently work-in-progress.

Identical Subproblems The case of block-diagonal decomposition, discussed in Section 3.2.1, where the subproblems have identical structure, is an important modeling paradigm. Many applications can be modeled in this way, and it would be nice if the framework could handle this situation. Theoretically, much of the machinery breaks down because the mapping between the compact space and the extended space is no longer unique. However, recent work by Vanderbeck in [90] might make it possible to still handle this situation in our framework.

Parallel Processing The decomposition framework has two obvious candidates for parallelization. The first, of course, is to parallelize the branch-and-bound tree search. Fortunately, since we are using ALPS as the base of our tree search method, moving the processing to a parallel environment should not be difficult. ALPS was designed to work in distributed or shared memory environments and already has the infrastructure in place to run in parallel. Since the amount of work done at each node of the tree is generally much higher for integrated methods, as opposed to standard branch-and-cut, there is good potential for speed-ups since the communication overhead will be relatively low.

The second area of parallelism is in the solution of the relaxed polyhedron when generating extreme points in the various methods. There are three areas where we can see potential for performance improvements. The first case is perhaps the most obvious. In the block-diagonal case, the subproblems are independent and can therefore be processed simultaneously. We have already done some preliminary work on making DIP multi-threaded for this case. The second case is for nested pricing. As mentioned in the MMKP application in Section 5.1, we can define many polyhedra which are all contained in the relaxed polyhedra to use in generation of extreme points. The more diverse this set of extreme points, the better the chance to find good incumbents. Since the optimization problems for these polyhedra can be solved independently, we can also do this processing in parallel. The third case is in the generation of decomposition cuts. This is quite similar to the nested pricing case, but perhaps even more flexible. Unlike nested pricing, there is no restriction

on the choice of polyhedra we choose when attempting to decompose the point $\hat{x}$, as long as it is a relaxation of the original problem. We can pick various polyhedra and in trying to decompose them into convex combinations of the extreme points of different polyhedra, we would in turn generate different types of decomposition cuts. In the same way nested pricing diversifies our collection of inner points, these ideas could help diversify the orientation of the cuts found. This idea can also be parallelized since the computation of each decomposition is independent.

Simplex-Based Cutting Planes in Price-and-Cut As mentioned in Section 4.2, we have integrated all of the cut generators present in CGL into DIP except for those which depend on the use of a simplex-based solver. Since the point we give to the cut generator has come from a mapping to the compact space, we only have a primal solution and no basis. For cuts like Gomory Mixed Integer, the separation routine depends on the existence of a basis. To provide one, we are considering the use of a crossover step similar to what is used by interior point methods when crossing over to a simplex-based method.

