THE UNIVERSITY OF QUEENSLAND
AUSTRALIA

Towards a general formulation of lazy constraints

Robin H. Pearce
B.Sc. (Hons)

A thesis submitted for the degree of Doctor of Philosophy at The University of Queensland in 2019

School of Mathematics and Physics

Abstract

Integer programming is an important tool for optimising large-scale industrial processes and problem solving in general. Many industries benefit greatly from the use of integer programming to solve large and complex problems. While general-purpose integer program solvers are becoming increasingly powerful, most problems of interest are too difficult to solve directly, and instead require special treatment, such as a problem decomposition. The difficulty in solving an integer programming problem is directly influenced by the number of constraints, in that problems with more constraints are more difficult to solve than similar problems with fewer constraints.

In this thesis, we explore a capability that has been developed since the 1990s, but only recently became available in the high-powered commercial solvers, Gurobi, CPLEX and XPRESS, known as lazy constraints. Also referred to as branch-and-cut, lazy constraints act as a modelling tool that allows the solver to add additional constraints during the solution process. This is often in response to having started with a reduced set of constraints that no longer completely describes the problem, and thus some constraints must be returned to the problem.

This thesis attempts to provide preliminary answers to the questions: "When and where are lazy constraints likely to be useful?" and "What are the important implementation issues when using lazy constraints?" In order to do this we investigate the three main scenarios where using lazy constraints may be beneficial. Each chapter of this thesis covers one of these scenarios with example problems and implementations.

The first scenario is the most direct: models that have sets of constraints where many (or all) constraints in the set may be unnecessary. By removing these constraints, we may solve a smaller model that will solve faster than the original, but if we have removed any individual constraints that are necessary, the solutions returned will not be feasible. When a solution to the relaxed problem violates one such constraint, that constraint is added to the relaxed problem and we continue solving. When an optimal solution to the relaxed problem that does not violate any constraints of the original problem is found, we thus have an optimal solution to the original problem.

We apply this to several models for a particular problem, the Liner Shipping Fleet Repositioning Problem. In this problem, we move several ships between two locations in such a way as to minimise their movement costs by serving extra demands along the way. There are many capacity constraints in the models, however there are few scenarios where ships actually exceed their capacity. By removing the capacity constraints, the problem solves much faster, and only a very small number ($< 1\%$) are actually used.

The second scenario is for problems which benefit from Benders decomposition, a problem decomposition first described in 1962. Benders decomposition benefits greatly in most cases from the use of lazy constraints by embedding the formulation in a branch-and-cut framework. Benders decomposition involves removing a number of variables and constraints from a problem, and instead approximating their contribution to the objective through some auxiliary variables. These variables are controlled through additional constraints, called Benders cuts.

When solving a problem using Benders decomposition, we need to add multiple rounds of Benders cuts. Originally, one would solve the master problem to or near optimality, one would add cuts, and then one would solve the master problem again. For most problems, the master problem takes the longest to solve, with the cut-generation process being very fast or even trivial. With the use of lazy constraints, one may add Benders cuts during the solution of the master problem, cutting out unnecessary repetition.

We apply Benders decomposition to a number of problems, such as the Uncapacitated Facility Location Problem, and demonstrate how large and difficult problems benefit from such a treatment. We also cover some implementation details and other ways of improving the efficiency of Benders decomposition. We show the most important aspects of Benders decomposition are the disaggregation of the sub-problem and the use of lazy constraints.

The third scenario is similar to the first: models that have exponentially sized sets of constraints. In these problems, it is especially likely that many of these constraints will be unnecessary. We demonstrate the power of handling these sets lazily for the Travelling Salesman Problem and two puzzles: Anne Bonney (the Pieces of 8) and the Fillomino Puzzle. We also note the interesting parallels between the lazy formulations for these problems and Benders decomposition.

Finally, we draw from the experience of the earlier chapters to provide initial answers to our original questions. We also note some interesting areas for future research that show promise. The most significant of these is the combination of lazy constraints and/or Benders decomposition with other decomposition strategies, especially Dantzig-Wolfe decomposition.

Declaration by author

This thesis is composed of my original work, and contains no material previously published or written by another person except where due reference has been made in the text. I have clearly stated the contribution by others to jointly-authored works that I have included in my thesis.

I have clearly stated the contribution of others to my thesis as a whole, including statistical assistance, survey design, data analysis, significant technical procedures, professional editorial advice, financial support and any other original research work used or reported in my thesis. The content of my thesis is the result of work I have carried out since the commencement of my higher degree by research candidature and does not include a substantial part of work that has been submitted to qualify for the award of any other degree or diploma in any university or other tertiary institution. I have clearly stated which parts of my thesis, if any, have been submitted to qualify for another award.

I acknowledge that an electronic copy of my thesis must be lodged with the University Library and, subject to the policy and procedures of The University of Queensland, the thesis be made available for research and study in accordance with the Copyright Act 1968 unless a period of embargo has been approved by the Dean of the Graduate School.

I acknowledge that copyright of all material contained in my thesis resides with the copyright holder(s) of that material. Where appropriate I have obtained copyright permission from the copyright holder to reproduce material in this thesis and have sought permission from co-authors for any jointly authored works included in the thesis.

Publications included in this thesis

1. [1] **Robin H. Pearce** and Michael A. Forbes, The Fillomino Puzzle, *INFORMS Transactions on Education*, 17 (2), pp. 85-89, 2017
2. [2] **Robin H. Pearce** and Michael Forbes, Disaggregated Benders decomposition and branch-and-cut for solving the budget-constrained dynamic uncapacitated facility location and network design problem, *European Journal of Operational Research*, 270 (1), pp. 78-88, 2018
3. [3] **Robin H. Pearce** and Michael Forbes, Disaggregated Benders decomposition for solving a network maintenance scheduling problem, *Journal of the Operational Research Society*, 70 (6), pp. 941-953, 2019

Submitted manuscripts included in this thesis

No manuscripts submitted for publication

Other publications during candidature

Peer-reviewed papers

1. [1] **Robin H. Pearce** and Michael A. Forbes, The Fillomino Puzzle, *INFORMS Transactions on Education*, 17 (2), pp. 85-89, 2017
2. [2] **Robin H. Pearce** and Michael Forbes, Disaggregated Benders decomposition and branch-and-cut for solving the budget-constrained dynamic uncapacitated facility location and network design problem, *European Journal of Operational Research*, 270 (1), pp. 78-88, 2018
3. [3] **Robin H. Pearce** and Michael Forbes, Disaggregated Benders decomposition for solving a network maintenance scheduling problem, *Journal of the Operational Research Society*, 70 (6), pp. 941-953, 2019

Conference abstracts

1. [4] **Robin H. Pearce**, Alexis Tyler and Michael Forbes, Extending the MIP toolbox to crack the Liner Shipping Fleet Repositioning problem, *Biarri Applied Mathematics conference 2016*.
2. [5] **Robin H. Pearce**, Solving Uncapacitated Facility Location and Network Design problems with disaggregated Benders decomposition, *Proceedings of INFORMS Annual Meeting 2017*.

Contributions by others to the thesis

No contributions by others

Statement of parts of the thesis submitted to qualify for the award of another degree

No works submitted towards another degree have been included in this thesis.

Research involving human or animal subjects

No animal or human subjects were involved in this research.

Acknowledgments

I extend my greatest thanks to my principal supervisor Michael Forbes for his support over the last three years. It was always interesting and educational, and I have certainly learned a great deal about mathematics and myself. I am especially grateful for your encouragement and ability to lift my spirits any time I doubted myself.

I would also like to thank Drs Ian Wood, Anthony Roberts, Thomas Taimre and Yoni Nazarathy for their detailed comments during the confirmation process and for encouraging me to expand my reading of the literature. It was a formative experience which prepared me for the hard yards remaining in my study.

My time as a student was further enriched by the extra collaborations and conversations with other mathematicians and researchers at the university. In particular, Mahboobeh Moghaddam for inviting me to collaborate on her project, Yannik Rist, Nathan D'Addio, Tom Anderson and the students of the last few MATH4202 courses with whom I have had many interesting discussions. The many opportunities to convey and articulate my understanding have helped to cement many complex concepts in my mind, and also to find and close gaps in my knowledge.

Studying so effectively for this long was only possible with a stable situation at home. For this I thank my housemates Matthew Habermann and Marvin Tas, for being team players and creating a warm, welcoming home that provided an excellent foundation for all our pursuits. It is a great place for entertaining, and we have had many memorable parties with our friends which made a big difference to all our lives.

The most important support I received was that from my friends and family, who put up with me when my mind was wholly consumed by my studies and I was emotionally absent. Christopher Falconer, Jessica Kempe, Trent Salter, Fiona Salter, Megan Bertramelli, Lewis Kelly, Marvin Tas, Samantha Edwards and Matthew Edwards, you are the absolute best. All the nights of pub trivia, bad movies, murder in the dark and other shenanigans have made the last few years the best of my life so far, and I hope we all remain good friends for many years to come. Mum and Dad, I could not have done this without your support and encouragement, so a big thank you to you both!

Robin H. Pearce

Brisbane 2018

Financial support

This research was supported by an Australian Government Research Training Program Scholarship

Keywords

Branch-and-cut, lazy constraints, Benders decomposition, network design

Australian and New Zealand Standard Research Classifications (ANZSRC)

ANZSRC code: 010206, Operations Research, 100%

Fields of Research (FoR) Classification

FoR code: 0102, Applied Mathematics, 100%

Contents

Abstract ii

Contents **ix**

List of figures **xii**

List of tables **xiii**

List of abbreviations and symbols **xv**

1 Introduction **1**

1.1 Linear programming 2

1.1.1 The revised simplex method 4

1.1.2 Duality theory 6

1.1.3 Interior-point methods 8

1.2 Integer programming 8

1.2.1 Branch-and-bound 10

1.2.2 Problem decompositions 12

1.2.3 Constraint programming 13

1.2.4 Integer program solvers 13

1.2.5 Lazy constraints 14

2 Branch and Cut **19**

2.1 Introduction 19

2.2 The Liner-Shipping Fleet-Repositioning Problem 20

2.2.1 Problem description and model formulation 21

2.2.2 Reduced MIP 24

2.2.3 Revised formulation 27

2.2.4 Model reformulation 28

2.2.5 Branch-and-price 30

2.2.6 Branch-and-cut 31

2.2.7 Results 32

2.2.8 Discussion . . . 34

3 Benders Decomposition **37**

3.1 History . . . 38

3.2 Theory . . . 39

3.2.1 Optimality . . . 40

3.2.2 Feasibility . . . 41

3.2.3 Procedure . . . 43

3.3 The Uncapacitated Facility Location Problem . . . 45

3.3.1 Implementation details . . . 48

3.3.2 Results . . . 49

3.3.3 Cut selection for the UFL . . . 51

3.4 Pareto-optimality . . . 53

3.4.1 Pareto-optimality of the UFL Benders cuts . . . 53

3.5 Solving a network maintenance scheduling problem . . . 57

3.5.1 Introduction . . . 57

3.5.2 Problem definition . . . 59

3.5.3 Disaggregated Benders decomposition and lazy constraints . . . 60

3.5.4 Results . . . 69

3.5.5 Conclusion . . . 76

3.6 Discussion of Paper: Disaggregated Benders Decomposition for Solving a Network Maintenance Scheduling Problem . . . 77

3.7 Branch-and-cut for solving the DUFLNDP . . . 78

3.7.1 Introduction . . . 78

3.7.2 Model Formulation . . . 80

3.7.3 Disaggregation and Benders decomposition . . . 82

3.7.4 Warm start . . . 89

3.7.5 Results . . . 92

3.7.6 Conclusion . . . 99

3.7.7 Acknowledgements . . . 99

3.8 Generalised UFL and Network Design Problem . . . 100

3.8.1 Formulation of the GUFLNDP . . . 101

3.8.2 Benders decomposition for the GUFLNDP . . . 103

3.8.3 Numerical example . . . 109

3.8.4 Applicability of the GUFLNDP formulation . . . 116

3.9 Discussion . . . 117

4 Lazy Formulations **119**

4.1 The Travelling Salesman Problem . . . 119

4.1.1 Compact formulation . . . 121

4.1.2 Benders decomposition 123
4.2 Anne Bonney (the Pieces of 8) 124
4.2.1 Lazy formulation 125
4.2.2 Compact formulation 128
4.2.3 Benders decomposition 131
4.3 The Fillomino Puzzle 132
4.4 Paper: Puzzle - The Fillomino Puzzle 134
4.5 Discussion of paper: Puzzle - The Fillomino Puzzle 141

5 Conclusion **143**
5.1 Future directions 144

Bibliography **147**

List of figures

1.1 Graphical representation of an integer program in two variables 10
1.2 Graphical representation of branching decision 11
1.3 Branch-and-bound tree 12
1.4 Graphical representation of lazy constraints in two variables 15

2.1 LSFRP: time-space graph of a service 22
2.2 LSFRP: Liner shipping network 23
2.3 LSFRP: Demand splitting 29

3.1 Number of papers on Benders decomposition per year according to Clarivate Web of Science [6] 37
3.2 Example of UFL problem 47
3.3 UFL: Cut comparison 52
3.4 UFL: A dominated cut 53
3.5 HVCC: Example of placing cuts on bottlenecks 63
3.6 HVCC: A layered network example 63
3.7 HVCC: Number of branch-and-bound nodes explored for instance set 1 74
3.8 HVCC: Number of branch-and-bound nodes explored for instance sets 2 and 3 74
3.9 DUFLNDP: Solution time of different implementations 93
3.10 DUFLNDP: Comparison of budget cover cuts 97
3.11 DUFLNDP: Comparison of analytic cuts for warm starts and main phase 97
3.12 DUFLNDP: Comparison of different branching priorities 98
3.13 Network for GUFLNDP numerical example 109

4.1 A solution to a 6-node TSP with sub-cycles 121
4.2 Pieces of 8: example puzzle 125
4.3 Pieces of 8: Example of correcting a puzzle solution using lazy constraints or Benders decomposition 127
4.4 Fillomino: Example of puzzle 135

List of tables

2.1 LSFRP results 33

3.1 UFL results 50
3.2 HVCC: Comparison of MIP and DBD on simulated data 70
3.3 HVCC: Summary of comparison between MIP and DBD 71
3.4 HVCC: Effect of saving solutions to sub-problems 71
3.5 HVCC: Number of flow problems solved and recalled 72
3.6 HVCC: Benders decomposition with and without initial cuts 72
3.7 HVCC: Benders decomposition with and without user heuristics and/or warm start . . . 73
3.8 HVCC: 2010 real-world results 75
3.9 HVCC: 2011 real-world results 75
3.10 DUFLNDP: Instance sizes 93
3.11 DUFLNDP: Comparison of previous results, MIP and two Benders implementations on existing network instances 94
3.12 DUFLNDP: Comparison of previous results, MIP and two Benders implementations on new network instances 95
3.13 DUFLNDP: Number of Benders cuts added by different implementations 96
3.14 Comparison of solution times for different levels of disaggregation 99

4.1 TSP: results 124
4.2 Pieces of 8: Lazy formulation results 128
4.3 Pieces of 8: Compact formulation results 130
4.4 Pieces of 8: Benders decomposition results 131
4.5 Fillomino: Runtime results 139
4.6 Fillomino: Problem size results 139

List of abbreviations and symbols

Abbreviations	
BDSP	Benders dual sub-problem
BMP	Benders master problem
BOC	Benders optimality cut
BSP	Benders sub-problem
CP	Constraint programming
DUFLNDP	Dynamic Uncapacitated Facility Location and Network Design Problem
GUFLNDP	Generalised Uncapacitated Facility Location and Network Design Problem
IIS	Irreducible infeasible subsystem
INFORMS	Institute for Operations Research and Management Science
IP	Integer Program
LP	Linear Program
LSFRP	Liner-Shipping Fleet-Repositioning Problem
MaxTF-FAO	Maximum Total Flow with Flexible Arc Outages (problem)
MIP	Mixed-Integer Program
MIS	Minimal infeasible subsystem
PDPTW	Pick-up and Delivery Problem with Time Windows
RHS	Right-hand side
RMP	Relaxed master problem
SGM	Shifted geometric mean
SOS	Sail-on-service
TSP	Travelling Salesman Problem
TUFLP	Two-Level Uncapacitated Facility Location Problem
UFL	Uncapacitated Facility Location (problem)
VLSI	Very-large-scale integration

Chapter 1

Introduction

> If I have seen further it is by
> standing on the shoulders of giants
>
> Isaac Newton

Operations Research is a relatively new field of mathematics, but it is one of the fastest-growing and arguably most important fields of the modern world. Strongly related to Management Science and Data Analytics, Operations Research is primarily focussed on finding high-quality solutions to optimisation problems, often in industrial settings. The importance of Operations Research as a field is best described by the Institute for Operations Research and Management Science (INFORMS) [7]:

> Operations Research and Analytics are proven scientific mathematical processes that enable organizations to turn complex challenges into substantial opportunities by transforming data into information, and information into insights that save lives, save money and solve problems.

Having celebrated its 70th birthday recently, the simplex method is one of the most widely used algorithms today and is the cornerstone of linear and integer programming. Linear and integer programming are important tools for modelling optimisation problems and are critical to the operation of large-scale industrial processes. On its website, the leading optimisation software package Gurobi boasts that more than 1600 companies choose to use its tools, and lists a number of the more prominent companies. As of November 2018, that list contains 28 companies from industries as diverse as natural resources, technology, hospitality, defence, utilities, logistics and sport.

This thesis is concerned with a modelling technique called *lazy constraints*, used for solving large and difficult problems. This technique is applicable to a wide range of problems from many industries, and as such is important to understand and warrants continued development. In this thesis, we attempt to provide preliminary answers to the questions: “When and where are lazy constraints likely to be useful?” and “What are the important implementation issues when using lazy constraints?” In this

chapter, we review the foundational theory upon which this thesis is based, before answering the first question and providing the layout of the remainder of the thesis.

1.1 Linear programming

Linear programming is the foundation of much modern industrial optimisation, and the topic of linear programming is a relatively new one, less than 100 years old. Although the first apparent description of a linear program (also referred to as an LP) was given by Fourier [8], and three other papers from the early 20th century were published on the topic, none of these definitively described a solution method, and thus "sparked zero interest on the part of other mathematicians" [9]. It was not until George Dantzig formulated the simplex method in 1947, and subsequently published further results in 1949 and the 1950s, that the topic of linear programming began to grow.

In his reflections on the origins of the simplex method, Dantzig notes a number of important factors that aided the rapid uptake of linear programming as a practical tool [9]. The first is that the simplex method is the first algorithm useful for solving systems of linear *in*equalities — rather than equalities — with more than three variables. Many other ideas that had developed during World War II "had never found expression" [9], but suddenly became tractable when a solution method was presented. Within two years of Dantzig first presenting his results, there was a sizeable literature from mathematicians, economists and statisticians in the new fields of Operations Research and Management Science.

The second important factor was the development of digital computers in the late 1940s and early 1950s. Dantzig writes:

> The computer became *the* tool that made the application of linear programming possible. Everywhere we looked, we found practical applications that no one earlier could have posed seriously as optimization problems because solving them by hand computation would have been out of the question. By good luck, clever algorithms in conjunction with computer development gave early promise that linear programming would become a practical science.

While the advent of digital computers represented a boon to linear programming, it was not until the mid-1950s that it became practical to apply it to real-world problems. Robert Bixby reported that the first computers to handle LPs were cumbersome to work with, and the majority of the total computation time was spent "manually feeding cards into the [computer]" [10]. In 1954 an LP was solved on an IBM 701, considered by some to be the "first real 'scientific computer'". From this point onwards, it became possible to solve larger and larger LPs because of algorithmic advancements and computational developments.

The first description of the simplex method was a tableau method, suitable for hand computations but inefficient to implement on a computer, especially those that existed at the time. By 1954, there were many studies covering the *revised simplex method* [11], a generalisation of the tableau-based

method using linear algebra notation and techniques. This was more suitable for implementation on computing systems, and is one of the algorithmic advancements referred to by Bixby [10].

A search of a university engineering and sciences library will reveal several shelves of books dedicated to the topic of linear programming, many of them named as such. Most of them cover the use of the simplex and revised simplex methods, and employ similar but not identical notation. In this introduction to linear programming, we cover some of the knowledge contained in these tomes, and recommend Winston and Goldberg (2004) as an introductory text to linear and integer programming and Nocedal and Wright (2006) as a more advanced reference, as they are well-presented with clean notation. We use our own notation throughout to ensure a clean and consistent presentation of the material. The standard formulation of a linear program, introduced by Dantzig, is as follows:

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax = b \\ & x \geq 0 \end{aligned} \tag{1.1}$$

where x is an n-vector of decision variables, A is an $m \times n$ matrix, b is an m-vector and c is an n-vector. For this program, $\min c^T x$ is called the *objective function*, and the other lines are called the *constraints*. The *objective value* of a particular solution, x^p, is $c^T x^p$.

More general linear programs, perhaps with inequality constraints or without the non-negativity restriction on the decision variables, can be transformed into the standard form using known procedures, the most useful of which can be found in Chapter 13 of Nocedal and Wright (2006). When discussing optimisation problems such as the standard linear program, the following definitions are important:

Definition 1. *An optimisation problem is* ***infeasible*** *if the set of feasible solutions is empty.*

In the context of the standard linear program, that is to say there does not exist a vector x that simultaneously satisfies $Ax = b$ and $x \geq 0$. In most cases, this will be due to an inconsistency of constraints in $Ax = b$.

Definition 2. *An optimisation problem is* ***unbounded*** *if the objective function is not bounded on the feasible region.*

Again, for the standard linear program, this corresponds to a sequence of points x^k, each of which are feasible, such that $c^T x^k \to -\infty$.

Definition 3. *If an optimisation problem is neither infeasible nor unbounded, then there exists at least one* ***optimal solution****, x^*, such that for any other solution x in the set of feasible solutions, $c^T x^* \leq c^T x$.*

Note that this is not a strict inequality as there may be multiple solutions that achieve the same objective value, in which case there are multiple optimal solutions.

1.1.1 The revised simplex method

A number of topics covered in later sections depend upon consequences of the revised simplex method, particularly duality theory. We begin with a linear program of the following form:

$$\begin{aligned} \max \quad & c^T x \\ \text{s.t.} \quad & Ax \leq b \\ & x \geq 0. \end{aligned} \tag{1.2}$$

Again, any linear program can be converted to this form using standard procedures. In particular, equality constraints can be replicated using two opposing inequality constraints; greater-than-or-equal-to constraints can be converted to less-than-or-equal-to constraints by negating both sides; and minimisation can be converted to maximisation by negating the objective function. With a program in this form, we convert the constraints to equality constraints by adding *slack variables*, s. That is, $Ax \leq b$ becomes $Ax + s = b$, but we will represent this as $\tilde{A}\tilde{x} = b$ where $\tilde{A}$ and $\tilde{x}$ have been augmented with the $m \times m$ identity matrix and the vector of non-negative slack variables, s, respectively. These artificial variables represent the difference between the left- and right-hand sides of their respective constraint.

We assume that the problem is feasible, and that we have an initial feasible solution to the problem. In many cases, this solution may be $x = 0$, so that $s_i = b_i \; \forall i \in \{1, ..., m\}$. If no such solution exists, then the problem is infeasible. It is assumed, without loss of generality, that $\tilde{A}$ has full row rank, *i.e.* all constraints of the problem are linearly independent. If this is not the case, then either there are redundant constraints that may be removed without changing the feasible region, or the problem is infeasible, which is against our assumption.

The variables that are non-zero in this solution are said to be in the *basis*, and are hence referred to as *basic variables*, denoted by x^B. Note that slack variables may also be basic variables, and often are initially. Variables not in the basis are called *non-basic variables*. Since these variables are zero-valued, the result of multiplying them by any matrix columns will also be zero. As such, we construct the *basis matrix*, B, which contains those columns of the constraint matrix $\tilde{A}$ corresponding to the basic variables. The vectors p^j are the columns of $\tilde{A}$, and hence B, which correspond to the variables x_j.

Each iteration of the revised simplex method yields a *basic feasible solution* to (1.2). A basic feasible solution must have exactly m basic variables, thus making B an $m \times m$ matrix. Since $\tilde{A}$ has full row rank and B is made up of m columns of $\tilde{A}$, B is invertible. The set of basic feasible solutions is only a subset of the set of feasible solutions; however, if (1.2) has at least one optimal solution, then it has at least one basic feasible solution that is optimal. For a more detailed explanation and proof of this claim, we refer the reader to Section 13.2 of Nocedal and Wright (2006).

The constraints of problem (1.2) are now written as:

$$Bx^B = b \tag{1.3}$$

$$x^B = B^{-1}b, \tag{1.4}$$

that is, the current solution x^B is computed from the inverse of the basis matrix B^{-1} and the original constraint vector b. Similarly, the objective value may be computed by:

$$z^B = c^{B^T} x^B, \tag{1.5}$$

where c^B contains the elements of the objective vector c corresponding to the basic variables. Each iteration of the revised simplex method moves from one basic feasible solution to another by choosing a non-basic variable to enter the basis, and a basic variable to leave. The algorithm will terminate once it has found an optimal solution or that the problem is unbounded.

STEP 1: Entering variable

To determine which, if any, non-basic variable should enter the basis, we first calculate the *dual variables* $y^T = c^{B^T} B^{-1}$. Next, we compute the *reduced cost* $c'_j = c_j - y^T p^j$ for all non-basic variables x_j. The reduced cost is a measure of how much the objective value would improve by if this variable was to enter the basis.

Since the problem is one of maximisation, we require a variable with a positive reduced cost. If there are no non-basic variables with a positive reduced cost, then the algorithm terminates with an optimal solution. Otherwise, an entering variable is chosen from the non-basic variables with positive reduced cost. A simple rule may be to choose the variable with the largest reduced cost, but modern solvers have more sophisticated schemes for choosing entering variables, which are beyond the scope of this thesis.

The entering variable is denoted x_{j^*}.

STEP 2: Leaving variable

Now that a variable is entering the basis, we must choose a basic variable to leave the basis. First, we compute the direction in which all variables will move by the introduction of x_{j^*} into the basis, $\alpha^{j^*} = B^{-1} p^{j^*}$. As we increase the value of x_{j^*}, the other variables will reduce by α^{j^*}. This ensures we remain on the boundary of the feasible region (*i.e.* where $\tilde{A}\tilde{x} = b$). To see this, consider increasing x_{j^*} by an amount, λ. The new solution can thus be represented by:

$$\begin{bmatrix} x^B - \lambda \alpha^{j^*} \\ \lambda \end{bmatrix}, \tag{1.6}$$

which is a vector of the basic variables augmented by the entering variable, x_{j^*}. Multiplying this point by the relevant parts of the constraint matrix gives:

$$\begin{aligned} \begin{bmatrix} B & p^{j^*} \end{bmatrix} \begin{bmatrix} x^B - \lambda \alpha^{j^*} \\ \lambda \end{bmatrix} &= Bx^B - \lambda B \alpha^{j^*} + \lambda p^{j^*} \\ &= b - \lambda B B^{-1} p^{j^*} + \lambda p^{j^*} \\ &= b \end{aligned}$$

since $Bx^B = b$ (because x^B is a basic solution) and $\alpha^{j^*} = B^{-1}p^{j^*}$ by definition. So, for any value λ, the constraints $Ax = b$ will be satisfied. Thus, the only constraints to be considered are the non-negativity constraints: $x \geq 0$.

Since the leaving variable will become non-basic, it will take zero value. Therefore, we choose the leaving variable to be the first variable to reach zero when moving in the direction α^{j^*}, or

$$r = \arg\min_k \left\{ \frac{x_k^B}{\alpha_k^{j^*}} \;\middle|\; \alpha_k^{j^*} > 0 \right\}. \tag{1.7}$$

The leaving variable will thus be x_r. If there is no such variable, then it is possible to continue moving in the direction α^{j^*} without ever violating the constraints. Since the entering variable has a positive reduced cost, we may choose as large a solution as desired, and thus the algorithm terminates with an unbounded solution.

STEP 3: New basis

Now that we have chosen entering and leaving variables, we change to our new basis, where x_{j^*} has entered and x_r has left. It is also a good time to compute the new basis matrix, B, and its inverse, as well as the new values of x^B and z^B. Return to step 1, and repeat until the algorithm terminates.

Note that it is possible that one may find a series of basic feasible solutions with the same objective value. In this case, the simplex method may cycle between these basic feasible solutions without ever terminating. For this reason, it is important to include cycle-elimination in an implementation of the revised simplex method, so that it will terminate in a finite number of steps.

1.1.2 Duality theory

A significant portion of the theory around linear programming focusses on *duality theory*. To highlight some of the important properties of duality theory, we start with the linear program introduced at the beginning of the revised simplex method (1.2). This program is called the *primal problem*:

$$\begin{aligned} \max \quad & c^T x \\ \text{s.t.} \quad & Ax \leq b \\ & x \geq 0 \end{aligned} \tag{1.8}$$

Every linear program has a *dual program*. The dual program for (1.8) is:

$$\begin{aligned} \min \quad & b^T y \\ \text{s.t.} \quad & A^T y \geq c \\ & y \geq 0 \end{aligned} \tag{1.9}$$

where y is an m-vector of *dual variables*. Note that these dual variables, y, are the same dual variables encountered in the revised simplex method. The primal and dual problems are closely related, and (for a maximisation primal problem) the following properties hold:

1. The dual of the dual is the primal.
2. The objective value for any feasible solution to the primal problem is a lower bound on the optimal solution to the dual problem, and any feasible solution to the dual problem is an upper bound on the optimal solution of the primal problem.
3. If the primal problem has an optimal solution, x^*, then the dual problem also has an optimal solution, y^*, and $c^T x^* = b^T y^*$.
4. If the primal problem is unbounded, then the dual problem is infeasible. Similarly, if the dual problem is unbounded, then the primal problem is infeasible.

These properties also hold for a minimisation problem, but point 2 must be appropriately reversed so that any solution to the primal problem is an upper bound on any solution to the dual problem *etc*. The first property can be seen simply by taking the dual of the dual to reveal the original primal problem. To take the dual of the dual, we put the dual in the same form as (1.8) and apply the same transformation. To do this, we must change the objective function to a maximisation, and the constraints to less-than-or-equal-to. Both of these can be achieved by multiplying by -1, giving us the following program:

$$\begin{aligned} \max \quad & -b^T y \\ \text{s.t.} \quad & -A^T y \leq -c \\ & y \geq 0 \end{aligned} \tag{1.10}$$

Taking the dual of this as above gives:

$$\begin{aligned} \min \quad & -c^T x \\ \text{s.t.} \quad & -Ax \geq -b \\ & x \geq 0 \end{aligned} \tag{1.11}$$

Now, changing the minimisation to a maximisation and the constraints to less-than-or-equal-to using the same process as before, we return to the primal problem.

The second property can be shown by multiplying the constraints of (1.8) by y^T and the constraints of (1.9) by x^T, to give $y^T Ax \leq y^T b$ and $x^T A^T y \geq x^T c$ respectively. Since $y^T Ax = x^T A^T y$, we have that $y^T b \geq x^T c$, which says the objective value of any feasible dual solution is no less than the objective value of any feasible primal solution. This property is known as *weak duality* and is very important for Benders decomposition, which will be examined in detail in Chapter 3.

The third property is known as *strong duality* and is easily shown from the results of the revised simplex method. Assuming the algorithm has terminated with an optimal solution, then the reduced costs are zero for all basic variables ($c^{B^T} - y^T B = c^{B^T} - c^{B^T} B^{-1} B = 0$) and non-positive for all non-basic variables. This means $c^T - y^T A \leq 0$, and thus $A^T y \geq c$. Also, if $y_j < 0$ for any constraint j,

then the slack variable corresponding to constraint j would have positive reduced cost, and thus the algorithm would not have terminated, so $y \geq 0$. Therefore, y constitutes a feasible solution to the dual problem, and $y^T b = c^{B^T} B^{-1} b = c^{B^T} x^B$, or the dual problem and primal problem have the same objective value. By weak duality, the solutions to the primal and dual problems must be optimal.

The fourth property can be seen from the second property. If the primal problem is unbounded, then there exists a sequence of points x^k such that $c^T x^k \to \infty$. Assume a feasible solution to the dual exists. Then, $y^T b = y^T A x^k \geq c^T x^k \to \infty$, which is a contradiction, so the dual problem must be infeasible. A similar argument can be used to show the second part of the property.

Note that this only applies in one direction: if the primal problem is infeasible, the dual problem does not have to be unbounded, as the primal and dual problems may both be infeasible. The most concise summary is that if a linear program is unbounded, then its dual program is infeasible, and if a linear program is infeasible, then its dual program is either infeasible or unbounded.

1.1.3 Interior-point methods

The revised simplex method is not the only algorithm for solving linear programs. There are different variants of the revised simplex method (referred to as pivot step algorithms), but there are also a number of *interior-point* algorithms. Where the simplex method traces along the boundaries of the feasible region until reaching an optimal vertex, interior-point algorithms move between solutions in the interior of the feasible region.

The first interior-point algorithm is attributed to John Von Neumann in 1948 [14]. This is an inefficient method and is generally not used in practice, but its introduction sparked interest in research into similar methods. The first practical algorithm was introduced by Karmarkar in 1984 [15]. For more information about the specifics of interior-point methods, we recommend Chapter 3 of the book by Dantzig and Thapa (2003) .

Both the revised simplex method and interior-point algorithms are used for solving linear programs today. Gurobi, one of the leading commercial solvers, explains why:

> Interior-point methods have benefited significantly from recent advances in computer architecture, including the introduction of multi-core processors and SIMD instructions sets, and are generally regarded as being faster than simplex for solving LP problems from scratch. However... neither algorithm dominates the other in practice. Both are important in computational linear programming. [16]

1.2 Integer programming

Ten years after the blossoming of linear programming, researchers were struggling with the problem of solving linear programs where integer-valued answers were desired. In an interview in 2017 reflecting on his early work in integer programming, Ralph Gomory [17] remembered working with a group of naval researchers in 1957 trying to design carrier fleets using linear programming:

> When you have an aircraft carrier, you've got to have some destroyers with it, and so forth. So you need to figure out the number of ships and things like that, and how far they can go – and do they need tankers. And they use linear programming to minimize the cost of the task force, how it was composed. But the trouble was that the answers came out things like 2 and a quarter aircraft carriers. They weren't whole numbers. And it wasn't clear what you did with 2 and a quarter aircraft carriers. You could fiddle around with it and decide, did they mean 2, or did they mean 3?

At this time there were many people and organisations with similar problems: they understood the power of linear programming but could not apply it to problems that required integer-valued answers. Motivated by his collaborations with the Office of Naval Research, Gomory developed an algorithm for solving linear programming problems where the answers were required to be integer-valued [18]. Such problems became known as integer (IP) or mixed-integer (MIP) programming problems, the distinction being that MIPs also contain continuous-valued variables where IPs do not. Many highly educated and respected individuals at the time suspected that it may not even be possible to find such answers. Gomory recalls:

> I can remember meeting Martin Beale in the hall. He was a very good man. He died young, unfortunately. And he said, I see you're down to give the seminar next week. What are you going to talk about? I said, how to solve linear programs in integers. And he looked at me. He said, "that can't be done". [17]

Gomory's method is to solve the linear programming problem, and if the solution is not integer-valued, then there exists a *valid inequality* that is satisfied by all possible integer solutions, but is violated by the current optimal solution. By adding this inequality to the problem and solving again, a new optimal solution is obtained. This process is repeated until an integer solution is found. Gomory is also able to prove that this algorithm converges in a finite number of steps [19], thus making it a viable method for solving IPs and MIPs.

This method is known as a *cutting-plane* approach, as each *Gomory cut* is a hyperplane through the solution space that "cuts off" a previously feasible region. This is equivalent to adding the Gomory cut to the original linear program and solving it again. The term *valid inequality* is used here to distinguish it from another type of cutting plane, called a *lazy constraint*, which is introduced in a later section.

Cutting plane algorithms are useful for dealing with the difference between the *LP hull* and the *IP hull*. The LP hull is the convex region containing feasible solutions to the linear programming relaxation of an integer program, while the IP hull is the convex hull of the feasible integer solutions, as illustrated in Figure 1.1. If one finds a set of cutting planes that covers the optimal face of the IP hull of the given problem, then the simplex algorithm will give the optimal integer solution to that problem. This is not always practical, as high-dimensional problems may require an exponentially large set of cutting planes to achieve this. As such, few solvers exclusively use cutting-plane algorithms for solving IPs or MIPs.

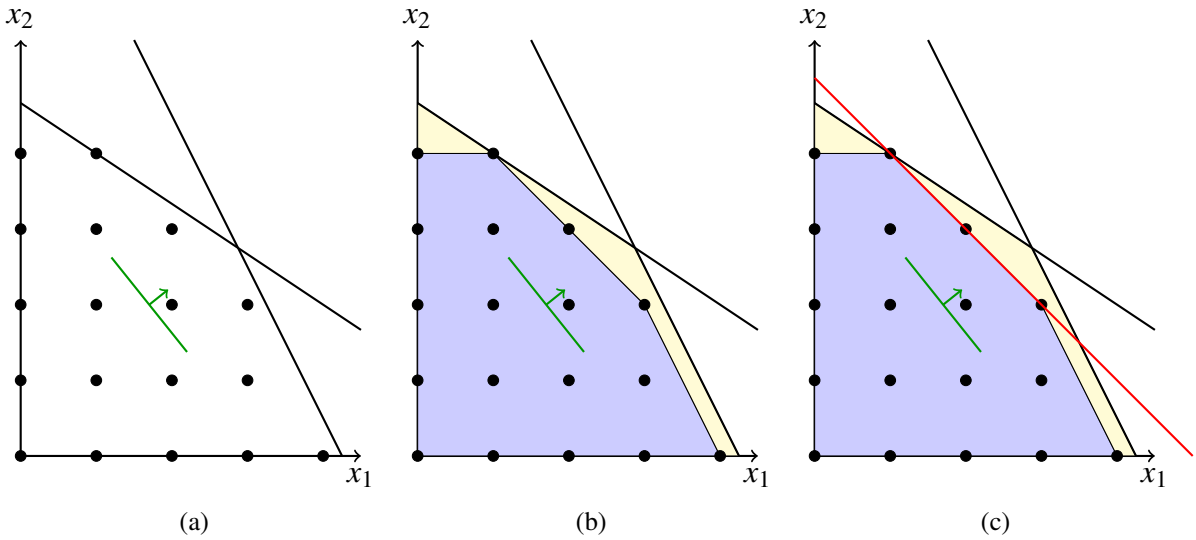


Figure 1.1: Graphical representation of an integer program in two variables. (a) The nodes represent feasible integer solutions, the straight lines are constraints and the green line is a contour of the objective function, showing the direction of increase. (b) The IP hull is highlighted in blue and the difference between the LP hull and IP hull is in yellow. (c) A valid inequality is shown in red.

1.2.1 Branch-and-bound

Within a few years, other methods for solving integer programs emerged with varying degrees of success. One of the most pervasive is the method now known as branch-and-bound. It was first presented by Land and Doig in 1960 as a method for solving general mixed-integer programming problems [20]. At the time, the authors believed it would be less effective than "successful ad hoc methods" that were tailored to specific problems, instead thinking it would be used for "testing the validity of such ad hoc methods for new problems".

The essence of branch-and-bound is as follows: solve the problem as a linear programming problem (i.e. without the integrality constraints). If the solution is integer, stop. Otherwise, find a variable that should be integer-valued but is not, and create some new problems where the chosen variable is fixed to integer values about its current value. This is slightly more restrictive than more modern versions, where instead of fixing the variables, they are constrained to be less-than-or-equal-to the floor or greater-than-or-equal-to the ceiling of their current value.

Each iteration of this algorithm creates a *branch*, a partition of the search space into two smaller spaces. A visual representation of this for the model

$$\max_{x_1,x_2}\{5x_1+4x_2 | 2x_1+3x_2 \leq 14, 4x_1+2x_2 \leq 17, x_1 \geq 0, x_2 \geq 0\} \tag{1.12}$$

is given in Figure 1.2. The LP-optimum occurs at (2.875,2.75), but both x_1 and x_2 must be integer, so we choose one variable to branch on, in this case x_1. This leads to two new problems, the *up* branch where $x_1 \geq 3$ and the *down* branch where $x_1 \leq 2$. Each of these new problems can be solved as linear programming problems, and either their solution is an integer solution, or it is not and another branch can be constructed. Each of these sub-problems is called a *node* of the branch-and-bound *tree*.

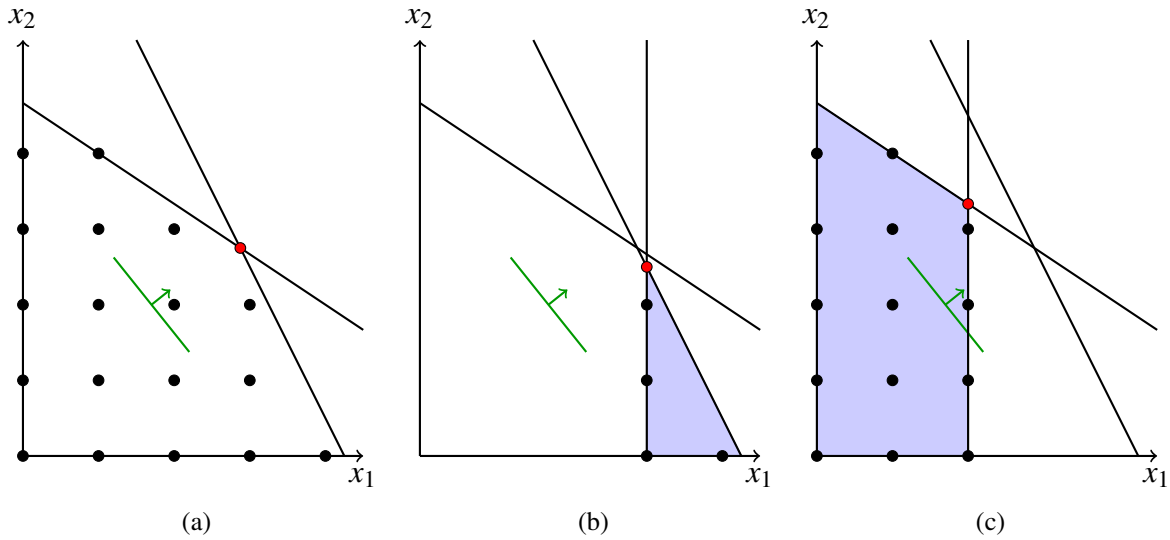


Figure 1.2: Graphical representation of branch-and-bound, where we choose to branch on x_1. (a) The original IP with the LP-optimum marked in red. (b) The *up* branch, and (c) the *down* branch. The feasible region of each branch is highlighted in blue

The bound part of branch-and-bound refers to handling nodes of the branch-and-bound tree that cannot yield the optimal solution. When an additional constraint is added to a maximisation linear program, the optimal value will be less-than-or-equal-to its previous optimal value. This is intuitive, as either the new constraint does not make the previous solution infeasible and thus the optimal solution will not change, or it does make it infeasible and the new optimal solution will be no better than the previous solution. If it was feasible and better than the previous optimal solution, then it would satisfy the new constraint and all the previous constraints, which would make it a feasible solution of the original problem, and be better than the original optimal solution, contradicting the assumption that the original solution was optimal.

Using this fact, if an integer solution has been found at one node, and the objective of another node is less-than-or-equal-to the objective value of the integer solution, then evaluating any subsequent branches of that node can not possibly deliver an improved integer solution. Similarly, if a node is infeasible, adding more constraints will not make it feasible. In this way the tree can be pruned, reducing the amount of computation necessary.

An example of this is in Figure 1.3. If the left side of the tree has been explored, then we have an integer solution with an objective value of 22. If we encounter the node $(4, 0.5)$ which also has an objective value of 22, we do not need to explore further, since it could not possibly yield a better solution. This tree also demonstrates that it is sometimes necessary to branch on one variable multiple times, as the optimal solution is found by branching on x_1 twice. Once an integer solution has been found and all other nodes have objective values less-than-or-equal-to the current solution, the process terminates with the optimal solution.

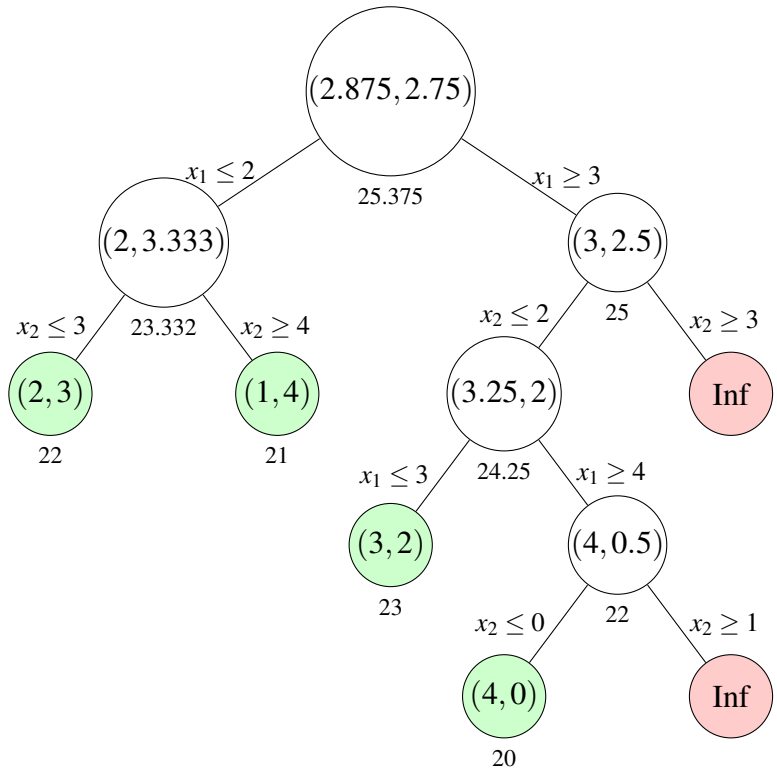


Figure 1.3: Complete branch-and-bound tree for the example IP in Figure 1.2. Green nodes are integer solutions and red nodes are infeasible problems. The objective value is displayed beneath each node

1.2.2 Problem decompositions

The foundational solution methods for mixed-integer programming problems were in place by the early 1960s, with the revised simplex algorithm for linear programs and branch-and-bound for integer programming. Since then, most methodological advancements have been cutting plane algorithms or problem decompositions.

Dantzig-Wolfe decomposition was introduced in 1960 for solving linear programming problems with a particular structure [21]. This formed the basis of a number of important techniques, generally referred to as *column-generation*, as they involve introducing additional variables that correspond to the columns of the constraint matrix. The most influential paper in this space is by Barnhart *et al.* in 1998, which lays out the method known as branch-and-price (informally referred to as delayed column-generation) for solving large integer programming problems with a special structure.

Dantzig-Wolfe decomposition and branch-and-price create formulations involving *composite variables*, which implicitly contain a number of constraints of the original problem. This is particularly desirable when the original problem contains a vast number of constraints that could be built into such variables. For example, in a vehicle routing problem, variables of the original problem may correspond to whether a vehicle traverses a particular arc, where a composite variable may be whether or not a vehicle takes a particular route from start to finish. This variable implicitly assigns values to variables of the original problem, as well as respecting flow-conservation and capacity constraints.

The benefits from such a formulation, presented by Barnhart *et al.* (1998), may include a tighter LP-relaxation than the original problem, elimination of symmetry, or simply that it may be the only

choice. The main drawback of such a formulation is the huge number of variables. Often, the number of variables that must be considered is impractical and cannot be generated *a priori.* In this case, branch-and-price may be necessary.

This is a significant drawback, as branch-and-price does not take advantage of the power of commercial solvers. While the master and sub-problems can be solved using a commercial LP solver, no commercial solvers support the addition of variables during the optimisation process, and thus do not accommodate the branching process. There are modern, non-commercial solvers such as SCIP [23] which support branch-and-price, however such solvers are significantly slower than commercial solvers. For problems that Barnhart *et al.*suggested that branch-and-price may be the only choice, then these non-commercial solvers may be the best option, but if one can formulate the problem in such a way that they can generate all columns *a priori* , the commercial solvers will be much faster.

Soon after Dantzig-Wolfe decomposition appeared, Benders decomposition was introduced for solving integer programming problems that have a particular (but different from Dantzig-Wolfe decomposition) structure [24]. Benders decomposition can be described as a row-generation technique, as it involves introducing new constraints that correspond to the rows of the constraint matrix. It is also described as a divide-and-conquer algorithm, as it takes a large MIP and separates it into a master problem and one or more sub-problems, all of which are much easier to solve than the original problem. Benders decomposition will be covered in more detail in Chapter 3.

1.2.3 Constraint programming

Another style of modeling and solving a range of decision problems is known as Constraint programming (CP). Where Linear and Integer programming were developed primarily in the fields of Mathematics and Statistics, Constraint programming was developed in Computer Science and Artificial Intelligence [25]. Operations research (LP and IP) and Constraint programming developed during the same period of time to solve many similar problems in quite different ways. It was only in the last few decades that researchers began comparing the different approaches and even looking to integrate them.

A nice summary of the similarities between OR and CP can be found in the Ph. D. Thesis by Greger Ottosson (2000), which looks at integrating the techniques of both fields to solve common problems. More recently, Hooker and van Hoeve (2018) explore several methods for solving problems using OR and CP in conjunction. John Hooker in particular has published much research on integrating OR and CP, particularly in the form of logic-based Benders decomposition [26]. While the methods are different, the problems that OR and CP aim to solve are the same.

1.2.4 Integer program solvers

The increasing interest in practical applications of linear and integer programming from industry encouraged the development of efficient computer solvers. In some cases, implementations were devised to solve specific problems quickly, such as the Concorde solver for solving the Travelling Salesman

Problem (TSP) [27]; however, the most important developments were those of the commercial MIP codes, such as CPLEX and Gurobi.

CPLEX was first released in 1988, and by version 1.2 it was capable of solving general MIPs. In his retrospective on LP computing, Bixby displayed the results of a study on the version-to-version power increase of the CPLEX code over 12 versions [10]. He compared the time that each code took to solve 1892 problems taken from academia and industry over the previous 20 years. The computations were performed on identical machines, so only the difference in the software was being compared. He found that the time to solution was roughly halved every version, with two exceptions where there were 5.5× and 10× speed increases. After 10 years of development, the CPLEX code was already 1000× faster than when it had started.

This trend of roughly doubling in speed with each new version continues to this day, and can be found in other codes. Gurobi, CPLEX's main competitor, was released in 2009 and was found to be similar in terms of speed to CPLEX version 11, which had been available since 2007. Gurobi and CPLEX have seen consistent improvements each release, and in a presentation at the INFORMS Annual Meeting in 2017, Bixby stated that he had no reason to believe the trend would end any time soon [28]. Until the end of 2018, performance benchmarks comparing Gurobi and CPLEX were maintained on Hans Mittelmann's website [29]. These comparisons have now stopped, however many of the historical benchmarks can be found through a link on the website.

The leading commercial MIP solvers CPLEX and Gurobi use a combination of branch-and-bound, cutting planes, logical pre-processing, heuristics, and parallelism to achieve higher speeds. Logical pre-processing involves looking for combinations of constraints that force variables to take particular values, allowing them to be effectively removed from the problem. Heuristics look for integer solutions without following the branch-and-bound tree and can help prune the tree more quickly, because if a good integer solution is found, we may be able to terminate a number of dead-end branches that otherwise would have wasted computation time. Parallelism refers to the independence of the nodes of the branch-and-bound tree, so that multiple nodes may be processed simultaneously, allowing the tree to be explored more quickly.

1.2.5 Lazy constraints

Perhaps the most significant development for practitioners in modern-day solvers — besides the consistent speed increases — was the introduction of *lazy constraints*. Lazy constraints are user-generated cutting planes that can be added during the branch-and-bound process. This is sometimes referred to as branch-and-cut, but that term is also applied to the use of *valid inequalities*. The difference is that a valid inequality is one that is satisfied by all integer solutions, but not by the solution of the linear relaxation, and so pares away at the space between the LP hull and the IP hull. Lazy constraints are allowed to cut off previously feasible integer solutions, and the importance of this ability is the subject of this thesis.

