© 2014 by David Robert Morrison. All rights reserved.

NEW METHODS FOR BRANCH-AND-BOUND ALGORITHMS

BY

DAVID ROBERT MORRISON

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Doctoral Committee:

Professor Sheldon H. Jacobson, Chair
Professor Edward C. Sewell, Southern Illinois University, Edwardsville
Assistant Professor P. Brighten Godfrey
Professor David A. Forsyth

Abstract

Branch-and-bound (B&B) algorithms, and extensions such as branch-and-price (B&P) are powerful tools for optimization. These algorithms are used in a wide variety of settings, and thus it is beneficial to develop new techniques to improve the performance of B&B algorithms that are independent of the specific problem being studied. This dissertation describes three such techniques.

First, new results for the **cyclic best-first search** (CBFS) strategy are presented. This strategy groups subproblems into a list of **contours** which it repeatedly cycles through. The strategy selects one subproblem to explore from each contour on every pass through the list. Theoretical results are proven showing the generality of the CBFS strategy, and bounds are given on the number of subproblems the strategy explores. Moreover, an analysis of various contour definitions is performed to ascertain the factors that drive its performance.

In addition, two general-purpose methods are described for B&P algorithms that enable standard integer branching rules to be used while limiting the computation time required to solve the constrained pricing problem (i.e., the pricing problem which respects the branching decisions at the current subproblem). The first method uses a data structure called a **zero-suppressed binary decision diagram** (ZDD) to solve the pricing problem and keep track of previous branching decisions. Bounds are proved on the size of a ZDD for the maximum-weight maximal independent set problem, which is used to solve the pricing problem in a B&P algorithm for the graph coloring problem.

The last method described in this dissertation restructures the search tree in a B&P setting using a wide branching strategy so as to minimize the number of times the constrained pricing problem must be solved. This restructuring is motivated by the Wide Branching Theorem, which guarantees the existence of a smallest search tree for a fixed set of pruning rules. A **delayed branching** technique is described that limits the branching factor of the search tree, and **forgetful branching** is applied to further reduce the number of times the constrained pricing problem needs to be solved in the tree.

Computational results are presented for all methods on various optimization problems (mixed integer programming, graph coloring, the generalized assignment problem, and the simple assembly line balancing problem). Finally, future research directions are presented.

For Karen, with love

Acknowledgments

The trouble with writing acknowledgments is twofold: there's never enough space to acknowledge everyone who ought to be acknowledged, and they're rarely very interesting to read unless you're one of the people being acknowledged. Nevertheless, the work of obtaining my doctoral degree would literally[1] be impossible without the endless support of many people, and so I shall endeavor to acknowledge as many of you as possible while at the same time providing some enjoyment for those of you who didn't make the cut. If your name isn't here, and you think it ought to be, the next time I see you I'll give you a pen and let you scribble it in the margins.

I have to start with my family: to my wife Karen, thanks for everything—for rejoicing with me when things were good, for encouraging me when they weren't, and for telling me to put on my big boy britches when I was being needlessly melodramatic. To Mom, thanks for setting my foundation. I made it this far because of what you taught me, even if I fought you tooth and nail sometimes. To Daddy, thanks for teaching me to never give up and how to think for myself; these are basically required skills for doing research of any kind. To my father-in-law Bob, thank you for your encouragement, and for sparking my interest in OR; I miss you. And to Barbara, thank you for your support and and love, and for lots of entertaining board games. For the rest of my family: thank you for teaching me how to have fun, for teaching me how to laugh (and for laughing at all of my hilarious jokes), and for teaching me how to love. But most of all, thanks for all the margaritas—I never would have made it this far without those.

I also want to express my gratitude to all of the teachers and mentors who have guided me on my educational journey, but two people stand above the rest. Firstly is my advisor, Dr. Sheldon Jacobson. Thank you for providing guidance when I needed it, but also for giving me the space to make mistakes and learn from them. Thank you also for helping to mature my writing abilities. Your feedback on papers is always beneficial and insightful. I look forward to our future relationship, both professionally and personally. Secondly is my undergraduate advisor, Dr. Susan Martonosi (I even spelled it right!): your constant insistence that I should pursue my PhD is the only reason I even considered this path, but it was definitely the right

[1] I use the literal meaning of the word

path to take. Thank you for your continuing support and mentorship, and for all those bleary-eyed meetings that were too early in the morning for either of us. Let's not do those again.

To the rest of my doctoral committee, Dr. Edward Sewell, Dr. Brighten Godfrey, and Dr. David Forsyth, thank you for your time and support. Without your many good suggestions and advice, the research presented in this dissertation would be woefully incomplete.

To my academic siblings: Golsheed Baharian, Banafsheh Behzad, Arash Khatibi, Doug King, Estelle Kone, Alex Nikolaev, Laura McLay, J.D. Robbins, and Jason Sauppe, thank you for your long conversations about research, your long conversations about life, and all our group dinners. I look forward to many future collaborative endeavors. When I start my new journal entitled *Facts*, you can all be in the first issue.

Finally, I'd like to briefly mention the friends who have helped me along this journey. It is quite a long list: Nathan and Audrey, Jason and Colleen, Ben and Jen, Trevis and Jacinda, Scott and Lindsay, Matt and Anna, Mike and Lynn, Ben and Moriah, Peter and Kathryn, Mike and Meredith, Joe and Mel, Jacob and Sarah—and there are many others, as well. There's some space in the margins for you to add your name if I forgot it. I love you all!

I'd like to offer a special thank you to a place that's near and dear to my heart: the Midwest. When I first moved there, I hated it, but in the last five years, it has grown on me quite a bit. Now, if only it had less snow in the winters, less humidity in the summers, and more mountains and oceans all year round, I think I would like living there just fine. So long, and thanks for all the fish!

Gloria Patri, et Filio, et Spiritui Sancto.
Sicut erat in principio,
et nunc, et semper,
et in sæcula sæculorum.
Amen.

Kyrie eleison.

The computational results reported were obtained using the Simulation and Optimization Laboratory at UIUC. This research has been supported in part by the Air Force Office of Scientific Research (FA9550-10-1-0387), the Department of Defense (DoD) through the National Defense Science & Engineering Graduate Fellowship (NDSEG) Program, and the National Science Foundation Graduate Research Fellowship program under Grant Number DGE-1144245. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the United States Government.

Table of Contents

List of Notation . . . . **vii**

Chapter 1 Introduction **1**

Chapter 2 The Branch-and-Bound Algorithm **4**

Chapter 3 Cyclic Best-First Search **30**

Chapter 4 Solving the Pricing Problem using ZDDs **56**

Chapter 5 A Wide Branching Strategy for Branch-and-Price **79**

Chapter 6 Conclusion **95**

Appendix A Data Tables **98**

References **107**

List of Notation

a_i	An element of the constraint matrix A
A	Constraint matrix for an integer programming problem
$\mathscr{A}$	An arbitrary search strategy for a branch-and-bound algorithm
b	Constraint bounds for an integer programming problem
β	A numerical constant
c	A vector of cost coefficients
C	A combinatorial set representing a column (or variable) in a column generation scheme
$\bar{C}$	A subset of a column (or variable) in a column generation scheme
$\mathscr{C}$ ($\mathscr{C}'$)	The (restricted) set of columns in a column generation scheme
d	The maximum depth of a search tree T
$d(\cdot)$	The degree function for vertices in a graph G
D	A directed graph
D^+	The transitive closure of a directed graph D
$\delta^-(\cdot)$	The indegree function (number of incoming arcs to a vertex in a directed graph D)
Δ_j	A bound on the size of the subtree rooted at the jth child of the current subproblem
$\Delta(G)$	The degree of the largest-degree vertex in an undirected graph
E	The set of edges for a graph
$\mathscr{E}$	An ordered set of objects or elements $(e_1, e_2, \ldots, e_n)$
f	The objective function for an optimization problem
F_i	The ith Fibonacci number
$\phi(y_1, y_2, \ldots, y_n)$	A boolean formula on the variables $y_1, y_2, \ldots, y_n$
φ	$(1+\sqrt{5})/2$
Φ_j^* (Φ_j)	The set of (direct) successors of a task j
G	An undirected graph with vertex set V and edge set E
$G[U]$	The induced subgraph of G with respect to a set of vertices U

$\Gamma_i(\bar{C})$	The set $\bar{C} \cap \{v_1, v_2, \ldots, v_{i-1}\}$ for an independent set $\bar{C}$ in a graph G
i, j, k	Generic indices
I_m	The idle time for a machine m in a scheduling problem
$\mathscr{I}$	A set of indices of non-empty contours for a CBFS strategy
J	A set of jobs or tasks in a scheduling or assignment problem
K_i	The ith contour in a CBFS strategy
$\mathscr{K}$	The set of contours for a CBFS strategy
$\mathrm{supp}(\mathscr{K})$	The support of the set of contours $\mathscr{K}$, i.e., the set of non-empty contours in $\mathscr{K}$ over the course of the algorithm.
$\kappa(\cdot)$	The contour labeling function for the CBFS strategy
$\ell_i(Z_{\mathscr{C}})$	The set of nodes of $Z_{\mathscr{C}}$ which satisfy $\mathrm{var}(z) = i$; the ith level of $Z_{\mathscr{C}}$
L_z $(\overline{L}_z)$	The (reduced) eligibility set for a node z in the maximal independent set ZDD; that is, the set of vertices available for use (used by at least one valid path) at z
λ	Lagrange multiplier vector in an integer programming problem
M	A bound on the length of time needed to explore a subproblem
$\mu(\cdot)$	The measure-of-best function for BFS and CBFS
n, m	The number of elements or variables in a problem or set
$[n]$	The set $\{1, 2, \ldots, n\}$
$N(U)$	The set of vertices in a graph G adjacent to vertices in U
$N[U]$	The closed neighbor set of U in a graph G, that is, $N(U) \cup U$
P	A path in a ZDD or in a graph
$\mathscr{P}$	An optimization problem defined over a search space X with objective function f
$(\pi_1, \pi_2, \ldots, \pi_n)$	A set of weights (or prices) in a column generation scheme
$\pi(C)$	$\sum_{i \in C} \pi_i$; if C is a singleton, sometimes the set notation is dropped
Π_j^* (Π_j)	The set of (direct) predecessors of a task j
q	A vector of auxiliary variables for a mixed integer programming problem
r	The branching factor of the search tree T
R	A list of restricted elements, or a tabu list
S	A subset of the search space X; a subproblem in T
$\mathscr{S}$	A set of unexplored subproblems in a branch-and-bound algorithm
σ_i	The set of jobs assigned to the ith worker in an assembly line problem
t	A vector of job or processing times for a scheduling/assignment problem
T	The search tree built up by a branch-and-bound algorithm

τ_z	The totally dominated set of vertices at a node z in the maximal independent set ZDD; that is, the set of vertices dominated by all valid paths from z
u, v	Vertices in an undirected graph
$u \leftrightarrow v$ ($u \not\leftrightarrow v$)	Vertices u and v are (non-)adjacent.
U	A subset of vertices in a graph
V	The set of vertices for an undirected graph
$w_i(Z_{\mathscr{C}})$	The width of the ith level of $Z_{\mathscr{C}}$
x	An element of the search space X; a feasible solution to an optimization problem
$\hat{x}$ ($\hat{x}'$)	The current (candidate) incumbent solution for a branch-and-bound algorithm
x^*	An optimal solution to an optimization problem
X	A set of feasible solutions to an optimization problem; the search space
ξ	The capacity or cycle time for a machine in an assembly line or assignment problem
y	A vector of variables for a mixed integer programming problem
z	A node in a ZDD Z
z_{root}	The root node of a ZDD Z
$z \vdash \bar{C}$	Denotes that a z-to-$\mathbf{1}$ path in a ZDD $Z_{\mathscr{C}}$ corresponds to a subset $\bar{C}$ of some column in $\mathscr{C}$; i.e., z yields $\bar{C}$.
$Z, Z_{\mathscr{C}}$	A zero-suppressed binary decision diagram; a ZDD characterizing the family of sets $\mathscr{C}$
χ	The chromatic number of a graph G
$\lvert Z_{\mathscr{C}} \rvert$	The total number of nodes and edges in the ZDD $Z_{\mathscr{C}}$
$1_C(\cdot)$	The indicator or characteristic function for a set C
$\mathbf{1}, \mathbf{0}$	The “true” and “false” (or “accept” and “reject”) nodes of a ZDD
2^Y	The power set of a set Y

Chapter 1

Introduction

The **branch-and-bound** (B&B) framework is a fundamental and widely-used methodology for producing exact solutions to NP-hard optimization problems. The technique, which was first proposed by Land and Doig (1960), is often referred to as an *algorithm*; however, it is perhaps more appropriate to say that B&B encapsulates a family of algorithms that all share a common core solution procedure. This procedure implicitly enumerates all possible solutions to the problem under consideration, by storing partial solutions called **subproblems** in a tree structure. Unexplored nodes in the tree generate children by partitioning the solution space into smaller regions that can be solved recursively (i.e., **branching**), and rules are used to prune off regions of the search space that are provably suboptimal (i.e., **bounding**). Once the entire tree has been explored, the best solution found in the search is returned. A early overview of the core B&B algorithm was provided by Lawler and Wood (1966); the solution procedure is also covered in the excellent texts by Nemhauser and Wolsey (1988), Bertsimas and Tsitsiklis (1997), and Papadimitriou and Steiglitz (1998).

One reason for the popularity of B&B methods is due to the generality of the solution procedure: B&B-based algorithms can be used to solve many different types of optimization problems in areas of practical interest, including scheduling problems (e.g., airline crew scheduling (Barnhart et al., 2003), sports scheduling (Easton et al., 2003), hospital staff scheduling (Beliën and Demeulemeester, 2008; Gendreau et al., 2007)), graph problems (e.g., coloring problems (Mehrotra and Trick, 1996), partitioning problems (Clausen, 1999), clustering problems (Fukunaga and Narendra, 1975)), network design and network flow (e.g., facility location (Görtz and Klose, 2012), vehicle routing (Fukasawa et al., 2006), multicommodity flow (Barnhart et al., 2000)), and many others. B&B algorithms are also applied to problems of theoretical importance, including mixed integer programming (Nemhauser and Wolsey, 1988) and non-linear programming (Tawarmalani and Sahinidis, 2004).

An additional reason for the success of B&B-based approaches is due to the simplicity of the method. B&B is easy to explain and implement; moreover, it is easily extensible. This fact allows lessons learned in one problem domain to be easily transferred to other settings. Its simplicity also ensures that mistakes in the algorithm or implementation are less common and easier to detect.

Many problem-specific enhancements to B&B methods have been developed that have resulted in dramatic improvements in computation time and solution quality. While problem-specific algorithmic enhancements are in many cases critical to the success of a B&B algorithm, new general-purpose enhancements to the framework can yield similar performance gains in many settings at once. Therefore, this dissertation proposes three new general-purpose enhancements for B&B algorithms that can be applied to a wide class of problems while maintaining the simplicity of the approach.

Chapter 3 describes the first of these extensions, called **cyclic best-first search** (CBFS). This extension is a new search strategy that can be used with B&B which attempts to balance the diversification and intensification properties of the search process more effectively than other standard search strategies such as depth-first search (DFS) and best-first search (BFS). The eponymous characteristic of the CBFS strategy is its cycling behavior: CBFS groups subproblems together into sets called **contours**, and repeatedly cycles through these contours, selecting one subproblem from each contour to explore on each pass through the search tree. By constructing the contours appropriately, the search strategy can delay exploration of some subproblems and encourage exploration of others, similarly to the way a tabu list operates in local search algorithms. This strategy provides algorithm designers with greater control over how subproblems are explored, and can lead to significant improvements in performance in some settings.

The remaining two extensions described in this dissertation are applied in the context of **branch-and-price** (B&P); B&P is an extension of B&B which operates in a setting where the number of branching variables is exponential in the input size of the problem. The entire set of branching variables cannot in general be stored in available memory; thus, only a subset of the branching variables is stored at any given time. B&P algorithms combine a conventional B&B search over this restricted pool of variables together with an auxiliary problem referred to as the **pricing problem**, which is used to introduce new variables into the pool as necessary.

Problem formulations with an exponential number of decision variables are often used to improve the value of the lower bounds used to prune in the search process, as well as to break symmetry that may be present in simpler formulations. However, the pricing problem (which needs to be solved exactly at least once at every subproblem in the search tree) is usually NP-hard as well. Thus B&P algorithms need to balance the computational savings gained by better pruning with the increased computation time needed to repeatedly solve the pricing problem.

A secondary challenge that arises when solving an exponentially-sized formulation with B&P is the necessity of the pricing problem to respect previous branching decisions. In particular, when the pricing problem is solved to introduce new variables into the variable pool, the produced variables must not violate

any of the branching decisions at the current subproblem. However, communicating these branching decisions to the pricing problem generally significantly increases the difficulty of finding a solution, which in turn increases the overall computation time. The pricing problem with additional side constraints reflecting the branching decisions at the current subproblem is called the **constrained pricing problem**.

A final complication due to the large number of variables present in B&P settings is that the resulting search trees are often extremely unbalanced. If the algorithm gets unlucky, it may spend much computation time exploring the deeper regions of the search tree before finding optimal or near-optimal solutions, when instead it could have found an optimal solution much more quickly if it had explored the shallower side of the tree first. For example, given a binary mixed integer programming problem with a large number of covering constraints (of the form $\sum a_i y_i \geq b$), a single branching decision setting $y_i = 1$ may satisfy a large number of constraints simultaneously, whereas a branching decision setting $y_i = 0$ may not change the structure of the IP very much. Thus, a comparatively small number of the former type of assignments will be performed before either a feasible solution is found or the path is pruned. On the other hand, a large number of the latter type of assignments will need to be made before the corresponding path can be pruned.

Chapter 4 presents an extension to B&P algorithms that attempts to address these issues. This extension uses a data structure called a **zero-suppressed binary decision diagram** (ZDD) to encode solutions to the pricing problem; this data structure requires some additional up-front computation time, but then can be queried and updated efficiently to solve the constrained pricing problem as the search progresses. The use of this data structure enables bounds to be computed more quickly in a B&P context, and thus pruning to occur more quickly. This extension also uses the CBFS strategy to counteract the effects of an unbalanced search tree.

A second B&P extension, called **wide branching**, is presented in Chapter 5. Instead of trying to improve the solution times of the pricing problem, this extension instead modifies the the branching strategy used by the algorithm so as to limit the number of times the constrained pricing problem must be solved, using an operation called **path compression**. The path compression operation collapses long chains in the search tree into a single subproblem, and **forgetful branching** is applied to remove branching constraints that conflict with the pricing problem. A **delayed branching** technique is applied to limit the branching factor of the search tree. This new branching strategy has the added benefit of restructuring the search tree so that it is not as unbalanced.

Finally, in Chapter 6, future research directions for these methods are discussed.

Chapter 2

The Branch-and-Bound Algorithm

At the most general level, a branch-and-bound algorithm is used to solve some optimization problem $\mathscr{P} = (X, f)$, where X (called the **search space**) is a finite set of valid solutions to the problem, and $f : X \to \mathbb{R}$ is the **objective function**. The algorithm's goal is to find an **optimal** solution $x^* \in \arg\min_{x \in X} f(x)$. To solve $\mathscr{P}$, B&B iteratively builds a **search tree** T of **subproblems**, that is, subsets of the search space. Additionally, a feasible solution $\hat{x} \in X$ (called the **incumbent solution**) is maintained. At each iteration, the algorithm selects a new subproblem to explore from a list $\mathscr{S}$ of **unexplored** subproblems; if a solution $\hat{x}' \in S$ (called a **candidate incumbent**) can be found with a better objective value than $\hat{x}$ (i.e., $f(\hat{x}') < f(\hat{x})$), the incumbent solution is updated. On the other hand, if it can be proven that no solution in S has a better objective value than $\hat{x}$ (i.e., $\forall\, x \in S, f(x) \geq f(\hat{x})$), the subproblem is **pruned** (or **fathomed**), and the subproblem is **terminal**. Otherwise, child subproblems are generated by partitioning S into an exhaustive (but not necessarily mutually exclusive) set of subproblems $S_1, S_2, \ldots, S_r$, which are then inserted into T. Once no unexplored subproblems remain, the best incumbent solution is returned; since subproblems are only fathomed if they contain no solution better than $\hat{x}$, the solution returned by the algorithm must be $x^* \equiv \hat{x}$. Pseudocode for the generic B&B procedure is given in Algorithm 2.1.

Algorithm 2.1: Branch-and-Bound(X, f)

1. Set $\mathscr{S} = \{X\}$ and initialize $\hat{x}$
2. **while** $\mathscr{S} \neq \varnothing$**:**
3. Select a subproblem $S \in \mathscr{S}$ to explore
4. **if** a solution $\hat{x}' \in \{x \in S \mid f(x) < f(\hat{x})\}$ can be found**:** Set $\hat{x} = \hat{x}'$
5. **if** S cannot be pruned**:**
6. Partition S into $S_1, S_2, \ldots, S_r$
7. Insert $S_1, S_2, \ldots, S_r$ into $\mathscr{S}$
8. Remove S from $\mathscr{S}$
9. Return $\hat{x}$

With respect to this pseudocode, the search strategy affects the order in which nodes are selected for exploration in Line 3 of Algorithm 2.1; the branching strategy affects the way the subproblem is partitioned and the number of produced children (Line 6, Algorithm 2.1); and the pruning rules used in Line 5 of

Algorithm 2.1 determine whether or not S can be fathomed. Usually, an initial incumbent solution is found (Line 1, Algorithm 2.1) via a heuristic procedure (see, for example, Malaguti et al., 2011).

Note that the set X (and all subproblems) is generally given implicitly; that is, given an element x, membership in a particular subproblem can be checked efficiently, and a partition of any subproblem can be computed efficiently without knowing all the members of X. Then, the complexity of B&B algorithms is related to two factors: the **branching factor** r of the tree, which is the maximum number of children generated at any node in the tree, and the **depth** d of the tree, which is the longest path from the root of T to a leaf. Thus, any B&B algorithm operates in $O(Mr^d)$ worst-case running time, where M is a bound on the length of time needed to explore a subproblem; however, the presence of pruning rules can substantially improve the algorithm performance.

2.1 Relationships Between Algorithm Components

There are two important phases of any B&B algorithm: the first is the **search** phase, in which the algorithm has not yet found an optimal solution x^*. The second is the **verification** phase, in which the incumbent solution is optimal, but there are still unexplored subproblems in the tree that cannot be pruned. Note that an incumbent solution cannot be proven optimal until no unexplored subproblems remain; also note that the delineation between the search phase and the verification phase is unknown until the algorithm terminates. In a slight abuse of terminology, a problem $\mathscr{P}$ is said to be **solved** if the B&B algorithm has completed the verification phase. In this case, the algorithm is said to have produced a **certificate of optimality**.

The three algorithmic components (search strategy, branching strategy, and pruning rules) each play a distinct role in B&B algorithms with respect to these two phases of operation (see Figure 2.1). In particular, the choice of search strategy primarily impacts the search phase. To see this, suppose the pruning rules and branching strategy are fixed, and only depend on the value of the incumbent solution (e.g., they compare a subproblem's lower bound to the incumbent value). In this setting, any search strategy must explore the same remaining set of subproblems once an optimal solution is found.

Moreover, observe that the choice of pruning rules primarily impacts the verification phase, since pruning rules are often relatively weak before an optimal (or near-optimal) solution is known (using the above example, if the incumbent solution has a poor objective value early in the search process the lower bounds will not be able to prune effectively, even if they are very tight). However, the choice of branching strategy has significant impacts on both the search phase and the verification phase: by branching appropriately at subproblems, the strategy can guide the algorithm towards optimal solutions, and limiting the branching

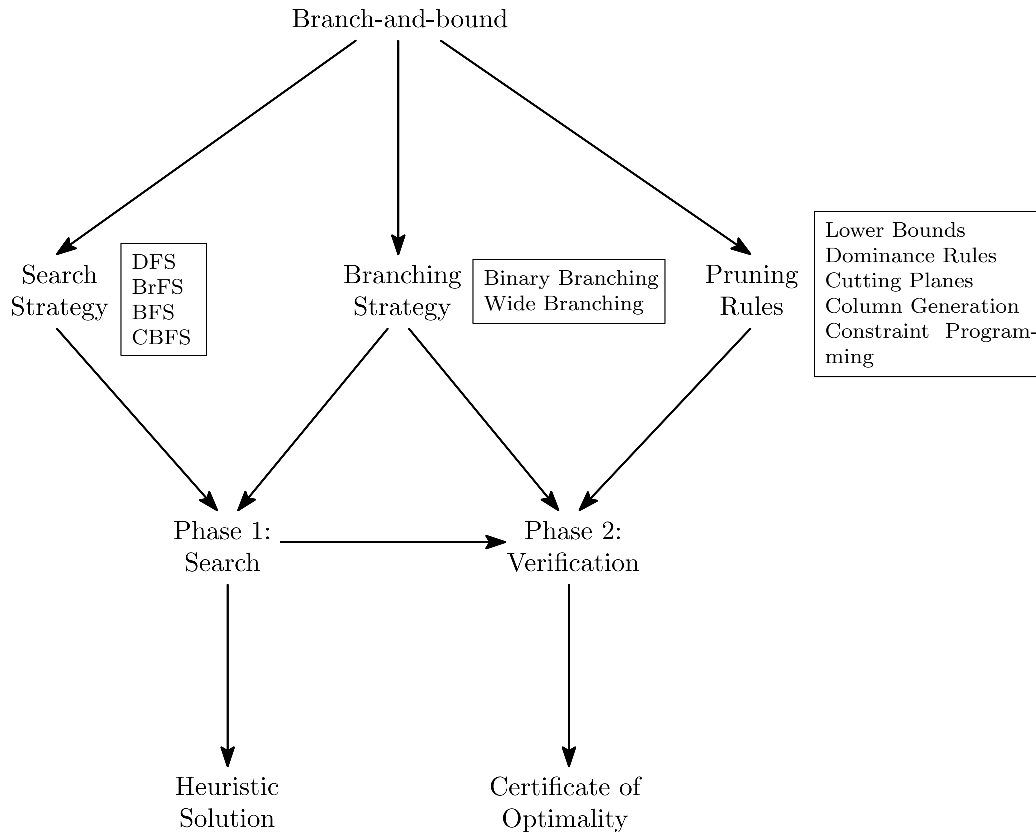


Figure 2.1: A diagram of the three main B&B components. The search strategy and the pruning rules primarily impact the search phase and verification phase, respectively, whereas the branching strategy impacts both.

decisions made during verification prevents unnecessary work from being performed to produce a certificate of optimality.