Appendix A

Detailed Tables of Results

	CPX10.2					DIP-CPM				
Instance	**Time**	**LB**	**UB**	**Gap**	**Nodes**	**Time**	**LB**	**UB**	**Gap**	**Nodes**
I1	0.00	-173.00	-173.00	OPT	0	0.02	-173.00	-173.00	OPT	15
I10	T	-61485.54	-61456.00	0.05%	135090	T	-61484.92	∞	∞	47807
I11	T	-73796.87	-73776.00	0.03%	128985	T	-73796.44	∞	∞	45295
I12	T	-86099.80	-86088.00	0.01%	106922	T	-86099.36	∞	∞	35160
I13	T	-98448.03	-98427.00	0.02%	94485	T	-98447.54	∞	∞	29896
I2	0.01	-364.00	-364.00	OPT	0	0.01	-364.00	-364.00	OPT	3
I3	1.17	-1602.00	-1602.00	OPT	6243	23.23	-1602.00	-1602.00	OPT	19582
I4	15.71	-3597.00	-3597.00	OPT	80438	T	-3600.12	∞	∞	107716
I5	0.01	-3905.90	-3905.70	0.01%	0	0.01	-3905.70	-3905.70	OPT	3
I6	0.14	-4799.30	-4799.30	OPT	1	0.07	-4799.30	-4799.30	OPT	59
I7	T	-24604.24	-24584.00	0.08%	212311	T	-24602.79	∞	∞	73433
I8	T	-36902.51	-36868.00	0.09%	200759	T	-36901.40	∞	∞	64035
I9	T	-49192.89	-49161.00	0.06%	156984	T	-49192.16	∞	∞	60411
INST01	T	-10747.51	-10702.00	0.43%	530376	T	-10745.94	∞	∞	96236
INST02	T	-13608.89	-13597.00	0.09%	611249	T	-13612.15	∞	∞	100130
INST03	T	-10976.54	-10935.00	0.38%	481301	T	-10974.37	∞	∞	94806
INST04	T	-14472.06	-14423.00	0.34%	324323	T	-14469.89	∞	∞	83122
INST05	T	-17074.63	-17044.00	0.18%	397541	T	-17073.04	∞	∞	84945
INST06	T	-16852.72	-16818.00	0.21%	426871	T	-16851.42	∞	∞	80697
INST07	T	-16456.43	-16398.00	0.36%	335416	T	-16454.86	∞	∞	85158
INST08	T	-17530.44	-17487.00	0.25%	372261	T	-17529.08	∞	∞	86583
INST09	T	-17777.24	-17740.00	0.21%	397692	T	-17775.97	∞	∞	87234
INST11	T	-19459.70	-19417.00	0.22%	329214	T	-19458.41	∞	∞	88541
INST12	T	-21754.47	-21716.00	0.18%	289646	T	-21753.05	∞	∞	76864
INST13	T	-21590.57	-21574.00	0.08%	225896	T	-21589.88	∞	∞	36381
INST14	T	-32885.70	-32870.00	0.05%	157631	T	-32885.12	∞	∞	25402
INST15	T	-39173.60	-39157.00	0.04%	138190	T	-39173.01	∞	∞	21484
INST16	T	-43378.19	-43354.00	0.06%	113757	T	-43377.43	∞	∞	18590
INST17	T	-54371.33	-54356.00	0.03%	110296	T	-54370.98	∞	∞	16018
INST18	T	-60478.08	-60462.00	0.03%	96441	T	-60477.67	∞	∞	21774
INST19	T	-64942.99	-64926.00	0.03%	120924	T	-64942.53	∞	∞	19293
INST20	T	-75626.52	-75607.00	0.03%	84953	T	-75626.12	∞	∞	16156

Table A.1: MMKP: CPX10.2 vs CPM (Detailed Table)

	DIP-PC					DIP-DC				
Instance	**Time**	**LB**	**UB**	**Gap**	**Nodes**	**Time**	**LB**	**UB**	**Gap**	**Nodes**
I1	0.04	-173.00	-173.00	OPT	14	0.14	-173.00	-173.00	OPT	7
I10	T	-61636.62	-55102.00	11.86%	74	T	-61485.90	-61395.00	0.15%	158
I11	T	-74025.27	-65948.00	12.25%	110	T	-73797.39	-73697.00	0.14%	108
I12	T	-86271.06	-79934.00	7.93%	92	T	-86100.15	-86014.00	0.10%	94
I13	T	-98708.40	-88217.00	11.89%	119	T	-98448.31	-98329.00	0.12%	96
I2	0.05	-364.00	-364.00	OPT	15	0.05	-364.00	-364.00	OPT	3
I3	T	-1618.16	-1601.00	1.07%	1677	T	-1612.95	-1601.00	0.75%	812
I4	T	-3631.61	-3454.00	5.14%	463	T	-3617.59	-3590.00	0.77%	658
I5	0.13	-3905.70	-3905.70	OPT	11	0.05	-3905.70	-3905.70	OPT	3
I6	T	-4812.81	-4799.30	0.28%	129	0.63	-4799.30	-4799.30	OPT	55
I7	T	-24677.94	-21587.00	14.32%	87	T	-24605.98	-24584.00	0.09%	300
I8	T	-37021.81	-32658.00	13.36%	85	T	-36903.19	-36828.00	0.20%	240
I9	T	-49358.35	-44585.00	10.71%	71	T	-49193.47	-49099.00	0.19%	193
INST01	T	-10782.59	-9803.00	9.99%	91	T	-10750.84	-10676.00	0.70%	423
INST02	T	-13670.19	-12729.00	7.39%	152	T	-13625.00	-13564.00	0.45%	459
INST03	T	-11017.95	-10612.00	3.83%	132	T	-10981.36	-10889.00	0.85%	405
INST04	T	-14506.40	-13497.00	7.48%	82	T	-14475.27	-14410.00	0.45%	404
INST05	T	-17132.11	-15542.00	10.23%	90	T	-17076.04	-16971.00	0.62%	398
INST06	T	-16882.29	-15373.00	9.82%	84	T	-16854.19	-16791.00	0.38%	363
INST07	T	-16518.21	-14270.00	15.75%	83	T	-16458.15	-16356.00	0.62%	350
INST08	T	-17582.56	-15762.00	11.55%	96	T	-17531.60	-17452.00	0.46%	344
INST09	T	-17828.76	-15471.00	15.24%	159	T	-17778.59	-17708.00	0.40%	354
INST11	T	-19508.10	-18070.00	7.96%	68	T	-19461.01	-19386.00	0.39%	322
INST12	T	-21833.21	-20235.00	7.90%	63	T	-21755.86	-21665.00	0.42%	332
INST13	T	-21661.71	-21036.00	2.97%	117	T	-21592.40	-21563.00	0.14%	120
INST14	T	-32980.39	-31744.00	3.89%	139	T	-32887.01	-32856.00	0.09%	66
INST15	T	-39267.40	-37965.00	3.43%	118	T	-39174.44	-39137.00	0.10%	53
INST16	T	-43478.36	-42545.00	2.19%	111	T	-43379.15	-43355.00	0.06%	52
INST17	T	-54523.77	-53408.00	2.09%	165	T	-54371.97	-54324.00	0.09%	15
INST18	T	-60661.12	-58089.00	4.43%	205	T	-60478.53	-60442.00	0.06%	49
INST19	T	-65103.37	-63130.00	3.13%	120	T	-64943.67	-64915.00	0.04%	33
INST20	T	-75784.90	-73539.00	3.05%	106	T	-75626.97	-75594.00	0.04%	16

