BIG-DATA DRIVEN OPTIMIZATION METHODS WITH APPLICATIONS TO LTL-FREIGHT ROUTING

BIG-DATA DRIVEN OPTIMIZATION METHODS WITH APPLICATIONS TO LTL-FREIGHT ROUTING

BY

Srinivas Subramanya Tamvada, M.E.

A THESIS

SUBMITTED TO THE SCHOOL OF COMPUTATIONAL SCIENCE AND ENGINEERING

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Doctor of Philosophy (2020) McMaster University

(Computational Science and Engineering) Hamilton, Ontario, Canada

TITLE:	Big-Data Driven Optimization Methods with Applications to LTL-Freight Routing
AUTHOR:	Srinivas Subramanya Tamvada M.E.
SUPERVISOR:	Dr. Elkafi Hassini
NUMBER OF PAGES:	xviii, 155

Lay Abstract

Less-than-truckload (LTL) freight transportation is a vital part of Canada's economy, with revenues running into billions of dollars and a cascading impact on many other industries. LTL operators often have to deal with large volumes of shipments, unexpected changes in traffic conditions, and uncertainty in demand patterns. In an industry that already has low profit margins, it is therefore vitally important to make good routing decisions without expending a lot of time.

The optimization of such LTL freight networks often results in complex big-data driven optimization problems. In addition to the challenge of finding optimal solutions for these problems, analysts often have to deal with the complexities of big-data driven inputs. In this thesis we develop several solution strategies for solving the LTL freight routing problem including an exact model, novel heuristics, and techniques for solving the problem efficiently on a cluster of computers.

Although the techniques we develop are inspired by LTL routing, they are more generally applicable for solving big-data driven optimization problems from other domains. Experiments conducted over the years in consultation with industry experts indicate that our proposals can significantly improve solution quality and reduce time to solution. Furthermore, our proposals open up interesting avenues for future research.

Abstract

We propose solution strategies for hard Mixed Integer Programming (MIP) problems, with a focus on distributed parallel MIP optimization. Although our proposals are inspired by the Less-than-truckload (LTL) freight routing problem, they are more generally applicable to hard MIPs from other domains. We start by developing an Integer Programming model for the Less-than-truckload (LTL) freight routing problem, and present a novel heuristic for solving the model in a reasonable amount of time on large LTL networks. Next, we identify some adaptations to MIP branching strategies that are useful for achieving improved scaling upon distribution when the LTL routing problem (or other hard MIPs) are solved using parallel MIP optimization.

Recognizing that our model represents a pseudo-Boolean optimization problem (PBO), we leverage solution techniques used by PBO solvers to develop a CPLEX based look-ahead solver for LTL routing and other PBO problems. Our focus once again is on achieving improved scaling upon distribution. We also analyze a technique for implementing subtree parallelism during distributed MIP optimization. We believe that our proposals represent a significant step towards solving big-data driven optimization problems (such as the LTL routing problem) in a more efficient manner.

Dedicated to his holiness Sri Shirdi Sai Baba, and to the memory of my mother Tamvada Krishnaveni

Acknowledgements

First of all I would like to thank my supervisor Dr. Elkafi Hassini for giving me the opportunity to join his group, and for his guidance throughout the course of this study. Not only did he give me interesting projects, he also gave me complete freedom for exploring them. His biggest virtue is his patience, which was required in good measure given all the mistakes I made along the way.

Dr. Ed Klotz of IBM CPLEX has been very generous to us with his time. His role has been wonderfully complementary to Dr. Hassini's supervisory role. Practically everything I know about CPLEX programming today, I owe it to Ed Klotz.

My classmate Dr. Bahareh Mansouri was very helpful with the project work. It was Bahareh's preliminary work with our industry partner (a major Canadian LTL operator) that set the stage for this thesis. We are very grateful to our industry partner for helping us understand their business, and for answering all our questions in a timely manner in spite of their own busy schedules.

I would like to thank my father Dr. T.V.S. Ramamohan Rao, and professors Dr. S.S. Prabhu, Dr. Raghuvir Sharan, Dr. B.G. Fernandes, and Dr. Sanjeev Swami of I.I.T. Kanpur for their guidance throughout my academic journey. No less valuable was the guidance I got from my countless teachers since my kindergarten days. Some names that come to mind are Mr. R.K.Tewari, Mr. Khalil Khan, Ms. Neera Negi,

Ms. Anjula Chauhan, Siddiqui sir, Ms. Gujral, Pachori sir, the late Shri Prem Kumar Shukl and Chaudari sir (senior), my mentor K.M. Krishna, and my kindergarten teachers Mrs. Nigam and Shobha madam.

A big thumbs up to all my classmates over the years, starting from my childhood best friend (late) Vikram "Vinku" Utpreti, Tony Abraham, Sanghmitra, Shiv Shankar, Santosh Upadhyay, Irfan Siddiqui, Mayank Verma, Atul Kumar Shrivastav, Abhishek Sharma, Ishvinder Singh, Manoj Rai, Paritosh Tyagi, Anup Agnihotri, Vikram Hardiker, and other members of our famous (or infamous) Sunday League Cricket Club.

And finally, I would like to thank my wife Naga-Madhuri and daughter Ishani Surya Prasanna. Without their support, this thesis would have been completed in the proverbial half the time.

Declaration of academic achievement

I, Srinivas Subramanya Tamvada, declare this thesis to be my own work. I am the sole author of this document with exception to those chapters included as works published, submitted, or accepted for publication in research journals in which case authorship, credit, and copyright are duly noted with respect to each such included item.

As this thesis contains materials published, submitted, or accepted for publication in journals, all steps have been taken to ensure that the necessary copyright limitations and rights have been respected. My supervisor Dr. Elkafi Hassini and supervisory committee members Drs. Kai Huang and Frantisek Franek have provided guidance to me throughout my Ph.D. I have completed all the research work included within this thesis.

Contents

List of Figures

List of Tables

List of Abbreviations

Abbreviations

API	Application Programming Interface
BCP	Boolean Constraint Propagation
CDCL	Conflict Driven Clause Learning
CFG	Characteristic Function Game
DLIS	Dynamic Largest Individual Sum
EOL	End of Line
GRASP	Greedy Randomized Adaptive Search Procedure
IP	Integer Program
LCA	Lowest Common Ancestor
LTL	Less-than-truckload
MIP	Mixed Integer Program

MOMS	Maximum Occurrence in clauses of Minimum Size
MPI	Message Passing Interface
PBO	Pseudo Boolean Optimization
PFG	Partition Function Game
SAT	Satisfiability Proposition
SCIP	Solving Constraint Integer Programs

Chapter 1

Introduction

The advent of the internet has lead to an increased interest in the field of big-data driven optimization. Broadly defined, big-data driven optimization deals with the formulation and solution of optimization problems of unprecedented sizes and complexity, based on information collected from various sources (Emrouznejad, 2016). Big-data driven optimization is now being applied in many important disciplines such as Computer science, Operations research, Agricultural engineering, Psychology, Medicine and Biochemistry to name a few.

The popularity of big-data driven decision making and the scientific possibilities it opens up have led to new opportunities and challenges for practitioners and researchers alike. Practitioners are focused on collecting good quality data from their operations and applying the insights gained from it to streamline their business (McAfee *et al.*, 2012). Researchers on the other hand have been preoccupied with developing tools and technologies for analyzing the data and translating it into actionable insights (Saggi and Jain, 2018). To this end, big-data analytics platforms such as Apache Hadoop and Apache Spark (Zaharia *et al.*, 2010) have been integrated with

powerful optimization engines in order to solve machine learning problems more efficiently (Bosagh Zadeh *et al.*, 2016; Cao and Sun, 2016; Sagratella, 2016).

Our work on this thesis started with the long term goal of integrating IBM's state-of-the-art MIP solver *CPLEX* (CPLEX, 2007) into Apache Spark. Soon after that, we discovered that MIP solver parallelization was an active area of research with many interesting and unanswered questions. In fact, efficient MIP solver parallelization is a precursor for integrating any optimization engine into a big-data platform. Around the same time, we also started our collaboration with a major Canadian Logistics and Freight transportation company. Therefore it was natural choice to select a big-data driven optimization problem from the Freight transportation industry as our use case in this thesis.

The transportation problem we study in this thesis is called the *Less-than-truckload (LTL)* route optimization problem (Özener, 2019; Hejazi and Haghani, 2007). This routing problem is a pseudo-Boolean optimization problem (Eén and Sorensson, 2006), and has been studied extensively over the years given its impact on the freight transportation industry. It exhibits the classic $4\,V$*'s* of big-data, namely:

1. Volume: for LTL networks that transport large volumes of freight, the sheer size of the problem can be too large to handle on a single computer.

2. Variety: there are different formats in which freight accepted for shipment can be encoded. In some cases, manual intervention is required to ingest freight into the system.

3. Veracity: owning partly to manual intervention and partly to the lack of adherence to strict standards, some records can be incomplete or damaged.

4. Velocity: last but not least, the rate at which freight records are ingested and the rate at which routing decisions are made can both be quite high.

In addition, LTL operators also have to deal with unexpected changes in traffic conditions and uncertain demand patterns, sometimes making it necessary to make (or alter) routing decisions in real-time. Given these characteristics and its importance to the industry, the LTL routing problem is a good example of a big-data driven optimization problem. Moreover, the problem is suitable for solving in a distributed fashion given its size and complexity, and represents an important class of MIPs called pseudo-Boolean optimization problems (PBOs). In this thesis, we develop computational techniques for solving the LTL routing problem that are also applicable more generally.

1.1 Motivation and goals

Our primary motivation in this thesis is to take a step towards integrating big-data processing platforms (such as Apache Spark) with powerful optimization engines (such as CPLEX) by exploring effective techniques for MIP solver parallelization. Optimization problems from the industry are often big-data driven (an example is the aforementioned LTL routing problem). Moreover, the prescriptive stage of big-data analytics (Delen and Ram, 2018) includes optimization and requires the most computational effort. Therefore, it is beneficial to move towards a single, unified platform both for solving large scale optimization problems and for processing big-data.

Since the turn of the century, online retailers and marketplaces such as Amazon, EBay, and Alibaba are cutting into the market share of Brick-and-mortar businesses.

Indeed, most large businesses such as Walmart, BestBuy, Ikea, and many others now have a strong online presence (Bernstein *et al.*, 2008). This has lead to an increased burden on the freight transportation industry, with products ordered online adding to conventional freight volume. Companies such as Amazon have even experimented with designing their own freight transportation solutions, or have long term contracts with LTL providers like our industry partner.

These changes, coupled with advances in technology, have led to businesses demanding faster and better solutions to the increasingly larger problems they face in their day-to-day operations. The LTL routing problem is but one example of an operational decision that is made everyday in the industry.

In this thesis, our goal is to develop both conventional and unconventional solutions for the LTL routing problem. The conventional solutions are based on exact models and hybrid heuristics. The unconventional solutions focus on solving the LTL routing problem (and indeed other optimization problems) in a distributed fashion. *In our view, effective MIP solver parallelization is a vital first step towards integrating them with big-data platforms.*

1.2 Organization of the thesis

Chapter 2 begins with a brief description of the LTL freight transportation industry. We present a literature review in this area of research as it pertains to our work. This is followed by our model for routing LTL freight and novel heuristics for solving the model. Results from experiments conducted in partnership with a major Canadian LTL operator are presented at the end of the chapter.

Recognizing that our model is hard to solve, Chapter 3 identifies some properties

of popular MIP branching strategies that can be leveraged and adapted for increased speedups upon distribution (*i.e.*, for improved scaling when such models are solved in a distributed fashion). This chapter also sets the stage for the next two chapters which expand on some of the proposals in the chapter.

Chapter 4 describes a specialized solver we developed for the LTL routing problem. This solver incorporates branching heuristics from Boolean satisfiability problems (SAT) (Marques-Silva, 1999) into a popular commercial MIP solver, and can be used to solve other Pseudo-Boolean optimization problems (PBOs) (Eén and Sorensson, 2006). While the integration of SAT and MIP solution techniques is an active area of research, our focus is on implementing a solver that can be parallelized more efficiently.

Chapter 5 extends an existing technique for implementing subtree parallelism in distributed MIP solver implementations. We show that for some hard MIPs, it is beneficial to implement subtree parallelism using this technique rather than to iterate through a collection of disjoint subproblems. The technique is evaluated by integrating it into a commercial MIP solver.

Chapter 6 summarizes our achievements and results, and concludes the thesis by discussing some opportunities for future research in addition to those mentioned at the end of each chapter.

1.3 Contributions

Both LTL freight routing and MIP solver parallelization are active areas of research. In this thesis, we have extended the body of literature in these areas of research as follows:

- We have developed an integer programming model for routing LTL freight. While similar models exist in literature, we are the first to use our model for finding a minimum cost directed Steiner-forest in a time-space network. This minimum cost forest is subsequently used for developing a heuristic for routing LTL freight.

- We show how search restarts can be applied to MIP solver parallelization by constructing and using a central repository of variable pseudo-costs. To the best of our knowledge, we are the first to apply search restarts in this manner for MIP solver parallelization.

- We incorporate branching heuristics used for solving pseudo-Boolean optimization problems into CPLEX, and show that this approach can be effective for solving MIPs. This approach is based on our observation that branching strategies can scale well upon distribution if they only use properties of the branching node. We also describe the notion of trigger equivalence and domination for Boolean Constraint Propagation (BCP), which we found to be useful for speeding up the BCP process.

- We analyze a technique for implementing subtree parallelism during MIP solver parallelization. While this technique has been used in existing implementations, we prove that it can be used to merge an arbitrary collection of search tree leaf nodes by solving the smallest number of node relaxations. We also show how this technique can be incorporated into a commercial MIP solver (CPLEX), and used for dynamic load balancing.

Chapter 2

An Integer Programming model and heuristics for LTL routing

2.1 Introduction

Less Than Truckload (LTL) freight transportation is a multi-billion dollar industry, with its operations having a significant impact on other industries (Hernández *et al.*, 2011; Özkaya *et al.*, 2010). The principal task of *LTL operators* is to collect freight from customers for delivery to the intended recipients within a specified *timeline* (*i.e.,* within a *service contract*, or at a *service level*). Normally, freight is picked up from the customers using small trucks and delivered to a regional *End-of-line (EOL)* terminal. Freight from EOL terminals is carried by line-haul trucks to the nearest *break-bulk terminal* (Katayama and Yurimoto, 2016; Powell and Sheffi, 1989). At each break-bulk, freight is *consolidated* (*i.e.*, sorted and reloaded) into *trailers* for transportation to an adjacent break-bulk (*i.e.*, to a break-bulk to which *direct service* is available). Freight may travel through several intermediate break-bulks before eventually arriving

at its destination EOL terminal, from where it is dispatched in small trucks to the respective consignees. This last distribution operation is often referred to as *last-mile delivery*. In countries like Canada and Spain, break-bulks also serve as EOL terminals (Barcos *et al.*, 2010).

Typically, the volume of freight travelling from a break-bulk to a destination EOL terminal is less than the volume capacity of a standard trailer (hence the name, "Less-than-truckload"). Consolidation of freight at the break-bulks is required in order to properly utilize trailer capacity, since dedicating an entire trailer for each individual shipment is not cost effective. LTL networks are carefully designed to allow for effective consolidation of freight at each break-bulk. Our goal in this chapter is to minimize the cost of routing LTL freight given an operational network and a service contract. Note that trailers that are fully packed with freight (all of it bound for the same destination) are not considered part of the LTL problem because such trailers do not participate in freight consolidation.

Each indivisible unit of shipment at each break-bulk is called a "*skid*". A *skid* is a wooden container marked with its weight and volume, a unique "pro-bill" number, a destination, and a delivery deadline. Skids can differ in weight and volume but are much smaller in size than the standard trailer capacity, and need to be packed into trailers rented from third-party *carriers*. LTL operators often have long-term contracts with these third-party carriers to guarantee the availability of trailers on popular routes at reasonable prices. In addition to trailer rental costs, there are some *handling costs* associated with packing skids into trailers at each break-bulk.

To summarize, *the essence of the LTL routing problem is to pack skids at each*

break-bulk into trailers to minimize transportation and handling costs, while ensuring that the route chosen for each skid meets its delivery deadline. In addition, it is desirable that the average trailer capacity utilization exceeds a certain threshold. More detailed descriptions of the design and operation of LTL networks can be found in Crainic (Crainic *et al.*, 1998), Erera *et al.* (Erera *et al.*, 2013b), and Powell and Sheffi (Powell and Sheffi, 1983).

In this chapter, we present a novel integer programming formulation for the LTL freight routing problem. The key difference between our approach and most existing approaches is that we do not restrict skids having the same origin and destination to travel along the same route. Such restrictions are commonly used for routing freight on large LTL networks, and are implemented using *load plans* as described later in Section 2.2. Our model can easily be adapted to handle larger LTL routing problems, and we use it as a foundation for developing a hybrid heuristic which is based on the notion of a *minimum cost directed Steiner-forest* (Feldman *et al.*, 2012). To the best of our knowledge, we are the first to extend the notion of a directed Steiner-forest to a *time-space network* and use it for routing LTL freight. Time-space networks are described in some excellent papers on LTL routing (Erera *et al.*, 2013a; Kennington and Nicholson, 2010).

The rest of the chapter is organized as follows. We start with a brief survey of existing literature in Section 2.2. Our model and heuristics appear in Section 2.3. Experimental results are presented in Section 2.4. We conclude the chapter and propose future research directions in Section 2.5.

2.2 Literature Review

Crainic (Crainic *et al.*, 1998) present an excellent survey of all aspects of the LTL freight scheduling problem, starting from network design and strategic issues to operational decisions such as routing (which is our focus). The survey describes some typical problems facing LTL routing such as the classic "back-haul problem", the unpredictability of future demand, and the need for making routing decisions in real time. Of these, the back-haul problem can be understood as the accumulation of trailers at a location because of large amounts of incoming freight and relatively smaller amounts of outgoing freight, which can necessitate return trips with empty trailers ("empties"). Juan *et al.* (Juan *et al.*, 2014) note that back-hauls represent 25% of road transportation activities in Europe. Both Juan *et al.* (Juan *et al.*, 2014) and Bailey *et al.* (Bailey *et al.*, 2011) propose approaches based on cooperation between carriers for addressing the back-haul problem.

We note that back-hauls are a concern for third-party carriers, and not for LTL operators like our industry partner. Most third party carriers already factor in back-hauls into the contracted rates for trailers offered to LTL operators. For this reason, we do not explicitly address the back-haul problem, although it can easily be incorporated into our model by making back-haul trailers available at a discounted price (in addition to the regular trailers that are available at the contracted price). Powell and Sheffi (Powell and Sheffi, 1983) address back-hauls by solving a "balancing of empties" sub-problem after routing, whereas more recent approaches from Erera *et al.* (Erera *et al.*, 2013a) and Lindsey *et al.* (Lindsey *et al.*, 2016) guide load plans to route freight along arcs with known "empties".

The most common approach to routing LTL freight is to first generate a *load plan*,

given an operational LTL network (Erera *et al.*, 2013a,b; Powell, 1986). A load plan identifies a single route for every origin-destination pair in the LTL network over which freight must be routed. This results in all routes terminating at a given destination forming an "in-tree". At any break-bulk, freight destined for a given destination is always forwarded to the same next transfer terminal (Erera *et al.*, 2013b). Erera *et al.* (Erera *et al.*, 2013a) note that the in-tree requirement is becoming less important with advances in technology and computing power. Jarrah *et al.* (Jarrah *et al.*, 2009) note that load plans are changed about four times every year to account for seasonal patterns.

The LTL routing problem is a variation of the classic *multicast routing problem*. Oliveira and Pardalos (Oliveira and Pardalos, 2005) present an excellent survey of the applications of multicast routing. One approach to solving such problems is to use a *minimum cost directed Steiner-forest*. Such forests have been studied extensively, for example by Charikar *et al.* (Charikar *et al.*, 1999), Feldman *et al.* (Feldman *et al.*, 2012), Chekuri *et al.* (Chekuri *et al.*, 2011), and Berman *et al.* (Berman *et al.*, 2013). Our hybrid heuristic in Section 2.3 uses minimum cost directed Steiner-forests to create load plans.

Most existing papers on directed Steiner-forests only consider the cost of each arc in the network, but not the time required to traverse it. Our model can be used to find the minimum cost directed Steiner-forest in a time-space LTL network, which is then used as a load plan. See (Kennington and Nicholson, 2010; Erera *et al.*, 2013a) for a description of time-space networks. As noted by Jarrah *et al.* (Jarrah *et al.*, 2009), when multiple service levels are allowed, load plans can be generated for each service level (a similar approach is used in our hybrid heuristic, see Section 2.3.4).

In a classic paper, Powell (Powell, 1986) presents a two-phased approach to LTL routing. During the first "network design" phase, network arcs are identified over which direct service should be offered, and the number of trailers that should be made available on those arcs. In the second phase a load plan is generated for the network, and freight is routed in accordance with the load plan. The network model used does not have an explicit time dimension, and service contracts are implemented by ensuring a minimum number of trailers on each arc. Such "flat network" models are also used in related papers by Powell and Sheffi (Powell and Sheffi, 1983, 1989) and Powell and Koskosidis (Powell and Koskosidis, 1992). In contrast, we use a time-space network to ensure that delivery deadlines are met, similar to the approach taken by Erera *et al.* (Erera *et al.*, 2013a,b). Powell (Powell, 1986) also describes a local improvement heuristic which sequentially adds arcs to (and drops arcs from) the load plan.

Erera *et al.* (Erera *et al.*, 2013a) present an integer programming model for generating load plans. As in our model, they use a set of pre-calculated routes for each origin-destination pair. However, they do not consider individual skids like we do. For models that are large and difficult to solve, they present a heuristic for starting from a feasible solution and solving the model by fixing most variables at their current values. The fixed variables correspond to routes that carry relatively small amounts of freight. The load plan is thus improved by re-routing freight for terminals to which a lot of freight is destined, one terminal at a time. Lindsey *et al.* (Lindsey *et al.*, 2016) improve upon this heuristic by simultaneously adjusting freight transfer paths into multiple terminals in a small sub-section of the network.

In variations of their basic approach, Erera *et al.* (Erera *et al.*, 2013a) use their

model to generate separate load plans each day. Moreover, by relaxing the in-tree constraints, their model can be used to generate an "unrestricted" load plan.

Erera *et al.* (Erera *et al.*, 2013b) present a two-phased approach to routing LTL freight given a load plan. In the first phase freight is routed greedily on the cheapest route available, starting with freight that has the tightest delivery deadline. Freight can be held for a few days at intermediate break-bulks to achieve better consolidation. In the second phase, the routing solution from the previous phase is improved upon using a linear program whenever there is some freight that can be re-routed on longer (and cheaper) routes. Our Greedy algorithm in Section 2.3 is very similar to their first phase, except that we are not confined by a load plan.

Leung *et al.* (Leung *et al.*, 1990) develop a non-linear model formulation for routing LTL freight. They decompose the model into two linear mixed-integer programs (MIPs), each of which can be solved using Lagrangian relaxation. The first step in their solution process is to assign a first and last break-bulk for freight between every origin and destination. The second step is to find optimal routes for all the freight, given the first and last break-bulks from the previous step. The routing problem (*i.e.*, the second step) is a multi-commodity network-flow problem.

Hejazi and Haghani (Hejazi and Haghani, 2007) present a model for routing LTL freight that uses a time-space network, similar to our work. The key difference between their model and ours is that we use variables that refer to a set of pre-calculated routes for each origin-destination pair (see Section 2.3). This allows us to restrict the size of the model by ignoring routes which anyway cannot meet skid deadlines. Recognizing that the model is often too large to solve in real-time with commercial MIP solvers such as CPLEX (CPLEX, 2007), Hejazi and Haghani propose three heuristic

approaches. The first of these heuristics is a local improvement heuristic that starts from a feasible solution, and uses a Greedy approach to improve the solution. The second heuristic enhances the first one by using meta-heuristics (simulated annealing), to avoid getting stuck at local optima. The third heuristic constructs the model using network arcs only from some of the shortest routes for each origin-destination pair.

In addition, Hejazi and Haghani calculate a lower bound on the routing cost on a given day by assuming a limitless number of trailers. In contrast, we calculate a lower bound on the routing cost by only considering a few skids (see Section 2.3 for details).

Katayama and Yurimoto (Katayama and Yurimoto, 2016) model the LTL freight routing problem as a multi-commodity network-flow problem, and solve it using Lagrangian relaxation. Akyilmaz (Akyilmaz, 1994) describes a heuristic which identifies routes that carry a relatively small amount of freight over long distances, using a metric called "empty ton-kilometers". Shipments along these routes are re-routed to be consolidated with other shipments, thereby improving the value of the metric. Barcos *et al.* (Barcos *et al.*, 2010) develop an ant-colony meta-heuristic approach for LTL routing. They make the assumption that there is no limit on the number of trailers available, and that the transit time from the origin break-bulk to the destination terminal is not more than two days.

In addition to routing costs, other concerns for LTL operators include the temporary non-availability of important arcs in the network, underutilized trailer capacity, and carbon dioxide emissions to name a few. These concerns are highlighted in a

broader context by some recent papers on effective supply chain management. Margolis *et al.* (Margolis *et al.*, 2018) develop a multi-objective optimization model for supply chains that can be used to evaluate the trade-off between cost minimization and improved network connectivity. Another multi-objective model (for maritime vessel scheduling) is described by Dulebenets (Dulebenets, 2018) where the cost of carbon dioxide emissions is considered. Dulebenets and Ozguven (Dulebenets and Ozguven, 2017) model the transportation of perishable items using decay costs.

Haass *et al.* (Haass *et al.*, 2015) note that shipping of "empties" not only adds to routing costs but also increases carbon dioxide emissions. Using a simulation, they show that losses due to the decay of perishable items and carbon dioxide emissions can both be minimized with "intelligent containers". Dulebenets (Dulebenets, 2019) considers the problem of scheduling inbound and outbound trucks at cross-docking terminals such as LTL break-bulks. A mixed integer program and an evolutionary algorithm are proposed to solve the problem.

To conclude this section, we note that the state-of-the-art in LTL routing is to start with a load plan. The load plan is essentially "static"; *i.e.*, for a given LTL network it is created in response to the skids received for shipment on that day. The load plan could even be seasonal and based largely on the LTL operator's knowledge of traffic patterns and trailer availability. Since routing costs are heavily influenced by the underlying load plan, we propose that load plans should be changed *dynamically* during routing (for example when trailer capacity is exhausted on some arc). Based on this proposal, we develop a heuristic that frequently regenerates load plans by constructing the minimum cost directed Steiner-forest for the remaining skids. We also present an integer programming model which shows that the LTL freight routing

problem is a tightly coupled combination of load plan generation and routing based on the load plan. Our model and heuristic are described in the next section.

2.3 Model and Heuristic Algorithms

Before presenting our integer programming model, we note that every node in our industry partner's LTL network plays the dual role of an EOL terminal and a break-bulk. We only consider freight movement between these nodes, *i.e.,* every skid travels between a pair of these nodes. In addition, our implementation completes the following preliminary tasks that are required for building the model:

- All possible routes between any two break-bulks are enumerated in advance. For a pair of origin-destination break-bulks, we only consider routes having a limited number of intermediate break-bulks. Similar pre-calculation of routes has also been used by Hejazi and Haghani (Hejazi and Haghani, 2007), Erera *et al.* (Erera *et al.*, 2013a), and Lindsey *et al.* (Lindsey *et al.*, 2016).

  See Appendix A for configuration parameters that can be used to control route enumeration.

- The skids received on previous days are routed so that, on the day for which we intend to find the least cost routes, we start with a "saturated" network. In other words, we inherit a network where some trailers have already been deployed, and routes already assigned for skids received on previous days. The Greedy algorithm described later in this section is used for saturating the network.

2.3.1 Notation

We make use of the following notation in our model formulation:

m	Index on the number of carriers. All the available carriers are enumerated in advance.
d	Index on the day number.
(p, q)	The arc identifier, denoting an arc from break-bulk 'p' to break-bulk 'q'.
r	Index on the skids received for shipment on a given day.
k_r	Index on the possible routes for the 'r^{th}' skid, considering only the skid's origin and destination. Possible routes for every origin-destination pair of break-bulks are enumerated in advance. Therefore, for a given r, the number of routes K_r available to the skid are known in advance, and $k_r = 1, ..., K_r$.
a_{k_r}	Index on the arc sequence number in the '${k_r}^{th}$' route for the 'r^{th}' skid. Here, $a_{k_r} = 1, ..., A_{k_r}$, where A_{k_r} is the number of arcs in the '${k_r}^{th}$' route. The sequence of arcs in every route is known in advance. For example, a route from Toronto to Vancouver with a stopover at Winnipeg has two arcs (*i.e.*, A_{k_r}=2); with the first arc from Toronto to Winnipeg, and the next one from Winnipeg to Vancouver.

$T_{(p,q)}$ The time required to traverse the network arc (p, q), in days. This time depends only on the two endpoints 'p' and 'q'. For example, the transit time for direct service from Toronto to Vancouver is always 5 days.

$T^r_{k_r a_{k_r}}$ The time required to traverse the '$a_{k_r}{}^{th}$' arc of the '$k_r{}^{th}$' route of the 'r^{th}' skid. This time equals $T_{(p,q)}$, where 'p' and 'q' are the endpoints of the arc. For example, this time equals 5 days for the second arc of the route from Montreal to Vancouver which has a single stopover at Toronto.

L^r Deadline (in days) within which the 'r^{th}' skid must be delivered. A 5 day deadline on a skid implies that this skid must be delivered to its destination EOL terminal no later than 5 days after it was received at its origin EOL terminal.

E^r Entry day for the 'r^{th}' skid, *i.e.*, the day on which it is received at an EOL terminal.

v^r Volume of the 'r^{th}' skid measured in cubic feet.

$V^{mdt}_{(p,q)}$ Remaining volume of the 't^{th}' trailer supplied by the 'm^{th}' carrier on the 'd^{th}' day on arc '(p, q)'. If this trailer has not been assigned to carry any freight, then its volume equals the standard trailer volume. Volume is measured in cubic feet.