An example of how lazy constraints work is shown in Figure 1.4. The problem is in two variables

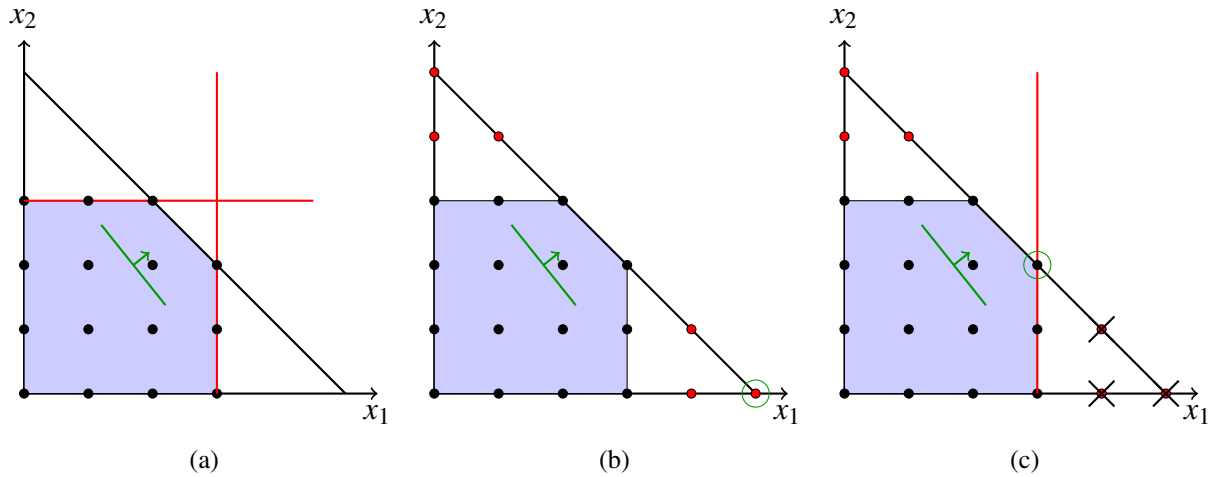


Figure 1.4: Graphical representation of lazy constraints in two variables. (a) The blue area is the feasible region for the underlying problem, the red lines are constraints to be handled lazily and the green line is a contour of the objective function, showing the direction of increase. (b) The red nodes are integer solutions which are now feasible after having removed the red constraints. The current optimal solution is circled in green (c) A constraint is added lazily, cutting off several integer solutions, and the optimal solution is now feasible in the underlying problem

with three constraints. We suspect that we may not need both red constraints, so we remove both. Note that these are the only pair of constraints that do not make the problem unbounded when they are removed. We now have a more relaxed problem with only one constraint, and this permits new integer solutions that are not feasible in the underlying problem. When we solve this problem, we find one such *illegal* solution, which is violating one of the constraints we left out. By reintroducing that constraint and solving again, we now find the correct integer solution.

CPLEX introduced the ability to add valid inequalities during the branch-and-bound process as early as 2006; however, it was not until 2012 that CPLEX separated the functions for adding user cuts and lazy constraints, and Gurobi also added full lazy constraint functionality in 2012. As such, there are not many studies that explicitly use this technique, and some studies that do will be referred to in the relevant sections of this thesis.

The amount of time required to solve an IP or MIP is heavily dependent on the number of constraints, and less so on the number of variables. This is due to how the revised simplex algorithm works: at each iteration, the dual variables associated with each constraint must be computed. The more constraints in the problem, the more dual variables there are that must be computed. Thus, models with fewer constraints are more likely to solve than models with more. This is not the only feature that dictates the difficulty of an MIP, but it is a significant one.

The power of using lazy constraints is that it is possible to "leave out" constraints from a problem that must be satisfied by any feasible solution, but which may not be required by the solver to find the optimal solution. This results in a smaller, easier-to-solve model. While solving this relaxed model, we inspect each integer solution found during the branch-and-bound process. If the solution violates one of the constraints not included in the relaxed model, we then add it as a lazy constraint, the integer solution is discarded, and all unprocessed nodes of the branch-and-bound tree are updated with the new

constraint. The result is that only those constraints that are required for finding the optimal solution are included, and only when they are needed. This can lead to dramatic improvements in solution time in some problems.

There are three main scenarios where using lazy constraints may be beneficial. They are as follows:

1. Models that have exponentially sized set(s) of constraints. Many of the constraints in these sets are unlikely to be required, so can be implemented as needed. This also includes problems where explicitly describing all constraints is difficult, but correcting invalid solutions is easy. These will be covered in Chapter 4

2. Models that have sets of constraints that may be mostly or wholly unnecessary. These problems are less common and the benefit of using lazy constraints is less but can still be significant. These problems will be covered in Chapter 2

3. Benders decomposition is often applied most efficiently using lazy constraints. These problems will be covered in Chapter 3.

Each of the chapters in this thesis cover a number of problems that fall into one of these three categories. These are examples of when lazy constraints can be beneficial, but there is a larger idea we are exploring here: the idea of modelling a problem in a lazy fashion. We can treat the modelling process the same as we do the constraints of our models and start with a smaller, relaxed formulation that gives reasonable but not necessarily feasible solutions. These solutions are then corrected using lazy constraints. This process has the potential to save time spent implementing formulations as well as improving the efficiency of those formulations.

All of this is summed up by the following "lazy maxim":

> Only that which is necessary, and only when it is necessary

The next chapter explores multiple formulations of a particular problem which has a set of mostly unnecessary constraints. We apply lazy constraints to this problem in the most direct way, and so it provides a good introduction to the idea of lazy constraints.

Chapter 2

Branch and Cut

> The more I threw away, the more I found
>
> Don DeLillo

2.1 Introduction

Branch-and-cut is a broad term used to describe a number of similar techniques for solving IPs and MIPs. The term first appeared in the late 1980s, but the ideas date back to the 1950s when Dantzig *et al.* used them for solving larger instances of the Traveling Salesman Problem (TSP) [30]. In the 1990s, branch-and-cut became state-of-the-art and has since then been standard in all MIP solvers.

In the simplest terms, branch-and-cut is similar to the branch-and-bound algorithm where additional constraints may be added after the branching process has begun. The differences are that these constraints are not just the branching constraints (which divide the search-space into smaller, distinct regions for each node of the branch-and-bound tree), and that they apply to all nodes in the branch-and-bound tree.

The first cuts used were called *valid inequalities*, constraints that were not *structural* to the problem, but which were satisfied by all valid integer solutions. Similar to Gomory cuts [19], they were used to reduce the gap between the LP and IP hulls without removing feasible integer solutions. Later, *lazy constraints* were introduced, which are cutting-planes that are allowed to cut off feasible integer solutions.

When solving an integer programming problem, there is some *underlying problem* that we want to solve which may not be the same as our current formulation. Lazy constraints are not used to cut off integer solutions that are feasible in the underlying problem. Rather, they are applied as though they are constraints that are supposed to be in our current formulation but, for whatever reason, are absent (see Figure 1.4). In practice, this allows us to formulate a smaller, relaxed problem that may

permit integer solutions that are not feasible in the underlying problem, and to remove such solutions as required using lazy constraints.

The idea of branch-and-cut predates that of branch-and-bound, with Dantzig applying branch-and-cut to the TSP in 1959 [31]. Gomory's paper on general cutting planes for solving integer programming problems [18] stimulated research into valid inequalities, however there were fewer studies that used lazy constraints [32–34]. By the 1990s, branch-and-cut was recognised as one of the most important techniques for solving MIPs, but the solvers of the 1980s were not able to effectively handle row generation.

In 1991, Grötschel and Holland noted that the code they were using, despite being one of the more powerful codes of the time, took a long time to update when adding new constraints [35]. "... if the LP-package used is better suited for a row generation process than [our current solver] is, the total speed-up obtained by faster recognition procedures might be worth the higher programming effort". In the same year, the MINTO solver developed at Georgia Tech was released, and was "the first general purpose MIP code to make systematic use of cutting-plane techniques".

In the 2000s, a few notable non-commercial solver packages with branch-and-cut capabilities were produced, particularly ABACUS (A Branch-And-CUt System) [36] and SCIP (Solving Constraint Integer Programs) [37]. These solvers allowed users to easily implement branch-and-cut in powerful solvers when the commercial solvers did not have such support. Around the same time, CPLEX implemented a branch-and-cut capability that allowed users to add valid inequalities during the branching process via a *callback*, a function that is called at nodes of the branch-and-bound tree from which additional constraints may be added. However, it was not until 2012 that branch-and-cut became particularly powerful, as both CPLEX and Gurobi implemented full lazy constraint capability.

As mentioned in Section 1.2.5, one type of problem where the use of lazy constraints is often beneficial is where there may be a number of constraints in the problem that are mostly or wholly unnecessary. This chapter contains an example problem with a number of different formulations, with results to show the difference that lazy constraints can make. This problem is the Liner-Shipping Fleet-Repositioning Problem.

2.2 The Liner-Shipping Fleet-Repositioning Problem

The Liner-Shipping Fleet-Repositioning Problem (LSFRP) is a type of vehicle routing problem that involves repositioning ships between regular service routes while maximising profit. This is achieved by visiting ports and delivering cargo while repositioning. However, despite the body of literature devoted to liner-shipping and its surrounding problems, very little of this research is focussed on the liner-shipping fleet-repositioning problem.

The first study that explores the LSFRP is by Tierney *et al.* (2012). In this paper, the authors solve a simplified version of the LSFRP without cargo flows, empty equipment, or sail-on-service (SOS) opportunities (discussed further below). Tierney and Jensen continue to explore this problem and incorporate cargo flows into the model [39]. In Tierney and Jensen (2012) , the authors use a

mixed-integer program (MIP) in conjunction with a specially constructed graph to solve the LSFRP. This graph incorporates many of the LSFRP-specific constraints (such as SOS opportunities) so they can be removed from the model formulation. This approach is able to solve several instances to optimality, but there are many larger instances where the problem cannot be solved because the solver runs out of memory or exceeds the maximum CPU time of one hour.

Another approach to solving the LSFRP is proposed by Kelareva, Tierney and Kilby (2014) . In this study, they solve the full LSFRP with SOS opportunities; however, they do not incorporate cargo flows into their model. A constraint programming (CP) method is used with lazy clause generation, and is tested against the MIP in Tierney *et al.* (2012). After testing the different models on a data set, the CP method is found to be faster than the MIP for all instances. However, this only occurs after choosing a search strategy for the particular problem. The authors note that, without sufficient understanding of the problem and of CP modelling techniques, it is difficult to choose a search strategy that is fast and successfully finds optimal solutions. Furthermore, the CP method cannot be extended to allow pre-computations, chaining of SOS, or opportunities to carry empty cargo containers.

A more recent study on the LSFRP is by Tierney *et al.* (2015), which expands on the work by Tierney and Jensen (2012). They improve the model, provide a public data set, and use a heuristic approach. This model is able to incorporate many complex aspects of the LSFRP, including SOS opportunities, phase-in/phase-out requirements, and flexible arcs. Some of these (SOS opportunities and phase-in/phase-out requirements) are processed into the graph structure, along with sailing costs and cabotage restrictions. The MIP forms a "disjoint path problem in which a fractional multi-commodity flow is allowed to flow over arcs in the vessel paths, along with a small scheduling component in the flexible nodes" [41].

Much of the state-of-the-art nature of this problem is covered in the book Tierney (2015). A number of different models are presented, but we are particularly interested in the models with cargo flows (Chapter 6) and without flexible visitations. Four main models are presented in Chapter 6 of Tierney's book [42]: an arc-flow model (6.2), a path-based model (6.3), and the equipment as flows and demands models (6.4.2-6.4.3). The arc-flow model is a standard MIP that can solve many small instances, but does not scale well. The path-based model must be implemented using branch-and-price, as there are too many variables in the larger instances to generate them *a priori.* The equipment as flows and demands models are a more interesting approach to modelling the problem.

The path-based model and the equipment as flows and demands models are individually able to solve the largest instances covered by Tierney (2015), but it is possible to do much better by combining these methods and applying branch-and-cut. First, we introduce the problem and then explore the methods for solving it.

2.2.1 Problem description and model formulation

The LSFRP consists of finding sequences of activities that move vessels between services in a liner shipping network, while maximising profit by trading off ship moving costs and cargo flow

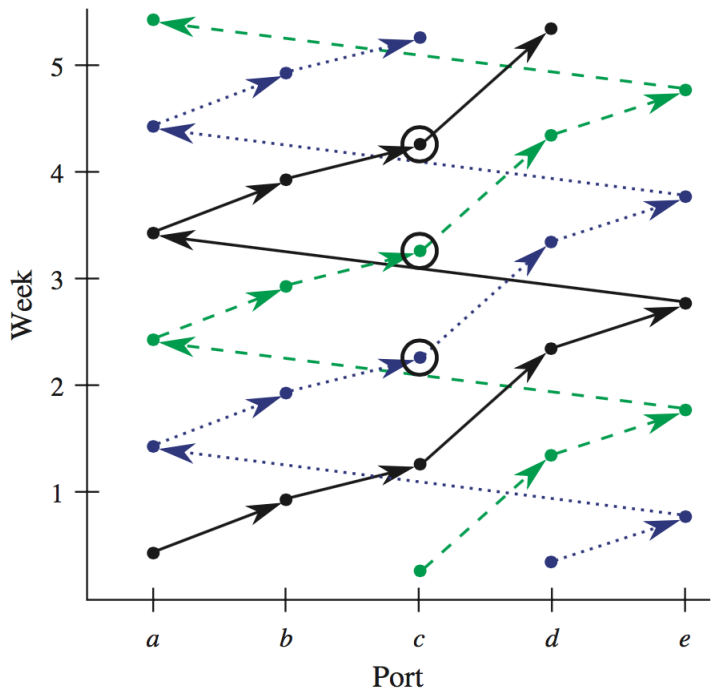


Figure 2.1: A time-space graph of a service with three vessels. Reprinted with permission from Tierney *et al.* (2015)

incomes [41]. "Liner shipping services are composed of multiple slots, each of which represents a cycle that is assigned to a particular vessel". The slots contain nodes or ports that must be visited by vessels at specific times in sequence. When a vessel is assigned a slot, it sails to all of its ports in order and delivers its cargo. Figure 2.1 shows an example of a service with three slots (represented by the differently styled lines in the graph) and five ports (a,b,c,d,e). The diagram shows that each slot takes three weeks to return to the start of the cycle, so three ships would be needed to run this service weekly. An instance of the LSFRP occurs when transitioning a fleet from one set of services to a new one.

Another aspect of repositioning that needs to be taken into account is the time constraints. The time at which a ship may begin repositioning is known as the phase-out time. The ship must finish repositioning by the phase-in time of the goal service. In between these two times the ship is available for repositioning and is able to undertake a number of activities to reach its goal service and reduce costs.

An example of this can be seen in Figure 2.1. Three ships are operating a weekly service on a three week journey, and must phase-in before the circled nodes in weeks 2, 3 and 4. Before the circled nodes, the ships may undertake repositioning activities and visit nodes not on the new service, but after the circled nodes, the ships must operate the full service.

The LSFRP is best described using Figure 2.2. This shows a ship that must be repositioned from its initial service (Chennai Express) to a goal service (Intra-WCSA). During repositioning the vessel can deliver cargo to ports to offset the cost of moving the ship, thus cargo flows are an important aspect of the problem. One way to do this is to take advantage of sail-on-service (SOS) opportunities, which are situations in which a repositioning ship can replace an on-service vessel for part of its service in

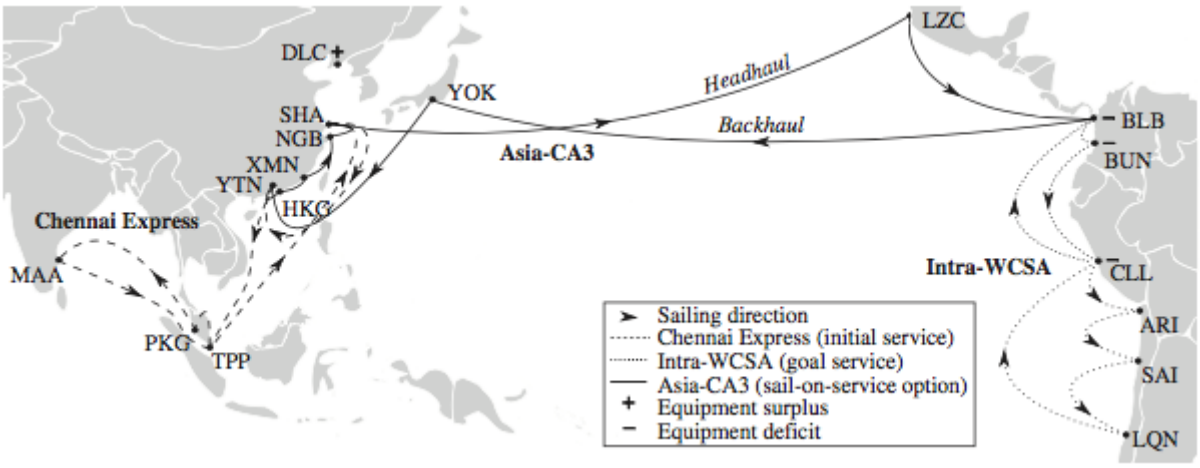


Figure 2.2: Liner shipping network. Reprinted with permission from Tierney *et al.* (2015)

order to reduce costs (by not having two ships sailing on the same course unnecessarily). There are two main methods of performing a SOS opportunity: transhipping, where all cargo from the on-service ship is moved onto the repositioning ship at a port, or parallel sailing, where the two ships visit the same ports sequentially and the on-service vessel only unloads cargo, while the repositioning vessel only loads cargo.

Empty containers and flexible arcs

Another way for ships to offset the cost of repositioning is to transport empty containers from ports with an empty equipment surplus to ports with a deficit. The revenue from performing this type of activity is calculated as an approximation of the savings from moving the equipment now, as opposed to at a later date, potentially through a more expensive channel. There are two types of cargo considered in this problem: dry and refrigerated (reefer). We must differentiate between the two types, since when transporting cargo the reefer containers must be plugged into a power outlet, which means that ships will only have a limited reefer capacity. The dry capacity, however, refers to the total capacity of the ship, as dry containers can be stored in reefer slots. This is not the case when moving empty equipment, but we still make the distinction as the deficit we are supplying may be for a specific container type. We use the term flexible visits to denote ports with empty equipment available but no actual cargo demands. These flexible visits are travelled to via flexible arcs.

There are also various restrictions placed on the cargo carried by repositioning ships such as trade zones. Trade zones are countries or groups of countries with trade agreements. Often, cargo cannot flow between trade zones without violating these agreements. To avoid the movement of cargo violating these trade zone restrictions, the law, or a customer contract, repositioning ships are disallowed from crossing into other trade zones while carrying cargo. A similar restriction is known as a cabotage restriction, which prevents international ships from performing domestic cargo services [41]. These are all aspects that need to be considered when modelling the LSFRP. Most of these restrictions have been incorporated directly into the network of potential ship paths, so they will not be represented in the MIP formulation.

For the original model formulation, we refer the reader to the paper by Tierney *et al.* (2015). We have maintained consistency in notation from previous studies. We now present a reduced formulation of the LSFRP.

2.2.2 Reduced MIP

Starting with the arc-flow model described by Tierney (2015) , a reduced version is formulated that does not incorporate flexible arcs or empty equipment. These aspects of the problem are omitted for simplicity to allow us to explore the core structure of the LSFRP without added complexity. By noting the percentage of the public data set that does not include these additional requirements (66%), it can be seen that the reduced problem is still able to provide much value, as the majority of the instances do not contain the more complex aspects of the problem. There are more reasons why omitting these aspects of the problem are reasonable, some of which will be explored in Section 2.2.8.

We now move the ships through a time-space network where each node, also called a *visit*, represents a particular port at a particular time. This network is thus acyclic, which will be useful in Section 2.2.4. Also, no two ships may visit the same node, since they would be at the same port at the same time. The reduced Liner Shipping Fleet Repositioning Problem is described as follows:

Parameters

S	Set of ships.
V	Set of visits
V'	Set of visits minus the graph sink.
A	Set of arcs
A'	Set of arcs minus those arcs connecting to the graph sink, *i.e.* $(i,j) \in A, i,j \in V'$.
Q	Set of cargo types; $Q = \{dc, rf\}$.
M	Set of demand triplets of the form (o,d,q), where $o \in V'$, $d \subseteq V'$, and $q \in Q$ are the origin visit, possible destination visits, and the cargo type respectively.
M_i^{Orig}, (M_i^{Dest})	Set of demands with an origin (destination) visit $i \in V$.
u_s^q	Capacity of vessel s for cargo type $q \in Q$.
v_s	Starting visit of ship $s \in S$.
$r^{(o,d,q)}$	Amount of revenue gained per TEU of loaded containers carried for the demand triplet.
c_{sij}^{Sail}	Fixed cost of vessel s utilizing arc $(i,j) \in A'$.
c_i^{Mv}	Cost to move a single TEU on or off a ship at visit $i \in V'$.
c_{si}^{Port}	Port fee associated with vessel s at visit $i \in V'$.
$a^{(o,d,q)}$	Amount of demand available for the demand triplet.
$In(i)$	Set of visits with an arc connecting to visit $i \in V$.
$Out(i)$	Set of visits receiving an arc from visit $i \in V$.
τ	Graph sink, which is not an actual visit.

Variables

$x_{ij}^{(o,d,q)}$ Amount of flow of demand triplet $(o,d,q) \in M$ on arc $(i,j) \in A'$

y_{ij}^{s} 1 if vessel s sails on arc $(i,j) \in A$, 0 otherwise

Objective

$$\max \quad \left\{ \sum_{(o,d,q)\in M} \left(\sum_{j\in d} \sum_{i\in In(j)} (r^{(o,d,q)} - c_o^{Mv} - c_j^{Mv}) x_{ij}^{(o,d,q)} \right) \right. \tag{2.1}$$

$$\left. - \sum_{s\in S} \sum_{(i,j)\in A'} c_{sij}^{\text{Sail}} y_{ij}^{s} - \sum_{j\in V'} \sum_{i\in In(j)} \sum_{s\in S} c_{sj}^{\text{Port}} y_{ij}^{s} \right\} \tag{2.2}$$

Constraints

$$\text{s.t.} \sum_{s\in S} \sum_{i\in In(j)} y_{ij}^{s} \leq 1 \qquad \forall j \in V' \tag{2.3}$$

$$\sum_{j\in Out(i)} y_{ij}^{s} = 1 \qquad \forall s \in S, i = v_s \tag{2.4}$$

$$\sum_{i\in In(\tau)} y_{i\tau}^{s} = 1 \qquad \forall s \in S \tag{2.5}$$

$$\sum_{i\in In(j)} y_{ij}^{s} - \sum_{i\in Out(j)} y_{ji}^{s} = 0 \qquad \forall j \in V' \setminus \bigcup_{s\in S} v_s, s \in S \tag{2.6}$$

$$\sum_{(o,d,rf)\in M} x_{ij}^{(o,d,rf)} \leq \sum_{s\in S} u_s^{rf} y_{ij}^{s} \qquad \forall (i,j) \in A' \tag{2.7}$$

$$\sum_{(o,d,q)\in M} x_{ij}^{(o,d,q)} \leq \sum_{s\in S} u_s^{dc} y_{ij}^{s} \qquad \forall (i,j) \in A' \tag{2.8}$$

$$\sum_{i\in Out(o)} x_{oi}^{(o,d,q)} \leq a^{(o,d,q)} \sum_{i\in Out(o)} \sum_{s\in S} y_{oi}^{s} \qquad \forall (o,d,q) \in M \tag{2.9}$$

$$\sum_{i\in In(j)} x_{ij}^{(o,d,q)} - \sum_{k\in Out(j)} x_{jk}^{(o,d,q)} = 0 \qquad \forall (o,d,q) \in M, j \in V' \setminus (o \cup d) \tag{2.10}$$

$$x_{ij}^{(o,d,q)} \geq 0, y_{ij}^{s} \in \{0,1\} \qquad \forall (i,j) \in A', (o,d,q) \in M, s \in S \tag{2.11}$$

The objective function maximises the profit of the shipping company. The first line (2.1) calculates the profit from delivering the cargo by adding the revenue minus the cost to transport the cargo on and off the ship. This is multiplied by the amount of cargo carried. The second line of the objective function (2.2) subtracts the sum of the sailing costs and the port fees for each port visited by each ship.

Constraint (2.3) ensures that only one ship visits each port, while (2.4-2.6) conserve the flow of each ship from its starting port to the sink node. If a ship uses an arc, that arc is assigned a reefer capacity by (2.7) and a total capacity by (2.8). Constraint (2.9) ensures that cargo can only flow along an arc if it is on a ship. Constraint (2.10) conserves the flow of cargo from its source node to its destination by ensuring that if it enters an intermediate node, it must also exit that node.

Note about the formulation

As stated, this model is adapted from the model described by Tierney (2015), however there is an issue with the original formulation, and thus this formulation. The revenue from serving a demand is

calculated from the amount of that demand flowing into each potential destination, however there is no provision for consumption of the demand. Thus, it is possible to pick up a demand and flow it through multiple destinations, collecting the revenue multiple times.

This is made possible because of the way the $x_{ij}^{(o,d,q)}$ variables are handled: flow conservation constraints apply to all nodes except the origin and any destinations, and the capacity cannot be exceeded. Because of this, it is possible to load demand at a destination or carry a demand through a destination to collect the revenue multiple times. It also seems possible to load more than the available amount of a demand if it is picked up from a demand point, as that is not constrained.

Having said this, the model works as intended on the data sets used by Tierney (2015). It is unclear whether this is by chance or by construction, but taking advantage of any opportunities for collecting demand revenues multiple times comes at a cost of lost opportunities for serving other demands or increased sailing costs that completely negate the benefits. Should one wish to use this model for their own purposes, this problem may need to be fixed. In the data used by Tierney (2015), multiple destinations for an order correspond to the same physical location at different times, and so have the same unloading cost, so it is always optimal to unload a demand at the first destination encountered. As such, the constraints

$$x_{ij}^{(o,d,q)} = 0 \quad \forall i \in d, \forall j \in Out(i), \forall (o,d,q) \in M \tag{2.12}$$

will prevent demand from being carried out of one of its destinations. If it is not the case that the unloading costs at all destinations are the same, then a more substantial modification to the model may be required.

Tightening the MIP

While the reduced MIP is able to solve a number of smaller instances [41], it is still unable to solve the last seven instances in the public data set. We note that one of the reasons the MIP struggles on larger problems is because the linear relaxation of the problem generates solutions in which fractional ship variables are used to transport all of a demand triplet. In order to prevent this, an additional set of constraints is added:

$$x_{oi}^{(o,d,q)} \leq a^{(o,d,q)} \sum_{s \in S} y_{oi}^{s} \quad \forall i \in Out(o), (o,d,q) \in M; \tag{2.9a}$$

These constraints prevent ships from moving a greater fraction of the demand triplet than the fraction of the ship used. This is a disaggregated version of constraint (2.9) from the reduced formulation, as it is no longer summed over $i \in Out(o)$. This is allowed since only one ship can visit any node, and thus only one arc leaving each node will have a non-zero value of y_{ij}^{s} in any integer solution. By the properties of disaggregation, this must give a tighter bound for the linear relaxation. This improved bound yielded strong improvement on some larger instances, but it is still unable to solve five instances to optimality within the timeout limit.

2.2.3 Revised formulation

Despite these tighter constraints, fractional parts of demand triplets can still be shipped, as the new constraints apply to all the ships rather than individual ships (the RHS is summed over s). To combat this, we reformulate the model for individual ships by adding a ship index to the x variables. As stated earlier, an important aspect of the problem to note is that the paths need to be node-distinct, meaning that only one ship can visit each node. This property means that no product can be transshipped, and allows the ship index to be added to the x variables. The revised formulation with $x_{ij}^{s,(o,d,q)}$ variables is shown below.

Variables

$x_{ij}^{s,(o,d,q)}$ Amount of flow of demand triplet $(o,d,q) \in M$ on arc $(i,j) \in A'$ and ship $s \in S$

y_{ij}^{s} 1 if vessel s sails on arc $(i,j) \in A$.

Objective

$$\max \quad \left\{ \sum_{(o,d,q)\in M} \sum_{s\in S} \sum_{j\in d} \sum_{i\in In(j)} (r^{(o,d,q)} - c_o^{Mv} - c_j^{Mv}) x_{ij}^{s,(o,d,q)} \right. \tag{2.13}$$

$$\left. - \sum_{s\in S} \sum_{(i,j)\in A'} c_{sij}^{\text{Sail}} y_{ij}^{s} - \sum_{j\in V'} \sum_{i\in In(j)} \sum_{s\in S} c_{sj}^{\text{Port}} y_{ij}^{s} \right\} \tag{2.14}$$

Constraints

$$(2.3-2.6)$$

$$\sum_{(o,d,rf)\in M} x_{ij}^{s,(o,d,rf)} \leq u_s^{rf} y_{ij}^{s} \qquad \forall (i,j) \in A', s \in S \tag{2.15}$$

$$\sum_{(o,d,q)\in M} x_{ij}^{s,(o,d,q)} \leq u_s^{dc} y_{ij}^{s} \qquad \forall (i,j) \in A', s \in S \tag{2.16}$$

$$\sum_{i\in Out(o)} x_{oi}^{s,(o,d,q)} \leq a^{(o,d,q)} \sum_{i\in Out(o)} y_{oi}^{s} \qquad \forall (o,d,q) \in M, s \in S \tag{2.17}$$

$$\sum_{i\in In(j)} x_{ij}^{s,(o,d,q)} - \sum_{k\in Out(j)} x_{jk}^{s,(o,d,q)} = 0 \qquad \forall (o,d,q) \in M, j \in V' \backslash (o \cup d), s \in S \tag{2.18}$$

$$x_{ij}^{s,(o,d,q)} \leq y_{ij}^{s} \min(a^{(o,d,q)}, u_s^{q}) \qquad \forall (i,j) \in A', s \in S, (o,d,q) \in M \tag{2.19}$$

$$x_{ij}^{s,(o,d,q)} \geq 0, y_{ij}^{s} \in \{0,1\} \qquad \forall (i,j) \in A', s \in S, (o,d,q) \in M \tag{2.20}$$

The objective value is unchanged from the reduced MIP, except now the x variables are also summed over all ships $s \in S$. Constraints (2.15-2.18) are disaggregated versions of constraints (2.7-2.10), so there is now one constraint for each ship. Finally, constraint (2.19) is a disaggregated version of constraint (2.9a), which ensures that for each ship, and on each arc, no more cargo can be transported than is available or able to be transported on the ship.

Note that the aforementioned problem also exists in this formulation, however it still gives the correct solutions for the data sets used by Tierney (2015), and the suggested correction can still be applied by adding the ship index to the z variables. While this formulation does introduce more

variables into the problem, it also provides a linear relaxation with a tighter bound, which allows it to solve much faster for larger instances.

2.2.4 Model reformulation

In his book, Tierney also considers a model reformulation that dramatically reduces the number of variables and constraints in the problem [42]. Instead of the flow of demands on ships and along arcs using variables $x_{ij}^{s,(o,d,q)}$, we consider only the amount of each demand carried on each ship, $x_s^{(o,d,q)}$. This is possible because the network is acyclic (since it is a time-space network), no transshipment is allowed, each node may be visited by at most one ship, and demands have a specific origin. This means each demand may be carried by at most one ship, and if any two demands are ever carried together, they will always be carried together.

Consider a ship that visits port A and loads some demand from that port. The ship then proceeds to port B where it loads another demand from this port. Finally, the ship proceeds to port C. Because the demands from ports A and B were carried together, the sum of demands from A and B cannot exceed the capacity of the ship on this arc. If demand B is unloaded at port C, it will not be possible to return to port A to load more demand of type A, and vice versa.

The following formulation uses these ideas originally presented by Tierney (2015), but the notation is slightly different. We define the set $\bar{M}_s$ as the set of all demands (o,d,q) that can be moved by ship s. By extension, $\bar{V}_{sq}^{\text{Orig}}$ is the set of all nodes from which ship s can pick up a demand of type q from $\bar{M}_s$, *i.e.*

$$\bar{V}_{sq}^{\text{Orig}} = \{o | (o,d,q) \in \bar{M}_s\}$$

Finally, we define $A^{(o,d,q)}$ as the set of all arcs $(i,j) \in A'$ across which a demand triple $(o,d,q) \in M$ can possibly travel, and M_{ij} is the set of all demand triples that could possibly travel across arc $(i,j) \in A'$. We also use the set M_i^{Orig} from the original formulation, which is the set of demands that originate from $i \in V$. Starting with the revised formulation from Section 2.2.3, we modify the x variables as mentioned above, remove constraints (2.15-2.19), and replace them with the following:

$$\sum_{(k,d,q)\in M_k^{\text{Orig}}} x_s^{(k,d,q)} \leq \sum_{\substack{j\in Out(k)\\(k,j)\in A'}} u_s^{dc} y_{kj}^s \qquad \forall k \in \bigcup_{q\in Q} \bar{V}_{sq}^{\text{Orig}}, s \in S \tag{2.21}$$

$$\sum_{(k,d,rf)\in M_k^{\text{Orig}}} x_s^{(k,d,rf)} \leq \sum_{\substack{j\in Out(k)\\(k,j)\in A'}} u_s^{rf} y_{kj}^s \qquad \forall k \in \bar{V}_{s,rf}^{\text{Orig}}, s \in S \tag{2.22}$$

$$x_s^{(o,d,q)} \leq \min\left(a^{(o,d,q)}, u_s^q\right) \sum_{\substack{j\in Out(o)\\(o,j)\in A^{(o,d,q)}}} y_{oj}^s \qquad \forall (o,d,q) \in \bar{M}_s, s \in S \tag{2.23}$$

$$x_s^{(o,d,q)} \leq \min\left(a^{(o,d,q)}, u_s^q\right) \sum_{j\in d} \sum_{\substack{i\in In(j)\\(i,j)\in A^{(o,d,q)}}} y_{ij}^s \qquad \forall (o,d,q) \in \bar{M}_s, s \in S \tag{2.24}$$

$$\sum_{(o,d,q)\in M_{ij}\cap\bar{M}_s} x_s^{(o,d,q)} \leq u_s^{dc} \qquad \forall(i,j)\in A', \forall s\in S \tag{2.25}$$

$$\sum_{\substack{(o,d,rf)\in M_{ij}\cap\bar{M}_s \\ q=rf}} x_s^{(o,d,rf)} \leq u_s^{rf} \qquad \forall(i,j)\in A', \forall s\in S \tag{2.26}$$

Constraints (2.21-2.22) ensure that the sum of all demands loaded at any node cannot exceed the total capacity of the ship, and the sum of reefer demands loaded at any node cannot exceed the reefer capacity of the ship. Constraints (2.23-2.24) ensure that a demand can only be carried if the ship passes through the demand's origin and one of its destinations, and caps the flow of each demand by the minimum of the demand's availability and the ship's capacity for the specific type. Constraints (2.25-2.26) are the capacity constraints for each arc for all demand types and specifically reefer cargo.

This formulation is now much smaller than the revised formulation. However, there are a few details to take note of to complete the formulation.

Splitting demand triples

While it is true for a specific path through the network that if two demands are carried together at any time, they will be carried together until one is unloaded, that does not mean that the two demands must be carried together in the first place. This can occur if one of the demands has more than one possible destination. Consider the example in Figure 2.3, where demand A will be picked up from Origin A. There are two choices: either proceed directly to Origin B, still carrying cargo A, and pick up cargo B, or proceed to destination A1, unload cargo A and continue to Origin B. In the first case, a capacity constraint would limit the amount of demands A and B that may be carried together, but in the second case no such restriction exists.

This means the variables for demand A must be separated into one for each destination, so the capacity constraints may be applied correctly. We now must search for any demand triples (o,d,q) that fit the following criteria:

- There are two destinations $d_1, d_2 \in d$ such that d_2 is reachable from d_1.
- There exists another demand m^* with origin o^* such that o^* is reachable from d_1, and d_2 is reachable from o^*.
- There exists a destination d^* of demand m^* such that either d^* is reachable from d_2 or vice versa.
- There exists a path between the origins o and o^* that does not pass through d_1.

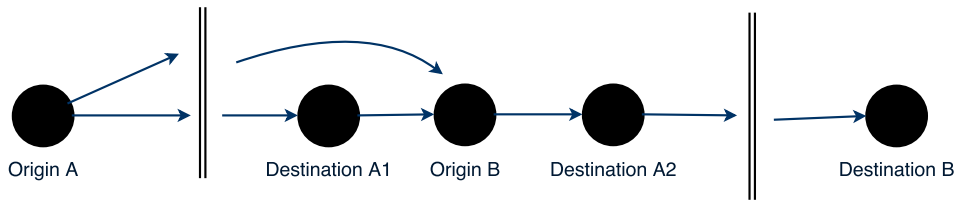


Figure 2.3: A scenario where the variables for a particular demand triple need to be separated.

If such conditions are met, then the variables for $x_s^{(o,d,q)}$ are split up into $x_s^{(o,d_i,q)}$ $\forall d_i \in d$, and an additional constraint is added:

$$\sum_{d_i \in d} x_s^{(o,d_i,q)} \leq a^{(o,d,q)}. \tag{2.27}$$

This constraint enforces the total availability of the split demand. The splitting of such demands ensures we do not unnecessarily over-constrain the problem. Each demand triple has an associated revenue and amount, both of which are inherited by the new split variables.

Modifying the objective function

Because we are no longer explicitly calculating which arcs the cargo travels along, we cannot easily determine which destination it is being delivered to. We assume that cargo is unloaded at the earliest possible time, that is, if the node i is visited, all demands (o,d,q) where $i \in d$ will be unloaded. In the public data set, each destination in a demand triple is the same physical port — the difference is the delivery time. This means the unloading cost of each demand triple is the same for all destinations. This allows us to take the unload cost of any destination from the demand triple. If the destinations have different unload costs, then it is necessary to split the demand triple as discussed in the previous section.

2.2.5 Branch-and-price

While the MIP models presented are effective for solving the smaller instances, they do not scale well and struggle to solve the larger instances. None of the MIP formulations are able to solve the two largest instances in under one hour. To overcome this, we use a path-based formulation similar to the ones found in Tierney (2015) and Tyler (2015) . Instead of considering individual arcs for the ships to travel along, we consider entire paths for the ships from start to finish. This greatly reduces the number of constraints, as many of them (such as capacity and flow-conservation constraints) are built into the path variables, so we need only ensure every ship takes exactly one path and no two paths visit the same node.

The model itself is relatively simple; however, the number of variables is vast. To handle this, we use branch-and-price, also known as delayed column generation, to generate paths that will improve our current solution until no more such paths can be found, at which point we have the optimal solution.

We begin with the Restricted Master Problem (RMP), which chooses a path for each ship.

Parameters

P Set of paths

C_{sp} Profit of vessel s sailing on path p (revenue from moving product less the cost of the path)

δ_{isp} 1 if vessel s sailing on path p goes through node $i \in V'$, 0 otherwise

Variables

z_{sp} 1 if vessel s sails on path $p \in P$, 0 otherwise

Objective

$$\max \sum_{p \in P} \sum_{s \in S} C_{sp} z_{sp} \tag{2.28}$$

Constraints

$$\sum_{p \in P} z_{sp} = 1 \qquad \forall s \in S \tag{2.29}$$

$$\sum_{p \in P} \sum_{s \in S} \delta_{isp} z_{sp} \leq 1 \qquad \forall i \in V' \tag{2.30}$$

$$z_{sp} \geq 0 \qquad \forall s \in S, \forall p \in P \tag{2.31}$$

The objective is to maximise the sum of the profits for the paths that are chosen. Constraints (2.29) say each ship is used exactly once, and constraints (2.30) ensure each node is visited at most once. This is a simple model because many of the complex constraints have been built into the structure of the paths. The difficulty comes from generating the paths. As there are far too many possible paths to enumerate *a priori* , we use branch-and-price to generate paths that could improve our solutions.

To generate new paths, we solve a series of sub-problems, one for each ship, which consist of constraints similar to (2.3-2.6) and (2.15-2.19). However, the objective function has been modified to include the dual variables π_s and λ_i associated with constraints (2.29) and (2.30) respectively:

$$\max \quad \left\{ \sum_{(o,d,q) \in M} \sum_{j \in d} \sum_{i \in In(j)} (r^{(o,d,q)} - c_o^{Mv} - c_j^{Mv}) x_{ij}^{s,(o,d,q)} \right. \tag{2.32}$$

$$\left. - \sum_{(i,j) \in A'} c_{sij}^{\text{Sail}} y_{ij}^s - \sum_{j \in V'} \sum_{i \in In(j)} c_{sj}^{\text{Port}} y_{ij}^s - \sum_{(i,j) \in A'} \lambda_i y_{ij}^s - \pi_s \right\} \tag{2.33}$$

If a solution is found with a positive objective value, then it represents a new path that may improve the solution to the master problem. The path is then added to the pool of potential paths, and the master problem is solved again. This continues until no such paths are found, at which point the solution to the master problem is the optimal solution to the original problem.

In this formulation, the variables z_{sp} are continuous variables, but a solution is only valid if they are binary, which means we require an integer solution to the master problem. Normally, this would require branching; however, for the LSFRP, the master problem solutions are naturally integer, and thus no branching is required. For proof of this, see Tyler (2015). This is one reason column generation is effective for this problem.

2.2.6 Branch-and-cut

The combination of column generation with the model reformulation is very effective, as shown by the results in Section 2.2.7. The only downside to this formulation is that there is a vast number of constraints (2.25-2.26), and many of them are likely to be unnecessary. As such, we can handle

them as lazy constraints, which makes the implementation a branch-and-price with a branch-and-cut sub-problem.

When solving the sub-problems, we must check every integer solution for violated capacity constraints. If a capacity constraint is violated, we add it back in as a lazy constraint and continue solving. Once the sub-problem has been solved, any capacity constraints introduced are added as regular constraints for the next time the sub-problem is solved. This leads to another dramatic improvement, especially for the largest instances. We can similarly handle the capacity constraints of the reformulated MIP as lazy constraints, which makes that a branch-and-cut implementation.

2.2.7 Results

Computational experiments were performed on a compute cluster running Linux. Each job was assigned a maximum of 8 cores running at 2.4GHz each, and 56GB of RAM. The instances used are from the public data set by Tierney *et al.* (2015). Table 2.1 shows for each instance the number of ships ($|S|$), number of nodes ($|V|$) and arcs ($|A|$) in the network and the number of potential requests ($|M|$).

Seven formulations are compared: the original MIP formulation (MIP), the revised MIP formulation from Section 2.2.3 (Rev. MIP), the reformulated MIP from Section 2.2.4 (Ref. MIP), the reformulated MIP with lazy capacity constraints (Lazy MIP), branch-and-price on the revised MIP model (B+P), branch-and-price on the reformulated model (Ref. B+P), and branch-and-price on the reformulated model with lazy capacity constraints (Ref. B+P+C). All formulations were implemented using Python 3 as part of the Anaconda distribution (4.1.1) and use the Gurobi 7.0.1 [44] optimisation package. All software is 64-bit. The maximum runtime for each instance was set to 24 hours for experimental purposes, but the results reported are capped at 1 hour.

A comparison of the runtimes for each formulation on a number of the instances is shown in Table 2.1. For the smaller instances (numbers up to 32), all formulations are able to solve to optimality in a few seconds. In most cases, one of the reformulated branch-and-price formulations performs best. For the larger instances, a clear pattern emerges: Ref. B+P+C is better than Ref. B+P, which in turn is better than B+P. For the two largest instances, B+P fails to solve in an hour, but it can solve the instances to optimality in approximately 9200 seconds (2.5 hours). Thus, Ref. B+P performs 10-20$\times$ better than the original B+P, and handling the capacity constraints lazily yields another 2-4$\times$ improvement.

The column labelled *Capacity cons.* contains the number of capacity constraints in the Ref. MIP and Ref. B+P formulations. This is presented for comparison against the numbers of lazy constraints added by the Lazy MIP and Ref. B+P+C formulations. The reason for the extra power in the Ref. B+P+C formulation over Ref. B+P is clear: there are many unnecessary constraints in the model that we do not consider. A large number of the instances are solved to optimality without ever finding a violated capacity constraint, and even the largest instances need fewer than 40 lazy constraints added, a remarkably small number compared to the 100,000 that were originally in the problem.

Instance ID	$\|S\|$	$\|V\|$	$\|A\|$	$\|M\|$	MIP time (s)	Rev. MIP time (s)	Ref. MIP time (s)	Lazy MIP time (s)	Capacity cons.	Lazy cons.	B+P time (s)	Ref. B+P time (s)	Ref. B+P+C time (s)	Lazy cons.
repos2p	3	36	150	28	0.11	0.08	0.09	0.06	368	0	0.06	0.07	0.09	0
repos4p	3	42	185	20	0.09	0.10	0.10	0.09	390	0	0.06	0.07	0.07	0
repos5p	3	51	270	22	0.14	0.14	0.12	0.09	562	0	0.09	0.08	0.08	0
repos6p	3	51	270	22	0.14	0.15	0.11	0.10	562	0	0.09	0.07	0.08	0
repos7p	3	54	196	46	0.12	0.16	0.10	0.08	450	0	0.11	0.07	0.08	0
repos10p	4	58	499	125	1.68	1.77	0.37	0.20	1722	0	1.98	0.47	0.51	4
repos12p	4	74	603	145	2.16	2.36	0.55	0.27	2154	0	3.54	0.83	0.52	4
repos13p	4	80	632	155	2.21	2.60	0.51	0.30	2272	0	2.06	0.65	0.56	4
repos15p	5	71	355	173	0.43	0.87	0.31	0.34	810	5	0.50	0.25	0.32	9
repos16p	5	106	420	320	0.57	1.17	0.51	0.44	1012	5	0.56	0.35	0.46	7
repos17p	6	102	1198	75	1.00	2.31	0.83	0.60	5796	0	1.96	1.11	0.62	0
repos18p	6	135	1439	87	1.53	3.12	1.11	0.81	6994	0	1.98	1.33	0.91	0
repos20p	6	142	1865	80	1.95	3.75	1.29	0.92	9412	0	4.57	1.70	1.18	0
repos24p	7	75	482	154	0.63	1.52	0.33	0.28	1338	0	0.66	0.36	0.32	0
repos25p	7	77	496	156	0.60	1.67	0.43	0.37	1416	0	0.70	0.38	0.37	0
repos27p	7	79	571	188	1.40	3.34	0.56	0.52	1846	0	2.23	0.58	0.43	0
repos28p	7	90	618	189	1.77	3.69	0.68	0.54	2096	0	2.15	0.62	0.54	0
repos30p	8	126	1450	265	3.22	3.90	1.47	0.98	7046	0	3.05	1.73	1.34	0
repos31p	8	130	1362	152	7.61	9.02	2.16	1.87	7148	0	8.57	3.65	2.03	0
repos32p	8	144	1501	170	6.49	4.95	2.20	3.32	7642	0	4.69	2.32	1.11	0
repos34p	9	304	10577	344	633.44	263.54	44.96	46.65	77400	0	143.80	28.59	14.13	0
repos36p	9	364	11972	1048	TIME	424.25	128.14	90.65	83564	1	244.93	52.59	34.81	1
repos39p	9	379	11666	1109	TIME	533.97	80.57	44.26	84682	1	392.23	75.74	34.63	2
repos40p	9	379	11666	1109	TIME	535.65	82.52	59.92	84682	1	389.24	77.45	48.10	4
repos42p	11	279	6596	1423	TIME	TIME	219.32	177.81	49538	21	966.30	93.20	45.86	37
repos43p	11	320	13058	1013	TIME	TIME	TIME	TIME	132346	20	TIME	473.83	263.49	27
repos44p	11	328	13705	1108	TIME	TIME	TIME	TIME	135388	26	TIME	879.17	216.46	28

Table 2.1: Comparison of reformulated MIP (MIP), reformulated MIP with lazy capacity constraints (Lazy MIP), branch-and-price on the revised MIP (B+P), branch-and-price on the reformulated MIP (Ref. B+P) and with lazy capacity constraints (Ref. B+P+C). Instances unsolved in 1 hour are marked TIME

One reason for this is that the data is such that there are very few opportunities for a ship to exceed its capacity since the sizes of the demands are very small compared to the capacity of the ship. As such, often a capacity violation occurs when many different demands are being carried. By leaving out all these constraints and adding in only the ones we need and only when we need them, we are able to solve the sub-problems significantly faster, and thus find the optimal solution much sooner.

The MIP formulations do not scale well with the problem size, with no MIP formulations solving the two largest instances in one hour. This is to be expected, as the numbers of variables and constraints in the MIP formulations increase much faster than those of the B+P formulations. However, between the MIP formulations, we see the same trend as before: the reformulated model is more powerful than the original model, and the lazy formulation is more powerful again, but the improvement is less dramatic in this case. This is because we are solving one aggregated problem rather than several independent sub-problems, so the capacity constraints represent a smaller proportion of the model than in the B+P formulations.

2.2.8 Discussion

For the LSFRP, applying the capacity constraints lazily is always a benefit for any difficult instances. While the lazy formulations may perform slower for the smaller instances, this should not be seen as a problem because they are effectively trivial instances. Again, the significant difference in the time taken to solve the most difficult instances is due to the reduction in the number of constraints that must be considered by the solver. This increases the number of simplex iterations that can be performed each second, and potentially reducing the number of simplex iterations required to process a node, allowing nodes to be explored more quickly.

The models presented in this chapter are all capable of considering the movement of empty containers. It is simply a matter of creating a number of demands for them, with specific origins and destinations. The only problem exists in the data: it is never profitable to carry empty cargo. This is because the sum of the cheapest loading and unloading costs is greater than the highest revenue from moving an empty cargo, which means the costs will never be covered by the revenue. By raising the profitability of moving empty cargo, improved solutions are obtained in a similar amount of time.

Branch-and-price scales far better for this problem than any of the MIP formulations, but the main reason for this is that the master problem returns naturally integer solutions. If this were not the case, we would need to implement a branching framework that would slow the process down considerably. As mentioned in Section 1.2.2, a complete branch-and-price implementation does not take full advantage of the solvers, which means that, given enough development time, the MIP may one day be more powerful than the branch-and-price implementation.

Problems with constraints that are mostly unnecessary are some of the easiest to apply lazy constraints to, as one can simply remove the constraints in question then check said constraints in a callback whenever an integer solution is found. If one such constraint is violated, add it lazily and continue solving. However, deciding which constraints to make lazy is more difficult, and relies upon

the experience and intuition of the modeller.

In the next chapter, we explore a common technique called Benders decomposition which often benefits from the use of lazy constraints. The use of lazy constraints in the LSFRP only concerns the feasibility of the solutions, while for Benders decomposition, lazy constraints can also be used to approximate the objective function. As such, Benders decomposition can be generalised to cover any problem that may benefit from lazy constraints.

The following publications have been incorporated as part of Chapter 3.