Table A.2: MMKP: PC vs DC (Detailed Table)

	DIP-PC-M2					DIP-PC-MM				
Instance	**Time**	**LB**	**UB**	**Gap**	**Nodes**	**Time**	**LB**	**UB**	**Gap**	**Nodes**
I1	0.16	-173.00	-173.00	OPT	11	0.08	-173.00	-173.00	OPT	10
I10	T	-61660.98	-57633.00	6.99%	301	T	-61710.03	-61325.00	0.63%	50
I11	T	-73991.84	-66567.00	11.15%	524	T	-74071.70	-73633.00	0.60%	86
I12	T	-86309.57	-77473.00	11.41%	429	T	-86576.93	-85902.00	0.79%	51
I13	T	-98709.15	-86851.00	13.65%	165	T	-98831.24	-98321.00	0.52%	60
I2	0.45	-364.00	-364.00	OPT	14	0.14	-364.00	-364.00	OPT	11
I3	T	-1619.88	-1601.00	1.18%	16973	T	-1618.63	-1601.00	1.10%	3702
I4	T	-3630.73	-3519.00	3.18%	18826	T	-3631.16	-3587.00	1.23%	929
I5	0.14	-3905.70	-3905.70	OPT	1	0.07	-3905.70	-3905.70	OPT	1
I6	483.53	-4799.30	-4799.30	OPT	381	T	-4811.20	-4799.30	0.25%	103
I7	T	-24660.72	-23520.00	4.85%	1265	T	-24676.73	-24439.00	0.97%	112
I8	T	-36956.59	-33661.00	9.79%	417	T	-37007.99	-36761.00	0.67%	91
I9	T	-49327.29	-44611.00	10.57%	500	T	-49385.20	-49027.00	0.73%	95
INST01	T	-10783.00	-10176.00	5.97%	489	T	-10791.15	-10594.00	1.86%	96
INST02	T	-13677.96	-12749.00	7.29%	247	T	-13674.36	-13441.00	1.74%	181
INST03	T	-11019.75	-9845.00	11.93%	9217	T	-11024.52	-10850.00	1.61%	73
INST04	T	-14512.80	-13558.00	7.04%	6578	T	-14525.21	-14302.00	1.56%	75
INST05	T	-17132.74	-15741.00	8.84%	745	T	-17122.34	-16935.00	1.11%	217
INST06	T	-16912.78	-15408.00	9.77%	786	T	-16905.11	-16673.00	1.39%	120
INST07	T	-16498.17	-15167.00	8.78%	1614	T	-16522.37	-16322.00	1.23%	81
INST08	T	-17584.13	-16207.00	8.50%	343	T	-17584.83	-17348.00	1.37%	102
INST09	T	-17827.13	-16433.00	8.48%	219	T	-17832.41	-17675.00	0.89%	81
INST11	T	-19521.25	-17955.00	8.72%	1948	T	-19496.31	-19278.00	1.13%	98
INST12	T	-21790.44	-20418.00	6.72%	279	T	-21825.86	-21603.00	1.03%	102
INST13	T	-21648.15	-21005.00	3.06%	986	T	-21675.42	-21511.00	0.76%	134
INST14	T	-32955.51	-31789.00	3.67%	158	T	-32983.86	-32812.00	0.52%	111
INST15	T	-39256.07	-38183.00	2.81%	266	T	-39414.68	-39111.00	0.78%	100
INST16	T	-43500.51	-42229.00	3.01%	51	T	-43528.10	-43311.00	0.50%	49
INST17	T	-54509.06	-53357.00	2.16%	94	T	-54509.42	-54297.00	0.39%	59
INST18	T	-60634.96	-59096.00	2.60%	68	T	-60638.95	-60389.00	0.41%	100
INST19	T	-65141.30	-62651.00	3.97%	118	T	-65151.86	-64855.00	0.46%	75
INST20	T	-75880.68	-72922.00	4.06%	150	T	-76288.35	-75581.00	0.94%	44