$s^{md}_{(p,q)}$ Number of trailers available from the 'm^{th}' carrier on the 'd^{th}' day on arc '(p, q)'.

t Index on the trailer number available from a carrier, on a given arc on a given day. Here $t = 1, ..., s^{md}_{(p,q)}$. Always used in conjunction with the carrier index, arc identifier, and day index.

$c^{md}_{(p,q)}$ Cost of a trailer from the 'm^{th}' carrier on the 'd^{th}' day along the network arc '(p, q)'. Note that this cost is non-zero only when a new trailer is rented. For a trailer that has already been rented and assigned to carry some skids, this cost is zero. The unit of cost is Canadian Dollars.

h_p Handling (cross-docking) cost at break-bulk 'p'. Currently a constant independent of 'p'.

2.3.2 Model Formulation

Let

$$x^{mdrt}_{k_r a_{k_r}} = \begin{cases} 1, & \text{if the '}t^{th}\text{' trailer of the '}m^{th}\text{' carrier is used for carrying the '}r^{th}\text{' skid on the '}a^{th}_{k_r}\text{' arc of its '}{k_r}^{th}\text{' route on the '}d^{th}\text{' day} \\ 0, & \text{otherwise} \end{cases}$$

$$y^{r}_{k_r} = \begin{cases} 1, & \text{if the '}r^{th}\text{' skid is scheduled on the '}{k_r}^{th}\text{' route for this skid's origin and destination} \\ 0, & \text{otherwise} \end{cases}$$

$$z^{mdt}_{(p,q)} = \begin{cases} 1, & \text{if the '}t^{th}\text{' trailer from the '}m^{th}\text{' carrier is commissioned on the '}d^{th}\text{' day on arc '}(p,q)\text{'} \\ 0, & \text{otherwise.} \end{cases}$$

Our integer programming formulation for minimizing the cost of routing on a given day is as follows:

$$\min \sum_{m}\sum_{d}\sum_{(p,q)}\sum_{t} c^{md}_{(p,q)} * z^{mdt}_{(p,q)} + \sum_{m}\sum_{d}\sum_{r}\sum_{k_r}\sum_{a_{k_r}}\sum_{t} x^{mdrt}_{k_r a_{k_r}} * h_p \tag{2.3.1}$$

$$\text{s. t.} \sum_{r}\sum_{k_r} x^{mdrt}_{k_r a_{k_r}} * v^r \leq V^{mdt}_{(p,q)} * z^{mdt}_{(p,q)} \ \forall m, d, (p,q), t \tag{2.3.2}$$

$$\sum_{m}\sum_{d}\sum_{t} x^{mdrt}_{k_r a_{k_r}} = y^r_{k_r} \ \forall r, k_r, a_{k_r} \tag{2.3.3}$$

$$\sum_{k_r} y^r_{k_r} = 1 \ \forall r \tag{2.3.4}$$

$$\sum_{t} z^{mdt}_{(p,q)} \leq s^{md}_{(p,q)} \ \forall m, d, (p,q) \tag{2.3.5}$$

$$\sum_{m}\sum_{d}\sum_{t} x^{mdrt}_{k_r A_{k_r}} * (d + T^r_{k_r A_{k_r}}) \leq E^r + L^r \ \forall r, k_r \tag{2.3.6}$$

$$\sum_{m}\sum_{d}\sum_{t} x^{mdrt}_{k_r a_{k_r}} * (d + T^r_{k_r a_{k_r}}) \leq \sum_{m}\sum_{d}\sum_{t} x^{mdrt}_{k_r (1+a_{k_r})} * d \ \forall r, k_r, a_{k_r} = 1, \ldots, (A_{k_r} - 1) \tag{2.3.7}$$

$$z^{mdt}_{(p,q)} \geq z^{md(t+1)}_{(p,q)} \ \forall m, d, (p,q), t = 1, \ldots, (s^{md}_{(p,q)} - 1) \tag{2.3.8}$$

$$x^{mdrt}_{k_r a_{k_r}}, y^r_{k_r}, z^{mdt}_{(p,q)} \in \{0,1\} \ \forall m, d, (p,q), t, r, k_r \tag{2.3.9}$$

2.3.2.1 The objective function

The objective is to minimize the routing cost on a given day. The first term of the objective function is the sum of the costs of trailers deployed to route the skids. Note that some of these trailers are deployed on a later date at an intermediate break-bulk, for onward routing. The second term of the cost is the cross-docking cost at the break-bulks. Recall that cross-docking is only done at the intermediate break-bulks. Therefore, this term overestimates the cross-docking cost by a constant amount equal

to h_p times the number of skids. In practice, this second term tends to be much smaller than the first term and is ignored to make the formulation simpler.

Note that skid induction and delivery costs (which are handling costs in addition to cross-docking costs) depend only on the number of skids being routed on that day, and are therefore not shown in the objective function. Although these costs were set to zero in this chapter, they were added to the objective function in subsequent chapters. This has no effect on the optimization problem but results in a different objective function value. Moreover, our tests in different chapters were conducted over the course of several years. Small changes in the default values, the ordering of skids, and varying software versions can all lead to a slightly different formulation (even though the general structure of the problem remains the same).

It is also important to preserve the order of the summations as shown in the objective function. This is because the number of trailers on offer depends on the carrier, the day, and the arc. Similarly, the arcs included in a route can only be iterated over once the route is chosen. In turn, the routes for a skid can be iterated over only after the skid is chosen.

2.3.2.2 The constraint set

Constraints in the constraint set (2.3.2) are trailer volume capacity constraints. The sum of the volumes of the skids packed into a trailer cannot exceed the remaining volume capacity of that trailer. Recall that trailers already commissioned are available free of charge, but will have some of their volume already filled. These constraints assume the existence of a function that accepts an arc identifier '(p,q)' and a route number 'k_r' for skid 'r', and returns the arc index 'a_{k_r}' (if it exists in the route).

Although not shown in the model, an identical constraint set also applies to the trailer weight.

Constraint set (2.3.3) ensures that, if a route is selected for a skid, then all the arcs of the route will be traveled in some trailer belonging to some carrier on some day. Constraint set (2.3.4) simply states that, of all the routes available to a skid, exactly one must be chosen. Since the routes are enumerated in advance, we know the minimum duration of travel along every route. Therefore, when preparing the model we can ignore routes that cannot meet the deadline for a given skid. This helps reduce the number of variables and constraints in the model.

Constraint set (2.3.5) ascertains that the number of trailers of a given carrier used on an arc on any given day cannot exceed the number of trailers that the carrier has on offer on that day on that arc. During implementation, this constraint set can be omitted and enforced implicitly by limiting the number of 'z' variables created for each carrier on a given arc on a given day.

Constraint set (2.3.6) ensures that skids are delivered by their due date. The day on which a skid starts traveling the last arc of its route, plus the time required to travel the last arc, must not be more than the skid's entry day plus its deadline.

Constraint set (2.3.7) applies to all but the last arc of the route chosen for each skid. It ensures that the day on which the skid starts traveling the next arc on this route is not before the day on which this arc was started plus the time it takes to travel this arc. In other words, this constraint enforces the travel time associated with every arc. These inequality constraints can be changed into equality constraints to prevent break-bulks from holding skids for a few days before routing them onward.

Constraints in set (2.3.8) are symmetry breaking constraints that impose a sequence order on the available trailers. They ensure that a second trailer from any carrier cannot be used before its first one is commissioned. This constraint set applies only when the number of trailers is more than one.

2.3.3 Greedy Heuristic

Our greedy approach to LTL routing arranges skids by ascending order of "deadline slack", and routes the skids one-by-one on the cheapest route available. Here, the deadline slack is defined as the skid's deadline (in days) minus the duration of the fastest route available to its destination (also in days). This approach is based on the observation that skids with larger deadline slacks are better suited for taking advantage of already deployed trailers, since they can be scheduled on convoluted routes without violating deadlines. Our Greedy heuristic can also hold skids for a few days at break-bulks to find cheaper routes for them.

Our implementation is similar to the GRASP heuristic described by Erera *et al.* (Erera *et al.*, 2013b), although we run only one iteration of the Greedy heuristic. In fact, even Erera *et al.* (Erera *et al.*, 2013b) use only a single Greedy iteration in their tests. Unlike their heuristic however, our Greedy heuristic does not use a load plan. Moreover, our Greedy heuristic can be forced to find routes without adding any new trailers, a feature that we use in the bin-packing sub-problem of our hybrid heuristic (see Section 2.3.4).

One advantage of the greedy approach is that it can route most of the skids, even if some skids cannot be routed due to lack of trailer capacity. The model on the other hand cannot route any of the skids if the problem is infeasible. We use this

observation to always remove those skids from the problem which were not routed by the Greedy heuristic, to ensure that our model is feasible.

2.3.4 Directed Steiner-Forest Heuristic

Our directed Steiner-forest heuristic is a hybrid heuristic inspired by techniques from functional decomposition. Observe that, if trailers had unlimited capacity, then the LTL routing problem would simply become a routing problem in a time-space network. Additionally:

- the solution to the LTL routing problem on a given day would be the minimum cost directed Steiner-forest in a time-space network.
- the routing cost would be the sum of the costs of the arcs in the time-space network used to route skids received for shipment on that day.
- arc cost would equal the rental cost of the cheapest trailer available on the arc on the day the arc was used. Arcs in the time-space network already being used for routing skids would cost nothing when used again on the same day.

Of course, in reality the capacity of each trailer is limited. Moreover, the assignment of skids to trailers can influence the set of additional trailers needed for routing all the skids, when already rented trailer capacity is exhausted. Therefore, the LTL routing problem becomes a combination of a routing problem and a bin-packing problem.

Based on these observations, our hybrid heuristic decomposes the LTL routing problem into a series of simpler sub-problems. The first of these is a "routing sub-problem" which solves our model using only a few carefully selected skids. If an

optimal solution is found, then the selected skids are routed as dictated by the optimal solution (this can include the commissioning of new trailers). These routes, along with any other arcs in the time-space network on which trailers have already been commissioned, are presented as a "load plan" to the next stage of the heuristic which is a "bin-packing sub-problem".

Each bin-packing sub-problem routes skids in accordance with the load plan it is given. *Note that the bin-packing sub-problem is not allowed to add any new trailers when using the load plan.* Trailers on each arc included in the load plan are used for routing skids, until no more skids can be routed using only these trailers. At this stage, any remaining skids (*i.e.,* skids that have not been assigned routes so far) are used to generate another load plan by solving another routing sub-problem.

This iterative solution process is repeated a few times, after which any remaining skids are routed greedily (allowing trailer addition). The Greedy heuristic is also used for routing all the remaining skids if the routing sub-problem is unable to find a feasible solution. A pseudo-code block for our hybrid heuristic appears in Algorithm 1.

2.3.4.1 Routing sub-problem

The objective of this sub-problem is to identify the lowest cost directed Steiner-forest for the skids being routed. Once this forest is identified, trailers are commissioned as needed along its arcs (line 10, Algorithm 1) for use by the subsequent bin-packing sub-problem. *The actions of the routing sub-problem are reminiscent of an ant leaving a strong pheromone trail along the preferred route for other ants in the colony to follow during the subsequent bin-packing phase of the algorithm.*

Algorithm 1: Directed Steiner-forest heuristic

```
1  Initialization:
2     R ← All skids on this day, iterationCount ← 1,
3     isFeasible ← true,  MaxIterations ← Maximum number of
    iterations
4
5  while (iterationCount ≤ MaxIterations) ∧ isFeasible ∧ (R is not empty) do
6  |
   |  // First solve the routing sub-problem
7  |  S ← Select skids from R for preparing the model
8  |  Prepare the model using skids in S, and solve it
9  |  if (model is feasible) then
10 |  |  Use optimal solution to commission new trailers and route
   |  |   skids in S
11 |  |  R ← R - S
12 |  |  Load plan ← Trailers already deployed on the time-space
   |  |   network
13 |  else
14 |  |  isFeasible ← false
15 |  end
16 |
   |  // Now solve the bin-packing sub-problem
17 |  if (isFeasible) then
18 |  |  Route skids in R using the Load plan (no new trailers can
   |  |   be commisioned)
19 |  |  Remove from R the skids that were successfully routed
   |  |   (i.e., packed into bins)
20 |  end
21 |
22 |  iterationCount ← iterationCount + 1
23 end
24  Route any remaining skids in R using the Greedy heuristic
    (allow trailer addition).
```

Routing sub-problems are constructed using our model by selecting only a few skids for routing according to the following rules:

1. Only one skid must be selected for each remaining origin-destination pair.

2. Each selected skid must have the tightest deadline among all the remaining skids with the same origin and destination. Ties must be broken in favor of the skid with the smallest volume.

The justification for these rules is as follows:

- any load plan that can route the tightest deadline skid for each remaining origin-destination pair can also route every other skid received on that day, without violating any deadline.

- the lowest cost solution to the routing sub-problem continues to be a feasible (although possibly non-optimal) solution even if the volume of a skid is artificially reduced. Here, we use the term "artificial" to describe any change we make to the test data obtained from our industry partner. Hence, the cheapest solution to the routing sub-problem will always be found when considering the smallest volume skids.

Observe that, if we artificially reduced the volume of all the selected skids to nearly zero before solving the model, then we would find the minimum cost directed Steiner-forest in the time-space network. However, in our implementation we retain the true volume of each selected skid. This ensures that the resulting solution can be used to route at least one skid for every origin-destination pair. In addition, observe that we need not consider all the trailers being offered by every carrier while solving for the load plan, since we are routing a very limited number of skids. This observation can be used to further reduce the size of the model.

We use the cost of the solution from the first routing sub-problem as a lower bound on the cost for routing all the skids on a given day. Ideally, if all the skids could be

routed using the solution produced by this sub-problem, then this solution would be the optimal solution to the overall routing problem.

2.3.4.2 Bin-packing sub-problem

The objective of this sub-problem is to route as many skids as possible, only using trailers that have already been deployed along the optimal routes. Given a load plan, we simply use our Greedy heuristic for routing (thereby packing trailers with skids). More sophisticated schemes could be used for such "bin-packing", but our approach is simple and similar to the GRASP heuristic used by Erera *et al.* (Erera *et al.*, 2013b). All the skids that cannot be routed for lack of trailer capacity are handed over to the next routing sub-problem for generating another load plan.

2.4 Experimental Evaluation

In order to evaluate the performance of our algorithms, we ran them on our industry partner's Canadian LTL network with real data from their operations in March 2016. Their network has 17 nodes and 67 arcs. We routed the skids on several days using all three algorithms (namely the Greedy heuristic, the directed Steiner-forest heuristic, and CPLEX). In addition, we also changed some parameters (see Appendix A) and studied the effect of those changes on solution quality.

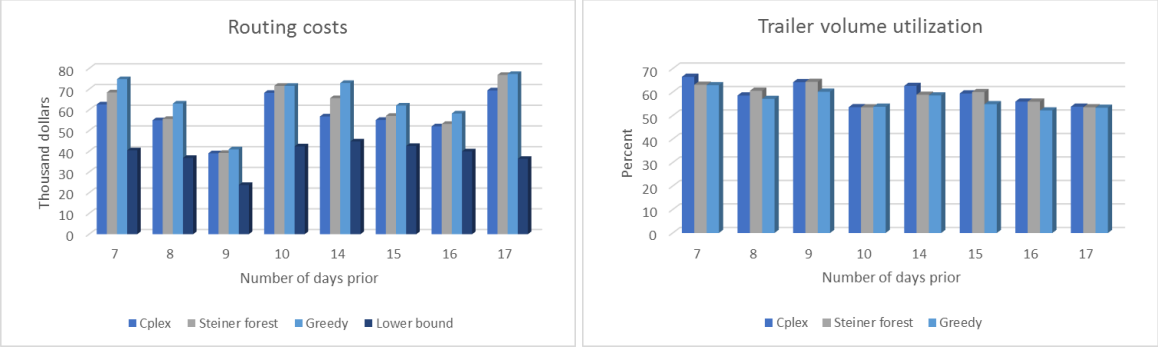
All our tests were executed on a virtual machine running Windows Server 2016 standard edition, with 32GB RAM and four Intel Xeon E5-2680 CPUs at 2.70 GHz each, using CPLEX 12.8 and Java 1.8. CPLEX's solution time limit was set to one hour. The results of our tests are summarized below.

Days prior	**CPLEX**	**Steiner-forest**	**Greedy**	**Lower bound**
7	62.631	68.431	74.821	40.483
8	54.976	55.623	63.048	36.782
9	39.022	39.22	40.935	23.74
10	68.224	71.575	71.575	42.366
14	56.801	65.657	72.968	44.757
15	55.131	57.1	62.079	42.571
16	52.023	53.207	58.242	39.957
17	69.397	76.856	77.319	36.348

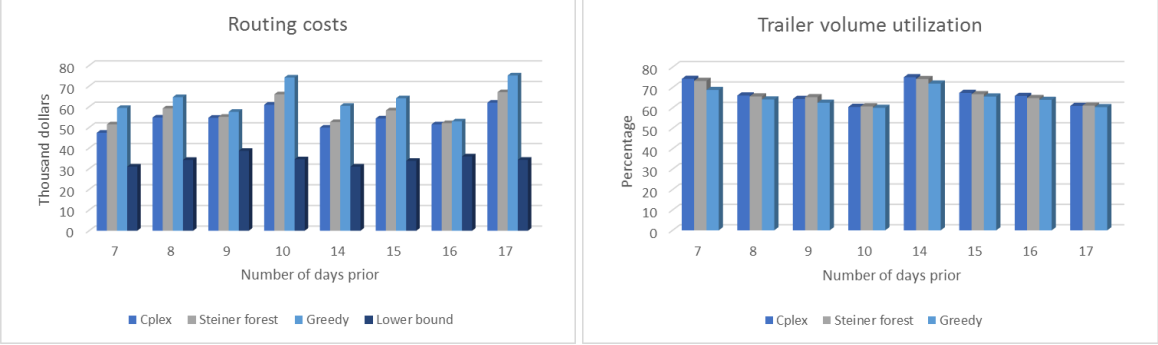
Table 2.1: Results from the base test (costs in thousands of dollars).

(i) *Routing costs and trailer volume utilization*: In this "base" test, we used our default configuration and our industry partner's data without any modifications. We saturated our industry partner's LTL network by routing skids greedily for a few days. Then we routed skids on the next day using all three approaches. The results are shown in Table 2.1 and Figure 2.1(a).

Our industry partner confirmed that the routing plans produced by our Greedy heuristic were marginally lower in cost compared to their actual routing costs. However, because the Greedy heuristic randomly picks one feasible solution among many, its performance on larger LTL networks is not expected to be good. CPLEX found the lowest cost routing schedules on all 8 days and although it could not prove optimality within one hour, the solutions were within 1% of the best bound (this is called the "MIP-gap"). The number of constraints in the model (after CPLEX pre-solve) was about 15000, and the number of variables was about 120000. We note that number of variables and constraints depends on the number of skids being routed on that day. It took about 30 minutes just to create the model using CPLEX Java APIs.



(a) Base test



(b) Repeating traffic pattern

Figure 2.1: Routing costs and Average trailer volume utilization

The cost of the solution found by CPLEX ranged from 86.5% to 99.5% of the cost of the solution found by the directed Steiner-forest heuristic. The Steiner-forest heuristic ran to completion more than ten times faster than CPLEX. The lower bound on the cost suggested by the first iteration of the Steiner-forest heuristic was always 50% or more than the cost of CPLEX's solution.

The average number of skids received for routing each day was about 900, with about 10% of them removed from the problem because the Greedy heuristic could not route them. The hybrid heuristic was sometimes unable to route a few skids for which the Greedy heuristic found a feasible route. This is because

the trailers commissioned by the Greedy heuristic are different from the trailers commissioned by the hybrid heuristic. Regardless of which algorithm is used for LTL routing, there can always be a few skids that must be removed from the problem to find a feasible solution using the selected trailers.

Trailer volume utilization on the day of routing was satisfactory with all three algorithms. Our industry partner expects an average trailer volume utilization of 50% or more, with 60% or more considered good and 70% or more considered excellent.

(ii) *Repeating traffic patterns*: We use the term "repeating traffic" to refer to skids with the same origin and destination accepted for shipment almost every day. Routing strategies that use load plans (such as our hybrid heuristic) can be very effective when traffic patterns repeat. This is because trailers that are commissioned to route skids from previous days are on valid routes for skids being routed on latter days. This can facilitate freight consolidation at the intermediate break-bulks. Moreover, recall that our hybrid heuristic uses routes that form a minimum cost directed Steiner-forest. Repeated use of such forests can result in cost savings.

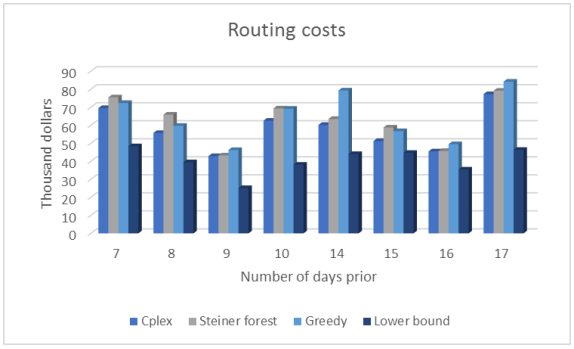
To test this hypothesis, we replaced the skids received by our industry partner at every EOL terminal on every day, with skids received on the first day. Then we routed the skids using all three algorithms. The results are shown in Figure 2.1(b). With repeating traffic patterns, the performance of the directed Steiner-forest heuristic improved relative to the Greedy heuristic (compare Figure 2.1(a) with Figure 2.1(b), and was always within 9% of CPLEX's solution. As before, CPLEX produced the cheapest schedules but did not always result

in the best trailer volume utilization.

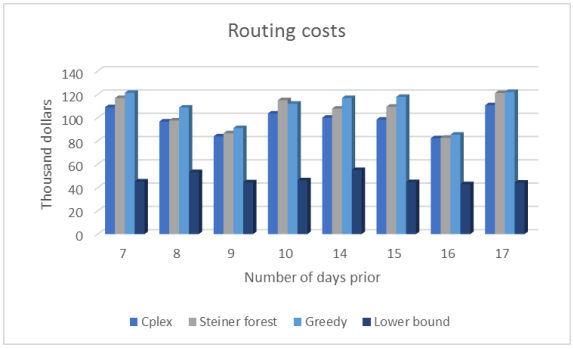
(iii) *Sort-to-Bin*: A relatively new technique for packing skids into trailers is called "Sort-to-Bin". This technique is currently being tested by our industry partner. In this scheme, several skids are packed into a single "Bin" and then the Bins are packed into trailers. Combining skids into Bins can facilitate last mile delivery at the EOL terminals. In our tests, we implemented a *preliminary version* of this scheme using 95 cubic feet Bins, allowing up to 60 skids per Bin (with every skid in a Bin required to have the same destination and deadline). This reduced the number of entities being routed from about 900 a day to about 500 a day. In practice, the restriction that all the skids in a Bin have the same deadline is relaxed, leading to a much larger reduction in the number of entities being routed.

The results of our tests are shown in Figure 2.2(a). Observe that the performance of our hybrid heuristic seems to be erratic in comparison to the Greedy heuristic. This behavior can be attributed to the difficulty of packing large Bins compactly into trailers, which affects both the Greedy heuristic as well as our hybrid heuristic.

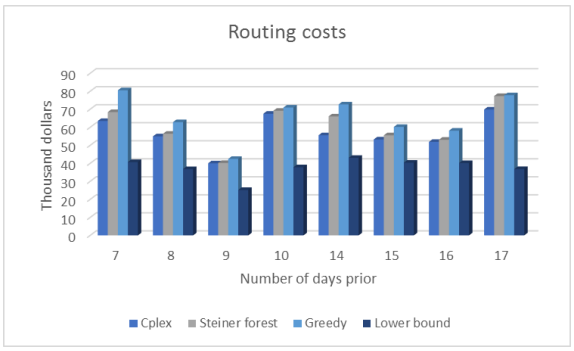
(iv) *Increased network densities*: In order to test on a denser network, we artificially increased the number of arcs in the network by about 10% to 75. These new arcs connected towns in the geographical center of the network to each other, or to large towns like Toronto and Vancouver. The carrier capacity available on these arcs, and the time required to travel them, were assumed to be the same as those for similar arcs (for example, the new arc from Vancouver to Saskatoon inherited its attributes from the existing arc from Vancouver to Regina).



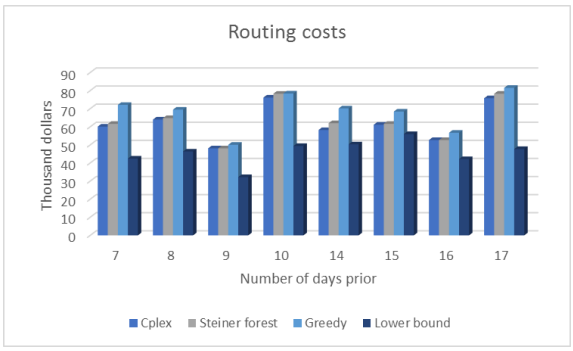
(a) Sort-to-bin



(b) Denser network



(c) Short routes



(d) Very short routes

Figure 2.2: Routing costs for variations of the base test

We also doubled the number of skids on each day by creating a "twin" for each skid, and doubled the number of trailers being offered by each carrier. The resulting MIPs were far larger than the MIPs in the base test, and required more than two hours to construct using the CPLEX Java APIs. The pre-solved MIPs had about 30000 constraints and 400000 variables, a significant increase in size compared to the MIPs in the base test. In comparison, the pre-solved model used by the first iteration of the hybrid heuristic's routing sub-problem had about 5000 constraints and variables, and it could be formulated and solved within minutes. This illustrates the difficulty of using the full model on large

Days prior	**CPLEX**	**Steiner-forest**	**Greedy**	**Lower bound**
7	109.067	116.909	121.411	45.258
8	96.723	97.691	108.706	53.295
9	83.979	86.712	91.048	44.530
10	103.596	115.018	112.042	46.257
14	99.980	107.833	116.891	55.117
15	98.377	109.432	117.895	44.684
16	82.350	82.748	85.489	42.930
17	110.711	121.219	122.144	44.262

Table 2.2: Results with increased network density (costs in thousands of dollars).

networks, and underscores the utility of our hybrid heuristic.

Table 2.2 and Figure 2.2(b) show the results of running our algorithms on these problems. While CPLEX produced the cheapest routes after one hour, the MIP-gaps were usually more than 5% (compared to MIP-gaps of within 1% in the base test). Our hybrid heuristic was again more than 10 times faster than CPLEX, and produced good quality solutions in 7 out of 8 tests. On day 11, the hybrid heuristic produced a solution that was 2% more expensive than the Greedy solution.

(v) *Reduced route sizes*: As pointed out by Erera *et al.* Erera *et al.* (2013b), many skids have tight deadlines and have to be routed along short routes with very few intermediate break-bulks (the same is true of our data set). To take advantage of this observation, we reduced the maximum number of arcs in any route to 2 more than the shortest route (down from the default value of 3), and then to 1 more than the shortest route. The results of running our algorithms with these shorter route sizes are shown in Figure 2.2(c) and (d) respectively. Observe that CPLEX's solution costs were consistently more than 80% (and often more than

90%) of our hybrid heuristic's solution costs. Our hybrid heuristic found near optimal solutions on several days. CPLEX was able to solve these smaller MIPs to optimality within 1 hour.

2.5 Conclusions and Future Work

In this chapter, we proposed an integer programming model for routing LTL freight. We tested our model on our industry partner's Canadian LTL network and found that it can significantly lower their routing costs. Our model can also be solved with a few carefully selected skids to generate a load plan and a lower bound on the routing cost for each day. The lower bounds we found were always more than 50% of CPLEX's solution cost on our industry partner's LTL network.

Our model can be difficult to formulate and solve on larger LTL networks. To address this issue, we developed a routing heuristic that extends the notion of a minimum cost directed Steiner-forest to a time-space network. Our heuristic used ideas from functional decomposition to separate the model into a routing sub-problem and a bin-packing sub-problem. Using the heuristic we were able to find low cost routes in a fraction of the time needed to solve the model with CPLEX. Our heuristic proved particularly effective with repeating traffic patterns and when the number of arcs in each route was limited to a small number.

The key managerial insight from our experiments is that exact solution approaches can be used for smaller Canadian LTL networks, but heuristic approaches are needed for larger networks. The load plan should be regenerated frequently to account for possible changes in the structure of the lowest cost directed Steiner-forest (for example when a trailer's capacity is exhausted).

Some possibilities for extending our work in this chapter are as follows:

- other objectives can be included in addition to total costs, such as maximizing trailer capacity utilization and reducing carbon dioxide emissions.
- existing heuristics for constructing minimum cost directed Steiner forests can be extended to generate such forests in time-space networks. This would eliminate our dependency on an MIP solver such as CPLEX.
- more sophisticated approaches can be used for bin-packing, instead of simply using the greedy heuristic.

Another interesting topic for further exploration is how to checkpoint and restart the computation in the face of sudden changes in the input. For example, an important network arc may suddenly become unavailable due to traffic or weather conditions. The Apache Spark big-data processing framework includes a mechanism (Zaharia *et al.*, 2012) that could be leveraged for this purpose. The following chapters focus on solving the LTL routing problem (and other optimization problems) in a distributed fashion on a cluster of computers, such as an Apache Spark cluster.

Chapter 3

Search restarts for MIP solver parallelization

3.1 Introduction

MIP solvers use a powerful algorithmic technique called Branch-and-bound (Lawler and Wood, 1966) for solving Mixed Integer Programs (MIPs) such as the LTL routing problem. Several commercial and open-source MIP solvers based on this technique are available, such as CPLEX (CPLEX, 2005, 2007), Gurobi (Gurobi Optimization, 2018), and SCIP (Achterberg, 2009). These solvers proceed by repeatedly branching the remaining subproblems in the MIP on an integer variable, where each remaining subproblem is represented by a leaf node in the search tree (see Figure 3.1). The set of remaining subproblems is sometimes referred to as the *frontier* (as in DryadOpt). A leaf node can be pruned out of the frontier when it is established that it is incapable of producing a solution superior to the incumbent. The computation completes when the frontier is empty.