There are two important reasons to improve performance of the B&B algorithm during the search phase. First, if the algorithm terminates before producing a certificate of optimality, the incumbent solution can still be returned as a heuristic solution, which may be sufficient in some problems. An example of this behavior can be seen in Fischetti and Lodi (2003), which introduces new constraints in a mixed integer programming framework to attempt to reach a feasible solution quickly.

Secondly, finding an optimal solution earlier in the search phase has a direct impact on the size of the search tree (and thus the time necessary to verify optimality), since no further nodes with bounds greater than the optimum value need be explored. Intuitively, this is the rationale behind the result of Dechter and Pearl (1985) showing that best-first search explores the fewest number of subproblems of any search strategy.

Figure 2.2 shows the internal relationships between the different types of pruning rules, branching strategies, and search strategies. In this figure, solid lines indicate a generalization relationship. For instance, as

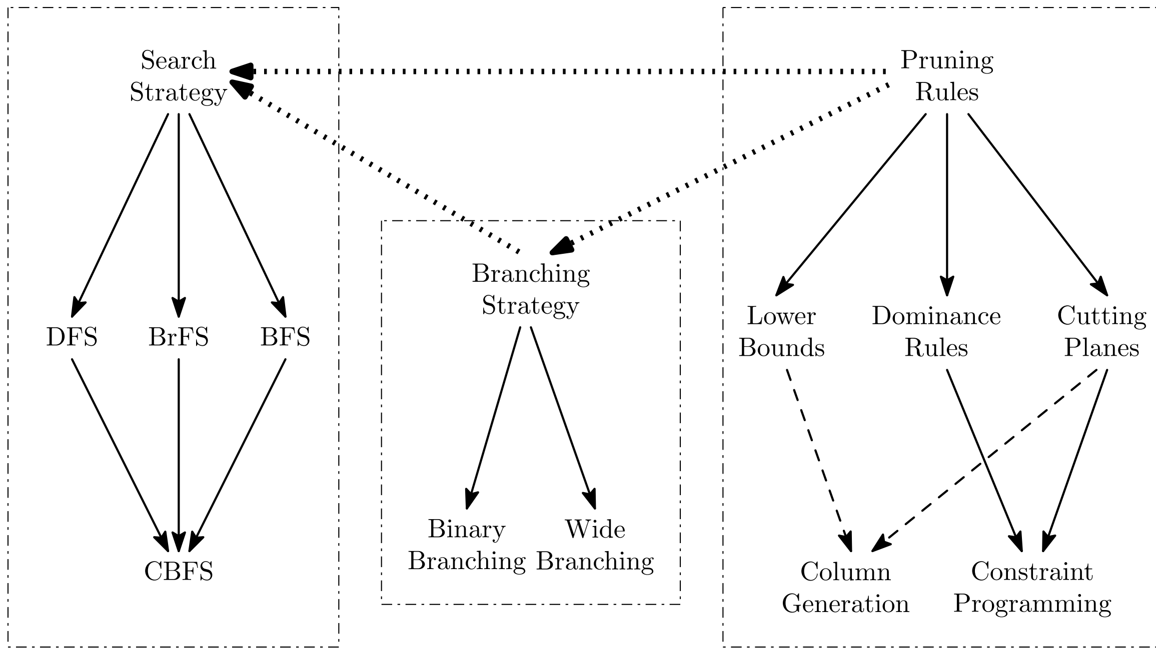


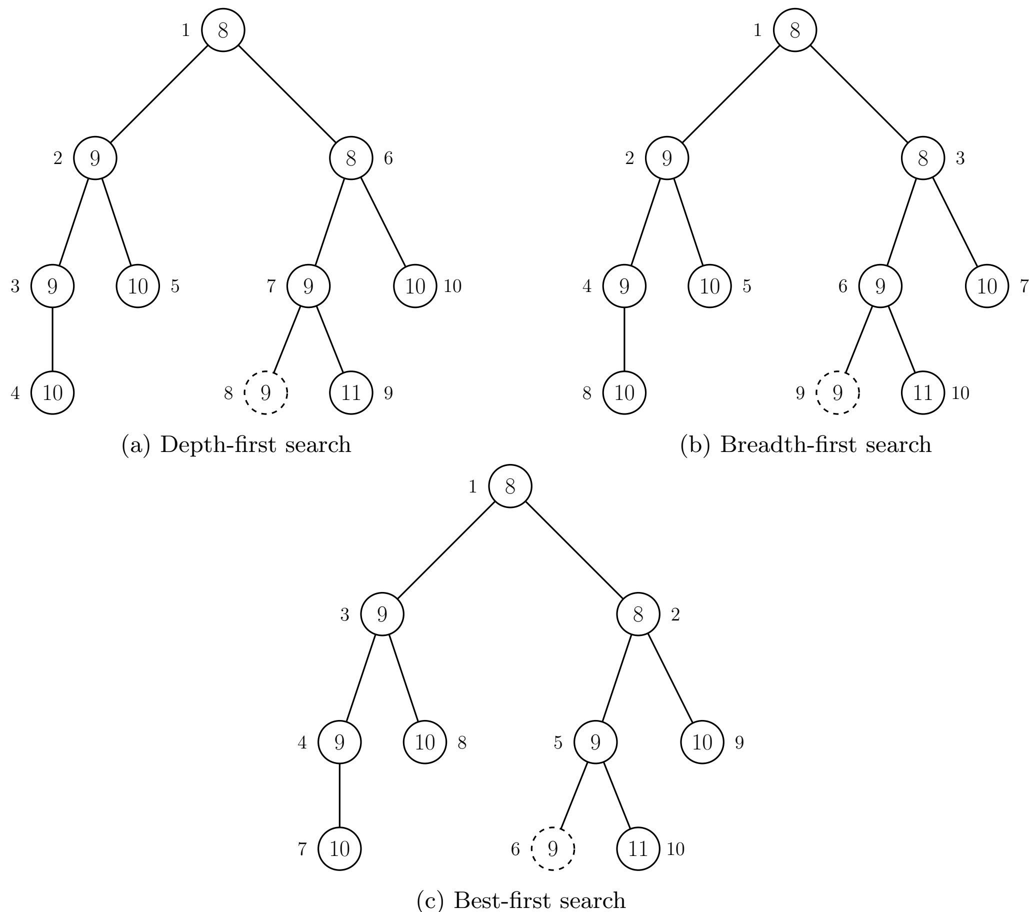
Figure 2.2: A diagram of relationships between various B&B algorithm components.

discussed in Section 2.4.4, many constraint programming techniques generalize cutting planes and dominance relations. Furthermore, the CBFS search strategy is a generalization of DFS, BrFS, and BFS (see Chapter 3). Column generation techniques, while not strictly a generalization of other techniques, are closely connected to lower bounding and cutting plane techniques (in essence, column generation adds cutting planes to the dual optimization problem to improve the computed lower bound); B&P algorithms combine a B&B search with column generation (see Section 2.5).

Figure 2.2 also shows the relationships between the pruning rules, the branching strategy, and the search strategy used by an algorithm. In particular, the choice of pruning rules often impacts or limits the choices that can be made in the other two areas. For example, as discussed in Section 2.5, if column generation is used to improve lower bounds, the choice of branching strategies that can be used is limited. Moreover, if dominance relations are used, this may cause BrFS to become a desirable search strategy, since it has the property of never exploring a dominated subproblem. Finally, the choice of branching strategy can itself impact the choice of search strategy. For instance, if the branching strategy chosen produces a particularly unbalanced tree, the CBFS strategy can balance the search process, or variants of DFS can limit the depth explored at any stage in the algorithm.

2.2 Search Strategies

The search strategy in a B&B algorithm determines the order in which unexplored subproblems in T are selected for exploration. The choice of search strategy has potentially significant consequences for the amount of computation time required for the B&B procedure, as well as the amount of memory used. In some cases, for very large or challenging problems, it may be necessary to choose a search strategy that requires low memory usage; however, for problems in which memory is not a concern, other search strategies exist which may find an optimal solution very quickly, and thus explore potentially fewer subproblems. A comparison of some search strategies is given in Ibaraki (1976). In this section, a discussion of common search strategies, along with their strengths and weaknesses is given. Figure 2.3 shows a small search tree, and the order in which nodes are explored for several different search strategies.



(a) Depth-first search

(b) Breadth-first search

(c) Best-first search

Figure 2.3: Subproblem exploration order for different search strategies. The dashed subproblem is optimal, numbers inside nodes are subproblem lower bounds, and numbers outside the nodes indicate exploration order. The algorithm starts with an incumbent solution of value 10. BFS uses the lower bound as the measure-of-best, with ties broken arbitrarily.

2.2.1 Depth-First Search

The **depth-first search** (DFS) strategy (sometimes called depth-first search with backtracking, or last-in, first-out search; see Nemhauser and Wolsey, 1988, Section 11.4) is a search strategy used in many different graph algorithms in addition to B&B (Golomb and Baumert, 1965; Tarjan, 1972). It can be implemented by maintaining the list of unexplored subproblems $\mathscr{S}$ as a stack. The algorithm removes the top item from the stack to choose the next subproblem to explore, and when children are generated as a result of branching, they are inserted on the top of $\mathscr{S}$. Thus, the next subproblem that is explored is the most recently generated subproblem.

However, a slight modification to this algorithm can be made to produce a substantial savings in memory usage. In particular, if the children of a subproblem can be ordered in some way, then DFS does not need to store the entire list of unexplored subproblems (which can grow quite large) over the course of the algorithm. Instead, the search strategy only stores the path from the root of T to the current subproblem; at each subproblem along this path, it also stores the index of the last-explored child subproblem. At the current subproblem, the next unexplored child is selected for exploration. If no unexplored children remain, the algorithm **backtracks** to the closest ancestor node with unexplored children.

In addition to its low memory requirements, another advantage of DFS arises when solving integer programming problems (Section 2.6.1) that use the LP relaxations as lower bounds. Since most branching decisions in this setting do not change the structure of the LP relaxation significantly, the LP solver can often reuse information from the parent LP solution as a starting point for the child LP solution. This procedure is called **warm starting**, and is used in many commercial LP solvers (Atamtürk and Savelsbergh, 2005).

Two problems arise with the use of the DFS strategy. The first problem is that naïve implementations of DFS do not use any information about problem structure or bounds, which means the search process can spend large amounts of exploration time in poor regions of the search space. A related phenomenon, called **thrashing**, occurs when different regions of the search space all fail for the same or similar reasons (Kumar, 1992). For instance, perhaps the presence of a single branching constraint always leads to infeasibility, but the algorithm must explore many more subproblems before the infeasibility is detected.

A different problem arises when the search tree is extremely unbalanced. In other words, if some optimal solutions are close to the root, but there exist long paths in T that do not lead to an optimal solution, DFS can (unluckily) choose many long, bad paths before it explores a path leading to an optimal solution. However, this computation time often could be avoided via pruning rules if the search strategy instead chose to explore a short optimal path first. In fact, this behavior of DFS was first noticed on problems where the search tree had unbounded depth (Slate and Atkin, 1983), but the same problem exists in trees with a few

extremely long paths.

A host of variants to the depth-first search strategy exist that attempt to overcome these limitations. One common variant is the **iterative deepening** DFS algorithm (Korf, 1985), which imposes a limit on the depth of subproblems explored by DFS; if the search process is not able to prove optimality using this depth limit, the depth is increased and the search is restarted from the root. This ensures that the search does not spend unneeded time exploring extremely long paths in the search tree while retaining the low memory overhead of regular DFS.

Another algorithm called **interleaved depth-first search** by Meseguer (1997) tries to overcome thrashing behavior by performing depth-first search from multiple locations in the search tree at once. This strategy can be performed by sequentially selecting exactly one subproblem to explore from each different DFS path in the search tree before returning to the first search path. This algorithm can improve performance over standard DFS with a relatively limited increase in memory usage (a single stack needs to be maintained for each search path).

A third variant of DFS is **depth-first search with complete branching**, which tries to exploit problem structure by selecting the next child subproblem to explore as the one with the best computed lower bound (Scholl and Klein, 1999). This method explores the search tree more intelligently, at the expense of increased memory usage, since all child subproblems must be generated when a subproblem is explored. However, if the tree has a relatively small branching factor, this increased memory usage is not likely to be significant.

2.2.2 Breadth-First Search

Breadth-first search (BrFS) is the opposite of DFS in that it is implemented with a first-in, first-out, or queue, data structure. BrFS explores all subproblems that are a fixed distance from the root before exploring any deeper subproblems. The BrFS strategy has the advantage of always finding an optimal solution that is closest to the root of the tree, thus operating well on unbalanced search trees. However, since complete solutions are usually at larger depths, BrFS is generally unable to exploit pruning rules that compare against the current incumbent solution. For this reason, the memory requirements for BrFS are often quite high, and it is generally not used in a B&B context. Two exceptions are in the presence of dominance relations (Section 2.4.2), which can prune effectively even in the absence of a good incumbent solution, and if a good incumbent solution can be found effectively by some other means, for example with a good heuristic search (Sewell and Jacobson, 2012). It is also worth noting that in the absence of pruning rules, the iterative deepening DFS strategy explores the same sequence of nodes as BrFS with substantially lower memory requirements.

2.2.3 Best-First Search

In settings where sufficient memory is available to store the entire unexplored search tree, the **best-first search** (BFS) strategy is often used. This strategy makes use of a heuristic **measure-of-best** function $\mu : 2^X \to \mathbb{R}$, which computes a value $\mu(S)$ for every unexplored subproblem, and selects as the next subproblem to explore the one minimizing μ. If $\mu(S) \leq \min_{x \in S} f(x)$ for all S (that is, the measure-of-best function never overestimates the best solution in a subproblem), the measure-of-best function is **admissible**. In the presence of an admissible μ, BFS is also called the A* algorithm (Dechter and Pearl, 1985), or sometimes **best-bound** search. BFS can easily be implemented by storing the list of subproblems in a heap data structure, using the value of μ as the key (Cormen et al., 2009).

There are many choices for the measure-of-best function; one common choice is a lower bound on the value of the best solution in the subproblem. If the lower bound is strongly correlated with the subproblem objective values, this measure-of-best will encourage exploration of subproblems with better solutions. However, in practice, lower bounds may not be a good proxy for the objective function value. For example, in integer programming problems which use the LP relaxation as a lower bound, a small lower bound may just indicate that the structure of the problem allows the LP to "cheat" in ways that the IP cannot. To overcome this, other candidate measure-of-best functions are heuristics which estimate the quality of a solution, such as in Sewell and Jacobson (2012). Additionally, Shi and Ólafsson (2000) use a probabilistic function called the **promising index** to estimate the quality of a solution, and commercial solvers such as CPLEX use a heuristic function to estimate the objective value of a particular subproblem (IBM Corp., 2014).

Best-first search offers a number of significant advantages over DFS; because it is not tied to exploring one specific branch of the tree before any other, it is often able to find good solutions earlier in the search process. In fact, this notion has been formalized in a theorem by Dechter and Pearl (1985) that states that, assuming an admissible μ with no ties and no dominance relations, BFS explores the fewest number of subproblems of any search strategy that has access to the same heuristic function and other information.

However, as observed in Sewell and Jacobson (2012), BFS does still have one potential drawback—if there exist many subproblems in T for which $\mu(S) = f(x^*)$, depending on the tie-breaking rule used, BFS may spend much time in middle regions of the search tree and never explore an optimal solution (see Figure 2.3c, in which nodes $3, 4$, and 5 are explored before the optimal solution is found). In this situation, BFS may be slower than some other strategy. To overcome this, many BFS implementations employ **diving** heuristics or other heuristic methods to drive a subproblem towards a new incumbent solution that can aid in pruning (Bixby et al., 2000; Achterberg et al., 2008).

2.3 Branching Strategies

The choice of branching strategy determines how children are generated from a subproblem. Branching strategies can be categorized into two groups: **binary** branching strategies and non-binary, or **wide**, branching strategies. Additionally, due to the prevalence of integer programming problems, there is a plethora of literature devoted to branching strategies in integer programming. These will be discussed separately at the end of the section.

2.3.1 Binary Branching

Binary branching strategies focus on subdividing a subproblem S into two mutually-exclusive, smaller subproblems. For example, in the knapsack problem, which seeks a maximum-cost selection of items to fit inside a storage bin with fixed capacity, a binary branching strategy for a subproblem S selects some unassigned item and creates two branches, one in which the item is included in the knapsack, and one in which the item is excluded from the knapsack (Kolesar, 1967). Most binary branching strategies are variants of this idea. The standard integer branching scheme for integer programming in Section 2.6.1 is another binary branching scheme.

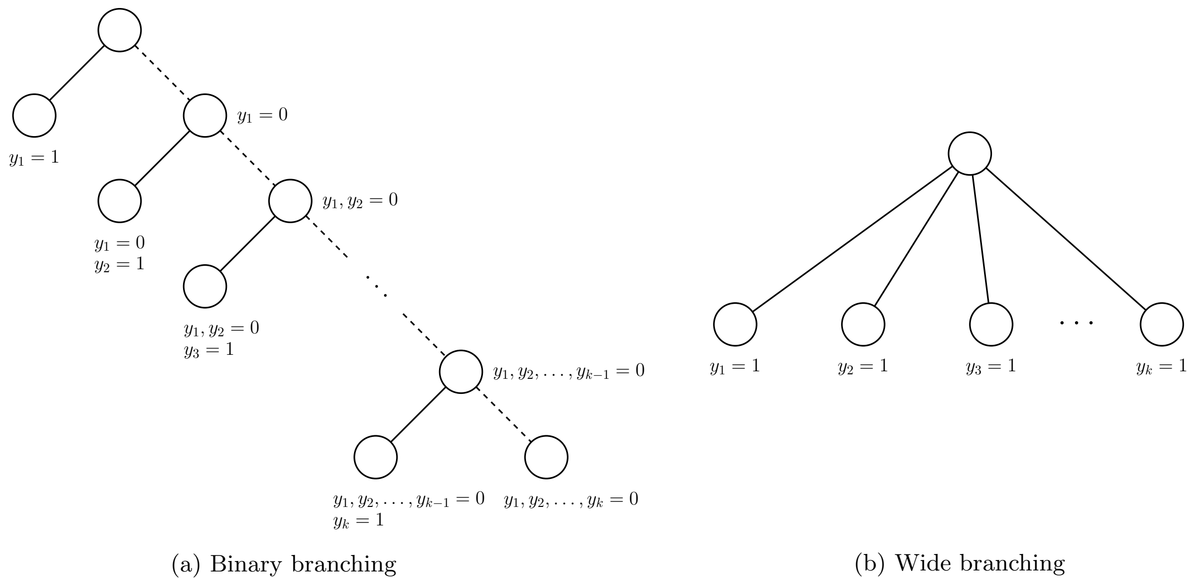
In some cases, the mechanism for performing the partitioning is more complicated. For the graph coloring problem (Section 2.6.2), Mehrotra and Trick (1996) use a branching rule that either adds edges or contracts vertices of G in order to force a pair of non-adjacent vertices to either share same color or use different colors. Similarly, in the branch-and-price solver for the generalized assignment problem (Section 2.6.3, (Savelsbergh, 1997)), branching is performed by either including or excluding all schedules that assign a particular task to a worker.

2.3.2 Wide Branching

In contrast to binary branching are wide branching strategies, which focus on selecting one element from a set of different options. For example, in B&B algorithms to compute maximum cliques or independent sets in a graph, a set of unused vertices is maintained for each subproblem, and each unused vertex generates a child with that vertex added to the child's set (Babel, 1994; Held et al., 2012). Wide branching methods allow for potentially large reductions in the size of the search tree. In the above example, a binary branching strategy would have to consider each unused vertex individually, creating a long sequence of subproblems. This long sequence can be bypassed with the wide branching technique (see Figure 2.4).

One important setting in which a wide branching strategy has been used successfully is with **special**

ordered sets (SOS), introduced by Beale and Tomlin (1970) and Beale and Forrest (1976). There are two types of special ordered sets, denoted by SOS1 and SOS2. An SOS1 is a set of elements for which at most one element can be used in a solution, and an SOS2 is a set of elements for which at most two adjacent elements can be used in a solution. SOS2 are useful for modeling piecewise linear approximations of nonlinear optimization problems (de Farias, Jr. et al., 2000; D'Ambrosio and Lodi, 2011). Wide branching strategies can be used in B&B to handle problems with SOS variables; for example, when an SOS1 is selected for branching, the strategy creates one branch for each element in the set. The subproblem for this set uses the chosen element and excludes all others. Finally, the branching strategy creates one subproblem which uses no elements from the set. This strategy can be generalized to handle SOS2, as well.



(a) Binary branching (b) Wide branching

Figure 2.4: Binary branching versus wide branching. Given a set of k elements from which one must be selected, binary branching must explicitly reject elements $1, 2, \ldots, j-1$ before creating a branch that considers element j. Conversely, wide branching can consider each of them immediately.

Two potential problems arise with wide branching strategies. The first is that such strategies usually do not create mutually-exclusive branches, so it is possible to arrive at the same subproblem from several different paths. It is generally easy to work around this problem using a lexicographic ordering rule (Geoffrion, 1969), memory-based dominance rules (Sewell et al., 2012; Sewell and Jacobson, 2012), or nogood recording (See Section 2.4 for more details on nogoods and dominance).

A second problem arises if the number of branches that can be created at a particular subproblem is very large. In this case, the algorithm could get stuck generating children at a particular subproblem and never move on to explore new regions of the search space. Moreover, if the number of generated children is very large at every subproblem, the size of the search tree will grow much more rapidly. There are two potential

ways around this issue; the first sets an arbitrary cap on the number of children that can be generated at a subproblem. If the branching factor ever exceeds this limit, any additional children are just discarded. This technique, used in the simple assembly line balancing solver of Sewell and Jacobson (2012), can prove optimality if the branching factor is never exceeded; otherwise, it performs as a heuristic. The other method uses a **delayed branching** technique, described in Section 5.1.2, in which the algorithm delays generation of the remaining children in the hopes that when it returns to the node, better bounds may have been computed that allow it to prune children more effectively.

Finally, a hybrid approach between binary branching and wide branching, called **orbital branching**, creates a large number of children when a subproblem is explored, but uses a group-theoretic concept of an **orbit** to prune off all but two of them. This technique is most useful for problems which demonstrate a high degree of symmetry (Ostrowski et al., 2011).

2.3.3 Branching in Integer Programs

Given that many optimization problems can be modeled using integer programming (Section 2.6.1), substantial effort has been devoted to branching strategies for integer programming problems. The branching strategy is generally divided into two phases: selecting a variable or set of variables to branch on, and creating child subproblems by imposing bounds on these variables to force them away from fractional values. The choice of branching variable can significantly impact the performance of the algorithm, and many different techniques exist to choose good branching variables.

One commonly-used, easy-to-implement rule is called the **most fractional** rule, which selects the variable y_i whose fractional part is closest to 0.5 as the branching variable. However, as shown by Achterberg et al. (2005), this branching rule is no better than selecting a branching variable at random in terms of computational time required and number of subproblems explored. Therefore, a number of more advanced techniques have been proposed in the literature to improve the performance of B&B integer programming solvers. The opposite branching rule, the **least fractional** rule selects y_i such that its fractional part is furthest from 0.5; this rule is less commonly used and is often outperformed by other methods (see, for example, Ortega and Wolsey (2003)).

Another approach is to branch on variables that induce the most change in the objective function (Linderoth and Savelsbergh, 1999). There are two methods of doing this; the first, called **strong branching**, computes the LP relaxation objective value of the children of a subproblem S for each candidate branching variable, and then selects the variable that induces the most change in the objective. However, this is computationally expensive, so an alternate method called **pseudocost branching** (Benichou et al., 1971)

is often used which attempts to predict the per-unit change of the objective function for each candidate branching variable, based on past experience in the tree.

One problem that arises with pseudocost branching, however, is how to initialize the pseudocosts at the beginning of the algorithm, since no information is available about past behavior. To address this, a method called **hybrid strong/pseudocost branching** can be used, which employs strong branching at upper levels of the search tree to initialize the pseudocosts, and then uses pseudocosts in lower regions of the tree once more information is available. An alternate method, called **reliability branching** proposed by Achterberg et al. (2005), uses strong branching for any variables whose pseudocosts have been deemed unreliable—that is, for which there is not enough historical information in the branching process to compute pseudocosts for the variable.

Recent research by Pryor and Chinneck (2011) has explored the use of branching rules that try to find feasible integer solutions to the problem quickly. They achieve this by branching on variables that induce change in the largest number of *variables* in the problem (as opposed to the largest change in objective value, as with pseudocost branching). The somewhat surprising result in their paper shows that branching on variables with the smallest probability of satisfying constraints in the LP often leads to integer feasibility more quickly, because this will require a large number of other variables in the LP to change to satisfy the constraints.

Another recent branching method developed by Fischetti and Monaci (2011) is called **backdoor branching**. This technique solves an auxiliary integer program to determine a small set of variables that should be branched on before any others; this auxiliary program is a set covering problem which computes a **backdoor**—that is, a set of variables which, if branched upon early in the search process, yield a small search tree.