Table A.3: MMKP: PC-M2 vs PC-MM (Detailed Table)

Bibliography

[1] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. CONCORDE TSP solver. http://www.tsp.gatech.edu/concorde.html.

[2] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. TSP cuts which do not conform to the template paradigm. In *Computational Combinatorial Optimization*, pages 261–303. Springer, 2001.

[3] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. Implementing the Dantzig-Fulkerson-Johnson algorithm for large traveling salesman problems. *Mathematical Programming*, 97:91–153, 2003.

[4] David L. Applegate, Robert E. Bixby, Vasek Chvatal, and William J. Cook. *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics)*. Princeton University Press, January 2007.

[5] P. Augerat, J. M. Belenguer, E. Benavent, A. Corberán, D. Naddef, and G. Rinaldi. Computational Results with a Branch and Cut Code for the Capacitated Vehicle Routing Problem. Technical Report 949-M, Université Joseph Fourier, Grenoble, France, 1995.

[6] E. Balas, S. Ceria, and G. Cornuéjols. Mixed 0-1 programming by lift-and-project in a branch-and-cut framework. *Management Science*, 42:1229–1246, 1996.

[7] F. Barahona and R. Anbil. The volume algorithm: Producing primal solutions with a subgradient method. *Mathematical Programming*, 87:385–399, 2000.

[8] F. Barahona and D. Jensen. Plant location with minimum inventory. *Mathematical Programming*, 83:101–111, 1998.

[9] C. Barnhart, C. A. Hane, and P. H. Vance. Using branch-and-price-and-cut to solve origin-destination integer multi-commodity flow problems. *Operations Research*, 48:318–326, 2000.

[10] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance. Branch and price: Column generation for solving huge integer programs. *Operations Research*, 46:316–329, 1998.

[11] J.E. Beasley. Lagrangean relaxation. In C.R. Reeves, editor, *Modern Heuristic Techniques for Combinatorial Optimization*. Wiley, 1993.

[12] Robert E. Bixby. Solving real-world linear programs: A decade and more of progress. *Oper. Res.*, 50(1):3–15, 2002.

[13] Robert E. Bixby, Mary Fenelon, Zonghao Gu, Edward Rothberg, and Roland Wunderling. MIP: Theory and practice - closing the gap. In *Proceedings of the 19th IFIP TC7 Conference on System Modelling and Optimization*, pages 19–50, Deventer, The Netherlands, The Netherlands, 2000. Kluwer, B.V.