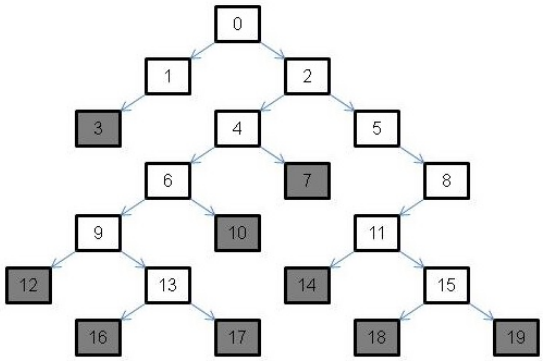


Figure 3.1: A branch-and-bound search tree being explored by a MIP solver (frontier shown shaded). In the text, a node numbered "i" is referred to as "`Node`i".

3.1.1 MIP solver Parallelization

For MIPs that are very complex and difficult to solve on a single computer, the problem can be distributed on a cluster of computers by first "ramping it up" into a suitably large frontier[1]. The frontier is then partitioned into roughly equal sized subsets, and one such subset is *migrated* to each computer in the cluster. Some leaf nodes may be held in a *node pool* for later assignment.

On each computer, there is a single *worker* process that is responsible for sequencing the subproblems (*i.e.*, the leaf nodes) that migrated to it. Each worker prioritizes its subproblems using heuristics such as best-first or depth-first. The highest priority subproblem is solved for a pre-configured *time quantum* by assigning it to a MIP solver, before the next subproblem in the priority sequence is given a turn (see Figure 3.2). The same subproblem (which is now a tree rooted at a leaf node that migrated to this computer) may get multiple turns at solution if the other subproblems continue to have lower priority as per the sequencing heuristic. Any integer feasible

[1]There are many variations to the simple theme for distribution we describe here. See Ralphs *et al.* (Ralphs *et al.*, 2018) for other variations, terminology, and details pertaining to MIP distribution.

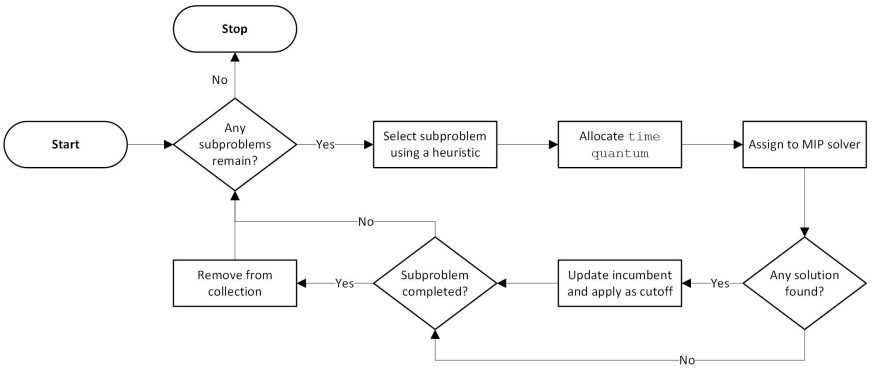


Figure 3.2: Flow chart for sequencing a collection of subproblems.

solution found by the MIP solver is used to update the *local incumbent*, which is the best solution known to the worker. The local incumbent also serves as a cutoff while solving the subproblems. A subproblem is completely solved when all its leaves get pruned.

This prioritized sequencing of subproblems on each computer is paused periodically when a synchronization point is reached, during which all the workers in the cluster are synchronized with each other by a *master* process. At this stage, load across the cluster can be balanced as needed by collecting some leaf nodes from burdened workers and migrating those nodes away to relatively less burdened workers (or into a node pool). The local incumbents are compared and the best one is saved as the *global incumbent* – the best solution found thus far. The new global incumbent is then used to update the local incumbent on every worker before starting another solution iteration. As in the sequential (*i.e.*, non-distributed) case, the computation completes (and these iterations stop) when there are no leaf nodes left to branch, and all the workers become idle.

3.1.2 Adapting branching strategies for MIP distribution

Implementations of distributed MIP usually treat the MIP solver as a "black-box" – attention is mainly focused on orchestrating the distributed computation. Attempts are made to minimize the network communication overhead (as in ALPS (Xu *et al.*, 2005)), and to fully utilize the available compute capacity by using dynamic load balancing as in ParaLEX (Shinano and Fujie, 2007). Different pooling strategies are used to maintain collections of pending subproblems. For example, ParaLEX uses node pools while DryadOpt uses subtree parallelism, and ParaSCIP (Shinano *et al.*, 2016b) uses a combination of both.

We take the view that the internal workings of MIP solvers need to be taken into careful consideration for effective MIP distribution. In particular, new branching strategies need to be devised (and existing ones adapted as needed) in order to make distribution more effective. This is because existing branching strategies were not designed with distribution in mind. For example, the popular Pseudo-cost based branching strategy (Bénichou *et al.*, 1971) relies on information updates from various parts of the search tree. Such updates are not readily available during a distributed computation. This can lead to a large number of inaccurate branching decisions and (consequently) to sublinear speedups upon distribution.

In this chapter, we discuss how popular branching strategies can be adapted for MIP distribution. We start with a review of existing literature in Section 3.2. A categorization of branching strategies appears in Section 3.3, along with some proposed extensions. Section 3.4 details a simple distributed implementation we used to test one of our proposals. We present results from our experiments in Section 3.5, and conclude in Section 3.6.

3.2 Literature Review

Ralphs *et al.* (Ralphs *et al.*, 2018) present an excellent summary of the state-of-the-art in distributed MIP. They note that *"most state-of-the-art [MIP] solvers generate a wide range of information while searching one part of the tree that, if available, could inform the search in another part of the tree"*. Since information updates are not instantly available in a distributed computation, its search tree usually contains more nodes than the search tree of the corresponding sequential (*i.e.*, non-distributed) computation. This phenomenon is called *redundant work*, and it results in sublinear speedups upon distribution (Ralphs *et al.*, 2003, 2018).

One aspect of the search that can be adversely effected by the lack of instant information updates is the branching strategy. An example is the popular Pseudo-cost based branching strategy introduced by Benichou *et al.* (Bénichou *et al.*, 1971). Pseudo-cost branching keeps a history of the success of variables on which branching has already happened (Achterberg *et al.*, 2005). Pseudo-costs must be estimated for variables that do not have any branching history associated with them, for example by using objective function coefficients (Linderoth and Savelsbergh, 1999; Glankwamdee and Linderoth, 2006). However, poor branching decisions made using inaccurate estimates can double the size of the search tree with each bad decision, especially when these decisions are made near the root of the search tree (Glankwamdee and Linderoth, 2006; Forrest *et al.*, 1974; Linderoth and Savelsbergh, 1999).

This problem can be more serious in a distributed computation than in a sequential one because Pseudo-cost information can only be shared periodically when all the computers in the cluster are synchronized with each other. In the meantime, inaccurate branching decisions can be made on every computer. A more detailed discussion

of Pseudo-cost branching can be found in (Linderoth and Savelsbergh, 1999).

Patel and Chinneck (Patel and Chinneck, 2007) develop branching strategies that only use properties of the node under branch by basing their branching decision on the constraint structure at the node. These strategies are especially useful for finding feasible solutions to MIPs. Patel and Chinneck also cite the following important observation from Linderoth and Savelsbergh (Linderoth and Savelsbergh, 1999) and Bénichou *et al.* (Bénichou *et al.*, 1971):

"Pseudo-costs of an integer variable in a particular branch direction remain the same in a branch-and-bound tree, with the exception of a few nodes. This means that once the Pseudo-cost of a variable is computed, it can be used throughout without having to recompute it at other nodes".

Another popular branching strategy that only uses the properties of the node under branch is *full strong branching*. Strong branching was introduced in CPLEX 7.5 and is discussed in (Achterberg *et al.*, 2005; Applegate *et al.*, 1995; Refalo, 2004; Klabjan *et al.*, 2001). Based on Patel and Chinneck's observation, a variable that has been a strong branching candidate at one node need not be considered again for strong branching at another node because its Pseudo-costs can be computed at the first node where it was a candidate, and reused thereafter. This variation of strong branching can be used to speed up full strong branching and is an example of the reliability branching technique proposed by Achterberg *et al.* (Achterberg *et al.*, 2005), with every initialized Pseudo-cost considered reliable throughout the computation. However, CPLEX API's currently do not allow users to restrict the strong branching candidate set by excluding variables whose Pseudo-costs have already been initialized.

An important class of MIPs called satisfiability propositions (SAT problems) have

very effective branching heuristics that are based only on the properties of the node under branch. These heuristics include the DLIS and MOMS heuristics and their variants (Marques-Silva, 1999). Pseudo-Boolean optimization problems are another important class of MIPs that can be converted into SAT problems and solved using the same branching heuristics (see Eén and Sorensson (Eén and Sorensson, 2006) for details).

In Section 3.3, we discuss some modifications to existing branching strategies which can make them more suitable for use in distributed MIP. We also outline a technique for a-priori decomposition of the MIP. A-priori decomposition of MIPs is discussed in detail by Bussieck *et al.* in (Bussieck *et al.*, 2009).

A simple implementation of distributed MIP that we have used to test our proposals appears in Section 3.4. It uses a variation of the static decomposition approach described by Malapert *et al.* (Malapert *et al.*, 2016). The master machine decomposes the original problem into a collection of subproblems that are then distributed to the workers. Details of our implementation appear in Section 3.4.

CHiPPS (Xu *et al.*, 2009) can be used for distributing tree search algorithms. It includes BiCePS (which supports relaxation based branch-and-bound algorithms) and a library called BLIS that provides functionality for solving MIPs. Detailed descriptions can be found in (Xu *et al.*, 2009). ParaSCIP (Shinano *et al.*, 2011, 2016b) is a distributed implementation of the SCIP solver. Other projects in ParaSCIP's family include ParaLEX (Shinano and Fujie, 2007; Shinano *et al.*, 2008) which distributes CPLEX over an MPI based cluster, ParaExpress (Shinano *et al.*, 2016a), and PUBB (Shinano *et al.*, 2003). DryadOpt (Budiu *et al.*, 2011) is an implementation of distributed MIP over the data-parallel cluster Dryad (Isard *et al.*, 2007). Some other

noteworthy implementations of distributed MIP include FATCOP (Chen and Ferris, 2001), PEBBL (Eckstein *et al.*, 2015), and BOB++ (Galea and Le Cun, 2007).

3.3 A Categorization of Branching Strategies

In general, branching strategies that do not rely on information from other parts of the search tree can be more suitable for MIP distribution. Such strategies could result in every node producing the same two children regardless of whether the computation is distributed or not (as long as the cuts and any preprocessing being applied at every node are the same in both computations). This in turn could lead to less redundant work because the search trees of the distributed and sequential computations would be almost identical. Based on this intuition, branching strategies can be categorized as follows:

i) The first category includes strategies that only use properties of the node under branch. Several such branching strategies are already available in CPLEX. One example is the aforementioned full strong branching strategy which makes very accurate branching decisions but can be time consuming. Strong branching is useful for making important branching decisions (*e.g.*, near the root node of the search tree).

   Another strategy is to branch on the least infeasible variable in the linear relaxed solution of the branching node. CPLEX documentation notes that least infeasible branching is useful for finding feasible solutions but is slow to reach the optimal integer solution. It is therefore of limited utility, even though it makes branching decisions quickly.

Lastly, the branching strategies of Patel and Chinneck fit into this category, as do SAT branching heuristics such as DLIS and MOMS. In Chapter 4, we use SAT branching heuristics to solve pseudo-Boolean problems.

ii) The second category includes branching strategies which rely on information from other parts of the search tree, but can be modified to reduce their reliance on such information. One example is Pseudo-cost based branching. As noted in Section 3.2, a variation of this strategy could be to initialize the Pseudo-costs for a large number of variables using strong branching, before starting the distributed computation. A central repository of unchanging Pseudo-costs could be created and used throughout the distributed computation. We test this variation of Pseudo-cost branching in Section 3.5.

Observe that the process of initializing the Pseudo-costs can itself be distributed relatively easily. This can be done for example by partitioning the set of variables in the model, and assigning one subset to each computer in the cluster for Pseudo-cost initialization.

iii) As in any distributed computation, an attempt could be made to partition the subproblems of the orignal MIP into *independent subsets*, so that the subproblems in every subset have *no variables in common* with the subproblems in any other subset. If one such subset were assigned to each computer in the cluster, it would obviate the need for information exchange between the computers *as far as branching is concerned*. Independent subsets would also need to be identified every time dynamic load balancing is performed.

To the best of our knowledge, none of the existing branching strategies fit into

this category. CPLEX's branching strategies emphasize improvement in the dual bound, especially when the user's emphasis is not on finding feasible solutions (refer to CPLEX documentation for the available emphasis configurations). It can be beneficial to prioritize variables that not only tighten the dual bound, but also help us quickly arrive at a set of leaf nodes that can easily be partitioned into independent subsets as above.

Of course, such partitioning is hard to achieve in practice. Nonetheless, some relief is offered by the observation that most of the variables in a MIP are never used for branching. Therefore, only some of the most "important" variables could be considered when creating the independent subsets. Pseudo-cost estimates could be used to separate important variables from the relatively less important ones. Another approximation could be to consider only those variables that are fractional in the linear relaxed solution of at least one subproblem. Since SAT branching heuristics emphasize constraints containing a small number of variables, some constraints could be ignored when partitioning a SAT or Pseudo-Boolean problem.

In Chapter 5 we analyze a technique called *Controlled Branching* which can be used to combine all the subproblems in a given "independent" subset into a single tree, thereby partitioning the MIP search tree into a collection of independent search trees.

3.4 Test setup

We distributed CPLEX over a small TCP-IP cluster in order to test our proposal of using a central repository of Pseudo-costs as described in Section 3.3. Our implementation can be considered an extension of the static decomposition approach described by Malapert *et al.* (Malapert *et al.*, 2016). Various aspects of our implementation are described below in more detail.

3.4.1 Map and Reduce

Our distributed implementation is built on top of a TCP-IP cluster and uses a "Map-Reduce" (Dean and Ghemawat, 2008) paradigm. One each computer, there is single *worker* responsible for sequencing the subproblems assigned to it during the Map cycle. Subproblems are prioritized in a best-first manner and the selected subproblem is assigned to CPLEX for a preconfigued *time quantum* (default 2 minutes). Each Map cycle lasts 10 minutes (this is configurable) during which time multiple subproblems can be assigned to the MIP solver.

Each subproblem has a unique ID and a status associated with it. The status can take on the values "completed", "in-progress", or "untouched". A completed subproblem is one which has been solved to provable optimality or infeasibility. An in-progress subproblem is one that has been assigned to CPLEX, but is neither solved to provable optimality nor proven infeasible. An untouched subproblem is a set of branching conditions representing a leaf node. These branching conditions can be applied on the original MIP model and the updated model assigned to CPLEX.

At the end of the Map cycle, every worker sends a status report to a *master* machine (this is the "Reduce" step). The status report includes the lowest dual bound

Worker	Before			After		
	Completed	In progress	Untouched	Completed	In progress	Untouched
Worker1	S1, S2	S3	S4, S5	S1, S2	S3	S8
Worker2	S6, S7	None	S8	S6, S7	None	S4, S5
Worker3	None	S9, S10	None	None	S9, S10	None

Table 3.1: An example of workload reassignment with 3 workers in the cluster.

of all the subproblems assigned to the worker, the best feasible solution solution known to the worker (if any), and the ID and status of each subproblem that is currently assigned to it. The master machine waits until status reports have been received from all the workers, and updates the global incumbent and the lowest known dual bound. In preparation for the next Map cycle, the master then sends each worker the updated global incumbent and a new assignment of subproblems. The new global incumbent is used to update the local incumbent on every computer, and is used as a cutoff during the next Map cycle. These Map-Reduce cycles stop when every worker reports that all its assigned subproblems are complete.

3.4.2 Load balancing

The procedure for reassigning subproblems to each worker is as follows. First, a list is assembled with the IDs of all the untouched subproblmes in the cluster. The master also makes a note of the in-progress subproblems on each worker. The untouched subproblems are then assigned one by one to the worker with the smallest remaining number of pending subproblems. Here, pending subproblems include both in-progress subproblems and untouched subproblems that have just been reassigned.

An example of this procedure appears in Table 3.1, where we assume 3 workers in the cluster. Worker1 is reporting that subproblems S1 and S2 are complete, S3 is in-progress, and S4 and S5 are untouched. Similarly Worker2 is reporting that subproblems S6 and S7 are complete, while S8 is untouched. Worker3 is reporting that S9 and S10 are both in-progress.

First, a list of untouched subproblems is prepared with S4, S5, and S8 included in it. The in-progress subproblems are left with the same worker that was solving them. Since Worker2 has no pending subproblems, an untouched subproblem (say S4) is assigned to it. The next untouched subproblem S5 is assigned to either Worker1 or Worker2, since both of them now have 1 pending subproblem. If S5 gets assigned to Worker2, then the last untouched subproblem S8 gets assigned to Worker1 which only has 1 pending subproblem. Table 2 shows the result of this reassignment operation, where every worker ends up with 2 pending subproblems.

Upon receiving its updated assignment, each worker prioritizes its new assignment of subproblems which are either already in-progress, or untouched.

3.4.3 Ramp-up

Our ramp-up is run on the master computer until the number of leaf nodes reaches a certain threshold. This threshold is configured to a moderately large number to allow for some dynamic load balancing during the early stages of computation. Our default is to assign 30 subproblems to each computer in the cluster.

All the leaf nodes are sent to every worker in the cluster at the end of ramp-up. Each leaf node has a unique ID and is represented by the branching conditions needed to arrive at it. Every worker is informed of its assignment, which is an equal share of

the leaf node set. This ramp-up operation is not very expensive because our cluster is small. On larger clusters a subproblem should be copied to a worker only if it is assigned to that worker.

Note that a subproblem can benefit from cut generation and presolve routines being reapplied on it once the distributed computation starts, especially when the number of branching conditions needed to create the subproblem is large. However, we do not expect this to be a factor in our tests since our ramp-ups are limited to about 150 leaves.

3.4.4 Pseudo-cost repository

In order to create a repository of Pseudo-costs, the MIP is solved using CPLEX for 1 hour. A control callback is used in single-threaded mode to collect the up and down Pseudo-costs for each variable that is being branched upon. For all such variables, the up and down Pseudo-costs are combined into a single Pseudo-cost using a weighted average (the weight is configurable). Weighted averaging is needed because CPLEX API's do not provide a single Pseudo-cost value for each variable.

There are many possibilities for weighted averaging of the Pseudo-costs. We simply use the larger of the up and down Pseudo-costs, an option that is also available in GAMS/XPRESS. This choice of weight works reasonably well for some MIPs in Section 3.5. The weights used by SCIP are mentioned by Achterberg *et al.* (Achterberg *et al.*, 2005). Other possibilities for combining the up and down Pseudo-costs are noted by Atamtürk and Savelsbergh (Atamtürk and Savelsbergh, 2005).

Our Pseudo-cost repository is implemented using CPLEX variable priority lists. A branching priority is established for all the variables that were branched upon using

the weighted Pseudo-costs calculated above, with higher priority given to the larger Pseudo-cost values. Our implementation can also be configured truncate the resulting priority list to a smaller number of variables.

This priority list is used during ramp-up, and is also attached to every subproblem in the cluster once the distributed computation begins. Note that we configure CPLEX to use Pseudo-cost based branching both while collecting the Pseudo-costs, and during the distributed computation.

3.4.5 Distributed mip-gap

By default, CPLEX treats the incumbent solution as optimal when it is within 0.01% of the dual bound. This configurable percentage is called the *relative mip-gap*. In our implementation, the master computer can halt the distributed computation when the dual bound is within a configurable percentage of the global incumbent. We call this parameter the *distributed mip-gap*.

Recall that each worker supplies its local incumbent as a *cutoff* to CPLEX (and *not* as a *mip-start*)[2]. Therefore, individual subproblems are still solved until it is established that they cannot produce a solution superior to the cutoff. An alternate implementation could be to discard subproblems when their dual bound is within 0.01% of the cutoff, or to use such subproblems only for finding feasible solutions.

[2]Note that feasible solutions for one subproblem are not valid mip-starts for another subproblem. This is because subproblems are created by changing variable bounds in the MIP model.

3.5 Experimental results

The objective of our experiments is to verify that using a Pseudo-cost repository leads to good scaling upon distribution. We used the distributed implementation outlined in the previous section with CPLEX 12.10 as our MIP solver. We tested with some hard problems from MipLib's Benchmark collection Gleixner *et al.* (2019); Koch *et al.* (2011) using pre-solved versions of the MIPs. CPLEX's search strategy was configured to "traditional" and its solution finding heuristics were disabled. The branching strategy was set to "Pseudo-costs".

Our tests were conducted on a cluster of 5 Dell PowerEdge R330 Servers running CentOS 7.4, each equipped with 192 GB of RAM and two Intel Xeon E3-1240 v5 3.5GHz CPUs having 8 cores each. These hyper-threaded cores allow CPLEX to run 32 threads on each server. Our tests are written in Java 1.8 and the source code is on the public internet. Log files from our tests are available upon request.

Results are shown in Table 3.2 and Table 3.3, where only Table 3.2 uses variable priority lists. Columns T_1 and T_2 show the time taken to solve the MIP to provable optimality using the CPLEX emphasis shown. All times are in minutes. Speedup due to distribution was calculated as equal to T_1/T_2. Ideally, we would like the speedup to be 5.

The column "Ramp-up" shows the number of leaves in the ramped-up tree. The number of subproblems assigned to each worker initially is therefore one-fifth of this number. The columns "Solution" show the best integer-feasible solutions found. The time taken to find the optimal integer-feasible solution (pending proof of optimality) is shown in column T_{opt}.

The column TQ shows the time quantum duration. This value was increased from

its default value of 2 minutes when more time was needed to generate cuts and run preprocessing routines at the root node of the MIP. Not allowing CPLEX sufficient time can result in an infinite-loop problem, with root node preprocessing attempted repeatedly but never completed successfully.

For some tests, we first ran the distributed computation for 10 minutes with CPLEX emphasis set to feasibility and only 1 subproblem assigned to each worker. The best feasible solution found (shown in the "Init" column) was treated as the starting solution known before ramp-up. This small investment of time can result in significantly less *non-critical work* performed during the distributed computation. Non-critical work is the work that would never have been done if the optimal solution was known in advance (Ralphs *et al.* Ralphs *et al.* (2003)).

The "Efficiency" column shows the fraction of time the distributed computation spent doing *some* work (as opposed to *simply idling*). This value is calculated using the formula $1 - (TQ * N/T_2)$, where N is the average number of time quanta wasted by each worker for lack of work. Note that our formula is *only an estimate*, since workers can report that they are idle even if the time remaining in a Map cycle is less than the time quantum. Similarly, workers can report that they are "working" even if they are wasting time doing non-critical work.

Observations from our experiments in Table 3.2 are summarized below:

1. Our distributed computation produced good speedups when compared to the corresponding sequential computation that used the same variable priority list. However, one problem is the poor utilization of cluster resources as indicated by the low efficiency. Another problem is the amount of non-critical work performed by the distributed computation, which can be large when a strong solution is

MIP	Sequential			Distributed							
	T_1	Solution	Emphasis	Ramp-up	Init	TQ	T_2	Solution	T_{opt}	Speedup	Efficiency
cryptanalysiskb128n5obj16	710	None	0	25	None	20	180	None	None	3.9	0.64
				25	None	20	180	None	None	3.9	0.58
dws008-01	2091	37412.60	0	150	None	2	290	37412.60	260	7.2	0.79
				750	None	2	190	37412.60	150	11.0	0.75
neos-3754480-nidda	552	12941.74	0	150	None	2	80	12941.74	40	6.9	0.33
				750	None	2	30	12941.74	10	18.4	0.57
neos-4954672-berkel	546	2612710	0	150	None	2	200	2612710	20	2.7	0.25
				750	None	2	160	2612710	10	3.4	0.45
				1500	None	2	130	2612710	10	4.2	0.55
neos-5093327-huahum	206	6260	0	150	None	2	60	6260	20	3.4	0.61
				750	None	2	50	6260	30	4.1	0.90
opm2-z10-s4	39	-33269	0	150	None	2	50	-33269	50	0.8	0.46
				150	-32286	2	40	-33269	40	1.0	0.33
				150	-33269	2	20	-33269	10	2.0	0.66
				25	None	2	40	-33268	30	1.0	0.55
				25	-32286	2	30	-33269	30	1.3	0.48
				25	-33269	2	20	-33269	10	2.0	0.30
roi5alpha10n8	1292	-52.32	0	150	None	2	500	-52.32	440	2.6	0.54
				150	-46.41	2	340	-52.32	230	3.8	0.75
LTL 11 March 2016	162	75082.81	2	150	75082.8083	2	80	75082.8083	10	2.0	0.57
				750	75082.8083	2	70	75082.8083	10	2.3	0.81
splice1k1	3218	-394	2	150	None	5	210	-394	70	15.3	0.45
b1c1s1	3181	24544.25	3	150	None	2	270	24544.25	210	11.8	0.75
				750	None	2	200	24544.25	60	15.9	0.93
sing326	74	7753697.91	3	150	7753714.75	5	30	7753678.54	20	2.5	0.83
traininstance2	4533	71820	3	750	None	5	1750	71820	1720	2.6	0.65

Table 3.2: Solving benchmark problems with CPLEX (using variable priority lists).

MIP	Sequential			Distributed							
	T_1	Solution	Emphasis	Ramp-up	Init	TQ	T_2	Solution	T_{opt}	Speedup	Efficiency
cryptanalysiskb128n5obj16	688	None	0	25	None	20	60	None	None	11.5	0.20
				25	None	20	80	None	None	8.6	0.40
dws008-01	236	37412.60	0	150	None	2	100	37412.60	60	2.4	0.38
				750	None	2	140	37412.60	20	1.7	0.40
neos-3754480-nidda	691	12941.74	0	150	None	2	30	12941.74	30	23.0	0.51
neos-4954672-berkel	1365	2612710	0	150	None	2	960	2612710	300	1.4	0.37
neos-5093327-huahum	285	6260	0	150	None	2	90	6260	20	3.2	0.35
opm2-z10-s4	600	-33269	0	150	None	2	250	-33269	250	2.4	0.35
roi5alpha10n8	118	-52.32	0	150	None	2	310	-52.32	180	0.4	0.69
LTL 11 March 2016	120	75083.61	2	150	75082.8083	2	220	75082.8083	10	0.55	0.35
splice1k1	104	-394	2	150	None	5	90	-394	30	1.2	0.27
b1c1s1	657	24544.25	3	150	None	2	90	24544.25	70	7.3	0.76
				750	None	2	130	24544.25	100	5.1	0.90
sing326	60	7753691.13	3	150	7753714.75	5	80	7753676.17	40	0.8	0.91
traininstance2	224	71820	3	750	None	5	40	71820	20	5.6	0.75

Table 3.3: Solving benchmark problems with CPLEX (no priority lists).

not found quickly. Both of these problems can result in reduced speedups.

2. The problem of low efficiency can sometimes be solved by increasing the number of leaf nodes assigned to each worker. This allows for load balancing well after the ramp-up, and can result in higher efficiency and speedups. Examples include `neos-3754480-nidda`, `neos-5093327-huahum`, and `neos-4954672-berkel`. However, increasing the number of leaf nodes assigned to each worker can also be counterproductive for some MIPs if more non-critical work is performed as a result. An example is the MIP `opm2-z10-s4` where a lot of time was needed to find a strong solution, thereby resulting in more non-critical work.

3. A large value of T_{opt} (almost equal to T_2) usually indicates a large amount of non-critical work. Examples include the MIPs `opm2-z10-s4` and `roi5alpha10n8`. Starting with a good feasible solution can increase the speedup by reducing non-critical work. In fact, the highest speedups for `opm2-z10-s4` were obtained when the distributed computation was deliberately initialized with the well-known optimal solution for the MIP.

   Finding a strong solution quickly can even lead to a super-linear speedup (for *e.g.,* `splice1k1` and `neos-3754480-nidda`). It can also be useful for reducing the mip-gap.

4. With CPLEX emphasis set to "best-bound", the computation resembles a best-first search. In fact, for `b1c1s1` the sequential computation found a feasible solution only after 40 hours. The corresponding distributed computation benefited from finding a good solution sooner.

5. The MIP `cryptanalysiskb128n5obj14` was ramped-up to only 25 leaves because its ramp-up was slow. Both the time quantum and Map cycle time had to be increased to 20 minutes to allow sufficient time for cut generation[3]. While non-critical work is not a concern for infeasible MIPs, efficiency continues to be an important consideration. Dynamic load balancing should be used to achieve better efficiency. Our tests also indicate that performance variability is a concern for this MIP when a branching priority list is not used.

6. Best-first sequencing is effective in iterating through a collection of subproblems when the dual bounds of the subproblems can be tightened steadily. Examples include `neos-3754480-nidda` and `neos-4954672-berkel`. However, for some MIPs there can be a large number of subproblems with the same linear relaxation objective (*e.g.*, when there are no variables in the objective function). For other MIPs it could be difficult to tighten the best subproblem's dual bound quickly. This can result in some subproblems getting multiple turns at the MIP solver while other subproblems wait indefinitely.

   In such cases, good feasible solutions in the waiting subproblems can be at risk of not being discovered for a long time. This in turn can make it harder to narrow the mip-gap quickly. The node sequencing strategy should be altered to set a limit on the number of consecutive turns at the MIP solver a subproblem is given.

Comparing the results from Table 3.3 with those in Table 3.2 leads to the following observations:

[3]Our implementation includes the option of disabling cut generation.

- When variable priority lists were not used (Table 3.3), speedups seemed to decrease for some MIPs with more leaves in the ramp-up. This decrease in speedup can be attributed to more redundant work being performed by the distributed computation, due to a larger number of sub-optimal branching decisions being made across the cluster. Examples include the MIPs `b1c1s1` and `dws008-01` – observe the reduced speedup in spite of improved efficiency.

  On the other hand, speedups improved for some MIPs due to improved efficiency when variable priority lists were used (Table 3.2). An example is the MIP `neos-4954672-berkel`.

- In Table 3.3, the speedups achieved for some MIPs were very low. Examples include `neos-4954672-berkel`, `roi5alpha10n8`, and `sing326`. In general, higher speedups were achieved in Table 3.2.

- A simplified version of the LTL routing problem (with routes allowed to have at most 3 arcs) was also used as a test case. We were able to achieve much better speedups when variable priority lists were used. For this problem, we limited the priority list to 20 variables.