1. [3] **Robin H. Pearce** and Michael Forbes, Disaggregated benders decomposition for solving a network maintenance scheduling problem, *Journal of the Operational Research Society*, 70 (6), pp. 941-953, 2019

Contributor	Statement of contribution	%
Robin H. Pearce	writing of text	100
	proof-reading	80
	theoretical derivations	80
	numerical calculations	100
	preparation of figures	100
	initial concept	50
Michael Forbes	proof-reading	20
	supervision, guidance	100
	theoretical derivations	20
	initial concept	50

2. [2] **Robin H. Pearce** and Michael Forbes, Disaggregated Benders decomposition and branch-and-cut for solving the budget-constrained dynamic uncapacitated facility location and network design problem, *European Journal of Operational Research* 270 (1), 2018

Contributor	Statement of contribution	%
Robin H. Pearce	writing of text	100
	proof-reading	80
	theoretical derivations	70
	numerical calculations	100
	preparation of figures	100
	initial concept	70
Michael Forbes	proof-reading	20
	supervision, guidance	100
	theoretical derivations	30
	initial concept	30

Chapter 3

Benders Decomposition

> For the great doesn't happen through impulse alone, and is a succession of little things that are brought together.
>
> *Vincent van Gogh*

In integer programming, many problems can be difficult to solve on useful instances. This means techniques for decomposing the problems into smaller, easier to manage problems are desired. The two main options for this are row-generation or column-generation. One of the main techniques is known as Benders decomposition, so named after Jacques F. Benders [24], and involves relaxing a difficult MIP by projecting out a number of variables, and instead generating constraints to replace them, which makes it a row-generation procedure. While the base technique is more than 50 years old, there have been many improvements suggested over the years, and it appears to be undergoing a revival as many methods that take advantage of improvements in parallel computing and new software tools are making Benders decomposition significantly more effective.

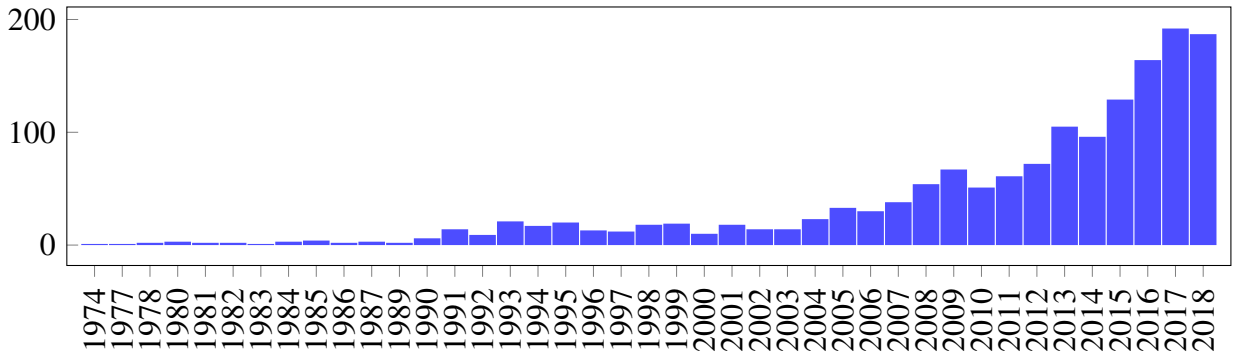


Figure 3.1: Number of papers on Benders decomposition per year according to Clarivate Web of Science [6]

3.1 History

The late 1950s and early 1960s were a time of great development in the theory of mathematical programming. The first hints of branch-and-bound arrived at the start of the 1960s [20], and Gomory's cuts for solving mixed-integer programming problems had only just been published [19]; however, both methods were limited by the size of the problem they could successfully be applied to. There was still very little published knowledge around how to solve large programming problems with integer variables.

Benders' paper from 1962 entitled "Partitioning procedures for solving mixed-variables programming problems" lays out an idea for decomposing difficult problems with a certain structure [24]. In this paper, Benders decomposition is proposed quite generally, but the author gives a specific example of a "mixed-integer programming problem in which certain variables may assume any value in a given interval, whereas others are restricted to integral values only". Benders decomposition is most commonly applied to this type of problem.

In his 1972 paper, Geoffrion further generalises Benders decomposition to any problem with a set of complicating variables, where the optimisation problem is "a much easier optimisation problem in [the other variables] when [the vector of complicating variables] is temporarily held fixed" [45]. While this extends the applicability of Benders decomposition beyond mixed-integer programming problems, the majority of studies involving Benders decomposition concern MIPs, and they will be the focus of this chapter.

There were only a handful of problems solved using Benders decomposition in the 1970s. During this decade, much of the focus was on improving the effectiveness of computational solvers. During the 1980s, more studies on Benders decomposition began to appear with the intention of improving its practicality. The most influential studies were released by T.L. Magnanti and R.T. Wong.

Magnanti and Wong introduced the idea of *Pareto-optimal* cuts: cuts that are not *dominated* by any other cuts [46]. The definitions and details around Pareto-optimal cuts are explored further in Section 3.4. In their paper, Magnanti and Wong propose a method for generating Pareto-optimal Benders cuts which is shown to accelerate Benders decomposition, greatly in some situations. Importantly, they apply Benders decomposition to the Uncapacitated Facility Location (UFL) problem, which is discussed further in Section 3.3.

Magnanti and Wong continue to demonstrate the ability of their algorithm with studies on network design [47] and transportation planning [48]. Despite the usefulness of their algorithm, it is not simple to implement and suffers from issues that affect its performance, such as a dependency on the Benders sub-problem. Papadakos [49] presents a modified version of the algorithm that removes its dependence on the Benders sub-problem, giving it more consistent performance. Even with this improvement, studies that apply Benders decomposition seldom consider or strive for Pareto-optimal Benders cuts. As we show in Section 3.4, there are alternatives to the Magnanti-Wong method for generating Pareto-optimal Benders cuts.

There are also a number of survey papers that cover the recent history of Benders decomposition

more thoroughly than this introduction. Costa (2005) covers a variety of network design problems that often form the basis of more complicated problems. Rahmaniani *et al.* (2017) focusses more on the implementation specifics. Many of the details covered in this chapter have been separately covered by Rahmaniani *et al.*, but with different context and levels of detail.

Today, there are two main improvements to Benders decomposition that are consistently proving to be powerful techniques: disaggregation of the Benders sub-problem, and embedding the whole process in a branch-and-cut framework. Disaggregating the sub-problem (breaking it into multiple, independent problems) and applying one Benders cut for each sub-problem tightens the solution space more than using aggregated cuts. For problems where the sub-problems are easier to solve than the master problem, using a branch-and-cut approach (*i.e.* implementing the Benders cuts as lazy constraints) is significantly more efficient.

3.2 Theory

Since all the problems covered in this chapter are MIPs, we present the theory of Benders decomposition for MIPs of the required structure. The paper by Geoffrion [45] provides a framework for Benders decomposition in general.

Benders decomposition is applicable to MIPs of the following form:

$$\min_{x,y} \quad c^T x + f^T y \tag{3.1}$$

Subject to:

$$Ax + By \geq b \tag{3.2}$$

$$x \geq 0, y \in Y \tag{3.3}$$

where Y is some restricted set of potential solutions. Typically, Y at least constrains y to be integer, but it also includes any constraints on the y-variables that do not involve the x-variables. The only reason the x-variables are stated to be continuous is so the sub-problem is an LP, which is beneficial for reasons that will soon become apparent. If the x-variables are not continuous, there are still ways of solving the problem using Benders decomposition.

While it is possible for many MIPs to be transformed into this form by arbitrarily partitioning the variables into two sets, it does not yield any improvement in practice, as the power in using Benders decomposition comes from exploiting certain properties that are not present in every problem. Benders notes that the method "... may be advantageous if the structure of the problem indicates a natural partitioning of the variables" [24]. Over time, this has become known as the following:

> The variables $y \in Y$ are considered "complicating" variables, as if a solution for them is known, the problem reduces to a simple linear program that may be efficiently solved using well-established techniques. [45, 51–54]

We will consider the case where the problem reduces to an LP, but it may also be a dynamic program or other optimisation problem, so long as it is convex. It must be convex, because Benders decomposition works by removing the objective component and all constraints containing the x-variables, instead approximating their objective contribution by some auxiliary variable, θ. We then add constraints on the value of this new variable that depend upon the master problem variables, so as to always provide an underestimate (in the case of minimisation) of the actual contribution of the x-variables.

The procedure is then to solve the reduced problem, which is smaller and easier to solve, and when a feasible solution to this problem is found, a sub-problem is solved to ensure the value of θ correctly estimates the contribution of the x-variables. If it does not, a new cut is added to update this approximation. This continues until a solution is found to the master problem where θ gives the correct value, at which point the solution is considered a valid integer solution to the original problem.

First, we begin with the original problem (3.1-3.3). The x-variables are removed from the problem, and instead their contribution to the objective function is approximated by a new variable, θ. This gives us the initial *Benders master problem* (BMP), also known as the relaxed master problem:

$$\min_{\theta,y} \quad \theta + f^T y \tag{3.4}$$

Subject to:

$$y \in Y \tag{3.5}$$

This problem is now smaller than the original problem, in that it has fewer variables and constraints, and thus should be easier to solve. There are two problems with this: the variable θ begins unconstrained; and the relaxed master problem may permit solutions that are infeasible in the original problem. These two issues are covered in Sections 3.2.1 and 3.2.2 respectively.

3.2.1 Optimality

To ensure θ gives the correct objective value for the current master problem solution, y^*, we may need to add additional constraints. To achieve this, we construct the *Benders sub-problem* (BSP):

$$\min_{x} \quad c^T x \tag{3.6}$$

Subject to:

$$Ax \geq b - By^* \tag{3.7}$$

$$x \geq 0 \tag{3.8}$$

Since this is a linear program, we can find its dual program and corresponding dual solutions. The *Benders dual sub-problem* (BDSP) is:

$$\max_{u} \quad u^T(b - By^*) \tag{3.9}$$

Subject to:

$$A^T u \leq c \tag{3.10}$$

$$u \geq 0 \tag{3.11}$$

where u is the vector of dual variables. By strong duality, $u^{*T}(b - By^*) = c^T x^*$, *i.e.* the optimal solution to the dual problem is the same as the objective value of the BSP for the current master problem solution.

If the value of y^* in the RHS of the BSP were instead $y^* + \Delta y$, then the objective value of the dual problem would become $u^{*T}(b - B(y^* + \Delta y))$. Since y^* does not appear in the dual constraints (3.10), u^* is still a feasible solution, and $u^{*T}(b - B(y^* + \Delta y)) \leq u'^T(b - B(y^* + \Delta y)) \leq c^T x'$, where u' is the optimal solution to the problem $\max_u\{u^T(b - B(y^* + \Delta y)) | A^T u \leq c, u \geq 0\}$ and x' is the optimal solution to the problem $\min_x\{c^T x | Ax \geq b - B(y^* + \Delta y), x \geq 0\}$. This means that $u^{*T}(b - By)$ provides an underestimate of the objective value of the BSP for all master problem solutions $y \in Y$, and the constraint

$$\theta \geq u^{*T}(b - By) \tag{3.12}$$

is valid. We add this constraint to the master problem to update our approximation. This cut is known as a *Benders optimality cut*. Because this provides an underestimate for all potential master problem solutions, it will not cut off any legal solutions to the original problem. Thus, if a solution to the master problem is found and the value of θ is equal to the objective of the sub-problem, this is a candidate solution to the original problem.

This constraint not only corrects the value of the approximation for the current solution, but also provides information about how the objective value will change as y changes, often providing correct estimates for a number of other master problem solutions. The effectiveness of Benders decomposition relies upon the *strength* of the Benders cuts produced. Notions of dominance of Benders cuts have been investigated over the past 30 years, and will be covered in Section 3.4.

3.2.2 Feasibility

When removing the sub-problem components from the original problem, the relaxed master problem is now allowed to take more solutions than the original problem because it is no longer constrained by the feasibility of the sub-problems. In the ideal case, there exist constraints that are implicit in the original problem that can be made explicit in the master problem to ensure feasibility. A good example of where this occurs is the Uncapacitated Facility Location (UFL) problem, which is described in Section 3.3.

If this is not the case, then at some point we may find the sub-problem is infeasible for a given master problem solution, y^*. If this occurs, then instead of adding a Benders optimality cut, we must

add a *Benders feasibility cut*. This new cut is designed to prevent the master problem from finding solutions that render the sub-problem infeasible again. There are several different ways of constructing Benders feasibility cuts.

The first is the traditional Benders feasibility cut. By duality theory, if the BSP is infeasible, then the BDSP is either unbounded or infeasible. The constraints of the BDSP do not depend upon the master problem solution, y^*, which means that if the BDSP is infeasible for any master problem solution, it will be infeasible for all master problem solutions. This in turn means the BSP is infeasible or unbounded for all master problem solutions, and thus the original problem is either infeasible or unbounded. So if we assume the original problem has a feasible solution, then whenever the BSP is infeasible, the BDSP is unbounded.

If the BDSP is unbounded, there is some direction u^* such that $u^{*T}(b - By^*) > 0$ and $A^T \alpha u^* \leq c$ for any $\alpha \in \mathbb{R}^+$, which means we can move in the direction u^* without ever hitting a constraint. To prevent this, we add the traditional Benders feasibility cut:

$$(b - By)^T u^* \leq 0 \tag{3.13}$$

which cuts off the unbounded direction. If this cut bounds the dual sub-problem, then it will be feasible and bounded, and hence the primal sub-problem will be feasible and bounded. This type of cut is best detailed by Geoffrion [45] and, in conjunction with Benders optimality cuts, can solve any Benders decomposition problem to optimality *given sufficient resources*. This qualifier is necessary since it has been noted that traditional Benders feasibility cuts are ineffective in many cases [55].

The most common applications of Benders decomposition tend to have mostly binary variables in the master problem, as they represent a number of decisions to be made (open this facility, close that arc). As such, practitioners of Benders decomposition have started using a new type of feasibility cut, a *combinatorial Benders cut*. Introduced by Codato and Fischetti (2006), the idea is to find a minimal source of infeasibility, called a *minimal (or irreducible) infeasible subsystem* (MIS or IIS). The Gurobi documentation concisely describes an IIS:

> An IIS is a subset of the constraints and variable bounds of the original model. If all constraints in the model except those in the IIS are removed, the model is still infeasible. However, further removing any one member of the IIS produces a feasible result. [44]

Such constraints can often be "removed" by changing the value of a master problem binary variable, especially in the case of big-M constraints. By adding a constraint that the sum over such binary variables (or possibly one minus the variables) must be greater-than-or-equal-to 1, we are saying at least one of these constraints must be relaxed, which is likely to remove the infeasibility. These constraints often have intuitive explanations and can be constructed using an algorithm if one does not have the ability to compute an MIS or IIS. An example of these is the sub-tour elimination constraints of the Travelling Salesman Problem, explored in more detail in Section 4.1.

Note that there may be multiple IISs in a given infeasible problem. This means it may be possible to add multiple feasibility cuts for the same infeasible problem. These cuts have been shown to be

more effective than traditional Benders feasibility cuts [2, 52, 55] and are easy to obtain using the functionality available in Gurobi and CPLEX. All examples of Benders decomposition in this thesis that require feasibility cuts use combinatorial Benders cuts.

3.2.3 Procedure

The recipe for implementing Benders decomposition has changed over the years, but the general idea has remained the same. The original algorithm for implementing Benders decomposition is as follows:

1. Start with the original problem
2. Remove x-variables and replace objective contribution with approximation, θ. This new problem is the Benders Master Problem (BMP)
3. Construct the Benders sub-problem (BSP)
4. Solve the BMP and retrieve the optimal solution, (y^*, θ^*)
5. Solve BSP for the current solution, (y^*, θ^*), and retrieve the optimal solution, x^*
6. If BSP is infeasible, add a Benders feasibility cut and go to step 4
7. If $c^T x^* = \theta^*$ (or approximately equal to), terminate with the optimal solution to the original problem, (x^*, y^*)
8. Retrieve the dual variables u^* from the BSP, use them to add an optimality cut to the BMP, and go to step 4

As mentioned in the introduction to this chapter, there are two main improvements to this algorithm which have greatly improved the effectiveness of Benders decomposition: disaggregation of the Benders sub-problem, and embedding the whole process in a branch-and-cut framework.

Disaggregation of the Benders sub-problem

Consider the Benders sub-problem (BSP) (3.6-3.8). If part of the constraint matrix A is block-diagonal, we can partition this problem into a set of separate optimisation problems, one corresponding to each block. This occurs frequently in practice, where the sub-problem contains a number of independent decisions or scenarios that can be solved individually. Assume we can partition the BSP into a series of sub-problems indexed by $j \in \{1, ..., J\}$, and A_j, c_j, $(b - By^*)_j$ are the corresponding portions of data and x_j are the relevant variables. Then we have J problems of the form:

$$\min_x \quad c_j^T x_j \tag{3.14}$$

Subject to:

$$A_j x_j \geq (b - By^*)_j \tag{3.15}$$

$$x_j \geq 0 \tag{3.16}$$

Each of these sub-problems now produces corresponding dual variables, which can be used to add Benders optimality cuts for a variable, θ_j. Now, the *disaggregated* Benders master problem is:

$$\min_{\theta, y} \quad \sum_j \theta_j + f^T y \tag{3.17}$$

Subject to:

$$\theta_j \geq (u_j^{*k})^T (b - By)_j \quad \forall k \in \{1, ..., |K|\}, \forall j \in \{1, ..., J\} \tag{3.18}$$

$$y \in Y \tag{3.19}$$

where $|K|$ is the number of sets of Benders optimality cuts generated. The benefits from this formulation are numerous. The first is that the sub-problems are individually easier to solve than the original sub-problem, and solving multiple smaller sub-problems is often faster than solving one aggregated sub-problem. The second is that the approximation provided by the collection of θ_j is tighter than the associated aggregated version. This is because the $(u_j^{*k})^T (b - By)_j$ terms are being maximised individually, then the sum of their maxima is taken to provide the approximation of θ. In the aggregated version, the $(u_j^{*k})^T (b - By)_j$ terms are first summed together, and then the maximum is taken. The sum of the maxima will always exceed the maximum of the sums.

Proposition 1. *For a set of dual variables u^*, the disaggregated Benders cuts will be tighter than the aggregated Benders cut.*

Proof. Let y' be an arbitrary solution to the BMP. Then define $a_j^k = (u_j^{*k})^T (b - By')_j$, that is, the estimate of the contribution of θ_j by cut k. Then $\theta_j = \max_k \{a_j^k\}$ and $\theta = \max_k \left\{ \sum_{j \in \{1,...,J\}} a_j^k \right\}$. For any specific j and for every k, $a_j^k \leq \max_k \{a_j^k\}$. Now, summing both sides over j, we have:

$$\sum_{j \in \{1,...,J\}} a_j^k \leq \sum_{j \in \{1,...,J\}} \max_k \{a_j^k\}, \tag{3.20}$$

or rather:

$$\theta \leq \sum_{j \in \{1,...,J\}} \theta_j, \tag{3.21}$$

so the disaggregated formulation will always be tighter than the aggregated version. □

Branch-and-cut

In the majority of problems that are suitable for Benders decomposition, the sub-problems are much easier to solve than the master problem. In this case, it is far more efficient to embed Benders decomposition in a branch-and-cut framework. That is, rather than solving the master problem to

optimality, adding Benders cuts, and solving the master problem again, one simply solves the master problem once and evaluates the sub-problems at nodes of the branch-and-bound tree, adding Benders cuts where necessary. Each time a Benders cut is added, the tree is updated appropriately, *i.e.* nodes that are now rendered infeasible or non-optimal are removed.

It is difficult to determine who first embedded Benders decomposition in a branch-and-cut framework, but Fischetti attributes it to Miliotios in the 1970s by folklore [57]. One of the first publications to demonstrate embedding Benders decomposition in a branch-and-cut framework is that by Geoffrion and Graves in 1974, where they solve the master problem until a feasible solution better than their current upper bound is found, rather than solving it to optimality [58]. More recently, a number of studies have implemented Benders decomposition in a branch-and-cut framework and found it to be very efficient [52, 59, 60]. The main question is when to separate cuts. This is covered in the last subsection of 3.7.3. The next section contains an example application of Benders decomposition with details on some of the techniques mentioned.

3.3 The Uncapacitated Facility Location Problem

The Uncapacitated Facility Location (UFL) problem is one of the simplest problems to benefit from Benders decomposition. Benders decomposition was first applied to the UFL problem by Balinski [61, 62], and has since been improved upon by others, most notably by Magnanti and Wong [46] and Fischetti, Ljubić and Sinnl [60]. It is an ideal problem for exposition, not just because it is simple and its decomposition is intuitive, but also because it allows disaggregation of the sub-problems.

The UFL problem concerns a set of locations, N, and a set of potential facilities, F. A number of facilities must be opened, and each location must be connected to a facility. The objective is to minimise the combined non-negative costs of opening facilities, f_j, and connecting locations to facilities, c_{ij}. The UFL is described as:

Sets

N Set of locations

F Set of potential facilities

Data

c_{ij} Cost of connecting location $i \in N$ to facility $j \in F$

f_j Cost of opening facility $j \in F$

Variables

x_{ij} 1 if location $i \in N$ is connected to facility $j \in F$, 0 otherwise

y_j 1 if facility $j \in F$ is open, 0 otherwise

Objective

$$\min \sum_{i \in N} \sum_{j \in F} c_{ij} x_{ij} + \sum_{j \in F} f_j y_j \tag{3.22}$$

Constraints

$$\sum_{j \in F} x_{ij} \geq 1 \qquad \forall i \in N \tag{3.23}$$

$$x_{ij} \leq y_j \qquad \forall i \in N, \forall j \in F \tag{3.24}$$

$$x_{ij} \in \{0,1\}, y_j \in \{0,1\} \qquad \forall i \in N, \forall j \in F \tag{3.25}$$

Constraints (3.23) say that each customer must be connected to at least one facility and constraints (3.24) only allow customers to be connected to open facilities. Note that in this formulation all variables are binary. It is possible to relax the x-variables to be continuous and the optimal solution will still be integer. This is easy to see, since if all values of y are binary, for each location $i \in N$, the optimal solution will have $x_{ij} = 1$ for the cheapest open facility $j \in F$ and all other values of x will be 0.

For the UFL problem, the disaggregation of the Benders sub-problem is intuitive, since if one knows which facilities are already open, the problem is trivial to solve: connect every location to its closest open facility. Thus, the job of the BMP is to decide which facilities to open, and the BSP updates the approximation of the connection costs.

We replace $\sum_{j \in F} c_{ij} x_{ij}$ with θ_i and remove any constraints that contain x-variables. The BMP is thus:

$$\min \sum_{i \in N} \theta_i + \sum_{j \in F} f_j y_j \tag{3.26}$$

Subject to:

$$\sum_{j \in F} y_j \geq 1 \tag{3.27}$$

$$\theta_i \geq 0, y_j \in \{0,1\} \qquad \forall i \in N, \forall j \in F \tag{3.28}$$

All of the original constraints have been removed, as they all contain x-variables. To ensure feasibility of the sub-problems, at least one facility must be open, so we introduce constraint (3.27). This constraint was implied by constraints (3.23-3.24), but it must now be added explicitly. Note that this constraint is sufficient to ensure feasibility of all sub-problems, so we will not need to generate Benders feasibility cuts for this problem.

Solving this problem results in a selection of the potential facilities to open, and the θ_i variables give estimates of the connection costs for that selection. Initially, since θ_i is only constrained to be non-negative, $\theta_i = 0$ which is incorrect. We then solve the sub-problems, one for each location $i \in N$:

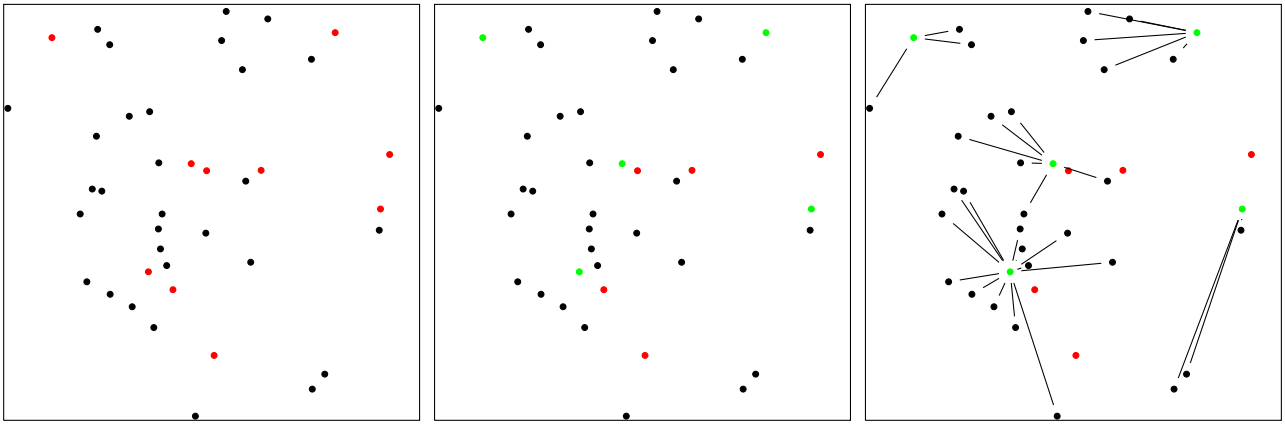
$$\min \sum_{j \in N} c_{ij} x_{ij} \tag{3.29}$$

Subject to:

$$\sum_{j \in F} x_{ij} \geq 1 \tag{3.30}$$

$$x_{ij} \leq y_j^* \qquad \forall j \in F \tag{3.31}$$

$$x_{ij} \geq 0 \qquad \forall j \in F \tag{3.32}$$



(a) Locations (b) Master solution (c) Sub-problem solution

Figure 3.2: Example of UFL problem. (a) Black represents customer locations and red is potential facilities (b) Green is open facilities and red is closed facilities (c) The customers are connected to their closest open facilities

The solution to this problem is trivial: x_{ij} is 1 if j is the closest open facility to location i and 0 otherwise. To generate a Benders optimality cut, we must solve the dual of the BSP. Let λ_i and π_{ij} be the dual variables associated with constraints (3.30) and (3.31) respectively. Then the dual of the BSP for location $i \in N$ is:

$$\max \lambda_i - \sum_{j \in N} \pi_{ij} y_j^* \tag{3.33}$$

Subject to:

$$\lambda_i - \pi_{ij} \leq c_{ij} \qquad \forall j \in N \tag{3.34}$$

$$\lambda_i \geq 0, \pi_{ij} \geq 0 \qquad \forall j \in F \tag{3.35}$$

This problem has an *analytic solution*, one that can be found without explicitly solving the dual of the BSP:

$$\lambda_i^* = \min\{c_{ij} | y_j^* = 1\} = c_{ij(i)} \tag{3.36}$$

$$\pi_{ij}^* = \max(0, c_{ij(i)} - c_{ij}) \tag{3.37}$$

These values have a natural interpretation: λ_i is the cost of connecting location i to a facility for the solution y^*, and π_{ij} is the potential saving from connecting location i to facility j. The Benders optimality cut is thus:

$$\theta_i \geq \lambda_i^* - \sum_{j \in N} \pi_{ij}^* y_j, \tag{3.38}$$

i.e. the cost of connecting location i to a facility is the current cost minus any potential savings. When these cuts are added to the BMP, the approximation of θ_i improves and the current master problem solution is rendered infeasible. As the solver continues to encounter integer solutions, the

approximations are checked and updated if necessary. This continues until a solution to the master problem is found where $\theta_i = \sum_{j \in F} c_{ij} x_{ij} \; \forall i \in N$, *i.e.* the approximations are all correct. This is then accepted as an incumbent integer solution, and the solver continues to branch-and-bound until the optimal solution is found.

3.3.1 Implementation details

There are a number of other small improvements to Benders decomposition that are useful in a number of problems. They are to *warm start* the model, suggest *user heuristics* and choose a *branching direction*.

Warm start or initial cuts

The master problem of a Benders decomposition is a relaxation of the original model since constraints and variables have been removed. While this makes it easier to process branch-and-bound nodes for the smaller model, the solution to the LP-relaxation may be worse, and so more branching may be required. As part of this, any auxiliary variables that are introduced will start with only the standard non-negativity constraints (and an upper bound in the case of maximisation problems). This means the approximations will be wrong for many of the early integer solutions and many Benders cuts will be added at the beginning.

A way to overcome this is to *repair* the LP-relaxation by adding some initial Benders cuts. These may either be sensible Benders cuts to add, as demonstrated in Section 3.5.3 (under the Initial Cuts subheading), or the more generic *warm start*. First introduced by McDaniel and Devine, the idea is to solve the LP-relaxation and add a Benders cut for the (possibly fractional) solution repeatedly [63]. As the LP-relaxation is faster to solve than the MIP, it quickly builds up a collection of Benders cuts that close the gap between the LP-optimum of the Benders master problem and the LP-optimum of the original problem. This is equivalent to adding Benders cuts at the root node of the branch-and-bound tree, or to solving the LP-relaxation using Benders decomposition.

Once the warm start is complete, it is possible to remove any Benders cuts that are not *active* (have a non-zero slack). This leaves the solver with a small set of Benders cuts that give a tight LP-relaxation, thus minimising the number of constraints in the model at this point without sacrificing progress. There are scenarios where warm starts do not have a huge impact, but in most problems they are necessary for Benders decomposition to be more effective than the MIP solvers.

It is also possible to construct warm start cuts analytically, as is done in Section 3.7.4. For some problems this is a simple task, but for others it can be very difficult. Analytic warm start cuts can provide strong benefits over *default* warm start cuts (generated using dual variables returned by the LP solver) as shown in Section 3.7.5, but it is possible that the algorithm for generating these cuts may be inefficient enough to remove all benefit from using analytic cuts.

More generally, a warm start refers to adding Benders cuts at the root node of the branch-and-bound tree, which is usually a fractional solution. We also add Benders cuts at other nodes of the

branch-and-bound tree, whenever we find an integer solution. It is possible to add Benders cuts at any node of the branch-and-bound tree, whether it is fractional or integer, so long as the node is feasible. The main benefit of this is that the tree may be pruned more aggressively, potentially reducing the amount of nodes that need to be explored. The drawbacks are the extra time spent solving the Benders sub-problems and the additional Benders cuts being carried by the solver. Often the drawbacks outweigh the benefits, as discussed at the end of Section 3.7.3.

User heuristics

When adding a Benders optimality cut, we are discarding a solution that is infeasible in the original problem because the values of the auxiliary variables are incorrect. When this happens, it is possible to construct a new solution that is feasible in the original problem by keeping the master problem solution, and setting the values of the approximation variables to the values calculated by the sub-problem(s). If this solution is better than the incumbent solution, the solver will benefit from knowing it; however, there are no guarantees that it will find this solution by itself. In these cases, we suggest this solution as a *user-generated heuristic* to the solver for consideration. This is very important in the UFL problem, but it is not always useful, especially if the solver is able to construct these solutions itself.

Branching direction

By default, Gurobi and CPLEX choose which branches to explore in the branch-and-bound tree based on their analysis. It is possible to override this and choose to always explore a particular branch first, either up or down. When all the variables in the master problem are binary, these correspond to setting variables to 1 or 0 respectively. In the context of the UFL problem, setting a variable to 0 corresponds to forcing a facility to be closed, while setting a variable to 1 forces a facility to be open.

In theory, by forcing a number of facilities to be open, the objective function will close many of the others, which leads to an integer solution more quickly than forcing facilities to be closed. When Benders cuts are only added at integer solutions, finding such solutions earlier in the solution process allows us to cut off parts of the search space earlier, potentially helping to find the optimal solution more quickly. Thus, setting a branching direction of 1 may yield some benefit. This is problem-dependent, and relies on one branch being *stronger* than another.

3.3.2 Results

We now compare three different formulations: MIP, DBD and DBD+. MIP is a direct MIP implementation, DBD is a standard disaggregated Benders decomposition formulation with no improvements, and DBD+ is disaggregated Benders decomposition with warm start, removing slack warm start cuts, user heuristics and a branching direction of 1. All code is implemented in Python 3.5 (64-bit) as part of the Anaconda 4.1.1 distribution and uses the Gurobi 7.0.1 (64-bit on 8 threads) solver package [44], running on a distributed computing machine. The machine contains Intel Xeon processors (2.4GHz)

with 8 cores and 56GB of RAM assigned to each job. This amount of RAM is more than necessary, and is requested so it is not a constraint. All instances are given 1 hour in which to prove optimality.

Table 3.1 contains a summary of the performance of the three implementations on four different instance sets. The number of instances in each set is shown in brackets. The instance sets have all been sourced from the UFLLib [64], a library of UFL problem instances, some of which remain unsolved. For each of the implementations, we report the number of instances solved to optimality. We also report the average time in seconds and the average number of branch-and-bound nodes explored by each implementation. These averages are only taken over instances that were solved to optimality by all three implementations to provide an accurate comparison of performance.

For the KoerkelGhosh-sym instance set, there is a clear pattern: DBD+ is the strongest implementation and MIP is the weakest. This is a typical example of how Benders decomposition performs: applying disaggregated Benders decomposition in a branch-and-cut framework provides a benefit, but there is more that can be achieved through improvements such as warm starting the solver or supplying user heuristics.

The Large Duality Gap instance sets are designed specifically to be difficult for branch-and-bound-based techniques, becoming progressively harder from set A to set C. This is evident in the performance of the MIP, which solves instances in set A more quickly than set C. Benders decomposition solves more quickly than the MIP for all instance sets; however, in this case the extra improvements such as warm-starting and user heuristics do not have a positive impact.

There are some instances where such methods will not make any difference to the solution, and adding them only wastes time that could have been spent by the solver processing more nodes. To see this, compare the number of branch-and-bound nodes required to prove optimality for the KoerkelGhosh instances. DBD+ explores far fewer nodes than DBD, showing that the improvements had a positive impact. In the Large Duality Gap instances, however, the number of nodes explored

Instance set	Formulation	Optimal	Time (s)	Nodes
KoerkelGhosh-sym (45)	MIP	6	2066.76	26331.6
	DBD	12	212.47	54800.5
	DBD+	14	45.14	12082.5
LargeDualityGapA (30)	MIP	30	40.63	31354.6
	DBD	30	10.48	43161.4
	DBD+	30	12.41	39978.6
LargeDualityGapB (30)	MIP	30	78.73	69969.9
	DBD	30	4.91	16725.6
	DBD+	30	6.48	16750.7
LargeDualityGapC (30)	MIP	30	758.36	661190.1
	DBD	30	129.26	479610.5
	DBD+	30	157.87	498991.3

Table 3.1: Comparison of three formulations for solving the UFL problem. The number of instances in each instance set is shown in brackets. Reported are the number of instances solved to optimality, the average solution time and the average number of branch-and-bound nodes explored across the instances which all formulations solved to optimality

by DBD and DBD+ is similar, almost exactly the same in the case of set B. This shows that the improvements do not help reduce the number of nodes to explore, and thus do not assist in finding the optimal solution more quickly.

Another comparison of the number of branch-and-bound nodes is interesting. In theory, Benders decomposition is a relaxation of the original model, which means the LP-relaxation will be less tight and thus more branch-and-bound nodes may need to be explored to find the optimal solution. If the solver is warm-started, then the LP-relaxation will be tighter and fewer nodes would have to be explored, but potentially still more than the original model, as the LP-optimum of the Benders model cannot possibly be better than that of the original model. Exploring more nodes in Benders decomposition is not necessarily a problem because the master problem is much smaller than the original model, and nodes are explored much more quickly, often orders of magnitude faster.

The KoerkelGhosh-sym instance set behaves mostly as one would expect: Benders decomposition requires more nodes to be explored than the MIP, and warm-started Benders decomposition explores fewer than Benders decomposition without a warm start. The interesting part is that warm-started Benders decomposition explores fewer nodes than the MIP. The same behaviour is evident in the Large Duality Gap instance sets B and C, where Benders decomposition explored far fewer nodes than the MIP.

We speculate the reason for such behaviour is the automated cutting planes and other improvement techniques built into the solvers. The master problem shared by the Benders models is much smaller than the original MIP model, so Gurobi is able to apply cutting planes and pre-processing algorithms more effectively, leading to a tighter model that solves faster. This behaviour is evident elsewhere, such as in the network maintenance scheduling problem [3] explored in Section 3.5. It is also possible that this behaviour is a result of finding good integer solutions earlier, thus cutting off more sub-trees that do not contain the optimal solution. This may occur due to the above reasons, or simply because the model is much smaller.

3.3.3 Cut selection for the UFL

The Benders optimality cuts for the UFL can be identified by which facility j was chosen for the value of λ_i (*i.e.* $\lambda_i = c_{ij}$), usually the closest open facility to i. We say this cut is *centred about facility* j. Balinski [62] and Magnanti and Wong [46] note that one could also use the cut centred about the second-closest open facility, as it gives the correct estimate for the current solution. So which cut should we use?

Consider a toy example with four potential facilities, labelled A, B, C and D. Facilities B and D are currently open, and facilities A and C are closed. The cost of connecting location i to facilities A, B, C and D are 20, 30, 40 and 50 respectively. We first consider the cut centred about the closest open facility, then the cut centred about the second-closest open facility.

The current closest facility to i is B at a distance of 30, so $\lambda_i = 30$. If facility A were opened, then i could be served for a cost of 20 instead, so $\pi_{i\mathrm{A}} = 10$. Since the cost of connecting location i to

facilities B, C and D is no less than the cost of connecting to B, $\pi_{ij} = 0$ for $j \in \{\text{B, C, D}\}$. This gives the following Benders cut:

$$\theta_i \geq 30 - 10y_{\text{A}}, \tag{3.39}$$

which says that the cost of connecting location i to a facility is at least 30, unless we open facility A, at which point it will cost only 20. This cut will give the correct objective value whenever at least one of A or B is open. However, if A and B are both closed, this cut will underestimate the objective value, since i will be connected to either C or D, both of which cost more than 30.

If we take the cut centred about D, then we get $\lambda_i = 50$, $\pi_{i\text{A}} = 30$, $\pi_{i\text{B}} = 20$, $\pi_{i\text{C}} = 10$ and $\pi_{i\text{D}} = 0$. This yields the Benders cut

$$\theta_i \geq 50 - 30y_{\text{A}} - 20y_{\text{B}} - 10y_{\text{C}}, \tag{3.40}$$

which also gives the correct objective value of 30 for the current configuration (since $y_{\text{A}} = y_{\text{C}} = 0$). This cut also holds as long as at most one facility in {A,B,C} is open. If more than one facility from this set is open, the cut will underestimate the objective value.

In general, a Benders cut for the UFL centred about a facility j will give the correct approximation as long as no more than one facility closer than j is open, and at least one facility at a distance less than or equal to j is open. This means that each Benders cut will give the correct objective value for a number of solutions and underestimate it for all others. Since there are only $2^4 = 16$ master problem solutions for this toy example, we can compare these two different cuts to see which master problem solutions they give the correct objective value for.

Figure 3.3 shows which master problem solutions are covered by the cuts centred about facilities B and D. The black squares represent the only infeasible master problem solution. Notice that the cut centred about B covers more master problem solutions than the one centred about D, but the cut centred about D covers solutions not covered by the cut centred about B. Is the cut centred about B *stronger* than the cut centred about D? Are any Benders cuts for the UFL stronger than the others? With the exception of the trivial cut ($\theta_i \geq \min_j c_{ij}$), the answer is *no*, and the reason is because all Benders cuts for the UFL problem are *Pareto-optimal*.

y_A	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
y_B	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
y_C	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
y_D	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Centred about B	■				✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Centred about D	■	✓	✓	✓	✓	✓			✓	✓						

Figure 3.3: Comparison of master problem solutions where different cuts give the correct objective value

3.4 Pareto-optimality

The notion of Pareto-optimality of Benders cuts was introduced by Magnanti and Wong [46]. The following definitions are derived from their original description, and are taken directly from [2]. The Benders cuts are all defined for a minimisation problem, but the theory holds for a maximisation problem by reversing inequalities appropriately. Let $\bar{\theta}^a(y)$ be the value that Benders cut a attains for master problem solution y.

Definition 4. *A Benders cut $\theta \geq \bar{\theta}^a(y)$* ***dominates*** *another Benders cut $\theta \geq \bar{\theta}^b(y)$ if $\bar{\theta}^a(y) \geq \bar{\theta}^b(y)$ for all feasible $y \in Y$ and is a strict inequality for at least one feasible y.*

An example of this can be seen in Figure 3.4. This definition leads to the following lemma:

Lemma 1. *If $\theta \geq \bar{\theta}^a(y)$ is dominated by $\theta \geq \bar{\theta}^b(y)$, then for all feasible solutions y^i where $\bar{\theta}^a(y^i) = \bar{\theta}^*(y^i)$, $\bar{\theta}^b(y^i) = \bar{\theta}^*(y^i)$,*

where $\bar{\theta}^*(y^i)$ is the objective value of the sub-problem for master problem solution y^i. This is easy to see, since $\bar{\theta}^b(y^i) \leq \bar{\theta}^*(y^i)$ by definition of being a valid Benders cut, and $\bar{\theta}^b(y^i) \geq \bar{\theta}^a(y^i) = \bar{\theta}^*(y^i)$ by definition of being a dominating cut. An example of this can be seen in figure 3.4, where the cut centred about B covers a superset of the solutions covered by the cut centred about A.

Definition 5. *A Benders cut $\theta \geq \bar{\theta}^a(y)$ is Pareto-optimal if it is not dominated by any other Benders cuts.*

A Pareto-optimal cut is then a Benders cut that is not strictly worse than any other Benders cut. This is, in a sense, the strongest type of cut one can construct. There are a number of methods for finding Pareto-optimal Benders cuts in the literature. The first was introduced by Magnanti and Wong [46], in the same study that introduced the concept of Pareto-optimality of Benders cuts. This method was later improved by Papadakos [49] and has since been used in a number of different studies [52, 65–67].

3.4.1 Pareto-optimality of the UFL Benders cuts

For the UFL problem, the Benders cut centred about j (except when j is the closest-possible facility) is Pareto-optimal. This was shown for the aggregated sub-problem case by Magnanti and Wong [46] in

y_A	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
y_B	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
y_C	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
y_D	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Centred about A	■								✓	✓	✓	✓	✓	✓	✓	✓
Centred about B	■				✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Figure 3.4: Comparison of master problem solutions where different cuts give the correct objective value. Note that the cut centred about A is dominated by the cut centred about B

the context of their Pareto-optimal cut generation scheme, but it can also be shown for the disaggregated sub-problems, and in a more intuitive manner.

To prove a cut is Pareto-optimal, begin by assuming that it is not, *i.e.* there exists another Benders cut that dominates it. Then, by Lemma 1, this dominating cut must give the correct objective value at all the same points as the original Benders cut. By finding enough of these points, the dominating cut reduces to the original cut. Since a cut cannot dominate itself, there does not exist any cut that dominates the original Benders cut, and thus it is Pareto-optimal. We now provide such a proof for the Benders cuts of the UFL problem.

First, define the following subsets of facilities for convenience:

$$\begin{aligned} O &= \{j \in F | y_j^* = 1\} && \text{The set of open facilities} \\ C &= \{j \in F | y_j^* = 0\} && \text{The set of closed facilities} \\ L_i^+ &= \{j \in F | c_{ij} \geq c_{ij(i)}\} && \text{The set of facilities at least as far away as } j(i) \\ L_i^- &= \{j \in F | c_{ij} < c_{ij(i)}\} && \text{The set of facilities closer than } j(i) \end{aligned}$$

Note that $O \cup C = F$ and $L_i^+ \cup L_i^- = F$. Denote the Benders cut centred about j for location i as $\theta_i \geq \bar{\lambda}_i - \sum_{j \in F} \bar{\pi}_{ij} y_j$. Assume there exists a cut that dominates this cut, say $\theta_i \geq \hat{\lambda}_i - \sum_{j \in F} \hat{\pi}_{ij} y_j$. For the current master problem solution, the Benders cut gives the correct objective value of $c_{ij(i)}$, so the dominating cut must also give the correct objective value, that is:

$$c_{ij(i)} = \hat{\lambda}_i - \sum_{j \in O} \hat{\pi}_{ij}. \tag{3.41}$$

Now, if a facility $j^* \in L_i^+ \cap C$ is opened, the objective value does not change as the closest open facility has not changed. The Benders cut gives the correct objective value at this point (since $\bar{\pi}_{ij} = 0$ whenever $c_{ij} > c_{ij(i)}$), so the dominating cut must also give the same objective value, *i.e.*

$$c_{ij(i)} = \hat{\lambda}_i - \sum_{j \in O} \hat{\pi}_{ij} - \hat{\pi}_{ij^*}. \tag{3.42}$$

If we subtract equation (3.42) from equation (3.41), we find that $\hat{\pi}_{ij^*} = 0$, the same value as the Benders cut. Similarly, if any facility $j^* \in L_i^+ \cap O \setminus \{j(i)\}$ is closed, the objective value does not change and the Benders cut gives the correct objective value in this case. So the dominating cut must also give the same value, that is

$$c_{ij(i)} = \hat{\lambda}_i - \sum_{j \in O} \hat{\pi}_{ij} + \hat{\pi}_{ij^*}. \tag{3.43}$$

This time, subtracting equation (3.41) from equation (3.43) gives us $\hat{\pi}_{ij^*} = 0$, again the same value as the Benders cut. So we have that $\hat{\pi}_{ij} = \bar{\pi}_{ij}\ \forall j \in L_i^+ \setminus \{j(i)\}$. Note that $L_i^- \cap O = \emptyset$, since otherwise there would be an open facility closer to i than $j(i)$, which is impossible by definition.

For any facility $j^* \in L_i^- \cap C$, if that facility is opened, it will be the new closest facility, and so the objective value will change to c_{ij^*}. The Benders cut gives the correct objective value in this case, and

so must the dominating cut, that is:

$$c_{ij^*} = \hat{\lambda}_i - \sum_{j \in O} \hat{\pi}_{ij} - \hat{\pi}_{ij^*}. \tag{3.44}$$

Now, subtracting equation (3.44) from equation (3.43), we find that $c_{ij(i)} - c_{ij^*} = \hat{\pi}_{ij^*}$, and since $c_{ij(i)} > c_{ij^*}$, $\hat{\pi}_{ij^*} = \bar{\pi}_{ij^*}$. So now $\hat{\pi}_{ij} = \bar{\pi}_{ij}\ \forall j \in F \setminus \{j(i)\}$. If we now consider the scenario where a facility $j^* \in L_i^- \cap C$ is opened and the facility $j(i)$ is closed, the objective value will be c_{ij^*}, and the Benders cut will give the correct objective value for this point, so the dominating cut must also give the correct objective value, or:

$$c_{ij^*} = \hat{\lambda}_i - \sum_{j \in O} \hat{\pi}_{ij} - \hat{\pi}_{ij^*} + \hat{\pi}_{ij(i)}. \tag{3.45}$$

Subtracting equation (3.45) from equation (3.44) shows that $\hat{\pi}_{ij(i)} = 0$, the same as the Benders cut. Finally, considering the original solution, the Benders cut and dominating cuts are equal, so we have:

$$\bar{\lambda}_i - \sum_{j \in O} \bar{\pi}_{ij} = \hat{\lambda}_i - \sum_{j \in O} \hat{\pi}_{ij} \tag{3.46}$$

$$= \hat{\lambda}_i - \sum_{j \in O} \bar{\pi}_{ij} \tag{3.47}$$

$$\therefore \bar{\lambda}_i = \hat{\lambda}_i, \tag{3.48}$$

and thus the dominating cut is exactly the Benders cut. Since a cut cannot dominate itself, there are no cuts that dominate the Benders cut, and thus it is Pareto-optimal. This proof considers the cut centred about $j(i)$, but since the master problem solution, and hence $j(i)$, is arbitrary, this proof applies to all such Benders cuts on the condition that the closest possible facility is closed. If this was not the case, one could not select a $j^* \in L_i^- \cap C$ to finish the proof, and the cut in this case happens to be the trivial cut.

The same method for proving Pareto-optimality appears in the paper by Pearce and Forbes [2], shown in Section 3.7. It is also useful for finding algorithms that generate Pareto-optimal Benders cuts analytically. One benefit to generating Benders cuts this way is that it does not require the solution of multiple (or in the case of the UFL, any) LPs.

The usefulness of generating Pareto-optimal cuts compared to simply using the dual variables returned by the LP solver varies depending upon the problem. In some cases, they may make a small difference; in others they are almost necessary for finding solutions to large instances. The only time using them is not beneficial is when they give little improvement over the default cuts and it takes a large amount of time to compute them.

Benders decomposition is useful for a range of problems and the remainder of this chapter will consider a few examples of such problems. The first example is the Maximum Total Flow with Flexible Arc Outages (MaxTF-FAO) problem, which schedules the maintenance of arcs in a network to minimise the impact on the flow through that network over a number of time periods. Benders

decomposition is useful for this problem because the time periods are independent, and so the sub-problem can be disaggregated by time. There are a few other problems with a similar structure that would also benefit from Benders decomposition.

The second example is the Dynamic Uncapacitated Facility Location and Network Design Problem (DUFLNDP), which is an extension of the UFL problem. In this problem, customers must be served by facilities, but their connections are no longer direct. Instead, a network is constructed and the demands must be routed to facilities. This problem also controls the network over multiple time periods, where the flows in each time period are independent of each other. This means the sub-problems can be disaggregated by customer and time periods.

Lastly, we generalise the work on the DUFLNDP and describe a class of problems that are likely to benefit from Benders decomposition.

3.5 Paper: Disaggregated Benders Decomposition for Solving a Network Maintenance Scheduling Problem

Abstract

We consider a problem concerning a network and a set of maintenance requests to be undertaken. The aim is to schedule the maintenance in such a way as to minimise the impact on the total throughput of the network. We embed disaggregated Benders decomposition in a branch-and-cut framework to solve the problem to optimality, as well as explore the strengths and weaknesses of the technique. We prove that our Benders cuts are Pareto-optimal. Solutions to the linear programming relaxation also provide further valid inequalities to reduce total solving time. We implement these techniques on simulated data presented in previous papers, and compare our solution technique to previous methods and a direct mixed-integer programming formulation. We prove optimality in many problem instances that have not previously been proven.

3.5.1 Introduction

Network design and scheduling problems are an important area of study, particularly as they have widespread practical applications. Examples of these problems include minimising the cost of maintaining a network [68], restoring a damaged network [69] or extending an existing network [65]. In practice, networks are often large, and optimising their design can be difficult and time-consuming. Industry is always interested in any improvements to operations that result in reduced costs.

Benders decomposition is a powerful technique for breaking a difficult mixed-integer program (MIP) into smaller, easier-to-solve problems [24]. It has been successfully applied to a number of problems, particularly network design and facility location problems. This technique is especially powerful when the sub-problems can be disaggregated to allow us to add stronger disaggregated Benders cuts [52, 60]. Magnanti and Wong [48] show that the use of Pareto-optimal cuts with Benders decomposition can improve convergence time by up to 50 times over other Benders cuts. In 2008 Camargo et al. apply disaggregated Benders decomposition to the design of hub-and-spoke networks [70]. Tang and Jiang use disaggregated Benders decomposition to solve a capacitated facility location problem with existing facilities that can be removed or extended [65], and Lusby, Muller and Petersen use disaggregated Benders decomposition for scheduling the maintenance of power plants in France [71].

Embedding Benders decomposition into a branch-and-cut framework can lead to significant improvements [60, 72]. Thanks to recent advancements, many state-of-the-art solvers now allow users to add their own cuts in the branch-and-bound framework. For Gurobi and CPLEX, this feature is known as lazy constraints. By supplying a “user callback”, one can add additional constraints at each node of the branch-and-bound tree. This allows us to claim the advantages inherent in using both branch-and-cut and the modern solvers multi-processing and parallel-computing abilities.

Network design and scheduling problems are perfect candidates for Benders decomposition, because they can be separated into sub- and master problems. The sub-problems for these problems involve computing the maximum flow through the network, and the master problem handles the higher-level design of the network in order to have the largest flow subject to design constraints. Disaggregated Benders decomposition can be applied if the flow at any time is independent of the flow at other times. This often makes the solution of large network design problems easier [48, 72].

We consider a problem of performing maintenance on a network to minimise the impact on total flow through the network over time. Boland *et al.* (2014) solve this problem to near-optimality using heuristics. In this paper we show this problem can be solved to optimality in many cases using disaggregated Benders decomposition embedded in a branch-and-cut framework. The problem formulation we use is similar to that of Boland *et al.* (2014), with different notation and a necessary change to one constraint. Our main contributions are the implementation of Benders decomposition to this network maintenance scheduling problem, a proof of Pareto-optimality of the Benders cuts used, and a discussion of the benefits of using Benders decomposition.