Finally, Gilpin and Sandholm (2011) use information-theoretic results to guide the search process by branching so as to remove **uncertainty** from subproblems in the search tree. Subproblems close to the root in the search tree have a large amount of uncertainty, since few variables have been fixed; terminal subproblems have no uncertainty, since all variables have assumed integer values. To do this, they treat the values of fractional variables as probabilities, and compute the **entropy** (i.e., the amount of uncertainty) for each candidate branching variable, selecting the one with the least entropy to branch upon. A related technique by Karzan et al. (2009) uses machine learning techniques to train a B&B algorithm to choose branches that lead to small search trees.

2.4 Pruning and dominance rules

A critical aspect of B&B search is the choice of pruning rules used to exclude regions of the search space from exploration. Note that for a fixed branching strategy, any node that cannot be pruned by the pruning rules must be explored by *any* search strategy, even if an optimal solution is known before the search begins. The only way to reduce the size of the search tree in this case is to use better pruning rules. There are many different classes of pruning rules, but they are usually problem-specific and must be derived anew for each different problem type under consideration. Again because of its prevalence, many pruning rules are focused on integer programming problems.

2.4.1 Lower Bounds

The most common way to prune is to produce a lower bound on the objective function value at each subproblem, and use this to prune subproblems whose lower bound is no better than the incumbent's solution value. Lower bounds are computed by relaxing various aspects of the problem. For example, in the simple assembly line balancing problem (Section 2.6.4) and its variants, one common relaxation is to compute the optimal solution value ignoring the precedence constraints (Vilà and Pereira, 2014; Sewell and Jacobson, 2012). In general, as many different lower bounds can be computed as necessary; some lower bound computations may be easy to compute, whereas others may be more computationally intensive. Thus a common practice is to attempt to prune using the easy lower bounds first, and then move on to the more complex, but tighter, lower bounds if the easy methods are unsuccessful.

If the problem can be formulated as an integer program, the optimal value of the LP relaxation is an extremely common lower bound choice. The quality of the LP relaxation value is measured by the **integrality gap** of the formulation, that is, the ratio between the best integer solution and the best LP relaxation value across all problem instances. However, there may be many different ways to formulate the problem using integer programming, and some of these problems may have tighter integrality gaps than others. Thus, one technique for improving lower bounds is to derive a new formulation with a tighter integrality gap (Arora et al., 2002). The branch-and-cut and branch-and-price algorithms described in Sections 2.4.3 and 2.5 are common methods for exploiting integer programming formulations with tighter bounds. A related approach for polynomial programming problems called the **reformulation-linearization technique** (RLT) transforms a mathematical program with polynomial objective function and constraints into a linear program, and uses the resulting LP bound to prune in B&B algorithm to find global optimal solutions to the polynomial program (Sherali and Tuncbilek, 1992).

Another method for deriving lower bounds on integer programming problems is through duality. Though

there is no strong duality theorem for integer programming, one can still arrive at a notion of weak duality. Given an integer program $\min\{f(y) \mid Ay \leq b, y \in \mathbb{Z}\}$, the **Lagrangian relaxation** problem is $\mathscr{P}(\lambda) = \min\{f(y) + \lambda(b - Ay) \mid y \in \mathbb{Z}\}$, where λ is a non-positive vector of real-valued weights called **Lagrange multipliers**. The optimal solution value for the Lagrangian relaxation is always bounded above by the value of the optimal solution to the original problem. Thus, the best bound possible may be computed as the solution to the **Lagrangian dual** problem, $\max_{\lambda \leq 0} \mathscr{P}(\lambda)$. The Lagrangian dual problem can be solved using **subgradient optimization**, a modification of Newton's method for piecewise linear concave functions (Bertsimas and Tsitsiklis, 1997). Integer programming duality methods have been used in Vilà and Pereira (2014); Desrosiers et al. (2013); Gendron et al. (2013), and Phan (2012), among others.

2.4.2 Dominance Relations

In contrast to lower bounding rules, dominance relations allow subproblems to be pruned if they can be shown to be **dominated** by some other subproblem—in other words, if subproblem S_1 dominates subproblem S_2, this means that for any solution that is contained S_2, there exists a complete solution in S_1 that is at least as good. Thus, it suffices to just explore S_1. Dominance relations, first studied by Kohler and Steiglitz (1974), are closely related to the Bellman equations from dynamic programming (Bellman, 1954). Note that, as shown by Ibaraki (1977), it is not always true that using dominance relations will improve the quality of the search process; however, there are many cases in which dominance relations will improve the search.

There are two primary types of dominance relations, memory-based and non-memory-based. Memory-based dominance rules compare unexplored subproblems to other problems previously generated and stored in the tree (Sewell et al., 2012). As the name implies, memory-based dominance rules require the entire search tree to be stored for the duration of the algorithm, instead of just the unexplored subproblems. However, this may allow for additional pruning to be performed that would be otherwise impossible.

Non-memory-based dominance relations do not require the dominating state to have been previously generated in the search process—instead, non-memory-based dominance rules are able to imply the *existence* of a dominating subproblem, regardless of whether it has been explored or generated. Such rules have the advantage that they do not require additional memory to store the generated search tree, but they may not be able to prune the same subset of problems that memory-based dominance rules can.

In B&B algorithms that employ dominance, the BrFS strategy has the useful property that it never explores a dominated subproblem, as long as the dominance relations are formulated in such a way so that subproblems are only compared if they are within the same level of T (see, for example, Nazareth et al. (1999); Sewell and Jacobson (2012)).

Note that care must be taken when implementing dominance rules to avoid mutual dominance relations. In particular, depending on the structure of the dominance rules employed, cycles of dominating subproblems $S_1, S_2, \ldots, S_k$ could exist where S_{i+1} dominates S_i, and S_1 dominates S_k (Demeulemeester et al., 2000). In such cases, at least one subproblem in the dominance cycle must not be pruned; often, this can be accomplished using some lexicographic ordering rule.

2.4.3 Cutting Planes

The discussion in this section is restricted to problems that can be formulated as integer programs. A significant advance in the theory of linear and integer programming was developed by Gomory (1958), who introduced the idea of **cutting planes**. A cutting plane is a constraint that can be added to an integer program to tighten the feasible region without removing any integer solutions. This fundamental idea was applied to B&B by Padberg and Rinaldi (1991) to develop an algorithm called **branch-and-cut**. In this algorithm, new cutting planes (sometimes called **valid inequalities**) are added to the LP relaxation at every subproblem in the search tree (note that a valid inequality is a global constraint—it must apply at the LP relaxation of the root subproblem). The algorithm of Padberg and Rinaldi (1991) is specific to the well-known traveling salesman problem, but in Balas et al. (1996a) a generalization of branch-and-cut for binary integer programs is presented.

There are a number of different types of valid inequalities; an overview is given in Cornuéjols (2008). The initial cutting planes described by Gomory are called **Gomory cuts**, and are based on the structure of the simplex tableau. These cuts were shown to be of both practical and theoretical interest by Balas et al. (1996b). Some other types of valid inequalities are **Chvátal-Gomory cuts** (Letchford and Lodi, 2002), **disjunctive cuts** (Balas, 1979), and **lift-and-project** cuts (Lovász and Schrijver, 1991; Balas et al., 1993). Of these, lift-and-project cuts provide an extremely general method to add valid inequalities, and thus are used in many different branch-and-cut algorithms (Balas and Perregaard, 2003). Lift-and-project operates by lifting the LP relaxation into a higher-dimensional space by adding additional variables, finding valid inequalities in this higher-dimensional space, and then projecting the valid inequalities back into the original space by deleting the extra variables.

Another method of generating valid inequalities is through decomposition methods such as **Benders' decomposition** (Benders, 1962; Geoffrion, 1972). A decomposition method splits apart a problem into a **master problem** and one or more **slave problems** (sometimes referred to as "subproblems", but this terminology is avoided herein to avoid confusion with the search tree subproblems). For example, in Benders' decomposition, a set of **complicating variables** in the integer program are identified which drive the

intractability of the problem. The non-complicating variables are then projected out of the integer program. The master problem therefore seeks a solution to the new integer program, and the slave problem either determines that the master problem is feasible for the original integer program or produces a constraint that it violates. An algorithm to solve such a slave problem is also known as a **separation oracle**, because it *separates* feasible solutions from infeasible ones. These **Benders' cuts** can then be added in to the master problem. Hernández-Pérez and Salazar-González (2004) give an example of using Benders' cuts in a branch-and-cut context to solve the traveling salesman problem with both pickups and deliveries.

One interesting question with regards to this method of pruning involves the interplay between cutting plane generation and branching. In many cases, the set of generated cuts is too large to allow all of them to be added, and it is often computationally expensive to generate new cuts, so at some point cutting planes are no longer generated and branching occurs. However, the question of when to stop generating cutting planes and start branching is an important problem when implementing a branch-and-cut algorithm (Jünger et al., 1995; Mitchell, 2002).

2.4.4 Constraint Programming

Recently, interest has increased in using constraint programming techniques for solving optimization problems. Constraint programming is a subfield of artificial intelligence that has been very successful in solving logic problems such as SAT. Constraint programming has many potential applications to B&B algorithms, and many other applications in optimization. For a complete discussion of the field of constraint programming and applications in AI and OR, see Rossi et al. (2006). Also, a comparison of constraint programming and operations research techniques, along with an application of constraint programming to the fixed-charge network flow problem can be found in Kim and Hooker (2002).

Two primary ideas behind many constraint programming techniques for B&B algorithms are **constraint propagation** and **nogood learning** (van Beek, 2006). Constraint propagation rules exploit the repeated application of logical inference rules in an attempt to derive contradictions that allow a subproblem to be pruned. On the other hand, a nogood is a structural property of the problem that has been proven to not lead to a feasible or optimal solution by complete exploration of some subtrees. Nogoods are generally learned over the course of the algorithm, and enable the search to check the validity of subproblems under consideration. Constraint programming techniques in B&B share many commonalities with other pruning techniques such as cutting planes and dominance relations. However, as pointed out by Caseau and Laburthe (1996), one significant difference is that constraint programming techniques are usually local techniques that tighten bounds at a particular subproblem, unlike the global pruning rules introduced by dominance or valid

inequalities.

An example of constraint propagation rules is given in Fahle (2002); in this paper, a technique called **domain filtering** is used to solve the maximum clique problem in a B&B context. Here, two results are proven showing when it is impossible for vertices in a graph to participate in a maximum clique, and these results are iteratively applied (or *propagated*) at each subproblem in T. If the propagation reduces this list of candidate vertices to the empty set, the subproblem is fathomed.

Constraint propagation techniques often have close relations to lower bounding techniques. Fahle (2002) show that their domain filtering rule subsumes seven out of eight common lower bounds for the maximum clique problem. Moreover, Li et al. (2005) use a similar constraint propagation rule to aid in the computation of lower bounds that can be used for pruning in a Max-SAT solver (the Max-SAT problem seeks an assignment that satisfies the maximum number of clauses in a Boolean formula).

Conversely, nogoods are more closely related to dominance relations and cutting planes. For instance, Sandholm and Shields (2006) learn a sequence of nogoods (in this case, invalid assignments to sets of variables) that can be added as cuts to the integer program. These nogoods are derived by constraint propagation based on the branching decisions. Additionally, Fischetti and Salvagnin (2010) use nogood recording to develop dominance-like relations for a B&B solver for generic mixed-integer programming problems. Their solver uses an auxiliary integer programming problem to identify dominated subproblems; if a dominated subproblem is identified, it is stored as a nogood so that the auxiliary integer program for that subproblem does not need to be re-solved in the future.

2.5 Branch-and-Price

While B&B is useful in a wide variety of settings, a number of additional issues must be taken into consideration when the problem is very large. In this section, an extension of B&B called branch-and-price is discussed, which is applicable when the problem under consideration is formulated as an integer program with an exponential number of decision variables. However, since B&P is just an extension of B&B, the three core components (search strategy, branching strategy, and pruning rules) are still applicable in this setting.

In a sense, B&P can be thought of as the dual algorithm to branch-and-cut (Section 2.4.3). Here, instead of adding new constraints to the (primal) master problem, a separation oracle for the dual of the master problem is used to add new constraints to the dual. This procedure is known as **column generation**, since new constraints in the dual of the master problem correspond to new variables (or constraint matrix columns) in the primal master problem. The column generation approach was first described in Dantzig

and Wolfe (1960), together with a decomposition method called **Dantzig-Wolfe decomposition**. Detailed descriptions of B&P algorithms are given in Barnhart et al. (1998) and Lübbecke and Desrosiers (2005), and pseudocode is presented in Algorithm 2.2.

The Dantzig-Wolfe decomposition method transforms an integer program into a new program where the variables correspond to the extreme points of the original IP. Since any solution to the original IP can be written as a convex combination of its extreme points, no information is lost. Moreover, in practice the reformulated program often yields much tighter bounds and less symmetry than the original. The principal drawback arises from the fact that the original IP has an exponential number of extreme points, which means that the number of variables (or columns) for the reformulated problem is too large to all be stored simultaneously. Therefore, to solve the LP relaxation and get a valid lower bound, **column generation** must be used.

In particular, if $\mathscr{C}$ is the (exponentially-sized) set of variables for the reformulated problem, a smaller problem called the **restricted master problem** (RMP) is solved over a subset $\mathscr{C}'$ of the columns. Then, one or more slave problems (or **pricing problems**) are solved to identify new variables with the potential to improve the value of the LP relaxation, which can then be added to $\mathscr{C}'$. The pricing problem is usually a weighted combinatorial optimization problem. The weights for the pricing problem are related to the optimal dual price vector π of the RMP, and the pricing problem identifies new variables (or columns) for inclusion in $\mathscr{C}'$ by searching for $C \in \mathscr{C} \setminus \mathscr{C}'$ with negative reduced cost. Note that in most cases, the pricing problem itself is NP-hard.

Algorithm 2.2: Branch-and-Price(X, f)

```
1  Set 𝒮 = {X}
2  Initialize x̂ and the initial RMP pool 𝒞′
3  while 𝒮 ≠ ∅:
4      Select a subproblem S ∈ 𝒮 to explore
5      if a solution x̂′ ∈ {x ∈ S | f(x) < f(x̂)} can be found:  Set x̂ = x̂′
6      if S cannot be pruned:
7          Partition S into S_1, S_2, ..., S_r
8          for each S_i ∈ {S_1, S_2, ..., S_r}:
9              ⟪ Column generation loop ⟫
10             while ∃ C ∈ 𝒞 \ 𝒞′ with negative reduced cost at S_i: Add C to 𝒞′
11             Compute a lower bound at S_i using added columns
12         Insert S_1, S_2, ..., S_r into 𝒮
13     Remove S from 𝒮
14 Return x̂
```

Additional complexity arises when attempting to incorporate column generation with a B&B algorithm, because typical branching rules usually interfere with the structure of the pricing problem. In other words,

once some branching decisions have been fixed, new negative-reduced-cost variables must be found that respect the branching decisions. This problem, known as the constrained pricing problem, is closely related to the kth**-shortest-path** problem. Moreover, when using standard integer branching rules (see Section 2.6.1) in B&P algorithms, there is often an asymmetry in the branching rule. For example, if the integer program contains a large number of covering constraints (of the form $\sum a_i y_i \geq b$), fixing a variable $y_i = 1$ has the potential to satisfy a large number of constraints, whereas fixing a variable $y_i = 0$ may have minimal impacts on the problem structure. This can lead to extremely unbalanced search trees, which in turn can impact the performance of the algorithm.

Therefore, to avoid interfering with the pricing problem structure and to attempt to create a more balanced search tree, most B&P algorithms use alternative branching strategies that do not disrupt the structure of the pricing problem. The branching strategy for graph coloring (Mehrotra and Trick, 1996) or for the generalized assignment problem (Savelsbergh, 1997) (see Section 2.3.2) are two such examples. Another general-purpose branching strategy for B&P algorithms involves branching on the original (non-decomposed) problem variables (Vanderbeck, 2011).

Due to the intractability of the pricing problem in many IP models, significant research has also gone into ways to solve the pricing problem. Gualandi and Malucelli (2012) use constraint programming techniques (Section 2.4.4) to more efficiently solve the pricing problem in a graph coloring B&P solver, and Easton et al. (2003) use a similar approach for a sports timetabling problem.

Finally, a new field of research is emerging in **branch-and-cut-and-price** algorithms, which use separation oracles to produce new constraints for both the primal and dual master problems. These methods suffer from many of the same problems as B&P algorithms, since now the pricing problem must respect both the branching decisions and the additional cutting planes added to the problem. However, de Aragão and Uchoa (2003) developed a new method called **robust branch-and-cut-and-price** (robust BCP) which further reformulates the master problem to eliminate the interference of cuts and branching decisions with the pricing problem. This method has been used with success in a number of vehicle routing and other graph problems (Fukasawa et al., 2006; Uchoa et al., 2008).

2.6 Problems of Interest

B&B methods are quite general, and thus the best way to gauge their performance in practice is to implement them for problems that are of practical or real-world interest. The techniques proposed in this dissertation are tested and validated against four different problems, which are described in the remainder of this section.

2.6.1 Mixed Integer Programming

The mixed integer programming problem is a very general NP-complete problem that can describe many different optimization problems using a set of algebraic constraints. Specifically, let y be a vector of variables, some of which are constrained to integer values. Also, let A be a real-valued matrix called the **constraint matrix**, and b be a real-valued vector of constraint bounds. Then the search space for an integer programming instance is defined by the system of equations $Ay \leq b$, with objective function $f(y) = c'y$, where c is a real-valued vector and $'$ denotes the transpose operator.

In this setting, bounds are commonly produced by solving the **LP relaxation** of the problem, where the integrality constraints on y are relaxed. Branching decisions are imposed by adding additional constraints to the problem to shrink the feasible region without removing any optimal integral solutions. For example, the standard integer branching rule (sometimes called $0-1$ **branching** if y is a vector of binary variables) selects a variable y_i with fractional value β in the LP relaxation and creates two new branches, one with $y_i \leq \lfloor \beta \rfloor$ (called a **null assignment**), and one with $y_i \geq \lceil \beta \rceil$ (called a **positive assignment**). If no fractional variables y_i exist, a new candidate incumbent has been found.

Many different optimization problems can be formulated as mixed integer programs, and the LP relaxation often provides tight bounds in practice. Thus, a number of B&B techniques have been developed specifically for this setting. Moreover, many very efficient software packages (both commercial and freeware) exist for solving integer programs using B&B techniques, including CPLEX (IBM Corp., 2014), SYMPHONY (Ladanyi et al., 2014), Gurobi (Gurobi Optimization, Inc., 2014), LINDO (LINDO Systems, Inc., 2014), SCIP (Konrad-Zuse-Zentrum für Informationstechnik Berlin, 2014), and Xpress-MP (Fair Isaac Corporation (FICO), 2014).

There is a well-known database of mixed integer programming problem instances called MIPLIB 2010 (Koch et al., 2011); this database contains a very large number of instances that span a range of difficulties, and which are divided into different subsets to provide a consistent set of instances that can be used for testing purposes. There are two instance subsets of interest: first is the set of benchmark problems, which contains 87 instances, all of which can be solved in under an hour of computation time by at least one commercial or open-source MIP solver. The second instance subset considered is the set of challenge problems, which contain 164 MIP problems which cannot be solved in 2 hours of computation time by any commercial or open-source MIP solver.

2.6.2 Graph Coloring

The graph coloring problem is an important problem in many areas of operations research; in addition to being of theoretical interest, it appears as a subproblem in many practical settings, including resource allocation, processor scheduling, and others (Pardalos et al., 1998). Despite this, it remains a very challenging problem to solve exactly in most practical applications, and it is NP-hard to approximate within $n^{1-\varepsilon}$ (Zuckerman, 2006).

The following notation will be useful when discussing graph coloring problems: given an undirected graph $G = (V, E)$ and vertices $u, v \in V$, u and v are **adjacent** (**non-adjacent**), denoted $u \leftrightarrow v$ ($u \not\leftrightarrow v$), if $(u, v) \in (\notin) E$. Further, given a set $U \subseteq V$, the **induced subgraph** of G with respect to U, denoted $G[U]$, is the subgraph of G defined on the vertex set U with all edges of G having both endpoints in U. The **neighbor set** of $U \subseteq V$, denoted $N(U)$, is the set of vertices adjacent to vertices in U, and the closed neighbor set $N[U]$ is $N(U) \cup U$ (if U contains a single vertex u, the set notation is dropped, e.g. $N(u)$). The degree $d(u)$ of a vertex u is $|N(u)|$, and $\Delta(G)$ is the largest degree over all vertices in G.

The objective of the graph coloring problem is to find a minimum **proper coloring** of V (i.e., a coloring in which no adjacent vertices share a color). The **chromatic number** χ of G is the minimum number of colors required in any proper coloring. Given a partial assignment of colors to vertices, the the **degree-of-saturation** function $d_{sat} : V \to \mathbb{N}_0$ counts how many differently-colored neighbors a vertex has.

There is a close relationship between the graph coloring problem and the **independent set** problem; an **independent set** $C \subseteq V$ is a set of vertices such that $G[C]$ has no edges, and C is a **maximal** independent set if there exists no vertex $v \in V \setminus C$ such that $C + v$ is an independent set. Observe that by definition, any group of vertices with a common color is an independent set, and therefore, any proper coloring of G is a partition of the vertices into independent sets. Finally, say that for any set of vertices U, a node $v \in V$ is **dominated** by U if $v \in N[U]$; note that for any maximal independent set C, $N[C] = V$.

A standard IP formulation for the graph coloring problem creates binary variables y_{vj} for each vertex $v \in V$ and each color $j \in \{1, 2, \ldots, \overline{\chi}\}$, where $\overline{\chi}$ is any upper bound on the chromatic number. Additional binary variables q_j are introduced for each $j \in \{1, 2, \ldots, \overline{\chi}\}$. In this formulation, setting $y_{vj} = 1$ assigns color j to vertex v, and setting $q_j = 1$ indicates that color j is assigned to some vertex in the graph. Using these variables, the following IP encodes the graph coloring problem:

$$
\begin{aligned}
\text{minimize } & \sum_{j=1}^{\overline{\chi}} q_j \\
\text{subject to } & \sum_{j=1}^{\overline{\chi}} y_{vj} = 1 \ \forall \ v \in V \\
& y_{uj} + y_{vj} \leq q_j \ \forall \ (u,v) \in E, j \in \{1, 2, \ldots, \overline{\chi}\} \\
& y_{vj}, q_j \in \{0, 1\} \ \forall \ v \in V, j \in \{1, 2, \ldots, \overline{\chi}\}.
\end{aligned} \tag{2.1}
$$

Here, the first set of constraints ensures that every vertex receives exactly one color, and the second set of constraints ensures that adjacent vertices are assigned different colors. However, as noted by Mehrotra and Trick (1996), this formulation has two weaknesses: first, it contains many symmetric solutions which arise by permuting the colors used in a solution. These symmetries lead to many redundant subtrees in a branch-and-bound search algorithm, which cannot be easily detected and substantially slow the search process. Secondly, the linear programming bound of (2.1) is often quite weak, which does not allow for much pruning to occur in the search tree. Therefore, Mehrotra and Trick (1996) propose the following integer program with an exponential number of variables:

$$
\begin{aligned}
\text{minimize } & \sum_{C \in \mathscr{C}} y_C \\
\text{subject to } & \sum_{C : v \in C} y_C \geq 1 \ \forall \ v \in V \\
& y_C \in \{0, 1\} \ \forall \ C \in \mathscr{C}.
\end{aligned} \tag{2.2}
$$

In this formulation, $\mathscr{C}$ is the family of maximal independent sets in G; since any proper coloring can be viewed as a partition of V into independent sets, this is equivalent to searching for the smallest coloring. The binary variables y_C indicate whether the maximal independent set C is used in the coloring, and the constraints ensure that each vertex in the graph appears in some color class. This formulation eliminates the symmetry inherent in (2.1), as well as producing much tighter bounds for many instances.

However, since there are potentially an exponential number of maximal independent sets in G, it is generally infeasible to keep all variables in memory. Thus, to solve the LP relaxation of (2.2), column generation techniques must be employed. The pricing problem for the graph coloring problem as formulated in (2.2) is a maximum-weight maximal independent set problem, where the weights on the vertices are given by the values of the optimal dual variables of the RMP. If a maximal independent set C with weight larger than 1 is found, then variable y_C has negative reduced cost, which means that y_C is a candidate to improve the solution value of the RMP and C can be added to $\mathscr{C}'$.

The formulation presented in (2.2) is used in most state-of-the-art exact solvers for the graph coloring problem. These solvers often use a branching rule called **edge branching** to avoid destruction of the pricing problem. Edge branching selects two non-adjacent, uncolored vertices in G and creates two branches, one in which the vertices are linked by an edge, and one in which they are merged together (Mehrotra and Trick, 1996). In contrast, standard $0-1$ branching (called **vertex branching** by Malaguti et al., 2011) selects one maximal independent set to either use or discard at each branching decision.

There is a substantial body of literature on the graph coloring problems that covers both exact and heuristic methods for the problem. For reference, see Galinier and Hertz (2006), who provide a comparison of twenty different heuristic methods that have been applied to graph coloring, or Malaguti and Toth (2010), who describe a number of heuristic and exact approaches to graph coloring.

A large assortment of graph coloring problem instances is available in the DIMACS graph coloring challenge (Johnson and Trick, 1996; Trick, 2005). This database contains 206 different graph instances ranging from easily-solved instances, to problems which are difficult for solvers but which were constructed to have a particular chromatic number, to problems for which the chromatic number is unknown.