3.6 Conclusions and Future Work

In this chapter, we categorized some branching strategies often used for solving Mixed Integer Programs. We tested our proposal of using a central repository of Pseudo-costs. Results indicate that our proposal can help achieve good speedups upon distribution. However, inefficient use of cluster resources (due to static load balancing) and non-critical work can both reduce the speedup. Our recommendation is to use

dynamic load balancing, and to spend some time initially finding feasible solutions (regardless of the user's choice of MIP emphasis).

A more sophisticated scheme is needed for combining the up and down Pseudo-costs. One possibility is to *learn how to combine the up and down Pseudo-costs* by recording those values and the corresponding branching decisions made by CPLEX at several nodes for many benchmark MIPs. Other properties of the node under branch may also influence the branching decision. We are currently working on this project.

Search restarts are a powerful feature of commercial MIP solvers. CPLEX *dynamic search* implements search restarts by using information obtained from the current search tree. Our approach of using a central repository of pre-caluclated pseudo-costs is an example of such a restart. In a distributed computation, restarts can be enhanced to utilize information obtained from search trees on other computers as well.

Another important avenue for future research is to devise strategies for partitioning a given MIP into independent subsets of leaf nodes, as discussed in Section 3.3. One approach for binary MIPs could be to represent the MIP as a graph with variables at the vertices, and links in the graph joining variables that occur together in some constraint. Independent subgraphs could then be identified using graph partitioning heuristics.

Testing on larger clusters is a future work item for us. Our proposals can also be integrated into existing implementations such as ParaLEX and ParaSCIP.

Chapter 4

CLTL: A CPLEX based solver for the LTL routing problem

4.1 Introduction

In this chapter, we describe an unconventional approach to solving the LTL routing problem with CPLEX. By incorporating branching heuristics used for solving satisfiability propositions (Marques-Silva, 1999) into CPLEX, we show that the LTL routing problem and some other pseudo-Boolean problems from the MIPLIB library (Koch *et al.*, 2011; Gleixner *et al.*, 2019) can be solved in times comparable to CPLEX in its default configuration. Motivation for taking this approach to branching comes from our observation in Chapter 3 that branching decisions made only using the properties of the node being branched can be beneficial for achieving improved scaling upon distribution.

In our experience, some important routing problems tend to be Pseudo-Boolean Optimization Problems (PBOs) (Roussel and Manquinho, 2009; Eén and Sorensson,

2006). A similar observation is made by Aloul *et al.* (Aloul *et al.*, 2002a) who use routing problems for testing their PBO solver (Aloul *et al.*, 2002a,b). Our formulation of the LTL routing problem is a PBO and was described in Chapter 2. PBOs are MIPs in "n" binary variables having the following structure:

$$\text{Minimize } c^T x$$
$$\text{s.t. } Ax \geq b$$
$$\text{where } x_i \in \{0, 1\}, \forall i \in \{1..n\}$$

Equality constraints can be represented by a pair of inequalities. Each constraint in a PBO with "m" constraints is therefore of the following form:

$$a_{j1}x_1 + a_{j2}x_2 + ... + a_{jn}x_n \geq b_j, \forall j \in \{1..m\}$$

There are three popular approaches for solving PBOs (Aloul *et al.*, 2002a):

1. The first approach is to treat the problem as a generic MIP, and assign it directly to an integer programming solver like CPLEX or Gurobi.

2. Another approach is to use a *native support solver* like PBS (Aloul *et al.*, 2002b) or Pueblo (Sheini and Sakallah, 2005).

3. The third approach is to convert the problem into a Satisfiability Proposition (SAT) problem, and then use a SAT solver like zChaff (Fu *et al.*, 2004), Satz (Li and Anbulagan, 1997), Glucose (Audemard and Simon, 2009), or Sato (Zhang, 1997). The reader is referred to (Aloul *et al.*, 2002a; Eén and Sorensson, 2006) for details of the conversion process. SAT problems are a special class of Constraint Programming problems (Freeman, 1995; Marques-Silva, 1999; Zhang and Malik, 2002; Gomes *et al.*, 2008; Heule and van Maaren, 2009).

In this chapter, we develop a CPLEX based solver called CLTL which uses a SAT converted version of the PBO to guide CPLEX's branching for some time. In section 4.2 we present an overview of SAT, PBO, and MIP literature with a focus on branching heuristics. Section 4.3 describes our notion of *trigger equivalence and domination*. Section 4.4 details the implementation of CLTL. Results from computational experiments are presented in section 4.5. We conclude in section 4.6.

4.2 Literature Review

Aloul *et al.* (Aloul *et al.*, 2002a) note that PBO solvers need to answer the following two basic questions:

1. How should Pseudo-Boolean constraints be handled?

2. How should the objective function be accounted for?

One approach for handling Pseudo-Boolean constraints is to convert them into SAT constraints in the conjunctive normal form (CNF), using well known techniques such as Tseitin transformations (Eén and Sorensson, 2006), binary decision diagrams (BDDs) (Eén and Sorensson, 2006), or Boole's expansion theorem (Brown, 2012). This is the approach taken by MINISAT+ (Eén and Sorensson, 2006). A linear time algorithm for transforming linear inequalities into SAT constraints appears in (Warners, 1998). The resulting SAT problem can then be solved using SAT solvers such as Chaff (Moskewicz *et al.*, 2001).

Specialized PBO solvers such as PBS (Aloul *et al.*, 2002b) and Pueblo (Sheini and Sakallah, 2005) can work with Pseudo-Boolean constraints directly, without any need to convert them into CNF format. Nonetheless, they are still "SAT powered" (Aloul

et al., 2002a) (*i.e.*, they rely on many of the techniques used for solving SAT problems). Since SAT solvers are not equipped to handle an objective function, a common strategy is to solve the problem repeatedly – each time adding a constraint requiring the next feasible solution (if any) to have a better objective value than the current solution. This strategy is used by both PBS (Aloul *et al.*, 2002b) and MINISAT+ (Eén and Sorensson, 2006).

An alternate way to handle the objective function is to use a bounding technique, such as the linear relaxed objective of the search tree node. Other options include Lagrangian relaxation and reduced costs. Although it is inefficient to use these techniques when the problem has no objective function (Sheini and Sakallah, 2005), Branch-and-Bound based MIP solvers become a viable alternative when an objective function is present. Brady and Catanzaro (Brady and Catanzaro, 2008) solve PBOs using CPLEX, but invoke MINISAT+ periodically in order to check if the current solution is optimal.

4.2.1 Heuristics for branching

Both MIP and SAT literature recognize the impact of branching strategies on solution time. Achterberg *et al.* (Achterberg *et al.*, 2005) note that the success of a Branch-and-Bound algorithm strongly depends on the strategy used to select the variable to branch on. Silva (Marques-Silva, 1999) discusses the impact of branching strategies on SAT solvers, and details the MOMS (Freeman, 1995; Zabih and McAllester, 1988), BOHM (Buro and Büning, 1992), Jeroslow-Wang (Jeroslow and Wang, 1990), and *literal count* heuristics such as DLIS and its variants.

In a classic paper, Barth (Barth, 1995) describes the *OPBDP* solver for PBOs and

notes the need for branching strategies that simultaneously reduce the search space and improve the dual bound. *OPBDP* branches on the variable with the maximal coefficient in the PBO's objective function. Our experiments indicate that branching decisions for PBOs can be made solely based on the constraint structure at the branching node. In other words, it is adequate to remove a large amount of infeasible space from the branching node which is selected in a best-first manner by the MIP solver[1].

Look-ahead based solvers for SAT (Heule and van Maaren, 2009) use *Boolean Constraint Propagation* (BCP) for making branching decisions. BCP can result in more accurate branching decisions because it takes into account all the variables that get fixed at a value as a result of the branch. On the flip side, BCP can be time consuming. Freeman (Freeman, 1995) describes BCP in more detail along with some simplifications that can be used to speed up the process, such as binary BCP which only considers constraints having 2 variables. Another approach is to restrict the set of variables using which BCP is performed; an example is the PROP heuristic developed by Li and Anbulagan (Li and Anbulagan, 1997). CLTL is a look-ahead based solver and its branching heuristics use the notion of *trigger equivalence and domination* to restrict the variable set for performing BCP (see Section 4.3).

Heule *et al.* (Heule *et al.*, 2011) show that it can be beneficial to use look-ahead techniques near the root node of the search tree. Branching literature for MIPs also recognizes the importance of making accurate branching decisions near the root of the search tree. Forrest *et al.* (Forrest *et al.*, 1974) and Glankwamdee and Linderoth (Glankwamdee and Linderoth, 2006) note that branching decisions

[1]CPLEX can be configured to search in a strict best-first manner, although this is not its default configuration.

made earlier on in the solution process are more important than the ones made later. Inaccurate branching decisions made near the root of the search tree can double its size with each bad decision (Glankwamdee and Linderoth, 2006; Forrest *et al.*, 1974; Linderoth and Savelsbergh, 1999).

Strong branching is a look ahead based branching technique for MIPs that was introduced in CPLEX 7.5, and is discussed in (Achterberg *et al.*, 2005; Applegate *et al.*, 1995; Refalo, 2004; Klabjan *et al.*, 2001). Like BCP, strong branching is time consuming. Thus it can be useful to restrict the number of variables which are considered as strong branching candidates. Although CPLEX currently does not permit users to select strong branching candidates, we envision that variables shortlisted for BCP can be used as strong branching candidates in the future. Moreover, either BCP or strong branching could be used for making branching decisions for PBOs depending on which one is faster for the given problem. We reiterate that both full strong branching and BCP only use properties of the node under branch.

Patel and Chinneck (Patel and Chinneck, 2007) develop branching strategies for MIPs that also try to eliminate infeasible space from the search tree as quickly as possible, using only the constraint structure at the branching node. However, their strategies are designed for generic MIPs and they do not use SAT branching techniques like CLTL does.

Achterberg (Achterberg, 2004) describes the SCIP solver which integrates MIP and Constraint programming techniques. SCIP includes the *inference history branching rule* which maintains a history of the average number of deductions[2] obtained after branching on a variable. Variables which lead to many deductions are preferred for branching. The goal of inference history branching is similar to our goal, namely

[2]Here, a deduction can be the tightening of another variable's bound as a result of this branch.

to quickly eliminate a large number of infeasible vertices with good objective values (thereby tightening the dual bound). Unlike SCIP however, CLTL bases the branching decision only on the properties of the node under branch, and not on historical information.

4.2.2 Integrating MIP and SAT solution techniques

Several papers have integrated MIP and SAT solution techniques. Devriendt *et al.* (Devriendt *et al.*, 2020) integrate MIP cut generation into a PBO solver that uses conflict-driven clause learning (CDCL). CDCL based PBO solvers are discussed and enhanced by Elffers and Nordstrom (Elffers and Nordstrom, 2018). Wolfman and Weld (Wolfman and Weld, 1999) develop a PBO solver called LPSAT which is based on their LCNF language. LPSAT uses an LP engine to construct *nogood* constraints. Nieuwenhuis (Nieuwenhuis, 2015) shows how clause learning can be beneficial not only for solving Pseudo-Boolean problems, but also generic MIPs. Sandholm and Shields (Sandholm and Shields, 2006) develop a technique for learning nogood constraints for MIPs. A similar approach from Achterberg (Achterberg, 2007) generates nogood constraints and is used to strengthen the constraint set for their SCIP solver (Achterberg, 2004).

In this paper, we show that incorporating SAT branching heuristics into a MIP solver can be beneficial for solving PBOs.

4.3 Trigger Equivalence and Domination

A key component of CLTL is its implementation of *trigger equivalence and domination* for selecting variables with which to perform BCP. Consider a search tree node in a

SAT problem, where setting a variable x_1 to 0 or 1 initiates a "large" amount of BCP. We refer to the variable x_1 and its fixing (say at 0) as a "trigger", represented by $(x_1 \hookrightarrow 0)$. Our challenge is to identify a trigger that causes a large amount of BCP, without explicitly performing BCP on every variable at this node that appears in constraints having exactly two variables.

Given a search tree node where "n" variables x_1 to x_n are not yet fixed to any value, and a trigger $(x_i \hookrightarrow v)$, $i \in \{1..n\}$ and $v \in \{0, 1\}$, CLTL uses the trigger to perform BCP and records all the variable fixings it leads to. The resulting variable fixings are represented by a set of tuples, where each tuple (x_j, v_j) in the set represents the variable x_j fixed to $v_j \in \{0, 1\}$, $j \in \{1..n\}$, $j \neq i$. Observe the following:

(a) if the trigger $(x_j \hookrightarrow v_j)$ leads to the fixing (x_i, v), then the two triggers $(x_j \hookrightarrow v_j)$ and $(x_i \hookrightarrow v)$ are *equivalent*. That is, both triggers (when applied on the constraint set at this node) will lead to the same final set of variable fixings.

(b) if the trigger $(x_j \hookrightarrow v_j)$ does not lead to the fixing (x_i, v), then it must be true that the trigger $(x_i \hookrightarrow v)$ leads to at least as many variable fixings as the trigger $(x_j \hookrightarrow v_j)$.

Therefore, it follows that the trigger $(x_i \hookrightarrow v)$ *dominates* (i.e., always leads to at least as many variable fixings as) any of the triggers (x_j, v_j), $j \in \{1..n\}, j \neq i$. If we know the result of BCP due to the trigger $(x_i \hookrightarrow v)$, we need not perform BCP on the dominated triggers (x_j, v_j), $j \in \{1..n\}, j \neq i$. CLTL uses this observation to limit the number of triggers using which BCP is performed, and picks a variable for branching that leads to the "largest" amount of BCP in either the up or down branch. The following points should be noted :

- A heuristic is needed to measure the amount of BCP caused by a trigger. One option is to simply count the number of variables fixed by the trigger. CLTL's heuristic is similar to Jeroslow-Wang's heuristic, and is described in more detail in Section 4.4.

- When using trigger equivalence and domination to make a branching decision, CLTL only considers the amount of BCP on one side of the branch. It is more common to consider a weighted sum of progress on both sides of the branch, for example by averaging the up and down Pseudo-costs in Pseudo-cost branching (Atamtürk and Savelsbergh, 2005; Achterberg *et al.*, 2005). GAMS/XPRESS can be configured to branch on the variable having the largest Pseudo-cost on either the up or down branch.

- If a trigger dominance relation is established at a node "N", then the relation continues to hold at every descendant of "N". This is because additional branching conditions can only result in more variable fixings. This observation can be used to further speed up the BCP implementation (although we do not use it currently).

4.4 Implementation Overview

Several aspects of CLTL's implementation are key to its performance. These are detailed below.

Branching condition	**List of hypercubes**
$x_1 = 0$	$(x_2 = 0)$, and $(x_3 = 0, x_4 = 0, x_5 = 1)$, and $(x_4 = 1)$
$x_1 = 1$	$(x_3 = 0, x_4 = 0, x_5 = 1)$, and $(x_4 = 1)$, and $(x_2 = 1, x_3 = 0, x_4 = 0)$

Table 4.1: Infeasible hypercubes in the child nodes.

4.4.1 Constraint representation

In preparation for making branching decisions, CLTL converts the constraints at the root node into *infeasible hypercubes* using Boole's expansion theorem. These hypercubes are nogood constraints that represent an infeasible region of the search space. For example, the constraints:

$$x_1 + x_2 \geq 1 \text{ , and}$$
$$x_3 + x_4 + (1 - x_5) \geq 1 \text{ , and}$$
$$3x_1 + 5.1x_2 - 4.2x_3 + 10x_4 \leq 5.4$$

are converted into the infeasible hypercubes:

$$(x_1 = 0, x_2 = 0) \text{ , and}$$
$$(x_3 = 0, x_4 = 0, x_5 = 1) \text{ , and}$$
$$(x_4 = 1) \text{ , and } (x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0)$$

Equality constraints are converted into a pair of upper and lower bound constraints, and Boole's expansion theorem is applied on both. We define the "size" of an infeasible hypercube as equal to the number of variables in it.

Starting from the root node, every node in the search tree stores a list of infeasible hype-cubes. When a node branches on a variable, it passes its list of hypercubes to

both child nodes. Each child node uses the list inherited from its parent and prepares its own list of hypercubes, which it passes on to its own children, and so on.

Continuing with our example, if the parent node branches on variable x_1, then each child node applies its branching condition (and any other discovered variable fixings, as indicated by CPLEX) on the hypercubes inherited from its parent, and ends up with the hypercubes shown in Table 4.1. Although we show hypercubes with a single variable in Table 4.1, such hypercubes are excluded from the list unless the corresponding variable fixing is not discovered by CPLEX. The following points are evident:

i) child nodes discard references to hypercubes that are in conflict with their variable fixings. A child node's variable fixings include its branching conditions, plus any additional variable fixings that may have resulted from BCP.

ii) when the node variable fixings partly match the conditions in a hypercube, a new hypercube is created by omitting the matching conditions. The child node then includes a reference to this new hypercube in its list of hypercubes.

iii) when the node variable fixings do not match any of the conditions in a hypercube, the hypercube is included as it is by the child node in its list of hypercubes (i.e., a new hypercube is not created). This is important to keep CLTL's memory usage within limits.

At each child node, the list of variables that are fixed at either 0 or 1 is obtained from CPLEX. This of course includes the branching conditions of the node, in addition to any variable fixings found by CPLEX. Child nodes inherit their parent node's variable fixings, and only work with their own variable fixings that are in addition to

their parent's fixings.

4.4.1.1 Capacity constraints

CLTL treats capacity constraints from the LTL problem in a special manner. The constraint:

$$a_1 * x_1 + a_2 * x_2 + a_3 * x_3 + \ldots + a_n * x_n - b_1 * y \leq 0$$

where the coefficients a_1 to a_n and b_1 are all positive (with b_1 much larger than all the other coefficients) is represented by the infeasible hypercubes:

$$(x_1 = 1, y = 0)\ ,\ (x_2 = 1, y = 0),\ \ldots \text{ and } (x_n = 1, y = 0).$$

For purposes of branching, CLTL ignores the resulting capacity constraint when $y = 1$. This is because the size of the smallest infeasible hypercube resulting from the constraint when $y = 1$ is quite large, and is therefore unimportant for making branching decisions. This is similar to the behavior of the MOMs heuristic.

In general, CLTL can be configured to ignore some constraints altogether during the process of hypercube collection at the root node (see Appendix B for CLTL configuration). This is needed because the application of Boole's expansion theorem on some constraints can result in an extremely large number of infeasible hypercubes (Aloul *et al.*, 2002a).

4.4.1.2 Merge and absorb

CLTL can be configured to repeatedly *merge* hypercubes that differ in only 1 variable fixing. For example, the hypercubes $(x = 1, y = 0, w = 0)$ and $(x = 0, y = 0, w = 0)$ can be merged into the single hypercube $(y = 0, w = 0)$. Similarly, the hypercube

$(x = 1, y = 0, w = 0)$ can be *absorbed* into the hypercube $(x = 1, y = 0)$. However, these operations are time consuming and CPLEX presolve routines should be used whenever possible.

4.4.2 Boolean constraint propagation

CLTL's BCP implementation first applies the given trigger on hypercubes of size 2. Any resulting variable fixes are repeatedly applied on hypercubes of size 2, until there are no more variable fixings. At that stage, all the variable fixings found so far (including the original trigger) are applied on hypercubes of size 3. Once again, any new variable fixings found are collected. If new variable fixings are found at any stage, CLTL "climbs down" to hypercubes of size 2 (if some of them don't already have all variables fixed), and re-applies the known variable fixings on those hypercubes. CLTL "climbs up" one level only when the current level is not yielding any new fixings. The BCP process is complete when no new fixings are found by applying the known fixings on hypercubes of any size. The pseudo-code for this implementation is shown in Algorithm 2. Note that any variable fixings found due to BCP reduce the size of every hypercube that contains any of those variables.

The BCP implementation halts at once if it detects a conflicting fixing resulting from the trigger, which indicates that this trigger will lead to infeasibilty. In such a case, CLTL instantly branches on the variable included in the trigger, expecting CPLEX to detect infeasibilty in one of the child nodes created by the branch.

4.4.2.1 BCP metric

The purpose of the BCP metric is to construct a measure of progress if the variable in a trigger is selected for branching. CLTL's BCP metric has three components which are calculated as follows:

i) First, CLTL assembles the hypercubes at this node for which every variable gets fixed as a result of this trigger. The volumes of these cubes are summed and added to the metric. The volume of an individual cube is set equal to 2^{-n}, where n is the size of the hypercube. This metric is similar to the one used by Jeroslow-Wang's heuristic. For problems in which every hypercube has the same size, this volume is simply set equal to 1.

ii) Next, CLTL identifies the hypercubes where some variables are not fixed by BCP, but at least one variable is in conflict with the variables fixings resulting from this trigger. The volumes of these hypercubes are summed as well, and added to the metric.

iii) Optionally, our BCP metric also considers hypercubes of size 3 where one of the variable fixings is matching, and no variable is in conflict with the fixings resulting from this trigger. Such hypercubes correspond to infeasible space that has been partly eliminated. In this case, the amount of infeasible space eliminated is set equal to $0.5 * 2^{-3} = 2^{-4}$. We ignore hypercubes of size 4 or more, because the infeasible space they represent is small.

Referring back to our example in Section 4.4.1, the trigger $(x_2 \hookrightarrow 0)$ would fix $(x_1 = 1)$, resulting in the elimination of the hypercube $(x_1 = 0, x_2 = 0)$ which has

volume $= 2^{-2}$. Next, the hypercube $(x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0)$ also gets eliminated because of the mismatch on the value of x_2, even though x_3 and x_4 are not fixed by the trigger. The volume of this cube equals 2^{-4} and is added to the metric. The final value of the metric is therefore 0.3125. Note that our metric overestimates the volume of infeasible space removed by BCP since it assumes that the hypercubes do not have any variables in common, and that their volumes can simply be added together.

By default, CLTL greedily chooses to branch on the variable whose metric has the largest value on one side (we call this the "one-sided BCP metric"). In case there are several such variables, CLTL attempts to break the tie by choosing the variable whose metric in the other branch has the largest value. This metric may not be available for both the up and down branches, if BCP was not performed on the corresponding trigger (recall that some triggers are eliminated from consideration for BCP by trigger equivalence). In such cases, the value of the metric on the missing side is assumed to be zero.

When trigger equivalence is not used, this metric is available for both the up and down branches, and CLTL can be configured to use the sum of the metric on the up and down branches to make the branching decision. We call this the "two-sided BCP metric". Using a weighted sum of the progress on either side (as described in (Achterberg *et al.*, 2005)) is also an option for future implementations.

4.4.2.2 Variables for BCP

By default, CLTL performs BCP with every variable that appears in a hypercube of size 2. However, CLTL includes configuration parameters which can be used to reduce

the number of variables that are considered. These are listed in Appendix B.

Algorithm 2: CLTL's BCP implementation

```
1  Initialization:  level←2, Set S ← the initial trigger.
2  while level ≤ (size of the largest remaining hypercube) do
3  |  Apply variable fixings in S on hypercubes having size = level
4  |  Set U ← new variable fixings found
5  |  Set S ← (S ∪ U)
6  |  if (U is empty) then
7  |  |  level ← level + 1
8  |  else
9  |  |  level ← 2
10 |  end
11 end
```

4.4.3 Heuristics for branching

CLTL's base heuristic for branching is as follows. At the branching node, if any hypercube having exactly 1 variable is discovered, then that variable is branched on at once (this corresponds to CLTL discovering a variable fixing before CPLEX). If there are hypercubes with 2 variables, then BCP is performed on some of these variables and one of them is picked for branching. If there are no hypercubes at a node with 2 or fewer variables in them, we use a variant of MOMS heuristic as described below. If for any reason we cannot arrive at a branching variable decision, we simply default to CPLEX's branching decision. Note that a variable is considered for branching a node only if it is fractional in the linear relaxed solution at the node.

For some MIPs such as the LTL routing MIP, the objective function only includes some of the variables in the model. For such MIPs, CLTL can be configured to branch only on those variables that do appear in the objective function (see Appendix B for a description of the available configuration parameters).

If our heuristics suggest more than one variable for branching at a given node,

then one of these variables is selected at random. This can be a source of performance variability.

4.4.3.1 MOMS heuristic

Our variation of MOMS heuristic counts the frequency of variables in hypercubes having exactly 2 variables. If some variable occurs with the highest frequency in such hypercubes, then it is chosen for branching. In case of a tie, we only use the highest frequency variables as candidates, and check which candidate has the highest frequency among hypercubes having the next largest size. This process is repeated until there is only a single candidate left, or until there are no hypercubes left. A tie that remains unbroken at the end of this process is broken at random.

4.4.3.2 Jeroslow-Wang heuristic (JW)

We have implemented our version of the one-sided Jeroslow-Wang heuristic. Our implementation closely follows the description in Silva (Marques-Silva, 1999), although we only consider hypercubes whose size is $(T + n)$ or less, where n is the number of variables in the smallest size hypercube at a given node. The threshold value T is user configurable and defaults to 10.

4.4.4 Integration with CPLEX

Our strategy for integration with CPLEX is to overrule CPLEX's branching decisions with the branching suggestions from our heuristics for the first few hours. We call this CLTL's "ramp-up" phase. The downside of our approach is that CLTL is slower than CPLEX initially, since BCP is expensive and because we always overrule CPLEX's

branching decision after it has been made. However, experiments indicate that the time invested in making good branching decisions at the outset compensates for the initial time investment (recall that the first few branching decisions are the most important ones).

Since we maintain infeasible hypercubes in every node of the search tree, our approach is also memory intensive. This is another reason why we only use our heuristics for the first few hours before allowing CPLEX to take over completely.

4.4.5 Sources of performance variability

There are several sources of performance variability in CLTL:

i) whenever our heuristics suggest more than one candidate branching variable, the tie is broken using a pseudo-random configuration parameter.

ii) even a few seconds of difference in the time spent on "ramp-up" can influence the number of nodes processed before Cplex takes over completely.

iii) the hyper-cubes collected using Boole's expansion theorem are influenced by the ordering of the variables in the constraints. This can effect branching decisions.

iv) lastly, any performance variability inherent in CPLEX is automatically inherited by CLTL.

4.5 Experimental Evaluation

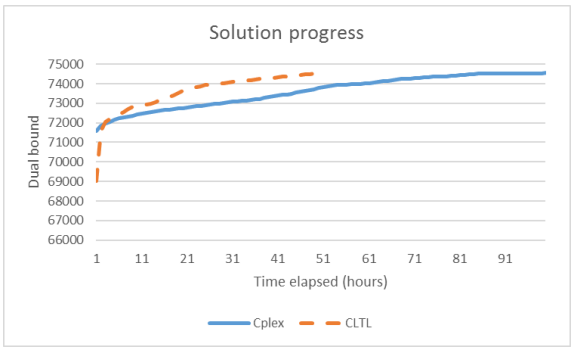
In order to demonstrate the effectiveness of CLTL, we solved some PBOs from MipLib 2010 and 2017 (Koch *et al.*, 2011; Gleixner *et al.*, 2019), LTL routing problems using

real data from our industry partner, and one problem from the SAT 2009 competition used in (Heule *et al.*, 2011). These PBOs have many constraints that can be converted into a small number of infeasible hypercubes using Boole's expansion theorem. Our goal is to show that CLTL can match the performance of CPLEX (version 12.9) on these hard-to-solve instances. The tests were executed on Dell PowerEdge R330 Servers running CentOS 7.4, equipped with 192 GB of RAM and two Intel Xeon E3-1240 v5 3.5GHz CPUs having 8 cores each. These hyper-threaded cores allow CPLEX to execute up to 32 threads on each server.

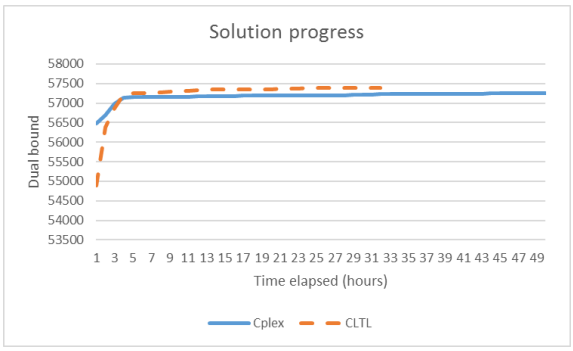
Since CLTL's heuristics are implemented in a CPLEX branch callback, an empty branch callback was used when solving these MIPs with "pure" CPLEX (i.e., without any intervention). This allows for a fair comparison between CLTL and CPLEX.

As mentioned previously, CLTL has several parameters that can be used to control its behavior. In the tests below, we make a note of any parameters that were changed from their default values. Since it is not possible to vary every parameter for every test, we have selected representative results for each MIP. Our parameter choices also offer guidance for future parameter selection when solving other MIPs.

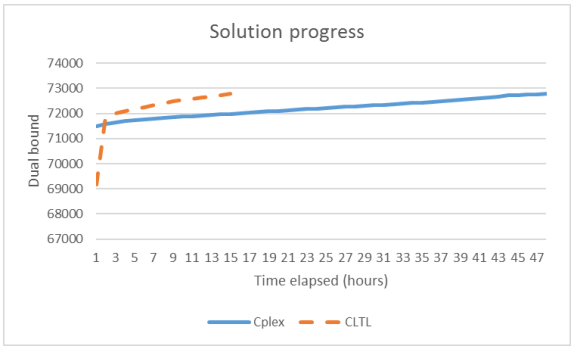
(i) `LTL routing problems` : We solved the LTL routing problem for 3 different days in March 2016 using real data from our industry partner. CLTL was able to solve the problem to provable optimality faster than CPLEX. The results are shown in Figure 4.1(a), (b), and (c). As expected, CLTL is slow to start because it invests time in performing BCP for the first hour. However, the quality of the branching decisions is better than CPLEX (in its default mode), leading to faster convergence.