The networks considered by Boland *et al.* (2014) are rail networks for moving coal from the Hunter Valley to shipping terminals in Newcastle. The arcs of the network represent railway lines and the nodes are junctions, where it is possible to choose which direction to take. In the real network, the shipping terminal is comprised of different machines and railways that are also modelled as being part of the same network, and similarly have maintenance jobs assigned to them. Note that the presented approach can be applied to any situation where a network is to be modified over a set of time periods, and the flow in each time period is independent of the flow at all other times.

The remainder of this section will be a short description of the max-flow min-cut theorem, which is useful in solving this problem. In Section 2, we define the problem and present the formulation. Section 3 is where we describe the use of Benders decomposition and lazy constraints to separate and solve the problem. We also examine a scenario that makes this problem more difficult to solve, and prove the Pareto-optimality of our Benders cuts. In Section 4, we present our results and compare them to those found by Boland *et al.* (2014), as well as to a direct MIP implementation in our version of Gurobi. Section 5 contains concluding remarks.

Max-flow min-cut theorem

The max-flow min-cut theorem was discovered and proven by Elias, Feinstein and Shannon [73], and independently by Ford and Fulkerson [74]. We will use the nomenclature from Elias, Feinstein and Shannon [73] in talking about the max-flow min-cut theorem.

A **cut-set** of a network is defined as a set of arcs which, when removed, prevents all flow from the source to the sink. This does not necessarily have to make the graph disconnected, because arcs are allowed to flow backwards, but there will be no complete path flowing forwards from the source to the sink. A **simple cut-set** is a cut-set that would no longer be a cut-set if any arc was omitted from it. The **value** of a cut-set is the sum of the capacities of all arcs in the set. A **minimal cut-set** is a cut-set with the smallest value of all possible cut-sets of the network.

With these definitions, we can state the max-flow min-cut theorem, which we have paraphrased from Elias, Feinstein and Shannon [73]:

Theorem 1. *The maximum possible flow from the source to the sink through a network is equal to the minimum value among all simple cut-sets.*

For proof of this theorem we refer the reader to the original paper by Elias et al. [73]. We use this theorem to place bounds on the flow of the network based on the availability of arcs that are in a simple cut-set, especially the minimum cut-set.

3.5.2 Problem definition

We start with a network $G = (N, A)$ where N is the set of nodes and A is the set of directed arcs. Without loss of generality, we assume the network has only one source and one sink, and that there is a directed arc from the sink to the source, denoted arc $\tilde{a}$. If this is not the case, an augmented network can be constructed by adding two new nodes: the global source and global sink. Directed arcs are added from the global source to all original sources, and from all original sinks to the global sink. Finally, the global sink is connected to the global source by arc $\tilde{a}$, which measures the total flow through the network. All arcs added to the original network must have a sufficiently high capacity so as to not affect the maximum flow through the network.

We have a set of maintenance requests that must be performed. Each request $r \in R$ has a release time Re_r, a deadline De_r and a duration Dur_r. We assume a work crew will be available for each job regardless of where in the time window the job occurs. If maintenance is being performed on an arc, the flow along that arc must be 0. If the arc is open, the flow must not exceed the arc capacity, u_a.

The problem *maximum total flow with flexible arc outages (MaxTF-FAO)* is now as follows:

Sets

N	Set of network nodes
A	Set of network arcs, $A \subseteq N \times N$, and $A \ni a = (a_0, a_1)$
T	Set of time periods
R	Set of maintenance requests
R_a	Set of maintenance requests to be performed on arc $a \in A$

Data

u_a	Capacity of arc $a \in A$
$\delta^-(i)$	Set of arcs entering node $i \in N$
$\delta^+(i)$	Set of arcs leaving node $i \in N$
Dur_r	Duration of maintenance job $r \in R$
Re_r	Earliest time job $r \in R$ can be started
De_r	Deadline for job $r \in R$

Variables

x_{at} Flow over arc $a \in A$ at time $t \in T$

y_{at} 1 if arc $a \in A$ is operational at time $t \in T$ and 0 if it is undergoing maintenance

z_{rt} 1 if maintenance request $r \in R$ starts at time $t \in T$ and 0 otherwise

$$\text{Maximise} \sum_{t \in T} x_{\tilde{a}t} \tag{MIP-OBJ}$$

Subject to:

$$\sum_{a \in \delta^-(i)} x_{at} - \sum_{a \in \delta^+(i)} x_{at} = 0 \qquad \forall i \in N, \forall t \in T \tag{MIP1}$$

$$x_{at} \leq u_a y_{at} \qquad \forall a \in A, \forall t \in T \tag{MIP2}$$

$$\sum_{t=Re_r}^{\mathrm{De}_r - \mathrm{Dur}_r + 1} z_{rt} = 1 \qquad \forall r \in R \tag{MIP3}$$

$$y_{at} + \sum_{r \in R_a} \sum_{t' = \max\{\mathrm{Re}_r, t - \mathrm{Dur}_r + 1\}}^{\min\{t, \mathrm{De}_r\}} z_{rt'} = 1 \qquad \forall a \in A, \forall t \in T \tag{MIP4}$$

$$x_{at} \geq 0, \quad y_{at} \in \{0,1\}, \quad z_{rt} \in \{0,1\} \qquad \forall a \in A, \forall t \in T, \forall r \in R \tag{MIP5}$$

The objective (MIP-OBJ) is the sum of flow across $\tilde{a}$, which measures the total flow of the network, over the considered time periods. Constraints (MIP1-MIP2) ensure that flow into and out of a node are the same, and flow along any arc does not exceed the capacity. Constraints (MIP3) ensure every maintenance job is performed exactly once, and constraints (MIP4) say if a maintenance job is currently operating on arc a at time t, then $y_{at} = 0$ (the arc is closed), otherwise $y_{at} = 1$ (the arc is open). Finally, (MIP5) ensures all x variables are non-negative and all other variables are binary.

As in Boland *et al.* (2014), we make the assumption that no two jobs in R_a for any a can overlap. This allows us to describe constraint (MIP4) as an equality, rather than an inequality. The importance of this is that if no maintenance is being performed upon an arc at a certain time, constraints (MIP4) will force the relevant arc to be open at that time. This does not change the optimal solution of the problem, but it may help when applying Benders decomposition. Boland *et al.* (2014) prove that this problem is NP-Hard.

3.5.3 Disaggregated Benders decomposition and lazy constraints

In this problem, the continuous variables x_{at} only occur with integer variables in one constraint: the capacity constraint (MIP2). This allows us to apply Benders decomposition by separating out the continuous variables into a sub-problem, and approximating the solution to the sub-problem with a new variable θ. The result of this is a smaller, more relaxed problem that can be explored faster.

This decomposition has a natural interpretation. The master problem finds the optimal maintenance schedule given the estimates of the network throughput, and the sub-problems calculate the actual throughput given the network configuration. Because the flow through the network at each time $t \in T$ is independent of the flow at other times, we further break up the problem by disaggregating the sub-problems in t, so we solve one sub-problem for each time period. This is an important step,

because without disaggregation, Benders decomposition performs poorly, as is discussed in the results section.

Given a feasible solution for y_{at} found by the master problem, denoted by y^*_{at}, we solve the sub-problems for each time period. Each sub-problem is of the form:

$$\max \quad x_{\tilde{a}t'} \tag{SP-OBJ}$$

Subject to:

$$\sum_{a\in\delta^-(i)} x_{at'} - \sum_{a\in\delta^+(i)} x_{at'} = 0 \qquad \forall i \in N \tag{SP1}$$

$$x_{at'} \leq u_a y^*_{at'} \qquad \forall a \in A \tag{SP2}$$

$$x_{at'} \geq 0 \qquad \forall a \in A \tag{SP3}$$

for every $t' \in T$. These are small linear programs (LPs) that are easily solved by any good optimisation software package. Using a commercial solver to solve these as LPs takes a negligible amount of time and allows us to easily extract the information we require. Solvers that are specialised in solving network flow problems exist, but their use may only provide marginal benefits. To apply Benders decomposition, we must find the dual to this sub-problem, also known as the primal problem. Let π_i be the unconstrained dual variable associated with constraints (SP1) and $\gamma_a \geq 0$ the dual variable associated with constraints (SP2). The dual problem is then:

$$\min \quad \sum_{a\in A} u_a y^*_{at'} \gamma_a \tag{DP-OBJ}$$

Subject to:

$$(\pi_{a_0} - \pi_{a_1} + \gamma_a) \geq 0 \qquad \forall a \in A \setminus \{\tilde{a}\} \tag{DP1}$$

$$\pi_{\tilde{a}_0} - \pi_{\tilde{a}_1} + \gamma_{\tilde{a}} - 1 \geq 0 \tag{DP2}$$

$$\gamma_a \geq 0 \qquad \forall a \in A \tag{DP3}$$

This problem is the minimum cut problem [75]. As $\tilde{a}$ has a large objective coefficient (larger than any other cut-set of the network), $\gamma_{\tilde{a}} = 0$ in any optimal solution. To satisfy (DP2) with $\gamma_{\tilde{a}} = 0$, we must set $\pi_{\tilde{a}_0} = 1$. The nodes are then partitioned into two sets, P_1 and P_2, where $\pi_i = 0, \forall i \in P_1$ and $\pi_j = 1, \forall j \in P_2$. Note the global source is in P_1 and the global sink is in P_2. For all arcs that go from P_1 to P_2, $\gamma_a = 1$, and for all other arcs, $\gamma_a = 0$. The partitions are chosen so the combined capacity of the connecting arcs is minimised. In practice, these values are returned by the solver when the primal problem is solved.

We approximate the solutions to the sub-problems with new variables in the master problem, $\theta_{t'}$. For each $t' \in T$, any feasible solution of the dual problem provides a valid upper bound on the objective of the dual problem, and hence the objective of the primal problem. Thus, for an optimal dual solution γ^*, our Benders optimality cut is given by:

$$\theta_{t'} \leq \sum_{a\in A} u_a y_{at'} \gamma^{*k}_{at'}, \qquad \forall k \in \{1...K\}, \forall t' \in T \tag{BOC}$$

where K is the number of times we have added Benders optimality cuts. We now present the Benders master problem for the disaggregated Benders decomposition formulation for the MaxTF-FAO problem:

$$\text{Maximise} \sum_{t \in T} \theta_t \tag{MP-OBJ}$$

Subject to:

$$\theta_t \leq \sum_{a \in A} u_a y_{at} \gamma_{at}^{*k} \qquad \forall k \in \{1...K\}, \forall t \in T \tag{BOC}$$

$$\sum_{t=\text{Re}_r}^{\text{De}_r - \text{Dur}_r + 1} z_{rt} = 1 \qquad \forall r \in R \tag{MP1}$$

$$y_{at} + \sum_{r \in R_a} \sum_{t'=\max\{\text{Re}_r, t - \text{Dur}_r + 1\}}^{\min\{t, \text{De}_r\}} z_{rt'} = 1 \qquad \forall a \in A, \forall t \in T \tag{MP2}$$

$$\theta_t \geq 0, \quad y_{at} \in \{0,1\}, \quad z_{rt} \in \{0,1\} \qquad \forall a \in A, \forall t \in T, \forall r \in R \tag{MP3}$$

The master problem includes all constraints from the original MIP that do not contain x_{at} variables. The contribution of the x_{at} variables to the objective function is approximated by the new variables θ_t. While exploring the branch-and-bound tree, at each integer solution, the values of θ_t are compared to the actual maximum flow of the given network configuration. For each $t \in T$, if the approximation given by θ_t is incorrect, we add a new Benders cut to the problem, which we add as a lazy constraint. Thus, we may add up to $|T|$ Benders cuts at each integer solution, however in practice we add many fewer. This is how Benders decomposition is embedded in a branch-and-cut framework.

An advantage of the disaggregation of the flow problems is they now only depend on the configuration of arcs $y_{at'}$ for each $t' \in T$. As such, we store solutions to these flow problems, where the key is the vector $(y_{at'}, \forall a \in A)$. While solving linear flow problems is fast, recalling the solution from memory is faster and only marginally increases memory usage of the solver. This is especially important for this problem, because there will often be many duplicates of specific flow problem instances. The number of times we solve and recall flow problems will be shown in the results section for some cases, as well as a comparison of the impact on solving time. There are several other improvements we make to increase the effectiveness of the solver.

Initial cuts

We begin by considering the case where all arcs are turned on, *i.e.* $(y_{at} = 1, \forall a \in A, \forall t \in T)$. This gives an upper bound on the flow through the network in any case, because turning an arc off cannot possibly increase the total flow. The set of arcs that have a non-zero dual variable associated with their capacity constraints ($\{a | \gamma_a^* > 0\}$) is a "minimum cut-set" from the max-flow min-cut theorem [73]. In other words, they are bottlenecks of the network, since turning any of them off will directly affect the total flow through the network.

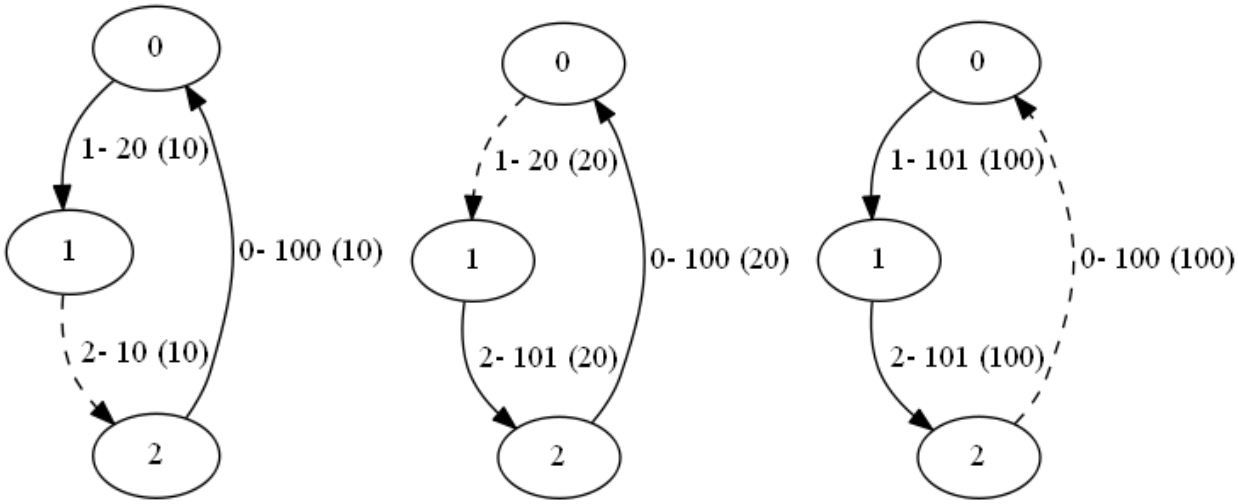


Figure 3.5: Example of placing cuts on bottlenecks. Numbers next to arcs are of the form: ArcID - capacity (flow). Dashed lines are arcs whose associated dual variables are non-zero

This is not the only cut-set of the network. We can find other cut-sets by increasing the capacity of all arcs in the original cut-set to an amount higher than arc $\tilde{a}$. This may increase the maximum flow of the network. However, if the total flow is not equal to the capacity of $\tilde{a}$, we can identify another cut-set using the new dual variables. By repeating this process until the total flow is equal to the capacity of $\tilde{a}$, we can add multiple initial constraints to assist the solver.

Consider the trivial case in Figure 3.5. To start with, the arc with capacity 10 is the bottleneck. It is the only arc with a non-zero dual variable, so the initial cut will be $\theta_t \leq 10y_{2,t}$ for every t. Next, we change the capacity of this arc to be larger than that of the total flow arc (*i.e.* 101). When we solve the flow problem again, we see arc 1 will be the bottleneck. Since it is the only arc with a non-zero dual variable, we add another cut $\theta_t \leq 20y_{1,t}$ for every t. We increase the capacity of this arc as before, calculate the solution to the new flow problem, and find the total flow arc is now the bottleneck. When this occurs, we are finished adding initial cuts.

Both these initial cuts are valid, since turning off either of these arcs will restrict all flow through the network and $\theta_t = 0$. For larger problems, the bottlenecks will consist of multiple arcs, and the cuts provide information about how closing arcs in those bottlenecks affects the flow.

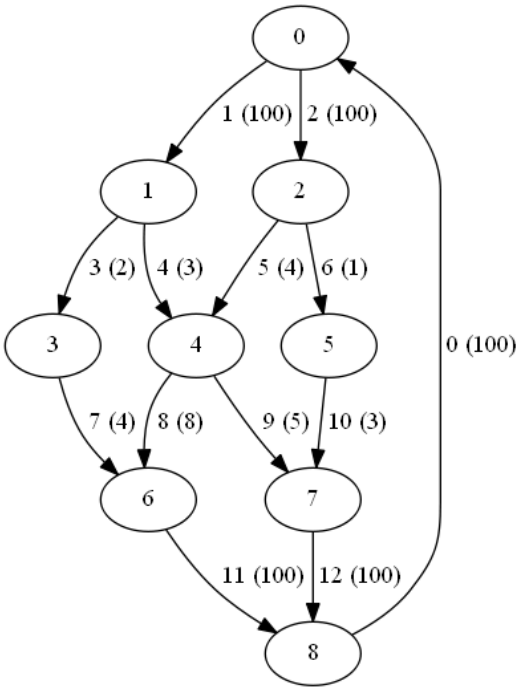


Figure 3.6: A layered network. The edges are labelled by ArcID (capacity). The two layers have capacities of 10 and 20

Consider now the less-trivial example in Figure 3.6. This is a layered network, where the sum of the capacities of arcs 3-6 is 10 and of arcs 7-10 is 20. In this case the initial cut-set will be arcs 3-6 since they form

the minimum cut-set. The cuts we add will be:

$$\theta_t \leq 2y_{3,t} + 3y_{4,t} + 4y_{5,t} + y_{6,t}. \qquad \forall t \in T$$

Because this is a minimum cut-set by the max-flow min-cut theorem, cutting any of these arcs will reduce the total flow through the network. When we increase these capacities and solve the maximum flow problem again, we find the second cut-set of arcs 7-10. We then add the cuts:

$$\theta_t \leq 4y_{7,t} + 8y_{8,t} + 5y_{9,t} + 3y_{10,t}, \qquad \forall t \in T$$

which are also valid. With these initial cuts, we find a feasible solution to the master problem. We then solve the problem as before, except now we start with a tighter LP bound.

User heuristics

Another potential improvement is user-suggested heuristics. While the MIP solver is exploring nodes, adding Benders cuts will cut off the current solution because the values of θ_t are too high. When this happens, we construct a feasible solution with the same values of y_{at} and z_{rt}, but set θ_t equal to the solutions to the sub-problems that were solved, and suggest this to the solver as a heuristic solution. In our experiments, it has in some cases led to significant jumps in reducing the optimality gap, particularly on larger problems.

Warm start

Finally, we implement what is known as a "warm start", where we relax the integrality constraints and run the main algorithm to add Benders optimality cuts (BOC). First suggested by McDaniel and Devine (1977), a warm start can often improve the initial bound of the Benders master problem [52, 76]. Due to the relaxation of the Benders master problem, the optimal objective value of the linear relaxation of the master problem will be higher than that of the original problem. By adding Benders optimality cuts based on the solution to the linear relaxation, we tighten the master problem, reducing the objective value of the LP-relaxation.

We continue adding Benders cuts and solving the linear relaxation until the objective value found by the relaxed problem stops decreasing, or no more Benders cuts are added. Once this occurs, we restore the integrality constraints and solve the problem once more. The results of this are a tighter LP bound and possible improvements in the solving time of the MIP. However, the time it takes to solve the relaxed problem multiple times must be taken into account.

Strength of the LP-Relaxation

Any job $r \in R$ can start during the time window [Re_r,De_r-Dur_r+1]. If the size of this window is larger than the duration of the job, the LP-relaxation of the problem provides a weaker bound. This is because it is possible to fractionally assign values to z_{rt} and thus have arcs fractionally open for more than the

duration of the maintenance. The result of this is a weaker LP bound on the objective value and thus a longer solving time, which applies to all MIP implementations.

Consider the example of a job r where $\mathrm{Re}_r = 0$, $\mathrm{Dur}_r = 10$ and $\mathrm{De}_r = 29$. This job could start at times $t' \in [0, 20]$. We are only considering the scheduling constraints (MIP3-MIP4) here. When the variables y_{at} and z_{rt} are allowed to be continuous, it is possible for z_{rt} to take values of $\frac{1}{3}$ at times 0, 10 and 20, and 0 elsewhere. Because in constraint (MIP4) we sum the z_{rt} over values of t' within Dur_r time periods previous to t, at every $t \in [\mathrm{Re}_r, \mathrm{De}_r]$,

$$\sum_{r \in R_a} \sum_{t'=\max\{\mathrm{Re}_r, t-\mathrm{Dur}_r+1\}}^{\min\{t, \mathrm{De}_r\}} z_{rt'} = \frac{1}{3}.$$

This implies that

$$y_{at} + \frac{1}{3} = 1, \text{ so} \qquad \forall a \in A, \forall t \in T$$
$$y_{at} = \frac{2}{3}. \qquad \forall a \in A, \forall t \in T$$

This means the arc the job is being performed on will be fractionally closed for the entire time between Re_r and De_r, whereas in an integer solution it must be fully closed for Dur_r. If closing this arc results in a change in the minimum cut-set of the network, but fractionally closing the arc does not, the relaxed problem will not properly reflect the impact on the objective value from closing this arc. Instances with this property are thus more difficult to solve.

Algorithm details

We include the pseudo-code for our algorithms to give a brief idea of how our implementations are set up. Algorithm 1 is the main procedure, which includes potential calls to the sub-routines PRE-CUTS and WARM START, depending on whether or not they are being used. When we talk about building models, we are referring to creating a Model object in Gurobi [44] and attaching all relevant variables and constraints.

Because we disaggregate the sub-problems in time, and they are all identical, we only need to build one model and use it to solve the flow sub-problems for all time periods. The results of these sub-problems are stored in a hash table, Y. For any time t', ($y_{at'}$, $\forall a \in A$) will be the configuration of arcs of the network, *i.e.* which arcs are open and which are closed. The configuration ($y_{at'}$, $\forall a \in A$) is the key to a hash table entry that holds a tuple. The first value is the total flow through the network and the second is a vector (γ_a, $\forall a \in A$) of the dual variables associated with the capacity constraints of each arc.

When we require the solution to a sub-problem for a certain configuration of arcs, we first check to see if we have already solved it. If $y_{at'}$ is a valid key to the hash table, we simply recall the tuple stored in that entry. If that particular configuration has not been solved, we pass the values $y_{at'}$ to the

Algorithm 1 Main Procedure

Initialise information about Network and Jobs: N, A, u_a, R, R_a, T
Create empty hash table Y to hold solutions to Sub-Problems
Build Sub-Model and Master Model
Initialise Sub-Model variables x_a and Master Model variables y_{at}, z_{rt}
Set $y_{at} = 1, \forall a \in A, \forall t \in T$
OPTIMISE Sub-Model for one time value
Add constraints: $\theta_t \leq x^*_{\tilde{a}}, \forall t \in T$

if Pre-cuts **then**
  Run procedure PRE-CUTS
if LP-Relax **then**
  Run procedure WARM START
Run procedure OPTIMISE MASTER MODEL

Algorithm 2 OPTIMISE MASTER MODEL

OPTIMISE Master Model with callback MMCB
MMCB:
if Found new incumbent solution **then**
  for all $t' \in T$ **do**
    Retrieve values of $y^*_{at'}$ and pass to Sub-Model
    if not $y^*_{at'}$ in Y **then**
      OPTIMISE Sub-Model
      $\mathrm{Y}[y^*_{at'}] \leftarrow (x_{\tilde{a}}, (\gamma_a, \forall a \in A))$
    else
      $(x_{\tilde{a}}, (\gamma_a, \forall a \in A)) \leftarrow \mathrm{Y}[y^*_{at'}]$
    $\bar{\theta}_{t'} \leftarrow x_{\tilde{a}}$
    if $\theta_{t'} > x_{\tilde{a}}$ **then**
      Add lazy constraint $\theta_{t'} \leq \sum_{a \in A} u_a y_{at'} \gamma_a$
  if USE HEURISTIC **then**
    if $\sum_{t \in T} \theta_t > \sum_{t \in T} \bar{\theta}_t$ **then**
      Suggest $\theta_t = \bar{\theta}_t \ \forall t \in T$ as heuristic solution

Algorithm 3 PRE-CUTS

Retrieve dual variables from Sub-Model $(\gamma_a, \forall a \in A)$
while not $\gamma_{\tilde{a}} > 0$ **do**
  Add constraints $\theta_t \leq \sum_{a \in A} u_a y_{at} \gamma_a, \forall t \in T$
  for all $a \in A$ **do**
    if $\gamma_a > 0$ **then**
      $u_a = u_{\tilde{a}} + 1$
  OPTIMISE Sub Model
  Retrieve dual variables from Sub-Model $(\gamma_a, \forall a \in A)$
Reset values of $(u_a, \forall a \in A)$

Algorithm 4 WARM START

```
Relax integrality constraints for y_at and z_rt
while True do
    OPTIMISE Master model without callback
    for all t' ∈ T do
        Retrieve values of (y_at', ∀a ∈ A) and pass to Sub-Model
        OPTIMISE Sub-Model
        if θ'_t > x_ã then
            Add constraint θ'_t ≤ Σ_{a∈A} u_a y_at' γ_a
    if Objective has not improved, time limit expired or max iterations reached then
        Exit While
Enforce integrality constraints for y_at and z_rt
```

sub-problem model and solve the max-flow problem. We then store the results of this in the hash table under the key $y_{at'}$.

Algorithm 2 describes the user callback for computing and adding the Benders optimality cuts inside the branch-and-bound tree. The last "if" statement of Algorithm 2 is where we suggest a user heuristic solution to the solver. Algorithm 3 implements the initial cut method described in Section 3.5.3, and Algorithm 4 is the warm start described in Section 3.5.3.

Proof of Pareto-optimality of Benders cuts

It has been shown that the use of Pareto-optimal cuts can greatly improve the convergence rate of Benders decomposition [46, 48, 49, 65]. Pareto-optimal cuts are especially powerful when there is degeneracy in the sub-problems of the Benders decomposition, which is the case in network design problems [48]. The definitions of dominating and Pareto-optimal cuts we use come from Magnanti and Wong [46], but are modified to match our problem.

Since these cuts are disaggregated in time, we will omit all t parameters for simplicity. This means we will consider θ instead of θ_t, and likewise x_a, y_a, γ_a. Since our Benders cuts depend only upon our y variables, we write them in a general form $\theta \leq \bar{\theta}^k(y)$, where $\bar{\theta}^k(y) = \sum_{a \in A} u_a y_a \gamma_a^k$ and $k \in \{0, ..., K\}$ represents the number of different Benders cuts.

Definition 6. *A Benders cut $\theta \leq \bar{\theta}^k(y)$ dominates another Benders cut $\theta \leq \bar{\theta}^l(y)$ if $\bar{\theta}^k(y) \leq \bar{\theta}^l(y)$ for all feasible y and is a strict inequality for at least one feasible y.*

The contrapositive of this is that if there exists a feasible solution y^* such that $\bar{\theta}^k(y^*) > \bar{\theta}^l(y^*)$, then $\theta \leq \bar{\theta}^k(y)$ does not dominate $\theta \leq \bar{\theta}^l(y)$.

Definition 7. *A Benders cut $\theta \leq \bar{\theta}^k(y)$ is considered Pareto-optimal if it is not dominated by any other cuts.*

That is to say, if for any other Benders cut $\theta \leq \bar{\theta}^l(y)$ one can find a feasible solution y^* such that $\bar{\theta}^k(y^*) < \bar{\theta}^l(y^*)$, then $\theta \leq \bar{\theta}^k(y)$ is Pareto-optimal.

Observation 1. *For a given network configuration y_a and its primal and dual flow solutions x_a and γ_a^k, the set $A^k = \{a \in A | \gamma_a^k = 1\} \subset A$ must constitute a simple cut-set of the original network.*

This comes from the max-flow min-cut theorem. This is necessary for the Benders cut generated by A^k to be Pareto-optimal. If A^k is not a cut-set, the Benders cut generated by A^k is not a valid cut. We show this by closing all arcs in A^k and opening all others. Since A^k is not a cut-set, it is still possible for flow between the source and the sink to occur, so $\theta > 0$. However, we also have that $\theta \leq \sum_{a \in A^k} u_a y_a^* = 0$. If the set A^k is a cut-set but not a simple cut-set, there exists an arc $a' \in A^k$ such that $A^k \setminus \{a'\}$ is still a cut-set. Let $A^l = A^k \setminus \{a'\}$, which means $A^l \cup \{a'\} = A^k$. We now compare the Benders cuts generated by these two sets:

$$\theta \leq \sum_{a \in A^l} u_a y_a < \sum_{a \in A^l} u_a y_a + u_{a'} y_{a'} = \sum_{a \in A^l \cup \{a'\}} u_a y_a = \sum_{a \in A^k} u_a y_a.$$

So the Benders cut $\theta \leq \bar{\theta}^l(y)$ dominates $\theta \leq \bar{\theta}^k(y)$, and thus the Benders cut generated by A^k cannot be Pareto-optimal. Using this we show that all the Benders cuts we generate are Pareto-optimal.

Theorem 2. *Given a simple cut-set A^k, the Benders cut $\theta \leq \sum_{a \in A^k} u_a y_a = \bar{\theta}^k(y)$, is Pareto-optimal.*

Proof. The cut-set A^k does not have to be minimal in the original network. Now, for any other Benders cut $\theta \leq \bar{\theta}^l(y)$, we have another cut-set $A^l = \{a \in A | \gamma_a^l = 1\} \subset A$, and $A^k \neq A^l$. If we compare these two Benders cuts, we get

$$\theta \leq \sum_{a \in A^k} u_a y_a = \bar{\theta}^k(y), \text{ and}$$
$$\theta \leq \sum_{a \in A^l} u_a y_a = \bar{\theta}^l(y).$$

For $\theta \leq \bar{\theta}^k(y)$ to be Pareto-optimal, we need to find a solution y^* such that $\bar{\theta}^k(y^*) < \bar{\theta}^l(y^*)$. Because $A^k \neq A^l$, we choose an arc a' such that $a' \in A^l$ and $a' \notin A^k$. Now we can take the solution

$$y_a^* = \begin{cases} 0, & a \in A^k \\ 1, & \text{otherwise,} \end{cases}$$

which is to turn off all arcs in A^k and open all other arcs. Our Benders cuts now look like:

$$\theta \leq \sum_{a \in A^k} u_a y_a^* = 0, \text{ and}$$
$$\theta \leq \sum_{a \in A^l} u_a y_a^* \geq u_{a'} > 0.$$

So $\theta \leq \bar{\theta}^l(y)$ does not dominate $\theta \leq \bar{\theta}^k(y)$. Since A^l is arbitrary, we have that $\theta \leq \bar{\theta}^k(y)$ is Pareto-optimal. □

3.5.4 Results

We test our implementation on the same data as Boland *et al.* (2014), using all combinations of the improvements described in Section 3.5.3. All our code is implemented in Python 3.5 (64-bit) as part of the Anaconda 4.1.1 distribution and uses the Gurobi 6.5 (64-bit on 8 threads) solver package [44], running on a distributed computing machine. Python has an in-built hash table functionality called dictionaries. The machine contains Intel Xeon processors (2.4GHz) with 8 cores and 24GB of RAM assigned to each job. All instances are given 1 hour in which to prove optimality, with the exception of the real-world instances, which are given 2 hours.

Data we collect includes the total run time of the program, the optimality gap, the number of branch-and-bound nodes processed and how many times the sub-problem was solved and recalled from memory. For the warm start, we also record the number of warm start iterations performed. When comparing solving times of the programs over instances of differing difficulty, we use the shifted geometric mean with shifting parameter 10s, as it focuses on the ratios between solving times, and helps to "prevent the hard instances from dominating the reported result" [**?**]. For a list of n solving times, t_i, where $i \in \{1, ..., n\}$, the shifted geometric mean with shift parameter h is given by

$$\text{SGM} = \exp\left[\frac{1}{n}\sum_{i=1}^{n} \ln(t_i + h)\right] - h. \tag{SGM}$$

Because $u_a \in \mathbb{N}$ for our data sets, all feasible solutions will have integer objective values. This allows us to set a termination condition for the MIP gap, since an absolute gap of less than one is sufficient to prove optimality. Without this condition, in some rare cases the solver finds the optimal solution in less than 10 minutes, and then spends vast amounts of time trying to close the gap by exploring hundreds of thousands of nodes. Since this is unnecessary, we will terminate the program if the absolute gap is less than 0.999.

Comparing our results with those of Boland *et al.* (2014) is not a simple task. As they use a straight MIP formulation in CPLEX and a number of heuristics, it is difficult to report optimality gaps. For the heuristics, the optimality gap is computed using the best upper bound found by the CPLEX implementation, which is not proving optimality in many cases. This means their optimality gaps are over-estimates, and their heuristics may be closer to optimality than reported. As such, we will compare our results to a direct MIP formulation in Gurobi.

Simulated data

Each of the three constructed data sets from Boland *et al.* (2014) has eight networks of increasing size, and each network has 10 randomly-generated lists of maintenance requests, giving 80 instances per instance set. For all instances, the number of jobs per arc is between 5 and 15, and the duration is between 10 and 30 time steps. For the first instance, the number of possible starting times for each job ranges between 1 and 35, whereas in the second instance set, each job has between 25 and 35 potential start times. The second instance set is thus more difficult to solve in general, because there is a much higher chance of having jobs where the potential starting window is larger than the duration of the job,

Table 3.2: Comparison of MIP and DBD implementations on simulated networks. For each instance set and each network, the average solving time and number of instances solved to optimality are shown.

		1	2	3	4	5	6	7	8
					Instance set 1				
MIP	Time (s)	1691.8	3606.6	55.5	3612.9	3617.6	3613.4	3625.9	3215.1
	Solved	6	0	10	0	0	0	0	2
DBD	Time (s)	21.3	57.7	8.3	1540.1	3600.1	933.4	2633.0	927.3
	Solved	10	10	10	7	0	9	3	8
DBD+PC	Time (s)	18.0	61.3	10.7	1611.9	3600.1	605.5	2667.1	846.2
	Solved	10	10	10	7	0	10	3	8
DBD+WS	Time (s)	22.5	73.3	8.5	1517.3	3600.1	1073.6	2536.2	935.7
	Solved	10	10	10	7	0	10	4	8
DBD+HE	Time (s)	17.8	64.1	12.2	1851.9	3600.1	730.7	2681.8	874.6
	Solved	10	10	10	6	0	10	3	9
					Instance set 2				
MIP	Time (s)	3255.7	3607.0	57.2	3614.1	3617.7	3613.8	3626.3	3634.8
	Solved	1	0	10	0	0	0	0	0
DBD	Time (s)	57.4	3418.3	11.0	2899.6	3600.7	3600.2	3314.2	1560.2
	Solved	10	3	10	2	0	0	1	7
DBD+PC	Time (s)	52.2	3598.6	12.3	2944.8	3600.6	3600.1	3301.6	865.0
	Solved	10	1	10	2	0	0	1	8
DBD+WS	Time (s)	55.6	3425.4	11.3	2924.6	3601.6	3600.2	3600.3	1126.7
	Solved	10	1	10	2	0	0	0	8
DBD+HE	Time (s)	53.4	3402.3	15.2	2919.0	3602.5	3600.1	3261.8	1094.5
	Solved	10	2	10	2	0	0	1	8
					Instance set 3				
MIP	Time (s)	9.0	25.7	14.6	155.6	219.5	83.7	749.8	591.9
	Solved	10	10	10	10	10	10	10	10
DBD	Time (s)	3.1	4.9	5.3	10.6	16.1	11.3	27.8	32.4
	Solved	10	10	10	10	10	10	10	10
DBD+PC	Time (s)	3.5	5.4	6.1	12.0	16.6	13.7	30.2	36.0
	Solved	10	10	10	10	10	10	10	10
DBD+WS	Time (s)	3.3	5.2	5.4	11.1	16.8	11.9	31.2	34.5
	Solved	10	10	10	10	10	10	10	10
DBD+HE	Time (s)	4.0	5.9	7.0	13.3	20.4	13.9	30.0	39.6
	Solved	10	10	10	10	10	10	10	10

which causes the problem discussed in Section 3.5.3. Finally, there is a third instance set where the number of possible starting times for each job is between 1 and 10. This is an especially easy case because there will almost never be a job with the aforementioned problem.

The single most important requirement for solving any of these problems is disaggregation of the sub-problems. Applying standard Benders decomposition without any separation of the sub-problem results in performance worse than that of the direct MIP implementation. Where the MIP implementation may solve within seconds, the Benders decomposition implementation terminates after 1 hour with an optimality gap of a few percent. However, applying disaggregated Benders decomposition gives a significant speed increase over the MIP, and thus an even greater increase over standard Benders decomposition.

Table 3.2 contains a comparison of MIP and DBD with each of the three main features: pre-cuts

Table 3.3: Comparison between direct MIP implementation in Gurobi and disaggregated Benders decomposition implementation with recall but no other features (DBD). For each instance set, there are 80 instances. Reported are the number of instances solved to optimality and the number of instances where the LP-Relaxation is solved (LPR) by the MIP, the shifted geometric mean of solving times for instances that are solved by both methods, the geometric mean of optimality gaps for instances that are not solved by either and the fraction of CPU time spent solving the sub-problems in the callback

	Instance set	1	2	3
MIP	Completed (LPR)	18 (80)	11 (69)	80 (80)
	S.G.M. Time (s)	138.86	58.27	87.20
	Avg. Gap (%)	0.673	1.565	-
DBD	Completed	57	33	80
	S.G.M. Time (s)	13.03	10.90	11.90
	Avg. Gap (%)	0.046	0.179	-
	Avg. Callback Fraction	0.0198	0.0135	0.0596

Table 3.4: Comparison of disaggregated Benders decomposition with and without saving solutions to sub-problems and recalling them when they reoccur. We report the number of times each technique solves fastest or to a smaller optimality gap, the number of instances solved to optimality and the shifted geometric mean of the times in seconds for the instances where both algorithms solve to optimality

	Without Recall			With Recall		
Instance set	Wins	S.G.M. Time (s)	Completed	Wins	S.G.M. Time (s)	Completed
1	26	90.33	58	54	81.93	57
2	30	69.76	33	50	57.89	33
3	1	17.39	80	79	11.90	80

(+PC), warm start (+WS) and user-suggested heuristics (+HE). Shown are the arithmetic means of the solving times for each network and each instances set for each method, as well as the number of schedules (out of 10) solved to optimality for each. In all cases, DBD outperforms MIP, however the usefulness of each improvement on their own is less clear.

Table 3.3 shows the difference between the direct MIP and disaggregated Benders decomposition (DBD). The MIP fails to solve many instances from sets 1 and 2, and for the problems it does solve, it has a significantly higher solving time. There are 11 instances in set 2 for which the MIP can not solve the LP-relaxation within 1 hour. All instances solved by the MIP are solved by DBD, but DBD solves three times as many instances in sets 1 and 2 as does the MIP. For the instances not solved by either implementation, DBD manages to close the optimality gap significantly better than MIP, by an order of magnitude in all cases. Even for the easiest instances (instance set 3), DBD is able to prove optimality in a fraction of the time of the direct MIP. This shows that disaggregated Benders decomposition is significantly more efficient than solving the direct MIP for this problem, and is able to solve many instances to optimality.

The first improvement is the saving and recalling of solutions to the sub-problems using a hash table. Table 3.4 compares two cases, using disaggregated Benders decomposition with and without the recall feature. The three statistics reported are: the number of times each case is faster or solved to a lower optimality gap; the shifted geometric mean of the run times in seconds for instances where the

Table 3.5: Comparison of number of flow problems solved and recalled for networks and instance sets. We report the number of flow problems solved and recalled averaged over 10 schedules for each network and instance set, as well as the fraction of flow problems solved

		Network							
Instance set		1	2	3	4	5	6	7	8
	Solved	3231	7494	3668	15735	24160	16981	31046	24293
1	Recalled	28739	42790	13328	46701	62044	49272	53407	29900
	Fraction	0.102	0.151	0.221	0.254	0.282	0.259	0.365	0.419
	Solved	5404	14749	4245	30526	37926	43339	48533	36784
2	Recalled	38721	70469	12350	70217	71496	87960	74913	38063
	Fraction	0.123	0.174	0.273	0.311	0.348	0.331	0.390	0.460
	Solved	1420	2082	2298	3539	4515	3661	5270	5203
3	Recalled	13246	15001	10596	16276	16801	17854	19063	18129
	Fraction	0.098	0.124	0.183	0.181	0.214	0.172	0.220	0.229

Table 3.6: Comparison of disaggregated Benders decomposition with and without adding initial cuts. We report the number of times each technique solves fastest or to a smaller optimality gap, the number of instances solved to optimality and the shifted geometric mean of the times in seconds for the instances where both algorithms solve to optimality

	Without Initial Cuts			With Initial Cuts		
Instance set	Wins	S.G.M. Time (s)	Completed	Wins	S.G.M. Time (s)	Completed
1	44	81.93	57	36	79.52	58
2	40	66.77	33	40	56.99	32
3	74	11.90	80	6	13.20	80

solver finds the optimal solution in both cases; and the number of instances in each set that solve to optimality in each case.

The only plus for the "without recall" case is that it solves one more instance to optimality in instance set 1 than when the solver uses the recall feature. This can be attributed to the possibility of finding alternate dual solutions, and hence Benders cuts, and thus taking a different solution trajectory that allows the solver to find the solution more quickly. All other data suggests using the recall feature is better than not using recall. It wins more than half the time, with lower average run times. Using the recall feature is significantly better on the easiest instances, and less so on the harder instances. All future algorithm comparisons will use the recall feature.

Table 3.5 shows how often the same flow problem occurs in various instances. For each instance set and network, the average number of flow problems solved and recalled are shown, as well as the fraction of times a network flow problem is solved as opposed to recalled from memory. In all cases, more solutions are recalled than solved, and in the smaller instances, as many as 90% of sub-problems have already been solved for other time periods.

The next feature we consider is adding initial cuts as described in Section 3.5.3. The results in this case are less obvious, with initial cuts having a smaller impact than the recall feature. As the easiest instances solve so quickly, adding initial cuts only slows down the procedure, with 74 of the 80 instances solving faster without initial cuts. For the more difficult instances, it is evenly split in terms

Table 3.7: Comparison of disaggregated Benders decomposition with and without adding user heuristics and using a warm start. The number of times each technique solved fastest or to a smaller optimality gap, the number of instances that were solved to optimality and the shifted geometric mean of the times in seconds for the instances where both algorithms solved to optimality are reported

	Without Heuristics			With Heuristics		
Instance set	Wins	S.G.M. Time (s)	Completed	Wins	S.G.M. Time (s)	Completed
1	52	81.93	57	28	91.88	59
2	51	61.15	33	29	61.28	31
3	74	11.90	80	6	12.58	80
	Without Warm Start			With Warm Start		
Instance set	Wins	S.G.M. Time (s)	Completed	Wins	S.G.M. Time (s)	Completed
1	48	78.16	57	32	78.08	57
2	52	75.53	33	28	75.61	33
3	75	11.90	80	5	14.41	80

of the number of wins, but the average solving time is smaller when using initial cuts. This means using initial cuts is faster in the cases when they win, and only slightly slower when they lose.

The use of user-suggested heuristics or a warm start, as described in Section 3.5.3, do not improve the performance of Benders decomposition for small instances of this problem. Often adding these features adds extra computational effort for little benefit. The features have the least negative impact on instance set 2, the hardest of the simulated instances. Using a warm start is slightly more beneficial than user heuristics, particularly for instance set 1.

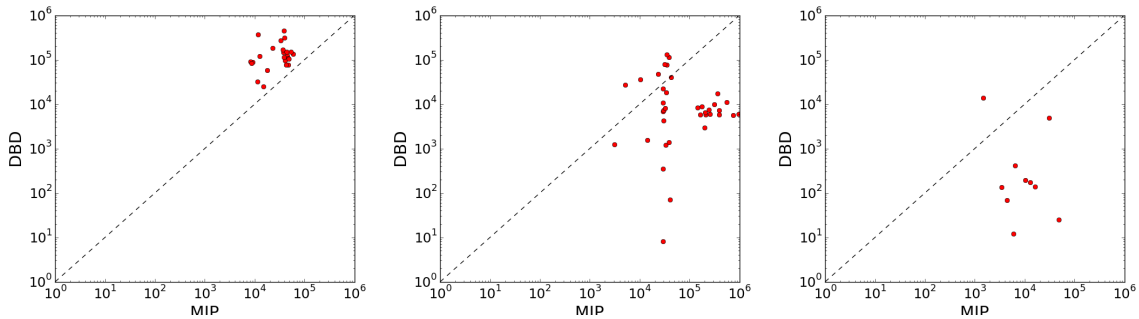
We also test all combinations of the features on the simulated instance sets. None of the combinations have a substantial impact on the time taken to solve problems or the number solved to optimality. For the larger, real-world instances, this is not the case, so we show the results for all possible combinations.

While Benders decomposition does exhibit poor convergence in these instances, it is still possible to prove optimality in many cases, as well as find good solutions quickly. The key features that enable this are the disaggregation of the sub-problem and embedding Benders decomposition inside a branch-and-cut framework.

Number of branch-and-bound nodes

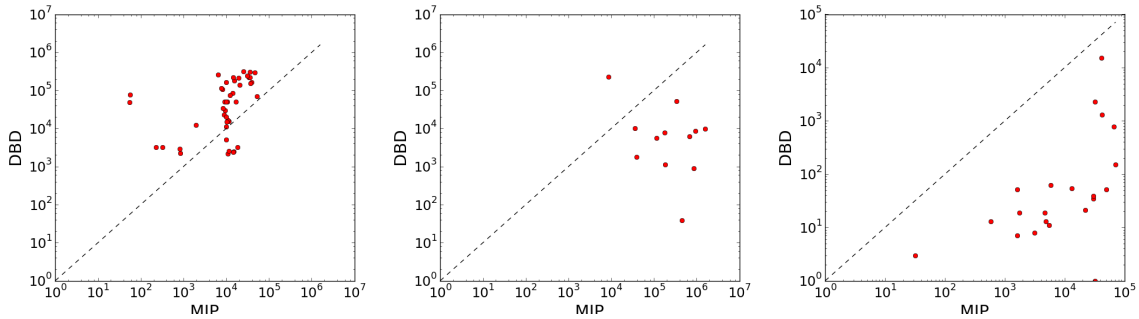
One particularly interesting result is the number of branch-and-bound nodes processed by Gurobi in the MIP and DBD models. Since the Benders master problem is a relaxation of the original MIP, we expect it to process branch-and-bound nodes more quickly, because it is a smaller model with fewer variables and constraints. Also, we know the LP-relaxation of the Benders master problem can be no tighter than the original MIP, and thus one could reasonably expect that to solve the Benders master problem to optimality would require exploring more branch-and-bound nodes than in the case of the MIP. Our results show the opposite.

Figure 3.7 compares the numbers of branch-and-bound nodes explored by MIP and DBD for each instance in instance set 1. The dashed line represents equality: points on this line are instances where



(a) Neither (b) DBD but not MIP (c) Both

Figure 3.7: Comparison of the number of branch-and-bound nodes explored by the MIP and DBD implementations for instance set 1. (a) Instances where neither DBD nor MIP solve to optimality (b) Instances where DBD solves to optimality but MIP does not (c) Instances where both DBD and MIP solve to optimality



(a) Neither (IS 2) (b) DBD but not MIP (IS 2) (c) Both (IS 3)

Figure 3.8: Comparison of the number of branch-and-bound nodes explored by the MIP and DBD implementations for instance sets 2 and 3. (a) Instances in set 2 where neither DBD nor MIP solve to optimality (b) Instances in set 2 where DBD solves to optimality but MIP does not (c) Instances in set 3 where both DBD and MIP solve to optimality

DBD and MIP explore the same number of nodes. Points above the line represent instances where DBD explores more nodes than MIP, and points below the line are instances where MIP explores more nodes than DBD. The instances are also separated by which methods solve them to optimality.

To start, there are no instances solved by MIP and not by DBD. Figure 3.7a contains instances that are not solved to optimality by either method in the 1-hour time frame. All points are above the line, which means DBD explores more nodes than MIP. That is to say, given a fixed amount of time, DBD consistently processes more nodes than MIP. This lines up with our expectations.

Figure 3.7b represents instances where DBD solves to optimality, but MIP does not. This means each of these points is further to the left, but no lower than they would be if both solvers are allowed to run to optimality. The majority of the points are below the line, and the ones above the line are likely to move below it if MIP is given more time to run. Figure 3.7c shows instances where both methods solve to optimality. In all but one case, DBD uses fewer nodes than MIP.

Figure 3.8 shows the same trend for instance sets 2 and 3. Note that for instance set 2, the only instances MIP solves to optimality are from network 3, all of which are solved to optimality without

Table 3.8: Comparison of implementation performance for 2010 real-world instance. Optimality gap (in percent), best solution and best upper bound are reported for all eight combinations of the three main techniques

		Without Initial Cuts		With Initial Cuts	
		No Heuristics	Heuristics	No Heuristics	Heuristics
No Warm Start	% Gap	3.70	5.22	15.38	4.54
	Best Soln.	136.15	134.98	125.28	135.08
	Best Bound	141.19	142.03	144.54	141.22
Warm Start	% Gap	13.74	7.25	8.52	5.57
	Best Soln.	124.12	131.57	130.52	133.87
	Best Bound	141.18	141.11	141.64	141.33

Table 3.9: Comparison of implementation performance for 2011 real-world instance. Optimality gap (in percent), best solution and best upper bound are reported for all eight combinations of the three main techniques

		Without Initial Cuts		With Initial Cuts	
		No Heuristics	Heuristics	No Heuristics	Heuristics
No Warm Start	% Gap	6.33	4.62	8.64	6.17
	Best Soln.	140.43	139.26	134.00	139.93
	Best Bound	149.33	145.70	145.57	148.57
Warm Start	% Gap	8.68	3.98	7.65	6.17
	Best Soln.	134.17	140.10	135.07	136.96
	Best Bound	145.81	145.67	145.40	145.41

branch-and-bound. Figure 3.8a shows DBD processes nodes more quickly than MIP, and 3.8b shows DBD solves to optimality using fewer nodes than MIP. For instance set 3, all problems are solved to optimality by both methods, and in every case DBD uses fewer nodes than MIP.

This shows that in most cases DBD can solve instances to optimality using fewer nodes than MIP. Combined with the fact that DBD can process nodes more quickly than MIP, it demonstrates the practical strength of Benders decomposition.

Real world instances

We also test our techniques on the instance sets derived from real-world data, provided by Boland *et al.* (2014). The network is representative of the Hunter Valley Coal Chain, and there are two lists of jobs designed to span one year each, for 2010 and 2011. The time is discretised into hours, so there are 8761 time periods for each problem (since 2010 and 2011 are not leap years). The unrestricted flow over 1 year is 161.3 megatonnes (Mt), and the success of an algorithm is measured in the minimisation of the impact on the network. The majority of jobs have durations between 1 and 18 hours, while the potential time window is set at 2 weeks for each job.

The fact that the potential job window is significantly larger than the duration of the job in all cases leads to the problem described in Section 3.5.3. As such, Benders decomposition again exhibits poor convergence. However, solutions are comparable to those found by Boland *et al.* (2014). After two hours, the best solution found for the 2010 data is between 124.1 Mt and 136.2 Mt, with a confirmed

bound of 141.1 Mt. This translates to an impact of between 25.1 Mt and 37.18 Mt, with a bound of 20.2 Mt. Boland *et al.* (2014) found solutions with an impact between 24.6 Mt and 26.4 Mt in two hours. In this instance, the best solution is once again found without the use of any of the features, but there are some differences in relative performance.