2.6.3 Generalized Assignment

The generalized assignment problem is an NP-complete optimization problem that deals with the assignment of a set of jobs or tasks $J = \{1, 2, \ldots, n\}$ to a set of workers $W = \{1, 2, \ldots, m\}$. Assigning job j to worker i incurs a cost c_{ij}; moreover, performing this assignment uses up an amount w_{ij} of the ith worker's capacity ξ_i. The objective of the problem is to find an assignment of tasks to workers that satisfies all of the worker capacities and minimizes the total cost. If c_{max} is an upper bound on the cost of assigning tasks to workers, by performing the transformation $c'_{ij} = 1 + c_{max} - c_{ij}$, the problem can be instead viewed as a maximization problem where workers receive a profit or reward for completing a task (Martello and Toth, 1990).

The generalized assignment problem can be formulated as an integer program as follows:

$$
\begin{aligned}
\text{minimize} \quad & \sum_{i \in W, j \in J} c_{ij} y_{ij} \\
\text{subject to} \quad & \sum_{i \in W} y_{ij} = 1 \ \forall \ j \in J \\
& \sum_{j \in J} w_{ij} y_{ij} \le \xi_i \ \forall \ i \in W \\
& y_{ij} \in \{0, 1\} \ \forall \ i \in W, j \in J.
\end{aligned}
\tag{2.3}
$$

Here, the binary variables y_{ij} indicate that worker i has been assigned task j. The first set of constraints

ensures that every task is assigned to exactly one worker; the second set ensures that every worker does not exceed his capacity requirements. As with the graph coloring problem, this IP formulation produces relatively weak bounds and can have a lot of symmetry, so Dantzig-Wolfe reformulation is applied to produce the following IP with an exponential number of variables:

$$\begin{aligned} \text{minimize } & \sum_{i \in W} \sum_{C \in \mathscr{C}_i} c(C) y_C \\ \text{subject to } & \sum_{i \in W} \sum_{C: j \in C, C \in \mathscr{C}_i} y_C = 1 \ \forall \ j \in J \\ & \sum_{C \in \mathscr{C}_i} y_C \leq 1 \ \forall \ i \in W \\ & y_C \in \{0, 1\} \ \forall \ i \in W, C \in \mathscr{C}_i. \end{aligned} \tag{2.4}$$

In this reformulation, each set C is a complete assignment of tasks to a particular worker; the cost of such an assignment $c(C)$ is simply the sum of the costs of assigning each task in C to that worker. Note in particular the existence of n independent pools of columns, denoted $\mathscr{C}_1, \mathscr{C}_2, \ldots, \mathscr{C}_n$. Thus, when applying column generation, there are n distinct pricing problems that must be solved in order to find negative-reduced-cost columns. Each of the n pricing problems for the generalized assignment problem are weighted knapsack problems, where the weights of tasks are taken as the optimal dual weights for the first set of constraints in (2.4). Then the reduced cost for the schedule assignment is computed by subtracting the optimal dual price for the ith worker from the cost of the assignment returned by the pricing problem. The first B&P algorithm to solve the generalized assignment problem was that of Savelsbergh (1997).

An additional complication arises when attempting to use B&P methods for a generalized assignment problem: notably, in practice the dual prices for the LP relaxation of (2.4) tend to oscillate between very large (positive) and very small (negative) values. Because the weights in the pricing problem are derived from the dual prices, this means that the column generation procedure can demonstrate very slow convergence. To counteract this, Pigatti et al. (2005) use a stabilized column generation procedure which imposes bounds on the dual prices that are gradually relaxed as column generation progresses.

A database of generalized assignment problem instances is available through the OR Library (Beasley, 2012). This database contains 133 different problem instances, ranging from instances with 5 workers and 15 jobs up to instances with 80 workers and 1600 jobs. Many of the smaller instances in this database are trivial for modern solvers; however, the larger instances still pose some challenge.

2.6.4 Simple Assembly Line Balancing

The assembly line balancing problem is a well-studied problem with many applications, including the automotive industry, consumer electronics, and household items (Baybars, 1986; Sarker and Pan, 2001). This problem has many variants with different objectives and side constraints; see Battaïa and Dolgui (2013) for a recent survey of problem formulations and solution techniques. One of the most basic assembly line balancing problems is the Simple Assembly Line Balancing Problem (SALBP). In this problem, a set of jobs $J = \{1, 2, \ldots, n\}$ is given that must be accomplished by a set of m workers or stations. In many applications, stations are designed to complete specific tasks; however, the SALBP relaxes this assumption so that all stations are considered identical. Each task requires a certain amount of time t_j (called the **processing time**) to complete, and each station has a specified fixed amount of time ξ (called the **cycle time**) that it can spend completing tasks.

Additionally, a directed acyclic graph D, called the **precedence graph**, is given with vertex set J and arc set E. An arc $(i, j) \in E$ indicates that task i must be completed before task j. A task i is a **predecessor** (alternately, **successor**) of j if there is a path from i to j (alternately, from j to i) in G; if this path has length 1, i is a **direct** predecessor or successor. The set of direct predecessors (successors) of j is denoted Π_j (Φ_j), and the set of predecessors (successors) of j is Π^*_j (Φ^*_j).

The objective of SALBP is to find the minimum number of stations needed to complete all tasks, subject to the cycle time at each station and all relations given in the precedence graph. Given a set of tasks σ_m assigned to the mth station, define the **idle time** I_m as the amount of time the station is not working; that is, $I_m = \xi - \sum_{j \in \sigma_m} t_j$. For a complete assignment of tasks to stations, the total idle time I is the sum of the idle times at each station. Note that SALBP is NP-complete, since relaxing the precedence graph constraints yields a bin-packing problem.

A recent problem generator called `SALBPGen` (Otto et al., 2013) can be used to generate SALBP instances with a wide range of different structures and difficulties. These structures are described briefly below: firstly, the generator can create problem instances with a large number of **bottleneck tasks** and **chains**. A bottleneck task j has high in- and out-degree in D; furthermore, it is the only direct successor for at least two tasks in Π_j, and it is the only direct predecessor for at least two tasks in Φ_j. A chain of tasks, on the other hand, is a set of tasks $Ch \subseteq J$ with $|Ch| \geq 2$ such that Ch forms a path in D and $|\Pi_j| = |\Phi_j| = 1$ for each $j \in Ch$.

Another important property that can be controlled by `SALBPGen` is the **order strength**; this value, denoted by OS, is computed as $|E(D^+)|/\binom{n}{2}$, where $E(D^+)$ is the arc set of D^+, the **transitive closure** of D. That is, D^+ is the graph with vertex set J where arc (i, j) indicates that task i is a (not necessarily

direct) predecessor of task j. As stated in Scholl and Klein (1999), "Small values of OS indicate that the precedence constraints are not very restrictive such that many sequences of tasks are feasible." There are some indications that middle values of OS are harder than low or high order strength values.

Finally, `SALBPGen` can control the distribution of task times for each generated instance. Task times are randomly generated according to some pre-specified probability distribution. The problem database contains instances with task times that have been generated according to three different distributions, described below:

- Short task time distribution - task times are drawn from a normal distribution with the mean centered around small times
- Bimodal task time distribution - task times are drawn from a combination of two normal distributions with means centered around small and large times
- Centralized task time distribution - task times are drawn from a normal distribution with a mean task time of $\xi/2$

The first two task time distributions emulate properties seen in real-world instances of the assembly line balancing problem; the latter is designed to produce challenging instances.

The `SALBPGen` generator was used to generate a large database of SALBP instances that can be used for testing. The database contains instances with $n = 20, 50, 100$, and 1000 tasks (called small, medium, large, and very large, respectively). There are 525 instances of each problem size, which have been generated with varying order strengths and distribution of task times. A third of the problems (called BN instances) have been generated with bottleneck nodes having minimum degree eight (or minimum degree four in the small instances). A third (called CH instances) have been generated with 40% of the nodes in chains, and a third of the instances (called MIX instances) have no such requirements on the structure of the precedence graph. Finally, for each problem instance in the medium dataset, there are 9 additional permuted instances, which share a common precedence graph and set of task times, but have randomly assigned the task times to tasks. Thus, there are a total of 6825 instances in the dataset.

Chapter 3

Cyclic Best-First Search

As described in Section 2.2 the search strategy for a B&B algorithm (sometimes called the **node selection strategy**) determines the order in which subproblems are explored, in order to create new subproblems via branching. In this chapter, properties of a recent search strategy called **cyclic best-first search** (CBFS) are analyzed. The CBFS strategy is a generalization of other common search strategies, and can be thought of as an extension of the BFS strategy. Originally called *distributed best-first search* by Kao et al. (2009), CBFS has been used effectively in several different scheduling problems (Sewell et al., 2012; Vilà and Pereira, 2014).

The fundamental improvement of the CBFS strategy comes from the use of **contours** (or collections of subproblems) together with a measure-of-best function to guide the search process. The strategy groups "comparable" subproblems together, selecting new subproblems to explore by only comparing the value of the measure-of-best function μ within a contour, and repeatedly cycling through the set of contours. The use of contours gives algorithm designers two degrees of freedom when implementing a B&B procedure: what definition of μ to use, and what contour definition to use. The CBFS strategy is used to improve the diversification behavior of B&B in order to allow the algorithm to more quickly find an optimal solution to the problem.

Such cycling behavior has been used in a number of other applications in the literature as well. A heuristic algorithm by Choi et al. (2006) shares the cyclic best-first search name with the algorithm described herein; however, the behavior is somewhat different. In particular, the algorithm of Choi et al. (2006) uses a local cycling mechanism that enables intelligent backtracking in a heuristic branching algorithm, whereas this paper describes a global cycling mechanism that allows B&B to select subproblems for exploration from vastly different regions of the search tree. In the remainder of this dissertation, the term CBFS is used to refer to the latter search strategy, not the algorithm of Choi et al. (2006).

Another related algorithm by Shi and Ólafsson (2000) employs a B&B-like probabilistic heuristic for optimization problems called **nested partitioning**. The NP II variant of this algorithm uses a cycling mechanism similar to that of CBFS for the purpose of search diversification. Shi and Ólafsson (2000) prove that their heuristic algorithm converges almost-surely to a global optimal solution using this cycling behavior; however, this chapter describes a cycling strategy in the context of a (deterministic) exact algorithm. Finally,

the **local branching** algorithm of Fischetti and Lodi (2003) uses variable fixing methods (in contrast to cycling) to improve the diversification behavior of B&B solvers for mixed integer programs.

A third related algorithm to CBFS is called **restrict-and-relax** (Guzelsoy et al., 2013). The restrict-and-relax method allows the B&B algorithm to relax previously-made branching decisions. In this manner, the algorithm can move either up or down through the levels of the search tree, in a similar manner to the way that CBFS moves through the list of contours. The objective in this approach is to maintain a relatively small LP relaxation by fixing additional variables, while simultaneously relaxing branching decisions made on other variables to limit the size of the search space.

However, the CBFS strategy has a number of features not captured by these other algorithms. It turns out that the cycling mechanism of CBFS provides a large amount of fluidity to the strategy. In particular, it is shown that for any search strategy, there exists a contour definition for CBFS allowing it to simulate the other strategy's node exploration order, and thus CBFS provides a unifying framework for search strategies in B&B algorithms. CBFS also has practical benefit for algorithm designers; for example, using an appropriate contour definition can serve as a tie-breaker mechanism for μ, grouping subproblems with the same measure-of-best into different contours. This tie-breaking mechanism can more quickly drive the search to an optimal solution, increasing the amount of pruning that can be performed and reducing the tree size and search time. Thus, the primary contribution of this chapter is threefold: first, several key results that govern the behavior of the cyclic best-first search strategy are proved; secondly, heuristic properties of the contour definition are discussed that demonstrate how contours can influence the search process; and thirdly, computational results are presented for both mixed integer programming problems and the simple assembly line balancing problem that show the effectiveness of the CBFS strategy.

The remainder of this chapter is organized as follows: Section 3.1 introduces the CBFS strategy and discusses some benefits and drawbacks of the method. In Section 3.2, properties of the contour definition are proved and a proof showing that CBFS is a complete generalization of other search strategies is presented. Section 3.3 discusses heuristic methods for determining the best contour definition, and Section 3.4 provides some computational results on mixed integer programming problems and the simple assembly line balancing problem. Finally, Section 3.5 offers some concluding remarks and future research directions.

3.1 The Cyclic Best-First Search Strategy

The CBFS strategy can be viewed as a hybrid algorithm between DFS and BFS, whereby terminal subproblems are generated early in the search process (similarly to DFS), and a ranking function is used to aid in

the subproblem selection process (similarly to BFS). The CBFS strategy attempts to balance the **diversity** of the search (that is, the amount of time spent exploring different regions of the search space) with the **intensity** of the search (that is, the amount of time spent exploring similar regions of the search space in order to improve the incumbent) (Glover, 1990).

Formally, CBFS maintains a set $\mathscr{K} = \{\ldots, K_{-3}, K_{-2}, K_{-1}, K_0, K_1, K_2, K_3, \ldots\}$ of **contours**, or collections of unexplored subproblems, that are indexed by a **contour labeling function** $\kappa : 2^X \to \mathbb{Z}$. Additionally, CBFS uses a measure-of-best heuristic μ, which may or may not be admissible. As shown in Algorithm 3.1, the CBFS strategy keeps track of a list $\mathscr{I}$ of non-empty contour indices, together with a current contour index i; at each iteration, the strategy selects the best subproblem S with respect to μ from contour K_i (Line 3, Algorithm 3.1). When child subproblems are generated, they are inserted into their appropriate contours according to the labeling function (Lines 7-9, Algorithm 3.1). Contour indices are inserted to maintain a monotonically increasing order for $\mathscr{I}$ (Line 9, Algorithm 3.1). Before a new subproblem is selected for exploration, the contour index is updated (Lines 12-13, Algorithm 3.1).

Algorithm 3.1: Branch-and-Bound with Cyclic Best-First Search

1 Set $i = \kappa(X)$, $\mathscr{I} = \{i\}$, $K_i = \{X\}$, and initialize $\hat{x}$
2 **while** $\mathscr{I} \neq \varnothing$**:**
3   Let $S \in \arg\min_{S' \in K_i} \mu(S')$
4   **if** a solution $\hat{x}' \in \{x \in S \mid f(x) < f(\hat{x})\}$ can be found**:** Set $\hat{x} = \hat{x}'$
5   **if** S cannot be pruned**:**
6   Partition S into $S_1, S_2, \ldots, S_r$
7   **for each** $j \in 1, 2, \ldots, r$ **:**
8   Insert S_j into contour $K_{\kappa(S_j)}$
9   **if** $\kappa(S_j) \notin \mathscr{I}$**:** Insert $\kappa(S_j)$ into $\mathscr{I}$
10   Remove S from K_i
11   **if** $K_i = \varnothing$**:** Set $\mathscr{I} = \mathscr{I} - \{i\}$
12   **if** $\exists\, k \in \mathscr{I}$ s.t. $k > i$ **:** Set $i = \min\{k \mid k \in \mathscr{I}$ and $k > i\}$
13   **else:** Set $i = \min\{k \mid k \in \mathscr{I}\}$
14 Return $\hat{x}$

The contour index update mechanism ensures that the search process repeatedly cycles through all non-empty contours (hence the *cyclic* in CBFS). In particular, if there exists a non-empty contour with index greater than i, then i is set to the least index of such a contour. Otherwise, i is set to the least index over all non-empty contours.

As an example, consider the **depth labeling function** κ_d, which maps a subproblem to its depth in the search tree (see Figure 3.1a). Another example labeling function is the **positive assignment labeling function** κ_p; for this labeling the children of each subproblem are ordered, the contour label of the current subproblem is incremented for the “left” child subproblem, but is not incremented for the “right” child

subproblem (Figure 3.1b).

Examples of CBFS behavior for κ_d and κ_p on the search tree in Figure 3.1 are shown in Figure 3.2, as well as a comparison to the behavior of BFS on the same search tree. Observe that, despite the same measure-of-best function, the order of the explored subproblems is quite different for each of the three search strategies. In particular, note how the contour definition serves as a tie-breaking mechanism for subproblems with the same measure-of-best value. Also note that the contour definition can force the search to explore subproblems with a (comparatively) poor value for μ by placing those subproblems as the only element in their contours. For example, both BFS and CBFS with κ_d explore the right-most branch of the search tree first, because this branch has the best μ value. On the other hand, CBFS with κ_p explores the left-most branch of the search tree first, due to the structure of the contours.

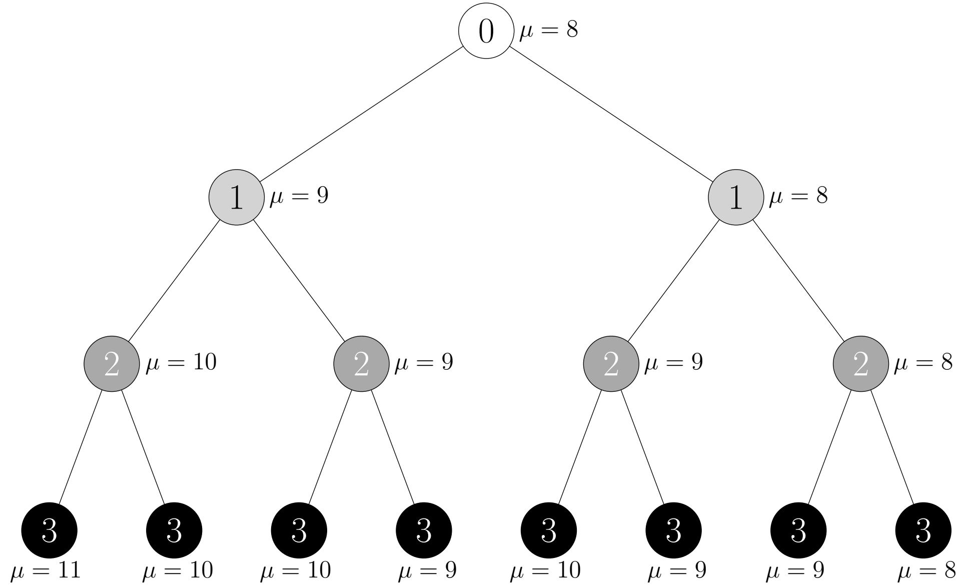
Further analysis of the CBFS strategy provides motivation for the term "contour", which is borrowed from the field of topography. There, the term *contour* or *level set* indicates a set of points for which some function takes the same value. Here, the term is used to indicate a collection of subproblems which should be considered equivalent or comparable with respect to μ, since the strategy never compares two subproblems residing in different contours. The meaning of "comparable" with respect to B&B is almost certainly problem- or instance-specific, and the possible choices for the labeling function κ are infinite. As will be shown, the choice of κ is a dominant factor in the performance of the strategy.

3.1.1 Implementing CBFS

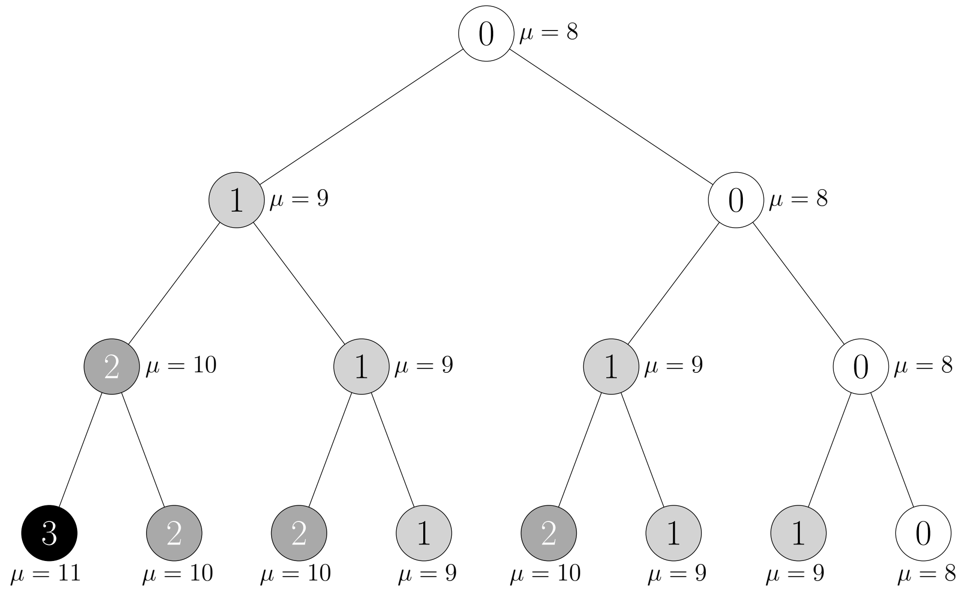
CBFS can be easily implemented using two standard data structures, an ordered map and several heaps. The ordered map data structure maps contour labels to the contours themselves. If a contour label is not present in the map, the contour is assumed to be empty. Furthermore, the ordered map can be traversed in order of increasing contour label. Such an ordered map can be implemented using a red-black tree; insertion, deletion, and look-up in the tree can be performed in $O(\log |\mathrm{supp}(\mathscr{K})|)$ time, where $\mathrm{supp}(\mathscr{K})$ is the **support** of $\mathscr{K}$, that is, the complete set of non-empty contours ever generated in the algorithm. Additionally, a single traversal of the ordered map can be done in $O(|\mathrm{supp}(\mathscr{K})|)$ amortized time (Cormen et al., 2009).

Moreover, each non-empty contour K_i can be implemented using a heap data structure, which stores a collection of elements in a nearly complete binary tree structure which obeys the **heap property**: each element in the tree is smaller than all its children. The use of the heap allows for $O(\log |K_i|)$ insertion of new subproblems into the contour, and the subproblem minimizing the measure-of-best heuristic can likewise be found and removed from the contour in $O(\log |K_i|)$ time, where $|K_i|$ is the maximum size of the ith contour over the course of the algorithm (Cormen et al., 2009).

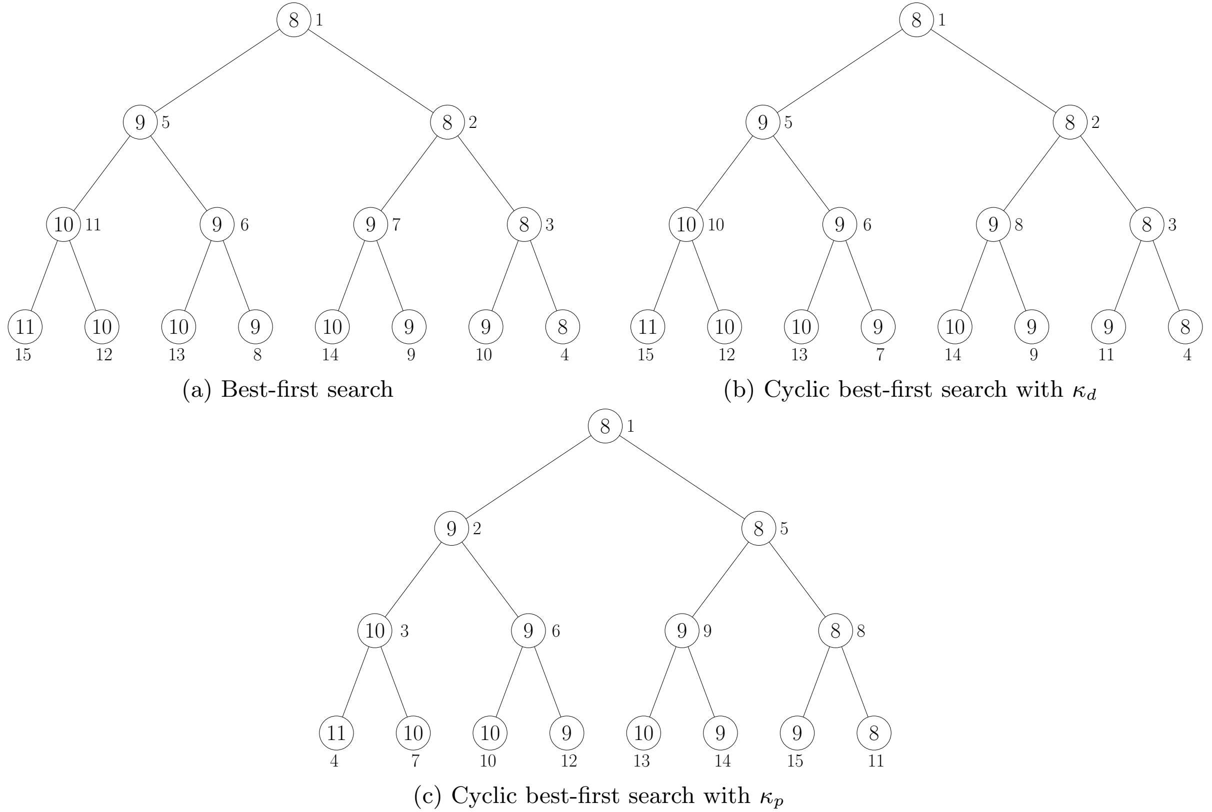
Figure 3.1: Examples of two different contour labeling functions for the same B&B tree. Contour labels are given in the center of each node, and darker node shades indicate "deeper" contours. The value of the measure-of-best function μ is also given for each subproblem.



(a) The depth labeling function κ_d; the contour label of a subproblem is equal to its distance from the root subproblem.



(b) The positive assignment labeling function κ_p; the contour label of a subproblem is equal to the number of "left" branches taken on the path from the root.



(a) Best-first search

(b) Cyclic best-first search with κ_d

(c) Cyclic best-first search with κ_p

Figure 3.2: A comparison of the behavior of CBFS with two different labeling functions to the behavior of BFS on the search tree shown in Figure 3.1. Values of μ are in the center of each node, and search order is outside.

3.1.2 Benefits and Drawbacks of CBFS

Depending on the choice of labeling function, the CBFS strategy has the potential for two improvements over other search strategies.