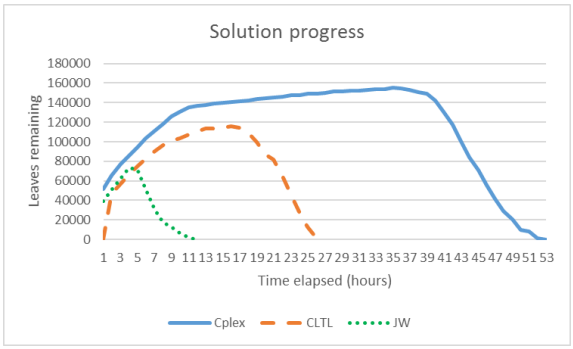
(a) LTL routing, 18 March 2016



(b) LTL routing, 17 March 2016

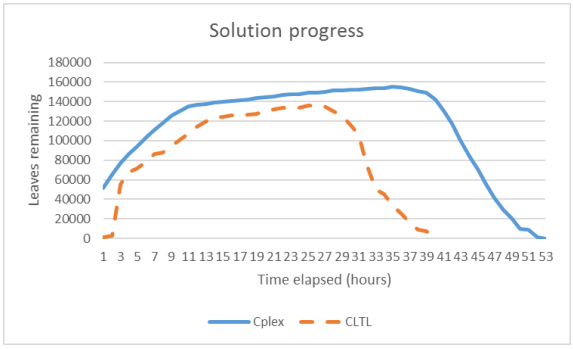


(c) LTL routing, 11 March 2016



(d) 1 hour ramp-up

(e) Performance variability



(f) 2-sided metric with a 2 hour ramp-up

Figure 4.1: Solving LTL routing problems and `eq.atree.braun.12.unsat` with CLTL.

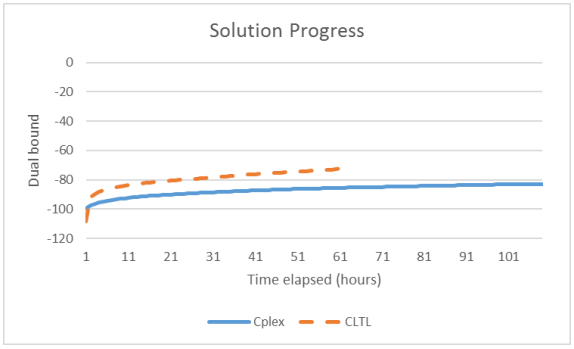
(ii) `eq.atree.braun.12.unsat`: This is an infeasible MIP from the SAT 2009 competition (Heule *et al.*, 2011). CLTL proved that this MIP is infeasible faster than CPLEX (Figure 4.1(d)) when considering the volume removed by partly matched hyper-cubes in the BCP metric. The Jeroslow-Wang heuristic was more effective than both CPLEX and CLTL. Some performance variability was observed with 2 different random seeds used for breaking ties while making branching decisions (Figure 4.1(e)). The 2-sided BCP metric appears to be less effective than the 1-sided metric, even with the ramp-up duration doubled (Figure 4.1(f)).

(iii) `2club200v15p5scn:` This is a feasible Pseudo-Boolean problem from MipLib. Figure 4.2(a) shows that CLTL was able to solve the problem to optimality in about 3 days after a one hour ramp-up. CPLEX (in its default configuration) was not able to solve the MIP to provable optimality even after 4 days.

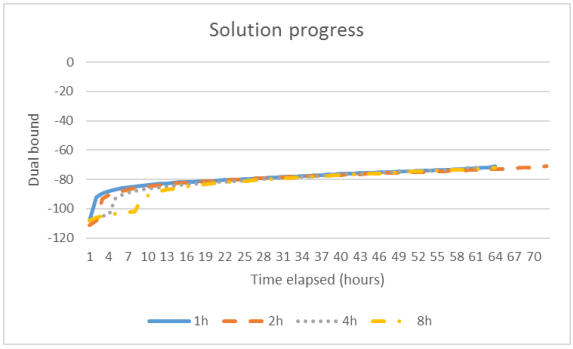
Figure 4.2(b) shows the progress in dual bound when the ramp-up duration was changed. For this MIP, the time to solution did not change considerably with ramp-up duration. Figure 4.2(c) shows that the memory consumption increased with increasing ramp-up duration. This is expected since infeasible hypercubes are held in memory for longer. Memory usage was measured by running the `vmstat` program every six minutes.

Figure 4.2(d) shows very little performance variability with 3 random seeds. Finally, Figure 4.2(e) shows that the 2-sided BCP metric was quite effective for this MIP.

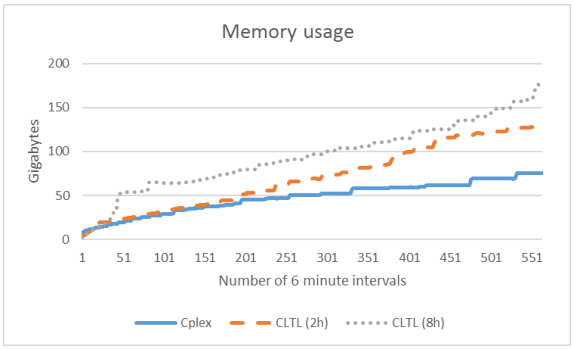
(iv) `reblock354:` This is an extremely hard feasible problem from MipLib. In an



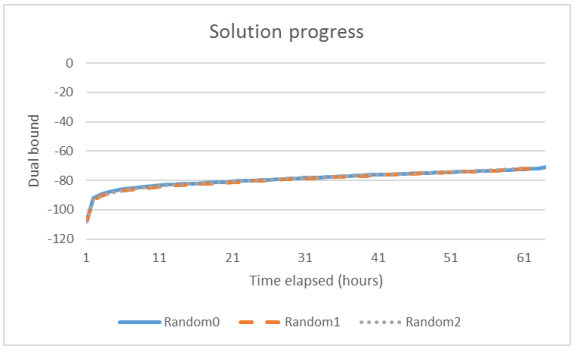
(a) 1 hour ramp-up



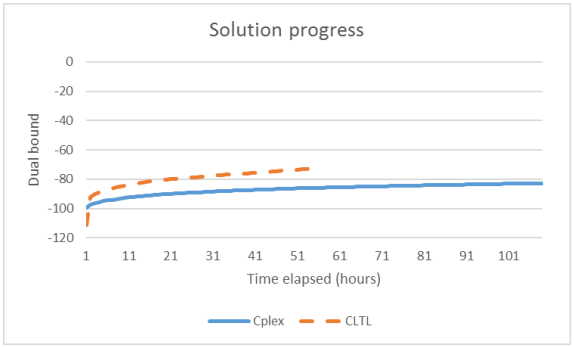
(b) Increasing ramp-up duration



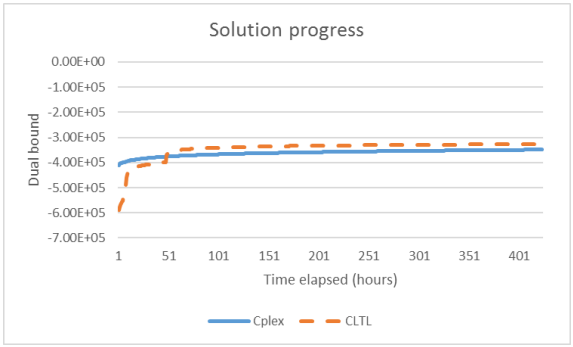
(c) Memory consumption



(d) Performance variability



(e) 2-sided BCP metric



(f) `reblock354`, 48 hour ramp-up

Figure 4.2: Solving `2club200v15p5scn` and `reblock354` with CLTL.

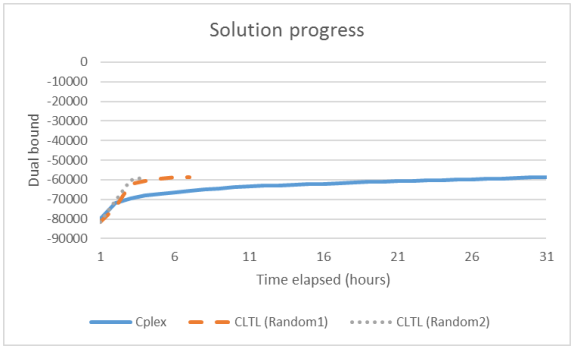
attempt to solve it to provable optimality, we used a huge ramp-up (48 hours). Even then, neither CLTL nor CPLEX could solve the problem to optimality after several days (although CLTL achieved better dual bound than CPLEX). The results are shown in Figure 4.2(f), with a constant (39 million) added to the bound. Observe the huge "boost" to the dual bound at the conclusion of the ramp-up.

CLTL was configured to ignore the capacity constraints in the MIP, effectively treating it like a 2-SAT problem.

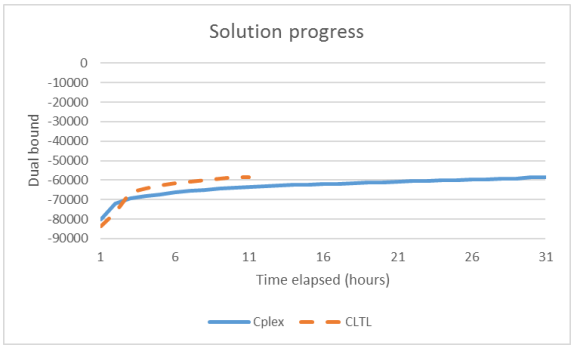
(v) `opm2-z12-s8`: As with `reblock354`, CLTL was configured to ignore the capacity constraints for this MIP. In two runs with different random seeds and a ramp-up duration of 1 hour, we solved this problem to provable optimality under 10 hours (Figure 4.3(a)). With trigger equivalence turned off, we were still able to solve the problem faster than CPLEX but it took almost 10 hours (Figure 4.3(b)). With no BCP, the MOMS heuristic also performed quite well with the ramp-up time configured to 2 hours (Figure 4.3(c)).

With full strong branching used for the first 2 hours, CPLEX solved the problem faster than in its default configuration (Figure 4.3(c)). This underscores the similarities between strong branching and using the results of BCP to make branching decisions.

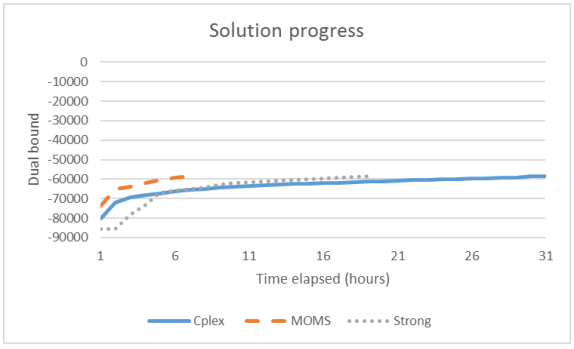
(vi) `opm2-z12-s7`: We were able to solve this MIP to provable optimality faster than CPLEX using 1 hour ramp-ups. As with `opm2-z12-s8`, CLTL was configured to ignore capacity constraints. Figure 4.3(d)) shows the results with trigger equivalence switched on, and then switched off.



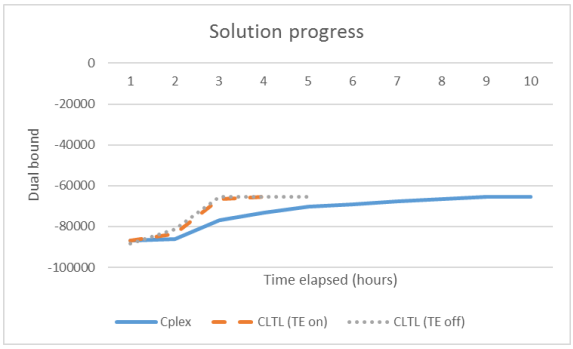
(a) 1 hour ramp-up, 2 random seeds



(b) Trigger equivalence turned off



(c) MOMS heuristic, and full strong branching



(d) Trigger equivalence turned on, and off

Figure 4.3: Solving `opm2-z12-s8` and `opm2-z12-s7` with CLTL.

(vii) `bnatt500`: This is an infeasible Pseudo-Boolean problem from MipLib. We solved this MIP with the ramp-up duration set to 2 hours. CLTL proved infeasibility slightly faster than both CPLEX and Jeroslow-Wang's heuristic (Figure 4.4(a)). When we varied the random seed, some performance variability was seen (Figure 4.4(b)). Figure 4.4(c) shows the results with trigger equivalence turned off and ramp-up duration reduced to 1 hour.

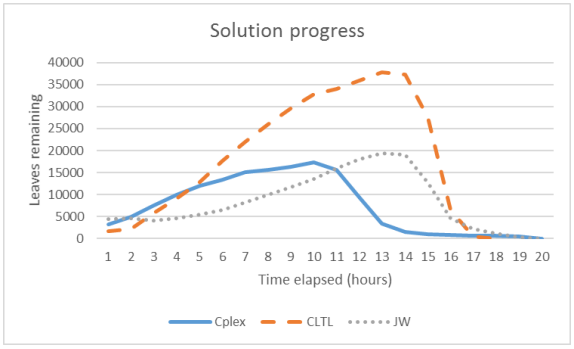
Next, we turned on the switch for considering the volume removed by partly matched hypercubes, which seemed to slow down the computation. The results are shown in Figure 4.4(d).

(viii) `p6b`: This is a feasible 2-SAT problem from MipLib. With a 1 hour ramp-up using 2 different random seeds, the results are shown in Figure 4.5(a). With the ramp-up duration increased to 2 hours, the results are in Figure 4.5(b) where one of the tests turned off trigger equivalence and used the two sided BCP metric. In all four tests, we outperformed CPLEX.

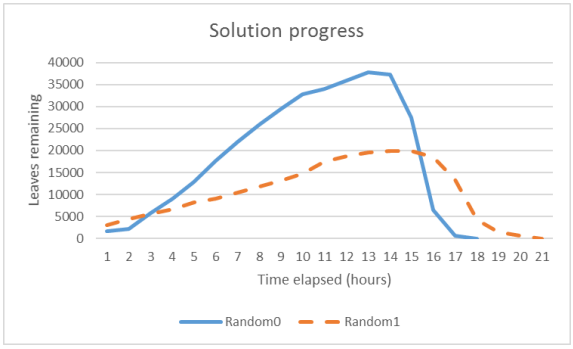
(ix) `seymour-disj-10`: We used the presolved version of this problem, since CPLEX pre-solving eliminates a lot of redundant constraints for this MIP. With a 1 hour ramp-up, the results are shown in Figure 4.5(c).

Next, we increased the ramp-up duration to two hours. By ignoring all constraints that resulted in more than 10 hyper-cubes when Boole's expansion was applied on them, we essentially ignored the larger constraints while making branching decisions. The results are shown in Figure 4.5(d), where we used two random seeds. We outperformed CPLEX in all our tests with this MIP.

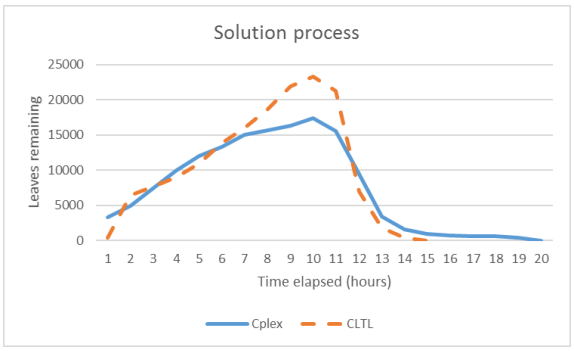
(x) `wnq-n100-mw99-14`: When collecting hyper-cubes for this MIP, we ignored the



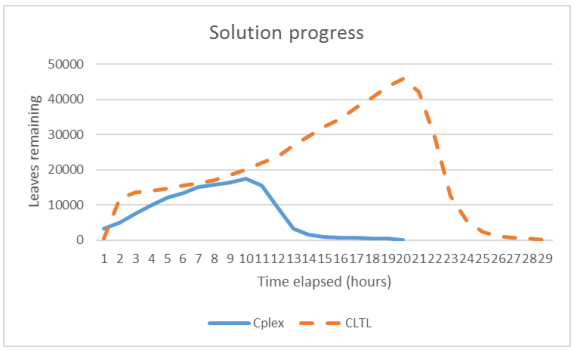
(a) 2 hour ramp-up



(b) Performance variability



(c) 1 hour ramp-up, trigger equivalence switched off

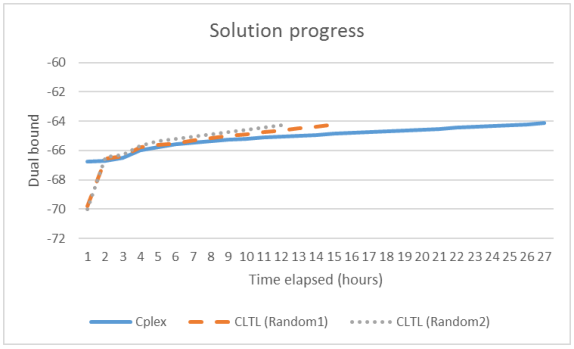


(d) Volume removed by partly matched hypercubes considered

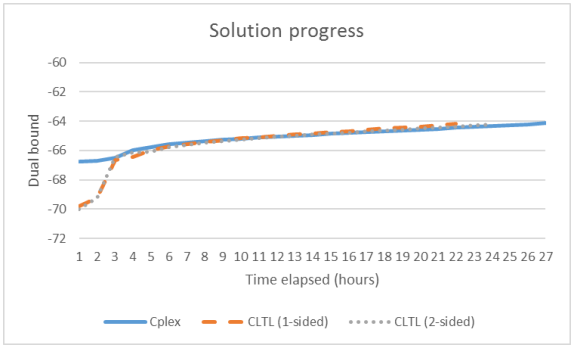
Figure 4.4: Solving `bnatt500` with CLTL.

few constraints that have more than two variables. Then we solved this MIP by performing BCP for 1 hour on variables that occurred with the highest frequency in the collected hypercubes. These simplifications were needed because this MIP has a very large number of constraints.

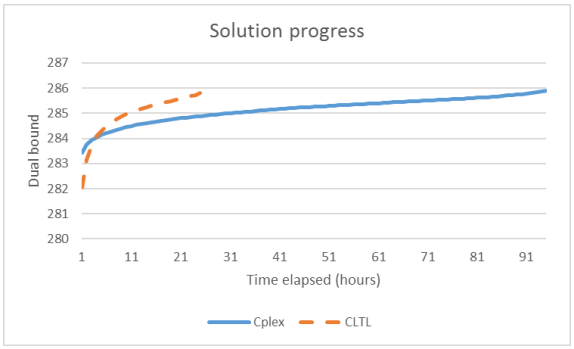
The results are shown in Figure 4.5(e). Note that MOMS heuristic also performed quite well for this MIP. Given the large number of constraints, this problem required a lot of memory to solve (Figure 4.5(f)). Note the sudden drop in memory usage after the first hour (*i.e.*, when the ramp-up was halted).



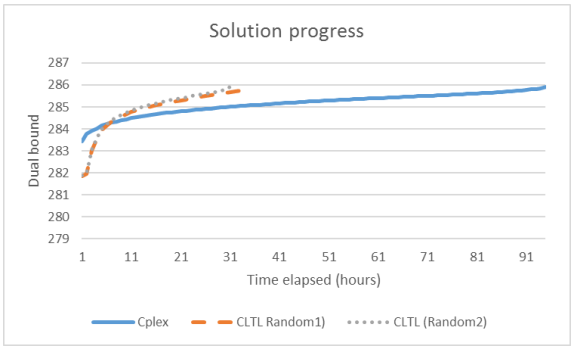
(a) 1 hour ramp-up



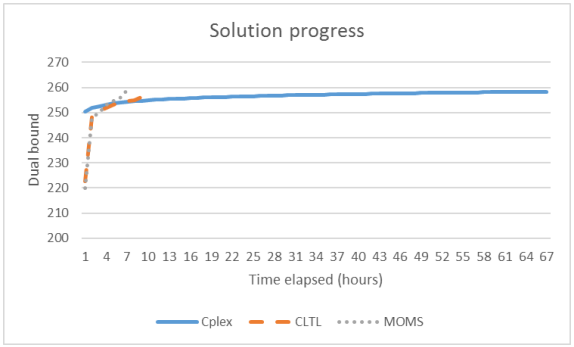
(b) 2 hour ramp-up



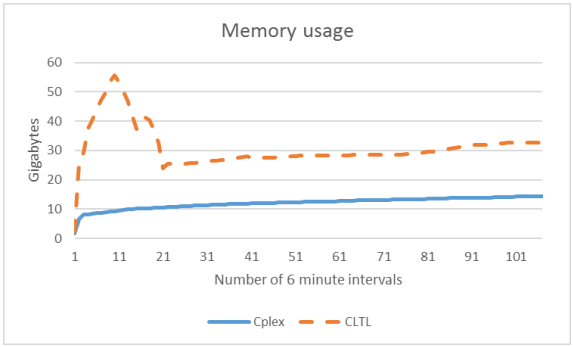
(c) 1 hour ramp-up



(d) 2 hour ramp-up



(e) 1 hour ramp-up



(f) Memory consumption

Figure 4.5: Solving `p6b`, `seymour-disj-10`, and `wnq-n100-mw99-14` with CLTL.

4.6 Conclusions and Future Work

In this chapter we described CLTL, a CPLEX based solver for the LTL routing problem and some Pseudo-Boolean optimization problems. By comparing the performance of CLTL with CPLEX in its default configuration, we demonstrated that CLTL can outperform CPLEX on some hard-to-solve instances. We conclude that SAT branching techniques can be quite effective for MIPs which have a constraint structure similar to the problems we solved. Using SAT branching heuristics (and BCP) to make branching decisions can compliment MIP lookahead techniques such as strong branching.

CLTL is able to make good branching decisions independently of CPLEX using only the properties of the node under branch. This can be useful when solving hard problems in a distributed fashion. Implementing a distributed version of CLTL remains a future work item for us.

Chapter 5

Merging leaves from the search tree of a mixed integer program

5.1 Introduction

In some implementations of distributed MIP, the workers are not assigned collections of individual leaf nodes. Instead, each worker is assigned a single tree that contains a subset of leaf nodes from the frontier. This approach to MIP distribution is called *subtree parallelism* (Ralphs *et al.*, 2018), and its variations have been used in DryadOpt (Budiu *et al.*, 2011) and ParaSCIP (Shinano *et al.*, 2011, 2016b).

Subtree parallelism offers some advantages over the sequencing approach of Figure 3.2 used in Chapter 3. For example it allows the workers to assign their tree to the MIP solver as a single unit, thereby freeing the workers from the responsibility of having to sequence leaf nodes. Another benefit is that there is no need to periodically exchange cutoff and other information between the subproblems assigned to a worker. Ralphs *et al.* (Ralphs *et al.*, 2018) provide a more detailed discussion.

In this chapter, we analyze two simple schemes which can be used to implement subtree parallelism. The first scheme, which we call the *LCA scheme*, is based on the notion of a *Lowest Common Ancestor* introduced by Aho *et al.* in (Aho *et al.*, 1976). This scheme is somewhat restrictive and we identify the conditions under which it can be used. The second scheme, which we call *Controlled Branching*, extends the LCA scheme and overcomes its restrictions.

The rest of the chapter is organized as follows. In Section 5.2 we review existing literature as it pertains to our work. In Section 5.3 we analyze the LCA scheme. This is followed by a description of the Controlled Branching scheme in Section 5.4. Section 5.5 includes an outline of our implementation and details our testing strategy. We present results from our experiments in Section 5.6, and conclude in Section 5.7.

5.2 Literature Review

Our work in this chapter can be viewed as an extension of the leaf merging strategies used in ParaSCIP (Shinano *et al.*, 2011, 2016b) and DryadOpt (Budiu *et al.*, 2011). ParaSCIP uses the Ubiquity Generator (UG) framework (Shinano, 2018) to distribute SCIP over an MPI (Message Passing Interface) based cluster. It merges leaf nodes into a single node by loosening the bounds on some variables that may be tightly bounded in some of the leaves. In effect, this approach discards some work that is already complete by moving up in the search tree from a given subset of leaf nodes to their LCA node. This can result in a significant amount of work being repeated when the LCA node is used instead of the leaf nodes in the subset. As a safeguard, ParaSCIP uses the LCA node only when the linear relaxation objective of the LCA node does not change significantly as a result of the merging. As we show in this

paper, our approach for merging complements ParaSCIP in situations where simply using the LCA node is not an option.

ParaSCIP concludes that the benefits of merging leaf nodes are due to the "re-rooting" of the problem which usually results in the MIP solver reapplying pre-solve routines on the merged node. In fact, for a minimization problem the linear relaxation objective of the merged node can even increase due to the reapplication of pre-solve routines. In our experiments (Section 5.6) we use pre-solved versions of the MIPs and disable CPLEX pre-solve routines in order to show that merging leaf nodes can be beneficial even in the absence of pre-solving. Moreover, we disable solution finding heuristics which can introduce an element of luck into the computation.

ParaSCIP is the latest implementation in a family of projects (Shinano and Fujie, 2007; Shinano *et al.*, 2011, 2016a, 2003, 2016b, 2008). In this family, ParaLEX (Shinano and Fujie, 2007; Shinano *et al.*, 2008) distributes CPLEX over an MPI based cluster using the master-worker paradigm (Ralphs *et al.*, 2018). ParaLEX selects an unsolved leaf node from a central node pool and assigns it to a worker that is idle.

DryadOpt (Budiu *et al.*, 2011) is an implementation of distributed MIP on the data-parallel cluster Dryad (Isard *et al.*, 2007). DryadOpt merges several leaf nodes selected at random from the frontier into a single subproblem. However, unlike in our approach DryadOpt still selects one leaf node at a time from this merged subproblem and assigns it to a user-defined solver method. In contrast, *we delegate node sequencing entirely to CPLEX and assign the merged subproblem to it as a single entity requiring solution.* The leaf selection heuristics implemented by DryadOpt are depth-first and breadth-first, and it alternates between them depending on how many leaf nodes are pending.

CHiPPS (Xu *et al.*, 2009) is a framework for distributing tree search algorithms. It includes a library called ALPS (Xu *et al.*, 2005), which can be used to implement distribution frameworks for specific problem domains, such as distributed MIP. Instead of moving individual leaves between computers, ALPS moves entire subtrees where each leaf is represented as a difference (*i.e.* its branching condition) from its parent. Since ALPS is designed for data intensive applications, it is concerned about the compactness of subtree representation during data movement. Our schemes are also useful for compacting data that needs to be migrated between computers.

ALPS requires application developers to implement methods for sequencing leaf nodes, and also for processing each leaf. In contrast (and as already mentioned), we assign a single merged subproblem to CPLEX and delegate node sequencing to it entirely. ALPS includes a generic MIP solver called ABC (ALPS branch-and-cut) that implements best-first sequencing of leaf nodes. In addition to ALPS, CHiPPS includes BiCePS (which supports relaxation based branch-and-bound algorithms) and a library called BLIS that is derived from it (which provides functionality for solving MIPs). Detailed descriptions can be found in (Xu *et al.*, 2009).

Many implementations of distributed MIP maintain node pools of pending subproblems. Crainic *et al.* (Crainic *et al.*, 2006) categorize pool management strategies according to where nodes from the frontier are held. One approach is to have a distributed strategy where every computer holds some frontier nodes (for example, in PEBBL (Eckstein *et al.*, 2015) every worker can have its own pool). In SYMPHONY (Ralphs *et al.*, 2003), BOB++ (Galea and Le Cun, 2007) with its global priority queues, and ParaLEX and ParaSCIP, frontier nodes are managed in a centralized manner.

The reader is referred to papers by Aho *et al.* (Aho *et al.*, 1976), Harel *et al.* (Harel and Tarjan, 1984), and Bender *et al.* (Bender *et al.*, 2005) for an exposition of the lowest common ancestor (LCA) concept. The idea of merging several leaf nodes into a single tree using LCA nodes is similar to the advanced parallelization technique of *tree contraction* mentioned in Bader *et al.* (Bader *et al.*, 2005), who also note the benefits of hybrid node sequencing heuristics.

5.3 The LCA Scheme

The LCA scheme simply substitutes a collection of leaf nodes with their Lowest Common Ancestor node. A worker process in a distributed MIP computation can then be assigned the LCA node instead of assigning it the collection of leaves. By doing so, some computation that is already complete is wasted (equivalently, some work will be repeated once the LCA node is assigned to a MIP solver by the worker). As an example, the subset of leaf nodes $S = \{$`Node7`, `Node10`, `Node12`, `Node16`, `Node17`$\}$ from the tree in Figure 3.1 can be substituted by the *nonleaf node* `Node4`. The wasted computational effort is represented by the time needed to branch `Node4` and its descendants in order to arrive at the leaf nodes in S.

Of course, such substitution cannot always be made. We need to carefully choose which leaves to include in every subset that is selected for merging. For example, the two subsets of leaves $Q = \{$`Node10, Node16`$\}$ and $T = \{$`Node12, Node17`$\}$ cannot both be chosen for merging from the tree in Figure 3.1 with the intention of assigning them to workers in a distributed MIP computation. This is because the LCA node of one subset is an ancestor of the LCA node of the other subset.

In another example, if we choose to merge the leaves in subset $R = \{$`Node7,`

`Node14, Node18`, `Node19`} and replace them with their LCA node `Node2`, then we have the following two problems:

1. There are two nodes (`Node5` and `Node8`) under the LCA node that have only one active branch. We do not know exactly how much time was needed to fathom the missing branches. If these branches took a long time to fathom (or if there are many such branches), then simply using the LCA node `Node2` could result in a massive amount of repeated work when `Node2` is assigned to a MIP solver.

   ParaSCIP addresses this problem indirectly by checking the change in the linear relaxation objective as a result of substituting the leaves with their LCA node (see Section 3.2). However, sometimes even a small change in the linear relaxation objective can represent several hours worth of computation. Moreover, ParaSCIP's safeguard becomes ineffective when there are few or no variables in the objective function. We present a more robust safeguard in Section 5.3.1.

2. The LCA node also includes several leaf nodes that we do not intend to include in the set R, namely nodes `Node10, Node12, Node16,` and `Node17`. This problem is similar to the previous one – only this time we have an unwanted branch (the left branch of `Node4`) instead of the missing branches. In our analysis, we treat unwanted branches like missing branches that need to be avoided.

5.3.1 Packing Factor

Our observation is that *individual LCA nodes are safe to use when they have no descendant nonleaf nodes having only one branch (or having an unwanted branch).* For such LCA nodes, the amount of wasted computational effort due to merging can

be accurately estimated, and is in fact equal to the time taken for branching the LCA node and all its nonleaf descendants. This time is linear in the number of leaf nodes being merged, as we prove shortly.