Suggesting heuristic solutions is better in all cases except where no features are used at all. The best solution is often significantly better, and the optimality gap is lower. Similarly, using initial cuts is always better, with the exception of the no features case. Using a warm start, however, is rarely better. It is important to remember that Integer Programming is inherently chaotic, and as such the high quality of the solution where no features are used may simply come down to a good solution trajectory, where a good solution is found by the solver "by chance". Nevertheless, the trend is that pre-cuts and user-suggested heuristics are beneficial in this instance.

For the 2011 data, solutions between 134.0 Mt and 140.4 Mt are found, with a confirmed bound of 145.4 Mt. This translates to an impact of between 20.9 Mt and 27.3 Mt, with a bound of 15.9 Mt. Boland *et al.* (2014) found solutions with an impact between 19.8 Mt and 20.5 Mt in the same time.

In this case, the trends are slightly different. Suggesting user heuristics is still always a good idea for these large problems, with an improvement occurring in all cases. Using initial cuts is often a bad idea, with only one improvement out of four cases. Using a warm start is the better option, with three cases showing improvement. The smallest optimality gap is found using Heuristics and a warm start without initial cuts.

3.5.5 Conclusion

We have shown that, while disaggregated Benders decomposition displays poor convergence for many instances of this problem, it is still an effective technique for solving the MaxTF-FAO problem. The key aspects that provide the most benefit are the disaggregation of sub-problems and the saving-and-recalling of solutions to those sub-problems. In many simulated cases, optimal solutions can be found in a short enough time to be practically useful, and we have proven optimality of a number of cases that have not previously been proven. The amount of choice in the real world problems makes it difficult to prove optimality, but reasonable solutions can be returned in a short amount of time, with objective values similar to those returned by heuristic methods.

Implementing disaggregated Benders decomposition in a state-of-the-art solver leads to fewer branch-and-bound nodes needing to be processed than the solver would normally require. This suggests the smaller master problem is more tractable to the solver and allows for more effective automatic use of techniques such as cutting planes or heuristic generation. It is likely that similar benefits will occur in other problems when Benders decomposition is applied. The strength of these benefits will likely increase over time as state-of-the-art solvers improve.

In future we would like to consider more broadly the class of integrated network design and scheduling problems to which this technique applies. It would also be interesting to look at other problems to which disaggregated Benders decomposition can be applied, and see if similar benefits

can be obtained.

3.6 Discussion of Paper: Disaggregated Benders Decomposition for Solving a Network Maintenance Scheduling Problem

This paper demonstrates the usefulness of Benders decomposition for solving large MIPs. It also exemplifies the key issues with implementing Benders decomposition: disaggregation of the sub-problems is necessary and embedding Benders decomposition in a branch-and-cut framework is efficient. On top of this, it also shows there are a number of details that can have an effect on the power of Benders decomposition.

The most important point is that disaggregation of the sub-problem is necessary in this problem. The sub-problems are independent between time periods, and in most instances there are 1000 time periods, which makes disaggregation extremely powerful. As noted in the paper, Benders decomposition performs far worse than the MIP without disaggregation of the sub-problem. This is mostly because applying the Benders cuts separately to 1000 different auxiliary variables is far tighter than applying aggregated cuts to a single variable.

There are also details about the similarity of the sub-problems. Since they are identical in every way except the time index, which does not appear in any of the parameters of the sub-problem, it is possible to build a single model for solving all sub-problems. More than this, it is likely that some network configurations will appear multiple times, perhaps for different time periods. As a result, storing the solutions to sub-problems and recalling them when they reappear leads to a small reduction in time taken to solve the problem. This is mostly a reduction in the time spent solving the sub-problems.

The storage of solutions to the sub-problems is also important since when we find a new Benders cut for a certain time period, we do not apply it to all time periods. While it is a valid and possibly useful Benders cut for all time periods, it is likely that the cut will be unnecessary for a large number of time periods, and the extra cuts would thus unnecessarily slow the master problem down. This is in line with the lazy maxim: only that which is necessary, and *only when it is necessary*.

The methods outlined in this paper are applicable to a wide range of problems. One of the closest-related problems to the network maintenance scheduling problem is the network restoration problem considered by Nurre *et al.* (2012). The formulation of the network restoration problem is almost identical to that of the network maintenance problem, and the only differences appear in the master problem of the Benders decomposition, meaning one could easily adapt such formulations to handle both problems.

There are many groups of problems with such similarities, and could easily be described as a class of problems, where all members of the class benefit from the same techniques. The next problem we consider is the Dynamic UFL and Network Design problem (DUFLNDP), which is part of a more general class of facility location and network design problems. This class will be described in Section 3.8.

3.7 Paper: Disaggregated Benders decomposition and branch-and-cut for solving the budget-constrained dynamic uncapacitated facility location and network design problem

Abstract

We present an approach for solving to optimality the budget-constrained Dynamic Uncapacitated Facility Location and Network Design problem (DUFLNDP). This is a problem where a network must be constructed or expanded and facilities placed in the network, subject to a budget, in order to satisfy a number of demands. With the demands satisfied, the objective is to minimise the running cost of the network and the cost of moving demands to facilities. The problem can be disaggregated over two different sets simultaneously, leading to many smaller models which can be solved more easily. Using disaggregated Benders decomposition embedded in a branch-and-cut framework, we solve many instances to optimality that have not previously been solved. We use an analytic procedure to generate Benders optimality cuts which are provably Pareto-optimal.

3.7.1 Introduction

In this paper we apply Benders decomposition to a facility location and network design problem, specifically looking at a number of ways of improving convergence of the algorithm. In particular, we disaggregate the Benders sub-problems, use an alternative to the standard Benders feasibility cuts and analytically construct Benders optimality cuts. We also prove the Pareto-optimality of the analytic Benders cuts and discuss the importance of using Pareto-optimal cuts.

Facility location problems are important in many areas of both industry and government. From deciding the location of stores and warehouses, to important services such as police, fire and health, facility location problems can have a large impact on a population. Equally important are network design problems, such as road or utility network optimisation. We are interested in the combination of these two types of problems.

The facility location problem dates back to the start of the 20th century [77], and is the basis of many more detailed problems. Benders decomposition [24] is an ideal technique for solving facility location problems, particularly the uncapacitated facility location (UFL) problem [60]. Geoffrion [45] generalises the concept of Benders decomposition and lays out a framework that includes optimality and feasibility cuts, and Geoffrion and Graves [58] apply Benders decomposition to a multicommodity variant of the facility location problem to great effect. Magnanti and Wong [46] explore regular and disaggregated Benders decomposition, apply it to the UFL problem, and propose an interior point method for accelerating convergence of the algorithm.

More recently, an efficient implementation of Benders decomposition for the UFL is demonstrated by Fischetti et al. (2017). They apply disaggregated Benders decomposition with a number of additional features which are useful, particularly for the UFL. Tang, Jiang and Saharidis (2013) use disaggregated Benders decomposition to solve a capacitated facility location problem where the

capacities could be modified for a cost [65]. They also consider adding extra constraints to enforce feasibility and tighten the lower bound on the objective value, which are important in the application of Benders decomposition.

Capacitated facility locations are more difficult to solve with Benders decomposition. This is because the shared capacity constraints prevent disaggregation of the sub-problems, one of the most powerful improvements of Benders decomposition. Fischetti, Ljubić and Sinnl (2017) examine how to adapt the techniques for solving the uncapacitated version to the capacitated problem. Castro *et al.* (2017) also apply Benders decomposition to a capacitated facility location problem. An example of a richer variant of capacitated facility location problems is presented in Jena, Cordean and Gendron (2016), who solve the problem using Lagrangian relaxation-based techniques.

Network flow and design problems have also been a major area of study over the last century. Today, many efficient methods for finding the maximum flow through a network exist [73, 74]. As such, more recent studies tend to focus on network design problems, where the network itself is optimised to achieve some goal, such as maximising the throughput of the network over time. Many of these problems are excellent candidates for Benders decomposition. Nurre, Cavdaroglu and Wallace (2012) consider a problem where a utilities network has been partially destroyed, and the reconstruction must be scheduled to maximise total throughput of the network over time. Boland *et al.* (2014) find the optimal maintenance schedule of a network, also to maximise throughput.

Magnanti, Mireault and Wong (1986) apply Benders decomposition to the Uncapacitated Network Design problem, which forms the basis of many fixed-charge network design problems. In particular, they generate Pareto-optimal Benders cuts to assist convergence of the algorithm. We extend their work to our problem, and in particular generate Pareto-optimal Benders cuts without needing to solve any additional linear programs (LP). A survey of Benders decomposition applied to fixed-charge network design problems can be found in Costa (2005).

A subset of network design problems are hub location problems, where a number of hubs must be located to minimise the cost of routing demands through a network. One example of this is the Hub Line Location problem, considered by de Sa *et al.* (2015), where hub facilities must be built in a public transit network and connected in a line. The objective is to minimise the weighted travel time of all demands through the network. Another example is the Uncapacitated Multiple Allocation Hub Location problem considered by Camargo, Miranda Jr. and Luna (2008), where hubs must be built so demands can be routed between locations via hubs. All of the above studies apply Benders decomposition to great effect. Contreras and Fernández (2014) solve the Supermodular Hub Location problem using techniques very similar to Benders decomposition, and also employ branch-and-cut as an efficient solution technique. de Sa *et al.* (2013) also apply Benders decomposition to another hub location problem, with a number of improvements such as a “warm start”, disaggregation of the sub-problems and modified feasibility cuts. For more information on hub location problems, the reader is directed to Laporte, Nickel and da Gama (2015).

It is known that embedding Benders decomposition in a branch-and-cut framework is efficient [80, 82]. Since 2012, a feature has been available in Gurobi (and earlier in CPLEX, although the feature

was introduced and improved gradually), known as “lazy constraints”, which provides the ability to add additional constraints to the model at nodes of the branch-and bound tree. This is, in essence, branch-and-cut. As such, our implementation of Benders decomposition will use lazy constraints to add Benders cuts during the solution process of the master problem.

We are considering the budget-constrained Dynamic Uncapacitated Facility Location and Network Design Problem (DUFLNDP) presented by Ghaderi and Jabalameli (2013). The government sets a fixed budget every year for the construction of new health clinics and roads, and one must work within that budget to minimise the running cost of the network while satisfying all demand for health services by routing demand through the network to health clinics.

The remainder of this paper is structured as follows: Section 3.7.2 contains our reformulation of the DUFLNDP, which is the base model to which we apply disaggregated Benders decomposition in Section 3.7.3. This section also covers many details around the implementation of disaggregated Benders decomposition such as Pareto-optimality of Benders optimality cuts and feasibility of sub-problems. In Section 3.7.4 we describe the use of a warm start with Benders decomposition to improve the initial LP-bound. Our computational results are in Section 3.7.5, before concluding with Section 3.7.6.

3.7.2 Model Formulation

Ghaderi and Jabalameli (2013) introduce the budget-constrained Dynamic Uncapacitated Facility Location and Network Design problem (DUFLNDP), which is defined on a network of locations. Every location is a client, and all have the potential to host a facility for servicing clients. There is a set of potential links between locations, on which arcs of the network can be constructed.

The problem covers a number of time periods. At each time there are budgets for opening new facilities and links. Open facilities and links also have associated maintenance or operating costs, which, together with the demand routing costs, form the total cost which is to be minimised.

The main assumptions in this problem are:

- Facilities and links have unlimited capacity
- Once opened, facilities and links will remain open until at least the end of the planning horizon
- Facilities and links are opened instantaneously between time periods

Our notation is slightly different from Ghaderi and Jabalameli (2013), in particular the variable names. We also present a simplified version of the budgetary constraints which achieve the same outcome. The time periods we are optimising over start at 1, and if a network exists already, we denote that as being at time 0. We now present the model formulation:

Sets

N	Set of network nodes. These include clients and facilities
A	Set of network arcs, both existing and potential. $A \subseteq N \times N$
T	Set of time periods

Parameters

d_{kt}	Demand of client $k \in N$ at time $t \in T$
g_{it}	Cost of opening facility at node $i \in N$ at time $t \in T$
c_{ijt}	Cost of constructing arc $(i,j) \in A$ at time $t \in T$
ρ_{ijt}	Cost per unit of routing demand on arc $(i,j) \in A$ at time $t \in T$
f_{it}	Operating cost of open facility $i \in N$ at time $t \in T$
h_{ijt}	Operating cost of open arc $(i,j) \in A$ at time $t \in T$
$\bar{B}_t$	Available budget for opening facilities at time $t \in T$
$\hat{B}_t$	Available budget for opening arcs at time $t \in T$

Variables

W_{it}	1 if facility $i \in N$ is open at time $t \in T$, 0 otherwise
X_{ijt}	1 if arc $(i,j) \in A$ is open at time $t \in T$, 0 otherwise
Z_{ijkt}	Fraction of demand of client $k \in N$ travelling along arc $(i,j) \in A$ at time $t \in T$
U_{it}	1 if facility $i \in N$ is constructed at time $t \in T$, 0 otherwise
V_{ijt}	1 if arc $(i,j) \in A$ is constructed at time $t \in T$, 0 otherwise

Objective

$$\text{Minimise} \sum_{t \in T} \left(\sum_{i \in N} f_{it} W_{it} + \sum_{k \in N} \sum_{(i,j) \in A} \rho_{ijt} d_{kt} Z_{ijkt} + \sum_{\substack{(i,j) \in A \\ i<j}} h_{ijt} X_{ijt} \right) \tag{3.49}$$

Constraints

$$W_{kt} + \sum_{j \in N} Z_{kjkt} \geq 1 \qquad \forall k \in N, \forall t \in T \tag{3.50}$$

$$\sum_{j \in N} Z_{jikt} \leq \sum_{j \in N} Z_{ijkt} + W_{it} \qquad \forall i,k \in N, i \neq k, \forall t \in T \tag{3.51}$$

$$Z_{jkkt} = 0 \qquad \forall k \in N, \forall j \in N, \forall t \in T \tag{3.52}$$

$$Z_{ijkt} + Z_{jikt} \leq X_{ijt} \qquad \forall (i,j) \in A, i<j, \forall k \in N, \forall t \in T \tag{3.53}$$

$$W_{i,t-1} + U_{it} = W_{it} \qquad \forall i \in N, \forall t \in T \tag{3.54}$$

$$X_{ij,t-1} + V_{ijt} = X_{ijt} \qquad \forall (i,j) \in A, \forall t \in T \tag{3.55}$$

$$\sum_{t'=1}^{t} \sum_{i \in N} g_{it'} U_{it'} \leq \sum_{t'=1}^{t} \bar{B}_{t'} \qquad \forall t \in T \tag{3.56}$$

$$\sum_{t'=1}^{t} \sum_{(i,j)\in A} c_{ijt'} V_{ijt'} \leq \sum_{t'=1}^{t} \hat{B}_{t'} \qquad \forall t \in T \tag{3.57}$$

$$X_{ijt} = X_{jit} \qquad \forall (i,j) \in A, i < j, \forall t \in T \tag{3.58}$$

$$W_{it} \in \{0,1\}, U_{it} \in \{0,1\} \qquad \forall i \in N, \forall t \in T \tag{3.59}$$

$$X_{ijt} \in \{0,1\}, V_{ijt} \in \{0,1\}, Z_{ijkt} \geq 0 \qquad \forall (i,j) \in A, \forall t \in T, \forall k \in N \tag{3.60}$$

The objective function (3.49) is the sum of three costs: the facility operating costs, the cost of routing demand to other facilities and the arc operating costs. Constraints (3.50) say that if a node k has an open facility, then it services its own demand. If not, all demand must leave the node. Constraints (3.51) are flow-conservation constraints at the nodes. Constraints (3.52) ensure demand can not be returned to the node of origin, thus eliminating cycles. Constraints (3.53) restrict the routing of demand to open arcs only. Constraints (3.54) and (3.55) control the opening of facilities and arcs based on the relevant construction variables, and constraints (3.56) and (3.57) ensure that the budget is not exceeded in any time period. Finally, constraints (3.58) enforce bi-directionality of the arcs.

3.7.3 Disaggregation and Benders decomposition

In this problem, the variables Z_{ijkt} are continuous, where all others are integer (binary). The constraints which contain the continuous variables are (3.50-3.53), and these constraints are separate for each $k \in N$ and $t \in T$. Thus it is possible to disaggregate the sub-problems by time and facility. A discussion of disaggregation level can be found in Section 3.7.5.

Benders Master Problem

We denote the contribution of the sub-problem (k,t) as θ_{kt}. The master problem is:

$$\text{Minimise} \sum_{t\in T} \left(\sum_{i\in N} f_{it} W_{it} + \sum_{k\in N} d_{kt} \theta_{kt} + \sum_{\substack{(i,j)\in A \\ i<j}} h_{ijt} X_{ijt} \right) \tag{3.61}$$

Subject to:

$$W_{i,t-1} + U_{it} = W_{it} \qquad \forall i \in N, \forall t \in T \tag{3.62}$$

$$X_{ij,t-1} + V_{ijt} = X_{ijt} \qquad \forall (i,j) \in A, \forall t \in T \tag{3.63}$$

$$\sum_{t'=1}^{t} \sum_{i\in N} g_{it'} U_{it'} \leq \sum_{t'=1}^{t} \bar{B}_t \qquad \forall t \in T \tag{3.64}$$

$$\sum_{t'=1}^{t} \sum_{(i,j)\in A} c_{ijt'} V_{ijt'} \leq \sum_{t'=1}^{t} \hat{B}_t \qquad \forall t \in T \tag{3.65}$$

$$\theta_{kt} \geq \text{BendersOptimalityCut}(m, \mathbf{W}, \mathbf{X}, k, t) \qquad \forall k \in N, \forall t \in T, \forall m \in \{1, ..., M\} \tag{3.66}$$

$$\text{BendersFeasibilityCut}(p, \mathbf{W}, \mathbf{X}) \qquad \forall p \in \{1, ..., P\} \tag{3.67}$$

$$W_{it} \in \{0,1\}, U_{i,t} \in \{0,1\}, \theta_{it} \geq 0 \qquad \forall i \in N, \forall t \in T \tag{3.68}$$

$$X_{ijt} \in \{0,1\}, V_{ijt} \in \{0,1\} \qquad \forall (i,j) \in A, \forall t \in T \tag{3.69}$$

Constraints (3.66) represent the disaggregated Benders cuts, which are added as necessary after solving the associated sub-problems, which will be covered in the Sub-Problems subsection. Similarly, constraints (3.67) represent the added constraints required for feasible sub-problems. M and P are the number of added Benders optimality and feasibility cuts respectively. We now solve this relaxed master problem with a single branch-and-bound tree. For each feasible integer solution, W^* and X^*, we solve each of the sub-problems and calculate their actual contributions to the master problem objective function. If necessary, we add more Benders optimality or feasibility cuts. This is called the main phase of the algorithm.

Initial feasibility

For the solution to be feasible, it must be possible to service the demand of every client for every time period. In the original MIP, this was ensured by the routing variables and constraints. After separating out the sub-problems, our master problem now has no constraints ensuring that there will be a path from every source to a facility, meaning that we may encounter feasible solutions to our master problem that are infeasible in the original MIP, and will make the sub-problems infeasible. The standard Benders decomposition framework includes Benders feasibility cuts [45] which find unbounded rays in the dual of the sub-problem and cut them off, however these are often ineffective [52, 55].

A second option is to augment the master problem with additional constraints to remove these solutions, without removing any solutions that are feasible in the original problem. Since links and facilities are only constructed, never destroyed, if the network is feasible in the first time period, it will be feasible for every time period. To ensure this happens, we modify the master problem to make the first time period a special case. The objective, parameters and variables remain unchanged, we only modify some constraints and add new ones. The modified and new constraints are:

Constraints

$$Z_{ijkt} \leq X_{ijt} \qquad \forall (i,j) \in A, \forall k \in N, \forall t \in T \tag{3.53a}$$

$$X_{ij,t-1} + V_{ijt} = X_{ijt} \qquad \forall (i,j) \in A, i < j, \forall t \in T, t > 2 \tag{3.55a}$$

$$X_{ijt} = X_{jit} \qquad \forall (i,j) \in A, i < j, \forall t \in T, t > 1 \tag{3.58a}$$

$$X_{ij,1} + X_{ji,1} \leq 1 \qquad \forall (i,j) \in A, i < j \tag{3.70}$$

$$X_{ij,0} + V_{ij,1} = X_{ij,1} + X_{ji,1} \qquad \forall (i,j) \in A, i < j \tag{3.71}$$

$$X_{ij,1} + X_{ji,1} + V_{ij,2} = X_{ij,2} \qquad \forall (i,j) \in A, i < j \tag{3.72}$$

$$\sum_{\substack{j \in N \\ (i,j) \in A}} X_{ij,1} + W_{i,1} \geq 1 \qquad \forall i \in N \tag{3.73}$$

$$\sum_{i \in N} W_{i,1} \geq 1 \tag{3.74}$$

The modification to constraint (3.53) combined with constraint (3.70) enforces directionality of arcs in the first time period and (3.55,3.58) are modified appropriately. Constraints (3.71-3.72) handle the budget constraints, to ensure that if a direction is built in the first time period, the opposite direction will be built for free in the second time period.

These modifications allow us to add constraints (3.73-3.74), which ensure that each location has either a facility at the location or an arc leaving the location, and that at least one facility must exist, respectively. This way, either a node is a facility, or it is connected to a node which is either a facility, or connected to a node... and so on. This only fails if a cycle occurs where multiple nodes are connected to each other and none have facilities, so cycle-breaking may be necessary. This change in formulation is more useful in the instances when there is no pre-existing network, as when there are fixed elements of the network there is less choice in its design.

IIS feasibility cuts

To handle the case where cycles occur, we add cycle-breaking feasibility cuts, where the sum of facilities in the cycle plus the sum of arcs leaving the cycle must be at least one. In the main phase, we identify such cycles and add the necessary constraints. In the warm start (see Section 3.7.4), however, identifying such cycles is more difficult due to the fractional values placed on arcs and nodes.

Using Gurobi, we compute the Irreducible Inconsistent Subsystem (IIS), which is "a subset of the constraints and variable bounds of the original model. If all constraints in the model except those in the IIS are removed, the model is still infeasible. However, further removing any one member of the IIS produces a feasible result." [44] The IIS is then a collection of capacity constraints on nodes and arcs which, when lifted, make the sub-problem feasible. This leaves us with the nodes and arcs which can be expanded or added to resolve the infeasibility. We then add a feasibility cut of the form:

$$\sum_{i \in \text{IIS}} W_{i0} + \sum_{(a,b) \in \text{IIS}} X_{ab0} \geq 1 \tag{3.75}$$

This ensures that enough facilities and arcs will be opened that the demand from the infeasible source nodes can be served. This is known as a combinatorial Benders cut [56], and they have been shown to be significantly stronger than the standard Benders feasibility cuts [55].

Sub-Problems

If we have a feasible solution for the integer variables W^* and X^*, we can solve the sub-problems as a collection of linear programs. Since d_{kt} only depends on k and t, we can leave it out of the objective of the sub-problem and instead apply it to the objective of the master problem. The contribution of each sub-problem to the master problem is represented by θ_{kt}. The goal of each sub-problem is to find the cheapest way of servicing the demand of that facility at that time. There are two possibilities for this: either the site is a facility and can service its own demand for free, or the demand is routed to the nearest (cheapest) open facility. For each $k \in N$ and $t \in T$ we have the sub-problem:

$$\text{Minimise} \sum_{(i,j) \in A} \rho_{ijt} Z_{ijkt} \tag{3.76}$$

Subject to:

$$-\sum_{j\in N} Z_{kjkt} \leq W^*_{kt} - 1 \tag{3.77}$$

$$\sum_{j\in N} Z_{jikt} - \sum_{j\in N} Z_{ijkt} \leq W^*_{it} \qquad \forall i \in N \setminus \{k\} \tag{3.78}$$

$$Z_{jkkt} = 0 \qquad \forall j \in N \tag{3.79}$$

$$Z_{ijkt} \leq X^*_{ijt} \qquad \forall (i,j) \in A \tag{3.80}$$

$$Z_{ijkt} \geq 0 \qquad \forall (i,j) \in A \tag{3.81}$$

Constraint (3.77) has been rearranged to show its similarity to (3.78), which we will take advantage of when formulating the explicit dual. There are two collections of dual variables that we are interested in: γ_i for each node constraint (3.77,3.78) and λ_{ij} for each arc constraint (3.80). These variables then lead to the following dual formulation:

$$\text{Maximise } \gamma_k - \sum_{i\in N} \gamma_i W^*_{it} - \sum_{(i,j)\in A} \lambda_{ij} X^*_{ijt} \tag{3.82}$$

Subject to:

$$\rho_{ijt} + \lambda_{ij} + \gamma_j - \gamma_i \geq 0 \qquad \forall (i,j) \in A, j \neq k \tag{3.83}$$

$$\lambda_{ij} \geq 0, \gamma_i \geq 0 \qquad \forall (i,j) \in A, \forall i \in N \tag{3.84}$$

The reason that constraints (3.83) do not apply when $j = k$ is because in those cases γ_j is replaced by the unbounded dual variable associated with constraints (3.79), and since this dual variable does not appear in the objective function, it can be set to a suitably large number, thus ensuring feasibility of the dual constraints for those arcs.

Constraint (3.83) ensures that the reduced cost of each arc is non-negative. This formulation yields the following Benders cut:

$$\theta_{kt} \geq \gamma_k - \sum_{i\in N} \gamma_i W_{it} - \sum_{(i,j)\in A} \lambda_{ij} X_{ijt} \tag{3.85}$$

One can solve these sub-problems as linear programs and extract the dual variables provided by the solver in order to construct a Benders cut. Alternatively, one can solve the sub-problems and produce the required dual variables analytically.

Analytic solution to Benders sub-problems

Each sub-problem is a shortest path problem. For each location, one must find the cheapest way of servicing its demand, either at a facility at the source or by routing the demand to another location with a facility. Each sub-problem is indexed by k and t, where k is the source node and t is the time period.

Magnanti and Wong (1981) note that the analytic dual variables for the UFL problem have a natural interpretation; the dual variables for the DUFLNDP also have a natural interpretation. γ_i represents the

Algorithm 5 Algorithm for computing dual variables for analytic Benders optimality cut for sub-problem (k,t), assuming the sub-problem is feasible

Begin with master problem solution $W^*_{it}, X^*_{abt}, \theta^*_{kt}\ \forall i \in N, \forall (a,b) \in A$
Compute shortest distance D_i from k to i for all nodes $i \in N \setminus \{k\}$
$\gamma_k = \min_j \{D_j | W^*_{jt} = 1, j \in N \setminus \{k\}\}$
for $i \in N \setminus \{k\}$ **do**
  $\gamma_i \leftarrow \max(0, \gamma_k - D_i)$
for $(i,j) \in A$ **do**
  if $X^*_{ijt} = 0$ **then**
    $\lambda_{ij} \leftarrow \max(0, \gamma_i - \gamma_j - \rho_{ijt})$
  else
    $\lambda_{ij} \leftarrow 0$
Add Constraint $\theta_{kt} \geq \gamma_k - \sum_{i \in N} \gamma_i W_{it} - \sum_{(i,j) \in A} \lambda_{ij} X_{ijt}$

saving associated with opening a facility at location i and λ_{ij} the saving from opening an arc from i to j. Magnanti and Wong also demonstrate for the UFL problem that Pareto-optimal Benders cuts can be determined without solving additional LPs. The DUFLNDP shares this property, as we now show.

Analytic Benders cut

The algorithm for computing the dual variables can be found in Algorithm 5. Note that it assumes that the facility at the source node k is closed, as otherwise the solution is trivial. The approach is also similar to that used by Magnanti *et al.* (1986), however there are some minor differences to accommodate the fact we can change the destination by placing additional facilities. We begin by constructing a shortest path tree from the source location, giving each node a distance D_i from the source node. If there is no path between k and i, then $D_i = \infty$. These distances follow a shortest-path property, namely $D_j \leq D_i + \rho_{ij}$. Now, the value of γ_k is assigned the length of the shortest path to the nearest open facility i^*, that is, $\gamma_k = \min\{D_i | W^*_{it} = 1\} \equiv D_{i^*}$. For all other nodes, $\gamma_i = \max(0, \gamma_k - D_i)$.

Next we calculate the values for the dual variables λ_{ij}, associated with the arcs $(i,j) \in A$. For all arcs (i,j), $\lambda_{ij} = \max(0, \gamma_i - \gamma_j - \rho_{ij})$. For open arcs ($X^*_{ijt} = 1$), $\lambda_{ij} = 0$ by the shortest path property.

Theorem 3. *The dual variables calculated using Algorithm 5 are dual optimal.*

Proof. For these dual variables to form a dual feasible solution, they must satisfy the Constraints (3.83). The constraints are trivially satisfied for any arc where $\gamma_i = 0$. For all closed arcs, $\lambda_{ij} \geq \gamma_i - \gamma_j - \rho_{ijt}$, which satisfies Constraint (3.83).

For any open arc (i,j), $\lambda_{ij} = 0$, so we must show that $\rho_{ij} + \gamma_j - \gamma_i \geq 0$. By the property of the shortest path distances, $D_j \leq D_i + \rho_{ij}$, or $\rho_{ij} + D_i - D_j \geq 0$. We also have, by construction of the dual variables, that $\gamma_j \geq \gamma_k - D_j$, or $D_j \geq \gamma_k - \gamma_j$. Finally, we are only considering where $\gamma_i > 0$, and in this case we have that $\gamma_i = \gamma_k - D_i$. Combining these, we get:

$$\rho_{ij} + \gamma_j - \gamma_i = \rho_{ij} + \gamma_j - (\gamma_k - D_i)$$

$$\geq \rho_{ij} + (\gamma_k - D_j) - (\gamma_k - D_i)$$
$$\geq \rho_{ij} + D_i - D_j$$
$$\geq 0$$

So the dual variables obtained from Algorithm 5 are dual feasible. The objective value given by these dual variables is the same as the optimal objective value of the primal problem, which is the length of the shortest path ($D_{i^*} = \gamma_k$), since for all nodes, either $\gamma_i = 0$ or $W^*_{it} = 0$, and likewise for arcs. Thus the dual variables form a dual optimal solution, and may be used to add a Benders optimality cut to the master problem. □

The cuts generated using these dual variables are Pareto-optimal, which is important for improving convergence of the master problem [46]. We prove they are Pareto-optimal in the following subsection. In terms of the natural interpretation, we place much of the savings on the arcs, since if facilities beyond closed arcs are opened, no saving will be obtained until the arcs are open.

Pareto-optimality of the analytically-derived Benders cuts

Magnanti and Wong (1981) describe the importance of using Pareto-optimal cuts when using Benders decomposition. In this section we show that the analytic Benders cuts are Pareto-optimal. Since a Benders cut is a linear function of the current network configuration, it can be described as $\theta \geq \bar{\theta}(y)$, for $y \in Y$ where Y is the set of all feasible solutions to the master problem. Let the contribution to the objective value for network configuration y be given by $\bar{\theta}^*(y)$. The following definitions are paraphrased from Magnanti and Wong:

Definition 8. *A Benders cut $\theta \geq \bar{\theta}^a(y)$ dominates another Benders cut $\theta \geq \bar{\theta}^b(y)$ if $\bar{\theta}^a(y) \geq \bar{\theta}^b(y)$ for all feasible $y \in Y$ and is a strict inequality for at least one feasible y.*

This definition leads to the following lemma:

Lemma 2. *If $\theta \geq \bar{\theta}^a(y)$ is dominated by $\theta \geq \bar{\theta}^b(y)$, then for all feasible solutions y^i where $\bar{\theta}^a(y^i) = \bar{\theta}^*(y^i)$, $\bar{\theta}^b(y^i) = \bar{\theta}^*(y^i)$.*

This is easy to see, since $\bar{\theta}^b(y^i) \leq \bar{\theta}^*(y^i)$ by definition of being a valid Benders cut, and $\bar{\theta}^b(y^i) \geq \bar{\theta}^a(y^i) = \bar{\theta}^*(y^i)$ by definition of being a dominating cut.

Definition 9. *A Benders cut $\theta \geq \bar{\theta}^a(y)$ is Pareto-optimal if it is not dominated by any other Benders cuts.*

One can prove that a Benders cut is Pareto-optimal by assuming that there exists another cut which dominates it, finding enough points $y \in Y$ where the Pareto-optimal cut equals the objective value, and specifying that the dominating cut must also equal the objective value at these points. This leads to all terms of the dominating cut being fixed to those of the Pareto-optimal cut. Thus there are no cuts which dominate the original cut, and it is Pareto-optimal. We now show that our analytic Benders optimality cuts are Pareto-optimal.

Theorem 4. *Benders optimality cuts derived using Algorithm 5 are Pareto-optimal*

Proof. All Benders cuts for this problem are of the form:

$$\theta_{kt} \geq \gamma_k - \sum_{i \in N} \gamma_i W_{it} - \sum_{(i,j) \in A} \lambda_{ij} X_{ijt} \tag{3.86}$$

Let the dual variables associated with the analytic Benders cut (*i.e.* the cut generated by Algorithm 5) be $\bar{\gamma}_i$ and $\bar{\lambda}_{ij}$. In the current solution, the closest open facility is i^* ($D_{i^*} = \bar{\gamma}_k$).

We assume that the current cheapest facility i^* is not the same as the location, *i.e.* $i^* \neq k$. If $i^* = k$, then the cut is trivial and not Pareto-optimal. In that scenario, one could take the same approach as Balinski by constructing the cut about the second-closest facility. We also assume that there exists another location $j \in N$ such that $d_j < d_{i^*}$, since otherwise the cut will again be trivial and not Pareto-optimal. We begin by defining some partitions of the nodes and arcs of the problem:

$\mathbf{F}^o = \{i | W^*_{it} = 1\}$, the set of open facilities,

$\mathbf{F}^c = \{i | W^*_{it} = 0\}$, the set of closed facilities,

$\mathbf{F}^+ = \{i | d_i \geq \bar{\gamma}_k\}$, the set of facilities at equal or greater distance than i^*,

$\mathbf{F}^- = \{i | d_i < \bar{\gamma}_k\}$, the set of facilities closer than i^*,

$\mathbf{L}^o = \{i | X^*_{ijt} = 1\}$, the set of open links, and

$\mathbf{L}^c = \{i | X^*_{ijt} = 0\}$, the set of closed links

Now assume there exists a Benders cut using the dual variables $\hat{\gamma}_i$ and $\hat{\lambda}_{ij}$, which dominates the analytic Benders cut. As they are both Benders cuts, they must both equal the objective value, and thus each other, for the current solution to the master problem. If we open a facility at the source, k, the objective value will be zero and the analytic cut will be tight, so the dominating cut must also equal zero for this solution. This leads to:

$$\begin{aligned} 0 =& \bar{\gamma}_k - \bar{\gamma}_k - \sum_{\substack{i \in N \\ i \neq k}} \bar{\gamma}_i W^*_{it} - \sum_{(i,j) \in A} \bar{\lambda}_{ij} X^*_{ijt} \\ =& 0 - \sum_{\substack{i \in \mathbf{F}^o \\ i \neq k}} \bar{\gamma}_i - \sum_{(i,j) \in \mathbf{L}^o} \bar{\lambda}_{ij} X^*_{ijt} \end{aligned}$$

Since $\bar{\gamma}_i \geq 0$ and $\bar{\lambda}_{ij} \geq 0$, we have that $\bar{\gamma}_i = \hat{\gamma}_i = 0 \; \forall i \in \mathbf{F}^o$ and $\bar{\lambda}_{ij} = \hat{\lambda}_{ij} = 0 \; \forall (i,j) \in \mathbf{L}^o$. Note that $i^* \in \mathbf{F}^o$, and so $\gamma_{i^*} = 0$. Returning to the current solution, the two cuts must equal each other, and for both cuts, either $\gamma_i = 0$ or $W_{it} = 0 \; \forall i \in N$, and similarly for arcs, so we have that $\hat{\gamma}_k = \bar{\gamma}_k$.

Now, for any other location $i \in N$, if we open a facility at i, the analytic cut will still be tight, so the dominating cut must also be tight. Since the only changes in both cuts is W_{it}, we have that $\bar{\gamma}_i = \hat{\gamma}_i$ $\forall i \in N$. All that remains is to show that $\bar{\lambda}_{ij} = \hat{\lambda}_{ij} \; \forall (i,j) \in \mathbf{L}^c$.

For any arc $(i,j) \in \mathbf{L}^c$ where $\hat{\gamma}_i \leq \rho_{ijt}$, $\hat{\lambda}_{ij} = 0$, which will be tight since even if a facility were opened at j, it would still be further away than the closest open facility, so $\bar{\lambda}_{ij}$ will also be zero. The other case where $\hat{\lambda}_{ij} = 0$ is when $\hat{\gamma}_j > \hat{\gamma}_i - \rho_{ijt}$, that is, the arc does not create a short-cut in the network.

In this case, opening the arc does not change the objective value and the analytic cut will be tight, so $\bar{\lambda}_{ij} = 0$ for all arcs where $\hat{\lambda}_{ij} = 0$.

If $\hat{\lambda}_{ij} > 0$, then $\hat{\lambda}_{ij} = \hat{\gamma}_i - \hat{\gamma}_j - \rho_{ijt}$ and $\hat{\gamma}_i > \rho_{ijt}$. If we open the arc (i, j) and the facility j, then the analytic cut will be:

$$\begin{aligned}\hat{\gamma}_k - \hat{\gamma}_j - \hat{\lambda}_{ij} =& \hat{\gamma}_k - \hat{\gamma}_j - (\hat{\gamma}_i - \hat{\gamma}_j - \rho_{ijt}) \\ =& \hat{\gamma}_k - \hat{\gamma}_i + \rho_{ijt}\end{aligned}$$

which is the length of the shortest path between k and i plus the length of the arc from i to j. Since $\hat{\gamma}_i > \rho_{ijt}$, this will be lower than the original path length, and thus j will be closer than i^* to k. So the analytic cut is tight at these points, and the dominating cut must also be tight. Since $\bar{\gamma}_j = \hat{\gamma}_j$ for all $j \in N$, we have:

$$\begin{aligned}\hat{\gamma}_k - \hat{\gamma}_j - \hat{\lambda}_{ij} =& \bar{\gamma}_k - \bar{\gamma}_j - \bar{\lambda}_{ij} \\ =& \hat{\gamma}_k - \hat{\gamma}_j - \bar{\lambda}_{ij} \\ \hat{\lambda}_{ij} =& \bar{\lambda}_{ij}\end{aligned}$$

So $\bar{\lambda}_{ij} = \hat{\lambda}_{ij}\ \forall (i, j) \in A$, and thus the dominating cut is identical to the analytic cut. So there are no cuts which dominate the analytic cut, and thus it is Pareto-optimal. □

This algorithm thus provides a way of generating Pareto-optimal Benders cuts without solving additional LPs. It is a replacement for the Magnanti-Wong core-point methods, and is arguably simpler, easier to implement and more reliable. Magnanti and Wong (1981) showed that the natural Benders cut for the UFL is Pareto-optimal, and this can also be proven using the above method. In future, we would like to find more problems where algorithms for generating Pareto-optimal cuts can be applied in place of more complicated methods.

Benders cut separation

When to generate Benders cuts is an important consideration, as generating them too frequently can lead to a large number of unnecessary cuts burdening the model. In the literature, studies typically generate Benders cuts according to one of three schemes: at every feasible branch-and-bound node encountered, at nodes that yield an improvement in the lower bound, or at nodes where new incumbent solutions are found [52, 59].

In our case, generating Benders cuts only for new incumbent solutions is sufficient. Combined with the warm start and our procedure for generating Pareto-optimal Benders cuts at integer solutions, any additional benefit from adding Benders cuts for fractional solutions is outweighed by the extra time spent solving the sub-problems at each feasible node. Other studies draw similar conclusions [59].

3.7.4 Warm start

When applying Benders decomposition to a problem, the master problem is "relaxed" by projecting out the variables of the sub-problem. A result of this is that the initial LP-optimum of the master

problem is lower (when minimising) than that of the original problem. This will mean that more effort must be expended in the branch-and-bound phase to find the optimal solution. This can be overcome by using a "warm-start" [52, 63], which involves solving the linear relaxation of the problem and using the results to add Benders cuts to the master problem. Performing this repeatedly until the bound does not increase substantially, or no more cuts are added, significantly tightens the initial LP-bound and reduces the runtime of the solver.

This yields significant improvements to the runtime of the program, however it is sometimes more useful to use continuous analogues of the Pareto-optimal analytic Benders cuts in the warm start. Because of this, we analytically construct the dual variables to be used in the pre-cuts. This yields the strongest cuts possible, which can improve the solution speed of the master problem. Some studies have found that performance could be increased by removing any warm start cuts with non-zero slack at the end of the warm start phase to reduce the number of constraints in the master problem [60]. In our case, some instances performed better and others performed worse, because some cuts that had non-zero slack at the LP-optimum were required for finding the IP-optimum, and were added a second time. As such, the results shown are for implementations where all warm start cuts are carried through the whole optimisation procedure.

Feasibility of sub-problems

The first thing to check, just like the main phase, is the feasibility of the sub-problems. In the warm start, because all variables are continuous and not integer (or binary), the arcs of the network are allowed to be partially open, and likewise for facilities. As such, it is no longer enough that there be a path to an open facility, instead we may require a collection of paths to route all demand to one or more facilities. As with the main phase, infeasible sub-problems will only occur when a cycle exists in the network. The IIS feasibility cuts are capable of handling the relaxed problem, and as such are always used in the warm start.

Solution of the sub-problems

After having ensured the feasibility of the relaxed solutions to the master problem, we solve the flow sub-problems as LPs and extract the paths from these results. When more than one path is required, it is because partially opened arcs or facilities are restricting the flow of demand. In most cases, the longest path will not have any of these restricting factors.

If there are n paths in the solution, there will be at least $n-1$ restricting factors. These restricting factors, denoted by the set $\mathscr{C}$, correspond to potential non-zero values for γ or λ dual variables, and as such there are several constraints on these values. The first is that the sum of the dual variables corresponding to the arcs and final facility of each path must equal the saving from travelling along the path. That is, given a path p of length L_p which ends at node dest_p, and the set of arcs on that path A_p:

$$\gamma_k - L_p = \gamma_{\text{dest}_p} + \sum_{(a,b)\in A_p} \lambda_{ab} \tag{3.87}$$

This equation ensures the reduced cost of each path is zero. In most cases, γ_k will be the length of the longest path (opening a facility allows demand from the longest path to be serviced at the source), however in the case where the longest path has restricting factors, the RHS of the above equation will be non-zero for the longest path and γ_k will be greater than L_p.

Another condition is that the value of the Benders cut must be equal to the objective value of the sub-problem for the current master problem solution. This is necessary for the dual variables to form a dual-optimal solution. The final condition is that the reduced cost of each arc is non-negative. If a solution is found which satisfies these three conditions, then it is dual optimal, and the dual variables can be used to construct a Benders cut.

If there are n paths and $n-1$ restricting factors, then these dual variables can be calculated directly by solving equation (3.87) for all paths simultaneously. In this case, the matrix will be non-singular and thus the values of the dual variables for all restricting factors can be determined. However, as there is much degeneracy in network flow problems, often there will be more than $n-1$ restricting factors for n paths. This occurs when a path has two or more restricting factors which lie only on that path. In this case, one can either determine which $n-1$ factors to use by eliminating any "extra" factors, or one can solve the following linear program:

$$\text{Minimise } \gamma_k \tag{3.88}$$

Subject to:

$$L_p - \gamma_k + \gamma_{\text{dest}_p} + \sum_{\substack{(a,b)\in\mathscr{C} \\ (a,b)\in A_p}} \lambda_{ab} = 0 \qquad \forall p \in P \tag{3.89}$$

$$\gamma_k - \sum_{i\in N} \gamma_i \bar{W}_{it} - \sum_{(i,j)\in A} \lambda_{ij} \bar{X}_{ijt} = \sum_{(i,j)\in A} \rho_{ijt} d_{kt} \bar{Z}_{ijkt} \tag{3.90}$$

$$\gamma_j - \gamma_i + \lambda_{ij} + \rho_{ijt} d_{kt} \geq 0 \qquad \forall (i,j) \in A \tag{3.91}$$

The effect of constructing analytic warm start Benders cuts this way can be seen in the Results section.

Budget cover inequalities

In addition to this, we also add inequalities on the budget variables, U and V, to potentially tighten the relaxed problem. The budget constraints are effectively a knapsack problem, and as such we can add cover inequalities similar to those described by Gu, Nemhauser and Savelsbergh (1998). After a solution to the relaxed problem is found, we check, for each time period, which facilities and arcs have been partially or wholly constructed. We sum the variables over all facilities/links and time periods up to and including the current time period, and if this is not an integer value, then some facilities or links have been partially opened.

S is the sum of facilities/links that have been opened up to this point in time. We then order the facilities/links from cheapest to most expensive to open, and if the sum of opening costs of the first $\lceil S \rceil$ facilities/links is greater than the available budget, we add a new constraint of the form:

$$\sum_{t'=1}^{t} \sum_{i \in \bar{S}} U_{it'} \leq \lfloor S \rfloor \tag{3.92}$$

$$\sum_{t'=1}^{t} \sum_{\substack{(i,j) \in \bar{S} \\ i<j}} V_{ijt'} \leq \lfloor S \rfloor \tag{3.93}$$

where $\bar{S}$ is the cheapest $\lceil S \rceil$ facilities/links. If it is impossible to open all facilities/links in $\bar{S}$, then turning one off to open another facility/link which is more expensive will not be possible either. Thus, these cuts can be lifted to include all facilities/links more expensive than the most expensive member of $\bar{S}$.

3.7.5 Results

We are comparing two different formulations and a number of implementation features using the public data set from Ghaderi and Jabalameli (2013). The tests are performed on a high-performance computing system running Linux. Each job was assigned a maximum of 8 cores running at 2.4GHz each, and 56GB of RAM. The implementations are written in Python 3 as part of the Anaconda distribution (4.1.1) and use the Gurobi 7.0.1 [44] optimisation package. All software used is 64-bit. The maximum runtime for each instance is $50|N||T|$ seconds, where $|N|$ is the number of nodes and $|T|$ is the number of time periods, which is consistent with Ghaderi and Jabalameli (2013).

All instances are grouped in threes, where each instance in a group is on the same network, tested over 5, 10 and 20 time periods. Each instance has two cases: one where a network already exists, and one where it must be created from scratch. Table 3.10 shows the number of nodes, links and time periods of each instance, which can be used to calculate the runtime of each instance.

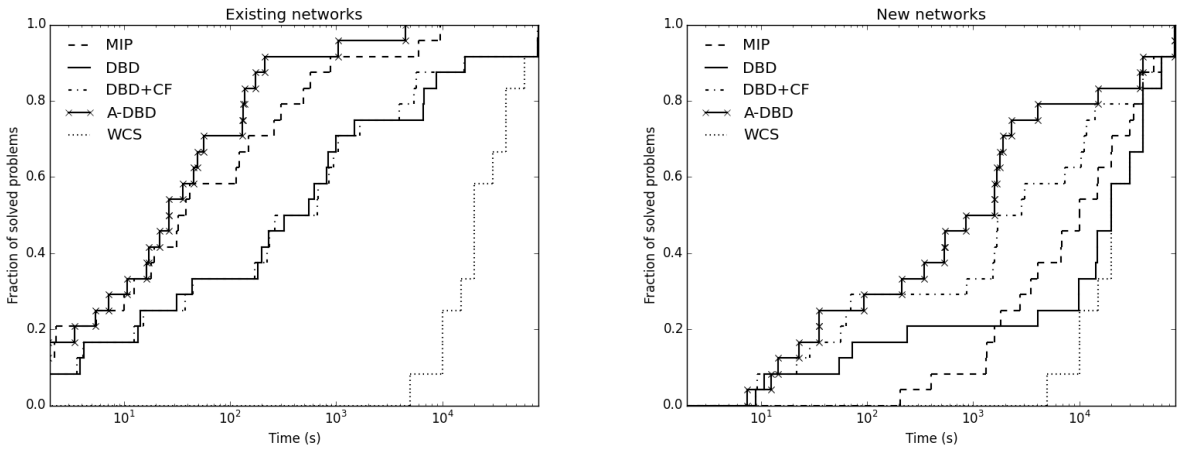
We start with the mixed-integer programming (MIP) implementation with no improvements. We then compare it to disaggregated Benders decomposition, as well as comparing the addition of the following implementation details: combinatorial feasibility cuts, analytic cuts for the warm start and main phases and budget-cover cuts. We also compare our initial results to those found by Ghaderi and Jabalameli (2013). The tests in their study were computed on a machine with dual quad-core 2.66GHz Intel Xeon X5550 processors with 32GB of RAM running Python 2.6 and CPLEX 12.1. While the hardware is similar, the software versions are quite different, and the performance difference over four years can be more than an order of magnitude. This is seen by comparing the time to solution of CPLEX and our MIP.

Tables 3.11 and 3.12 contain details about the results of different implementations for the existing and new network instances respectively. Shown are the optimal objective values for each instance solved to optimality (or the best objective and best bound for the two new network instances not solved to optimality), and the solving times using a number of different implementations. MIP is the standard MIP model when given to Gurobi without any decomposition. DBD is a standard implementation of

Table 3.10: Problem sizes for Ghaderi and Jabalameli instances [83]

Inst.	N	L	T	Inst.	N	L	T	Inst.	N	L	T
TP1	20	46	5	TP10	40	162	5	TP19	80	171	5
TP2	20	46	10	TP11	40	162	10	TP20	80	171	10
TP3	20	46	20	TP12	40	162	20	TP21	80	171	20
TP4	20	61	5	TP13	60	180	5	TP22	80	280	5
TP5	20	61	10	TP14	60	180	10	TP23	80	280	10
TP6	20	61	20	TP15	60	180	20	TP24	80	280	20
TP7	40	137	5	TP16	60	205	5	TP25	56	200	5
TP8	40	137	10	TP17	60	205	10	TP26	56	200	10
TP9	40	137	20	TP18	60	205	20	TP27	56	200	20

Figure 3.9: Comparison of different implementations on existing and new instances. Each curve shows the fraction of problems solved to optimality by the represented implementation within a given amount of time. Note the horizontal axis is plotted on a logarithmic scale



disaggregated Benders decomposition using standard feasibility cuts and a warm start. Accelerated DBD (A-DBD) is disaggregated Benders decomposition with combinatorial feasibility cuts used in the warm start and main phase, plus analytic Benders cuts in the warm start. For the majority of cases, A-DBD is better than the straightforward MIP, especially on larger networks and in the new network instances. There is only a minor benefit to A-DBD over MIP for the existing networks, however on the new network cases Accelerated DBD is a clear winner, with DBD performing worse than the MIP on medium to large instances.

We have not shown results for Benders decomposition without a warm start, because it performs terribly. For this problem, the initial LP-bound is very weak, because one only has to open enough arcs and facilities to satisfy the constraints described in Section 3.7.3. The main phase then takes an incredible amount of time to converge to the optimal solution, if at all. With the addition of the warm start, often the initial bound can be tightened to within 10% of the optimal objective value, and in some of the best cases for A-DBD, the initial bound is less than 3% from the optimal solution. The warm start makes Benders decomposition competitive for this problem.