1. Early determination of high-quality incumbents: due to the cyclic nature of the strategy, it is likely that many diverse terminal subproblems will be explored much earlier in the search process than either DFS or BFS. By exploring more terminal subproblems, the strategy increases the likelihood that a good incumbent solution is found early in the search. This often provides a substantial boost in performance, as a better incumbent solution can be used to prune more subproblems higher in the tree.

2. Ability to distinguish between similar subproblems: suppose that for some problem instance, the search tree has many subproblems such that $\mu(S)$ equals the (global) optimal value. Such problem instances arise, for example, in cases where there are a small number of discrete possible values that $f(x)$ can assume. This phenomenon is common in many scheduling problems, for instance. In this case, BFS

has no capability to break ties between subproblems and may spend a large amount of time in middle levels of the search tree without ever generating a complete solution. In contrast, CBFS is forced to lower levels of the search tree, where it may find a terminal (optimal) subproblem that can be used to prune away the other similar subproblems in the middle levels.

However, these improvements come at a cost; unlike DFS, CBFS requires significantly more memory overhead, since it must store all of the subproblems in all of the non-empty contours, whereas DFS only needs to store a single path through the search tree at a time. Additionally, if the measure-of-best function is very tight (meaning in general it provides an accurate reflection of the direction of improvement), CBFS may spend more time than BFS exploring unnecessary regions of the search space. Finally, the operations needed to search and maintain the ordered map impose some overhead over other search strategies. However, if the contour labeling function is chosen correctly, the amount of pruning achieved has the potential to outweigh these sacrifices in space and time complexity. Furthermore, as the following result shows, under some reasonable assumptions, BFS outperforms CBFS by at most a factor of $|\text{supp}(\mathscr{K})|$; this result is similar in spirit to (and makes use of) the result of Dechter and Pearl (1985) showing that the A* algorithm explores the fewest number of subproblems in a B&B algorithm.

Theorem 3.1. *Let μ be an admissible measure-of-best for a B&B algorithm that does not use dominance relations, and assume that there does not exist any pair of subproblems S, S' for which $\mu(S) = \mu(S')$. Furthermore, assume that there is exactly one optimal solution in X. Then, for a fixed labeling function κ, let $S_1, S_2, \ldots, S_m$ be the sequence of subproblems explored by BFS using μ. Then CBFS generates children at no more than $m|supp(\mathscr{K})|$ subproblems.*

Proof. Let S_{opt} be the terminal subproblem yielding the optimal solution. Note that by Dechter and Pearl (1985), CBFS explores S_{opt} no earlier than BFS (in terms of number of iterations). Further, note that once CBFS explores S_{opt}, there may be a large number of unexplored subproblems in T that were never explored by BFS. It must be the case that for every such subproblem S, $\mu(S) \geq f(x^*)$, because otherwise they would have been explored by BFS. Therefore, once CBFS finds S_{opt}, each such subproblem can be pruned without generating children, and moreover, any subproblem at which CBFS generates children must also be explored by BFS.

The proof proceeds by showing that on every pass through the collection of non-empty contours before CBFS finds S_{opt}, CBFS explores (and possibly generates children at) at least one subproblem that is explored by BFS. Since there are at most $|\text{supp}(\mathscr{K})|$ non-empty contours at any point in the algorithm, and CBFS explores exactly one subproblem from each non-empty contour on each pass, this will prove the desired result.

To see this, consider some pass through the collection of non-empty contours where CBFS does not explore any subproblem in $S_1, S_2, \ldots, S_m$; in other words, at every non-empty contour K_i, CBFS selects some subproblem to explore that BFS never explores. There are two reasons why this can happen: 1) K_i currently contains no subproblems explored by BFS, or 2) the subproblem explored by CBFS in K_i has a better measure-of-best across K_i than any subproblems explored by BFS in K_i.

Consider the second case above; in particular, let $S' \in K_i$ be a subproblem for which CBFS generates children but is not explored by BFS, and suppose that at the time CBFS explores S' there exists a subproblem $S_j \in K_i$ that is explored by BFS. Then, by definition and the fact that there are no ties in μ, it must be the case that $\mu(S') < \mu(S_j)$. Moreover, since BFS does not explore S', there must be some terminal subproblem $\hat{S}$ explored by BFS such that $f(\hat{S}) \leq \mu(S')$ (where $f(\hat{S})$ is the value of the incumbent solution found at $\hat{S}$ by BFS).

This implies that $f(\hat{S}) < \mu(S_j)$, which in particular means that S_j is not an ancestor of $\hat{S}$ (if S_j is an ancestor of $\hat{S}$, the best solution in $\hat{S}$ is also in S_j, which violates the admissibility of μ). Therefore, in the jth iteration of BFS, there exists an unexplored ancestor of $\hat{S}$ which has a smaller measure-of-best than S_j (again by the admissibility of μ). Thus, BFS should not have explored S_j in the jth iteration, which is a contradiction.

Therefore, in the pass in which CBFS explores no subproblems explored by BFS, every non-empty contour must contain no subproblems explored by BFS. However, this is impossible, since CBFS has not found S_{opt}, which means that some ancestor of S_{opt} is unexplored. ■

Note that some care was taken in the above proof to differentiate between *exploring* a subproblem and *generating children* at a subproblem. The result relates the number of subproblems at which CBFS generates children to the number of subproblems BFS explores—this is necessary, because it could be the case that when CBFS finds S_{opt}, it has generated a large number of children that BFS never explores, perhaps due to the chosen branching rule. Thus, upon finding S_{opt}, CBFS must go through a "clean-up" phase to remove all of these subproblems from the tree. However, in general, such a clean-up phase requires relatively little extra computational effort.

3.2 Properties of the Labeling Function

In this section, properties of the labeling function are explored in greater detail. Before proceeding, one important property is required:

Definition 3.1. *A B&B algorithm is said to be* **memoryless** *if the children, bounds, and incumbent solution produced at a subproblem S depend only on S and not the past history of the algorithm.*

Due to possible degeneracy in the LP relaxation solution, the presence of algorithmic enhancements such as cutting planes, and computational numerical instabilities, many commercial integer programming solvers are not memoryless. In particular, given subproblems S_1 and S_2, the lower bounds and generated children at S_2 can depend on whether S_1 has been explored or not. However, other B&B algorithms that do not rely on linear programming methods may satisfy the memorylessness requirement. All results in this section assume a memoryless solver.

3.2.1 The Simulation Labeling Function

The first result in the section establishes the generality of the CBFS strategy by providing a contour labeling function κ that enables CBFS to simulate the behavior of any other search strategy.

Definition 3.2. *For a search strategy $\mathscr{A}$, define $\mathscr{A}(j)$ to be the subproblem explored by $\mathscr{A}$ in the jth B&B iteration. For a specific search strategy such as CBFS, the notation CBFS(j) is used. The inverse function $\mathscr{A}^{-1}(S)$ is defined to be the B&B iteration in which subproblem S is explored.*

It is assumed that search strategies explore each subproblem exactly once, so that $\mathscr{A}^{-1}$ is well-defined. Some search strategies, such as iterative deepening, explore a subproblem multiple times; for the purposes of the following result, the inverse function is defined to be the first time that a subproblem is explored. In such cases, CBFS does not explore the exact sequence of subproblems as $\mathscr{A}$, but instead explores subproblems in the order that they are first explored by $\mathscr{A}$.

Theorem 3.2. *Using the* **simulation labeling function** *defined as $\kappa_{sim}(S) = \mathscr{A}^{-1}(S)$, CBFS will explore the same sequence of subproblems as $\mathscr{A}$.*

Proof. Consider the sequence of subproblems $\mathscr{A}(1), \mathscr{A}(2), \ldots, \mathscr{A}(m)$ explored by search strategy $\mathscr{A}$. Note that $\mathscr{A}(1) = \text{CBFS}(1) = X$, and that the current contour index i for CBFS is initially 1. Next assume that CBFS and $\mathscr{A}$ have explored the same initial sequence of $j-1$ subproblems, and at the $(j-1)^{st}$ iteration the contour index i is equal to $j-1$, for $2 \leq j \leq m$. $\mathscr{A}(j)$ must be a child of some subproblem $\mathscr{A}(j')$ with $j' < j$, which has been previously explored by CBFS. Therefore, $\mathscr{A}(j)$ has been generated by CBFS and is the sole subproblem in contour j, by definition of κ_{sim}. Since j is the next smallest non-empty contour index larger than i, $\text{CBFS}(j) = \mathscr{A}(j)$ and $i = j$, so the result holds by induction. ∎

It may not always be possible to calculate κ_{sim} exactly for an arbitrary search strategy; however, for the three most common search strategies (BFS, BrFS, and DFS), a surrogate labeling function can be used to produce the same results.

Theorem 3.3. *Using $\kappa_{BFS}(S) = 0$, CBFS will emulate BFS.*

Proof. Since every subproblem appears in the same contour, CBFS and BFS will make identical choices, assuming they use the same measure-of-best function μ. ■

Theorem 3.4. *Let S_j be the jth subproblem generated by BrFS; then, using $\kappa_{BrFS}(S_j) = j$, CBFS will emulate BrFS.*

Proof. BrFS is typically implemented using a first-in, first-out queue (Cormen et al., 2009). By definition of a queue, then BrFS(j) = S_j, so by Theorem 3.2 CBFS will simulate BrFS with this labeling function. ■

Theorem 3.5. *Consider a subproblem S with children $S_1, S_2, \ldots, S_r$, and let $\Delta_j, j \in \{1, 2, \ldots, r\}$ be an upper bound on the number of subproblems contained in the subtree rooted at S_j. Define*

$$\kappa_{DFS}(S_j) = \kappa_{DFS}(S) + \sum_{k=0}^{j-1} \Delta_k.$$

Using this labeling function, CBFS will emulate DFS.

Proof. For any subproblem S with children $S_1, S_2, \ldots, S_r$, the DFS strategy explores all subproblems contained in the subtree rooted at S_j before exploring any children in subsequent subtrees. Using κ_{DFS}, no children in subsequent subtrees appear in earlier contours than children of S_j, proving the result. ■

In the last case, bounds Δ_j can be computed if upper bounds on the branching factor and subtree depth are known. For example, given a binary branching strategy and a finite number of branching choices, then $\Delta_j = 2^k$, where k is the number of remaining branching choices at S_j.

3.2.2 The Optimal Labeling Function

This section discusses the existence of an "optimal" contour labeling function for a particular problem instance. Intuitively, the best B&B algorithm will be one in which the first terminal subproblem is optimal. This enables the maximum amount of pruning to be performed, which in turn produces the smallest search tree and the fastest computation time. The following theorem establishes the existence of an (problem-instance specific) labeling function that enables CBFS to find an optimal solution before any other search strategy:

Theorem 3.6. *Fix a branching strategy and pruning rules for a B&B algorithm. Then, there exists a labeling function for CBFS which finds an optimal solution at least as quickly as any other search strategy $\mathscr{A}$.*

Proof. The statement is a natural corollary of Theorem 3.2 by letting $\mathscr{A}$ be a search strategy that finds an optimal solution as quickly as possible among all search strategies. However, an alternate proof of this theorem is presented for the purposes of discussing the computational complexity of finding the "best" contour ordering.

Assume that $\mathscr{P}$ has at least one feasible solution, and consider a shortest path from the root of the B&B tree to any optimal solution under the chosen branching strategy, say $S_1, S_2, \ldots, S_k$, where $S_1 = X$ and S_k is terminal. Define the labeling function

$$\kappa_{opt}(S) = \begin{cases} i & \text{if } S = S_i, i \in \{1, 2, \ldots, k\} \\ k+1 & \text{otherwise} \end{cases}$$

Such a labeling function ensures that each subproblem on the shortest path to an optimal solution is placed in its own contour and that the algorithm explores these subproblems before any other subproblem. Since the branching strategy, pruning rules, and LP relaxation solution method are all memoryless and fixed, no search strategy $\mathscr{A}$ can find an optimal solution more quickly than CBFS with this labeling function. ■

Note that the above result is purely an existence result, since to actually compute κ_{opt}, not only must an optimal solution be known, but it must be possible to prove that no other shorter path exists. This is usually a more challenging problem than just solving the optimization problem to begin with.

For example, suppose a B&B solver is being used to solve the satisfiability (SAT) problem, in which a satisfying assignment to a boolean formula $\phi(y_1, y_2, \ldots, y_n)$ is sought. Further suppose that subproblems at the ith level of the search tree branch by fixing variable y_i to either true or false. Then, explicitly computing κ_{opt} for this problem is equivalent to finding the smallest value k and an assignment of values to $y_1, y_2, \ldots, y_k$ such that ϕ is true irrespective of the remaining variables. Or, formally, this is equivalent to solving

$$\min_k \ \exists\, y_1, y_2, \ldots, y_k \ \forall\, y_{k+1}, y_{k+2}, \ldots, y_n \ \phi(y).$$

The decision version of this problem, which provides a fixed value for k and asks if a satisfying assignment to $y_1, y_2, \ldots, y_k$ exists for ϕ, is known as the 2-quantified boolean formula (2QBF) problem, which is Σ_2^p-complete. Σ_2^p is the computational complexity class of languages which can be verified in polynomial time, given access to an NP oracle (that is, a procedure that solves some NP-complete problem, e.g. 3SAT, in constant time); equivalently, Σ_2^p is the complexity class $\mathrm{NP}^{\mathrm{NP}}$. It is generally believed that Σ_2^p encompasses a more difficult class of languages than NP does, but this conjecture—a generalization of P $\neq$ NP—has not been proven. For a more detailed discussion of QBF problems and Σ_2^p, see Ranjan et al. (2004).

3.3 Finding Good Labeling Functions

From the perspective of designing faster algorithms in practice, the results from Theorems 3.2-3.6 are not particularly useful. In addition to the high computational complexity of finding a good labeling function discussed at the end of Section 3.2.2, none of the results in these theorems take advantage of the defining characteristic of CBFS—namely, cycling. Both κ_{sim} and κ_{opt} perform exactly one pass through the various contours before the algorithm terminates or before all remaining subproblems are placed in a single contour. Therefore, the natural question is "What heuristic labeling functions lead to good performance for B&B algorithms in practice?"

3.3.1 Relative Properties of the Labeling Function

It is important to note that the behavior of a B&B algorithm with CBFS is independent of the *absolute* values of the labels produced by κ. In particular, note that global scaling of the labeling function does not change the behavior of the algorithm. In other words, for $\beta \in \mathbb{Z}^+$ and some labeling function κ, the new labeling function $\kappa'(S) = \beta\kappa(S)$ will generate exactly the same search tree, since the β factor just inserts more empty contours in between each consecutive pair of non-empty contours. More generally, it can be observed that the labeling function only impacts the algorithm based on the *relative* contour orderings it produces. An example of this can be seen by comparing κ_{DFS} to κ_{sim}—both produce the same relative ordering, even though different absolute contour labels are assigned to each subproblem.

Therefore, this section focuses on the relative behavior of the search strategy with respect to the labeling function. Consider a subproblem S that is explored and generates two children, S_1 and S_2. How do different contour labels for S_1 and S_2 change the behavior of the algorithm? The following definition will be important:

Definition 3.3. *A subproblem S is* **considered for exploration** *if it is present in contour K_i when CBFS selects a subproblem to explore from K_i.*

Just because a subproblem is considered for exploration at some iteration of the algorithm does not mean that it will actually be explored at that point. In fact, a very large number of iterations could occur between when a subproblem is first considered for exploration and when it is finally explored. The important point here is that a subproblem will *never* be explored in any iteration in which it was not considered for exploration. If two subproblems are placed in the same contour, at most one of them will be explored on this pass through the list of contours (Figure 3.3a).

With regards to the three subproblems S, S_1, and S_2 mentioned above, first suppose that $\kappa(S) < \kappa(S_1) < \kappa(S_2)$; in this case, S_1 will be considered for exploration before S_2 (Figure 3.3b). It does not mean that S_1

actually *will* be explored before S_2, because there may be many other subproblems in $K_{\kappa(S_1)}$ with better μ values. However, also observe that both S_1 and S_2 will be considered for exploration before CBFS returns to the top of the non-empty contour list.

On the other hand, suppose that $\kappa(S_1) < \kappa(S) < \kappa(S_2)$ (Figure 3.3c). In this case, S_1 will not even be considered for exploration until CBFS has reached the bottom of the non-empty contour list. In other words, by putting S_1 in a higher contour than S, the labeling function introduces a (potentially large) delay before S_1 has the potential to be explored. This may be beneficial if the algorithm is able to find a new incumbent solution that can be used to prune S_1. Moreover, unlike before, subproblem S_2 will be considered for exploration before S_1, despite the fact that $\kappa(S_1) < \kappa(S_2)$ in both cases.

In the extreme case, when $\kappa(S_2) = \kappa(S)$, subproblem S_2 is not considered for exploration until CBFS has explored some subproblem from every other non-empty contour (Figure 3.3d). This introduces the longest possible delay between when S_2 is generated and when it is first considered for exploration. Thus, the contours act similarly to a tabu list for local search, by introducing a delay between when a subproblem is generated and when it can be explored.

3.3.2 Heuristic Labeling Function

Based on the discussion in Section 3.3.1, a heuristic labeling function for integer linear programming using the standard integer branching rule (see Section 2.6.1) is presented. This heuristic is a generalization of the depth labeling function and the positive assignment labeling function (Section 3.2). For a subproblem S, define S^+ and S^- to be the number of positive or null assignments made at branching decisions leading to S, respectively. Then, let p and n be user-specified weights, and consider the following labeling function:

$$\kappa_{p,n}(S) = pS^+ + nS^- \qquad (3.1)$$

Assume that the weights p and n are rational; then the first observation is that by the scaling argument of Section 3.3.1, these weights can be restricted to integer values. Secondly, note that the relative settings of p and n directly yield different orderings for children of a subproblem as considered in Figure 3.3. For example, given two children S_p and S_n of a subproblem S, where S_p is formed by a positive assignment from S and S_n is formed by a null assignment, observe that if $p = n > 0$, this leads to the configuration in Figure 3.3a. Likewise, if $p < 0$ and $n > 0$, this leads to the configuration shown in Figure 3.3c.

Moreover, observe that this labeling function can be used to implement a number of different previously-used labeling functions. For instance, if $p = n = 0$, then the resulting search strategy is simply BFS. Assigning $p = n = 1$ yields the depth-based contours shown in Figure 3.2b, and $p = 1, n = 0$ yields the

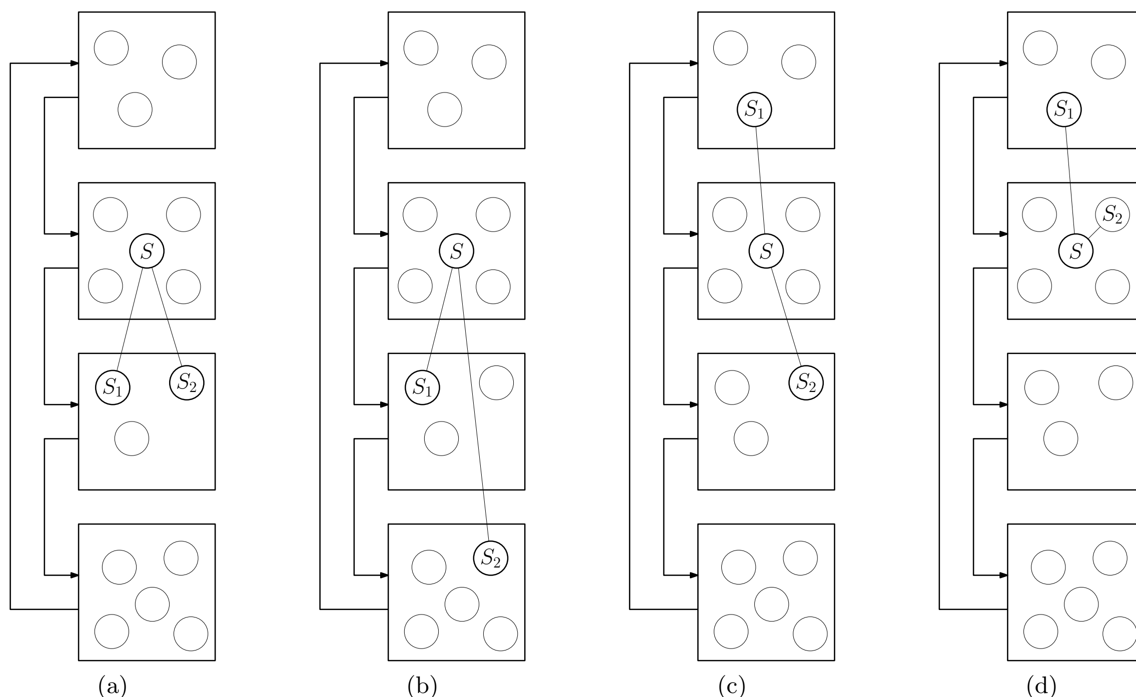
Figure 3.3: Relative contour labels for subproblems. In this example, subproblems are grouped into four contours, and subproblem S has just been explored. The differences in search strategy behavior are considered for different relative contour labels for the two children S_1 and S_2 of S.

(a) $\kappa(S) < \kappa(S_1) = \kappa(S_2)$; S_1 and S_2 are placed in the same contour, and thus are considered for exploration at the same time, but only one subproblem will be selected for exploration from this contour on this pass through the contour set. The explored subproblem from this contour might not be either of S_1 or S_2.

(b) $\kappa(S) < \kappa(S_1) < \kappa(S_2)$; S_1 is considered for exploration before S_2, but both will be considered for exploration before the algorithm returns to the top of the contour set.

(c) $\kappa(S_1) < \kappa(S) < \kappa(S_2)$; S_2 is considered for exploration before S_1. Moreover, subproblem S_1 will not even be considered for exploration until a subproblem from every contour with label greater than $\kappa(S)$ has been explored.

(d) $\kappa(S_1) < \kappa(S_2) = \kappa(S)$; S_1 is considered for exploration before S_2. Neither subproblem will be considered for exploration until a subproblem from every contour greater than $\kappa(S)$ has been explored, and S_2 will not be considered for exploration until a subproblem from every other contour is explored.



positive assignment contour function shown in Figure 3.2c.

The idea behind this heuristic labeling function is that for some problems, performing positive assignments are more likely to lead to good candidate incumbents than performing null assignments, or vice versa. Setting the weights appropriately allows the algorithm designer to use domain knowledge or experience to control how child subproblems are arranged in contours. Note that even more complicated labeling functions are possible. For example, the labeling function could take into account the variables that are selected for branching, in the spirit of pseudocosts or strong branching.

3.4 Computational Results

3.4.1 Mixed Integer Programming

To demonstrate the behavior of contours on algorithm performance, an extensive suite of tests was run on problems from MIPLIB 2010 (Section 2.6.1). The implementation of CBFS was written using callbacks with CPLEX 12.5, and was written in C++. The performance of CPLEX without CBFS was compared to CPLEX using CBFS and 12 different combinations of values for p and n (shown in Table 3.1). Tests were run against all 87 benchmark instances in the MIPLIB 2010 benchmark dataset. Additionally, tests were run against 160 of the problems in the MIPLIB 2010 challenge dataset; four problems were excluded due to memory limitations.

(0, 0)	(1, 1)	(1, 0)	(0, 1)
(1,-1)	(-1,1)	(3, 1)	(1, 3)
(1,-3)	(-1,3)	(3,-1)	(-3,1)

Table 3.1: The twelve combinations of parameter values used for CBFS. Note that (0, 0) is simply BFS, and (1, 1) is the depth contour function.

In order to provide the most fair comparison, CPLEX's parallel search and dynamic search options were disabled for all trials. A time limit of one hour was imposed for all computational tests. Each test was run on a single core of an Intel Core i7-930 2.8 GHz quad core desktop computer; four tests were run simultaneously to reduce the total testing time. For each instance, thirteen different trials were performed. The first used the default branching strategy used by CPLEX; the remaining twelve used CBFS with $\kappa_{p,n}$ as the labeling function, with weights taken from Table 3.1. Data were collected for all trials to determine the total number of nodes explored in the branch-and-bound tree, the iteration in which the optimal solution was first identified, and the total computational time needed to solve the instance.

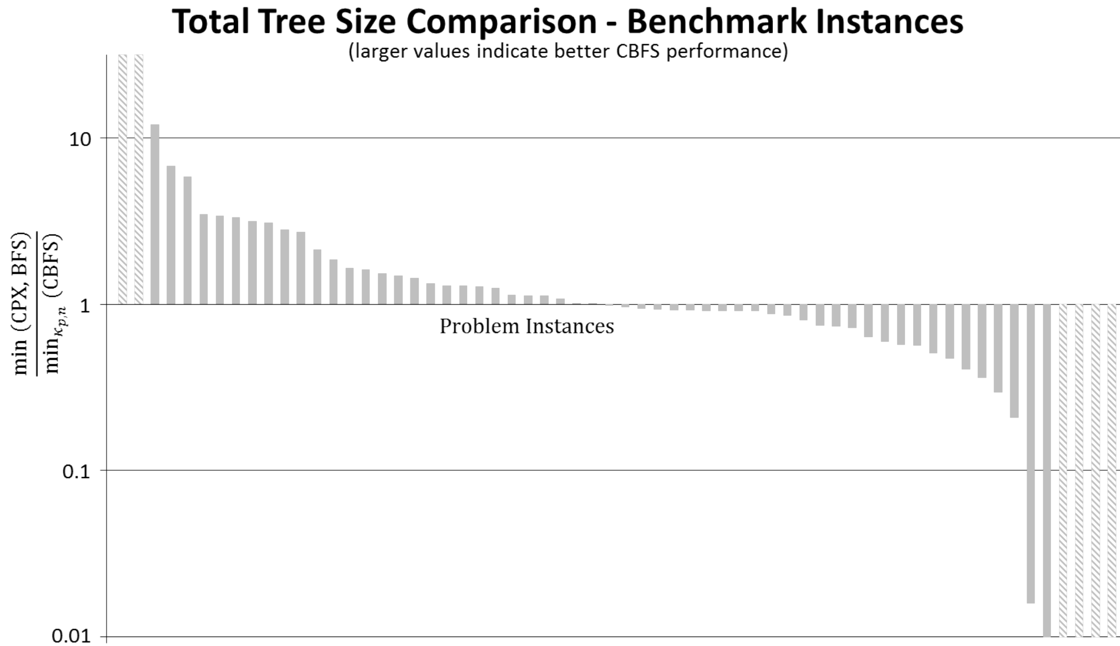


Figure 3.4: Comparison of tree size for the 62 MIPLIB benchmark instances solved using at least one search strategy, and not solved at the root node. Each data point is the ratio between the size of the smallest tree explored by CPX/BFS, and the smallest tree size explored by any CBFS variant. The values are shown on a log scale; each axis tick indicates an order-of-magnitude change in performance. Hashed bars represent instances that were only solved by some CBFS variant or CPX/BFS, but not both.