It is useful to define a metric for each LCA node that we call the "packing factor".

Definition 1. *The packing factor of an LCA node is the number* (N) *of nonleaf nodes (including itself) in the subtree rooted at the LCA node, divided by the number of leaf nodes* (L) *in the subtree that we intend to merge. The packing factor of a subtree is the packing factor of its root node.*

Definition 2. *An LCA node is perfect if it has a perfect packing factor* $N/L = 1 - (1/L)$*. In other words, it has the property that* $N + 1 = L$*.*

Proposition 1. *In a binary tree with* L *leaf nodes, the number of nonleaf nodes* $N \geq L - 1$*. Therefore,* $N/L \geq 1 - (1/L)$*. Equality holds when every nonleaf node has two branches.*

Proof. The proof is by induction. Clearly, the proposition is true when the root node branches into at most two child nodes. Subsequently, to increase the number of leaf nodes by one, an existing leaf node must branch into two leaves, thereby also adding to the count of nonleaf nodes by one. Therefore, whenever the count of leaf nodes increases by one, the count of nonleaf nodes also increases by one. Hence, the inequality $N \geq L - 1$ holds throughout. If any node branches to a single child, then N increases without increasing L, and the inequality becomes a strict inequality. On the other hand, if every nonleaf node branches twice, then the equality $N = L - 1$ is maintained throughout.

If a leaf node having a sibling is pruned from the tree, then L decreases without

changing N. If the pruned leaf had no sibling, then L and N both decrease by one. In either case, the inequality continues to hold. □

Corollary 1. *In a binary tree with N nonleaf nodes and ($L = N + 1$) leaf nodes, there cannot be any nonleaf nodes with only one branch.*

Proof. The proof is by contradiction. Assume that a binary tree B with N nonleaf nodes and L leaf nodes has a perfect packing factor, and also $K > 0$ nonleaf nodes with a single branch. Any node N_1 with a single branch can be skipped over by branching its parent directly to its child. By doing so, we eliminate N_1 from the tree, and decrease the value of N by 1 without altering the value of L. Continuing in this manner, we can eliminate all nonleaf nodes from the tree that have only one branch, eventually reducing the number of nonleaf nodes to $(N - K)$. At the end of this process, we have a binary tree all of whose nonleaf nodes have two branches. From Proposition 1, this tree must have a perfect packing factor, so $(N - K) + 1 = L$. For any $K > 0$, it therefore cannot also be true that $N + 1 = L$, contradicting our assumption that the root node of B was perfect. □

Observe that whenever a leaf node gets pruned from a binary tree, all the ancestors of the leaf node are rendered imperfect (*i.e.*, they cannot have perfect packing factors).

5.3.2 Reconstructing LCA Nodes

LCA nodes are represented by the branching conditions that need to be applied to the original MIP. In a distributed MIP computation, one way for a worker to reconstruct an LCA node is to change the variable bounds accordingly in the original MIP model (this is the approach we use in our tests). The downside of this approach

is that feasible solutions found by one worker are not feasible on any other worker (recall that every subproblem is necessarily conflicting in at least one variable bound). For the same reason, solution finding heuristics can become less effective because a feasible solution in one region of the feasible space cannot be "grown" into a better solution in another region of the feasible space. On the other hand, this approach to reconstruction is compatible with CPLEX dynamic search since it can be implemented without using control callbacks. Moreover, cut generation and pre-solve reductions can be reapplied on every node.

An alternate way to create an LCA node is to branch to it from the root node of the original MIP, instead of modifying the original model (a single, multi-variable branch can be used). This ensures that the feasibility region of every subproblem in a distributed MIP computation is the same as that of the original MIP, which can be useful when warm starting the workers.

5.3.3 Finding Perfect LCA nodes

A simple algorithm for decomposing a given search tree into a collection of perfect LCA nodes (and any leftover leaves) is shown in Algorithm 3 (Appendix C). The method `GetLCANodes()` is invoked on the root node in order to identify all the perfectly packed nodes. Applying this algorithm on the tree in Figure 3.1 yields the perfect LCA nodes `Node4` and `Node11`, and the leftover leaf `Node3`.

This algorithm assumes that the tree structure is available in memory (see Section 5.5 for details), and that the linear relaxation objective of every leaf node is available from the MIP solver (the linear relaxation objective is useful for best-first sequencing). Since every branch in the tree is traversed twice (moving downwards on

lines 10 and 11, and then back upwards with the return statement on line 32), the time complexity of this algorithm is $\mathcal{O}(N)$ where N is the number of nonleaf nodes in the tree.

In general, the difficulty with using the LCA scheme is that large search trees may have very few perfect LCA nodes that represent a sufficiently large number of leaves. A significant amount of time may have been spent establishing the infeasibility of certain branches. During dynamic load balancing in a distributed MIP computation, it is important to assign an idle worker a large portion of the remaining work. In such situations the Controlled Branching scheme can be better suited for distributing the workload. We describe this scheme next.

5.4 Controlled Branching

Controlled Branching is a scheme for combining an arbitrary subset of leaf nodes into a single tree. Our objective is to create a subproblem having exactly the leaves we want, *and then assign this subproblem in its entirety to CPLEX as a single unit requiring solution*. This scheme is useful when a given set of leaf nodes cannot simply be substituted by their LCA node.

Controlled Branching exploits the fact that the branching conditions for every leaf node are already known; so we can use these known branching conditions to arrive at the leaves we want by branching on multiple variables at a time if necessary, skipping past any nonleaf nodes that have leaves of interest on only one side.

Logically, this scheme works as follows. Let LCA_S be the LCA node of the leaves in a given subset S of the frontier. Let LCA_L and LCA_R be the LCA nodes of the leaves in S to the left and (respectively) to the right of LCA_S. From LCA_S, two

branches are created, one to LCA_L and the other to LCA_R. This procedure is applied recursively at LCA_L and LCA_R. If there is only one leaf from S on a given side of any LCA node, then that leaf is created and the recursive step is not applied on that side. At the end of this process, we have a single tree whose frontier includes all the leaves in S.

We reiterate that the purpose of Controlled Branching is to skip over all the nonleaf nodes that have only one branch of interest (*i.e.*, nonleaf nodes that have leaves from S on only one side). Therefore, the recursive step in the procedure above is not needed under any intermediate nonleaf node that already has a perfect packing factor. We present an example later in this section. For now, we state some propositions regarding trees created with Controlled Branching:

Proposition 2. *Controlled Branching creates trees where the number of leaf nodes L exceeds the number of nonleaf nodes N by one, i.e. $N + 1 = L$.*

Proof. By construction, every nonleaf node in a tree created with Controlled Branching is an LCA node that branches into two child nodes (each of which is either another LCA node, or a leaf). From Proposition 1, such a tree will have $N + 1 = L$. □

Corollary 2. *In a tree created with Controlled Branching, there are no nonleaf nodes that have a single branch (they have all been skipped over).*

Proof. This follows from Proposition 2 and Corollary 1. □

Proposition 3. *Controlled Branching creates a tree having the specified set of leaves by computing the smallest number of linear relaxations possible.*

Proof. From Proposition 1, it follows that the smallest number of linear relaxations

that must be computed to create a tree with L leaves is $N = L - 1$. Therefore, this proposition follows from Proposition 2. □

The reader should note that nonleaf nodes in a tree created with Controlled Branching are nodes that had imperfect packing factors in their tree of origin. If chosen for assignment to a worker in a distributed MIP computation, such nodes may include a massive amount of work that was carefully skipped over by Controlled Branching. We present a solution in Section 5.4.1.3.

5.4.1 CB Instruction Trees

To create a tree that only has the leaves we want, we encode a set of instructions into a tree format we call the *CB instruction tree*. A worker in a distributed MIP computation can force its MIP solver to execute these instructions in order to create its search tree. The procedure for constructing CB instruction trees appears in Section 5.4.1.1, and the procedure for executing them in Section 5.4.1.2.

5.4.1.1 Constructing CB Instruction Trees

The CB instruction tree is a recursive data structure each node of which minimally includes:

- the node ID of the LCA node it represents.
- the node ID of the LCA node (or leaf) on either side of this node.
- the branching condition required to arrive at the LCA node (or leaf) on either side. Note that the branching condition may involve more than one variable.

- the CB instruction tree (if any) to its left, and to its right.

We illustrate the procedure for constructing CB instruction trees with an example. Suppose we want to merge leaves in the set R = {`Node7, Node14, Node18, Node19`}. The root node of the CB instruction tree represents the LCA node `Node2`. It stores the ID of the LCA node (namely the string "Node2"), plus instructions on how to branch to `Node7` on its left and to `Node11` on its right. This is achieved by traversing the first branch on the appropriate side, and then skipping past all the nonleaf nodes that have only one branch of interest. Also, when a node is skipped over, continued direction of movement is indicated by the side which does have leaves of interest. We continue accumulating the branching conditions as we skip past nodes. When we reach a leaf node, or a nonleaf node "P" that has leaves of interest on both sides, we halt and record the accumulated branching instruction. We then apply the same procedure on "P" so that we can identify the branches it needs to make, and so on.

In our example, the left side branching instruction stored at the root of the CB instruction tree is the branching condition from `Node2` to `Node4`, appended with one more branching condition (that from `Node4` to `Node7`, since `Node4` needs to be skipped over). Similarly, the right side branching instruction stored at the root of the CB instruction tree is the branching condition from `Node2` to `Node5`, appended with two more branching conditions (from `Node5` to `Node8`, and from `Node8` to `Node11`, because both `Node5` and `Node8` need to be skipped over).

This procedure is repeated at every nonleaf node that is not skipped over, until the CB instruction tree has instructions for creating every leaf in the set R. The resulting CB instruction tree is shown in Figure 5.1(a). In our implementation we recognize

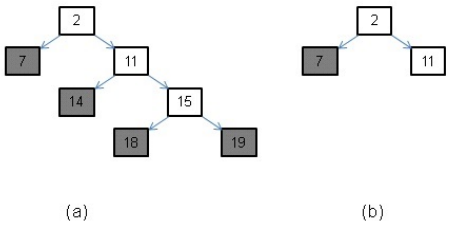


Figure 5.1: Two equivalent Controlled Branching instruction trees for set R={`Node7, Node14, Node18, Node19`}

.

that `Node11` has a perfect packing factor, and do not assemble any instructions for branching it. The resulting CB instruction tree is shown in Figure 5.1(b), where `Node11` appears as a leaf node.

If there are two more sets of leaves we want to merge, say Q = {`Node10, Node16`} and T = {`Node12, Node17`}, then the same procedure can be repeated to construct the CB instruction trees for these two sets. The resulting CB instruction trees are shown in Figure 5.2. Observe that unlike in the LCA scheme, there are no restrictions on how the sets Q and T can be composed (as long as a chosen leaf node appears in only one set).

The pseudo-code for constructing CB instruction trees appears in Appendix D. The first step (shown in Algorithm 4) is to start from each selected leaf and move up towards the subtree root, marking each nonleaf node (N) encountered along the way with the following "reference counts":

- the number of selected leaves on either side of N. We call these the left (and right) leaf reference counts for N.

- the number of nonleaf nodes that will be encountered on either side N while

Figure 5.2: Controlled Branching instruction trees for sets Q={`Node10, Node16`} and T={`Node12, Node17`}.

moving up from the selected leaves to N. We call these the left (and right) nonleaf reference counts for N.

These reference counts are used by the methods shown in Algorithm 5. To create the CB instruction tree, we invoke the method `CreateInstructionTree()` on the root node of the MIP. This method uses reference counts to identify perfectly packed nonleaf nodes, and does not assemble any instructions for branching them (on line 12). Also note that we start from the root node of the MIP, and create a single branch from it to the LCA node of the selected leaves (in case the LCA node is not already the root node of the MIP).

These algorithms can also be used for decomposing a tree into perfect LCA nodes, and are much faster in practice than Algorithm 3 because reference counts are used to identify perfectly packed nodes on the downward path (Algorithm 5).

5.4.1.2 Executing CB Instruction Trees

Logically, the procedure for executing CB instruction trees is as follows. We start by computing the linear relaxation of the node corresponding to the root node of the CB instruction tree. When the node is ready to branch, the MIP solver's branching decisions are replaced with the branching instructions for the left and right children from the root node of the CB instruction tree. We make a note of the node IDs of the

child nodes thus created, and the corresponding child node IDs in the CB instruction tree. Recall that the node IDs in the CB instruction tree are from the tree of origin.

As long as there is a leaf node in the (under construction) merged tree whose node ID corresponds to a nonleaf node in the CB instruction tree, the MIP solver is redirected to one such node. Once again, when this node is ready to branch, the MIP solver's branching decisions are replaced with those from the corresponding node in the CB instruction tree. *When this process is complete, we have a single tree with all the desired leaves; and we can stop controlling the MIP solver's node selection and branching.*

We have implemented this logic as follows:

(i) to each node of the (under construction) merged tree, we attach a CB instruction tree which the node uses to overrule CPLEX's branching decisions. In turn, each child node is attached with the CB instruction tree it needs to use, and so on. Recall that the CB instruction tree is a recursive data structure, so every parent knows the CB instruction tree each of its children should use.

(ii) if CPLEX selects a leaf node where the branching instructions in the attachment do not require an overrule of CPLEX's branching (equivalently, if this node corresponds to a leaf node in the CB instruction tree), then we redirect CPLEX to another leaf (if any) where the branching instructions do require overruling CPLEX. Our merge operation is complete when none of the leaves in the (under construction) merged tree require CPLEX's branching to be overruled.

In practice, the second step of this procedure can result in CPLEX's node selection being overruled frequently. Moreover, this implementation is not safe for use

in multi-threaded environments. An alternate merging scheme that addresses these issues appears in Section 5.4.1.3.

5.4.1.3 Deferred Execution of CB Instruction Trees

In this variation, we do not control CPLEX's node selection. As in Section 5.4.1.2, CB instruction trees are attached to nodes in the (under construction) merged tree. If CPLEX selects a node having an attachment, its branching decision is overruled as per the attached CB instruction tree. If the selected node has no attachment, then CPLEX's branching decision is accepted. This strategy is equivalent to embedding the search tree with the knowledge that certain branches need not be explored again, and overruling CPLEX's branching *"just in time"*.

Deferred Execution allows us to treat every node in the (under construction) merged tree like a normal node, with predetermined branching conditions attached to nodes that correspond to nonleaf nodes of the CB instruction tree. Such nodes can even be migrated to another worker in a distributed MIP computation *as long as the CB instruction tree attached to the node is migrated along with it, and is used to continue overruling CPLEX's branching.*

5.4.1.4 A CB Instruction Tree Variant

Recall that the branching instructions in the CB instruction tree are multi-variable branches (this is what enables us to skip over several nonleaf nodes with a single branch). For example, the branch from `Node2` to `Node7` in Figure 5.1(a) requires branching on two variables at once. A variation of this strategy is to branch on only one variable at a time exactly as in the original tree of Figure 3.1, instantly

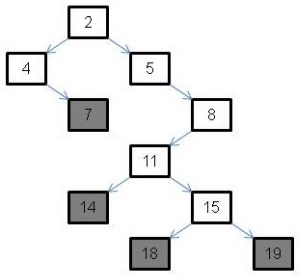


Figure 5.3: CB Instruction tree for set R={`Node7, Node14, Node18, Node19`} using only one variable per branch.

pruning the unwanted leaf. This variation can be useful for MIP solvers that only allow branching on one variable at a time.

The resulting CB instruction tree is shown in Figure 5.3. Testing with this variation is a future work item for us. Observe that the progression from Figure 5.3 to Figure 5.1(a) to Figure 5.1(b) is an example of tree contraction (as described in Bader *et al.* (2005)).

5.4.2 A Note about Overruling CPLEX Branching

We conclude this section by noting that there is no need to compute the linear relaxations of the nodes while overruling CPLEX's branching. This can result in significant time savings for some MIPs where computing the linear relaxation is time consuming. Information from the tree of origin could be supplied as a warm start, including any information that was gained by computing linear relaxations of nodes. However, CPLEX currently does not permit such an implementation .

5.5 Implementation Outline and Test Sets

In our implementation, we first ramp-up the MIP using CPLEX control callbacks. The control callbacks enable us to record the following information:

- the node ID of each child and the branching condition used to create it.
- a reference from each child node back to its parent.

Our ramp-up can be configured to run for a certain amount of time, or until the number of leaf nodes exceeds a threshold. Its objective is to run for a long time and create a search tree with a very large number of leaf nodes. The ramp-up is single threaded, and the heavy use of control callbacks can make it slow for some MIPs.

At the end of the ramp-up we travel upwards from each leaf and build references from parent nodes to their surviving child nodes. This organization makes it easy for us to traverse the ramped-up tree at will. The best solution found (if any) during the ramp-up is recorded and used as a starting cutoff for the following test sets:

(a) **The Optimality Test-set**: The objective of these tests is to establish the utility of the Controlled Branching scheme for large search trees when the user's emphasis is on proving optimality. First, the MIP is ramped-up for a few hours and all the leaves are merged using Controlled Branching. Then the original MIP and its merged counterpart are both solved for several hours with the best solution found during ramp-up (if any) applied as a cutoff. CPLEX emphasis is set to either "best bound" or "optimality", and the improvement in the dual bound is recorded after every hour.

The merged tree is expected to improve the dual bound faster for MIPs in which a lot of time was spent fathoming branches during the ramp-up. CPLEX

is restricted to its single-threaded mode for these tests so that the ramp-up duration can be compared to the solution duration.

(b) **The Feasibility Test-set**: The objective of these tests is to demonstrate the utility of the LCA scheme when CPLEX's emphasis is configured to "hidden feasibility". Popular sequencing heuristics like best-first and depth-first are not suitable for identifying nodes that might lead to feasibility. Therefore, it is useful to merge the leaf nodes into a single tree and delegate node sequencing entirely to CPLEX. Note that the estimated objective value of each leaf node can be obtained using CPLEX control callbacks (for "best-estimate-first" sequencing). However, it is impractical to get this value for every leaf given the sheer number of leaves.

It is expected that searching for feasible solutions in the merged tree will yield better solutions faster when large sections of the ramped-up tree have been fathomed due to several leaf nodes being infeasible. On the other hand, simply using the LCA node (with any solution found during ramp-up applied as cutoff) should be more efficient when the search tree has very few fathomed branches.

(c) **The Balanced Test-set**: The objective of these tests is to highlight some of the drawbacks of sequencing through a collection of leaf nodes. First, the MIP is ramped-up to a large number of leaves. Then the ramped-up up tree is decomposed into a collection of perfect LCA nodes (and any leftover leaves) using Algorithm 3. This collection of nodes is sequenced as per the flow chart of Figure 3.2 using the best-first heuristic. CPLEX's MIP emphasis is set to "balance optimality and feasibility".

Next, the deferred merging scheme from Section 5.4.1.3 is used with CPLEX's emphasis configured once again to "balance optimality and feasibility". The MIP is solved for a few hours, with the progress in the dual bound and the best solution found recorded after every hour. It is expected that better progress will be made when using the merged tree, for reasons we describe in Section 5.6.3. CPLEX is allowed to use up to 32 threads for this test-set.

5.6 Experimental Evaluation

We tested with several hard MIPs from MipLib's Benchmark collection Gleixner *et al.* (2019); Koch *et al.* (2011) using pre-solved versions of the problems. As mentioned in Section 3.2, CPLEX solution finding heuristics and pre-solve routines were disabled. The time quantum was set to 6 minutes when sequencing subproblems using the best-first heuristic. Pseudo-cost branching was used throughout. The MIP `supportcase10` was solved with cut generation disabled and pseudo-reduced cost branching in order to increase its branching rate.

Our tests were conducted on Dell PowerEdge R330 Servers running CentOS 7.4, each equipped with 192 GB of RAM and two Intel Xeon E3-1240 v5 3.5GHz CPUs having 8 cores each. These hyper-threaded cores allow CPLEX to run 32 threads on each server. The `vmstat` program was used to measure memory consumption.

All our tests are written in Java 1.8 using CPLEX 12.10 APIs. Our source code is on the public internet, and is available upon request along with the log files from our test runs.

5.6.1 Results from the Optimality Test-set

The results from our Optimality test set are shown in Table 5.1 and Table 5.2. The CB instruction tree execution scheme from Section 5.4.1.2 was used. All the times are in minutes.

The first three columns show the ramp-up duration, the number of leaf nodes in the search tree, and the number of nodes explored during ramp-up. The next two columns show the time needed for merging all the leaves from the ramped-up tree, and the leaf count in the merged tree. Table 5.1 also shows the number of times CPLEX's node selection was redirected during merging.

The next two columns show the time for which CPLEX was used to solve the original MIP and the dual bound it achieved. The two columns after that show the amount of time spent solving the merged tree and the best bound achieved. Recall that any solution found during ramp-up is applied as a cutoff for both.

The time allocated for solving the merged tree was equal to (or less than) the time allocated for solving the original MIP *minus the time needed for merging*. Note that the allocated times were all multiples of 5 hours, with more time allocated to harder problems. MIPs that were solved to completion (*i.e.*, to infeasibility or optimality) are marked with a ($^+$) sign. Tests in which solving the original MIP improved the bound faster are marked with an asterisk (*).

The last column of Table 5.2 also shows the *improvement* in the solution found by Controlled Branching over the solution found by solving the original MIP. A negative value in this column would indicate that Controlled Branching found the better (*i.e.*, lower) solution.

For some MIPs we used the deferred execution scheme from Section 5.4.1.3 without

MIP	Ramp-up			Merge			Original		CB	
	Time	Leaves	Nodes	Time	Leaves	Redirects	Time	Bound	Time	Bound
bab6*	600	26008	27111	40	894	883	600	-284336.29	540	-284336.52
LTL 11 March 2016	1500	3535	3717	168	635	629	600	109159.68	420	109206.76
neos-3656078-kumeu	600	3469	3851	189	734	702	600	-13375.63	360	-13364.12
neos-3754480-nidda	600	7596600	7644415	5	62069	61926	600	-83776.24	600	-52819.34
neos-4338804-snowy	300	1859134	2044608	91	172113	172111	600	1449.39	300	1449.93
neos-4387871-tavua	600	63701	64331	2	2038	2015	600	29.33	600	29.69
neos-5114902-kasavu	600	10193	10212	98	30	22	600	634.33	480	639.16
nursesched-medium-hint03*	600	6882	6913	34	648	629	600	101.87	540	98.94
opm2-z10-s4	600	51	54	204	15	11	600	-42509.74	360	-42218.62
	900	6029	6045	447	34	27	900	-41216.90	420	-35368.46
radiationm40-10-02	600	5272	5304	47	295	292	600	155325	540	155326
sorrell3+	300	5139	42596	9	2566	2560	300	-17.6	232	-16
	600	3813	199298	8	1905	1900	300	-17.6	40	-16

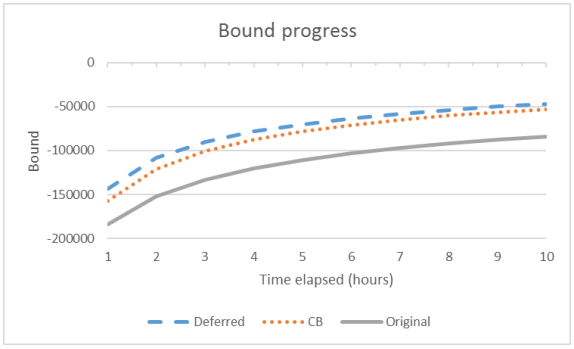
Table 5.1: Controlled Branching (emphasis on best bound)

MIP	Ramp-up			Merge		Original		CB		
	Time	Leaves	Nodes	Time	Leaves	Time	Bound	Time	Bound	Solution
cryptanalysiskb128n5obj14+	1500	12	4215	64	12	8294	Not Applicable	4365	Not Applicable	Not Applicable
neos-5093327-huahum	300	7568	9265	203	7143	600	5425.9	360	5511.51	24
roi5alpha10n8*+	600	227145	228939	3	1795	517	-52.3275	600	-55.97	2.60
s100	600	4876	5146	31	817	600	-0.170195	540	-0.170158	0.00
sing44*+	300	48530	88827	44	19013	476	8128424.11	926	8128528.67	70.49

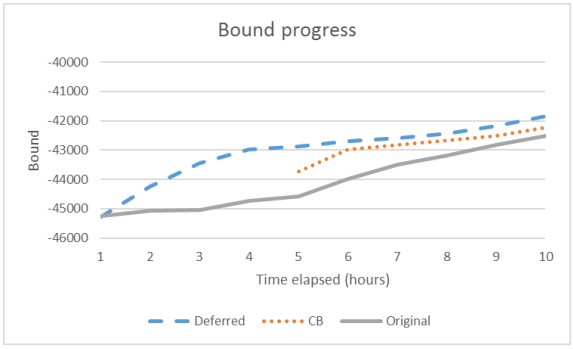
Table 5.2: Controlled Branching (emphasis on optimality)

MIP	Ramp-up			Original			CB (Deferred)			
	Time	Leaves	Overrides	Time	Nodes	Bound	Time	Nodes	Bound	Pending
neos-3656078-kumeu*	600	3468	733	600	6135	-13375.63	600	5647	-13379.12	0
neos-3754480-nidda	600	7631961	62268	600	4629997	-83776.24	600	7345566	-46899.77	0
neos-4338804-snowy	300	1852114	171619	600	3496158	1449.39	600	4125332	1450.78	154
nursesched-medium-hint03*	600	6953	670	600	4373	101.87	600	4642	99.51	0
opm2-z10-s4	600	51	14	600	39	-42509.74	600	61	-41846.50	0
proteindesign121hz512p9	1500	862	532	1500	84	1441	420	59	1442	3
sing44+	300	48563	19090	476	1013543	8128424.11	432	648343	8128480.30	0
splice1k1+	1800	14851	9979	1200	21090	-1498.86	1094	108858	-394	0
sorrell3	300	5140	2565	300	35519	-17.6	300	115321	-17.09	0
traininstance2	600	170443	84213	600	748470	217.43	600	781649	242.35	159

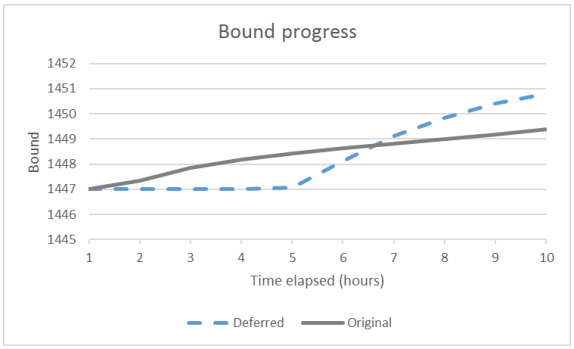
Table 5.3: Controlled Branching (Deferred execution of CB instruction trees)



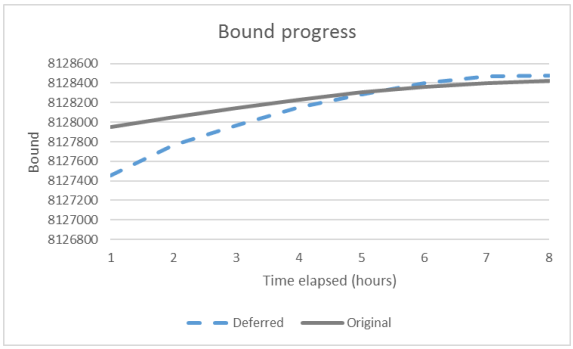
(a) `neos-3754480-nidda`



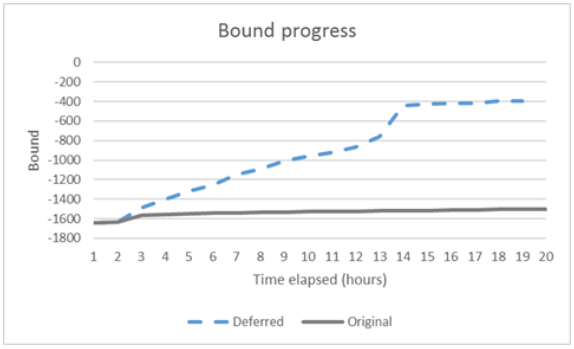
(b) `opm2-z10-s4`



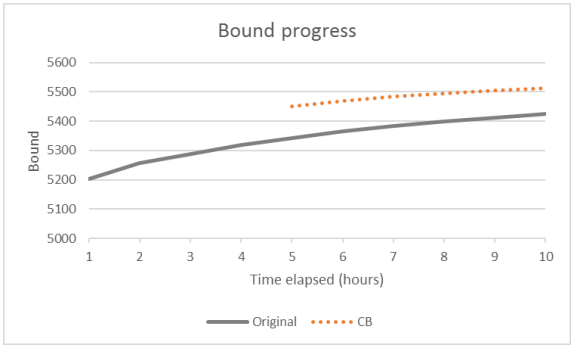
(c) `neos-4338804-snowy`



(d) `sing44`



(e) `splice1k1`



(f) `neos-5093327-huahum`

Figure 5.4: Progress in the dual bound with Controlled Branching.

changing the MIP emphasis used for solving them in Table 5.1 or Table 5.2. In addition, `proteindesign121hz512p9` and `traininstance2` were solved with emphasis set to "best-bound" while `splice1k1` was solved with emphasis set to "optimality". The results are shown in Table 5.3, where the "Overrides" column shows the number of nonleaf nodes in the CB instruction tree. The "Pending" column shows the number of search tree nodes that were not automatically selected by CPLEX, but would have resulted in CPLEX's branching being overruled had they been visited. The "Nodes" columns show the number of nodes explored.

Our tests indicate that using the merged tree is more effective than restarting the search from the LCA node when there is a large percentage of pruned leaves in the search tree (*e.g.,* `cryptanalysiskb128n5obj14`), or when the linear relaxations take a long time to compute (*e.g.*, `opm2-z10-s4`). More importantly, the position of the pruned leaves in the search tree should be such that the fathomed branches include a large fraction of the explored nodes. If these conditions are not met, then it is better to simply use the LCA node (as in ParaSCIP) instead of investing time in Controlled Branching. For example, the search trees for `bab6`, `neos-3656078-kumeu`, `nursesched-medium-hint03`, and `roi5alpha10n8` had very few pruned leaves, making Controlled Branching an inferior alternative to simply using the LCA node with the cutoff applied.