Figure 3.9 is a graphical representation of the effectiveness of each implementation. Each curve represents a particular implementation, and shows the fraction of all instances in a given class which

Table 3.11: Comparison of runtimes and objective values between the CPLEX implementation in Ghaderi and Jabalameli (2013) and our MIP, standard DBD and accelerated DBD on existing network instances. Runs which reached the time limit are reported as TIME (gap%). If no valid solution was found, they are reported as TIME (NS)

Instance	Objective	Ghaderi et. al. (2013) CPLEX				MIP time (s)	DBD time (s)	Accelerated DBD time (s)
		Objective	Lower Bound	Gap (%)	Time (s)			
TP1E	14924.87	14924.87	14924.87	0.00	4	0.13	1.54	0.86
TP2E	29698.23	29698.23	29698.23	0.00	14	0.57	4.2	1.74
TP3E	67676.54	67676.54	67676.54	0.00	81	5.50	14.18	5.35
TP4E	31419.98	31419.98	31419.98	0.00	5	0.27	1.04	0.63
TP5E	67884.16	67884.16	67884.16	0.00	17	2.21	3.88	1.94
TP6E	158091.31	158091.31	158091.31	0.00	77	9.97	13.58	3.42
TP7E	21159.56	21159.56	21159.56	0.00	63	2.28	31.64	7.16
TP8E	45581.23	45581.23	45581.23	0.00	1758	31.49	183.44	16.28
TP9E	111643.23	111643.23	110830.16	0.73	40000	121.57	555.29	36.00
TP10E	19232.89	19232.89	19232.89	0.00	289	19.21	44.1	17.25
TP11E	40310.81	40310.81	39730.11	1.44	20000	148.23	320.85	130.2
TP12E	99032.15	99103.48	96797.34	2.33	40000	493.7	806.9	133.77
TP13E	25110.09	25110.09	25110.09	0.00	347	12.57	200.31	10.71
TP14E	56892.74	56892.74	56892.74	0.00	1479	41.58	627.16	21.76
TP15E	151049.50	151115.62	149518.50	1.06	60000	889.79	16383.77	137.64
TP16E	26973.29	26973.29	26973.29	0.00	480	17.99	231.77	26.64
TP17E	58636.16	58636.16	58636.16	0.00	16514	113.78	1500.34	56.31
TP18E	143986.33	144200.68	142289.95	1.33	60000	569.65	8777.48	174.64
TP19E	33696.53	33696.53	33696.53	0.00	5622	32.24	845.68	49.04
TP20E	73969.28	74677.35	73256.40	1.90	40000	302.79	6637.88	214.32
TP21E	196674.19	198074.52	192701.54	2.71	80000	9520.11	TIME (2277%)	4528.19
TP22E	30999.44	30999.44	30999.44	0.00	1179	38.5	988.9	26.63
TP23E	70070.24	70092.80	69623.72	0.67	40000	258.14	6744.9	45.77
TP24E	182700.42	183511.97	179939.71	1.95	80000	5959.99	TIME (NS)	1037.89

Table 3.12: Comparison of runtimes and optimality gaps between the CPLEX and two heuristic implementations in Ghaderi and Jabalameli (2013) and our MIP, standard DBD and accelerated DBD on new network instances. Runs which reached the time limit are reported as TIME (gap%). If no valid solution was found, they are reported as TIME (NS)

Instance	Best Objective (Best Bound)	Ghaderi et. al. (2013) Times (s)			MIP Time (s)	DBD Time (s)	Accelerated DBD Time (s)
		CPLEX	Heuristic	Hybrid SA			
TP1N	20473.40	TIME (7.02%)	427 (0.00%)	427 (0.00%)	204.40	8.96	7.38
TP2N	35528.03	TIME (NS)	371 (0.59%)	397 (0.00%)	400.20	TIME (177%)	14.51
TP3N	70973.67	TIME (NS)	604 (1.10%)	645 (0.00%)	4097.81	72.59	35.57
TP4N	35614.49	TIME (10.66%)	1289 (2.23%)	1455 (0.00%)	1348.68	10.64	12.47
TP5N	69150.51	TIME (18.14%)	1208 (1.50%)	1534 (0.62%)	6727.01	54.36	23.02
TP6N	151120.93	TIME (NS)	1869 (0.63%)	4497 (0.28%)	TIME (6.52%)	238.14	35.07
TP7N	23120.55	TIME (9.06%)	519 (0.00%)	519 (0.00%)	1318.22	TIME (23%)	93.10
TP8N	46256.15	TIME (NS)	2296 (0.00%)	2296 (0.00%)	14868.28	TIME (924%)	539.14
TP9N	105113.30	TIME (NS)	3238 (0.07%)	6501 (0.05%)	TIME (0.48%)	TIME (NS)	1581.69
TP10N	19504.02	TIME (NS)	396 (4.63%)	2834 (2.31%)	1595.60	4083.18	211.20
TP11N	37316.52	TIME (8.79%)	1822 (4.03%)	2605 (0.26%)	2752.00	14334.68	858.21
TP12N	89331.29	TIME (2.68%)	2521 (3.16%)	9828 (1.17%)	6877.10	TIME (2.17%)	1675.91
TP13N	28827.77	TIME (9.42%)	7428 (0.14%)	1758 (0.39%)	1830.53	TIME (14.09%)	1917.19
TP14N	59320.43	TIME (NS)	14627 (2.34%)	26262 (1.28%)	10065.93	TIME (920%)	1579.44
TP15N	145023.30	TIME (NS)	TIME (3.31%)	15819 (1.28%)	50500.60	TIME (NS)	37363.24
TP16N	27741.57	TIME (NS)	7782 (2.55%)	8252 (0.55%)	TIME (0.21%)	TIME (1291%)	547.45
TP17N	55740.52	TIME (NS)	17136 (2.05%)	21361 (1.19%)	TIME (1.47%)	TIME (NS)	1803.15
TP18N	132520.16	TIME (6.25%)	TIME (2.29%)	13660 (1.12%)	20387.58	TIME (1203%)	4099.27
TP19N	37562.93	TIME (19.22%)	10805 (0.71%)	19255 (0.04%)	3507.57	TIME (2554%)	2286.62
TP20N	76063.44	TIME (NS)	TIME (0.22%)	15477 (0.37%)	TIME (0.67%)	TIME (NS)	15120.47
TP21N	183881.93 (183166.26)	TIME (NS)	TIME(2.15%)	24659 (1.16%)	TIME (1.43%)	TIME (2277%)	TIME (0.39%)
TP22N	33258.34	TIME (6.58%)	8332 (2.06%)	12874 (0.73%)	10071.79	TIME (2251%)	346.35
TP23N	70208.93	TIME (7.84%)	TIME (2.09%)	31868 (1.25%)	32412.29	TIME (2351%)	TIME (0.28%)
TP24N	171478.00 (170805.92)	TIME (NS)	TIME (2.48%)	60596 (1.65%)	TIME (0.39%)	TIME (NS)	TIME (2.84%)

	Implementation	Warm Start		Main Phase		
		Feasibility	Optimality	Feasibility	Optimality	Solved
Existing	DBD	1.30	15853.71	0.97	124.62	22
	DBD+CF	7.90	16062.98	0.30	121.95	22
	A-DBD	10.22	1736.02	0.34	136.36	24
New	DBD	140.11	551.71	308.20	24266.60	7
	DBD+CF	342.38	12237.88	0.47	372.89	19
	A-DBD	362.76	6203.81	0.21	423.57	21

Table 3.13: Comparison of number of cuts added in each phase and number of instances solved to optimality for three implementations. Results are split into existing network and new network instances. Number of cuts added are reported as shifted geometric means with a shift parameter of 1

solved to optimality in less time than indicated on the horizontal axis. The implementations MIP, DBD and A-DBD are as mentioned above. DBD+CF uses combinatorial feasibility cuts, but no analytic Benders cuts in the warm start, and WCS is the worst case scenario, which represents the time limits of all instances. In both plots, A-DBD is better than all other implementations, being able to solve all existing network instances to optimality in less time than any other method. Note that MIP is actually better than DBD and DBD+CF, which highlights the importance of using analytically calculated dual variables for this problem.

The main reason for the difficulty of the new network cases is the feasibility of the networks. In the existing network case, the networks are either already feasible or can be made feasible very easily, whereas for the new network case a brand new feasible network must be made from scratch. Standard Benders feasibility cuts are much less powerful in this problem, as can be seen in the new networks case. DBD fails to solve many instances to optimality in the given time and is better than the MIP in only 20% of the instances, however DBD+CF does better than the MIP for all instances.

Analytic Benders cuts in the warm start provide much benefit, which can be seen from comparing the DBD+CF and A-DBD implementations. We should, however, make a note that this depends upon the quality of the dual variables chosen by the solver. For these results, the dual variables chosen by Gurobi produced very weak Benders cuts, which was evidenced by the number of cuts generated in the warm start. Table 3.13 shows that the number of optimality cuts generated by DBD+CF in the warm start is significantly higher than the number generated by A-DBD. On other machines with different hardware and different versions of Gurobi, the cuts generated were stronger, and thus the improvement of A-DBD over DBD+CF was smaller. It was, however, always a positive difference.

In order to compare the usefulness of the budget cover cuts, we must turn off the cuts that Gurobi adds itself. Figure 3.10 shows a similar comparison to Figure 3.9. The new implementations NoCuts and NoCuts+Budg are the same as DBD, but with the solver-added cuts turned off, and NoCuts+Budg adds budget cover cuts during the warm start. In all cases, the budget cover cuts give a marginal benefit, however the loss of the solver-added cuts is also evident. In short, the only time one should have to add budget cover cuts is when they are not using a powerful, modern solver such as Gurobi or CPLEX, both of which are capable of finding such cuts without user input.

Our next comparison is the use of analytic cuts in the main phase of Benders decomposition, *i.e.* at

Figure 3.10: Comparison of budget cover cuts on existing and new instances. Each curve shows the fraction of problems solved to optimality by the represented implementation within a given amount of time. Note the horizontal axis is plotted on a logarithmic scale

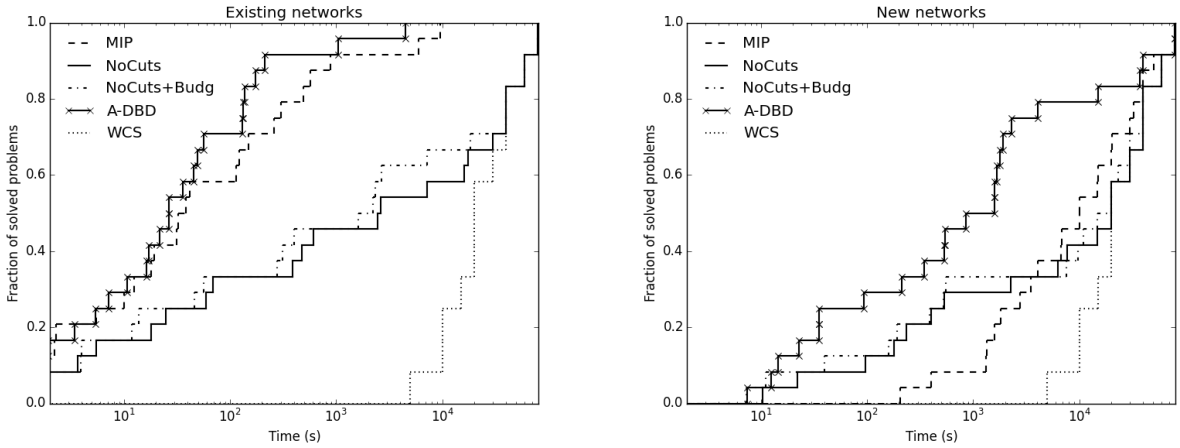
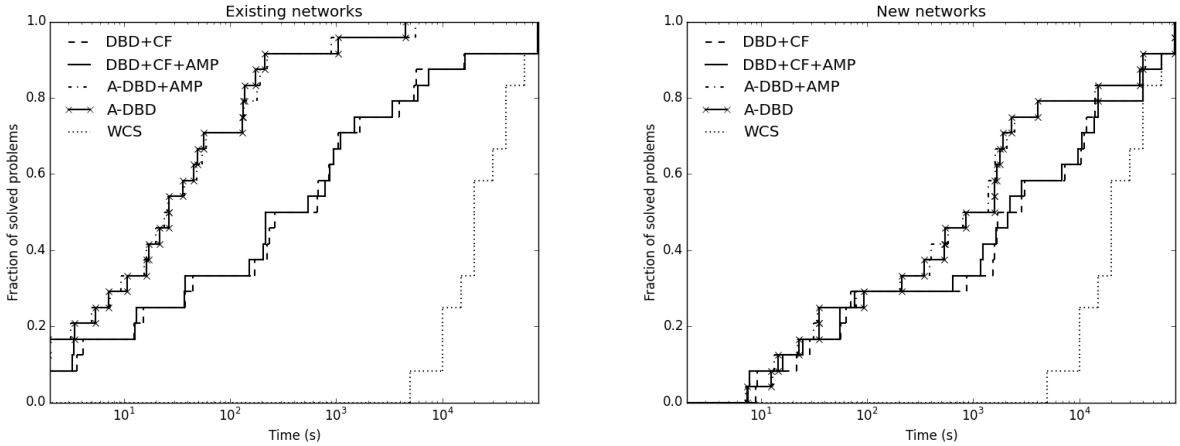


Figure 3.11: Comparison of analytic cuts for warm starts and main phase on existing and new instances. Each curve shows the fraction of problems solved to optimality by the represented implementation within a given amount of time. Note the horizontal axis is plotted on a logarithmic scale

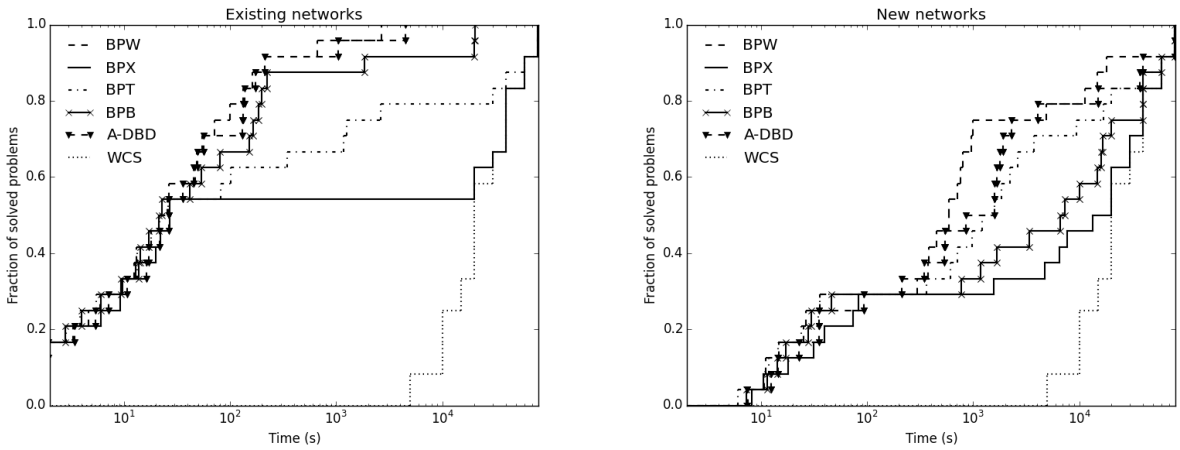


each potential incumbent solution from the branch-and-bound tree. Figure 3.11 compares the DBD+CF and A-DBD implementations with and without analytic benders cuts in the main phase (+AMP). In all cases, choosing dual variables analytically provides only marginal benefit. This can be attributed to Gurobi picking very strong dual variables in the main phase, unlike the warm start. Importantly, using analytic duals does not cause any implementations to perform worse, and there is little effort involved in implementing them.

Other technical considerations are branching priorities and direction. When searching the branch-and-bound tree, one must decide which variables to branch on first, and which direction should be explored first. Modern solvers have the ability to make good decisions about branching, however in some cases the user may have certain insight into the problem which the solver can not see.

In the DUFLNDP, the location of the facilities is the main structural decision, since there can be multiple paths to each facility, and opening or closing a particular arc has less impact than opening or closing a facility. This can be seen in Figure 3.12, which compares the A-DBD with different

Figure 3.12: Comparison of different branching priorities for the A-DBD implementation on existing and new instances. Each curve shows the fraction of problems solved to optimality by the represented implementation within a given amount of time. Note the horizontal axis is plotted on a logarithmic scale



branching priorities. BPW and BPX branch on the facilities and arcs first respectively. BPT prioritises the facilities and arcs in the first time period, and BPB first branches on the construction variables, U and V.

Branching priority does not make much difference to the smaller instances, as there are often many fewer nodes to explore. For the larger instances, only BPW performs better than A-DBD, that is, branching on facilities first is almost always better than not setting any priorities. BPT is more effective in the new networks case, as setting a variable in the first time period fixes the variables corresponding to the same arc or facility for all subsequent time periods. Branching on the arcs first is by far the worst option, failing to solve many of the existing network instances and under-performing all other implementations on the new networks.

Branching direction is best left to the solver. Gurobi and CPLEX have very advanced methods for determining whether to branch up or down first for each node, and choosing to always branch in a particular direction is often less effective. This is the case for the DUFLNDP, regardless of whether or not branching priorities have been set.

Disaggregation level

In this problem it is possible to disaggregate over two different sets: the source of each demand and each separate time period. We show here that it is best to disaggregate by both sets at the same time. Disaggregation of sub-problems, and thus Benders cuts, always results in tighter bounds. These tighter bounds allow the master problem to be solved more quickly. The trade-off is that having more sub-problems can take longer to solve, particularly if there are overheads associated with those sub-problems. In this problem, the most sub-problems we solve are 1600, which is acceptable considering the speed increases we obtain from this. In other problems, the number of sub-problems may enter the hundreds of thousands, at which point even the smallest overheads will start to add up.

Table 3.14: Comparison of solution times for problem TP9E with different levels of disaggregation

Disaggregation level	# S.P.s solved	S.P. cumulative time (s)	Master solve time (s)
Time only	580	24.74	283.35
Node only	760	14.72	232.77
Node and Time	11200	9.48	101.47

A specific example is data set TP9E, which we can compare results for if we disaggregate only by nodes, only by time and by both nodes and time. For this instance there are 40 nodes and 20 time periods. Table 3.14 shows the number of sub-problems (S.P.s) solved, the total time spent solving sub-problems and the total time spent solving the entire problem for the different levels of disaggregation.

We can see that disaggregating more leads to smaller sub-problems which solve significantly faster. The average solve time for each sub-problem is 43ms, 19ms and 0.85ms for time only, nodes only and both, respectively. Even though many more sub-problems must be solved when disaggregating by both nodes and time, the cumulative time spent solving them is less, and the tighter cuts provided by disaggregation leads to a faster solve time of the master problem.

3.7.6 Conclusion

Disaggregated Benders decomposition embedded in a branch-and-cut framework is an effective method for solving the DUFLNDP if implemented properly. Adding constraints that enforce feasibility to avoid relying upon Benders feasibility cuts, using combinatorial Benders feasibility cuts, and using a warm start are good ways of improving the effectiveness of the solver. Analytically derived Pareto-optimal Benders cuts can also be beneficial in some cases. For this particular problem, it is the disaggregation of the sub-problems and combinatorial Benders feasibility cuts which provide the most impressive speed increase, which has allowed us to solve almost all instances to optimality within the time limits. In the future, we would like to generalise this approach to a wide range of network design and facility location problems where similar techniques are beneficial.

3.7.7 Acknowledgements

The authors thank two anonymous referees for their constructive feedback which helped to improve the paper.

3.8 Generalised UFL and Network Design Problem

There are a number of problems with a structure similar to those of the UFL and DUFLNDP problems. We call the most general formulation the Generalised Uncapacitated Facility Location and Network Design Problem (GUFLNDP). This problem aims to minimise the running and/or construction costs of a network such that a number of requests for transportation may be served. The network is composed of nodes and arcs, and each node may have the capacity to support a facility. Each request has a specific origin node and a set of potential destination facilities. The problem may also cover multiple time periods, where facilities and arcs may be constructed or removed between time periods for a cost.

Any problem that satisfies the following conditions is a candidate GUFLNDP problem:

- Has a demand routing component where demands have specific origins and amounts
- No shared capacity between demands
- Demand flows are instantaneous (no storage between time periods)
- No constraints directly involving the routing variables other than the standard constraints presented below

If these conditions are met, the problem will fit the GUFLNDP framework. The main commonality among problems in the GUFLNDP class is that they are good candidates for disaggregated Benders decomposition. Even further, they all have identical Benders sub-problems, which can be solved the same way, independent of the master problem. The master problem supports a rich variety of constraints that control the network structure, but in the end the sub-problems are to move the requests through the generated network, and this can be solved using problem-independent methods.

There are a number of problems that are in the GUFLNDP class. A non-exhaustive list of these problems is:

- Uncapacitated Facility Location problem (UFL) [46, 60]
- Network Design problem [47]
- Dynamic UFL and Network Design problem (DUFLNDP) [2, 83]
- Two-Level UFL problem with Single-Assignment [85]
- Urban Rapid Transit Network Design problem [86]
- Hub and Shuttle Public Transit System Design problem [67]
- Tree of Hubs Location problem [52]
- Hub Line Location problem [53]
- Gateway hub location problem [87]

- Integrated Urban Heirarchy and Transportation Network Planning [88]

- TSP with Generalised Latency [54]

Each of these problems is essentially a network design problem with a multi-commodity flow component. A number of these studies show that Benders decomposition is useful for solving these problems [2, 46, 47, 52–54, 60, 67, 86, 87], particularly when the sub-problems are disaggregated [2, 52–54, 60, 67, 86] and when Benders decomposition is implemented in a branch-and-cut framework [2, 52, 53, 60]. Here, we present the formulation of the GUFLNDP, show how to apply Benders decomposition to it and discuss some of the common improvements.

3.8.1 Formulation of the GUFLNDP

The GUFLNDP concerns a network of nodes N and arcs A. There is a subset of nodes $F \subseteq N$ that are potential facilities for serving demands. There are potential costs associated with opening facilities and arcs, and with keeping them open. There is a set of requests R, all of which must be met. Each request has an origin α_r and a set of potential destination facilities Ω_r where it may be served. The demand of these requests must be routed through the network on open arcs from the origin to an open facility that can serve it.

There is a cost associated with routing demand through the network, but not with serving it. There may also be multiple time periods across which the network is to be controlled, as long as the serving of demands occurs instantaneously. A problem with multiple time periods is called dynamic, while a problem without is called static. The objective function is quite flexible, but it is always to minimise some linear cost associated with the network. The most common is to minimise the running cost of the network, but one could also find the minimum-cost network that satisfies all demands regardless of running cost. Any balance of the construction and running costs can be accommodated.

For neatness and without loss of generality, the constraints in our formulation assume that $F = N$ and $A = N \times N$, so the summations do not require conditions. For the arcs, it is also possible that there will be multiple copies of each arc, so that $A \subseteq N \times N \times P$ for some set P. This is a useful modelling technique for some problems, such as the Tree of Hubs Location problem [52], where arcs can be regular arcs or inter-hub arcs that are cheaper to use but can only exist in specific circumstances. Finally, we assume all costs are non-negative. We now present the formulation of the GUFLNDP:

Sets

N	Set of nodes
F	Set of potential facilities. $F \subseteq N$
A	Set of arcs
R	Set of requests
T	Set of time periods

Parameters

g^o_{it} Cost of opening facility at node $i \in F$ at time $t \in T$

g^c_{it} Cost of closing facility at node $i \in F$ at time $t \in T$

c^o_{ijt} Cost of opening arc $(i, j) \in A$ at time $t \in T$

c^c_{ijt} Cost of closing arc $(i, j) \in A$ at time $t \in T$

ρ_{ijt} Cost per unit for routing demand along arc $(i, j) \in A$ at time $t \in T$

f_{it} Operating cost of open facility $i \in F$ at time $t \in T$

h_{ijt} Operating cost of arc $(i, j) \in A$ at time $t \in T$

α_r Origin node of request $r \in R$

Ω_r Set of destination facilities for request $r \in R$

ω_{ir} 1 if $i \in \Omega_r$ (facility $i \in F$ can serve request $r \in R$), 0 otherwise

d_{rt} Weight of demand $r \in R$ at time $t \in T$

Variables

x_{it} 1 if facility $i \in F$ is open at time $t \in T$, 0 otherwise

y_{ijt} 1 if arc $(i, j) \in A$ is open at time $t \in T$, 0 otherwise

z_{ijrt} Fraction of request $r \in R$ that travels along arc $(i, j) \in A$ at time $t \in T$

u^o_{it} 1 if facility $i \in N$ is opened at time $t \in T$

u^c_{it} 1 if facility $i \in N$ is closed at time $t \in T$

v^o_{ijt} 1 if arc $(i, j) \in A$ is opened at time $t \in T$

v^c_{ijt} 1 if arc $(i, j) \in A$ is closed at time $t \in T$

Objective

$$\min \sum_{t\in T}\left\{\sum_{i\in F} f_{it}x_{it} + \sum_{(i,j)\in A}\left(h_{ijt}y_{ijt} + \sum_{r\in R} d_{rt}\rho_{ijt}z_{ijrt}\right) + \sum_{i\in F}(g^o_{it}u^o_{it} + g^c_{it}u^c_{it}) + \sum_{(i,j)\in A}\left(c^o_{ijt}v^o_{ijt} + c^c_{ijt}v^c_{ijt}\right)\right\} \quad (3.94)$$

Constraints

Request constraints

$$\omega_{\alpha_r r}x_{\alpha_r t} + \sum_{j\in N} z_{\alpha_r jrt} \geq 1 \qquad \forall r \in R, \forall t \in T \quad (3.95)$$

$$\sum_{j\in N} z_{jirt} \leq \sum_{j\in N} z_{ijrt} + \omega_{ir}x_{it} \qquad \forall i \in N \setminus \{\alpha_r\}, \forall r \in R, \forall t \in T \quad (3.96)$$

$$z_{j\alpha_r rt} = 0 \qquad \forall j \in N, (j, \alpha_r) \in A, \forall r \in R, \forall t \in T \quad (3.97)$$

$$z_{ijrt} \leq y_{ijt} \qquad \forall (i, j) \in A, \forall r \in R, \forall t \in T \quad (3.98)$$

Construction and deconstruction constraints

$$x_{i(t-1)} + u^o_{it} - u^c_{it} = x_{it} \qquad \forall i \in F, \forall t \in T \quad (3.99)$$

$$y_{ij(t-1)} + v^o_{ijt} - v^c_{ijt} = y_{ijt} \qquad \forall (i, j) \in A, \forall t \in T \quad (3.100)$$

Variable constraints

$$x_{it} \in \{0,1\}, u^o_{it} \in \{0,1\}, u^c_{it} \in \{0,1\} \qquad \forall i \in N, \forall t \in T \tag{3.101}$$

$$y_{ijt} \in \{0,1\}, v^o_{ijt} \in \{0,1\}, v^c_{ijt} \in \{0,1\} \qquad \forall (i,j) \in A, \forall t \in T \tag{3.102}$$

$$z_{ijrt} \geq 0 \qquad \forall (i,j) \in A, \forall r \in R, \forall t \in T \tag{3.103}$$

The objective (3.94) is the accumulation of all costs: construction, deconstruction, operating and routing. The request constraints govern how the requests can be served. Constraints (3.95) say that, for each request r, either it can be served at the origin node α_r if it is a viable facility and it is open, or the demand must leave α_r. Constraints (3.96) are the flow consistency constraints on the nodes, i.e. for all the demand that flows into a node, it must be served at that node or moved elsewhere. Constraints (3.97) prevent demand from returning to the origin node. Constraints (3.98) ensure demand cannot flow along arcs unless they are open. Finally, the construction and deconstruction constraints (3.99-3.100) link the relevant variables with the opening and closing of facilities and arcs.

The request constraints are the minimal set of constraints required for the GUFLNDP that must be present in all problems. They are also the only constraints in which the z_{ijrt} variables are allowed to appear. If multiple time periods are included, the construction and deconstruction constraints will also be required, since otherwise the time periods would be completely independent and could be solved as such. If deconstruction is not permitted as in the case of the DUFLNDP, one simply removes the relevant variables from the formulation.

3.8.2 Benders decomposition for the GUFLNDP

As highlighted earlier, many problems that fit the framework of the GUFLNDP are prime candidates for Benders decomposition. Here, we present the Benders formulation for the general problem and describe how this changes for more specific problems.

Benders master problem

The Benders master problem contains all constraints from the original formulation that do not include the continuous z_{ijrt} variables. For the GUFLNDP, this is only the construction and deconstruction constraints. For all other problems, any additional constraints pertaining to the design of the network remain in the master problem. The routing costs for each request $r \in R$ at each time $t \in T$ are now approximated by the new variables θ_{rt}, and these approximations are updated by adding Benders optimality cuts.

A usual problem with Benders decomposition is that the relaxation of the master problem allows solutions that make the sub-problems infeasible. In this case, it is because there are no constraints ensuring enough facilities and arcs are open to allow the requests to be served. Similar to the DUFLNDP, we handle this by adding some feasibility constraints that will place minimum requirements on the network. These may still not be enough to ensure the sub-problems are feasible, often due to the

occurrence of cycles in the network, at which point we add Benders feasibility cuts during the branch-and-bound process. The Benders master problem for the GUFLNDP is:

$$\min \quad \sum_{t\in T}\left\{\sum_{i\in F} f_{it}x_{it} + \sum_{(i,j)\in A} h_{ijt}y_{ijt} + \sum_{r\in R} d_{rt}\theta_{rt} + \sum_{i\in F}(g^o_{it}u^o_{it} + g^c_{it}u^c_{it}) + \sum_{(i,j)\in A}\left(c^o_{ijt}v_{ijt} + c^c_{ijt}v^c_{ijt}\right)\right\} \tag{3.104}$$

Subject to:

Feasibility constraints

$$\sum_{i\in\Omega_r} x_{it} \geq 1 \qquad \forall r\in R, \forall t\in T \tag{3.105}$$

$$\omega_{\alpha_r r}x_{\alpha_r t} + \sum_{\substack{j\in N\\(\alpha_r,j)\in A}} y_{\alpha_r jt} \geq 1 \qquad \forall r\in R, \forall t\in T \tag{3.106}$$

Construction and deconstruction constraints

$$x_{i(t-1)} + u^o_{it} - u^c_{it} = x_{it} \qquad \forall i\in F, \forall t\in T \tag{3.107}$$

$$y_{ij(t-1)} + v^o_{ijt} - v^c_{ijt} = y_{ijt} \qquad \forall (i,j)\in A, \forall t\in T \tag{3.108}$$

Variable constraints

$$x_{it}\in\{0,1\}, u^o_{it}\in\{0,1\}, u^c_{it}\in\{0,1\} \qquad \forall i\in N, \forall t\in T \tag{3.109}$$

$$y_{ijt}\in\{0,1\}, v^o_{ijt}\in\{0,1\}, v^c_{ijt}\in\{0,1\} \qquad \forall (i,j)\in A, \forall t\in T \tag{3.110}$$

The only change to the objective function is that the contribution from the routing costs is replaced by approximation variables θ_{rt}. The request constraints are removed as they are now part of the Benders sub-problems, and the construction and deconstruction constraints are unchanged. There are also two additional constraints to help ensure feasibility of the sub-problems. Constraints (3.105) say that for every request, at least one valid facility must be open, and constraints (3.106) ensure that for each demand $r\in R$, either it can be served at α_r, or there exists an arc for it to leave α_r. These constraints eliminate many feasible solutions to the relaxed master problem that would lead to infeasible sub-problems, but not all of them. The remaining infeasible master problem solutions are handled using Benders feasibility cuts in the sub-problems.

Benders sub-problems

For the GUFLNDP, we have one sub-problem for each request and each time period, and their corresponding master problem variables are θ_{rt}. These problems are solved for a particular solution to

the master problem, so here the variables x^* and y^* are fixed. For each $r \in R$ and $t \in T$, we have the following Benders sub-problem:

$$\min \quad \sum_{(i,j)\in A} \rho_{ijt} z_{ijrt} \tag{3.111}$$

Subject to:

Request constraints

$$\omega_{\alpha_r r} x^*_{\alpha_r t} + \sum_{j\in N} z_{\alpha_r jrt} \geq 1 \tag{3.112}$$

$$\sum_{j\in N} z_{jirt} \leq \sum_{j\in N} z_{ijrt} + \omega_{ir} x^*_{it} \qquad \forall i \in N \setminus \{\alpha_r\} \tag{3.113}$$

$$z_{j\alpha_r rt} = 0 \qquad \forall j \in N \setminus \{\alpha_r\}, (j, \alpha_r) \in A \tag{3.114}$$

$$z_{ijrt} \leq y^*_{ijt} \qquad \forall (i,j) \in A \tag{3.115}$$

$$z_{ijrt} \geq 0 \qquad \forall (i,j) \in A \tag{3.116}$$

Each sub-problem is a linear program, and as such a solution to its dual program can be used to construct a Benders optimality cut to inform the master problem how the θ_{rt} variables will change with the master problem variables x and y. Given a solution to the master problem, (x^*,y^*), the dual problem for the sub-problem (r,t) is:

$$\max \quad \gamma_{\alpha_r} - \sum_{i\in N} \gamma_i \omega_{ir} x^*_{it} - \sum_{(i,j)\in A} \lambda_{ij} y^*_{ijt} \tag{3.117}$$

Subject to:

$$\rho_{ijt} + \lambda_{ij} + \gamma_j - \gamma_i \geq 0 \qquad \forall (i,j) \in A, j \neq \alpha_r \tag{3.118}$$

$$\gamma_i \geq 0 \quad \forall i \in N, \quad \lambda_{ij} \geq 0 \quad \forall (i,j) \in A \tag{3.119}$$

The γ_i are the dual variables associated with constraints (3.112,3.113), and λ_{ij} with (3.115). The dual variables associated with (3.114) are unbounded, do not appear in the objective function, and appear only in the dual constraints when $j = \alpha_r$; that is, the arcs that flow back to the origin. In these cases, the dual variable can be set to a sufficiently large number, thus making the constraints feasible without changing the objective value. As such, we need only consider arcs that do not flow back to the request origin.

It is possible for a request to have no path from its origin to any of its potential destinations. This occurs when there is a disconnected segment of the network and there are no valid facilities at any of the nodes in the segment. To handle this, we add Benders feasibility cuts to prevent the master problem from generating such networks again.

We use combinatorial Benders feasibility cuts, where we find an Irreducible Inconsistent Subsystem (IIS), which returns a set of capacity constraints on the arcs and facilities that are preventing feasibility

of the sub-problems. We then add the feasibility constraint that at least one such arc or facility must be open. This is similar to the IIS feasibility cuts used for the DUFLNDP in Section 3.7.3; however, if deconstruction is allowed, these must be applied to specific time periods where infeasibility occurs rather than only the first time period.

Pareto-optimal Benders cuts for the GUFLNDP

It is possible to construct Pareto-optimal Benders cuts for all of the problems in the GUFLNDP class using the same techniques as the UFL and DUFLNDP problems. The only difficulty is that in the case of the DUFLNDP, we assume it is possible to open a facility at the origin of each demand, which allows us to set the dual variables associated with the arcs and facility of the current solution to 0. It is easy to prove Pareto-optimality, since the Benders cut will be tight in the scenario where the facility at the origin is opened and all arcs on the current solution are closed. For GUFLNDP problems, this assumption may not hold and the dual variables for parts of the current sub-problem solution may actually take non-zero values.

When designing an algorithm for producing Pareto-optimal Benders cuts for a GUFLNDP problem, one must consider which alternative network configurations are possible — with respect to the master problem constraints — and whether or not their Benders cut will give the correct objective value for those configurations. If there exists at least one alternative configuration that does not use the facility or an arc from the current solution, then the Benders cut must hold tightly for at least one such configuration. In the ideal case, the shortest-possible path would be arc-distinct from the path used in the current sub-problem solution, allowing all dual variables on the current solution to be 0. However, if this is not the case, then a longer path must be used and the dual variables for some arcs may be non-zero to account for this.

The solution for each sub-problem is the shortest path through the network defined by the fixed variables x^* and y^* from the origin to the nearest open facility. To analytically construct a Benders cut, we require information about the distances between nodes in the network. For simplicity, we will drop the time indices t, which does not matter since each sub-problem is independent of sub-problems at other times. For the rest of this section, a will represent (i, j) as a shorthand for an arc.

First, let $D^c_{ij}(\boldsymbol{\lambda})$ be the lambda-weighted length of the shortest path from i to j across open arcs with arc lengths adjusted by their corresponding λ value. Shortest path distance properties apply to these values, i.e. for any open arc, $D^c_{kj}(\boldsymbol{\lambda}) \leq D^c_{ki}(\boldsymbol{\lambda}) + \rho_{ij} + \lambda_{ij}$. Second, we require the quantity $D^o_i(\boldsymbol{\lambda}) = \min_{j,P} \left\{ \rho_{ij} + \sum_{a \in P} (\rho_a + \lambda_a) + \gamma_{p_n} | (i,j) \in A; P \text{ is a path from } j \text{ to a valid facility } p_n \right\}$, or 0 if i is a valid facility. If i is not a valid facility, then $D^o_i(\boldsymbol{\lambda})$ is the length of the shortest possible path from i to a valid facility where the arc-weights beyond the first arc have been increased by the value of the relevant dual variables. By shortest path properties, we also have for any arc $D^o_i(\boldsymbol{\lambda}) \leq D^o_j(\boldsymbol{\lambda}) + \rho_{ij} + \lambda_{jk^*}$, where k^* is the next node on the shortest λ-weighted path from j to a valid facility.

We also define a function $P^*(A', \boldsymbol{\rho}', o, \boldsymbol{d})$ which returns a triple: (l, k, p). p is the shortest path through the network using only arcs in A' with arc lengths $\boldsymbol{\rho}'$, from node o to a node in $\boldsymbol{d}$. $\boldsymbol{d}$ is the

Algorithm 6 Algorithm for computing dual variables for analytic Benders optimality cut for sub-problem (r,t), assuming the sub-problem is feasible.

1: Begin with master problem solution $x^*_{it}, y^*_{at}\ \forall i \in N, \forall a \in A$
2: Initialise $\boldsymbol{\lambda} = 0, \boldsymbol{\gamma} = 0$
3: $(\gamma_{\alpha_r}, i^*, CS) \leftarrow P^*(\{a \in A | y^*_{at} = 1\}, \boldsymbol{\rho}, \alpha_r, \{i \in \Omega_r | x^*_{it} = 1\})$
4: **for** $i \in \Omega_r \setminus \{\alpha_r\}$ **do**
5: $\quad \gamma_i \leftarrow \max\left(\gamma_{\alpha_r} - D^c_{\alpha_r i}(\boldsymbol{\lambda}), 0\right)$
6: Order $\leftarrow \{(D^o_i(\boldsymbol{\lambda}), -1, i) | i \in N \setminus (\Omega_r \cup \{\alpha_r\})\} \cup \left\{(\max(D^o_i(\boldsymbol{\lambda}), D^o_j(\boldsymbol{\lambda})), i, j) | (i,j) \in A\right\}$
7: Sort Order from smallest to largest, left to right
8: **for** (d, i, j) in Order **do**
9: $\quad$ **if** $i = -1$ **then**
10: $\quad\quad \gamma_j \leftarrow \max(\gamma_{\alpha_r} - D^c_{\alpha_r j}(\boldsymbol{\lambda}), D^o_j(\boldsymbol{\lambda}))$
11: $\quad$ **else**
12: $\quad\quad \lambda_{ij} \leftarrow \max(0, \gamma_i - \gamma_j - \rho_{ij})$
13: $(\bullet, \bullet, SP) \leftarrow P^*(A, \boldsymbol{\rho} + \boldsymbol{\lambda}, \alpha_r, \Omega_r)$
14: $Q \leftarrow CS \cap SP$
15: **for** $a \in Q$ **do**
16: $\quad \Delta_a \leftarrow P^*(A \setminus \{a\}, \boldsymbol{\rho} + \boldsymbol{\lambda}, \alpha_r, \Omega_r)[0] - \gamma_{\alpha_r}$
17: **while** $\max\limits_{a \in Q} \Delta_a > 0$ **do**
18: $\quad a \leftarrow \arg\min\limits_{\substack{a \in Q \\ \Delta_a > 0}} \Delta_a$
19: $\quad \lambda_a \leftarrow \Delta_a$
20: $\quad \gamma_{\alpha_r} \leftarrow \gamma_{\alpha_r} + \Delta_a$
21: $\quad$ Propagate()
22: $\quad$ **for** $a \in Q$ **do**
23: $\quad\quad \Delta_a \leftarrow P^*(A \setminus \{a\}, \boldsymbol{\rho} + \boldsymbol{\lambda}, \alpha_r, \Omega_r)[0] - \gamma_{\alpha_r}$
24: $\Delta_{i^*} \leftarrow P^*(A, \boldsymbol{\rho} + \boldsymbol{\lambda}, \alpha_r, \Omega_r \setminus \{i^*\})[0] - \gamma_{\alpha_r}$
25: **if** $\Delta_{i^*} > 0$ **then**
26: $\quad \gamma_{i^*} \leftarrow \Delta_{i^*}$
27: $\quad \gamma_{\alpha_r} \leftarrow \gamma_{\alpha_r} + \Delta_{i^*}$
28: $\quad$ Propagate()
29: Return $\boldsymbol{\gamma}$, $\boldsymbol{\lambda}$

set of potential destinations, where the γ value of each destination is included in the path length. l is the length of p, and k is the facility at the end of p. $P^*(A', \boldsymbol{\rho}', o, \boldsymbol{d})[0]$ returns only the length l of the shortest path. $D^c_{ij}(\boldsymbol{\lambda})$ and $D^o_i(\boldsymbol{\lambda})$ are special cases of this function which will be used most often. In practice, this will be used to compute shortest paths through the λ-weighted network. Algorithm 6 shows the procedure for computing the Pareto-optimal Benders cut.

In line 3, we find the shortest path across open arcs to an open valid facility, *i.e.* the current solution. This also assigns the initial value of γ_{α_r} and marks the facility i^* used in the current solution. In lines 4 to 5, we compute the γ values for the other valid facilities by taking the difference between the length of the current shortest path and the length of the shortest path from the origin to the new destination. If the facility is open, the γ value will be 0 as it cannot be closer than the current closest facility. If the facility is closed and closer than the current closest facility, its γ value will represent the immediate

Algorithm 7 Propagate(): Algorithm for propagating γ values after an adjustment has been made. This procedure maintains dual feasibility of the dual variables. The function min() minimises tuples from left to right.

```
R ← {(γ_{α_r} − ρ_{α_r i} − λ_{α_r i}, i) | ∀i ∈ Neigh_{α_r}}
while R is not empty do
    (d, i) ← min(R)
    if γ_i < d then
        γ_i ← d
        for j ∈ Neigh_i do
            R ← R ∪ {(γ_i − ρ_ij − λ_ij, j)}
```

saving from opening that facility.

In lines 6 to 12, we compute the γ values for all remaining nodes and λ values for all arcs. Since $\boldsymbol{\lambda}$ depends upon other values of $\boldsymbol{\gamma}$ and $\boldsymbol{\lambda}$, they must be processed in order. To do this, we create a list called Order and fill it with a series of triples, where the first value is the current shortest possible distance to a valid open facility, and the second and third identify which arc or node this value refers to. Starting with the nodes and arcs closest to valid facilities, we compute their values for γ and λ as specified in the algorithm.

Line 13 then computes the shortest-possible $\lambda-$ and $\gamma-$weighted path from the origin to a valid open facility. Lines 14 to 23 cover a special case where the shortest possible path and the current solution share at least one arc. The set Q contains all arcs that are on both the shortest possible path from the origin to a valid facility and the current shortest path across open arcs to an open valid facility. For each of these arcs, we compute the value Δ_a, which is the difference between the length of shortest possible λ- and γ-weighted path from the origin to a valid facility that does not contain the arc a, and the current value of γ_{α_r}. Initially, γ_{α_r} will be the length of the current solution.

For the arc a with the smallest positive value of Δ_a, we set λ_a to Δ_a and increase γ_{α_r} by Δ_a. We may then need to adjust some γ values to reflect the changes in $D^c_{ij}(\boldsymbol{\lambda})$ and $D^o_i(\boldsymbol{\lambda})$ caused by the additional λ value. This is achieved using the Propagate() function on line 21, shown in Algorithm 7. Starting from α_r, we consider all neighbours and check if the relevant dual constraint is satisfied. If not, we change the value of γ corresponding to the node at the end of the arc and check all neighbours of that node also. This will continue until we reach a, where the increased value of λ_a will have satisfied the constraint, or until we reach arcs with slack in their dual constraints.

We then recompute Δ_a for all $a \in Q$ on lines 22 and 23. We pick the arc with the smallest positive value of Δ_a, apply the same changes, and repeat until $\Delta_a \leq 0$ for all arcs in Q. In lines 24 to 28, we perform a similar procedure for the current facility, where we find the shortest possible path to an alternative facility. If this path is longer than the current solution, we add the difference to γ_{i^*} and γ_{α_r}, and then propagate the γ-values again. At this point, the values of $\boldsymbol{\gamma}$ and $\boldsymbol{\lambda}$ constitute a Pareto-optimal Benders cut.

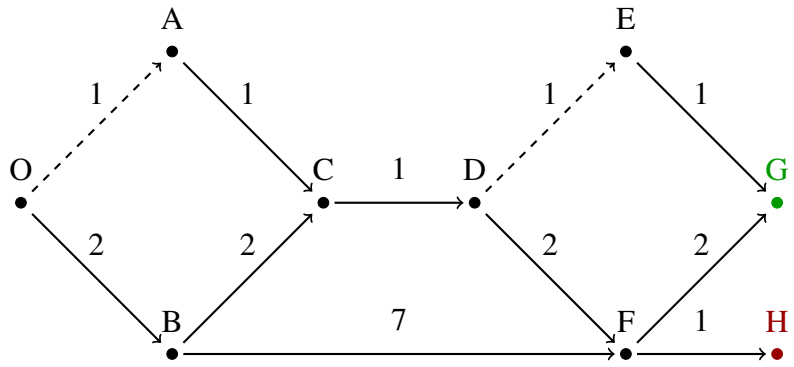


Figure 3.13: Network for GUFLNDP numerical example. O is the origin, G is an open facility and H is a closed facility. Solid arcs are open, dashed arcs are closed. Arcs are labelled with their lengths.

3.8.3 Numerical example

Consider the network in Figure 3.13, where we are solving the sub-problem for a request from O to either G or H, *i.e.* α_r =O and $\Omega_r = \{\text{G,H}\}$. Currently, G is open and H is closed. The current solution is to take the path O $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D $\rightarrow$ F $\rightarrow$ G, and has a length of 9. Lines 3 to 5 of Algorithm 6 assign initial values to γ for each node as shown in the following table:

i	O	A	B	C	D	E	F	G	H
γ_i	9	0	0	0	0	0	0	0	1

If we were to open facility H, then there would be a path of length 8 to a valid facility, which is a saving of 1 unit, as reflected by $\gamma_{\text{H}} = 1$. For all other nodes, the γ value is irrelevant except for dual feasibility as they do not occur in the dual objective function. The next step is to work out in which order the values of γ and λ are updated, as in line 6 of Algorithm 6. For this example, the sorted list Order looks like:

Order = {(1,-1,E), (1,E,G), (2,-1,D), (2,-1,F), (2,D,E), (2,D,F), (2,F,G), (2,F,H), (3,-1,C), (3,C,D), (4,-1,A), (4,A,C), (5,-1,B), (5,B,C), (5,B,F), (5,O,A), (5,O,B)}

Setting the dual variables in this order updates the remaining γ values and only sets two non-zero λ values ($\lambda_{\text{OA}} = 2$ and $\lambda_{\text{DE}} = 2$). The current non-zero dual variables are:

i	O	A	B	C	D	E	F	H	(i,j)	(D,E)	(O,A)
γ_i	9	6	7	5	4	1	2	1	λ_{ij}	2	2

If O were a valid facility, we would stop with a set of dual variables that generate a Pareto-optimal Benders cut. Since it is not, we must compare the shortest possible λ- and γ-weighted path with the current solution.

Because of the way the dual variables have been set, all paths from the origin to a facility will have a λ- and γ-weighted length of at least the length of the current solution, in this case 9. There are then multiple ways of selecting such a path, however this does not matter, because the only arcs we are concerned with are those that, when removed, increase the shortest possible weighted length. For this example, there are 4 distinct paths between O and G with the shortest possible weighted length, but the only arc which occurs on all such paths is (C,D). When computing Δ_a on line 16 of Algorithm 6, (C,D)

is the only arc to yield a positive value.

Removing (C,D) increases the length of the shortest possible weighted path to 11, which is 2 units more than γ_{O}. Now, in accordance with lines 18 to 21 of Algorithm 6, we set $\gamma_{\mathrm{O}} = 11$, $\lambda_{\mathrm{CD}} = 2$ and run the Propagate() function. This function initialises $R = \{(8,\mathrm{A}),(9,\mathrm{B})\}$, and runs the following steps:

(d,i)	Result
$(8,\mathrm{A})$	$\gamma_{\mathrm{A}} = 6 < 8$ so $\gamma_{\mathrm{A}} = 8$ and $R = \{(7,\mathrm{C}),(9,\mathrm{B})\}$
$(7,\mathrm{C})$	$\gamma_{\mathrm{C}} = 5 < 7$ so $\gamma_{\mathrm{C}} = 7$ and $R = \{(4,\mathrm{D}),(9,\mathrm{B})\}$
$(4,\mathrm{D})$	$\gamma_{\mathrm{D}} = 4 \geq 4$ so $R = \{(9,\mathrm{B})\}$
$(9,\mathrm{B})$	$\gamma_{\mathrm{B}} = 7 < 9$ so $\gamma_{\mathrm{B}} = 9$ and $R = \{(7,\mathrm{C}),(2,\mathrm{F})\}$
$(2,\mathrm{F})$	$\gamma_{\mathrm{F}} = 2 \geq 2$ so $R = \{(7,\mathrm{C})\}$
$(7,\mathrm{C})$	$\gamma_{\mathrm{C}} = 7 \geq 7$ so $R = \emptyset$

Now we are at line 24 of Algorithm 6, so we omit the facility at G and find the shortest possible weighted path to a different valid facility, *i.e.* H. In this case, the path is also 11 units long, so $\Delta_G = 0$ and the algorithm terminates. The non-zero dual variables are:

i	O	A	B	C	D	E	F	G	H	(i,j)	(D,E)	(O,A)	(C,D)
γ_i	11	8	9	7	4	1	2	0	1	λ_{ij}	2	2	2

These dual variables form a Pareto-optimal Benders cut for this case, and one can prove it using the procedure outlined in the proof of Theorem 9.

Dual-optimality of the solution given by Algorithm 6

We will begin by proving dual-optimality of the solution given by lines 1 to 12 of Algorithm 6, and then show that the changes made in lines 13 to 28 do not change the dual-optimality of the result. Before we prove dual-optimality of the results of Algorithm 6, we first must prove a relation between the γ-values of connected nodes.

Theorem 5. *For each node $i \in N$, for the node $j \in N$ and the path P that gives $\min_{j,P}\{\rho_{ij} + \sum_{a\in P}(\rho_{at}+\lambda_a) + \gamma_{p_n} | (i,j) \in A; P$ is a path from j to valid facility $p_n\}$, the inequality $\gamma_i \geq \rho_{ij} + \lambda_{ij} + \sum_{a\in P}(\rho_a + \lambda_a) + \gamma_{p_n}$ holds*

Proof. We start by considering all valid facilities. For each facility $k \in \Omega_r$, $\gamma_k = \max(\gamma_{\alpha_r} - D^c_{\alpha_r k}(\boldsymbol{\lambda}), 0)$. For all nodes $j \in N$ such that $(j,k) \in A$ and k is the closest facility to j, we have that $\gamma_j = \max(\gamma_{\alpha_r} - D^c_{\alpha_r j}(\boldsymbol{\lambda}), \rho_{jk} + \gamma_k)$. If $\gamma_j = \rho_{jk} + \gamma_k$, then as a result of line 12:

$$\lambda_{jk} = \max(0, \rho_{jk} + \gamma_k - \gamma_k - \rho_{jk}) \tag{3.120}$$

$$= 0 \tag{3.121}$$

So $\gamma_j = \rho_{jk} + \lambda_{jk} + \gamma_k$. If $\gamma_j = \gamma_{\alpha_r} - D^c_{\alpha_r j}(\boldsymbol{\lambda})$, then $\gamma_j \geq \rho_{jk} + \gamma_k$, and we have two options to consider: $\lambda_{jk} = 0$ and $\lambda_{jk} > 0$. If $\lambda_{jk} = 0$, then $\gamma_j \geq \rho_{jk} + \gamma_k = \rho_{jk} + \lambda_{jk} + \gamma_k$. If $\lambda_{jk} > 0$, then:

$$\lambda_{jk} = \gamma_j - \gamma_k - \rho_{jk} \tag{3.122}$$

$$\gamma_j = \rho_{jk} + \lambda_{jk} + \gamma_k \tag{3.123}$$

So in all cases, $\gamma_j \geq \rho_{jk} + \lambda_{jk} + \gamma_k$. Since γ_j is set before λ_{jk}, setting λ_{jk} does not change the value of γ_j and so γ_j is greater-than-or-equal-to the shortest λ-weighted path from j to a valid facility.