Benchmark Instance Tests

It was observed that in most cases the presence of callbacks negatively impacted the performance of the algorithm, even if the default CPLEX search strategy was used; this is due to some internal restructuring and clean-up that CPLEX performs when callbacks are present that does not need to be performed without callbacks (Achterberg, 2009). For example, running CPLEX on some problem instances without callbacks enabled allowed the solver to find a very good (or optimal) solution at the root node of the B&B tree. However, enabling callbacks for those same instances sometimes prohibited the solver from finding any incumbent solution within the 1-hour time limit, even without using CBFS or BFS. Thus, in order to isolate the impacts of CBFS on the algorithm performance on the benchmark instances, CPLEX was run with callbacks enabled for all trials; for trials that did not use CBFS, the default subproblem selection strategy was employed. In the remainder of this section, CPX refers to the CPLEX solver with callbacks enabled and the default subproblem selection strategy.

With these modifications, CPX was able to verify optimality for 64 instances in the benchmark dataset, and CBFS with $\kappa_{0,0}$ (henceforth called BFS) was able to verify optimality for 1 additional instance. Five of

Figure 3.5: Comparison of computational running time on a log scale for the 62 benchmark instances solved by at least one search strategy (see Figure 3.4). Hashed bars indicate instances that were solved only by a CBFS variant or by CPX/BFS. The length of hashed bars is computed as a ratio between the time taken by the best strategy and 3600 seconds.

these instances were solved at the root of the B&B tree. When CBFS with one of the 11 remaining labeling functions was used, the solver was able to verify optimality for 2 more instances that could not be solved by either CPX or BFS. However, there were 5 instances which could be solved by CPX or BFS, but not by CBFS with any other labeling function. Thus, there are a total of 67 instances in the benchmark dataset which could be solved by CPLEX using at least one of the 13 different search strategies, and 20 instances which could not be solved within one hour of computation time using any search strategy.

Of these 62 instances which were not solved at the root node of the B&B tree, at least one of the 11 CBFS variants outperformed CPX or BFS (with respect to the number of subproblems contained in the search tree) in 30 cases. On average, the best search tree explored by some CBFS variant for these instances was 42% smaller than the best search tree explored by CPX or BFS, and in one case, the size of the search tree for CBFS was an order of magnitude smaller than for CPX or BFS. For the remaining 32 instances, CPX or BFS explored a smaller search tree than CBFS, with an average improvement of 31%, and 1 instance which produced a tree an order of magnitude smaller with CPX than with any CBFS variant. These data are summarized in Figure 3.4, which shows the ratio of the best tree size for CPX or BFS to the best tree size across all CBFS variants.

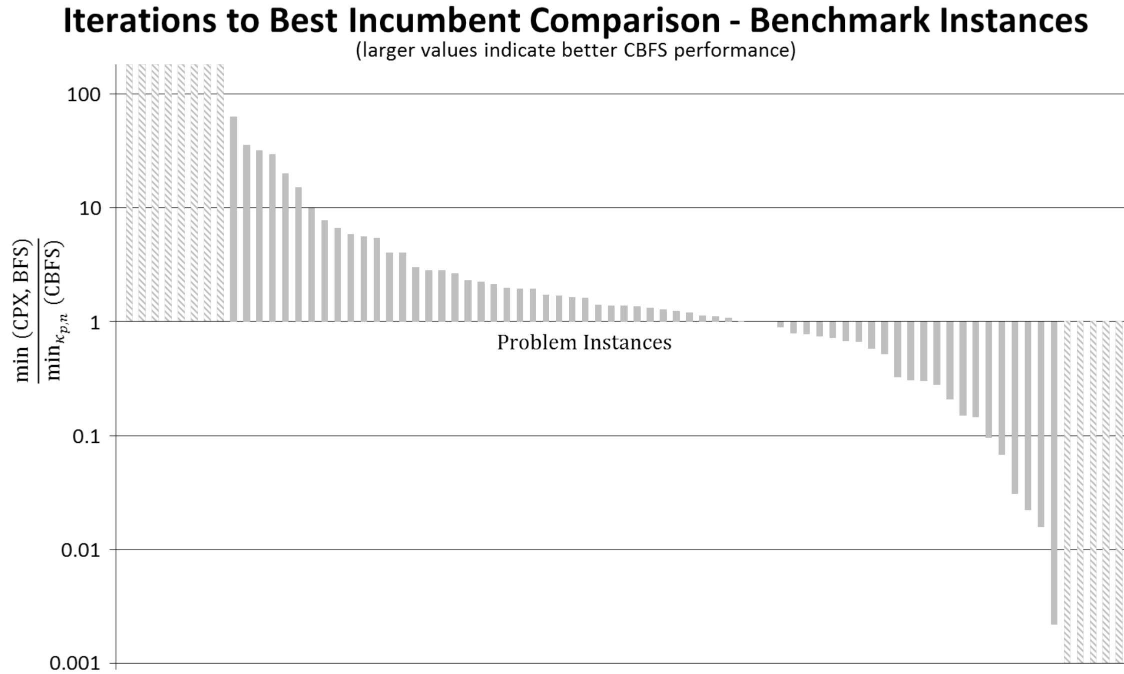


Figure 3.6: Comparison of the number of iterations to the best incumbent solution found for the MIPLIB benchmark instances, shown on a log plot. Hashed bars indicate instances which only CBFS or CPX/BFS found an incumbent solution, but not both.

A similar comparison was made with respect to computational running time. Here, CBFS outperforms CPX and BFS on 26 instances, one of which is by at least an order of magnitude. The average improvement in running time of CBFS over CPX or BFS is 35%; conversely, CPX or BFS outperforms all CBFS variants in 36 cases, with an average improvement of 33%, and two instances which are solved at least an order of magnitude more quickly than any CBFS variant. These data are plotted in Figure 3.5.

Figure 3.6 presents data on the number of iterations until the solver found the optimal solution (or in the case of the 20 unsolved instances, until the best incumbent was found). Here, at least one of the 11 CBFS variants found the best incumbent in fewer iterations than either CPX or BFS in 48 instances, and for 8 of these instances CPX and BFS were unable to find any incumbent solution within the time limit. In 7 cases, CBFS was able to find an incumbent at least an order of magnitude more quickly than CPX or BFS, and the average improvement in the number of iterations to the best solution was 55%. On the other hand, CPX or BFS found the best incumbent solution in fewer iterations than any CBFS variant for 27 instances. In 5 cases, no CBFS variant was able to find an incumbent solution, and in 6 cases the improvement is at least an order of magnitude. The average improvement in number of iterations to best for CPX and BFS was 63%. Detailed statistics on the comparison between CPX, BFS, and CBFS with the 11 different labeling

functions are shown in Table A.1.

Finally, to get a sense for which search strategies performed the best and worst overall, a tally was performed for each search strategy of the number of times it was the best with respect to iterations to best solution, total tree size, and total computation time. Results of this tally are shown in Table 3.2. The entries in each column indicates how many times that search strategy performed the best or worst with respect to the three metrics; the first two rows in each section compares across all search strategies, and the second two exclude CPX and BFS from the comparison.

Table 3.2: A tally of the number of times each search strategy performed the best in terms of iterations to best incumbent, total tree size, and computational running time, for the MIPLIB benchmark instances. The first two rows in each section compare to CPX and BFS; the second two only consider the best and worst CBFS labeling functions.

	CPX	BFS	1, 1	1, 0	0, 1	3, 1	1, 3	1, -1	-1, 1	1, -3	-1, 3	-3, 1	3, -1
Iterations to best incumbent													
best	27	3	4	8	4	10	5	7	8	8	4	4	5
worst	4	11	8	6	3	6	5	3	8	3	3	4	6
best (CBFS only)			8	12	6	15	5	11	10	8	4	5	5
worst (CBFS only)			11	8	5	7	6	3	9	6	3	4	7
Total tree size													
best	23	9	1	7	2	4	1	7	2	4	0	1	1
worst	4	9	4	5	1	7	4	3	4	3	6	1	5
best (CBFS only)			7	9	5	6	1	11	8	4	1	4	1
worst (CBFS only)			6	8	3	7	5	5	5	4	8	3	5
Total computation time													
best	24	12	2	6	2	4	2	4	1	2	0	2	1
worst	4	12	20	13	12	15	15	7	8	11	12	8	11
best (CBFS only)			9	7	4	9	3	8	4	3	3	6	1
worst (CBFS only)			20	13	14	16	16	8	8	11	12	9	12

Challenge Instances

An additional set of trials were performed on 160 instances in the MIPLIB challenge dataset. For these problems, CPLEX was run without callbacks enabled (but still with dynamic search disabled). Of these 160 instances, exactly 1 instance could be solved to optimality using any search strategy, and both CPLEX and CBFS were able to solve this problem. For this instance, CBFS was able to solve the problem with a search tree that was about half the size of the one needed by CPLEX, but the total computational running time required by CBFS was slightly larger.

Of the remaining 159 instances, some CBFS variant outperformed CPLEX and BFS in 58 instances, CPLEX or BFS outperformed all CBFS variants in 49 instances, and CBFS and CPLEX/BFS found incumbents with equal value for 32 instances. There were 20 instances for which no search strategy found an incumbent solution. In 6 out of the 58 cases that CBFS found a better incumbent, CPLEX and BFS failed to even find an incumbent. For the remainder, the average gap in solution value between CBFS and CPLEX or BFS was 9.3%, the minimum gap was 0.0025%, and the maximum gap was 180%. On the other

hand, the average gap in incumbent value for the 49 instance for which CPLEX/BFS outperform all CBFS variants was 19%, with a minimum gap of 0.0019% and a maximum gap of 630%.

For the 32 instances for which CBFS and CPLEX/BFS found the same incumbent value, in 7 cases the best solution was found at the root node of the B&B tree. For the remaining 25 instances, some CBFS variant found the best incumbent in fewer iterations than either CPLEX or BFS in 17 cases. In three of these cases, CBFS found the best incumbent at least an order of magnitude sooner than either CPLEX or BFS, and in one case a CBFS variant found the best incumbent nearly 3 orders of magnitude sooner. For the 8 instances in which CPLEX/BFS found the best incumbent in fewer iterations than any CBFS variant, only one was an order of magnitude improvement.

Analysis of Results

The computational results presented show that in many cases, using the CBFS search strategy will result in better performance when compared to the default search strategy used by CPLEX or to BFS. However, the data in Table 3.2 provide no clear indication of a contour labeling function which consistently outperforms the others. Labeling functions that favor positive assignments (for example, $\kappa_{1,0}$ and $\kappa_{1,-1}$) perform slightly better than others on this data set, but the trend is not especially strong. This suggests that good contour labeling functions may be instance-dependent. Thus, one focus of future research will be to identify characteristics of problem instances that lead to good labeling functions.

The presented computational results also validate the performance of CBFS as a heuristic; in particular, CBFS is able to find better incumbent solutions than either CPLEX or BFS in a majority of problem instances, as seen in Figure 3.6 and the tests performed on the MIPLIB challenge instances. However, this benefit is not so pronounced when considering total tree size or computational time. This suggests that while CBFS is able to find good incumbent solutions early, it may spend more time later in the search process exploring unnecessary regions of the search tree. Better contour labeling functions may improve this behavior; alternately, algorithms could use a hybrid search strategy that uses CBFS early in the process, and switch to BFS later in the search.

3.4.2 Application to the Simple Assembly Line Balancing Problem

A B&B solver using CBFS was developed and applied to instances of the simple assembly-line balancing problem (Section 2.6.4). The measure-of-best function used for this solver is a (non-admissible) heuristic function:

$$\mu(S) = I/m - 0.02 \cdot |U|,$$

where a subproblem S in the B&B tree is defined as $S = \{U, \sigma_1, \sigma_2, \ldots, \sigma_m\}$ and $U \subseteq J$ is the set of unassigned jobs for the subproblem. Recall that $\sigma_i \subseteq J \setminus U, i \in \{1, 2, \ldots, m\}$ is the sets of tasks assigned to station i. This measure-of-best function attempts to weight subproblems with a higher priority if they are more likely to lead to an optimal solution. In particular, the I/m term encourages the exploration of subproblems which have relatively low idle time per station. The remaining term acts as a tie-breaking function that encourages the exploration of subproblems with large numbers of remaining tasks, since these tasks are likely to be smaller and easier to schedule. The value of the parameter can be empirically chosen, and was selected to match the value in Sewell and Jacobson (2012).

As in Sewell and Jacobson (2012), the branching rule for the B&B solver is a wide branching rule called **station-oriented** branching; this method computes a number of possible **full loads** for the next available station. In particular, station σ_i is fully loaded if there are no tasks with satisfied precedence constraints that can be added to σ_i and satisfy the cycle time constraint at σ_i. A depth-first search mechanism is used to generate children at the current subproblem, in the following manner: each task which has all its precedence constraints satisfied at the current subproblem is considered for addition to the next station σ_{m+1}. For each such task j, depth-first search is used to enumerate all possible full loads for the next station such that it contains task j. Once a full load has been generated for σ_{m+1}, a new subproblem is generated and either pruned or inserted into the search tree. If the number of possible full loads at the current subproblem exceeds 10 000, no additional children are generated at that subproblem, and the algorithm proceeds heuristically.

For this problem, the depth contour labeling function κ_d is used to drive the behavior of the search strategy. The station-oriented branching rule tends to generate a reasonably-balanced search tree, and there is no obvious reason to favor one particular full load to σ_{m+1} over another, so in this setting the depth labeling function is the most natural.

Pruning rules for SALBP

The B&B solver for SALBP computes four different lower bounds and four dominance rules to prune the search tree. The lower bound rules are defined below (recall that t_j is the length of time needed to complete task j, and ξ is the cycle time for all machines; for a more detailed explanation of $LB1, LB2$, and $LB3$, see Scholl and Becker (2006) and Scholl and Klein (1997)):

$$LB1 = \left\lceil \sum_{j \in J} t_j / \xi \right\rceil, \quad LB2 = |\{j \in J \mid t > \xi/2\}| + \left\lceil \frac{|\{j \in T \mid t_j = \xi/2\}|}{2} \right\rceil, \quad LB3 = \left\lceil \sum_{j \in J} w_j \right\rceil,$$

where

$$w_j = \begin{cases} 1 & \text{if } t_j > 2\xi/3 \\ 2/3 & \text{if } t_j = 2\xi/3 \\ 1/2 & \text{if } \xi/3 < t_j < 2\xi/3 \\ 1/3 & \text{if } t_j = \xi/3. \end{cases}$$

At each subproblem in the branch-and-bound tree, the three lower bounds $LB1, LB2$, and $LB3$ are computed; if the maximum of these three values is greater than or equal to the value of the incumbent solution, then the subproblem can be pruned. On the other hand, if the three lower bounds above do not allow the subproblem to be pruned, a fourth lower bound, called *BPLB* (or **bin-packing lower bound**) is used to solve SALBP with the precedence constraints relaxed. As the bin-packing problem itself is NP-hard, a separate branch-and-bound solver is used to find good solutions for *BPLB*; if no good solutions can be found within 1 second of computation time, then the *BPLB* solver is terminated so that more progress can be made in the primary search tree.

Additionally, four different dominance rules are used to attempt to prune subproblems (see Sewell and Jacobson (2012) for more details):

- The Maximal Load Rule - If a subproblem contains a station load σ_i and an unassigned task j such that $\sigma_i \cup \{j\}$ does not violate the cycle time c or the precedence constraints, then that subproblem can be pruned.

- The Extended Jackson Rule - For a given subproblem, if the set of tasks assigned to the last station contains some task j, and there exists a task i such that (i, j) is not an edge of the precedence graph D, $t_i \geq t_j$, $\Phi_j \subseteq \Phi_i^*$, and task i can replace task j without violating the cycle time at the station or the precedence constraints, then the subproblem can be pruned.

- The No-Successors Rule - If the set of tasks assigned to the last station in a partial solution at some subproblem has no successors, and there exists an unassigned task which has at least one successor, then the current subproblem can be pruned.

- The Memory-based Dominance Rule - For this rule, it is necessary to store every subproblem that has been identified in the branch-and-bound tree in a hash table so that the rule can be checked efficiently. The rule states that if there exists a previously-identified subproblem in the search tree that has assigned all of the tasks as the current subproblem, and uses the same number or fewer stations, then the current subproblem can be pruned.

Whenever multiple dominance rules are used, it is important to ensure that there is no mutual dominance that could prevent the optimal solution from being found. First note that the memory-based dominance rule is only ever applied to two subproblems that are already in the search tree, and it only deletes subproblems with a strictly worse solution. Therefore, it is impossible for the memory-based rule to ever prune an optimal solution. Furthermore, no rule above ever yields a non-maximally-loaded station, so the first rule will never conflict.

The only remaining possible conflict is between the extended Jackson rule (EJR) and the no-successors rule (NSR). The following lemma establishes that these two rules can be used in concert.

Lemma 3.1. *Let S_1 and S_2 be two subproblems in the search tree for an instance of SALBP. If S_2 dominates S_1 via the EJR, and S_2 is dominated by the NSR, then S_1 is also dominated by the NSR.*

Proof. Suppose not. Let $S_1 = (U, \sigma_1, \sigma_2, \ldots, \sigma_{m-1}, \sigma_m)$ and $S_2 = (U', \sigma_1, \sigma_2, \ldots, \sigma_{m-1}, S'_m)$ (since S_1 and S_2 are related by the EJR, it must be the case that they each have m assigned stations, and the first $m-1$ are identical). Then, there must exist $j \in \sigma_m$ and $i \in U$ such that $\Phi_j \subseteq \Phi_i^*$ and $\sigma'_m = (\sigma_m - \{j\}) \cup \{i\}$. By the NSR, $\Phi_i^* = \varnothing$, which implies that $\Phi_j = \varnothing$ as well. All other tasks in σ'_m are identical to tasks in σ_m, which means that σ_m has no successors and can also be pruned by the NSR. ∎

Consider a set of subproblems in the search space that are all dominated by either the EJR or the NSR, and suppose that all optimal solutions to SALBP are descendants of some subproblem in this set. Then, it must be the case that some subproblem S_1 in the set is pruned by the EJR and not the NSR, and furthermore, the subproblem S_2 that dominates S_1 must also be in the set. In particular, there must exist a pair S_1 and S_2 for which S_1 is dominated only by the EJR and S_2 is dominated only by the NSR (otherwise, all subproblems in the set would be prunable by only a single dominance rule, and both the EJR and NSR have been proven correct independently). However, Lemma 3.1 implies that no such pair can exist. This proves the following corollary:

Corollary 3.1. *The EJR and NSR are compatible dominance rules for SALBP.*

Backtracking Rules

For instances that are particularly large, or for which each station can hold relatively few tasks, it is not practical to store the entire list of assigned and unassigned tasks, as well as the complete list of stations used in a partial solution at some subproblem in the search tree. In these settings, a backtracking method is incorporated that attempts to minimize memory usage within the branch-and-bound tree. This backtracking method was not incorporated in the algorithm described in Sewell and Jacobson (2012).

In this mode of operation, a subproblem in the branch-and-bound tree is represented by $S = \{p, \sigma_m\}$, where p is a pointer to the subproblem's parent, and σ_m is a list of tasks assigned to the current station. Since all subproblems must be stored in the tree for the memory-based dominance rule to be used, the complete partial solution represented by subproblem S can be reconstructed by following the parent pointers from S to the root of the search tree. Furthermore, the idle time and the hash value used to store the subproblem in the hash table for the memory-based dominance rule can be computed by tracing the parent pointers, and thus do not need to be stored at each subproblem.

The advantage of this method is that it substantially reduces the amount of memory used at a particular subproblem; this is most beneficial when the number of tasks assigned to any particular station is small compared to the total number of tasks. The principle disadvantage of this method is that it increases the computational time needed to process a subproblem. However, for the medium-sized instances in the database, it was observed that this method only increased the computation time needed to solve instances by about 30%.

SALBPGen Results

The B&B algorithm (in the sequel called `BBR`, or **branch, bound, and remember**—so named for the use of the memory-based dominance rule) for SALBP was implemented in C++, and run on all instances in the database generated by Otto et al. (2013) (see Section 2.6.4) using a single core of an Intel Core i7-930 2.8 GHz quad-core processor, with 12 GB of available memory. All running times reported are given in CPU seconds, and do not include the time needed to initialize the memory for the branch-and-bound tree, which is performed at the beginning of the algorithm. Each test was run with a time limit of one hour; the small, medium, and large instances each had a limit on the size of the search tree of 60 000 000 nodes, and the very large instances had an imposed limit of 80 000 000 search tree nodes, since the use of the backtracking code allows for more subproblems to be stored. The backtracking code was used for the very large problem instances; however, the bin-packing lower bound was disabled, due to its relative ineffectiveness and the large computation time for these problems. The results in this section are compared against the best results found by `Salome`, the best-performing previous algorithm for this problem (Scholl and Klein, 1997). `Salome` was run with relatively short time limits (20s, 50s, 70s, and 100s for the small, medium, large, and very large instances, respectively).

Table 3.3 summarizes the results for the runs against all problem instances. As shown, the `BBR` algorithm is able to solve all of the small- and medium-sized instances in the database, and all but 12 of the large instances. Finally, it is able to solve 350 of the very large instances. Additionally, `Salome` did not solve

any problem instances that were unsolved by `BBR`. Moreover, for the large instances, the `BBR` algorithm was able to improve upon the best-known upper bound in ten cases, leaving only two unsolved large instances which could not be solved to optimality or improved. Similarly, the `BBR` algorithm was able to improve the best-known upper bound for 149 very large instances, and it matches the best upper bound in five instances. However, for 21 of the very large instances, the solution found by `BBR` was worse than the solution found by `Salome`.

Table 3.3: Number of solved problem instances overall.

Size	`Salome` Solved	`BBR` Solved	`BBR` Improved	`BBR` Matched	`BBR` Worse
Small	521	525	0	0	0
Medium	4404	5250	0	0	0
Large	355	513	10	2	0
Very Large	186	350	149	5	21

A further analysis of the instances unsolved by `BBR` was performed, and the results are presented in Table 3.4. These results show that instances with lower order strength are often more challenging for `BBR` (45% have order strength of approximately 0.2, and 87% have order strength of less than 0.6); the graph structure has a less-clear relationship to instance difficulty. However, the most telling indicator of problem difficulty is the task time distribution: all 187 unsolved instances have a central distribution of task times.

Table 3.4: Problem statistics for the unsolved instances by `BBR`.

	Structure			Order Strength			Peak location		
Size	Bottleneck	Chain	None	0.2	0.6	0.9	bottom	central	bimodal
Large	7	4	1	10	2	0	0	12	0
Very large	50	50	75	75	75	25	0	175	0

3.5 Conclusion

The CBFS strategy is a relatively new search strategy that can be used with B&B algorithms to achieve better performance in many cases. In this chapter, the generality of the search strategy was established for the first time, showing that for any search strategy $\mathscr{A}$, a contour definition exists for CBFS that allows it to explore the same sequence of subproblems as $\mathscr{A}$. Moreover, a bound was proved on the number of subproblems for which CBFS generates children that shows in the worst case CBFS performs within a factor of $|\text{supp}(\mathscr{K})|$ of the performance of BFS. Finally, some properties of the contour labeling function are considered showing how it can be used to encourage or delay exploration of particular subproblems in a heuristic fashion.

Computational results are presented that show the effectiveness of a CBFS strategy for many instances in the MIPLIB 2010 mixed integer programming problem database. These performance gains appear to be instance-dependent; more work needs to be done to determine how the contour labeling function interacts with the problem instances so that better labeling functions can be developed. Moreover, it is believed that if the CBFS strategy were incorporated directly into CPLEX instead of through callbacks, better performance would result. Additional computational results are presented for the simple assembly line balancing problem, which, combined with the results from the MIPLIB challenge set, show the capabilities of CBFS for finding good incumbent solutions early in the search process for very large or very challenging problems.

Chapter 4

Solving the Pricing Problem using ZDDs

The CBFS strategy described in the preceding chapter is a general-purpose search strategy that can be used with many types of branching search processes. In contrast, the methods presented in this chapter and in Chapter 5 are specific to branch-and-price algorithms (described in Section 2.5). These two chapters describe general-purpose methods to overcome the challenges inherent in a B&B algorithm that uses column generation along with the standard integer branching rule. Specifically, recall that in order to solve the LP relaxation at a subproblem in a search tree using column generation and the standard integer branching rule, all branching decisions leading to that subproblem must also be imposed on the pricing problem, creating the constrained pricing problem. Moreover, in many cases due to the large number of problem variables and the asymmetry in positive versus null assignments, the resulting search tree can be quite unbalanced.