Another observation from our tests is that Controlled Branching is not very useful when branches are fathomed due to leaf nodes getting cutoff by a feasible solution (*e.g.*, `sing44` with emphasis on optimality). Controlled Branching should be used when a large number of leaf nodes are pruned due to *infeasibility*. For `sing44`, the

performance of Controlled Branching improved significantly when the deferred execution scheme from Section 5.4.1.3 was used. Deferred execution was also quite useful for MIPs in which the linear relaxations were taking a long time to compute (*e.g.*, `proteindesign121hz512p9` and `opm2-z10-s4`) or when the merge was taking a long time to accomplish (*e.g.*, `splice1k1`), and for MIPs with a large number of node redirects in Table 5.1 (*e.g.*, `neos-4338804-snowy`).

We used an LTL routing problem with increased network size as a test case (see Chapter 2, Section 2.4 for a description of how the network size is increased). Controlled Branching was useful for tightening the dual bound faster (see Table 5.1).

Figures 5.4(a) and 5.4(b) show the progress in the dual bound for the MIPs `neos-3754480-nidda` and `opm2-z10-s4`, with and without deferred execution of CB instruction trees. Figures 5.4(c), 5.4(d), and 5.4(e) show the progress for the MIP `neos-4338804-snowy`, `sing44`, and `splice1k1` with deferred execution of CB instruction trees. Figure 5.4(f) shows the progress for `neos-5093327-huahum` without deferred execution, starting from the fifth hour (the merge took about 3.5 hours to complete).

We conclude this section by noting that *our performance comparisons should be interpreted with caution*. CPLEX may not make the same branching decisions every time a MIP is solved, and even a small variation in the set of branching decisions (or in their order) can lead to a vastly different search tree.

5.6.2 Results from the Feasibility Test-set

Results from our Feasibility test-set are shown in Table 5.4, where we compare the LCA scheme with Controlled Branching. The first four columns show the ramp-up

MIP	Ramp-up				Original			CB		
	Time	Leaves	Nodes	Solution	Time	Nodes	Solution	Time	Nodes	Solution
bab2*	600	264360	361108	-357064.35	600	260030	-357478.74	600	462780	-357183.67
					900	396223	-357478.74	900	682318	-357192.28
bab6*	300	107822	226324	-284187.45	120	789255	-284223.77	720	778890	-284218.14
comp21-2idx*	300	29366	32908	100	60	63378	85	480	52545	99
markshare2^{D*}	300	11179400	321988937	10	300	351738292	8	300	325171319	8
					600	699531900	4	600	642429800	5
neos-3754480-nidda^{D}	300	17187342	36862523	13156.59	300	30208273	None	300	36712083	13116.81
					600	48721986	13098.64	600	54628761	13095.95
neos-4954672-berkel^{D}	300	16833508	18178595	2701039	300	6322036	2626733	300	8894220	2615849
					600	14363541	2625793	600	15091849	2615849
nursesched-medium-hint03	300	78098	108504	259	300	122392	164	300	128817	164
s100	600	12605	17942	-0.16955	300	10885	-0.16967	300	13833	-0.16967
sing326*	300	79069	157534	7759247.66	300	45462	7753674.85	300	153841	7754021.49
traininstance2	300	773188	20624037	None	300	42867782	82500	300	33271027	80360
					600	89143818	82500	600	76002105	80360

Table 5.4: Controlled Branching (finding feasible solutions, single-threaded mode).

MIP	Ramp-up				Original			CB		
	Time	Leaves	Nodes	Solution	Time	Nodes	Solution	Time	Nodes	Solution
b1c1s1^{D+}	300	973050	1119382	25123.03	180	4670102	24640.97	180	3919144	24544.25
bab2*	300	91292	163821	-357032.62	240	1062311	-357457.76	240	410856	-357300.47
dws008-01^{D*+}	300	5526678	8474164	39522.77	120	29301069	37412.60	120	30058062	None
					300	Not Applicable	Not Applicable	300	71819927	39060.91
highschool1-aigio^{D*}	600	35350	36235	None	300	121992	321	300	69827	486
markshare2^{D}	600	19696056	628031393	10	300	1160533647	7	300	845331686	7
neos-3754480-nidda^{D+}	300	17196052	36885683	13156.59	300	370015321	12941.74	300	217922932	12941.74
neos-4954672-berkel^{D*+}	300	16874566	18223113	2701039	60	27276747	2612710	240	76915554	2627915
nursesched-medium-hint03*	600	147857	206774	218	300	576505	132	300	264441	148
					600	905744	130	600	579953	134
s100^{D}	600	12551	17882	-0.169552	300	90787	-0.169663	300	105705	-0.169667
splice1k1*	600	20941	48522	-82	300	180293	-350	300	340302	-308
supportcase10^{+}	300	31	54	8	180	100	8	180	69	7

Table 5.5: Controlled Branching (finding feasible solutions, multi-threaded mode).

duration, the number of leaves in the search tree, the number of nodes explored, and the best solution found. The next six columns show the duration for which CPLEX was run with the ramp-up solution applied as cutoff, using the original MIP and then Controlled Branching. The number of nodes explored and the best solution found are also shown.

We repeated some tests with CPLEX allowed to use up to 32 threads after ramp-up. The results are shown in Table 5.5. In both Tables, the time awarded to Controlled Branching was reduced by the amount of time used up for merging. The deferred execution scheme from Section 5.4.1.3 was used for some tests, as indicated by the superscript (^{D}). Tests in which Controlled Branching produced a comparatively inferior solution are marked with an asterisk (*), and tests in which the best possible solution was found are marked with a ($^+$).

In general, MIPs used in this test-set can be broadly classified into the following two categories:

1. MIPs for which there is a large fraction of infeasible nodes in the search tree. These nodes need to be identified and pruned in order to reveal a feasible vertex. MIPs from MipLib's feasibility collection fit into this category. We ramped-up some of these MIPs for a long time in order to eliminate a large number of infeasible nodes from their search tree. However, we still could not find a feasible solutions for them in a reasonable amount of time. Examples include the MIPs `fhnw-binpack4-48`, `fhnw-sq2`, and `gfd-schedulen180f7d50m30k18`.

   The benchmark MIPs `supportcase10` and `traininstance2` appear to fit into this category.

2. MIPs in which very few infeasible nodes are encountered during the search for

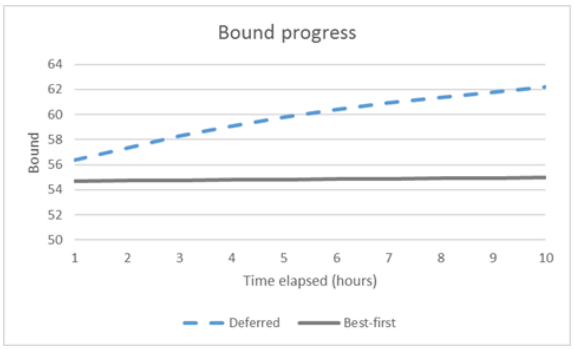
feasible solutions (*e.g.,* `highschool1-aigio`). Sometimes leaves get pruned not because they are infeasible, but because they are cutoff by a solution found during the search (*e.g.,* `sing326`). This phenomenon of a large number of leaves getting cutoff by a feasible solution was also seen occasionally in the Optimality test-set when CPLEX's emphasis was set to optimality (an example is the MIP `sing44`).

Most of the MIPs we used from MipLib's benchmark collection fit into this category. Many of them have multiple feasible solutions, with the lower quality solutions often easy to find.

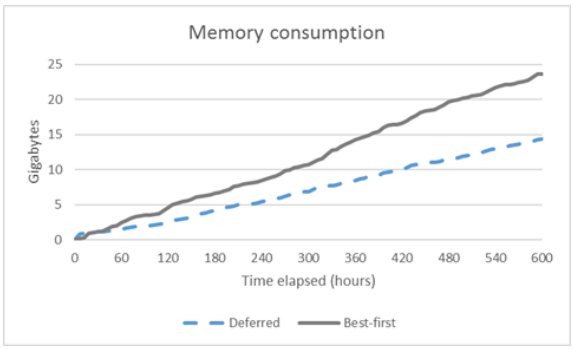
Our tests indicate that it is adequate to simply use the LCA node with the cutoff applied for MIPs in the second category. Controlled Branching may be more useful for MIPs that belong to the first category, so that a large number of infeasible nodes can be skipped over. Testing with MIPs from MipLib's feasibility collection remains a future work item for us, and may require access to more powerful computational resources.

5.6.3 Results from the Balanced Test-set

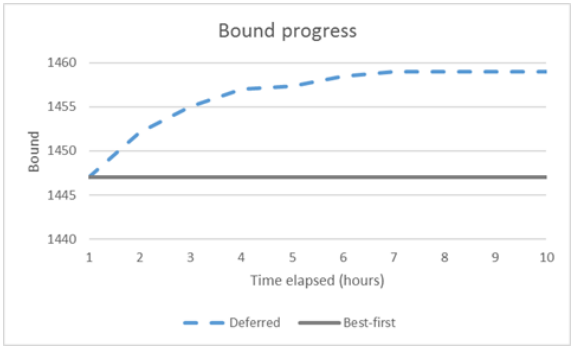
Representative results from our balanced test-set are shown in Figure 5.5. Figure 5.5(a) compares the progress in the dual bound achieved by Controlled Branching with the corresponding progress achieved by best-first sequencing for `comp21-2idx`. Similar comparisons are shown in Figure 5.5(c) to Figure 5.5(f) for `neos-4338804-snowy`, `opm2-z10-s4`, `roi5alpha10n8`, and `s100` respectively. Figure 5.5(b) compares the increase in memory consumption of Controlled Branching with that of best-first sequencing for `comp21-2idx`. The MIPs were ramped-up for 5 hours each.



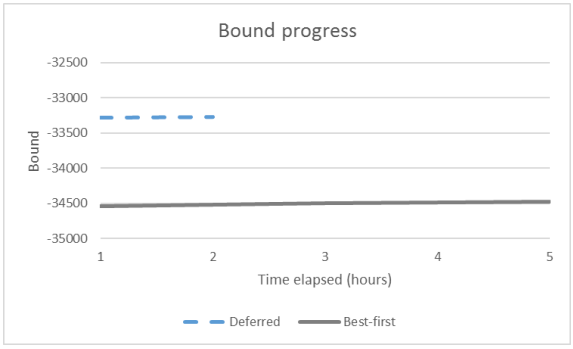
(a) `comp21-2idx`



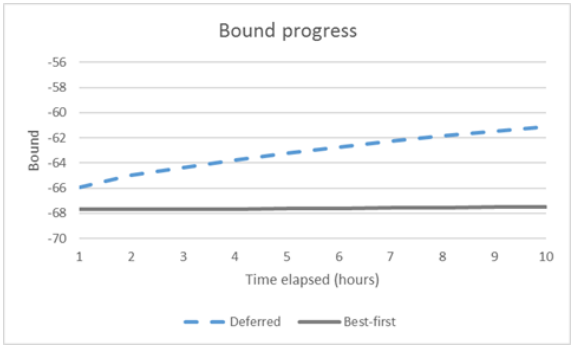
(b) `comp21-2idx memory consumption`



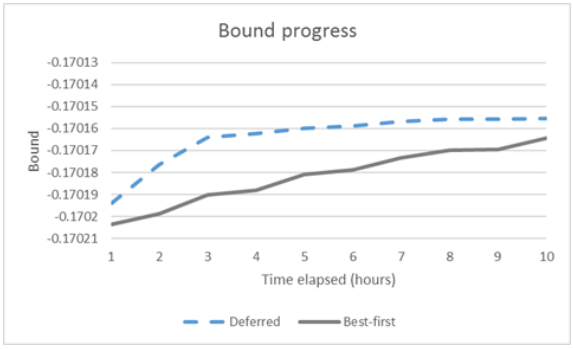
(c) `neos-4338804-snowy`



(d) `opm2-z10-s4`

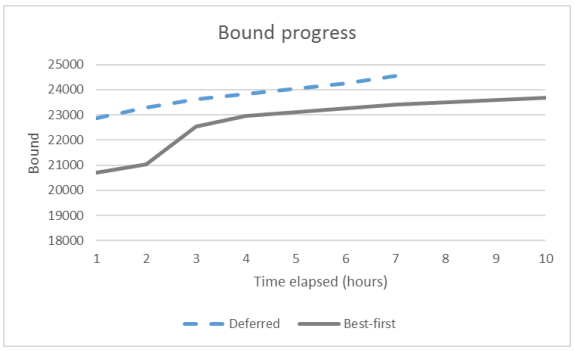


(e) `roi5alpha10n8`

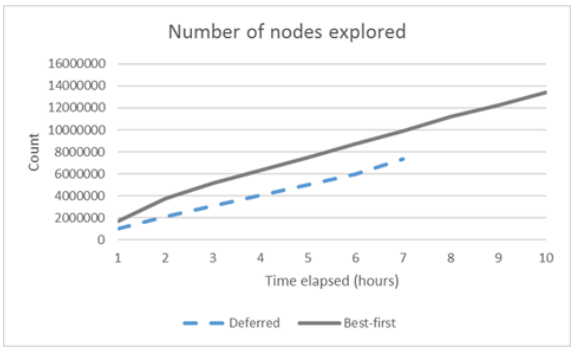


(f) `s100`

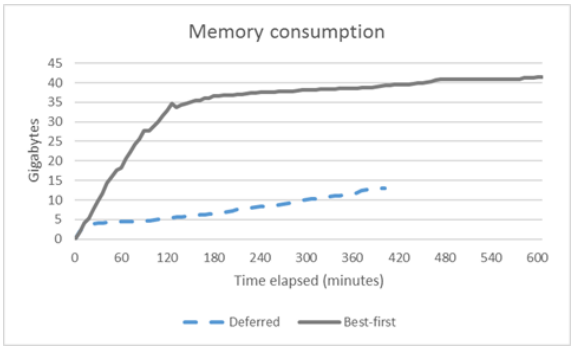
Figure 5.5: Best-first sequencing versus Controlled Branching (balanced emphasis, large ramp-up.)



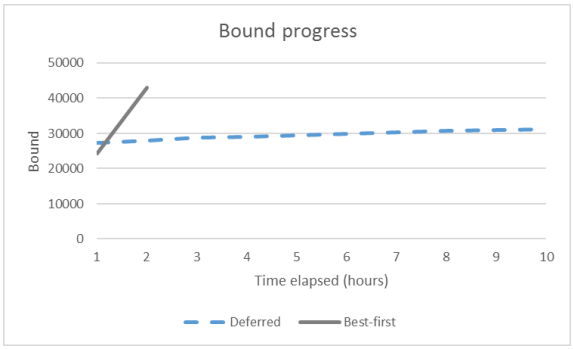
(a) `b1c1s1` bound progress



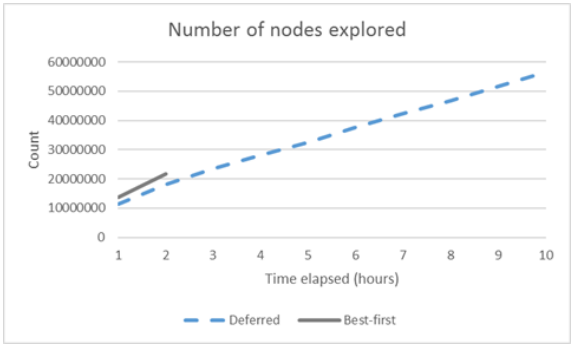
(b) `b1c1s1` nodes explored



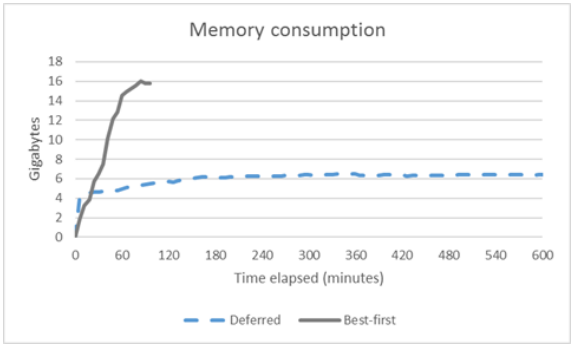
(c) `b1c1s1` memory consumption



(d) `dws008-01` bound progress



(e) `dws008-01` nodes explored



(f) `dws008-01` memory consumption

Figure 5.6: Best-first sequencing versus Controlled Branching (balanced emphasis, small ramp-up).

The key observation from these results is that it becomes practically impossible to iterate through a large collection of LCA nodes in a reasonable amount of time, even if each node represents a trivial subproblem that can be solved to completion within seconds. This was the case for `opm2-z10-s4` where the MIP solver generated new cuts for each LCA node being sequenced, thereby improving its dual bound significantly. But even after 5 hours there were still thousands of LCA nodes awaiting assignment to the MIP solver, and therefore there was almost no improvement in the best bound remaining. Controlled branching on the other hand solved the MIP to provable optimality within 2 hours. This problem was far more severe for `neos-4338804-snowy` because there were more than 8 million LCA nodes with the same linear relaxation objective value of 1447, all of them waiting to be assigned to the MIP solver. The bound progress with best-first sequencing was better for `s100`, although only 37 out of 1626 LCA nodes were solved to completion after 10 hours.

In order to limit the number of subproblems for best-first sequencing, we solved the MIP `b1c1s1` to provable optimality using a small ramp-up (200 leaf nodes). The ramped-up tree was decomposed into 96 LCA nodes and leftover leaves. Figure 5.6(a) compares the progress in the dual bound achieved by Controlled Branching with the progress achieved by best-first sequencing. Figure 5.6(b) compares the number of nodes explored after every hour, and Figure 5.6(c) shows the increase in memory consumption. This test highlights some more drawbacks of sequencing a collection of disconnected subproblems:

1. The total number of nodes explored by best-first sequencing is far larger than the number of nodes explored by Controlled Branching. This is a well known phenomenon in MIP distribution called *redundant work*. Redundant work is a

consequence of partitioning a large search tree into a collection of disconnected subproblems, thereby making it impossible to share information instantly between different parts of the search tree. This in turn leads to larger search trees and sublinear speedups upon distribution (or in our test, increased solution times upon best-first sequencing). Redundant work is discussed in detail by Ralphs *et al.* Ralphs *et al.* (2018, 2003).

2. Memory consumption for best-first sequencing can quickly exceed that of Controlled Branching. This is a consequence of having to retain several large search trees in memory simultaneously during best-first sequencing.

Controlled Branching found a better feasible solution than best-first sequencing for `opm2-z10-s4` and `neos-4338804-snowy`. Both schemes found the best possible solution for `b1c1s1`.

For some MIPs, it is actually quite useful to sequence through a small collection of nodes. An example is shown in Figure 5.6(d) for `dws008-01` which was ramped-up to 200 leaves and decomposed into 90 LCA nodes and leftover leaves. The new cuts generated for each of the 90 subproblems were so strong that all of them were solved to completion within 2 one hour cycles. This is similar to the behavior observed by ParaSCIP, as noted in Section 3.2. However, as with `b1c1s1` the number of nodes explored and the memory consumption were larger when using best-first sequencing for the first two hours (Figure 5.6(e) and (f)). Our conclusion is that Controlled Branching should be used instead of best-first sequencing *unless the generation of new cuts and the reapplication of pre-solve routines on each subproblem quickly results in the MIP being solved to provable optimality.*

5.7 Conclusions and Future Work

In this chapter we analyzed two related schemes for merging leaf nodes from the search tree of a Mixed Integer Program. We extended the leaf merging schemes of ParaSCIP and DryadOpt by using CPLEX to accomplish the merging. We developed a metric called the *packing factor* which is useful for identifying LCA nodes that are safe for assignment to workers in a distributed MIP computation. We proved that an arbitrary subset of leaf nodes can be combined into a single tree using the Controlled Branching scheme, while only computing the smallest number of linear relaxations possible.

Controlled Branching complements ParaSCIP's approach of simply using the LCA node. Our experiments indicate that Controlled Branching is particularly useful when large sections of the search tree have been fathomed due to node infeasibility, or when linear relaxations are taking a long time to compute. Controlled Branching is also superior to best-first sequencing, especially when the search trees are large.

There are several ways in which our work can be extended:

1. Controlled Branching can be integrated into existing implementations of distributed MIP such as ParaSCIP or ParaLEX.

2. The variation of Controlled Branching mentioned in section 5.4.1.4 can be tested with CPLEX, and also with MIP solvers that allow branching on only 1 variable at a time.

3. We have assumed throughout that the ordered branching conditions for each leaf node are available. It would be an interesting exercise to implement our proposals when such ordered branching conditions are not available.

Chapter 6

Summary and Extensions

The Less-than-truckload freight routing problem is interesting not only for its economic impact, but also for its theoretical richness and complexity. In this thesis, we proposed several solution techniques that can be used to solve this big-data driven optimization problem. Our proposals proved quite effective at solving the LTL routing problem in a reasonable amount of time. Moreover, our proposals also proved effective at solving other pseudo-Boolean optimization problems and generic mixed integer programs in a distributed fashion.

Our journey started in Chapter 1 with the long term goal of integrating a powerful commercial optimization engine (CPLEX) with a popular big-data analytics platform (Apache Spark). Our goal was motivated by the observation that many conventional optimization problems arising from day-to-day operations in the industry are increasingly big-data driven. Therefore, it can be beneficial to integrate powerful optimization engines with platforms for big-data analytics.

We selected the LTL routing problem as our use case. This problem was a natural

choice for our project because it exhibits the 4 *V's* of big-data, and also has a significant economic impact. Our collaboration with a major Canadian LTL operator gave us valuable insights into the problems they face during their day-to-day operations.

In Chapter 2, we introduced the LTL routing problem and outlined some of the existing approaches for solving it. We then developed an integer programming model for LTL routing. We showed that this optimization problem can be hard to formulate and solve on larger LTL networks, and developed a heuristic algorithm for solving it quickly.

In Chapter 3, we identified some approaches that can be useful for solving hard optimization problems (like the LTL routing problem) in a distributed fashion. While distribution is an obvious alternative for solving any hard problem, our choice was motivated by the observation that effective distribution is a vital first step towards integrating MIP solvers with big-data platforms. In this chapter, we identified that using branching strategies that only use properties of the branching node can be useful for improved scaling upon distribution. Moreover, we showed how search restarts can be applied to MIP solver parallelization by creating and using a repository of pseudo-Costs.

In Chapter 4 we applied the theme proposed in the previous chapter to pseudo-Boolean optimization problems. We developed an unconventional approach for solving the LTL routing problem. We showed that for some pseudo-Boolean MIPs it is possible to produce performance comparable to CPLEX (in its default mode) by employing branching heuristics that only use properties of the branching node.

In Chapter 5 we continued with our theme of finding effective techniques for distribution. We developed a metric that can be useful while implementing subtree

parallelism, and used it to show that an arbitrary collection of leaf nodes can be merged into a single tree by solving the smallest possible number of node relaxations.

We believe that our proposals represent a significant step towards achieving our long term goal of integrating MIP solvers into big-data frameworks. In the shorter term, our proposals are useful for solving the LTL routing problem and for improving the performance of MIP solver parallelization.

Two proposals for extending our work appear below. These are in addition to the opportunities for future work outlined at the end of each chapter. Early experiments indicate that our proposals are promising and should be explored further.

6.1 A Game-theoretic approach for solving the LTL routing problem

Rahwan *et al.* (Rahwan *et al.*, 2015) present an excellent survey of coalition structure generation. Here, a coalition is a grouping of *agents* which are collectively trying to complete a task. The collection of all the coalitions formed is called the *coalition structure*. Techniques for coalition structure generation are concerned with trying to find the optimal coalition structure; *i.e.*, a grouping of agents that maximizes a profit function that has been defined in advance.

While each individual agent may have its own objective function, many coalition structure generation problems try to optimize *social welfare*. In other words, all the agents have a single *shared objective* and it is beneficial to all of them to optimize its value. Therefore, the LTL routing problem (and possibly other optimization problems) can be modeled as coalition structure generation problems, with the social welfare

being the cost of routing skids received on a given day. A collection of skids that travel together in the same trailer on a given arc can be treated as a coalition.

Many coalition structure generation problems are *Partition function Games (PFGs)*. In such games, the value of a coalition is influenced by other coalitions that make up the coalition structure (Rahwan *et al.*, 2015). The optimal value of social welfare cannot be found unless all possible coalition structures are exhaustively enumerated and the value of each coalition in each structure is calculated. This of course can be very expensive.

In its full generality, the LTL routing problem is a partition function game. To see this, observe that the contribution to the overall routing cost made by a routing decision at one town is influenced by routing decisions made at other towns. However, simplified versions of the LTL routing problem can be modeled as *Characteristic Function Games (CFGs)* (Chalkiadakis *et al.*, 2011), which are usually easier to solve than PFGs (Rahwan *et al.*, 2015).

For example, consider the simplified LTL routing problem with a single origin, multiple destinations, and several intermediate transfer terminals available. Assuming that the destinations and transfer terminals have no towns in common, and that every route to any destination is allowed to have at most 1 transfer terminal, the LTL routing problem becomes a simplified version of the Steiner-tree problem in a time-space network.

Observe that this simplified version of the problem is a CFG. The cost of any coalition is the cost of the first arc traversed (which is unique to the coalition), plus the cost of any other arcs which start from the Steiner point used by this coalition. Hence, there are no common arcs between the coalitions, and the arc costs which determine

the characteristic function value of the coalition depend only on the members of that coalition. There is no sharing of arcs (and therefore of arc costs) between coalitions.

It is conceivable that the LTL routing problem can be solved using a heuristic which decomposes the underlying PFG into a series of easier to solve CFGs. In general, application specific heuristics could be used to approximate hard PFGs with a series of CFGs.

Coalition structure graphs can be used to arrange all possible coalition structures in a tree-like formation (Sandholm *et al.*, 1999; Larson and Sandholm, 2000). The utility of this data structure is that it allows for finding "good" coalition structures (*i.e.*, those that are within a bound of the optimum) without the need for exhaustive enumeration. Similar data structures (and algorithms that use them) are described in (Rahwan *et al.*, 2015).

These approaches may benefit from reorganizing their data structures based on the following observation. For a CFG, the movement of a single agent from one coalition to another leaves the value of all the other coalitions in the coalition structure unchanged. Therefore, calculating the change in the value of a coalition structure can be faster if adjacent coalition structures differ by only one agent that moved between two coalitions.

6.2 Heuristic for set-partitioning problems

The approach of converting a constraint into infeasible hypercubes (see Chapter 4) can be adapted for solving set-partitioning problems which have a small number (say ≤ 100) of variables per constraint. Recall from Section 2.4 that some LTL routing problems had more than a hundred thousand variables and tens of thousands of

constraints, with some constraints being set-partitioning constraints.

For such constraints, the lower bound portion of the equality constraint can be represented by a single infeasible hypercube, and the upper bound portion of the constraint by several hypercubes of size 2. For example, the constraint:

$$x_1 + x_2 + x_3 + x_4 = 1$$

can be converted into the lower bound constraint

$$x_1 + x_2 + x_3 + x_4 \geq 1$$

and the upper bound constraints

$x_1 + x_2 \leq 1$, $x_1 + x_3 \leq 1$, $x_1 + x_4 \leq 1$, $x_2 + x_3 \leq 1$, $x_2 + x_4 \leq 1$, and $x_3 + x_4 \leq 1$

and subsequently represented by the infeasible hypercubes

$$(x_1 = 0, x_2 = 0, x_3 = 0, x_4 = 0) \text{ , and}$$
$$(x_1 = 1, x_2 = 1) \text{ , } (x_1 = 1, x_3 = 1) \text{ , } (x_1 = 1, x_4 = 1) \text{ , } (x_2 = 1, x_3 = 1) \text{ ,}$$
$$(x_2 = 1, x_4 = 1) \text{ , and } (x_3 = 1, x_4 = 1)$$

Observe that valid constraints such as

$$x_1 + x_2 + x_3 \leq 1$$

need not be considered, because the corresponding infeasible hypercubes are absorbed into the hypercubes already used for representing the set-partitioning constraint.

Following this conversion, the BCP implementation from Chapter 4 can be used when making branching decisions. The rationale for identifying and collecting hypercubes of size 2 is to try and eliminate large regions of infeasible space from the problem as quickly as possible (equivalently, to fix a large number of variables to either 0 or 1 as quickly as possible).

Appendix A

Configuration parameters for LTL routing

The configuration parameters that can be used to customize our implementation are shown in Table A.1, and described below.

1. *Maximum route size in number of arcs*: When enumerating all possible routes from an origin break-bulk to a destination, routes having more arcs than this number are ignored.

2. *Maximum arcs in excess of the shortest route*: Routes between any origin break-bulk and a destination can have at most this many arcs more than the shortest route for the same origin-destination pair. Here we define the "shortest route" by considering the number of arcs in the route (and not the sum of the arc transit times).

3. *Schedule window*: The number of days for which skids can be routed. With our data-set, this window is March 1, 2016 to March 21, 2016. Includes an initial

Name	Data type	Default
`Maximum route size in number of arcs`	Integer	4
`Maximum arcs in excess of the shortest route`	Integer	3
`Schedule window (days)`	Integer	21
`Maximum days after the schedule window (Greedy)`	Integer	14
`Maximum CPLEX window (days)`	Integer	10
`Holidays`	List of Integers	Weekends and holidays
`Allow start on holidays`	Boolean	False
`Allow delayed scheduling`	Boolean	True
`Handling cost for cross-docking`	Real number	0
`Maximum number of skids per day`	Integer	5000
`Maximum days after deadline (Greedy)`	Integer	0
`Days added to all deadlines`	Integer	0
`Volume lowering factor`	Real number	0.9
`Trailer volume capacity`	Real number	0.9* 3600 = 3240 cubic feet
`Trailer weight capacity`	Real number	46000 pounds
`Saturation days`	Integer	7
`Maximum number of DSF iterations`	Integer	3
`Sort-to-Bin`	Boolean	False
`Bin volume capacity`	Real number	95 cubic feet
`Bin count capacity`	Integer	60
`CPLEX time limit (in seconds)`	Integer	3600
`CPLEX file strategy`	Integer	3

Table A.1: Configuration parameters for LTL model generation and heuristics

"warm up" period during which the network is saturated.

4. *Maximum days after the schedule window (Greedy)*: The Greedy heuristic is allowed to schedule skids on trailers after the schedule window. This may be required, for example, if routes are being found for skids that were received at EOL terminals on the last day of the schedule window. Note however that

trailer information is available only for March 2016.