[14] Robert E. Bixby and Edward Rothberg. Progress in computational mixed integer programming - a look back from the other side of the tipping point. *Annals OR*, 149(1):37–41, 2007.

[15] R. Bornd́orfer, C. Ferreira, and A. Martin. Matrix decomposition by branch-and-cut. Technical Report TR 97-04, Konrad-Zuse-Zentrum für Informationstechnik Berlin, 1997.

[16] O. Briant, C. Lemaréchal, Ph. Meurdesoif, S. Michel, N. Perrot, and F. Vanderbeck. Comparison of bundle and classical column generation. *Mathematical Programming*, 113(2):299–344, 2008.

[17] A. Caprara, M. Fischetti, and P. Toth. Algorithms for the set covering problem. *Annals of Operations Research*, 98:353–371, 2000.

[18] P. Carraresi, A. Frangioni, and M. Nonato. Applying bundle methods to optimization of polyhedral functions: An applications-oriented development. Technical Report TR-96-17, Universitá di Pisa, 1996.

[19] Lei Chen, Shahadat Khan, Kin F. Li, and Eric G. Manning. Building an adaptive multimedia system using the utility model. In *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 289–298, London, UK, 1999. Springer-Verlag.

[20] C. Cordier, H. Marchand, R. Laundy, and L.A. Wolsey. bc-opt: A branch-and-cut code for mixed integer programs. *Mathematical Programming*, 86:335–353, 1999.

[21] H. Crowder, E. L. Johnson, and M. W. Padberg. Solving large scale zero-one linear programming problems. *Operations Research*, 31:803–834, 1983.

[22] J. Czyzyk, M. Mesnier, and J. Moré. The NEOS server. *IEEE Journal on Computational Science and Engineering*, 5:68–75, 1998.

[23] G. B. Dantzig and P. Wolfe. Decomposition principle for linear programs. *Operations Research*, 8(1):101–111, 1960.

[24] G.B. Dantzig and R.H. Ramser. The truck dispatching problem. *Management Science*, 6:80–91, 1959.

[25] M. Poggi de Aragão and E. Uchoa. Integer program reformulation for robust branch-and-cut-and-price. Working paper, Pontifíca Universidade Católica do Rio de Janeiro, 2004. Available from http://www.inf.puc-rio.br/˜uchoa/doc/rbcp-a.pdf.

[26] M. de Moraes Palmeira, A. Lucena, and O. Porto. A Relax and Cut Algorithm for Quadratic Knapsack Problem. Technical report, Universidade Federal do Rio de Janeiro, 1999.

[27] J. Desrosiers and M. L´ubbecke. *Column Generation*, chapter A Primer in Column Generation, pages 1–32. Springer, 2005.

[28] ILOG CPLEX Division. `http://www.cplex.com`.

[29] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213, January 2002.

[30] I. Duff, A. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices (Monographs on Numerical Analysis)*. Oxford University Press, 1987.

[31] M.L. Fisher. The Lagrangian relaxation method for solving integer programming problems. *Management Science*, 27:1–18, 1981.

[32] M.L. Fisher. Optimal solution of vehicle routing problems using minimum k-trees. *Operations Research*, 42:626–642, 1994.

[33] M. Galati. DIP, 2009. Available from `http://www.coin-or.org/projects/Dip.xml`.

[34] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[35] B. Golden, S. Raghavan, and E. Wasil. *The Vehicle Routing Problem: Latest Advances and New Challenges*. Springer-Verlag, 2008.

[36] J. Gondzio and R. Sarkissian. Column generation with a primal-dual method. Technical report, University of Geneva, Logilab, HEC Geneva, 1996.

[37] M. Grötschel and M. Padberg. On the symmetric travelling salesman problem I: Inequalities. *Mathematical Programming*, 16:265–280, 1979.

[38] M. Grötschel and M. Padberg. On the symmetric travelling salesman problem II: Lifting theorems and facets. *Mathematical Programming*, 16:281–302, 1979.

[39] Grötschel, M. and Lovász, L. and Schrijver, A. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1:169–197, 1981.

[40] M. Guignard. Efficient cuts in Lagrangean "relax-and-cut" schemes. *European Journal of Operational Research*, 105:216–223, 1998.