No algorithm in the literature has described a way to perform B&P without using techniques like alternative branching rules (Section 2.3.1) or the robust BCP method (Section 2.5), which come at the expense of ease of implementation and less-direct (global) solution methods. Alternate branching rules do not allow variables to be directly fixed to values, but rely on problem structure to implicitly fix variables. Similarly, the robust branch-and-cut-and-price methods require the solution of a larger LP at each subproblem, and again use implicit methods to fix variables.

Therefore, this chapter establishes an efficient method for solving the pricing problem in a generic B&P algorithm that is directly compatible with the standard integer branching scheme. The fundamental idea is to use a data structure called a **zero-suppressed binary decision diagram** (ZDD) to compactly store *all* valid solutions to the pricing problem. A linear-time algorithm is presented which adds restrictions to the ZDD to prohibit previously-generated columns from being produced a second time, allowing the constrained pricing problem to be solved at every iteration of column generation. This is combined with a contour labeling function for CBFS which reduces the imbalance in the search tree and biases the search towards complete solutions that are closer to the root of the tree. Computational results are presented for the graph coloring problem and the generalized assignment problem showing nearly order-of-magnitude improvements in solution time for some instances when using these two ideas, together with a proof of optimality for several

previously unsolved instances.

The remainder of this chapter is organized as follows: Section 4.1 defines the ZDD data structure and shows how it can be used to solve the pricing problem in a B&P algorithm. Section 4.2 then presents the ZDD restriction algorithm, which is used to modify the ZDD in place so as to produce solutions to the constrained pricing problem. Next, Section 4.3 presents two different algorithms for building a ZDD for the graph coloring pricing problem, and proves a bound on the size of such a ZDD. Section 4.4 presents computational results for the graph coloring problem and the generalized assignment problem. Finally, Section 4.5 outlines several future research directions for this technique.

4.1 Zero-Suppressed Binary Decision Diagrams

A zero-suppressed binary decision diagram (Minato, 1993) is an extension of the **binary decision diagram** (BDD) data structure proposed by Lee (1959) and Akers (1978). A BDD is a directed acyclic graph (DAG) that compactly encodes a binary function. Previously, BDDs have been used in circuit design and verification, as well as a number of formal logic applications (Bryant, 1992). More recently, BDDs have been used in a number of different optimization applications: Bergman et al. (2012) explore different variable orderings for BDDs used to characterize the independent sets of a graph, and Hadžić and Hooker (2008) add weights to the edges of a BDD to perform post-optimality analysis in a discrete optimization setting. Finally, Cire et al. (2012) and Bergman et al. (2013) describe how to use BDDs to compute upper and lower bounds to prune subproblems in a B&B algorithm.

Despite their success in these related areas, BDDs and ZDDs have not appeared in conjunction with B&P in the literature before. Behle and Eisenbrand (2007) give a method for using BDDs to enumerate vertices and facets of 0/1 polyhedra (which can be viewed as solving the pricing problem for a problem which has been reformulated via Dantzig-Wolfe decomposition), but they do not extend this result to the B&P setting. Additionally, Behle (2007) uses BDDs to generate valid inequalities in a branch-and-cut algorithm to perform row generation instead of column generation.

However, the use of decision diagrams together with B&P algorithms can provide substantial benefits to algorithm performance. This is because decision diagrams yield a way to compactly store all the columns even for an exponentially-sized integer program. Note that column generation techniques must still be used to solve the RMP, because the columns encoded in the ZDD cannot be operated on directly by the LP solver. Nonetheless, since the LP solver has (implicit) access to all columns, the pricing problem can be solved exactly at every iteration of column generation, which may improve the convergence of the column generation

procedure. In contrast, most B&P solvers terminate the pricing problem solver as soon as a column with "sufficiently negative" reduced cost is found, due to the difficulty of solving the pricing problem. Moreover, as shown in Section 4.2, the set of valid pricing problem solutions can be modified in place, allowing B&P algorithms using ZDDs to employ standard integer branching methods.

A ZDD is a modified version of a BDD that removes some nodes from the data structure to reduce its size. ZDDs are most useful when the binary function it encodes is "sparse" in the sense that there are relatively few valid solutions to the function compared to the number of invalid solutions. Minato (1993) observed that many combinatorial optimization problems have the sparsity characteristic; thus, ZDDs are likely to be more useful in a B&P setting than ordinary BDDs.

Formally, a ZDD Z is defined as follows. Let $\mathscr{E}$ be an ordered set of n elements $(e_1, e_2, \ldots, e_n)$; then Z is a DAG satisfying the following properties:

1. There are two special nodes in Z (denoted $\mathbf{1}$ and $\mathbf{0}$), called the **true** node and **false** node, respectively. Additionally, there is exactly one "highest" node in the topological ordering of Z, called the **root** of Z, and denoted z_{root}.

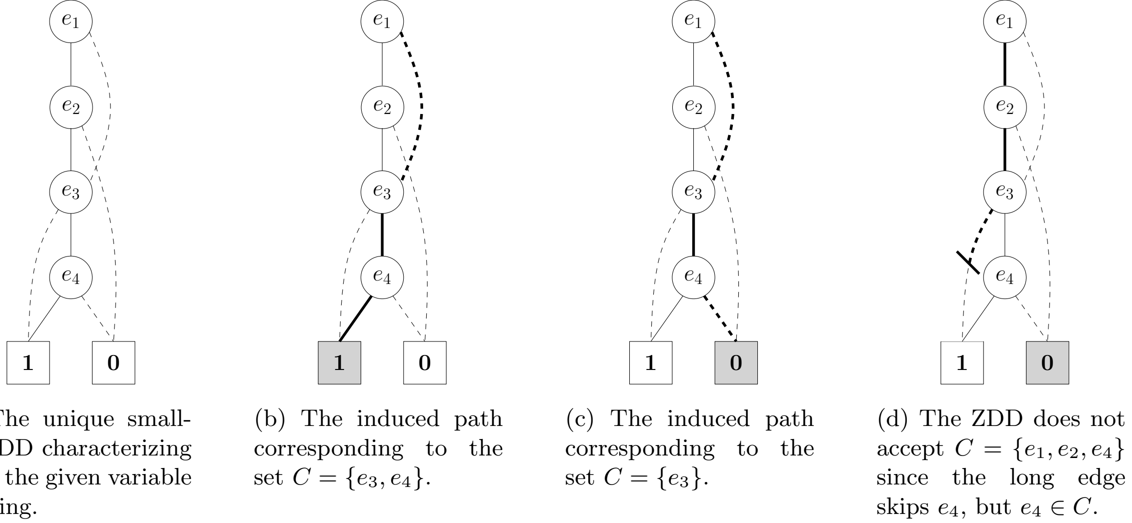
2. Every node $z \in Z - \{\mathbf{1}, \mathbf{0}\}$ has two outgoing edges, a **high edge** and a **low edge**, which point to the **high child** and **low child**, respectively. The high (low) child of z is denoted $\text{hi}(z)$ ($\text{lo}(z)$). The true and false nodes have no outgoing edges. The **indegree** of z, denoted $\delta^-(z)$, is the number of incoming edges to z; thus, $\delta^-(z_{root}) = 0$.

3. Every node $z \in Z - \{\mathbf{1}, \mathbf{0}\}$ is associated with some element $e_i \in \mathscr{E}$; the index of the associated element for z is given by $\text{var}(z)$, that is, $\text{var}(z) = i$. By convention, $\text{var}(\mathbf{1}) = \text{var}(\mathbf{0}) = n + 1$. Finally, if $\text{var}(z) = i$, then $\text{var}(\text{hi}(z)) > i$ and $\text{var}(\text{lo}(z)) > i$.

4. No $z \in Z$ has $\text{hi}(z) = \mathbf{0}$ (this property, called the **zero-suppressed** property, is not satisfied by ordinary BDDs).

Any set $C \subseteq \mathscr{E}$ induces a path P_C from the root of Z to either $\mathbf{1}$ or $\mathbf{0}$, in the following manner: starting at the root of Z, if z is the current node on the path, the next node along the path is $\text{hi}(z)$ if $e_{\text{var}(z)} \in C$, and $\text{lo}(z)$ otherwise. The **output** of Z on C, denoted $Z(C)$, is the last node along this path, which must be either $\mathbf{1}$ or $\mathbf{0}$. If $Z(C) = \mathbf{1}(\mathbf{0})$, then Z **accepts** (**rejects**) C. Note that it is not required for $\text{var}(z_2) = \text{var}(z_1) + 1$ when z_2 is a child of z_1; in the case when $\text{var}(z_2) > \text{var}(z_1) + 1$, the edge is called a **long edge**, and when an induced path P_C includes such an edge, if $\{e_{\text{var}(z_1)+1}, e_{\text{var}(z_1)+2}, \ldots, e_{\text{var}(z_2)-1}\} \cap C \neq \varnothing$, then Z rejects C. Finally, a ZDD **characterizes** a family of sets $\mathscr{C} \subseteq 2^{\mathscr{E}}$ (denoted $Z_{\mathscr{C}}$) if Z accepts all sets in $\mathscr{C}$, and rejects all sets not in $\mathscr{C}$ (see Figure 4.1).

Moreover, given a node $z \in Z_{\mathscr{C}}$, a **valid path** is a path from z to **1** in Z; such a path corresponds to a subset $\bar{C}$ of $C \in \mathscr{C}$ on the elements $\{e_{\text{var}(z)}, e_{\text{var}(z)+1}, \ldots, e_k\}$ where $e_j \in \bar{C}$ if and only if e_j is the tail of a high edge on the valid path. In this case, z **yields** C, denoted $z \vdash C$. Finally, define $\ell_i(Z_{\mathscr{C}}) = \{z \in Z_{\mathscr{C}} \mid \text{var}(z) = i\}$ to be the ith **level** of $Z_{\mathscr{C}}$.



(a) The unique smallest ZDD characterizing $\mathscr{C}$ for the given variable ordering.

(b) The induced path corresponding to the set $C = \{e_3, e_4\}$.

(c) The induced path corresponding to the set $C = \{e_3\}$.

(d) The ZDD does not accept $C = \{e_1, e_2, e_4\}$ since the long edge skips e_4, but $e_4 \in C$.

Figure 4.1: Let $\mathscr{E} = (e_1, e_2, e_3, e_4)$, and $\mathscr{C} = \{\varnothing, \{e_1, e_2\}, \{e_3, e_4\}, \{e_1, e_2, e_3, e_4\}\}$ (Example from Andersen, 1997). Solid lines represent high edges, and dashed lines represent low edges; all edges are directed downwards. Grey nodes indicate whether $Z_{\mathscr{C}}$ accepts C.

For an arbitrary family $\mathscr{C}$ and an arbitrary vertex ordering, the size of $Z_{\mathscr{C}}$ (that is, the number of nodes and edges in the graph, denoted $|Z_{\mathscr{C}}|$) may be exponential in n. However, Bryant (1986) shows that for any fixed variable ordering, every boolean function has a unique smallest BDD characterizing it. This result extends to ZDDs by observing that membership in $\mathscr{C}$ can be defined as a boolean function. One way to construct the unique smallest ZDD characterizing $\mathscr{C}$ is to first construct the BDD for $\mathscr{C}$'s indicator function, and then iteratively delete nodes whose high edge points to **0**, connecting the low edge to the node's parent. Alternately, there exists a recursive algorithm to construct $Z_{\mathscr{C}}$ directly (Knuth, 2008).

Note that the choice of ordering on the elements of $\mathscr{E}$ is important; Bryant (1986) shows examples where different variable orderings yield BDDs of dramatically different sizes for the same function. In fact, it is NP-hard to determine the variable ordering for any arbitrary boolean function that will yield the smallest BDD (Bollig and Wegener, 1996). These results apply for ZDDs as well; nevertheless, the use of heuristic variable orderings often results in tractably-sized ZDDs in practical applications.

To see how ZDDs can be used to solve the unconstrained pricing problem in a B&P algorithm, let $\mathscr{C}$ be the family of solutions to the pricing problem. Then, using the technique of Hadžić and Hooker (2008),

assign weights to the edges of $Z_{\mathscr{C}}$ and compute the longest path or shortest path in $Z_{\mathscr{C}}$ from the root to $\mathbf{1}$, depending on whether the pricing problem is a maximization or minimization problem. Specifically, let $(\pi_1, \pi_2, \ldots, \pi_n)$ be a weight vector for the elements of $\mathscr{E}$; set the weight of edge $(z_1, z_2) \in Z_{\mathscr{C}}$ to $\pi_{\text{var}(z)}$ if $z_2 = \text{hi}(z_1)$, and 0 otherwise. Then, finding the longest or shortest path with respect to $\{\pi\}$ from the root of $Z_{\mathscr{C}}$ to $\mathbf{1}$ can be found in $O(|Z_{\mathscr{C}}|)$ time using dynamic programming (Sedgewick and Wayne, 2011). The resulting path corresponds to the optimal solution to the pricing problem (see Figure 4.2).

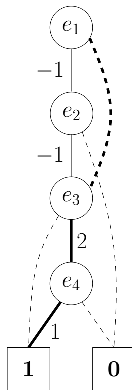


Figure 4.2: The ZDD from Figure 4.1a with weights given by the objective function $\max\,[-e_1 - e_2 + 2e_3 + e_4]$; the bold path corresponds to the maximum-weight valid set, that is $\{e_3, e_4\}$. Weights not shown are 0.

4.2 The ZDD Restriction Algorithm

In order to use standard integer branching methods in a B&P algorithm it is necessary to solve the constrained pricing problem. Recall that this problem seeks a new variable of minimum reduced cost that respects all branching decisions made at the subproblem. Note that it is sufficient to generate a new variable that does not appear in the pool $\mathscr{C}'$ for the RMP; to see this, observe that if any variable in $\mathscr{C}'$ has negative reduced cost, then the current solution to the RMP is not optimal. Therefore, in this section, an algorithm is presented to add restrictions to a ZDD characterizing the pricing problem so that any time a new column is generated and added to $\mathscr{C}'$, it can be immediately restricted from ever being generated as a solution to the pricing problem again. In this way, the ZDD will actually solve the constrained pricing problem at each iteration of the algorithm.

Let $\mathscr{C}$ be the family of valid solutions to the pricing problem, where each $C \in \mathscr{C}$ is a subset of $\mathscr{E} = (e_1, e_2, \ldots, e_n)$, and let $Z_{\mathscr{C}}$ be the ZDD characterizing $\mathscr{C}$. The restriction algorithm for ZDDs, called `RestrictSet`, takes as input a set $C \in \mathscr{C}$, and builds a new ZDD $Z_{\mathscr{C}^-}$ that accepts $\mathscr{C}^- = \mathscr{C} \setminus \{C\}$. The

key feature of the ZDD restriction algorithm that makes it effective in practice is that it operates in $O(n)$ time, and it increases the size of $Z_{\mathscr{C}}$ by at most n nodes and $2n$ edges (and often by much less).

Intuitively, the `RestrictSet` algorithm identifies the path P_C in $Z_{\mathscr{C}}$ corresponding to the set C, and updates this path so that it ends at the false node instead of the true node. However, if there exists $C' \neq C$ such that P_C and $P_{C'}$ overlap, this update could also restrict C'. Therefore, `RestrictSet` duplicates the portion of P_C that could overlap with some other root-to-**1** path, and sets the endpoint of the duplicate path to **0**. This ensures that no additional sets are restricted by the algorithm. The first node on P_C with indegree greater than one is the first node with some potential overlap; thus it, and all subsequent nodes, are duplicated.

Pseudocode for the `RestrictSet` algorithm is given in Algorithm 4.1; this algorithm makes use of a function called $Z_{\mathscr{C}}$.`insert`(i, z_ℓ, z_h), which takes as input an index $i \in \{1, 2, \ldots, n\}$ and pointers to two pre-existing nodes $z_\ell, z_h \in Z_{\mathscr{C}}$. The function inserts a new node into $Z_{\mathscr{C}}$ associated with element e_i, with low edge pointing to z_ℓ and high edge pointing to z_h, and returns a pointer to the newly-inserted node. It also updates the indegrees of the high and low children. $Z_{\mathscr{C}}$.`insert` can be implemented in (average) constant time (see Andersen, 1997 for details). The function $1_C : \mathscr{E} \to \{0, 1\}$ is an indicator function for C, i.e. $1_C(e_i) = 1$ if and only if $e_i \in C$.

The following theorem establishes the correctness of the `RestrictSet` algorithm and proves the claims made previously about its time and space complexity behavior; an example of the `RestrictSet` applied to the ZDD in Figure 4.1a is given in Figure 4.3.

Theorem 4.1. *Given a ZDD $Z_{\mathscr{C}}$ describing a family of subsets $\mathscr{C}$ of an ordered set $\mathscr{E}$ with n elements, together with a set $C \in \mathscr{C}$, the* `RestrictSet` *algorithm modifies $Z_{\mathscr{C}}$ in $O(n)$ time to produce a new ZDD called $Z_{\mathscr{C}^-}$ characterizing $\mathscr{C}^- = \mathscr{C} \setminus \{C\}$. Furthermore, $|Z_{\mathscr{C}^-}| \leq |Z_{\mathscr{C}}| + 3n$.*

Proof. First, note that `RestrictSet` visits each node along P_C at most twice, and P_C has at most n nodes. Furthermore, the algorithm performs a constant amount of work for each visited node. Thus the running time of `RestrictSet` is $O(n)$. Also, since node `parent` is at most the root of $Z_{\mathscr{C}}$, at most n nodes are added to $Z_{\mathscr{C}}$ to form $Z_{\mathscr{C}^-}$, and each new node has two outgoing edges.

To prove that $Z_{\mathscr{C}^-}$ has the desired properties, let $z'_1, z'_2, \ldots, z'_k$ be the nodes added to $Z_{\mathscr{C}}$ in Lines 18-19 (Algorithm 4.1), in increasing order of depth. Consider some set $C' \subseteq \mathscr{E}$; if $C' = C$, the path from the root of $Z_{\mathscr{C}^-}$ to the bottom of the ZDD is the same as the path from the root of $Z_{\mathscr{C}}$ up to the `parent` node. By construction, the next node visited in $Z_{\mathscr{C}^-}$ is z'_1 (Lines 21 and 22, Algorithm 4.1). Then, the remainder of the path in $Z_{\mathscr{C}^-}$ follows the added nodes; at each z'_i, the high and low children are constructed to agree with the values of C. Finally, the last node along this path is **0** (Line 15, Algorithm 4.1), so $Z_{\mathscr{C}^-}(C') = 0$.

Algorithm 4.1: RestrictSet($Z_{\mathscr{C}}, C$)

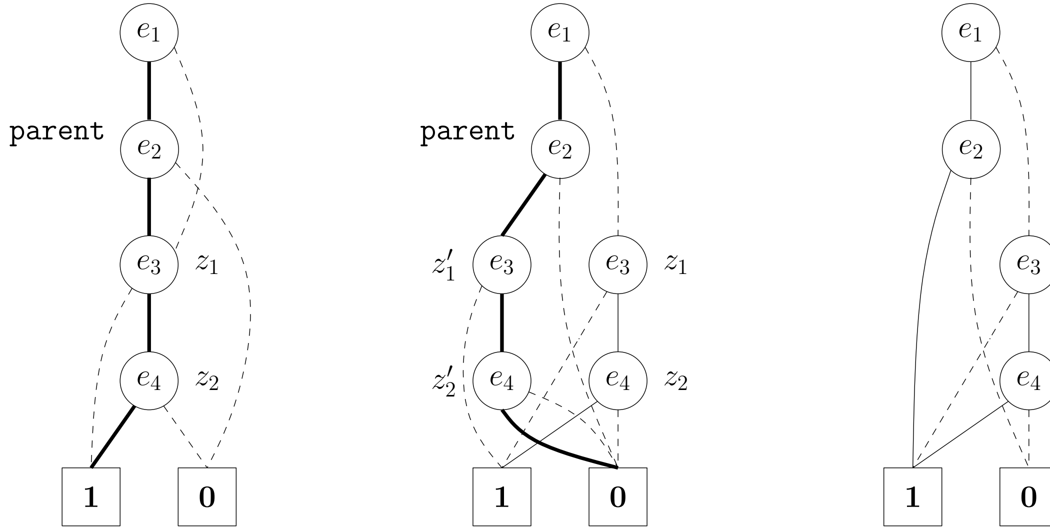
input: A ZDD $Z_{\mathscr{C}}$ and a set $C \in \mathscr{C}$
output: A modified ZDD $Z_{\mathscr{C}^-}$ such that $\mathscr{C}^- = \mathscr{C} \setminus \{C\}$

1 ⟪ *Find the first node on* P_C *with indegree higher than* 1 ⟫
2 `current` $= z_{root}$; `parent` $= -1$
3 **while** $\delta^-($`current`$) < 2$ and `current` $\notin \{\mathbf{1}, \mathbf{0}\}$**:**
4 $\quad i = \text{var}($`current`$)$
5 $\quad$ `parent` = `current`
6 $\quad$ **if** $1_C(e_i) == 1$**:** `current` = hi(`current`)
7 $\quad$ **else:** `current` = lo(`current`)
8 ⟪ *Make copies of all remaining nodes on* P_C *and point to* **0** ⟫
9 list = ()
10 **while** `current` $\notin \{\mathbf{1}, \mathbf{0}\}$**:**
11 $\quad i = \text{var}($`current`$)$
12 $\quad$ list.`append`(`current`)
13 $\quad$ **if** $1_C(e_i) == 1$**:** `current` = hi(`current`)
14 $\quad$ **else:** `current` = lo(`current`)
15 `new` = **0**
16 ⟪ *Insert the duplicated nodes into* $Z_{\mathscr{C}}$ ⟫
17 **for each** $z \in$ list (in reverse order)**:**
18 $\quad$ **if** $1_C(e_{\text{var}(z)}) == 1$**:** `new` = $Z_{\mathscr{C}}$.`insert`(var(z), lo(z), `new`)
19 $\quad$ **else:** `new` = $Z_{\mathscr{C}}$.`insert`(var(z), `new`, hi(z))
20 ⟪ *Point the correct edge of the parent node to the root of the duplicated path* ⟫
21 **if** $1_C(e_{\text{var(parent)}}) == 1$**:** hi(`parent`) = `new`
22 **else:** lo(`parent`) = `new`
23 return $Z_{\mathscr{C}}$

Furthermore, if $C' \neq C$, then consider the first index j where the characteristic vectors of C and C' differ; if $j \leq \text{var}($`parent`$)$, then the modifications to $Z_{\mathscr{C}^-}$ have no effect on whether C' is accepted, since the only nodes added to $Z_{\mathscr{C}^-}$ appear at depths greater than the `parent` node. However, if $j > \text{var}($`parent`$)$, the path will follow along the newly added nodes $z'_1, z'_2, \ldots, z'_i$, where $\text{var}(z'_i) = j$. At this point, C and C' differ, and by construction, the path returns to the original node in $Z_{\mathscr{C}}$ and never returns to the newly-added nodes. Therefore, $Z_{\mathscr{C}^-}(C') = Z_{\mathscr{C}}(C')$, as desired. ■

To reduce the size of $Z_{\mathscr{C}^-}$, a check can be performed to see if the high and low edges of newly-inserted nodes both point to **0**; in this case, the node is suppressed (see Figure 4.3c). Finally, note that the ZDD produced by the `RestrictSet` is no longer necessarily minimal with respect to $\mathscr{C}^-$. In particular, in the worst case, if all 2^n subsets of $\mathscr{E}$ are restricted, the size of the ZDD can grow by $O(n2^n)$ nodes. However, in this case, the resultant ZDD is $Z_\varnothing$, which can be described with only two nodes. In the event that the ZDD becomes too large, a reduction algorithm can be periodically called that searches for duplicate nodes in $Z_{\mathscr{C}}$ that can be merged.

Using the `RestrictSet` procedure, a B&P algorithm can be developed that uses traditional integer



(a) The path P_C is in bold; the path is split at the first node with indegree larger than 1. The `parent` node is its immediate predecessor on the path.

(b) Copies of nodes z_1 and z_2 are created, and the high edge from `parent` points to this new path. The new path points to **0**, thus restricting the set C.

(c) The new ZDD $Z_{\mathscr{C}^-}$; since both high and low edges of z_2' point to **0**, it can be suppressed. z_1' is also suppressed to satisfy the zero-suppressed property.

Figure 4.3: The result of applying the `RestrictSet` algorithm to $Z_{\mathscr{C}}$ from Figure 4.1a with $C = \{e_1, e_2, e_3, e_4\}$. The final ZDD accepts $\mathscr{C}^- = \{\varnothing, \{e_1, e_2\}, \{e_3, e_4\}\}$.

branching. This B&P algorithm first builds a ZDD characterizing all valid solutions to the pricing problem; in the worst case, this may take exponential time, but dynamic programming or memoization techniques can be used to speed up the construction. The ZDD is then used to produce new variables at every iteration of column generation, which correspond to solutions of the constrained pricing problem. Once a new set (or variable) has been generated, `RestrictSet` is called to prohibit that column from being generated again at a later time. The ZDD is therefore guaranteed to produce the optimal solution to the pricing problem at each stage, and since in most cases $n \ll |Z_{\mathscr{C}}|$, the increase in size of the ZDD over the course of the B&P search is small. Hence, the time needed to solve the pricing problem does not significantly increase over the course of the algorithm. Pseudocode for the resulting B&P search is given in Algorithm 4.2.