Now, for any node $i \in N$, assume that for the node $j \in N$ and path P that gives $\min\{\rho_{ij} + \sum_{a \in P}(\rho_{at} + \lambda_a) + \gamma_{p_n} | (i,j) \in A; P \text{ is a path from } j \text{ to a valid facility } p_n\}$, that $\gamma_j \geq \sum_{a \in P}(\rho_a + \lambda_a) + \gamma_{p_n}$. We again have two options to consider: $\lambda_{ij} = 0$ and $\lambda_{ij} > 0$. If $\lambda_{ij} = 0$, then as a result of line 10 we have:

$$\begin{aligned}
\gamma_i &\geq D_i^o(\boldsymbol{\lambda}) && (3.124)\\
&\geq \rho_{ij} + \sum_{a \in P}(\rho_a + \lambda_a) + \gamma_{p_n} && (3.125)\\
&\geq \rho_{ij} + \lambda_{ij} + \sum_{a \in P}(\rho_a + \lambda_a) + \gamma_{p_n} && (3.126)
\end{aligned}$$

In the case where $\lambda_{ij} > 0$, line 12 gives us:

$$\begin{aligned}
\lambda_{ij} &= \gamma_i - \gamma_j - \rho_{ij} && (3.127)\\
\gamma_i &= \rho_{ij} + \lambda_{ij} + \gamma_j && (3.128)\\
&\geq \rho_{ij} + \lambda_{ij} + \sum_{a \in P}(\rho_a + \lambda_a) + \gamma_{p_n} && (3.129)
\end{aligned}$$

So, by induction, we have that $\gamma_i \geq \rho_{ij} + \lambda_{ij} + \sum_{a \in P}(\rho_a + \lambda_a) + \gamma_{p_n}$ for all $i \in N$, which is that γ_i is greater than or equal to the length of the shortest λ-weighted path from i to a valid facility. □

Theorem 6. *For any arc (i,j), if $\gamma_i = D_i^o(\boldsymbol{\lambda})$, then $\lambda_{ij} = 0$*

Proof. This is easy to see, since if $\gamma_i = D_i^o(\boldsymbol{\lambda})$:

$$\begin{aligned}
\gamma_i &= D_i^o(\boldsymbol{\lambda})\\
&\leq D_j^o(\boldsymbol{\lambda}) + \lambda_{jk^*} + \rho_{ij}\\
&\leq \gamma_j + \rho_{ij}\\
\therefore \quad \gamma_i - \gamma_j - \rho_{ij} &\leq 0
\end{aligned}$$

by Theorem 5 and the shortest path property of $D_i^o(\boldsymbol{\lambda})$. Since $\lambda_{ij} = \max(0, \gamma_i - \gamma_j - \rho_{ijt})$, $\lambda_{ij} = 0$. The contrapositive of this is that if $\lambda_{ij} > 0$, then $\gamma_i = \gamma_{\alpha_r} - D_{\alpha_r i}^c(\boldsymbol{\lambda})$. □

Theorem 7. *During lines 15 and 16 of Algorithm 6, for any two arcs $a, b \in Q$, if $\Delta_a > \Delta_b > 0$, then a is on the shortest possible path from α_r to a valid facility that excludes b*

Proof. If it were not, then the shortest path from α_r to a valid facility that excludes b could also be used as the shortest possible path from α_r to a valid facility that excludes a, and thus $\Delta_a \leq \Delta_b$. Since $\Delta_a > \Delta_b$, a must be on the shortest possible path from α_r to a valid facility. □

Theorem 8. *The dual variables computed using Algorithm 6 are dual optimal*

Proof. The dual variables are dual feasible by construction. This is because when they are processed, the γ values at both ends of each arc must be set before the arc is processed, and then $\lambda_{ij} = \max(0, \gamma_i - \gamma_j - \rho_{ij})$ which will satisfy the dual constraints. The modifications made in the second half of Algorithm 6 do not make any constraints infeasible as the Propagate() function ensures $\gamma_j \geq \gamma_i - \rho_{ij} - \lambda_{ij}$.

For all locations $i \in N \setminus \{i^*\}$, either $\omega_{ir} = 0$, $x^*_{it} = 0$ or $\gamma_i = 0$. For arcs not on the current solution, either $y^*_{at} = 0$ or $\lambda_a = 0$. γ_{α_r} initially started as θ^* (by line 3), and for each arc on the current solution P^*, either $\lambda_a = 0$, or $\lambda_a = \Delta_a$ and Δ_a was added to γ_{α_r} (by line 20). In addition, either $\gamma_{i^*} = 0$ or $\gamma_{i^*} = \Delta_{i^*}$, and Δ_{i^*} was added to γ_{α_r} (on line 27), so for the current solution the Benders cut is

$$\gamma_{\alpha_r} - \gamma_{i^*} - \sum_{a \in P^*} \lambda_a = \theta^* + \sum_{a \in P^*} \Delta_a + \gamma_{i^*} - \gamma_{i^*} - \sum_{a \in P^*} \Delta_a \tag{3.130}$$

$$= \theta^* \tag{3.131}$$

This means these dual variables give exactly the objective value of the primal sub-problem. Since these dual variables give the primal objective value and satisfy all dual constraints, they form a dual optimal solution. □

So the dual variables computed using Algorithm 6 form a valid Benders cut. We now prove that this is a Pareto-optimal Benders cut using the definitions and proof method from Section 3.4

Theorem 9. *The Benders cut formed using the dual variables given by Algorithm 6 are Pareto-optimal*

Proof. The Benders cut generated by Algorithm 6, hereafter referred to as the analytic cut, is:

$$\theta_{rt} \geq \gamma^a_{\alpha_r} - \sum_{i \in N} \gamma^a_i \omega_{ir} x_i - \sum_{a \in A} \lambda^a_a y_a = \bar{\theta}^a(x, y) \tag{3.132}$$

Now assume there exists a cut that dominates this cut, called the dominating cut and denoted by $\bar{\theta}^d(x, y)$. We denote the coefficients of this cut as γ^d and λ^d, to match those of the analytic cut. When a Benders cut gives the correct objective value for a given master problem solution, we say it is *tight* at that solution. Since the analytic cut is tight for the current network configuration ($\bar{\theta}^a(x^*, y^*) = \bar{\theta}^*(x^*, y^*)$) the dominating cut must also ($\bar{\theta}^d(x^*, y^*) = \bar{\theta}^*(x^*, y^*)$). Now, we consider various scenarios that may or may not change the objective function as various values of x and y are changed.

Feasible master problem solutions

When proving Pareto-optimality of a Benders cut, only master problem solutions that produce feasible sub-problem solutions are considered. For the GUFLNDP, if there exists a path from the origin to an open valid facility, the master and sub-problems will both be feasible. As there are no other constraints on the design of the network, for example budget constraints or arc-dependencies, we may choose any network configuration with an open path from the origin to an open valid facility.

Locations that are not valid facilities

With the exception of the origin, α_r, any location that is not a valid facility ($\omega_{ir} = 0$) does not contribute directly to the value of either cut, and so is not important in assessing Pareto-optimality.

Valid facilities

For all valid facilities, $i \in \Omega_r$, $D_i^o(\boldsymbol{\lambda}) = 0$ since $\omega_{ir} = 1$. Therefore $\gamma_i^a = \max(\gamma_{\alpha_r}^a - D_{\alpha_r i}^c(\boldsymbol{\lambda}), 0)$, and so we have two cases for their associated dual variables: $\gamma_i^a = 0$ or $\gamma_i^a = \gamma_{\alpha_r}^a - D_{\alpha_r i}^c(\boldsymbol{\lambda}) > 0$.

If $\gamma_i^a = 0$, then $\gamma_{\alpha_r}^a \leq D_{\alpha_r i}^c(\boldsymbol{\lambda})$, that is, the shortest current λ-weighted path from the origin to the closest open valid facility, i^*, is no longer than the current shortest λ-weighted path from the origin to this location. Note that this includes i^*. For all facilities except i^* in this case, opening or closing the facility does not change the objective value. Since $\gamma_i^a = 0$ in this case, the analytic cut is tight, and thus the dominating cut must also be tight. This means $\gamma_i^d = \gamma_i^a = 0$ for all facilities except i^* where $\gamma_i^a = 0$.

If $\gamma_i^a > 0$, then there is an open path from the origin to this location that is shorter than the path from the origin to i^*, since $\gamma_{\alpha_r}^a > D_{\alpha_r i}^c(\boldsymbol{\lambda})$. If a facility were opened at this location, then the shortest path would now be $D_{\alpha_r i}^c(\boldsymbol{\lambda})$, and the objective value would reduce by $\gamma_{\alpha_r}^a - D_{\alpha_r i}^c(\boldsymbol{\lambda}) = \gamma_i^a$. So the analytic cut is tight, and so the dominating cut must also be tight. This means $\gamma_i^d = \gamma_i^a$ whenever $\gamma_i^a > 0$ with the exception of i^*.

So we now have that $\gamma_i^d = \gamma_i^a$ for any valid facility i except for i^*. Next we show that the dual variables for the arcs must also match.

Closed arcs with zero lambda

For any arc (i, j) where $\lambda_{ij}^a = 0$, we have:

$$\begin{aligned}
\gamma_i^a - \gamma_j^a - \rho_{ij} &\leq 0 \\
\gamma_i^a &\leq \gamma_j^a + \rho_{ij} \\
\gamma_{\alpha_r}^a - D_{\alpha_r i}^c &\leq \gamma_j^a + \rho_{ij} \\
\gamma_{\alpha_r}^a &\leq D_{\alpha_r i}^c + \gamma_j^a + \rho_{ij}
\end{aligned}$$

Now, if $\gamma_j^a = \gamma_{\alpha_r}^a - D_{\alpha_r j}^c$, the above inequality leads to $D_{\alpha_r j}^c \leq D_{\alpha_r i}^c + \rho_{ij}$, that is, the shortest path from the origin to j is shorter than the shortest path from the origin to i plus the length of the arc from i to j. This means that (i, j) is not a shortcut, and opening the arc will not affect the objective value, as reflected by $\lambda_{ij} = 0$.

If $\gamma_j^a = D_j^{o^*}(\boldsymbol{\lambda})$, then the above inequality instead leads to $\gamma_{\alpha_r}^a \leq D_{\alpha_r i}^c(\boldsymbol{\lambda}) + \rho_{ij} + D_j^{o^*}(\boldsymbol{\lambda})$, which is that the current solution is shorter than the current shortest path from the origin to i, plus the length of the arc from i to j, plus the shortest possible path from j to a valid facility. This again means that opening (i, j) will not yield any change in the objective value.

Thus, for any closed arc (i, j) where $\lambda_{ij}^a = 0$, opening just that arc will not change the objective value, and the analytic cut is tight for this new solution. As such, the dominating cut must also be tight for this solution, and so $\lambda_{ij}^d = \lambda_{ij}^a = 0$ for all closed arcs $(i, j) \in A$ such that $\lambda_{ij}^a = 0$.

Open arcs that are not part of the current solution

Similarly, for any open arc that is not on the current shortest path from the origin to a valid facility, closing that arc will not affect the objective value, and so $\lambda_{ij}^d = \lambda_{ij}^a = 0$ for all such arcs.

Closed arcs with non-zero lambda

Next, consider all paths on the shortest path trees from valid facilities to the other nodes. As we move backwards along one such path, P, we will cross arcs with zero and non-zero λ-values. Denote

the facility at the end of the path by k, and the arc on P closest to k with a non-zero λ value as (p_{i-1}, p_i). Since, for all arcs on P after this arc, $\lambda^a_{p_j p_{j+1}} = 0$, we know that $\gamma^a_{p_j} \leq \gamma^a_{p_{j+1}} + \rho_{p_j p_{j+1}}$. This leads to:

$$\begin{aligned}
\gamma^a_{p_i} &\leq \gamma^a_{p_{i+1}} + \rho_{p_i p_{i+1}} \\
&\leq \gamma^a_{p_{i+2}} + \rho_{p_{i+1} p_{i+2}} + \rho_{p_i p_{i+1}} \\
&\leq \ldots \\
&\leq \gamma^a_k + \sum_{j=i}^{n-1} \rho_{p_j p_{j+1}} \\
D^o_{p_i}(\boldsymbol{\lambda}) \leq \gamma^a_{p_i} &\leq D^o_{p_i}(\boldsymbol{\lambda})
\end{aligned}$$

where $p_n = k$. So $\gamma^a_{p_i} = D^o_{p_i}(\boldsymbol{\lambda})$. Since $\lambda^a_{p_{i-1} p_i} > 0$, $\gamma^a_{p_{i-1}} = \gamma^a_{\alpha_r} - D^c_{p_{i-1}}(\boldsymbol{\lambda})$ by Theorem 6, and $\lambda^a_{p_{i-1} p_i} = \gamma^a_{\alpha_r} - (D^c_{\alpha_r p_{i-1}}(\boldsymbol{\lambda}) + \sum_{j=i}^{n} \rho_{p_{j-1} p_j} + \gamma^a_k)$, or the savings on the arc are equal to the difference between the current solution and the current shortest path from the origin to p_{i-1}, plus the real length of the shortest possible weighted path from p_i to k and the amount already saved by opening the facility at k.

Opening the facility at k will reduce the objective value by γ^a_k as already shown. Opening the shortest path from p_i to k does not change the objective value since $\gamma^a_{\alpha_r} - D^c_{\alpha_r p_i}(\boldsymbol{\lambda}) \leq D^o_{p_i}(\boldsymbol{\lambda})$, which is reflected by the analytic cut (as all λ-values are zero). When (p_{i-1}, p_i) is opened also, the objective value will reduce by the difference between the current shortest path and this new weighted shortest path, i.e. $\lambda^a_{p_{i-1} p_i}$. As such, the analytic cut also correctly estimates this solution, and so must the dominating cut.

Now assume that when a facility, k, and the shortest possible weighted path from j to k, which has n non-zero λ-values, are opened, the analytic cut gives the correct objective value. Formally, this looks like $\gamma^a_k + \sum_{a \in P} \lambda^a_a = \gamma^a_{\alpha_r} - D^c_{\alpha_r j}(\boldsymbol{\lambda}) - \sum_{a \in P} \rho_a$, where P is the path from j to k we are concerned with. We now show that if an arc (i, j) has a non-zero λ value, then opening it and the shortest possible weighted path from j to k, as well as the facility k, will also give the correct objective value. That is, that $\gamma^a_k + \lambda^a_{ij} + \sum_{a \in P} \lambda^a_a = \gamma^a_{\alpha_r} - D^c_{\alpha_r i}(\boldsymbol{\lambda}) - \rho_{ij} - \sum_{a \in P} \rho_a$ holds.

Since $\lambda_{ij} > 0$, $\gamma^a_i = \gamma^a_{\alpha_r} - D^c_{\alpha_r i}(\boldsymbol{\lambda})$ by Theorem 6, and $\lambda^a_{ij} = \gamma^a_i - \gamma^a_j - \rho_{ij}$. For γ^a_j, there are two choices. If $\gamma^a_j = \gamma^a_{\alpha_r} - D^c_{\alpha_r j}(\boldsymbol{\lambda})$:

$$\begin{aligned}
\lambda^a_{ij} &= \gamma^a_i - \gamma^a_j - \rho_{ij} \\
&= \gamma^a_{\alpha_r} - D^c_{\alpha_r i}(\boldsymbol{\lambda}) - (\gamma^a_{\alpha_r} - D^c_{\alpha_r j}(\boldsymbol{\lambda})) - \rho_{ij} \\
&= D^c_{\alpha_r j}(\boldsymbol{\lambda}) - D^c_{\alpha_r i}(\boldsymbol{\lambda}) - \rho_{ij} \\
\text{So} \quad D^c_{\alpha_r j}(\boldsymbol{\lambda}) &= D^c_{\alpha_r i}(\boldsymbol{\lambda}) + \rho_{ij} + \lambda^a_{ij} \\
\text{and} \quad \gamma^a_k + \sum_{a \in P} \lambda^a_a &= \gamma^a_{\alpha_r} - D^c_{\alpha_r i}(\boldsymbol{\lambda}) - \lambda^a_{ij} - \rho_{ij} - \sum_{a \in P} \rho_a \\
\text{or} \quad \gamma^a_k + \lambda^a_{ij} + \sum_{a \in P} \lambda^a_a &= \gamma^a_{\alpha_r} - D^c_{\alpha_r it}(\boldsymbol{\lambda}) - \rho_{ij} - \sum_{a \in P} \rho_a
\end{aligned}$$

If instead $\gamma_j^a = D_j^o(\boldsymbol{\lambda})$ $\left(= \sum_{a \in P} (\lambda_a^a + \rho_a) + \gamma_k^a \text{ since } \lambda_{jb}^a = 0 \text{ for all } b \in N \text{ by Theorem 6} \right)$, then:

$$\begin{aligned} \lambda_{ij}^a &= \gamma_i^a - \gamma_j^a - \rho_{ij} \\ &= \gamma_{\alpha_r}^a - D_{\alpha_r i}^c(\boldsymbol{\lambda}) - \sum_{a \in P} (\lambda_a^a + \rho_a) - \gamma_k^a - \rho_{ij} \end{aligned}$$

$$\text{or} \quad \gamma_k^a + \lambda_{ij}^a + \sum_{a \in P} \lambda_a^a = \gamma_{\alpha_r}^a - D_{\alpha_r i}^c(\boldsymbol{\lambda}) - \rho_{ij} - \sum_{a \in P} \rho_a$$

So in all cases, if the analytic cut correctly gives the objective value when the shortest weighted path from a node j to its nearest facility k, and the facility itself, are opened, then it will also correctly give the objective value when opening the arc (i, j). By induction, the analytic cut is tight for any scenario where the shortest weighted path from a node j to its nearest facility k, and the facility itself, are opened. This can be used to show that, for all $(i, j) \in A$ such that $\lambda_{ij}^a > 0$ and (i, j) is closed, that $\lambda_{ij}^d = \lambda_{ij}^a$.

Arcs on the current solution

For arcs on the current solution that are not on the shortest possible path, $\lambda_a^a = 0$. When we open the shortest possible path, the Benders cut will give the correct objective value as shown above. Now, if we close the arcs on the current solution that are not on the shortest possible path, the objective value will not change, and the Benders cut is still tight, so $\lambda_a^d = \lambda_a^a = 0$ for these arcs.

The set Q contains the arcs that are on the current solution and the shortest possible path. For these arcs, Δ_a was calculated during the construction of the Benders cut. This value is the length of the shortest possible path that does not use arc a minus the value of γ_{α_r} at the time it was calculated. The values of λ_a were set in order from smallest to largest Δ_a, and we now consider these arcs in the same order. For the first arc that was set, a_1, every other arc in Q is on the shortest possible path that does not include a_1.

When we open the shortest possible path that does not include a_1, the change in the objective value is reflected by the Benders cut as shown above. Now, when we close a_1, the objective value will increase by λ_{a_1}, and the Benders cut will also show this, so the Benders cut is tight at this solution, and $\lambda_{a_1}^d = \lambda_{a_1}^a$.

For each subsequent arc $a_n \in Q$, we must consider those arcs that were set before it. For each arc a_k, where $k \in \{1, ..., n-1\}$, if it is on the alternative shortest path of a_n, then it does not affect the value of λ_{a_n}, since λ_{a_n} is the difference between the shortest possible path excluding a_n and γ_{α_r}. When λ_{a_k} was set, both quantities increased by the same value, and thus Δ_{a_n} did not change. These two arcs are thus independent of each other.

If a_k was not on the alternative shortest path of a_n, then it does affect the value of λ_{a_n}, for similar reasons as above. By closing only a_n, a saving of the initial value of Δ_{a_n} would be obtained, however since γ_{α_r} was increased by λ_{a_k} and the length of the alternative shortest path of a_n did not, λ_{a_n} underestimates the reduction in the objective value. In this case, a_n depends upon a_k.

For each arc $a_n \in Q$ where $n > 1$, we go through in order and start by opening the shortest possible path, a scenario at which our Benders cut is tight. We then open all arcs upon which a_n depends, as well as a_n itself. This scenario gives us the correct objective value, and the values of λ^d for all arcs

upon which a_n depends were previously fixed, so the only change is λ_{a_n}, and thus $\lambda^d_{a_n} = \lambda^a_{a_n}$ for all $a_n \in Q$.

Current facility and the origin

If we open the path to the closest possible facility that is not i^*, the Benders cut will give the correct objective value as shown above. If we then close i^*, the Benders cut is also tight at this solution in how γ_{i^*} was set. Thus $\gamma^d_{i^*} = \gamma^a_{i^*}$. Now the only value of the dominating cut that has not been fixed is $\gamma^d_{\alpha_r}$. Returning to the original network configuration, the Benders cut must give the correct objective value, and the only elements not shown to be equal between the two cuts are $\gamma^d_{\alpha_r}$ and $\gamma^a_{\alpha_r}$, so $\gamma^d_{\alpha_r} = \gamma^a_{\alpha_r}$. Thus the dominating cut is the analytic cut itself, and since a cut can not dominate itself, there are no cuts that dominate the analytic cut, so it is Pareto-optimal. □

3.8.4 Applicability of the GUFLNDP formulation

While it is possible to take a GUFLNDP problem and model it in the GUFLNDP framework, that may not always be the best approach. For example, the Tree of Hubs Location problem is best solved using a different formulation, first introduced by Contreras, Fernández and Marín (2009), and later used for Benders decomposition by de Sá, de Camargo and de Miranda (2013). While the formulation may not be identical, the ideas still apply: Benders decomposition is effective, the sub-problems disaggregate, and combinatorial feasibility cuts are appropriate. Similarly, the UFL problem can be reformulated to fit the GUFLNDP framework, but this introduces unnecessary complexity. The GUFLNDP should be used as a *guide* for how to apply Benders decomposition to these problems, not an exact recipe.

It is also worth noting that it is possible to design algorithms that yield valid Benders cuts that are not Pareto-optimal. For example, lines 1 to 12 of Algorithm 6 provide valid Benders cuts that may even be Pareto-optimal in some cases, but not all. The bulk of the computation in this algorithm resides in lines 13 to 28 due to the many shortest path computations, so the benefit of provably Pareto-optimal cuts should be weighed against the increased cost in computing them.

While the majority of GUFLNDP problems will benefit from Benders decomposition, there are a small number that will not. The GUFLNDP itself, for instance, does not benefit as it is far too general and a large number of Benders cuts are required to find sensible integer solutions, let alone an optimal solution. In this case, the number of additional constraints required may surpass the original problem size, and so it is more effective to solve the problem as a MIP rather than using Benders decomposition.

Another example is the Two-Level UFL (TUFLP) problem. First considered by Kaufman, Eede and Hansen (1977), this problem is an extension of the UFL problem, where customers are indirectly connected to facilities via satellite facilities. This adds a new layer of variables and constraints to the problem, greatly increasing the number of potential solutions to consider. This problem is also clearly a GUFLNDP problem.

A recent honours thesis by Rist [91] applies Benders decomposition to the TUFLP and shows that it is not particularly effective. The suggested reason for this is that the problem is very general and many Benders cuts are required for finding an optimal solution. In some cases, the number of Benders

cuts added is within an order of magnitude to the number of constraints removed from the original MIP. This greatly reduces the effectiveness of Benders decomposition by removing the primary benefit of a smaller master problem with far fewer constraints.

In both these cases, the problem is that the space of feasible master problem solutions is very large, and so a large number of Benders optimality cuts are required to build the initial support for the approximation variables. By the time the solutions are returning reasonable (but still incorrect) approximations, the number of Benders optimality cuts is so large that processing the nodes of the branch-and-bound tree will not be much faster than it was in the original MIP. Despite these examples, the majority of GUFLNDP problems have quite restrictive master problems that reduce the number of Benders cuts required to solve them, making Benders decomposition beneficial.

3.9 Discussion

There is a wide range of problems for which Benders decomposition is suitable, and there are many improvements that consistently yield improvements. The most important of these is the disaggregation of the sub-problem where possible, and the next most important is to embed Benders decomposition in a branch-and-cut framework. There are few scenarios where implementing either of these is not beneficial. The next thing to try is warm-starting the solver or adding initial cuts, which usually makes an improvement.

The main reason why Benders decomposition solves problems faster than a standard MIP model is that the master problem is significantly smaller, so it is easier to process nodes in the branch-and-bound tree. It may be the case that more nodes need to be explored, but this is more than compensated for by how much faster they are processed. In practice, the reduced size of the master problem can make the automatic cutting plane and heuristic generation algorithms in the main solvers more effective, leading to even more impressive speed increases.

Embedding Benders decomposition in a branch-and-cut framework is a prime application of lazy constraints, and there are interesting parallels between Benders decomposition and other lazy formulations. In particular, both cases involve solving incomplete problems that are “repaired” by adding constraints lazily. The main difference is that for Benders decomposition, we first formulate the complete problem and decompose it, whereas a lazy formulation starts with an incomplete problem that is completed with lazy constraints. We explore these parallels further in the next chapter.

The following publication has been incorporated as part of Chapter 4.

1. [1] **Robin H. Pearce** and Michael Forbes, Puzzle - The Fillomino Puzzle, *INFORMS Transactions on Education* 17 (2), 2017

Contributor	Statement of contribution	%
Robin H. Pearce	writing of text	100
	proof-reading	80
	theoretical derivations	90
	numerical calculations	100
	preparation of figures	100
	initial concept	80
Michael Forbes	proof-reading	20
	supervision, guidance	100
	theoretical derivations	10
	initial concept	20

Chapter 4

Lazy Formulations

> Never put off till tomorrow what may be done day after tomorrow just as well
>
> *Mark Twain*

This chapter contains some examples of problems with exponentially sized sets of constraints, many of which are unnecessary for finding an optimal solution. It also introduces a new way of thinking about modelling problems, where instead of formulating a *compact* model and then decomposing or otherwise finding a way of solving it efficiently, one finds the simplest model that would give a solution close to what is desired, and correct that solution with lazy constraints. The best problem with which to introduce these ideas is the Travelling Salesman Problem.

4.1 The Travelling Salesman Problem

The Travelling Salesman Problem (TSP) is one of the most widely studied problems in Operations Research. The objective is to find the shortest tour through a set of locations and return to the starting position. It was first considered by Hamilton in the mid-1800s, as it is equivalent to the problem of finding the shortest Hamiltonian cycle through a graph. While it had been mathematically described prior to the advent of linear programming, it became a central problem in 1954 when Dantzig *et al.*found the shortest tour of 49 US cities — 48 state capitals and Washington D.C., the national capital [30].

Since then, techniques for solving TSPs have been a hot topic of research, and many large TSPs once considered intractable are now almost trivial. As of August 2018, the largest TSP solved to optimality is a tour of 85,900 points in an application of Very-large-scale integration (VLSI), a technique for designing integrated circuits. There are a number of larger VLSI problems that have yet to be solved to optimality, and there is even a "world tour" of 1,904,711 cities across the globe, including several research bases in Antarctica. Every few years, a new, slightly improved tour is found,

the latest being in March 2018. A best bound for this problem was established in 2007, which puts the current optimality gap at 0.0474% [92].

The Travelling Salesman Problem is an excellent example of how lazy constraints can be useful, so much so that Gurobi uses it as one of its examples on implementing lazy constraints [93]. This includes sample code demonstrating the implementation of a TSP solver using integer programming with lazy constraints. There are multiple ways of formulating the TSP as an integer programming problem, one of which is as follows:

Sets

N Set of locations

Data

d_{ij} Distance from location $i \in N$ to location $j \in N$

Variables

x_{ij} 1 if location $i \in N$ is connected to location $j \in N$, 0 otherwise

Objective

$$\min \sum_{i \in N} \sum_{\substack{j \in N \\ j < i}} d_{ij} x_{ij} \tag{4.1}$$

Constraints

$$\sum_{j \in N} x_{ij} = 2 \qquad \forall i \in N \tag{4.2}$$

$$x_{ij} = x_{ji} \qquad \forall i \in N, \forall j \in N \tag{4.3}$$

$$x_{ii} = 0 \qquad \forall i \in N \tag{4.4}$$

$$\sum_{\substack{i,j \in S \\ i<j}} x_{ij} \leq |S| - 1 \qquad \forall S \subset V, S \neq \emptyset, |S| < |V|/2 \tag{4.5}$$

$$x_{ij} \in \{0,1\} \qquad \forall i \in N, \forall j \in N \tag{4.6}$$

The objective is to minimise the sum of the distances of the connections. Constraints (4.2) ensure that every node is connected to exactly two nodes, and constraints (4.3) make the variables symmetric, so the result is an undirected graph. Constraints (4.4) stop nodes connecting to themselves, and constraints (4.5) are to prevent disconnected sub-cycles from appearing in the solution. These are necessary, since a valid solution to a 6-node TSP without constraints (4.5) may be similar to Figure 4.1.

To prevent this case from happening, it is sufficient to enforce that for every set of three nodes, there can be at most two links connecting any nodes in that set. By enforcing this for every subset of every size (greater than 2), as in constraints (4.5), the solution is guaranteed to be free of sub-cycles. The problem with this is the number of constraints in (4.5) is of the order 2^{n-1}, which is exponential in the number of nodes. Thus, as the number of nodes increases, the difficulty in solving this problem grows very quickly.

Figure 4.1: A solution to a 6-node TSP with sub-cycles

An important note is that many of these sub-cycles are unlikely to occur in an optimal solution; for example, assuming the cost of travelling between two nodes is proportional to the euclidean distance between them, the two nodes on the left and the one furthest to the right in Figure 4.1 are unlikely to occur as a sub-cycle, as the cost of that sub-cycle and the one formed by the remaining nodes would be more expensive than the optimal solution. This also means the sub-cycle involving those remaining nodes, while small, is also unlikely because it forces a longer sub-cycle to occur.

Thus, a more efficient way to solve the TSP is to start without constraints (4.5), and each time a new incumbent integer solution is found, inspect it for sub-cycles. If a sub-cycle is found, add the relevant constraint to remove it, and continue searching the branch-and-bound tree. A specialised solver, the Concorde TSP solver [94], uses this among other techniques, and is considered one of the best TSP solvers.

Dantzig, in his papers in 1954 [30] and 1959 [31], describes how one could add only those constraints in (4.5) that are required, and only at the times they are required. This appears to be the first description of a lazy modelling approach. Since then, similar approaches have been used for a number of problems such as the Vehicle Routing problem [34], the Clique Partitioning problem [32] and the Weighted Node Packing problem [33].

4.1.1 Compact formulation

The presented formulation is obviously not the only way of solving the TSP, and there are other formulations that do not have an exponential number of constraints; however, they require the addition of auxiliary variables to enforce the connectivity of the tour. There are at least two that are easy to describe: a flow-based formulation and a time-based formulation.

The flow-based formulation involves flowing some commodity from an arbitrary source node to all other nodes. That way, if a disconnected sub-cycle exists, there is no way to flow the commodity to the nodes in the cycle, and the solution is infeasible. An auxiliary variable is added for each potential connection between two nodes, and two sets of constraints are added: capacity constraints and flow-conservation constraints. The capacity constraints limit the flow of the commodity to only the selected links in the tour, and the flow-conservation constraints assign a demand of 1 to each node, except for the source node, which has a sufficiently large supply.

The time-based formulation requires the tour to be directed and enforces an ordering on the nodes. If a connection between two nodes exists, the time that the end node is visited is at least the time the start node is visited plus the length of the connection. There is only one node that is exempt from this rule, which could be considered the start of the tour, and the last node of the tour can connect to the first without violating these constraints. Thus, if a sub-cycle exists, there will be a series of inequalities that, when combined, lead to a contradiction and render the solution illegal. This is a variation of *MTZ constraints*, due to the authors who first proposed them for the Traveling Salesman Problem [95].

It is best to consider the time-based formulation for reasons that will soon become apparent. The formulation is as follows:

Sets

N Set of locations

Data

d_{ij} Distance from location $i \in N$ to location $j \in N$

Variables

x_{ij} 1 if location $i \in N$ is connected to location $j \in N$, 0 otherwise

z_i *Time* of visit at location $i \in N$

Objective

$$\min \sum_{i \in N} \sum_{j \in N} d_{ij} x_{ij} \tag{4.7}$$

Constraints

$$\sum_{j \in N} x_{ij} = 1 \qquad \forall i \in N \tag{4.8}$$

$$\sum_{j \in N} x_{ji} = 1 \qquad \forall i \in N \tag{4.9}$$

$$x_{ii} = 0 \qquad \forall i \in N \tag{4.10}$$

$$z_j \geq z_i + d_{ij} - M(1 - x_{ij}) \qquad \forall i \in N, \forall j \in N \setminus \{0\} \tag{4.11}$$

$$x_{ij} \in \{0,1\}, z_i \geq 0 \qquad \forall i \in N, \forall j \in N \tag{4.12}$$

Constraints (4.8-4.9) now say that there must be one arc entering and one arc leaving each node, in contrast to constraints (4.2). Constraints (4.10) are unchanged, and constraints (4.11) are the new *time* constraints. If a link (i, j) is not used, the big-M term effectively turns the constraint off. If the link is used, the time of arrival at node j is at least the time of arrival at node i plus the distance between i and j.

This is now a compact formulation of the TSP that does not involve any exponentially sized sets of constraints. Perhaps the more interesting feature of this formulation is that it is suitable for Benders decomposition.

4.1.2 Benders decomposition

In this case, the master problem is the formulation above without (4.11), and the sub-problem is to solve the timing problem to ensure the master problem solution is a valid solution. Since the sub-problem does not appear in the objective function, no Benders optimality cuts are required, only Benders feasibility cuts. As seen in Chapter 3, *Irreducible Inconsistent Subset* (IIS) feasibility cuts are more effective than traditional Benders feasibility cuts, and are appropriate in this example.

Let us consider a case where there exists a sub-cycle of three nodes, $\{1,2,3\}$, and the distance between each pair of these nodes is 5. Then constraints (4.11) give the following:

$$\begin{aligned} z_2 &\geq z_1 + 5 \\ z_3 &\geq z_2 + 5 \geq z_1 + 10 \\ z_1 &\geq z_3 + 5 \geq z_1 + 15, \end{aligned}$$

which is clearly a contradiction. Thus, the sub-problem is infeasible, and a Benders feasibility cut is required. The only constraints in the sub-problem are (4.11), and if the right-hand side of any one of the three cuts above contained the large negative big-M value, the infeasibility would be resolved. That is, the IIS cut generated in this situation is:

$$\sum_{(i,j)\in \text{IIS}} x_{ij} \leq |\text{IIS}| - 1, \tag{4.13}$$

where IIS is the set of links in the sub-cycle. Note that as the graph is directed, the reverse cycle would be feasible under this cut. Also, if there are more than three nodes in the sub-cycle, a re-ordering of the nodes could also be considered feasible (although perhaps not optimal if the reordered tour is longer). Thus, we could *lift* this cut to include all links between all nodes in the sub-cycle. This would then make the cut equivalent to one of the constraints in (4.5).

This means that the original lazy formulation can be considered a Benders decomposition on an underlying compact formulation. This was shown for a similar formulation of the TSP by Gavish and Graves (1978). The flow-based compact formulation can be decomposed in the same manner; however, the IIS cuts are the inverse of (4.5), in that rather than limiting the number of links between the nodes in the sub-cycle, it enforces at least one arc to be opened from a node outside the sub-cycle to a node in the sub-cycle.

Table 4.1 shows a comparison of three formulations on 10 randomly generated instances with 50 cities each. The locations are uniformly distributed in a square area and the distances between locations are euclidean. The solution time, number of nodes processed, and lazy constraints added have been averaged over the 10 instances. The Lazy formulation is the formulation presented in (4.1-4.5), the Compact formulation is the time-based formulation shown in (4.7-4.11), and Benders is the result of applying Benders decomposition to the Compact formulation. In the Benders formulation, we detect multiple sub-cycles and add a cut for each one at each integer solution. This is achieved by modifying

Formulation	Time (s)	Nodes	Lazy Constraints
Lazy	0.46	1135	107
Compact	142.17	145209	-
Benders	115.23	42690	2554

Table 4.1: Comparison of different TSP implementations on 10 randomly-generated instances with 50 cities

the sub-problem to make the previous sub-cycle legal, re-solving and finding a new IIS. We repeat this until the sub-problem is feasible.

The lazy formulation is the fastest, because it is the smallest model and does not require the solution of additional LPs or the computation of any IIS. Instead, a simple labelling algorithm can find sub-cycles and the cuts can be constructed directly. The Compact formulation is the slowest, which is not surprising as it is the largest model with the most variables and constraints. The Benders formulation is faster than Compact, but slower than Lazy. This is because of the additional time spent solving the sub-problems and the fact the cuts are not as strong as those used in the Lazy formulation, which results in the generation of more Benders cuts.

An interesting note is that the behaviour exhibited here is similar to that in the paper from Section 3.5. The Benders formulation is faster than the Compact formulation for two reasons: the nodes in the branch-and-bound tree are smaller and easier to process, and fewer of them need to be explored to solve the problem. As noted previously, this is likely because the algorithmic features built into Gurobi are more effective on the smaller problem.

The TSP, while important, is not the only problem that has one or more exponentially sized set(s) of constraints that would benefit from the use of lazy constraints. As such, it is important to understand how to use them and in what contexts. Logic puzzles can be modelled as integer programs, and make good classroom examples as they are easy to understand and often benefit from techniques also applicable to industrial problems. The remainder of this chapter will cover two such puzzles: Anne Bonney (the Pieces of 8 puzzle), and the Fillomino puzzle.

4.2 Anne Bonney (the Pieces of 8)

The Melbourne University Mathematics and Statistics Society holds a “puzzle hunt” every year (since 2004 with the exception of 2017). In act 1 of the 2011 collection of puzzles is one titled Anne Bonney, which we refer to as the Pieces of 8 puzzle [97]. In this puzzle, one starts with an incomplete treasure map that has the location of a ship, some buried treasure, and a few numbers. The resulting solution is a contiguous path that connects the ship to the treasure, does not cross itself or touch itself except diagonally, and the remaining squares are divided into eight *pieces*. The pieces numbered 1 through 7 contain that many squares (*i.e.* four squares in the piece of 4), and the eighth piece is of undetermined size. Some maps have cells missing from the grid.

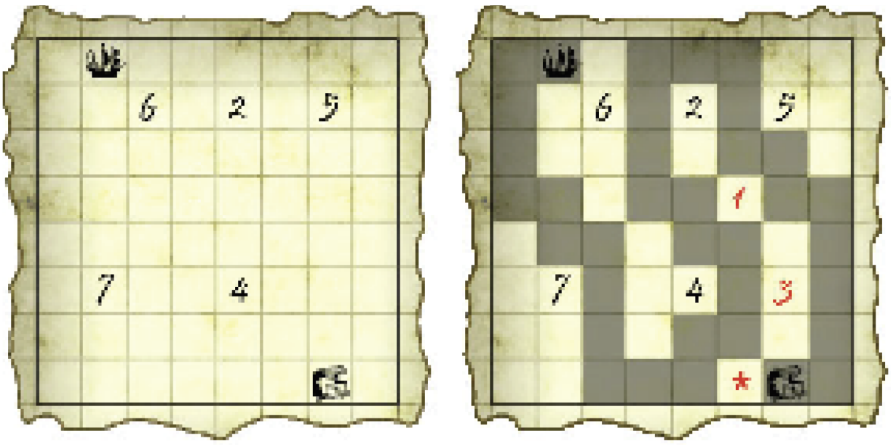


Figure 4.2: An example starting grid and corresponding solution to Anne Bonney (Pieces of 8). Image source: Corey Plover, MUMS Puzzle Hunt 2011 [97]

4.2.1 Lazy formulation

Like most logic puzzles, this can be modelled as an integer programming problem. Most of the rules can easily be described by constraints, except for the rules that the pieces must be contiguous and the path must be connected. While these rules can be tricky to describe in a compact model, they are easy to enforce using a lazy model. Something to note is that the path may be considered a piece as well, and it must be contiguous, which is the same as being connected. The path is denoted type 0, and the missing cells are denoted type 9. The lazy integer programming model for solving the pieces of 8 puzzle is as follows:

Sets

C Set of cells, represented as (i, j)

K Set of cell types. $K = \{0, ..., 9\}$, $K_7 = \{1, ..., 7\}$ and $K_8 = \{1, ..., 8\}$

N_{ij} Set of cells that share an edge with cell $(i, j) \in C$

Constants

p_{ij} Pre-set value for cell $(i, j) \in C$. -1 if no value given.

Variables

x_{ijk} 1 if cell $(i, j) \in C$ is of type $k \in K$, 0 otherwise

Constraints

$$\sum_{k \in K} x_{ijk} = 1 \qquad \forall (i, j) \in C \tag{4.14}$$

$$x_{ijp_{ij}} = 1 \qquad \forall (i, j) \in C, p_{ij} \geq 0 \tag{4.15}$$

$$x_{ij9} = 0 \qquad \forall (i, j) \in C, p_{ij} \neq 9 \tag{4.16}$$

$$\sum_{(i,j) \in C} x_{ijk} = k \qquad \forall k \in K_7 \tag{4.17}$$

$$\sum_{(i,j) \in C} x_{ij8} \geq 1 \tag{4.18}$$

$$x_{ijk} + \sum_{\substack{k' \in K_8 \\ k' \neq k}} x_{i'j'k'} \leq 1 \qquad \forall (i,j) \in C, \forall (i',j') \in N_{ij}, \forall k \in K_8 \tag{4.19}$$

$$\sum_{(i',j') \in N_{ij}} x_{i'j'0} = 1 \qquad \forall (i,j) \in C, p_{ij} = 0 \tag{4.20}$$

$$\sum_{(i',j') \in N_{ij}} x_{i'j'0} \geq 2x_{ij0} \qquad \forall (i,j) \in C, p_{ij} < 0 \tag{4.21}$$

$$\sum_{(i',j') \in N_{ij}} x_{i'j'0} \leq 4 - 2x_{ij0} \qquad \forall (i,j) \in C, p_{ij} < 0 \tag{4.22}$$

$$x_{ijk} \leq \sum_{(i',j') \in N_{ij}} x_{i'j'k} \qquad \forall (i,j) \in C, \forall k \in K_7 \setminus \{1\} \tag{4.23}$$

$$x_{ijk} \in \{0,1\} \qquad \forall (i,j) \in C, \forall k \in K \tag{4.24}$$

Constraints (4.14) ensure that each cell has exactly one type, and constraints (4.15) enforce the pre-set values. Constraints (4.16) make sure that a cell can only be of type 9 (blank square) if it is pre-set to type 9. Constraints (4.17) ensure the first seven pieces are of the correct size, and constraints (4.18) say that there must be at least one cell in the eighth piece. Constraints (4.19) make sure that no two pieces share an edge. Constraints (4.20) ensure the origin and destination have exactly one neighbour each, and constraints (4.21-4.22) make sure all other path squares have exactly two neighbours. Finally, constraints (4.23) say that for any cell of type $k \in \{2,...,7\}$, it has at least one neighbour of the same type. Note that constraints (4.22) can be tightened for cells on the boundary by changing the RHS to $3 - 2x_{ij0}$, and are redundant for cells in the corners of the grid.

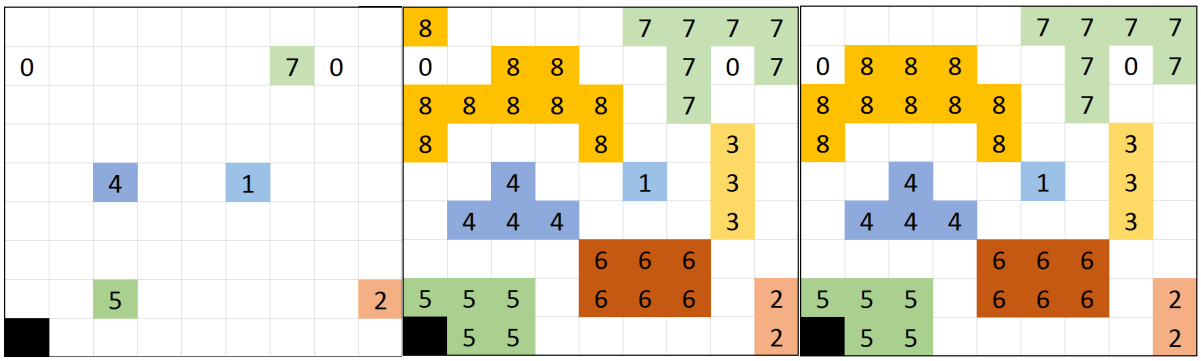
This formulation covers many of the rules and will produce solutions close to what is required, but it still allows the possibility of a disconnected piece or section of path. Such solutions are excluded through the use of lazy constraints. Now we must decide how to exclude these solutions. The most general method for excluding a particular integer solution, x^*, is to add the following constraint:

$$\sum_{(i,j) \in C} \sum_{\substack{k \in K \\ x^*_{ijk} = 1}} x_{ijk} \leq |C| - 1, \tag{4.25}$$

that is, the sum of all variables corresponding to assignments in the current solution must reduce by at least 1. This does not cut off any solutions other than the current one, as to violate this constraint, all values of x must be equal to x^*. The problem with this constraint is that if there are multiple disconnected pieces, fixing one will satisfy this constraint while the others may remain broken. Thus, we would like a tighter constraint for fixing these broken pieces.

Let P be the set of all cells in an offending piece or disconnected section of path, and k' be the type of piece or path. The following constraint will discard the current solution and prevent the same configuration from occurring again:

$$\sum_{(i,j) \in P} x_{ijk'} \leq |P| - 1. \tag{4.26}$$



(a) Initial grid (b) First solution (c) Solved grid

Figure 4.3: Example of correcting a Pieces of 8 puzzle solution with lazy constraints or Benders decomposition. (a) The starting grid showing the pre-set values. (b) The solution obtained without any contiguity constraints. (c) The solution to this instance

This cut ensures that this specific selection of cells cannot occur again, in that at least one of the offending cells must change its type. Importantly, we have not just cut off a single integer solution, but all solutions that contain this illegal configuration of cells. In doing this, we have identified the smallest portion of the solution that is breaking the rules, and have cut off only that offending portion.

The only exception to this is the eighth piece. Because its size is unknown, an illegal configuration could be fixed simply by adding additional cells to the eighth piece, rather than moving existing ones. This would make (4.26) an illegal cut. To remedy this, we use a more general cut:

$$\sum_{(i,j)\in P} (1 - x_{ijk'}) + \sum_{(i,j)\in PN} x_{ijk'} \geq 1, \tag{4.27}$$

where PN is the set of cells that neighbour a cell in P but are not in P themselves. This cut says either one of the cells in the broken piece must be removed, or one of the neighbours needs to become part of the same piece. This cut works for all pieces, so may be used as an alternative to (4.26).

Interestingly, in practice (4.26) is sufficient for solving all instances, even when applied to the eighth piece. This is because of the interconnectivity of the other constraints, so that if another cell were to be added to the eighth piece without taking away any existing cells, a cell from a different piece would have to be removed. The only option for this is the path, as all other pieces have fixed size. The need for the path to be connected is often enough to prevent this situation from occurring.

We apply each of these cuts to every part of a disconnected piece, which may mean that at a particular integer solution, we add multiple cuts. This is not a problem because the cuts are tight, so few of these cuts will be required overall. The other benefit is that there is always the potential that we do not require any lazy constraints at all, and the lazy model will be sufficient for finding the solution.

An example of how this works is shown in Figure 4.3. Image (a) shows the starting grid for instance 1 with the pre-set values. The two 0s represent the ship and the treasure. Since the path is undirected, these two are interchangeable. The black square in the bottom-left is a missing cell with a value of 9. After solving the model without any contiguity constraints, solution (b) is obtained. Note that piece 8 is disconnected.

Instance	Time (s)	Nodes	Lazy Constraints
1	0.47	0	1
2	0.47	71	0
3	2.66	2842	0
4	0.11	0	0
5	0.22	1	4
6	0.26	1	5
7	0.16	0	0
8	0.70	537	17
9	0.14	0	0

Table 4.2: Time to solution, number of branch-and-bound nodes explored and number of lazy constraints required for the nine instances of the Pieces of 8 puzzle using the lazy formulation

At this point, a constraint like (4.26) is applied to the 8 piece, so at least one of those cells must change. The solver then moves the cell in the top-left corner down and to the right, connecting it to the rest of the piece, as well as maintaining the path-connectivity. This is a legal solution, and so the optimisation terminates.

We implemented this formulation using Gurobi 8.1.0 with Python 3.7.0, both 64-bit, on a machine running an Intel i5-8250U quad-core 1.6GHz processor with 4GB of RAM. Table 4.2 shows the time it takes to solve each instance and the number of lazy constraints added before finding the solution. Each instance has exactly 810 variables and about 3126 constraints, give or take a few depending upon the number of pre-assigned values.

Note that four of the nine instances solved with 0 branch-and-bound nodes explored. In these cases, the entire problem was solved by Gurobi's pre-solver, a logical processor that looks to reduce the size of the problems by making logical deductions about the relationships between the variables. As stated previously, this is one of the main benefits of using lazy constraints with modern solvers: all of the powerful processing techniques are automatically applied to the smaller, easier-to-solve models, leading to even greater speed increases.

The other important result is that fewer than 10 lazy constraints were needed for all but one of the instances. To describe the contiguity rules using a compact formulation would have required far more than 10 additional constraints, and in more than half of the instances, those constraints are completely unnecessary. This is another main benefit of using lazy constraints: saving time by not having to design a compact formulation, and by not having to handle a number of constraints that may be mostly or completely unnecessary.

4.2.2 Compact formulation

For this puzzle, it is not difficult to describe a compact formulation to solve it; however, similar to the TSP, it does require a number of *auxiliary* variables to handle the contiguity of the pieces. The idea is that each piece has a *source* cell that supplies *commodity* of type $k \in K$, and every cell generates 1 demand for its relevant commodity. If some cells are not connected to an appropriate source, then at

least one piece is disconnected and the solution is infeasible. The compact formulation for the Pieces of 8 puzzle is as follows:

Sets

C Set of cells, represented as (i, j)

K Set of cell types. $K_7 = \{1, ..., 7\}$ and $K_8 = \{1, ..., 8\}$

N_{ij} Set of neighbours of cell $(i, j) \in C$

Constants

p_{ij} Pre-set value for cell $(i, j) \in C$. -1 if no value given.