[41] M. Guignard. Lagrangean relaxation. *Top*, 11:151–228, 2003.

[42] M. Held and R. M. Karp. The traveling salesman problem and minimum spanning trees. *Operations Research*, 18:1138–1162, 1970.

[43] M. Hifi. Benchmark Instances for Multi-Dimensional Multi-Choice Knapsack. Available from `http://www.laria.u-picardie.fr/hifi/OR-Benchmark/MMKP`.

[44] R. Jonker. A shortest path algorithm for dense and sparse linear assignment problems. *Computing*, 38:325–340, 1987.

[45] R. Jonker and A. Volegenant. Linear Assignment Problem Solver, 1987. Available from `http://www.magiclogic.com/assignment.html`.

[46] M. Jünger and S. Thienel. Introduction to ABACUS—a branch-and-cut system. *Operations Research Letters*, 22:83–95, 1998.

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

[48] L. Kopman. *A New Generic Separation Routine and Its Application In a Branch and Cut Algorithm for the Capacitated Vehicle Routing Problem.* PhD thesis, Cornell University, May 1999.

[49] L. Ladányi. CoinUtils. Available from `http://www.coin-or.org/`.

[50] A. H. Land and A. G. Doig. An automatic method for solving discrete programming problems. *Econometrica*, 28:497–520, 1960.

[51] G. Laporte, Y. Nobert, and M. Desrouchers. Optimal routing with capacity and distance restrictions. *Operations Research*, 33:1050–1073, 1985.

[52] J. T. Linderoth. *Topics in Parallel Integer Optimization.* PhD thesis, Georgia Institute of Technology, 1998.

[53] R. Lougee-Heimer. The Common Optimization INterface for Operations Research. *IBM Journal of Research and Development*, 47:57–66, 2003.

[54] R. Lougee-Heimer. CGL, 2009. Available from `http://www.coin-or.org/projects/Cgl.xml`.

[55] M. E. Lübbecke and J. Desrosiers. Selected topics in column generation. Technical Report 008-2004, Technische Universität Berlin, 2004.

[56] A. Lucena. Non-delayed relax-and-cut algorithms. Working paper, Departmaneto de Administração, Universidade Federal do Rio de Janeiro, 2004.

[57] J. Lysgaard. The CVRPSEP package. Available from `http://www.hha.dk/~lys/CVRPSEP.htm`.

[58] J. Lysgaard, A.N. Letchford, and R.W. Eglese. A new branch-and-cut algorithm for the capacitated vehicle routing problem. *Mathematical Programming*, 100:423–445, 2004.

[59] S. Martello and P. Toth. *Knapsack Problems: algorithms and computer implementation*. John Wiley & Sons, Inc., USA, 1st edition, 1990.

[60] Silvano Martello, David Pisinger, and Paolo Toth. Dynamic programming and strong bounds for the 0-1 knapsack problem. *Manage. Sci.*, 45(3):414–424, 1999.

[61] A. Martin. Integer programs with block structure. Habilitation Thesis, Technical University of Berlin, Berlin, Germany, 1998.

[62] C. Martinhon, A. Lucena, and N. Maculan. A relax and cut method for the vehicle routing problem. Unpublished working paper, 2001.

[63] C. Martinhon, A. Lucena, and N. Maculan. Stronger k-tree relaxations for the vehicle routing problem. *European Journal of Operational Research*, 158:56–71, 2004.

[64] D. Miller. A matching based exact algorithm for capacitated vehicle routing problems. *ORSA Journal on Computing*, 7:1–9, 1995.

[65] M. Müller-Hannemann and A. Schwartz. Implementing weighted b-matching algorithms: Towards a flexible software design. In *Proceedings of the Workshop on Algorithm Engineering and Experimentation (ALENEX99)*, volume 1619 of *Lecture notes in computer science*, pages 18–36, Baltimore, MD, 1999. Springer-Verlag.

[66] D. Naddef and G. Rinaldi. Branch-and-cut algorithms for the capacitated VRP. In P. Toth and D. Vigo, editors, *The Vehicle Routing Problem*, pages 53–84. SIAM, 2002.

[67] H. Nagamochi, T. Ono, and T. Ibaraki. Implementing an efficient minimum capacity cut algorithm. *Mathematical Programming*, 67:325–341, 1994.