5. *Maximum CPLEX window*: For any skid, travel along the last arc must not start any later than this many days after the skid was received at the EOL terminal. This number should be small enough to keep the model size under control.

6. *Holidays*: A list of integers. Indicates how many days past the first day a holiday or weekend falls. Since we start from Tuesday March 1, 2016, the first two numbers in our holiday list are 4 (Saturday) and 5 (Sunday).

7. *Allow start on holidays*: When set, allows trailers to be dispatched on weekends and holidays. In-transit trailers continue traveling on weekends and holidays, and are not effected by this parameter.

8. *Allow delayed scheduling*: Allows skids to be held at break-bulks for a few days, before onward routing.

9. *Handling costs*: Cross-docking at the break-bulks costs \$1.08 per skid. For simplicity, and because these costs tend to be much smaller than trailer rental costs, we default these costs to 0.

10. *Maximum skids per day*: Currently set to 5000.

11. *Maximum days after deadline (Greedy)*: When set to a non-zero number, forces the Greedy heuristic to repeatedly increase a skid's deadline by one day and try to find a route for it, in case the previous routing attempt was unsuccessful.

12. *Days added to all deadlines*: Some days can be added to every skid's deadline.

13. *Volume lowering factor*: Accounts for the fact that some volume in every trailer is lost due to structural restrictions.

14. *Trailer volume capacity*: Set to 90% of 3600 cubit feet.

15. *Trailer weight capacity*: Set to 46 thousand pounds. In practice, trailer weight capacity is usually not a concern (trailer volume fills up much faster).

16. *Saturation days*: The number of days for which skids are routed using the Greedy heuristic, before test measurements are made by routing skids on the next day.

17. *Maximum number of DSF iterations*: Number of iterations of the directed Steiner-forest heuristic, before iterations are stopped and Greedy routing is used.

18. *Sort-to-Bin*: Set to "True" to use Sort-to-Bin.

19. *Bin Volume capacity*: Volume capacity of Bins used during Sort-to-Bin.

20. *Bin count capacity*: Maximum number of skids in a Bin when using Sort-to-Bin.

21. *CPLEX time limit*: Time given to CPLEX to find feasible solutions and prove optimality.

22. *CPLEX file strategy*: CPLEX node files are saved to disk in compressed format for large models.

Appendix B

Configuration parameters for CLTL

Table B.1 summarizes CLTL's configuration parameters and their default values.

1. `Enable Equivalent Trigger Check for BCP`: Used to enable trigger equivalence and domination check while selecting candidates for BCP.

2. `Consider Partly Matched Cubes for Volume Removal`: Refer to Section 4.4.2.1 for a description. In its default position, this parameter places a heavy emphasis on removing hypercubes of size 2.

3. `Enable Two Sided BCP Metric`: Refer to Section 4.4.2.1 for a description.

4. `BCP Level`: Used to select variables from hypercubes of size 2 for performing BCP. The options are to select all of them for BCP, or to select only those that have above average frequency in these hypercubes, or only those that have the maximum frequency. The fourth option is to turn off BCP completely, in which

Table B.1: CLTL configuration parameters

Name	**Data type**	**Default**
ENABLE EQUIVALENT TRIGGER CHECK FOR BCP	Boolean	True
CONSIDER PARTLY MATCHED CUBES FOR BCP VOLUME REMOVAL	Boolean	False
ENABLE TWO SIDED BCP METRIC	Boolean	False
BCP LEVEL	Enumerated	All
CHECK DUPLICATES	Boolean	False
ABSORB COLLECTED HYPERCUBES	Boolean	False
MERGE COLLECTED HYPERCUBES	Boolean	False
MAX VARIABLES PER CONSTRAINT	Integer	Billion
MAX HYPERCUBES PER CONSTRAINT	Integer	Billion
DROP HYPERCUBES LARGER THAN SIZE	Integer	Billion
BRANCH ONLY ON VARIABLES IN OBJECTIVE	Boolean	False
RETAIN CUBES WITH VARIABLES IN OBJECTIVE	Boolean	False
LOOK AHEAD LEVELS (MOMS)	Integer	Billion
DEPTH LEVELS (Jeroslow-Wang)	Integer	10
PERFORMANCE VARIABILITY RANDOM SEED	Integer	0
RAMP-UP DURATION IN HOURS	Integer	1

case the MOMS heuristic is used for branching.

5. `Check Duplicates`: Used to discard hypercubes that are identical to infeasible hypercubes already collected from other constraints.

6. `Absorb Collected hypercubes`: Used to identify and discard infeasible hypercubes if their ancestor hypercubes already exist.

7. `Merge Collected hypercubes`: Used to identify and repeatedly merge sibling infeasible hypercubes into their parent hypercube.

8. `Max Variables per Constraint`: Used to ignore all constraints which include more variables than this parameter.

9. `Max Hypercubes per Constraint`: Used to ignore constraints that cannot be translated into a reasonably small number of infeasible hypercubes using Boole's expansion theorem.

10. `Drop hypercubes Larger Than Size`: Used to discard all hypercubes whose number of variables exceeds this value.

11. `Branch Only on Variables Appearing in the Objective`: Used to exclude variables from consideration for branching if they do not appear in the objective.

12. `Retain hypercubes with Variables Appearing in the Objective`: Used to discard hypercubes none of whose variables appear in the objective.

13. `Look Ahead Levels (MOMS)`: Used to instruct the MOMS heuristic to ignore all hypercubes whose size exceeds this value.

14. `Depth Levels (Jeroslow-Wang)`: Used to instruct Jeroslow-Wang's heuristic to ignore all hypercubes whose size exceeds this value.

15. `Performance Variability Random Seed`: Used to randomly select a variable for branching from a list of candidates, in case of a tie. This is used to test performance variability.

16. `Ramp-up Duration`: The number of hours for which CLTL's branching strategies are used before allowing Cplex to take over completely.

Appendix C

Decomposing a MIP Search Tree into Perfect LCA Nodes

This algorithm assumes that an estimate is available for the linear relaxation objective of each leaf.

Algorithm 3: Decompose a tree into perfect LCA nodes

```
1  Initialization: An empty list PL of perfect LCA nodes
2  Function GetLCANodes(thisNode):
3      isPerfectlyPacked ← false
4      if (thisNode is a Leaf) then
           // every leaf is perfect
5          add thisNode to PL
6          isPerfectlyPacked ← true
7      else
8          if (thisNode has 2 child nodes) then
9
10             leftChildIsPerfect ← GetLCANodes (thisNode.leftChild)
11             rightChildIsPerfect ← GetLCANodes (thisNode.rightChild)
12             if (rightChildIsPerfect and leftChildIsPerfect) then
13
14                 PL.append (thisNode)
15                 PL.remove (thisNode.leftChild)
16                 PL.remove (thisNode.rightChild)
17                 isPerfectlyPacked ← true
18
19                 x ← thisNode.leftChild.linearRelaxationObjective
20                 y ← thisNode.rightChild.linearRelaxationObjective
21                 thisNode.linearRelaxationObjective ← Minimum (x, y)
22             end
23
24         else
               // look for perfect LCA nodes in the only branch
25             if (thisNode has a left child) then
26                 GetLCANodes (thisNode.leftChild)
27             else
28                 GetLCANodes (thisNode.rightChild)
29             end
30         end
31     end
32 return isPerfectlyPacked
```

Appendix D

Algorithms Used in the Construction of CB Instruction Trees

As a first step in constructing CB instruction trees, Algorithm 4 is used for setting reference counts in the nonleaf nodes. Subsequently, these reference counts are used by the methods in Algorithm 5.

Algorithm 4: Set reference counts

```
1  Initialization: S ← leaf nodes selected for merging, with all the refcounts in
     every leaf set to 0.
2  The root node's parent ← NULL.
3  Function SetRefcounts(S):
4      foreach (leaf L ∈ S) do
5          current node C ← L
6          isCurrentNodeALeaf ← true
7          P ← L.parentNode
8
9          while (P is not NULL) do
10             if (C is the left child of P) then
11                 increment left leaf refcount for P
12             else
13                 increment right leaf refcount for P
14             end
15
16             if (not isCurrentNodeALeaf) then
17                 x ← left nonleaf refcount for C
18                 y ← right nonleaf refcount for C
19                 if (C is the left child of P) then
20                     left nonleaf refcount for P ← (1 + x + y)
21                 else
22                     right nonleaf refcount for P ← (1 + x + y)
23                 end
24             end
25
               // Climb up
26             C ← P
27             P ← P.parentNode
28             isCurrentNodeALeaf ← false
29         end
30
31     end
32 return
```

Algorithm 5: Create the CB instruction tree

```
1  Function CreateInstructionTree(thisNode):
2      InstructionTree ← A new CB instruction tree
3      InstructionTree.ID ← thisNode.nodeID
4
5      L ← thisNode's left leaf refcount + thisNode's right leaf refcount
6      N ← 1+ thisNode's left nonleaf refcount + thisNode's right nonleaf refcount
7      isPerfectlyPacked ← false
8      if (L = N+1) then
9          isPerfectlyPacked ← true
10     end
11
12     if (thisNode is neither a leaf nor isPerfectlyPacked ) then
13         if (thisNode's left leaf refcount > 0) then
14             (nextNode, InstructionList) ← GetNextNode (thisNode, "left")
15             InstructionTree.leftChildID ← nextNode.nodeID
16             InstructionTree.leftSideBranchingInstruction ← InstructionList
17             InstructionTree.leftInstructionTree ←
                CreateInstructionTree(nextNode)
18         end
19         if (thisNode's right leaf refcount > 0) then
20             (nextNode, InstructionList) ← GetNextNode (thisNode, "right")
21             InstructionTree.rightChildID ← nextNode.nodeID
22             InstructionTree.rightSideBranchingInstruction ← InstructionList
23             InstructionTree.rightInstructionTree ←
                CreateInstructionTree(nextNode)
24         end
25     end
26 return InstructionTree
27
28 Initialization: Set the branching instruction list InstructionList to empty
29 Function GetNextNode(thisNode, direction):
30     if (direction is "left") then
31         nextNode ← thisNode.leftChild
32     else
33         nextNode ← thisNode.rightChild
34     end
35     InstructionList.add(branching condition of nextNode)
36
37     while (nextNode has a 0 leaf refcount on exactly one side) do
38         if (nextNode's right leaf refcount is 0) then
39             nextNode ← nextNode.leftChild
40         else
41             nextNode ← nextNode.rightChild
42         end
43         InstructionList.add(branching condition of nextNode)
44     end
45 return (nextNode, InstructionList)
```

Bibliography

Achterberg, T. (2004). SCIP-a framework to integrate constraint and mixed integer programming.

Achterberg, T. (2007). Conflict analysis in mixed integer programming. *Discrete Optimization*, **4**(1), 4–20.

Achterberg, T. (2009). Scip: solving constraint integer programs. *Mathematical Programming Computation*, **1**(1), 1–41.

Achterberg, T., Koch, T., and Martin, A. (2005). Branching rules revisited. *Operations Research Letters*, **33**(1), 42–54.

Aho, A. V., Hopcroft, J. E., and Ullman, J. D. (1976). On finding lowest common ancestors in trees. *SIAM Journal on computing*, **5**(1), 115–132.

Akyilmaz, M. O. (1994). An algorithmic framework for routing LTL shipments. *Journal of the Operational Research Society*, **45**(5), 529–538.

Aloul, F. A., Ramani, A., Markov, I. L., and Sakallah, K. A. (2002a). Generic ILP versus specialized 0-1 ILP: An update. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 450–457. ACM.

Aloul, F. A., Ramani, A., Markov, I., and Sakallah, K. (2002b). PBS: a backtrack-search pseudo-boolean solver and optimizer. In *Proceedings of the 5th International Symposium on Theory and Applications of Satisfiability*, pages 346–353.

Applegate, D., Bixby, R., Chvátal, V., and Cook, W. (1995). *Finding cuts in the TSP (A preliminary report)*, volume 95. Citeseer.

Atamtürk, A. and Savelsbergh, M. W. (2005). Integer-programming software systems. *Annals of operations research*, **140**(1), 67–124.

Audemard, G. and Simon, L. (2009). Predicting learnt clauses quality in modern sat solvers. In *IJCAI*, volume 9, pages 399–404.

Bader, D. A., Hart, W. E., and Phillips, C. A. (2005). Parallel algorithm design for branch and bound. In *Tutorials on Emerging Methodologies and Applications in Operations Research*, pages 5–1. Springer.

Bailey, E., Unnikrishnan, A., and Lin, D.-Y. (2011). Models for minimizing backhaul costs through freight collaboration. *Transportation Research Record*, **2224**(1), 51–60.

Barcos, L., Rodríguez, V., Álvarez, M. J., and Robusté, F. (2010). Routing design for less-than-truckload motor carriers using ant colony optimization. *Transportation Research Part E: Logistics and Transportation Review*, **46**(3), 367–383.

Barth, P. (1995). A davis-putnam based enumeration algorithm for linear pseudo-boolean optimization.

Bender, M. A., Farach-Colton, M., Pemmasani, G., Skiena, S., and Sumazin, P.

(2005). Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, **57**(2), 75–94.

Bénichou, M., Gauthier, J.-M., Girodet, P., Hentges, G., Ribière, G., and Vincent, O. (1971). Experiments in mixed-integer linear programming. *Mathematical Programming*, **1**(1), 76–94.

Berman, P., Bhattacharyya, A., Makarychev, K., Raskhodnikova, S., and Yaroslavtsev, G. (2013). Approximation algorithms for spanner problems and directed steiner forest. *Information and Computation*, **222**, 93–107.

Bernstein, F., Song, J.-S., and Zheng, X. (2008). Bricks-and-mortar vs. clicks-and-mortar: An equilibrium analysis. *European Journal of Operational Research*, **187**(3), 671–690.

Bosagh Zadeh, R., Meng, X., Ulanov, A., Yavuz, B., Pu, L., Venkataraman, S., Sparks, E., Staple, A., and Zaharia, M. (2016). Matrix computations and optimization in apache spark. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 31–38.

Brady, B. and Catanzaro, B. (2008). Pseudo-boolean heuristics for 0-1 integer linear programming. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 132–135.

Brown, F. M. (2012). *Boolean reasoning: the logic of Boolean equations*. Springer Science & Business Media.

Budiu, M., Delling, D., and Werneck, R. F. (2011). Dryadopt: Branch-and-bound on

distributed data-parallel execution engines. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1278–1289. IEEE.

Buro, M. and Büning, H. K. (1992). *Report on a SAT competition*. Fachbereich Math.-Informatik, Univ. Gesamthochschule.

Bussieck, M. R., Ferris, M. C., and Meeraus, A. (2009). Grid-enabled optimization with gams. *INFORMS Journal on Computing*, **21**(3), 349–362.

Cao, Y. and Sun, D. (2016). Large-scale and big optimization based on hadoop. In *Big Data Optimization: Recent Developments and Challenges*, pages 375–389. Springer.

Chalkiadakis, G., Elkind, E., and Wooldridge, M. (2011). Computational aspects of cooperative game theory. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, **5**(6), 1–168.

Charikar, M., Chekuri, C., Cheung, T.-y., Dai, Z., Goel, A., Guha, S., and Li, M. (1999). Approximation algorithms for directed steiner problems. *Journal of Algorithms*, **33**(1), 73–91.

Chekuri, C., Even, G., Gupta, A., and Segev, D. (2011). Set connectivity problems in undirected graphs and the directed steiner network problem. *ACM Transactions on Algorithms (TALG)*, **7**(2), 18.

Chen, Q. and Ferris, M. C. (2001). Fatcop: A fault tolerant condor-pvm mixed integer programming solver. *SIAM Journal on Optimization*, **11**(4), 1019–1036.

CPLEX (2007). 11.0 users manual. *ILOG SA, Gentilly, France*, page 32.

CPLEX, I. (2005). High-performance software for mathematical programming and optimization.

Crainic, T. G. *et al.* (1998). *A survey of optimization models for long-haul freight transportation*. Centre for Research on Transportation.

Crainic, T. G., Le Cun, B., and Roucairol, C. (2006). Parallel branch-and-bound algorithms. *Parallel combinatorial optimization*, **1**, 1–28.

Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, **51**(1), 107–113.

Delen, D. and Ram, S. (2018). Research challenges and opportunities in business analytics. *Journal of Business Analytics*, **1**(1), 2–12.

Devriendt, J., Gleixner, A., and Nordstrom, J. (2020). Learn to relax: Integrating 0-1 integer linear programming with pseudo-boolean conflict-driven search.

Dulebenets, M. A. (2018). A comprehensive multi-objective optimization model for the vessel scheduling problem in liner shipping. *International Journal of Production Economics*, **196**, 293–318.

Dulebenets, M. A. (2019). A delayed start parallel evolutionary algorithm for just-in-time truck scheduling at a cross-docking facility. *International Journal of Production Economics*, **212**, 236–258.

Dulebenets, M. A. and Ozguven, E. E. (2017). Vessel scheduling in liner shipping: Modeling transport of perishable assets. *International Journal of Production Economics*, **184**, 141–156.

Eckstein, J., Hart, W. E., and Phillips, C. A. (2015). Pebbl: an object-oriented framework for scalable parallel branch and bound. *Mathematical Programming Computation*, **7**(4), 429–469.

Eén, N. and Sorensson, N. (2006). Translating pseudo-boolean constraints into sat. *Journal on Satisfiability, Boolean Modeling and Computation*, **2**, 1–26.

Elffers, J. and Nordstrom, J. (2018). Divide and conquer: towards faster pseudo-boolean solving. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pages 1291–1299.

Emrouznejad, A. (2016). *Big Data Optimization: Recent Developments and Challenges*, volume 18. Springer.

Erera, A., Hewitt, M., Savelsbergh, M., and Zhang, Y. (2013a). Improved load plan design through integer programming based local search. *Transportation Science*, **47**(3), 412–427.

Erera, A. L., Hewitt, M., Savelsbergh, M. W., and Zhang, Y. (2013b). Creating schedules and computing operating costs for ltl load plans. *Computers & Operations Research*, **40**(3), 691–702.

Feldman, M., Kortsarz, G., and Nutov, Z. (2012). Improved approximation algorithms for directed steiner forest. *Journal of Computer and System Sciences*, **78**(1), 279–292.

Forrest, J., Hirst, J., and Tomlin, J. A. (1974). Practical solution of large mixed integer programming problems with umpire. *Management Science*, **20**(5), 736–773.

Freeman, J. W. (1995). *Improvements to propositional satisfiability search algorithms*. Ph.D. thesis, Citeseer.

Fu, Z., Marhajan, Y., and Malik, S. (2004). zChaff SAT solver.

Galea, F. and Le Cun, B. (2007). Bob++: a framework for exact combinatorial optimization methods on parallel machines. In *International Conference High Performance Computing & Simulation*, pages 779–785.

Glankwamdee, W. and Linderoth, J. (2006). Lookahead branching for mixed integer programming. Technical report, Technical Report 06T-004, Lehigh University.

Gleixner, A., Hendel, G., Gamrath, G., Achterberg, T., Bastubbe, M., Berthold, T., Christophel, P. M., Jarck, K., Koch, T., Linderoth, J., *et al.* (2019). MIPLIB 2017: Data-driven compilation of the 6th mixed-integer programming library. *Optimization online*.

Gomes, C. P., Kautz, H., Sabharwal, A., and Selman, B. (2008). Satisfiability solvers. *Foundations of Artificial Intelligence*, **3**, 89–134.

Gurobi Optimization, L. (2018). Gurobi optimizer reference manual.

Haass, R., Dittmer, P., Veigt, M., and Lütjen, M. (2015). Reducing food losses and carbon emission by using autonomous control–a simulation study of the intelligent container. *International Journal of Production Economics*, **164**, 400–408.

Harel, D. and Tarjan, R. E. (1984). Fast algorithms for finding nearest common ancestors. *siam Journal on Computing*, **13**(2), 338–355.

Hejazi, B. and Haghani, A. (2007). Dynamic decision making for less-than-truckload trucking operations. *Transportation Research Record*, **2032**(1), 17–25.

Hernández, S., Peeta, S., and Kalafatas, G. (2011). A less-than-truckload carrier collaboration planning problem under dynamic capacities. *Transportation Research Part E: Logistics and Transportation Review*, **47**(6), 933–946.

Heule, M. and van Maaren, H. (2009). Look-ahead based SAT solvers. *Handbook of Satisfiability*, **185**, 155–184.

Heule, M. J., Kullmann, O., Wieringa, S., and Biere, A. (2011). Cube and conquer: Guiding cdcl sat solvers by lookaheads. In *Haifa Verification Conference*, pages 50–65. Springer.

Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. (2007). Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review*, volume 41, pages 59–72. ACM.

Jarrah, A. I., Johnson, E., and Neubert, L. C. (2009). Large-scale, less-than-truckload service network design. *Operations Research*, **57**(3), 609–625.

Jeroslow, R. G. and Wang, J. (1990). Solving propositional satisfiability problems. *Annals of mathematics and Artificial Intelligence*, **1**(1-4), 167–187.

Juan, A. A., Faulin, J., Pérez-Bernabeu, E., and Jozefowiez, N. (2014). Horizontal cooperation in vehicle routing problems with backhauling and environmental criteria. *Procedia-Social and Behavioral Sciences*, **111**, 1133–1141.

Katayama, N. and Yurimoto, S. (2016). The load planning problem for less-than-truckload motor: Carriers and a solution approach. In *Developments in Logistics and Supply Chain Management*, pages 240–249. Springer.

Kennington, J. L. and Nicholson, C. D. (2010). The uncapacitated time-space fixed-charge network flow problem: an empirical investigation of procedures for arc capacity assignment. *INFORMS Journal on Computing*, **22**(2), 326–337.

Klabjan, D., Johnson, E. L., Nemhauser, G. L., Gelman, E., and Ramaswamy, S. (2001). Solving large airline crew scheduling problems: Random pairing generation and strong branching. *Computational Optimization and Applications*, **20**(1), 73–91.

Koch, T., Achterberg, T., Andersen, E., Bastert, O., Berthold, T., Bixby, R. E., Danna, E., Gamrath, G., Gleixner, A. M., Heinz, S., *et al.* (2011). MIPLIB 2010. *Mathematical Programming Computation*, **3**(2), 103.

Larson, K. S. and Sandholm, T. W. (2000). Anytime coalition structure generation: an average case study. *Journal of Experimental & Theoretical Artificial Intelligence*, **12**(1), 23–42.

Lawler, E. L. and Wood, D. E. (1966). Branch-and-bound methods: A survey. *Operations research*, **14**(4), 699–719.

Leung, J. M., Magnanti, T. L., and Singhal, V. (1990). Routing in point-to-point delivery systems: formulations and solution heuristics. *Transportation science*, **24**(4), 245–260.

Li, C. M. and Anbulagan, A. (1997). Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the 15th international joint conference on Artifical intelligence-Volume 1*, pages 366–371. Morgan Kaufmann Publishers Inc.

Linderoth, J. T. and Savelsbergh, M. W. (1999). A computational study of search strategies for mixed integer programming. *INFORMS Journal on Computing*, **11**(2), 173–187.

Lindsey, K., Erera, A., and Savelsbergh, M. (2016). Improved integer programming-based neighborhood search for less-than-truckload load plan design. *Transportation Science*, **50**(4), 1360–1379.

Malapert, A., Régin, J.-C., and Rezgui, M. (2016). Embarrassingly parallel search in constraint programming. *Journal of Artificial Intelligence Research*, **57**, 421–464.

Margolis, J. T., Sullivan, K. M., Mason, S. J., and Magagnotti, M. (2018). A multi-objective optimization model for designing resilient supply chain networks. *International Journal of Production Economics*, **204**, 174–185.

Marques-Silva, J. (1999). The impact of branching heuristics in propositional satisfiability algorithms. In *Portuguese Conference on Artificial Intelligence*, pages 62–74. Springer.

McAfee, A., Brynjolfsson, E., Davenport, T. H., Patil, D., and Barton, D. (2012). Big data: the management revolution. *Harvard business review*, **90**(10), 60–68.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. (2001). Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM.

Nieuwenhuis, R. (2015). SAT-Based techniques for integer linear constraints. In *GCAI*, pages 1–13.

Oliveira, C. A. and Pardalos, P. M. (2005). A survey of combinatorial optimization problems in multicast routing. *Computers & Operations Research*, **32**(8), 1953–1981.

Özener, O. Ö. (2019). Solving the integrated shipment routing problem of a less-than-truckload carrier. *Discrete Applied Mathematics*, **252**, 37–50.

Özkaya, E., Keskinocak, P., Joseph, V. R., and Weight, R. (2010). Estimating and benchmarking less-than-truckload market rates. *Transportation Research Part E: Logistics and Transportation Review*, **46**(5), 667–682.

Patel, J. and Chinneck, J. W. (2007). Active-constraint variable ordering for faster feasibility of mixed integer linear programs. *Mathematical Programming*, **110**(3), 445–474.

Powell, W. B. (1986). A local improvement heuristic for the design of less-than-truckload motor carrier networks. *Transportation Science*, **20**(4), 246–257.

Powell, W. B. and Koskosidis, I. A. (1992). Shipment routing algorithms with tree constraints. *Transportation Science*, **26**(3), 230–245.

Powell, W. B. and Sheffi, Y. (1983). The load planning problem of motor carriers: Problem description and a proposed solution approach. *Transportation Research Part A: General*, **17**(6), 471–480.

Powell, W. B. and Sheffi, Y. (1989). Design and implementation of an interactive

optimization system for network design in the motor carrier industry. *Operations Research*, **37**(1).

Rahwan, T., Michalak, T. P., Wooldridge, M., and Jennings, N. R. (2015). Coalition structure generation: A survey. *Artificial Intelligence*, **229**, 139–174.

Ralphs, T., Shinano, Y., Berthold, T., and Koch, T. (2018). Parallel solvers for mixed integer linear optimization. In *Handbook of Parallel Constraint Reasoning*, pages 283–336. Springer.

Ralphs, T. K., Ladányi, L., and Saltzman, M. J. (2003). Parallel branch, cut, and price for large-scale discrete optimization. *Mathematical Programming*, **98**(1-3), 253–280.

Refalo, P. (2004). Impact-based search strategies for constraint programming. In *International Conference on Principles and Practice of Constraint Programming*, pages 557–571. Springer.

Roussel, O. and Manquinho, V. M. (2009). Pseudo-boolean and cardinality constraints. *Handbook of satisfiability*, **185**, 695–733.

Saggi, M. K. and Jain, S. (2018). A survey towards an integration of big data analytics to big insights for value-creation. *Information Processing & Management*, **54**(5), 758–790.

Sagratella, S. (2016). Convergent parallel algorithms for big data optimization problems. In *Big Data Optimization: Recent Developments and Challenges*, pages 461–474. Springer.

Sandholm, T. and Shields, R. (2006). Nogood learning for mixed integer programming. In *Workshop on Hybrid Methods and Branching Rules in Combinatorial Optimization, Montréal*, volume 20, pages 21–22.

Sandholm, T., Larson, K., Andersson, M., Shehory, O., and Tohmé, F. (1999). Coalition structure generation with worst case guarantees. *Artificial Intelligence*, **111**(1), 209–238.

Sheini, H. M. and Sakallah, K. A. (2005). Pueblo: A modern pseudo-boolean SAT solver. In *Proceedings of the conference on Design, Automation and Test in Europe-Volume 2*, pages 684–685. IEEE Computer Society.

Shinano, Y. (2018). The ubiquity generator framework: 7 years of progress in parallelizing branch-and-bound. In *Operations Research Proceedings 2017*, pages 143–149. Springer.

Shinano, Y. and Fujie, T. (2007). ParaLEX: A parallel extension for the CPLEX mixed integer optimizer. In *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*, pages 97–106. Springer.

Shinano, Y., Fujie, T., and Kounoike, Y. (2003). Effectiveness of parallelizing the ilog-cplex mixed integer optimizer in the pubb2 framework. In *European Conference on Parallel Processing*, pages 451–460. Springer.

Shinano, Y., Achterberg, T., and Fujie, T. (2008). A dynamic load balancing mechanism for new ParaLEX. In *2008 14th IEEE International Conference on Parallel and Distributed Systems*, pages 455–462. IEEE.

Shinano, Y., Achterberg, T., Berthold, T., Heinz, S., and Koch, T. (2011). ParaSCIP: a parallel extension of SCIP. In *Competence in High Performance Computing 2010*, pages 135–148. Springer.

Shinano, Y., Berthold, T., and Heinz, S. (2016a). A first implementation of ParaXpress: Combining internal and external parallelization to solve mips on supercomputers. In *International Congress on Mathematical Software*, pages 308–316. Springer.

Shinano, Y., Achterberg, T., Berthold, T., Heinz, S., Koch, T., and Winkler, M. (2016b). Solving open MIP instances with ParaSCIP on supercomputers using up to 80,000 cores. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 770–779. IEEE.

Warners, J. P. (1998). A linear-time transformation of linear inequalities into conjunctive normal form. *Information Processing Letters*, **68**(2), 63–69.

Wolfman, S. A. and Weld, D. S. (1999). The LPSAT engine & its application to resource planning. In *IJCAI*, volume 1999, pages 310–317. Citeseer.

Xu, Y., Ralphs, T. K., Ladányi, L., and Saltzman, M. J. (2005). Alps: A framework for implementing parallel tree search algorithms. In *The next wave in computing, optimization, and decision technologies*, pages 319–334. Springer.

Xu, Y., Ralphs, T. K., Ladányi, L., and Saltzman, M. J. (2009). Computational experience with a software framework for parallel integer programming. *INFORMS Journal on Computing*, **21**(3), 383–397.

Zabih, R. and McAllester, D. A. (1988). A rearrangement search strategy for determining propositional satisfiability. In *AAAI*, volume 88, pages 155–160.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., Stoica, I., *et al.* (2010). Spark: Cluster computing with working sets. *HotCloud*, **10**(10-10), 95.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pages 15–28.

Zhang, H. (1997). Sato: An efficient prepositional prover. In *International Conference on Automated Deduction*, pages 272–275. Springer.

Zhang, L. and Malik, S. (2002). The quest for efficient boolean satisfiability solvers. In *International Conference on Computer Aided Verification*, pages 17–36. Springer.