Variables

x_{ijk} 1 if cell $(i, j) \in C$ is of type $k \in K$, 0 otherwise

y_{ijk} 1 if cell $(i, j) \in C$ is the source of type $k \in K$, 0 otherwise

$z_{iji'j'k}$ Amount of flow from cell $(i, j) \in C$ to neighbouring cell $(i', j') \in N_{ij}$ of *commodity* $k \in K$

Constraints

$$\sum_{k \in K} x_{ijk} = 1 \qquad \forall (i, j) \in C \tag{4.28}$$

$$\sum_{(i,j) \in C} y_{ijk} = 1 \qquad \forall k \in K \tag{4.29}$$

$$x_{ijp_{ij}} = 1 \qquad \forall (i, j) \in C, p_{ij} \geq 0 \tag{4.30}$$

$$y_{ijp_{ij}} = 1 \qquad \forall (i, j) \in C, 1 \leq p_{ij} \leq 8 \tag{4.31}$$

$$y_{i^* j^* 0} = 1 \tag{4.32}$$

$$x_{ij9} = 0 \qquad \forall (i, j) \in C, p_{ij} \neq 9 \tag{4.33}$$

$$\sum_{(i',j') \in N_{ij}} x_{i'j'0} = 1 \qquad \forall (i, j) \in C, p_{ij} = 0 \tag{4.34}$$

$$\sum_{(i,j) \in C} x_{ijk} = k \qquad \forall k \in K_7 \tag{4.35}$$

$$\sum_{(i,j) \in C} x_{ij8} \geq 1 \tag{4.36}$$

$$x_{ijk} + \sum_{\substack{k' \in K_8 \\ k' \neq k}} x_{i'j'k'} \leq 1 \qquad \forall (i, j) \in C, \forall (i', j') \in N_{ij}, \forall k \in K_8 \tag{4.37}$$

$$\sum_{(i',j') \in N_{ij}} x_{i'j'0} \geq 2x_{ij0} \qquad \forall (i, j) \in C, p_{ij} < 0 \tag{4.38}$$

$$\sum_{(i',j') \in N_{ij}} x_{i'j'0} \leq 4 - 2x_{ij0} \qquad \forall (i, j) \in C, p_{ij} < 0 \tag{4.39}$$

$$x_{ijk} \leq \sum_{(i',j') \in N_{ij}} x_{i'j'k} \qquad \forall (i, j) \in C, \forall k \in K_7 \setminus \{1\} \tag{4.40}$$

$$z_{iji'j'k} \leq M x_{ijk} \qquad \forall (i, j) \in C, \forall (i', j') \in N_{ij}, \forall k \in \{0, 8\} \tag{4.41}$$

$$z_{iji'j'k} \leq (k-1)(x_{ijk} - y_{ijk}) \qquad \forall (i, j) \in C, \forall (i', j') \in N_{ij}, \forall k \in K \tag{4.42}$$

$$\sum_{(i',j') \in N_{ij}} z_{i'j'ijk} - \sum_{(i',j') \in N_{ij}} z_{iji'j'k} = x_{ijk} - k y_{ijk} \qquad \forall (i, j) \in C, \forall k \in K_7 \tag{4.43}$$

Instance	Time (s)	Nodes
1	0.23	0
2	0.78	1
3	5.39	641
4	0.21	1
5	0.34	1
6	0.35	0
7	0.27	0
8	0.42	1
9	0.23	1

Table 4.3: Time to solution and number of branch-and-bound nodes explored for the nine instances of the Pieces of 8 puzzle using the compact formulation

$$\sum_{(i',j')\in N_{ij}} z_{i'j'ijk} - \sum_{(i',j')\in N_{ij}} z_{iji'j'k} \geq x_{ijk} - My_{ijk} \qquad \forall (i,j) \in C, \forall k \in \{0,8\} \tag{4.44}$$

$$y_{ijk} \leq x_{ijk} \qquad \forall (i,j) \in C, \forall k \in K \setminus \{9\} \tag{4.45}$$

$$z_{iji'j'k} \geq 0 \qquad \forall (i,j) \in C, \forall (i',j') \in N_{ij} \tag{4.46}$$

$$x_{ijk} \in \{0,1\}, y_{ijk} \in \{0,1\} \qquad \forall (i,j) \in C, \forall k \in K \tag{4.47}$$

Constraints (4.28) ensure that each cell has exactly one type, and constraints (4.29) specify exactly one source for each piece. Constraints (4.30) enforce the pre-set values, and constraints (4.31) say that if a pre-set value for a piece is given, it must be used as the seed. Constraints (4.32) set the source for the path to one of the two pre-set values for the path, (i^*, j^*). Constraints (4.33) make sure that a cell can only be of type 9 (blank square) if it is pre-set to type 9. Constraints (4.34) say that the origin and destination have exactly one neighbour each. Constraints (4.35) ensure the first seven pieces are of the correct size, and constraints (4.36) say there must be at least one cell in the eighth piece. Constraints (4.37) make sure that no two pieces share an edge, and constraints (4.38-4.39) ensure all other path squares have exactly two neighbours. Constraints (4.40) say that for any cell of type $k \in \{2,...,7\}$, it has at least one neighbour of the same type.

Constraints (4.41-4.42) only allow commodity to flow out of a cell if it is of the correct type. Constraints (4.43-4.44) are flow-conservation constraints, which specify that the amount flowing into a cell must be at least the amount flowing out, but if it is of type k, it generates a demand of size 1, and if it is a source of type k it generates a sufficiently large supply. Since we know exactly how many cells of types 1 to 7 there are, the corresponding constraints can be equality, but for the path and the 8th piece, we must use a big-M constraint. Finally, constraints (4.45) constrain piece seeds to occur only on cells of the same commodity.

This is a compact formulation that solves the Pieces of 8 puzzle without the need for additional constraints. While this formulation is not very complicated, creating a similar formulation for other puzzles can become inefficient. Table 4.3 shows the time and number of branch-and-bound nodes required to solve the instances of the puzzle.

Note that most instances take longer to solve than with the lazy formulation, the exceptions being instances 1 and 8. One other interesting difference is that the number of branch-and-bound nodes

explored in this case is less than for the lazy formulation. Again, this is not a problem, as the lazy model is significantly smaller than the compact formulation (far fewer variables and constraints), and so processing each branch-and-bound node takes less time.

4.2.3 Benders decomposition

Now that we have a compact formulation, it is again possible to solve it using Benders decomposition as it has a natural structure: the master problem involves assigning values to the cells, and the sub-problem solves for the flow variables to enforce connectivity. Just like the TSP, there is no objective function, so only Benders feasibility cuts are added to fix any cases where the sub-problems are infeasible (*i.e.* a piece is not contiguous).

The master problem is exactly the same as the lazy formulation, as the y variables can also be moved to the sub-problem. Again, we use an IIS of the constraints of the sub-problem to find a feasibility cut. The scenario demonstrated in Figure 4.3 is the same for the Benders decomposition implementation as it is for the lazy formulation. This is not surprising given they have the same master problem and only one lazy constraint was required.

In this scenario, the cell $(0,0)$ is the source for piece 8, and the constraints in the IIS belong to one of two sets: (4.43) for the cells in the 8-piece excluding $(0,0)$ and (4.41) for the neighbours of cells in the 8-piece. This is because none of the cells are connected to the source, and so either one of them needs to become the source, or a connection to a new cell that may allow a connection to a source must be opened. Let P be the cells in one part of the broken piece, and PN be the cells that neighbour cells in P but are not in P themselves. Then the Benders feasibility cut is:

$$\sum_{(i,j)\in P}(1-x_{ijk})+\sum_{(a,b)\in PN}x_{abk}\geq 1, \tag{4.48}$$

that is, either one of the cells in the piece must be turned off, or one of the neighbours turned on. Notice that this is the same as (4.27). This means the lazy formulation is equivalent to a Benders decomposition on the underlying compact formulation. The difference is that in the lazy formulation, the disconnected pieces are found algorithmically (without solving an LP and computing an IIS).

Instance	Time (s)	Nodes	Benders Cuts
1	0.20	0	1
2	0.50	71	0
3	2.68	2842	0
4	0.16	0	0
5	0.28	0	4
6	0.30	0	3
7	0.22	0	0
8	0.64	162	6
9	0.20	0	0

Table 4.4: Time to solution and number of branch-and-bound nodes explored for the nine instances of the Pieces of 8 puzzle using Benders decomposition

Table 4.4 shows the results for the Benders decomposition formulation. It is faster than the compact formulation for every instance except instance 8, but slower than the lazy formulation for all but instances 1 and 8, same as the compact formulation. For this problem, the Benders formulation often explored more nodes than the compact formulation, but it still took less time to solve, again demonstrating the effectiveness of Benders decomposition and Gurobi's advanced techniques.

Constraint programming can also be used to solve the Pieces of 8 puzzle, and may be the preferred method for some. In particular, most problems which have no objective value (only a feasible solution is sought) are good candidates for Constraint programming, and some of the CP techniques may prove to be more effective. Further than this, when the sub-problems are feasibility problems (as they are for the two problems thus presented), CP also makes sense, particularly in a logic-based Benders decomposition framework [26].

4.3 The Fillomino Puzzle

The Fillomino puzzle is a more complex logic puzzle to model compared to the Pieces of 8 puzzle. The puzzle requires the user to enter numbers into a grid, such that the result is a series of *polyominoes* (dominoes for size 2, tetris tiles for size 4 *etc.*). Like the Pieces of 8 puzzle, many of the rules are easy to explain in an integer programming model: every cell must have exactly one number assigned to it, pre-set values must be obeyed. The difficulty comes in enforcing the connectivity of the polyominoes, as there are now multiple of each type, the number of each type is not necessarily known beforehand, and they are not allowed to touch each other.

In the following paper, Pearce and Forbes demonstrate two different methods for solving the Fillomino puzzle: lazy constraints and composite variables. Composite variables refers to a formulation where each variable represents a collection of decisions, and is a form of column generation. In the case of the Fillomino puzzle, it is *a priori* column generation, as we generate all variables before we begin solving the model.

As with the Pieces of 8 puzzle example, the lazy constraints formulation sets out a model that satisfies most of the requirements of the solution, and corrects the solution with additional constraints. The rules that polyominoes must be contiguous and are not allowed to touch are difficult to express beforehand, and to do so would require an exponential number of constraints or a more complicated compact model with auxiliary variables. Instead, we use lazy constraints to invalidate illegal solutions as they appear, until we find a solution where all rules are obeyed, at which point we stop.

The results section shows that the number of constraints required to find the optimal solution is numbered in the hundreds, which is negligible when compared to the size of the set of potential constraints. The straightforward model with lazy connectivity constraints is effective, but the composite variables formulation is more interesting.

Typically in an *a priori* column-generation approach, the formulation is effectively compact, as all constraints of the original problem are present in the composite variables formulation, or have been built into the definition of the variables. In this case, the formulation does not prevent two pieces of

the same type from bordering each other, which would result in a piece of a particular type with twice the allowed number of cells. These solutions are removed using lazy constraints.

As we have seen so far in this chapter, lazy formulations are similar to Benders decomposition. The composite variables formulation is a column-generation approach, which is based on Dantzig-Wolfe decomposition. This means the composite variables formulation in the following section is effectively a Benders decomposition on a Dantzig-Wolfe decomposition. This interesting combination holds much potential for future research, and will be discussed further in Chapter 5.

4.4 Paper: Puzzle - The Fillomino Puzzle

Abstract

Logic puzzles form an excellent set of problems for the teaching of advanced solution techniques in operations research. They are an opportunity for students to test their modelling skills on a different style of problem, and some puzzles even require advanced techniques to become tractable. Fillomino is a puzzle in which the player must enter integers into a grid to satisfy certain rules. This puzzle is a good exercise in using lazy constraints and composite variables to solve difficult problems.

Introduction

Logic puzzles form an excellent set of problems for the teaching of advanced solution techniques in operations research. They are an opportunity for students to test their modelling skills on a different style of problem, and some puzzles even require advanced techniques to become tractable. Puzzles are also typically modelled as integer programs (IP), for example crossword construction [98], Su Doku and the Log Pile puzzle [99], Rummikub [100], the Battleship problem [101] and more.

The use of composite variables can make the solution to some problems much easier to obtain, however they are not widely used. The same can be said for lazy constraints: they are an extremely powerful technique and can yield impressive results for difficult integer and mixed-integer programs, however there are few publications demonstrating the use of lazy constraints. This is perhaps because they are not widely known techniques, and as such should be taught more often in advanced undergraduate operations research courses.

Fillomino is a puzzle whose creation is credited to Nikoli Co., Ltd. [102]. The player must enter integers into a grid to satisfy certain rules. Some cells in the grid have preset values which cannot change. If two cells that share an edge have the same number, they become a tile. If a cell is neighbouring a tile of the same value, it joins the tile. The grid must be filled with numbers such that every cell is assigned a number, and every tile filled with ks has k cells belonging to it. Two tiles of the same number cannot share an edge, since they would merge into one tile which has too many cells. One last assumption that we make is every tile must contain at least one preset value, however there are versions of the puzzle where this is not the case.

The solution to each puzzle is a unique layout of polyominoes ”(shapes made by combining individual squares)”. An example of a starting grid and its unique solution can be seen in Figure 4.4. Every puzzle can be solved logically, and an efficient algorithm exists for solving it in this way [103], however we are interested in solving it using integer programming. This puzzle is an excellent example of the usefulness of composite variables and lazy constraints.

Integer Programming Formulation

We can formulate this puzzle as an integer program and apply lazy constraints to it to find the unique solution to each grid. We require variables representing the entries in each cell of the grid, plus variables recording how many tiles of each type there are. Since there is a unique solution, we do not require an objective function. The formulation we present is as follows:

Sets

N, M	The rows and columns of the grid
K	The range of valid cell entries
$Neigh_{ij}$	The set of cells which share an edge with (i, j)

Data

$Preset_{ij}$	The given value of cell (i, j). 0 implies cell (i, j) is empty

Variables

x_{ijk}	is 1 if cell (i, j) is of type k, 0 otherwise
y_k	is the number of tiles of type k

Constraints

$$\sum_{k \in K} x_{ijk} = 1 \qquad \forall i, j \in N \times M \tag{4.49}$$

$$x_{ijPreset_{ij}} = 1 \qquad \forall i, j \in N \times M | Preset_{ij} \neq 0 \tag{4.50}$$

$$x_{ij1} = 0 \qquad \forall i, j \in N \times M | Preset_{ij} \neq 1 \tag{4.51}$$

$$x_{ijk} \leq \sum_{(a,b) \in Neigh_{ij}} x_{abk} \qquad \forall i, j \in N \times M, \forall k \in K | k > 1 \tag{4.52}$$

$$\sum_{(a,b) \in Neigh_{ij}} x_{ab2} \leq 1 + (|Neigh_{ij}| - 1)(1 - x_{ij2}) \qquad \forall i, j \in N \times M \tag{4.53}$$

$$\sum_{(i,j) \in N \times M} x_{ijk} = k y_k \qquad \forall k \in K \tag{4.54}$$

Constraint (4.49) ensures every cell has exactly one value assigned to it. The next two constraints (4.50-4.51) fix the preset values, and make sure no extra 1s are added. Constraint (4.52) says that a

Figure 4.4: Example of Fillomino starting grid and corresponding solution. Underlined numbers are preset values

	1		1						6		1		1	3
3		7			1				4					
	1	6			6	1	4	4	1		1	8	1	
1		1		6				8				9	5	
			1				8							
	6				5		1	5	5	1	3	3		1
	4	5			1	3		5				1		
1		1		1		1					1			
				3	2		4		5	6		1	9	
1	7	1			1		4			5	1			1

3	1	7	1	7	7	6	6	6	6	8	1	8	1	3
3	7	7	7	7	1	6	6	4	4	8	8	8	8	3
3	1	6	6	6	6	1	4	4	1	8	1	8	1	3
1	6	1	6	6	8	8	8	8	9	9	9	9	5	5
4	6	6	1	8	8	8	8	9	9	9	9	9	5	5
4	6	6	6	5	5	3	1	5	5	1	3	3	5	1
4	4	5	5	5	1	3	3	5	5	5	3	1	9	9
1	7	1	7	1	2	1	6	6	6	6	1	9	9	9
7	7	7	7	3	2	4	4	5	5	6	6	1	9	9
1	7	1	3	3	1	4	4	5	5	5	1	9	9	1

cell can only have a value k if at least one neighbouring cell also has a value k. Since there is only one unique domino (a tile with two cells), we can add a constraint for the case of 2s that says that if a cell is a 2, it has exactly one neighbour which is of type 2, however if it is not a 2 then there are no restrictions on how many neighbouring 2s there are. This is enforced by constraint (4.53). Finally, we know that the number of cells of type k is k times the number of tiles of type k (4.54).

These are the base constraints to generate example solutions to the grid, however there is nothing to prevent tiles of sizes other than k from occurring. One option is to add new variables of a network flow nature, where each preset value has the option of being a sink of size k, and every cell is a source of size 1. This formulation, however, becomes very large very quickly, and easily becomes intractable, even on a 15×15 grid. We can avoid this by introducing lazy constraints.

Use of lazy constraints to enforce tile size

When a potential solution is found by the above implementation, we must check to make sure all tiles are the correct size. To do this, we measure the size of each tile, and if it is not correct, we add one of two lazy constraints.

Bounding tile size from above

Once we have a potential solution, we check the size of each tile. If any tiles are too big, then at least one of the cells in this tile must change its value. Let T be the set of cells in this tile, which are all numbered k^* with $|T| > k^*$. We then add the lazy constraint:

$$\sum_{(i,j)\in T} x_{ijk} \leq |T| - 1 \tag{4.55}$$

We cannot say that the number must be equal to k^*, since this one tile may in fact be two tiles which are joined by one incorrectly numbered cell. Enforcing an equality constraint would make the unique solution invalid and the model would become infeasible. This constraint forbids the current configuration of tiles, which forces the model to try something different. Eventually there will be no more tiles which are larger than they are meant to be. This does not, however, stop them from being smaller than they need to be.

Bounding tile size from below

If there are no tiles that are larger than they are allowed to be in a solution, we then check for tiles that are smaller than required. For each such tile, let T be the set of cells in this tile, which are all numbered k^* and $|T| < k^*$. Also let TN be the set of cells which are a neighbour of at least one cell in T, but are not in T, and whose preset value is 0. If their preset value was k^*, they would already be part of this tile. We then add the lazy constraint:

$$\sum_{(i,j)\in T} x_{ijk^*} - \sum_{(a,b)\in TN} x_{abk^*} \leq |T| - 1 \tag{4.56}$$

This constraint says that either the tile needs to get smaller and perhaps disappear, or some of the neighbours have to change their value to k^*. In other words, it will either remove the tile or pull at least one neighbour into it. This will eventually bound all tiles from below. These two lazy constraints, together with the integer programming formulation above, will find the unique solution to each puzzle, however it may take a long time for the larger grids. We can speed it up using a number of pre-processing techniques.

Pre-processing techniques

Upper bound on number of tiles

Since every tile has to cover at least one preset value, we can calculate an upper bound on the number of tiles of each type. The simplest way would be to count the number of preset cells of type k and use that as an upper bound for y_k, however this fails to take into account the possibility that there may be preset cells of the same type connected to each other. Thus, for every value $k \in K$, we count every connected group of cells of type k and call this number $\mathscr{C}_k$. If $k = 2$, then the number of tiles is exactly equal to $\mathscr{C}_2$, otherwise we add constraints of the form:

$$y_k \leq \mathscr{C}_k, \quad \forall k \in K, k > 1 \tag{4.57}$$

Lower bound on number of tiles

We can also work out a lower bound on the number of tiles of each type since every preset value has to be covered. Because the upper bound on the number of tiles is an equality for type 2, we only need to consider every value $k \in K$ greater than 2. For each of these, we can calculate the geodesic distance of each cell from the nearest preset value of k. The geodesic distance is the length of the shortest valid path of cells from a preset value to the current cell.

For each value $k > 2$, we calculate the geodesic distance from each empty cell to the nearest cell with preset value k. This distance represents the minimum number of cells of type k that must be added to include each cell in a tile of type k. If it is possible for two cells with preset values k to be part of the same tile, there will be two paths, one from each, of length at most $\lfloor \frac{k-1}{2} \rfloor$ which will touch. By removing any cells whose geodesic distance is greater than $\lfloor \frac{k-1}{2} \rfloor$, we can count the number of connected components remaining, which gives us the minimum number of tiles needed to cover all cells with preset values of k. The constraint is of the same form as above, except it will be a greater than or equal to constraint.

Restriction on location of cells

Because a cell can only have a value k if it is part of a tile of type k, and each tile must cover at least one preset value, if a cell has a geodesic distance of k or greater from the nearest preset cell of type k, it can not possibly be part of a tile of type k. We can add constraints to reflect this by following a similar procedure to the other pre-processing techniques. For each value of $k \in K$ greater than 2, we

calculate the geodesic distance from each preset value. Where n preset values of the same type are connected, we know that all cells with a geodesic distance of at most $k-n$ could possibly take the value k. For all cells (i,j) which are not within $k-n$ of any preset value, we add a constraint $x_{ijk}=0$, which removes many unnecessary variables.

Composite variables formulation

Another way of formulating this problem is to consider it as a tiling problem. If we can find every possible placement of tiles of every type, we can choose which combination of tiles gives us the unique solution. We now present our composite variables formulation:

Sets

N, M	The rows and columns of the grid
K	The range of valid cell entries
P	The set of all possible tiles that can be placed in the grid
$P_k \subseteq P$	The set of all possible tiles of type k that can be placed in the grid
$Neigh_{ij}$	The set of cells which share an edge with (i,j)
T_{ij}	The set of tuples (k,p) representing all tiles $p \in P$ which cover cell (i,j)

Data

$Preset_{ij}$	The given value of cell (i,j). 0 implies cell (i,j) is empty

Variables

x_{kp}	is 1 if tile $p \in P_k$ is used, 0 otherwise

Constraints

$$\sum_{(k,p)\in T_{ij}} x_{kp} = 1 \qquad \forall (i,j) \in N \times M, Preset_{ij} \neq 1 \tag{4.58}$$

There is one constraint for every cell of the grid which says that it must be covered by exactly one tile, with the exception of cells that have a 1 in them. Each tile is represented as a (k,p) pair, describing which value of k it covers and which $p \in P_k$ it is. If all possible tile placements are known, this will yield the unique solution to the problem. To find all possible tile placements, we start with the preset cells of each type and grow them outwards by adding neighbours one by one, until they are the correct size, being careful to remove duplicates as we go.

The runtime of this implementation is highly dependent on how quickly you can find all possible tile placements, as the integer program itself solves in a fraction of a second. Another possible concern is that two tiles of the same type may touch in a solution. If this is the case, we can use a modified version of the lazy constraints described above. By following the same procedure, checking each tile to see if any one is larger than k cells, we look at all the cells in this oversized tile and note which tiles $p \in P$ they belong to in the current solution. Where T is the set of tiles in this violation, we add the

Table 4.5: Comparison of runtimes for the Composite Variables and Lazy Constraints implementations. All times are in seconds. The times spent solving the IP and the whole problem are reported separately

	Composite Variables		Lazy Constraints	
Instance	IP	Total	IP	Total
1	0.03	1.54	28.05	28.22
2	0.03	1.75	33.32	33.48
3	0.02	1.13	4.07	4.23
4	0.03	1.34	2.79	3.01

constraint

$$\sum_{(k,p)\in T} x_{kp} \leq |T| - 1 \tag{4.59}$$

This will ensure we find the unique solution.

Results

We tested both implementations using Python 3.7 and the Gurobi [44] solver package. We sourced four randomly-generated puzzles from the Puzzle Baron website [104], all of which are 20×20 and follow our assumptions. Table 4.5 shows that, when implemented efficiently, the composite variables implementation is significantly faster. The majority of time is spent generating the tiles, and once completed, the IP solves in a fraction of a second. For the lazy constraints implementation, less than one second is spent pre-processing the grid and adding initial constraints.

Table 4.6 shows the number of variables and constraints for both implementations, as well as the number of nodes explored and lazy cuts generated for the lazy constraints implementation. For the composite variables implementation, the number of variables reflects the number of potential tiles which can be legally placed in the grid, and the number of constraints is 400 minus the number of squares that contain 1s. This is because there are no options for placing 1s outside the preset locations, so no tiles will cover those squares and as such they do not require constraints.

For the lazy constraints implementation, the number of variables we report here are the number left after Gurobi's presolve stage. Initially, the number is always 3609: $20 \times 20 \times 9$ for the x_{ijk} variables,

Table 4.6: Comparison between the Composite Variables and Lazy Constraints implementations. The number of variables and constraints used in solving the IP, number of lazy constraints added and number of nodes explored in the branch-and-bound tree are shown. The Lazy Cuts column is separated into (Upper,Lower) cuts

	Composite Variables		Lazy Constraints			
Instance	Variables	Constraints	Variables	Constraints	Lazy Cuts	Nodes
1	5219	325	755	6421	342, 545	28769
2	3979	324	536	6570	137, 379	51905
3	4348	326	612	6537	122, 332	6516
4	4842	329	648	6419	74, 335	5460

and 9 for the y_k variables. With our pre-processing, most of those variables will be fixed to 0 (or 1 in the case of cells whose values have been preset), so the number remaining reflects the actual number of possibilities for cell values. Since we know that there are around 325 cells which are not 1, and there are usually 600 variables remaining, this suggests that on average a blank cell only has two choices for which number it could be.

The number of lazy constraints added is also interesting. The number of times tiles have to be bounded from below is always higher than, and usually at least double, the number of times they are bounded from above. This may be because both constraints can be satisfied by moving a cell of the tile to a neighbouring blank cell, thus maintaining the same size of the tile in a different location, however it is much easier for this to occur with tiles that are smaller than needed compared to those which are larger.

Conclusion

This problem is an excellent demonstration of how lazy constraints and composite variables can be used to solve difficult problems, where the naive implementation is intractable. There are many other puzzles and industrial problems which can benefit from similar approaches.

4.5 Discussion of paper: Puzzle - The Fillomino Puzzle

In the lazy formulation for the Fillomino puzzle, the cuts used are the same as (4.27) from the Pieces of 8 puzzle. In fact, one could devise a compact formulation for the Fillomino puzzle similar to that of the Pieces of 8 puzzle, except in this case each polyomino is a distinct piece. This means there may be 30-40 pieces in a single board. The assumption that every piece has at least one pre-set value makes this easier, as these can act as the sources for the pieces.

The difficulties in this case are handling the sources when two pre-set values occupy the same piece, and preventing two pieces of the same number from bordering each other. This will require additional constraints similar to those in the Pieces of 8 puzzle that say cells can only neighbour their own type or the path, except in this case it is that cells can neighbour anything except other cells of the same type but a different piece.

Note that it is also possible to strengthen constraint (4.59) by replacing it with a set of constraints preventing illegal pairs of tiles from both occurring. This only matters if $|T| > 2$, otherwise the constraints are identical. In the case where $|T| = 3$, there may be two or three pairs of tiles illegally sharing borders, and these pairs must be identified, as it may be the case that three tiles are connected in a chain, and removing only the middle tile makes the solution feasible, so the constraint preventing both end tiles from being chosen is an invalid constraint.

The important point is that performing Benders decomposition on this formulation would yield the same cuts as the lazy formulation, just as the Pieces of 8 puzzle does. In fact, the three examples presented in this chapter follow a similar structure: they can be modelled using a lazy formulation, which is equivalent to a Benders decomposition on some underlying compact formulation.

This raises an interesting philosophical question: is it better to find that compact formulation and then apply Benders decomposition to it, or to model it in a lazy fashion from the start? We argue that the latter is better, as it yields not only better computational results, but also savings in implementation time. Rather than searching for a model that covers every aspect of a problem, instead look for the easiest problem to solve that could possibly yield an answer close to the desired one, and correct it as required with lazy constraints. In this case, one needs to know all the rules that integer solutions must follow, how to detect when those rules are violated and how to construct cuts that solve that violation without also removing valid integer solutions, but the alternative is to encode all those rules in the compact formulation to begin with, which is likely more difficult.

While the discussion about feasibility for the Fillomino puzzle is important and necessary for a theoretically-complete model, in practice feasibility issues occur rarely, as the four instances were all solved without any feasibility cuts. This is not to say that these feasibility cuts can be ignored, only that they occur rarely, and so can be handled lazily. This again exemplifies the point that some constraints in a compact model are unnecessary. If one started with the composite variables formulation and ignored the possibility of infeasible solutions until they arose, there is a good chance that they would never need to implement any feasibility cuts, and it is simple to implement them when they are needed.

Chapter 5

Conclusion

> Now this is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning.
>
> Winston Churchill

All examples presented in this thesis benefit from the use of lazy constraints to some degree. The examples also have many similarities in their structures, particularly how they have a number of *unnecessary* constraints, in that if we optimise without these constraints we still solve the underlying problem. When we choose a set of constraints to handle lazily, it is because many (but perhaps not all) constraints in the set are unnecessary. The constraints in the set which are necessary are added as needed. These constraints appear in different forms: they may be directly unnecessary as in the case of the LSFRP and the TSP, or they may be replaced by another, smaller set of constraints as in the case of Benders decomposition. In all cases, the point is to solve the smallest model that will give *reasonable* solutions, and correct those solutions with lazy constraints.

The lazy formulations presented in Chapter 4 are closely related to Benders decomposition. They have some underlying feasibility problem that can be explicitly formulated for the purposes of obtaining Benders feasibility cuts; however, it is also possible to construct the feasibility cuts without the explicit formulation. Conversely, Benders feasibility cuts could be constructed without explicitly computing an IIS or solving a sub-problem, if the form of the infeasibility and how to detect it is understood.

When to be lazy

As stated in the first chapter, there are three categories of models which are likely to benefit from the use of lazy constraints:

1. Models that have exponentially sized set(s) of constraints
2. Models that have sets of constraints that may be mostly or wholly unnecessary
3. Models that are suitable for Benders decomposition

Each chapter of this thesis covered one of these categories, presented examples of such problems and showed the potential benefits from taking a lazy approach. In all cases, the portions of the models that are handled lazily are important but not *structural* for finding feasible solutions. For example, if one tried to solve a network flow model without flow-conservation constraints, the solutions obtained would be useless and far from feasible. However, solving the same model without capacity constraints would produce more reasonable solutions which have a chance of being feasible.

How to be lazy

Implementing branch-and-cut is not difficult, and simply requires the user to learn how to use the features in their solver of choice. The difficult part is deciding what parts of the model to relax and how solutions will be repaired. The most common use of lazy constraints will be for enforcing feasibility after some constraints have been omitted from the formulation. When this occurs, one should strive for the strongest feasibility cuts possible.

As discussed in Section 4.1.2, lifting of feasibility cuts is important and should be considered anywhere they are used. A potential area of future research would be to consider notions of Pareto-optimality of feasibility cuts, where the dominance criterion is defined upon the master problem solutions excluded by the feasibility cuts. Lifting and tightening of cuts increases the number of master problem solutions they exclude, and to prove a feasibility cut is Pareto-optimal would ensure it is the strongest cut possible, *i.e.* one that defines a facet of the convex hull of the set of feasible solutions to the master problem.

Many of the improvements to Benders decomposition discussed, such as warm-starting, user-heuristics and Pareto-optimal Benders cuts, apply to Benders optimality cuts. All lazy formulations presented in Chapter 4 only use lazy constraints to ensure the feasibility of solutions to the lazy formulation, and as such these improvements would not apply. However, it is possible that a problem could be formulated with lazy updates to the objective value in the same style as Benders decomposition, in which case these improvements would carry across, as the approaches are mathematically equivalent.

5.1 Future directions

An exciting new area of research is the combination of the major decomposition methods: Dantzig-Wolfe decomposition and Benders decomposition. Dantzig-Wolfe decomposition works by building a number of constraints into the definition of some composite variables, thus reducing the number of constraints. As mentioned in Section 1.2.2 and Barnhart *et al.* (1998) , the main drawback of Dantzig-Wolfe decomposition is that it can lead to a large number of variables, perhaps too many to explicitly consider. This then requires branch-and-price, which is undesirable as it does not take full advantage of the commercial solvers.

An alternative is to change the composite variables used such that it is possible to generate them *a priori.* This makes it possible to solve without branch-and-price, but there may still be too many constraints. This is where Benders decomposition and/or lazy constraints enter the picture: handle a number of the constraints lazily so that the master problem is small enough to solve.

The composite variables formulation for the Fillomino puzzle is an example of this [1]. To build as many constraints into the variables as possible, each variable would represent the placement of a tile and the values of all cells neighbouring the tile. This way, we can be sure that no two tiles of the same number are adjacent. The problem is that the number of possible neighbour sets for each tile placement is vast, so there would be too many variables to generate *a priori.*

Instead, the formulation in Pearce and Forbes (2017) uses variables that correspond only to the placement of individual tiles. This encodes many but not all constraints of the problem, and the number of variables is now much smaller and it is possible to generate them all *a priori.* There are many constraints required for preventing two tiles of the same type from being adjacent, but handling these constraints lazily reduces the problem size even further to the point where it is very easy to solve.

There is at least one other recent study that explores this idea, which was presented at the Odysseus 2018 conference by Alyasiry, Forbes and Bulmer [105]. In this study, Alyasiry *et al.* considers the Pick-up and Delivery Problem with Time Windows (PDPTW), which is described as follows:

> ... vehicles with limited capacity must be routed to serve given requests each of which consists of a pickup location (origin) and corresponding delivery location (destination). For each request the origin must precede the destination and both must be in the same route. Any route must respect vehicle capacity and allowed time windows at each location, as well as constraints which apply to specific problem variants.

The previous approach to solving this problem by Cherkesly *et al.* (2016) involved generating entire routes for vehicles and choosing from those routes. The benefit to this approach is that many constraints such as respecting the time windows, vehicle capacity and the flow-conservation of vehicles and demands are incorporated into the design of the routes (only feasible routes are generated). This makes for a small master problem with a vast number of variables.

The alternative presented by Alyasiry *et al.*instead generates *fragments* of routes, a series of visits where a truck begins empty and finishes empty, but is never empty in the middle [105]. These fragments can then be connected together to form a complete route for a truck. It is likely that the number of fragments will be much smaller than the number of routes, since the number of ways a small collection of fragments can be chosen, connected and ordered to form a feasible route is large. This leads to far fewer variables in the master problem, to the point where it is possible to generate all fragments *a priori.*

The problem is that not all combinations of fragments are feasible because of the time windows. Each individual fragment may have some flexibility in when it can begin and end, and connections between fragments can be restricted to only those that are possible; however, when three or more fragments are joined together, the whole route may be infeasible. This infeasibility can be resolved using lazy constraints in a branch-and-cut framework, where the smallest infeasible collection of fragments is banned.

This idea of combining a composite variables formulation with lazy constraints represents an enormous opportunity for solving problems for which a “formulation with a huge number of variables

may be the only choice" [22]. Now we may have another choice, where the number of variables and constraints can be balanced to the point where difficult problems become tractable. The key is to get the tightest possible model (with the best LP bound) where all variables are generated *a priori* — but perhaps not all constraints — so that the resulting MIP can be solved using the commercial MIP solvers.

Because of the emerging importance of lazy constraints for solving practical problems, it would also be worth benchmarking the commercial solvers' lazy constraint functionality. Providing a stage for solvers such as Gurobi and CPLEX to show off their efficiency in handling lazy constraints will further improve the performance of models that use lazy constraints. This would be yet another factor compounding the year-to-year improvements in formulations that take full advantage of commercial solvers, ultimately leading to an increase in efficiency in all manner of industrial processes.

Lazy constraint capability represents the foundation of a new suite of tools for solving large modelling problems. Whether used simply to reduce the size of an existing model, or to combine them with an existing technique to provide even larger benefits, there is great potential for problems previously considered intractable to become solvable for many practical purposes in the coming years.

Bibliography

[1] R. H. Pearce and M. A. Forbes. The Fillomino Puzzle. *INFORMS Transactions on Education*, 17(2):85–89, 2017.

[2] R. H. Pearce and M. A. Forbes. Disaggregated Benders decomposition and branch-and-cut for solving the budget-constrained dynamic uncapacitated facility location and network design problem. *European Journal of Operational Research*, 270(1):78–88, 2018.

[3] R. H. Pearce and M. A. Forbes. Disaggregated Benders decomposition for solving a network maintenance scheduling problem. *Journal of the Operational Research Society*, 70(6):941–953, 2018.

[4] R. H. Pearce, A. Tyler, and M. A. Forbes. Extending the MIP toolbox to crack the Liner Shipping Fleet Repositioning problem. In *Biarri Applied Mathematics conference*, 2016.

[5] R. H. Pearce. Solving uncapacitated facility location and network design problems with disaggregated Benders decomposition. In *INFORMS Annual Meeting 2017*, 2016.

[6] Clarivate Analytics. Web of science. Online Resource, 2019. http://apps.webofknowledge.com/.

[7] INFORMS. Website, accessed 2018. www.informs.org.

[8] J. B. J. Fourier. Solution d'une question particulière du calcul des inégalités. *Oeuvres II*, 1826.

[9] G. B. Dantzig. Origins of the simplex method. Technical report, Systems Optimization Laboratory, 1987.

[10] R. E. Bixby. A brief history of linear and mixed-integer programming computation. *Documenta Mathematica*, Extra Volume: Optimization Stories:107–121, 2012.

[11] H. M. Wagner. A comparison of the original and revised simplex methods. *Operations Research*, 5(3):361–369, 1957.

[12] W. L. Winston and J. B. Goldberg. *Operations research: applications and algorithms*. Thomson/Brooks/Cole, 4th edition, 2004.

[13] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer Series in Operation Research and Financial Engineering. Springer Science and Business Media, 2nd edition, 2006.

[14] G. B. Dantzig and M. N. Thapa. *Linear programming 2: Theory and Extensions*. Springer Series in Operations Research. Springer, 2003.

[15] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984.

[16] Gurobi Optimization Inc. Linear programming basics, 2015.

[17] R. Gomory. INFORMS history and traditions interview with Ralph Gomory. Interview by Irv Lustig, 2017. `https://www.informs.org/Explore/History-of-O.R.-Excellence/Biographical-Profiles/Gomory-Ralph-E#oral_hist`.

[18] R. Gomory. Outline of an algorithm for integer solutions to linear programs. *Research announcements at Princeton University*, 1958.

[19] R. Gomory. An algorithm for the mixed-integer problem. *Rand Corporation*, 1960.

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

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

[22] C. Barnhart, E. L. Johnson, G. L. Nemhauser, and M. W. P. Savelsbergh. Branch-and-price: Column Generation for solving huge integer programs. *Operations Research*, 46(3):316–329, 1998.

[23] A. Gleixner, M. Bastubbe, L. Eifler, T. Gally, G. Gamrath, R. L. Gottwald, G. Hendel, C. Hojny, T. Koch, M. E. Lübbecke, S. J. Maher, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, F. Schlösser, C. Schubert, F. Serrano, Y. Shinano, J. M. Viernickel, M. Walter, F. Wegscheider, J. T. Witt, and J. Witzig. The SCIP Optimization Suite 6.0. Technical report, Optimization Online, July 2018.

[24] J. F. Benders. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, 4:238–252, 1962.

[25] G. Ottosson. *Integration of Constraint Programming and Integer Programming for Combinatorial Optimization*. PhD thesis, Uppsala University, 2000.

[26] J. N. Hooker and G. Ottosson. Logic-based Benders decomposition. *Mathematical Programming, Series A*, 96:33–60, 2003.

[27] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton Series in Applied Mathematics. Princeton Press, 2007.

[28] R. Bixby. Optimization: past, present, future. In *INFORMS Annual meeting*, 2017.

[29] H. Mittelmann. Benchmarks for optimization software. http://plato.asu.edu/bench.html. Accessed 2018.

[30] G. B. Dantzig, D. R. Fulkerson, and S. M. Johnson. Solution of a large-scale traveling-salesman problem. *Operations Research*, 2:393–410, 1954.

[31] G. B. Dantzig, D. R. Fulkerson, and S. M. Johnson. On a linear-programming, combinatorial approach to the traveling-salesman problem. *Operations Research*, 7:58–66, 1959.

[32] M. Grötschel and Y. Wakabayashi. A cutting plane algorithm for a clustering problem. *Mathematical Programming*, 45:59–96, 1989.

[33] G. L. Nemhauser and G. Sigismondi. A strong cutting plane/branch-and-bound algorithm for node packing. *Journal of the Operational Research Society*, 43(5):443–457, 1992.

[34] J. R. Araque G., G. Kudva, T. L. Morin, and J. F. Pekny. A branch-and-cut algorithm for vehicle routing problems. *Annals of Operations Research*, 50:37–59, 1994.

[35] M. Grötschel and O. Holland. Solution of large-scale symmetric travelling salesman problems. *Mathematical Programming*, 51:141–202, 1991.

[36] M. Jünger and S. Thienel. The abacus system for branch-and-cut-and-price algorithms in integer programming and combinatorial optimization. *Softw. Pract. Exper.*, 30(11):1325–1352, 2000.

[37] T. Achterberg. *Constraint Integer Programming*. PhD thesis, Technische Universität Berlin, 2007.

[38] K. Tierney, A. J. Coles, A. I. Coles, C. Kroer, A. M. Britt, and R. M. Jensen. Automated planning for liner shipping fleet repositioning. In *Proceedings of the 22nd International Conference on Automated Planning and Scheduling*, pages 279–287, 2012.

[39] K. Tierney and R. M. Jensen. The liner shipping fleet repositioning problem with cargo flows. In H. Hu, X. Shi, R. Stahlbock, and S. Voß, editors, *Computational Logistics*, volume 7555 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin Heidelberg, 2012.

[40] E. Kelareva, K. Tierney, and P. Kilby. CP methods for scheduling and routing with time-dependent task costs. *EURO Journal on Computational Optimization*, 2(3):147–194, 2014.

[41] K. Tierney, B. Áskelsdóttir, R. M. Jensen, and D. Pisinger. Solving the liner shipping fleet repositioning problem with cargo flows. *Transportation Science*, 49(3):652–674, 2015.

[42] K. Tierney. *Optimizing Liner Shipping Fleet Repositioning Plans*, volume 57 of *Operations Research/Computer Science Interfaces Series*. Springer, 2015.

[43] A. Tyler. Solving the liner shipping fleet repositioning problem with cargo flows. Honours, School of Mathematics and Physics, University of Queensland, 2015.

[44] Gurobi Optimization Inc. Gurobi Optimizer Reference Manual, 2016.

[45] A. M. Geoffrion. Generalized Benders decomposition. *Journal of Optimization Theory and Applications*, 10:237–260, 1972.

[46] T. L. Magnanti and R. T. Wong. Accelerating Benders decomposition: Algorithmic enhancement and model selection criteria. *Operations Research*, 29(3):464–484, 1981.

[47] T. L. Magnanti, P. Mireault, and R. T. Wong. Tailoring Benders decomposition for uncapacitated network design. *Mathematical Programming Study*, 26:112–154, 1986.

[48] T. L. Magnanti and R. T. Wong. Network design and transportation planning: Models and algorithms. *Transportation Science*, 18(1):1–55, 1984.

[49] N. Papadakos. Practical enhancements to the Magnanti-Wong method. *Operations Research Letters*, 36:444–449, 2008.

[50] A. M. Costa. A survey on Benders decomposition applied to fixed-charge network design problems. *Computers and Operations Research*, 32:1429–1450, 2005.

[51] R. Rahmaniani, T. G. Crainic, M. Gendreau, and W. Rei. The Benders decomposition algorithm: A literature review. *European Journal of Operational Research*, 259:801–817, 2017.

[52] E. de Sá, R. de Camargo, and G. de Miranda. An improved Benders decomposition algorithm for the tree of hubs location problem. *European Journal of Operational Research*, 226:185–202, 2013.

[53] E. M. de Sá, I. Contreras, J. F. Cordeau, R. S. de Camargo, and G. de Miranda. The hub line location problem. *Transportation Science*, 49(3):500–518, 2015.

[54] F. Errico, T. G. Crainic, F. Malucelli, and M. Nonato. A Benders decomposition approach for the symmetric TSP with generalized latency arising in the design of semiflexible transit systems. *Transportation Science*, 51(2):706–722, 2017.

[55] M. Fischetti, D. Salvagnin, and A. Zanette. A note on the selection of Benders' cuts. *Mathematical Programming*, 124:175–182, 2010.

[56] G. Codato and M. Fischetti. Combinatorial Benders' cuts for mixed-integer linear programming. *Operations Research*, 54(4):756–766, 2006.

[57] M. Fischetti. Modern Benders. In *Conference on the Mathematics of Operations Research 42*, Lunteren (The Netherlands), January 2017. DOI: 10.13140/RG.2.2.24273.94564.

[58] A. M. Geoffrion and G. W. Graves. Multicommodity distribution system design by Benders decomposition. *Management Science*, 20(5):822–844, 1974.

[59] Y. Adulyasak, J. Cordeau, and R. Jans. Benders decomposition for production routing under demand uncertainty. *Operations Research*, 63(4):851–867, 2015.

[60] M. Fischetti, I. Ljubić, and M. Sinnl. Redesigning Benders decomposition for large-scale facility location. *Management Science*, 63(7):2146–2162, 2017.

[61] M. L. Balinski and P. Wolfe. On Benders decomposition and a plant location problem. *Mathematica Working Paper*, 1963.

[62] M. L. Balinski. Integer programming: Methods, uses, computation. *Management Science*, 12(3):253–313, 1965.

[63] D. McDaniel and M. Devine. A Modified Benders' Partitioning Algorithm for Mixed Integer Programming. *Management Science*, 24(3):312–319, 1977.

[64] M. Hoefer. Ufllib. url: http://resources.mpi-inf.mpg.de/departments/d1/projects/benchmarks/UflLib/index.htm

[65] L. Tang, W. Jiang, and G. K. D. Saharidis. An improved Benders decomposition algorithm for the logistics facility location problem with capacity expansions. *Annals of Operations Research*, 210:165–190, 2013.

[66] O. Arslan and O. E. Karaşan. A Benders decomposition approach for the charging station location problem with plug-in hybrid electric vehicles. *Transportation Research B*, 93:670–695, 2016.

[67] A. Mahéo, P. Kilby, and P. Van Hentenryck. Benders decomposition for the design of a hub and shuttle public transit system. *Transportation Science*, 2017. Available in Articles in Advance.

[68] N. Boland, T. Kalinowski, H. Waterer, and L. Zheng. Scheduling arc maintenance jobs in a network to maximize total flow over time. *Discrete Applied Mathematics*, 163:34–52, 2014.

[69] S. G. Nurre, B. Cavdaroglu, J. E. Mitchell, T. C. Sharkey, and W. A. Wallace. Restoring infrastructure sytems: An integrated network design and scheduling (INDS) problem. *European Journal of Operational Research*, 223:794–806, 2012.

[70] R. S. de Camargo, G. Miranda Jr., and H. P. Luna. Benders decomposition for the uncapacitated multiple allocation hub location problem. *Computers and Operations Research*, 35:1047–1064, 2008.

[71] R. Lusby, L. F. Muller, and B. Petersen. A solution approach based on Benders decomposition for the preventive maintenance scheduling problem of a stochastic large-scale energy system. *Journal of Scheduling*, 16:605–628, 2013.

[72] B. Fortz and M. Poss. An improved Benders decomposition applied to a multi-layer network design problem. *Operations Research Letters*, 37:359–364, 2009.

[73] P. Elias, A. Feinstein, and C. E. Shannon. A note on the maximum flow through a network. *IRE Transactions on Information Theory*, IT-2:117–119, 1956.

[74] L. R. Ford Jr. and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[75] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, 1976.

[76] J. Cordeau, F. Soumis, and J. Desrosiers. A Benders decomposition approach for the locomotive and car assignment problem. *Transportation Science*, 34(2):133–149, 2000.

[77] A. Weber. *Ueber den Standort der Industrien*. University of Chicago Press, 1909.

[78] J. Castro, S. Nasini, and F. S. da Gama. A cutting-plane approach for large scale capacitated multi-period facility location using a specialized interior-point method. *Mathematical Programming*, 163(1):411–444, 2017.

[79] S. D. Jena, J. Cordeau, and B. Gendron. Solving a dynamic facility location problem with partial closing and reopening. *Computers and Operations Research*, 67:143–154, 2017.

[80] I. Contreras and E. Fernández. Hub location as the minimization of a supermodular set function. *Operations Research*, 62(3):557–570, 2014.

[81] G. Laporte, S. Nickel, and F. S. da Gama. *Location Science*. Springer International, 2015.

[82] M. Fischetti, I. Ljubić, and M. Sinnl. Benders decomposition without separability: A computational study for capacitated facility location problems. *European Journal of Operational Research*, 253(3):557 – 569, 2016.

[83] A. Ghaderi and M. S. Jabalameli. Modeling the budget-constrained dynamic uncapacitated facility location-network design problem and solving it via two efficient heuristics: A case study of health care. *Mathematical and Computer Modelling*, 57:382–400, 2013.

[84] Z. Gu, G. L. Nemhauser, and M. W. P. Savelsbergh. Lifted cover inequalities for 0-1 integer programs: Computation. *INFORMS Journal on Computing*, 10(4):427–437, 1998.

[85] B. Gendron, P. Khuong, and F. Semet. A lagrangian-based branch-and-bound algorithm for the two-level uncapacitated facility location problem with single-assignment constraints. *Transportation Science*, 50(4):1286–1299, 2016.

[86] A. G. Marín and P. Jaramillo. Urban rapid transit network design: accelerated Benders decomposition. *Annals of Operations Research*, 169:35–53, 2009.

[87] L. B. Real, M. O'Kelly, G. de Miranda, and R. S. de Camargo. The gateway hub location problem. *Journal of Air Transport Management*, 73:95–112, 2018.

[88] J. F. Bigotte, D. Krass, A. P. Antunes, and O. Berman. Integrated modeling of urban heirarchy and transportation network planning. *Transportation Research A*, 44:506–522, 2010.

[89] I. Contreras, E. Fernández, and A. Marín. Tight bounds from a path based formulation for the tree of hub location problem. *Computers and Operations Research*, 36:3117–3127, 2009.

[90] L. Kaufman, M. V. Eede, and P. Hansen. A plant and warehouse location problem. *Operational Research Quarterly*, 28(3):547–554, 1977.

[91] Y. Rist. Two-level facility location. Honours, School of Mathematics and Physics, University of Queensland, 2018.

[92] University of Waterloo. World traveling salesman problem. Online Resource, March 2018. http://www.math.uwaterloo.ca/tsp/world/.

[93] Gurobi Optimization Inc. The Travelling Salesman Problem with Integer Programming and Gurobi, 2015.

[94] U. of Waterloo. Concorde TSP solver. Website, 2015.

[95] C. E. Miller, A. W. Tucker, and R. A. Zemlin. Integer programming formulations of traveling salesman problems. *Journal of the Association for Computing Machinery*, 7(4):326–329, 1960.

[96] B. Gavish and S. C. Graves. The Travelling Salesman Problem and related problems. Working paper or-078-78, Operations Research Center, MIT, 1978.

[97] C. Plover. Anne Bonney. MUMS Puzzle Hunt, 2011. http://researchers.ms.unimelb.edu.au/ mums/ puzzlehunt/2011/puzzles/Act1.pdf.

[98] J. M. Wilson. Crossword compilation using integer programming. *The Computer Journal*, 32:273–275, 1989.

[99] M. J. Chlond. Classroom exercises in IP modeling: Su Doku and the log pile. *Transactions on Education*, 5(2):77–79, 2005.

[100] D. den Hertog and P. B. Hulshof. Solving Rummikub Problems by Integer Linear Programming. *The Computer Journal*, 49(6):665–669, 2006.

[101] W. J. M. Meuffels and D. den Hertog. Solving the battleship puzzle as an integer programming problem. *Transactions on Education*, 10(3):156–162, 2010.

[102] Nikoli Co. Ltd. Database of Japanese Puzzles, 2008.

[103] S. J. Yen, T. C. Su, and S. C. Hsu. An efficient algorithm for solving Fillomino. In *Fuzzy Systems (FUZZ), 2011 IEEE International Conference on*, pages 190–194, June 2011.

[104] Puzzle Baron. Fillomino by Puzzle Baron, 2016. Accessed 2016.

[105] A. M. Alyasiry, M. Forbes, and M. Bulmer. An exact algorithm for the pickup and delivery problem with time windows and its variants. In *Odysseus 2018*, 2018.

[106] M. Cherkesly, G. Desaulniers, S. Irnich, and G. Laporte. Branch-price-and-cut algorithms for the pickup and delivery problem with time windows and multiple stacks. *European Journal of Operational Research*, 250(3):782–793, 2016.

Only that which is necessary, and only when it is necessary