[68] G.L. Nemhauser, M.W.P. Savelsbergh, and G.S. Sigismondi. MINTO, a Mixed INTeger Optimizer. *Operations Research Letters*, 15:47–58, 1994.

[69] G.L. Nemhauser and P.H. Vance. Lifted cover facets of the 0-1 knapsack polytope with gub constraints. *Operations Research Letters*, 16:255–264, 1994.

[70] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization*. Wiley, New York, 1988.

[71] P. ́Ostergård. Cliquer - Routines for Clique Searching, 2003. Available from `http://users.tkk.fi/pat/cliquer.html`.

[72] James Ostrowski, Jeff Linderoth, Fabrizio Rossi, and Stefano Smriglio. Orbital branching. *Mathematical Programming*, 2009.

[73] M. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale traveling salesman problems. *SIAM Review*, 33:60–100, 1991.

[74] D. Pisinger. David Pisinger's optimization codes. Available from `http://www.diku.dk/hjemmesider/ansatte/pisinger/codes.html`.

[75] D. Pisinger. A minimal algorithm for the multiple-choice knapsack problem. *European Journal of Operational Research*, 83:394–410, 1995.

[76] L. Qi and D. Sun. Polyhedral method for solving three index assignment problems. Working paper.

[77] Ted K. Ralphs and Matthew V. Galati. Decomposition and dynamic cut generation in integer linear programming. *Math. Program.*, 106(2):261–285, 2006.

[78] T.K. Ralphs. *Parallel Branch and Cut for Vehicle Routing*. PhD thesis, Cornell University, May 1995.

[79] T.K. Ralphs and M. Guzelsoy. The SYMPHONY callable library for mixed-integer linear programming. In *Proceedings of the Ninth INFORMS Computing Society Conference*, pages 61–76, 2005.

[80] T.K. Ralphs, L. Kopman, W.R. Pulleyblank, and L.E. Trotter Jr. On the capacitated vehicle routing problem. *Mathematical Programming*, 94:343–359, 2003.

[81] T.K. Ralphs and L. Ladányi. *SYMPHONY Version 4.0 User's Manual*, 2004. `http://www.brandandcut.org`.

[82] J. Rouse. SAS/OR OPTMODEL Procedure, note =.

[83] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, Chichester, 1986.

[84] J. Siek. The Boost Graph Library. Available from `http://www.boost.org`.

[85] V. Chvátal. *Linear Programming*. W.H. Freeman and Company, 1983.

[86] J.M. van den Akker, C.A.J. Hurkens, and M.W.P. Savelsbergh. Time-indexed formulations for machine scheduling problems: Column generation. *INFORMS Journal on Computing*, 12:111–124, 2000.

[87] F. Vanderbeck. BapCod : a generic Branch-and-Price Code, note =.

[88] F. Vanderbeck. Lot-sizing with start-up times. *Management Science*, 44:1409–1425, 1998.

[89] F. Vanderbeck. On Dantzig-Wolfe decomposition in integer programming and ways to perform branching in a branch-and-price algorithm. *Operations Research*, 48:111–128, 2000.

[90] François Vanderbeck. Branching in Branch-and-Price: a Generic Scheme. Technical report, 2009.

[91] R.K. Watson. *Packet Networks and Optimal Admission and Upgrade of Service Level Agreements: Applying the Utility Model*. PhD thesis, University of Victoria, 2001.

[92] G. Wei and G. Yu. An improved $O(n^2 \log n)$ algorithm for the degree-constrained minimum k-tree problem. Technical report, The University of Texas at Austin, Center for Management of Operations and Logistics, 1995.

[93] L.A. Wolsey. *Integer Programming*. Wiley, New York, 1998.

[94] Y. Xu, T.K. Ralphs, L. Ladányi, and M.J. Saltzman. ALPS: A framework for implementing parallel search algorithms. In *Proceedings of the Ninth INFORMS Computing Society Conference*, pages 319–334, 2005.

[95] E. Alper Yildirim and Stephen J. Wright. Warm-start strategies in interior-point methods for linear programming. *SIAM J. on Optimization*, 12(3):782–810, 2002.