4.3 Constructing the Maximal Independent Set ZDD

In this section, specific properties of a ZDD characterizing the family of maximal independent sets of an undirected graph G are analyzed. As discussed in Section 2.6.2, the maximal independent set problem is the pricing problem for the graph coloring problem. Thus, a ZDD characterizing all such sets can be used to build a B&P solver for graph coloring, as described in Section 4.2. In this setting, the vertex set V plays the

Algorithm 4.2: B&P+ZDD(X, f)

```
1  Set 𝒮 = {X}
2  Construct Z_𝒞, where 𝒞 is the set of valid columns
3  Initialize x̂ and the initial RMP pool 𝒞′
4  for each C ∈ 𝒞′: Z_𝒞 = RestrictSet(Z_𝒞, C)
5  while 𝒮 ≠ ∅:
6      Select a subproblem S ∈ 𝒮 to explore
7      if a solution x̂′ ∈ {x ∈ S | f(x) < f(x̂)} can be found: Set x̂ = x̂′
8      if S cannot be pruned:
9          Partition S into S_1, S_2, ..., S_r
10         for each S_i ∈ {S_1, S_2, ..., S_r}:
11             ⟪ Column generation loop; use Z_𝒞 to search for C ⟫
12             while ∃ C ∈ 𝒞 \ 𝒞′ with negative reduced cost at S_i:
13                 RestrictSet(Z_𝒞, C)
14                 Add C to 𝒞′
15             Compute a lower bound at S_i using added columns
16         Insert S_1, S_2, ..., S_r into 𝒮
17     Remove S from 𝒮
18 Return x̂
```

role of $\mathscr{E}$, and the family $\mathscr{C}$ is the family of all maximal independent sets of G. In a slight abuse of notation, say that a vertex $v \in V$ is **dominated** by a valid path from z corresponding to a (not-necessarily-maximal) independent set $\bar{C}$ if $v \in N[\bar{C}]$ (recall that a vertex in G is dominated by a set C if $v \in N[C]$).

4.3.1 Recursive ZDD Construction

A natural recursive algorithm following the general approach given in Knuth (2008) can be used to construct the ZDD characterizing the maximal independent sets of G. This algorithm stores each node $z \in Z_{\mathscr{C}}$ in a hash table which can be searched in constant time. If there exists a node $z \in Z_{\mathscr{C}}$ with the same associated vertex, high child, and low child as some node z', z and z' can be merged, with the parent of z' adjusted to point to z. In this case, z and z' are equivalent, denoted $z \cong z'$. Pseudocode for this algorithm is given in Algorithm 4.3.

The algorithm takes as input a set of k undominated vertices U, and a current index $i \leq k+1$, and it behaves as follows: first, two base cases are checked (Lines 1 and 2, Algorithm 4.3). In the first case, some vertex in $\{u_1, u_2, \ldots, u_{i-1}\}$ is not adjacent to any vertex in $\{u_i, u_{i+1}, \ldots, u_k\}$. In this case, there is no way to construct a maximal independent set with the remaining vertices. Alternately, if there are no undominated vertices, the set is maximal by definition. If neither base case is met, two branches are constructed. The high branch is constructed by removing the current vertex u_i and all its neighbors from the set of undominated vertices, and advancing the current index to the next undominated vertex in U (if no such vertex exists,

the current index is set to an "off-the-end" value of $k+1$). Similarly, the low branch is constructed by advancing the current index to the next undominated vertex in U, if one exists. Once the high and low children have been computed (recursively constructing them if necessary), Line 7, Algorithm 4.3 enforces the zero-suppressed condition; Line 9, Algorithm 4.3 checks for the existence of a node with the same label, high child, and low child, and Line 10, Algorithm 4.3 inserts a new node into the ZDD if it does not exist. To construct the complete maximal independent set for a graph G, `MakeIndSetZDD` (V, 1) is called. An example graph is shown in Figure 4.4, and the steps showing the application of `MakeIndSetZDD` to this example are given in Figure 4.5.

Algorithm 4.3: MakeIndSetZDD(U, i)

input: A set $U = \{u_1, u_2, \ldots, u_k\}$ of undominated vertices such that $u_j < u_{j+1}$ with respect to the vertex ordering on V, and a "current index" i

output: The root node of a ZDD characterizing all the maximal independent sets in $G[U]$ that can be formed with vertices in $\{u_i, u_{i+1}, \ldots, u_k\}$

1 **if** $G[U]$ has a vertex with no neighbor in $\{u_i, u_{i+1}, \ldots, u_k\}$**:** return **0**
2 **if** $U == \varnothing$**:** return **1**
3 $U_H = U \setminus N[u_i]$ ⟪ *Use vertex u_i; remove it and its neighbors from U* ⟫
4 $h = \min\{j \mid j > i$ and $u_j \in U_H\}$ or $|U_H| + 1$ if no j exists
5 $z_h =$ `MakeIndSetZDD`(U_H, h)
6 $z_l =$ `MakeIndSetZDD`$(U, i+1)$
7 **if** $z_h ==$ **0:** return z_l
8 ⟪ *Look up element in reverse hash table* ⟫
9 **if** $\exists\, z \in Z_{\mathscr{C}}$ s.t. $\text{var}(z) = i, \text{lo}(z) = z_l,$ and $\text{hi}(z) = z_h$**:** return z
10 **else:** insert a new node z' into $Z_{\mathscr{C}}$, and return z'

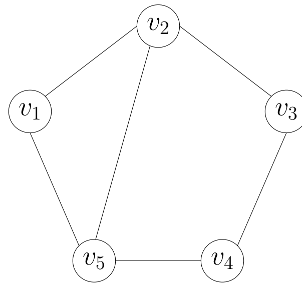
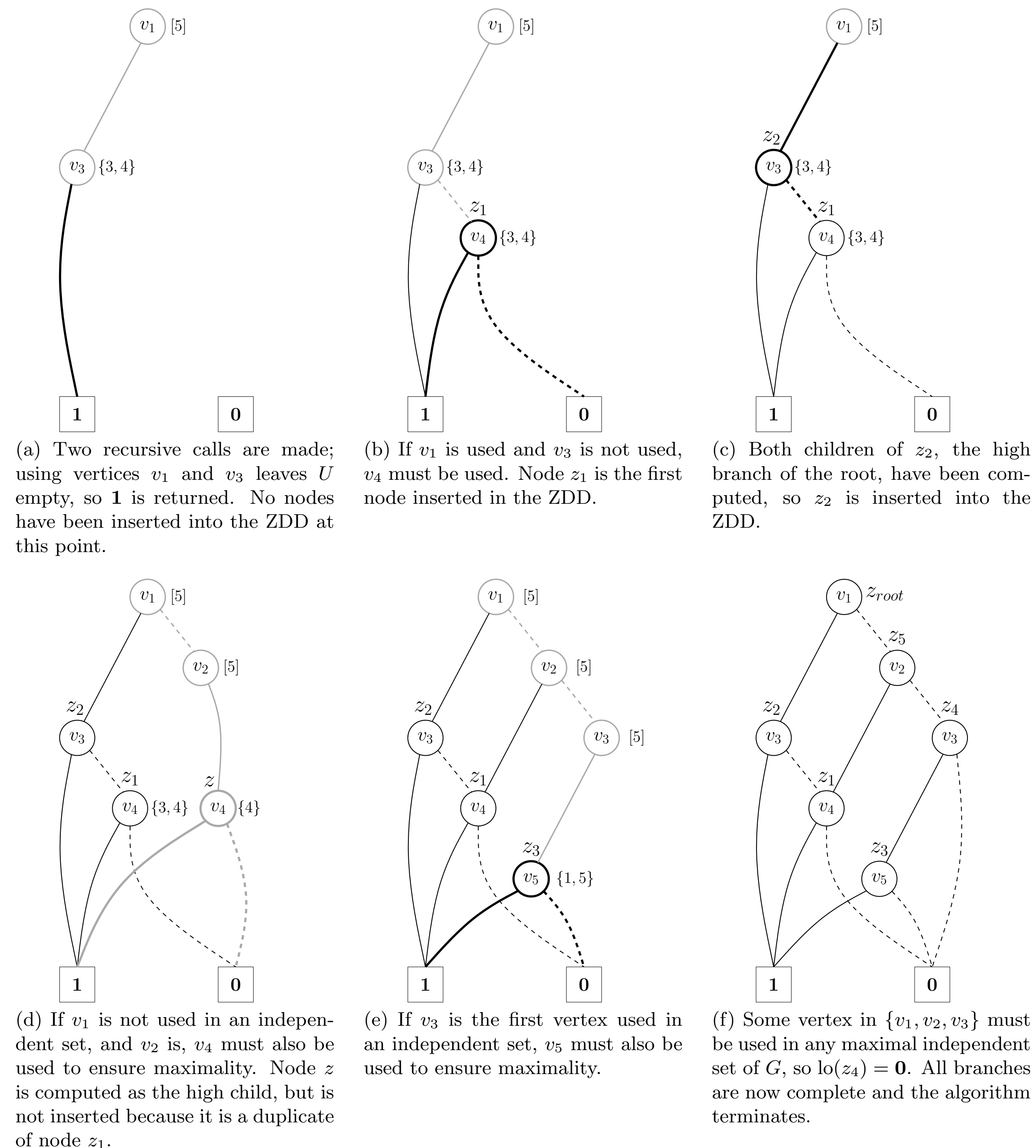


Figure 4.4: An example graph with a vertex ordering.

Note that `MakeIndSetZDD` does not actually maintain the vertices that are used in a current independent set C during the ZDD construction. It is sufficient to maintain a list of vertices that are left undominated by *some* independent set, since many independent sets may yield the same set of undominated vertices. The theoretical running time of Algorithm 4.3 is $O(n|\overline{Z_{\mathscr{C}}}|)$, where $\overline{Z_{\mathscr{C}}}$ is the size of the ZDD produced if the merge step (Line 9, Algorithm 4.3) is not present. A crude upper bound on $\overline{Z_{\mathscr{C}}}$ is 2^n; the results

Figure 4.5: A visualization of the steps taken by `MakeIndSetZDD` to build $Z_{\mathscr{C}}$ for the example graph given in Figure 4.4. Grey nodes and edges have been visited by a recursive call, but are not yet stored in the ZDD. Black nodes and edges have been stored in the ZDD's lookup table. Bold elements have been inserted in the most recent step of the algorithm; nodes are indexed in order of insertion. The set listed to the right of each node is the set U for that recursive call; the notation $[n]$ denotes the set $\{1, 2, \ldots, n\}$.



(a) Two recursive calls are made; using vertices v_1 and v_3 leaves U empty, so **1** is returned. No nodes have been inserted into the ZDD at this point.

(b) If v_1 is used and v_3 is not used, v_4 must be used. Node z_1 is the first node inserted in the ZDD.

(c) Both children of z_2, the high branch of the root, have been computed, so z_2 is inserted into the ZDD.

(d) If v_1 is not used in an independent set, and v_2 is, v_4 must also be used to ensure maximality. Node z is computed as the high child, but is not inserted because it is a duplicate of node z_1.

(e) If v_3 is the first vertex used in an independent set, v_5 must also be used to ensure maximality.

(f) Some vertex in $\{v_1, v_2, v_3\}$ must be used in any maximal independent set of G, so $\mathrm{lo}(z_4) = \mathbf{0}$. All branches are now complete and the algorithm terminates.

in Section 4.3.3 will improve this bound slightly, and in many practical cases $\overline{Z_{\mathscr{C}}}$ is much smaller. In comparison, the independent-set enumeration algorithm of Bron and Kerbosch (1973) runs in $O(3^{n/3})$ worst-case time (Tomita et al., 2006). The advantage gained by `MakeIndSetZDD` is the ability to store all maximal independent sets for reuse, which cannot be done by a simple enumeration algorithm.

4.3.2 Dynamic Programming ZDD Construction

An alternative construction algorithm for $Z_{\mathscr{C}}$ based on dynamic programming can also be formulated. To describe this algorithm, necessary and sufficient conditions for nodes in $Z_{\mathscr{C}}$ are shown that do not rely on a hash table to detect equivalent nodes. In particular, this condition allows for a non-recursive top-down construction of the smallest $Z_{\mathscr{C}}$ for some graph G and a fixed variable ordering. The resulting equivalence conditions are similar to those in Theorem 1 from Bergman et al. (2012), but are somewhat more complicated due to the maximality requirement for sets in $\mathscr{C}$. Specifically, the independent set BDD described in Bergman et al. (2012) stores a single set of vertices E, called the **eligible set**, at every node in the BDD; it is shown that two nodes in the BDD are equivalent if they have the same eligible set (to avoid confusion with the edge set of the graph, this dissertation denotes the eligibility set by L). This idea is extended for the maximal independent set ZDD; here, however, two sets are needed, the **totally dominated set** and the **reduced eligibility set**. The main result is to show that two nodes in $Z_{\mathscr{C}}$ are equivalent if and only if their totally dominated sets and reduced eligibility sets are identical. First, a few definitions are needed. For the remainder of this section, let $z \in Z_{\mathscr{C}}$ with $\text{var}(z) = i$.

Definition 4.1. *Consider an independent set $\bar{C} \subseteq \{v_i, v_{i+1}, \ldots, v_n\}$ such that $z \vdash \bar{C}$. The* ***i*-dominated set** *for $\bar{C}$, denoted $\Gamma_i(\bar{C})$, is $N(\bar{C}) \cap \{v_1, v_2, \ldots, v_{i-1}\}$, that is, the neighbors of $\bar{C}$ smaller than v_i.*

Definition 4.2. *The* **totally dominated set** *of vertices at z, denoted τ_z, is the subset of $\{v_1, v_2, \ldots, v_{i-1}\}$ such that every valid path from z dominates all vertices in τ_z. In other words,*

$$\tau_z = \bigcap_{\bar{C}: z \vdash \bar{C}} \Gamma_i(\bar{C}).$$

For example, in Figure 4.5f, $\tau_{z_1} = \{v_3\}$, $\tau_{z_2} = \{v_2\}$, and $\tau_{z_3} = \{v_1, v_2, v_4\}$. Note that, given some root-to-z path, it is not necessary for τ_z to actually be undominated at z; the definition of totally dominated only requires that all valid paths starting from z dominate v. However, any undominated nodes along such a root-to-z path must belong to τ_z.

Additionally, note that for nodes $z, z' \in Z_{\mathscr{C}}$, if there exists a vertex $v \in \tau_{z'}$ and $v \notin \tau_z$, then v is undominated by at least one valid path from z, but is dominated by all valid paths from z'. This proves the

following lemma:

Lemma 4.1. *Any two equivalent nodes $z, z' \in Z_{\mathscr{C}}$ must satisfy $\tau_z = \tau_{z'}$.*

The next definitions describe the other set needed to determine node equivalence in $Z_{\mathscr{C}}$.

Definition 4.3. *Define an* **eligibility set** *at z to be $L_z \subseteq \{v_i, v_{i+1}, \ldots, v_n\}$, where $v_j \in L_z$ if $v_j \geq v_i$ is available for inclusion in an independent set, given some root-to-z path.*

Note that the eligibility set at z is not necessarily unique. Depending on the path taken from the root to z, some vertices could be undominated at z, but not usable in any valid path from z. Such a vertex is called an **eligible impostor**. For example, suppose that there exists a vertex $v \in L_z$ such that $N(v) = L_z \setminus v$. Additionally, suppose that at z, some vertex $u \in \tau_z$ with $u \not\leftrightarrow v$ has not been dominated. Then, any path from z that uses v cannot dominate u, and thus cannot be a valid path. In this case, v is an eligible impostor. The reduced eligibility set at z removes all such impostors.

Definition 4.4. *The* **reduced eligibility set** *at z, denoted $\overline{L}_z$, is the set of vertices which are used by at least one valid path from z. Equivalently, $\overline{L}_z = L_z \setminus R$, where R is the set of eligible impostors with respect to some path from the root of $Z_{\mathscr{C}}$ to z.*

The following theorem gives a necessary and sufficient condition for when two nodes of a maximal independent set ZDD are equivalent and can be merged:

Theorem 4.2. *Nodes $z, z' \in Z_{\mathscr{C}}$ are equivalent if and only if $\tau_z = \tau_{z'}$ and $\overline{L}_z = \overline{L}_{z'}$.*

Proof. ($\Rightarrow$) Suppose $z' \cong z$; the requirement that $\tau_z = \tau_{z'}$ is established by Lemma 4.1. Furthermore, every valid path from z must exactly correspond to a valid path from z', which means that every vertex used on a valid path from z must be eligible at z', and vice versa. The only eligible vertices present at z that do not need to be present at z' are those vertices which are used in no valid paths from z—that is, the eligible impostors. Thus, it must be the case that $\overline{L}_z = \overline{L}_{z'}$.

($\Leftarrow$) Suppose $\tau_z = \tau_{z'}$ and $\overline{L}_z = \overline{L}_{z'}$ for $z, z' \in Z_{\mathscr{C}}$. It suffices to show that valid paths from z are in bijection with valid paths from z'. Note that in any root-to-z path, the undominated vertices at z are a subset of τ_z, and similarly for z'. Therefore, since $\tau_z = \tau_{z'}$, any valid path from z (z') to **1** must dominate all undominated vertices at z (z'). Additionally, since $\overline{L}_z = \overline{L}_{z'}$, all valid paths from z are also valid paths from z', and vice versa. Thus, there is a bijection between valid paths from z and valid paths from z', proving the result. ∎

Theorem 4.2 provides a method for building $Z_{\mathscr{C}}$ for a particular graph by computing τ_z and $\overline{L}_z$ for every node in the graph. Pseudocode for algorithms to compute τ_z and $\overline{L}_z$ at a node z is given in Algorithms 4.4

Algorithm 4.4: ComputeTotalDomination(U, i)

input: A set of undominated vertices $U = \{u_1, u_2, \ldots, u_k\}$, and an index i
output: The totally dominated set given U and i

1 ⟪ *The only nodes that could be in τ_z are the neighbors of eligible vertices* ⟫
2 $L_z = \{u_i, u_{i+1}, \ldots, u_k\}$
3 $\tau_z = \left(\bigcup_i^k N(u_i)\right) \setminus L_z$
4 **for each** $v \in \tau_z$ **:**
5 Search for an independent set with vertices from $L_z \setminus N(v)$ dominating all of L_z
6 **if** such a set exists**:** remove v from τ_z
7 return τ_z

and 4.5, respectively. The first algorithm is motivated by the observation that if a vertex v is totally dominated at z, no maximal independent set can be found using vertices in $L_z \setminus N(v)$. Thus, Algorithm 4.4 searches for such an independent set for each v, and removes v from τ_z if one can be found. The second algorithm operates similarly; it takes as input a list of undominated vertices at z, and checks each of them to see if they are contained in some maximal independent set from z. If no such set is found, the vertex is marked as an eligible impostor. Both of these algorithms use depth-first search to find independent sets.

Note that a few enhancements to Algorithm 4.4 can be made; namely, all of $\{u_1, u_2, \ldots, u_{i-1}\}$ must be dominated in a valid path from z, and thus, do not have to be checked in Line 4. Additionally, if the reduced eligibility set $\overline{L}_z$ is known at the start, it can be used in place of L_z. Finally, note that multiple vertices can be checked at once—in particular, if an independent set is found in one iteration of the loop that leaves multiple vertices from τ_z undominated, all of them can be removed from τ_z and not considered further. The worst-case running times for Algorithms 4.4 and 4.5 are $O(k2^k)$, but these bounds are often quite weak.

Algorithm 4.5: ComputeReducedEligibility(U, i)

input: A set of undominated vertices $U = \{u_1, u_2, \ldots, u_k\}$, and an index i
output: The reduced eligibility set given U and i

1 $L_z = \{u_i, u_{i+1}, \ldots, u_k\}$
2 $\overline{L}_z = \varnothing$
3 **for each** $u_j \in \{u_i, u_{i+1}, \ldots, u_k\}$**:**
4 Search for an independent set dominating U using u_j and vertices from $L_z \setminus N[u_j]$
5 **if** such a set exists**:** add u_j to $\overline{L}_z$
6 return $\overline{L}_z$

The maximal independent set ZDD can now be constructed. Whenever a node z is inserted into $Z_{\mathscr{C}}$, store τ_z and $\overline{L}_z$ for that node. Then, when a new node is being considered for insertion, compute the totally dominated set and the reduced eligibility set for the node, and compare it to all nodes at the current level in the ZDD. If one is found that satisfies the equivalence conditions, merge the new node into the current node. Algorithm 4.6 gives a top-down construction routine for $Z_{\mathscr{C}}$; in this algorithm, U_z is the set of undominated

vertices at a node z.

To store the totally dominated sets and reduced eligibility sets, note that for any node z, all $v_i \in \tau_z$ have $i < \text{var}(z)$ and all $v_i \in \overline{L}_z$ satisfy $i \geq \text{var}(z)$. Therefore, τ_z and $\overline{L}_z$ can be stored in a single boolean array, where all true entries in the array with index less than var(z) are members of τ_z, and similarly for $\overline{L}_z$.

Algorithm 4.6: MakeIndSetZDD+DP($G = (V, E)$)

output: The root node of $Z_{\mathscr{C}}$
1 Insert the root of $Z_{\mathscr{C}}$ into $\ell_1(Z_{\mathscr{C}})$, with $U_{\text{root}} = V$
2 **for each** $i \in \{1, 2, \ldots, n\}$**:**
3   **for each** $z \in \ell_i(Z_{\mathscr{C}})$**:**
4         $U_H = U_z \setminus N[v_i]$
5         $h = \min\{j \mid j > i \text{ and } v_j \in U_H\}$
6         $\overline{L}_{z'} =$ `ComputeReducedEligibility`(U_H, h)
7         $\tau_{z'} =$ `ComputeTotalDomination`(U_H, h)
8   **if** $\exists$ node in $\ell_h(Z_{\mathscr{C}})$ with $\tau_z = \tau_{z'}$ and $\overline{L}_z = \overline{L}_{z'}$**:** merge z and z'
9   **else:** insert z' into $\ell_h(Z_{\mathscr{C}})$
10         $U_L = U$
11         $l = \min\{j \mid j > i \text{ and } v_j \in U_L\}$
12         $\overline{L}_{z'} =$ `ComputeReducedEligibility`(U_L, l)
13         $\tau_{z'} =$ `ComputeTotalDomination`(U_L, l)
14   **if** $\exists$ node in $\ell_l(Z_{\mathscr{C}})$ with $\tau_z = \tau_{z'}$ and $\overline{L}_z = \overline{L}_{z'}$**:** merge z and z'
15   **else:** insert z' into $\ell_l(Z_{\mathscr{C}})$
16 return the root of $Z_{\mathscr{C}}$

Algorithm 4.6 is a top-down construction, in the sense that a node is inserted into $Z_{\mathscr{C}}$ before either of its children. A bottom-up construction can also be used, in which both the high and low child of a node are inserted into the ZDD before the parent is inserted. The bottom-up construction is a natural modification of Algorithm 4.3, where a check is inserted after Line 2 to see if the current node can be merged with any other node at the current level.

Note that for such a construction, the totally dominated set and reduced eligibility set at a node can be computed recursively and stored with each node $z \in Z_{\mathscr{C}}$, as follows (where var$(z) = i$):

$$\tau_z = \big((N(v_i) \cap \{1, \ldots, v_{i-1}\}) \cup \tau_{\text{hi}(z)}\big) \cap \tau_{\text{lo}(z)},$$

where $\tau_{\mathbf{0}} = \{1, 2, \ldots, v_n\}$ and $\tau_{\mathbf{1}} = \varnothing$. Additionally, $\overline{L}_z$ can be computed by

$$\overline{L}_z = \{v_i\} \cup \overline{L}_{\text{hi}(z)} \cup \overline{L}_{\text{lo}(z)}$$

where $\overline{L}_{\mathbf{1}} = \overline{L}_{\mathbf{0}} = \varnothing$. The first condition follows since any totally dominated vertex at z must be dominated by all paths from hi(z) and from lo(z); the second follows because a vertex at z is in $\overline{L}_z$ if it is used at z, or

at hi(z), or at lo(z). Computing these sets in this manner requires constant time. The bottom-up variant of Algorithm 4.6 using these recursive conditions is called the **merging algorithm**.

4.3.3 Variable Ordering Heuristics

In this section, several different heuristic ordering rules are explored for the maximal independent set ZDD:

- **Random Order**: Vertices are randomly permuted and used to construct the ZDD. Several different permutations can be tried, and the one yielding the smallest ZDD can be used.
- **Degree Ordering**: Vertices are ordered by increasing or decreasing degree sequence of G.
- **Degeneracy Ordering**: Vertices are ordered by increasing or decreasing **induced degree sequence**. In other words, the ith vertex in the ordering v_i has the highest or lowest degree in $G[V \setminus \{v_1, v_2, \ldots, v_{i-1}\}]$.
- **Clique Cover Ordering**: This rule computes a covering of G by maximal cliques; the cliques are sorted by size, and the vertices within each clique are ordered arbitrarily.
- **Maximal Path Decomposition Ordering**: This rule computes a set of paths $P_1, P_2, \ldots, P_k$ such that P_i is maximal in $G[V \setminus \bigcup_{j=1}^{i-1} P_j]$. The vertices are then ordered as

  $$v_1^1, v_2^1, \ldots, v_{|V(P_1)|}^1, v_1^2, v_2^2, \ldots, v_{|V(P_2)|}^2, \ldots v_1^k, v_2^k, \ldots, v_{|V(P_k)|}^k,$$

  where v_i^j is the ith vertex along the path P_j, and $|V(P_j)|$ is the length of path P_j.

Eppstein and Strash (2011) use the degeneracy ordering in their algorithm for finding maximal cliques. Moreover, Bergman et al. (2012) describe the maximal path decomposition rule for the independent set BDD, and prove that the width of the ith level of a BDD using this ordering is bounded by the $(i+1)^{st}$ Fibonacci number F_{i+1}. The proof of this result does not directly translate to the maximal independent set case; however, a slightly tighter bound can be proven in this setting:

Theorem 4.3. *The width of the ith level of $Z_{\mathscr{C}}$ is bounded by F_i when the vertices of G are ordered according to a maximal path decomposition.*

Proof. Let $P_1, P_2, \ldots, P_k$ be a maximal path decomposition of G; without loss of generality, assume that G is connected with at least 3 vertices, so $|P_1| \geq 3$. Let $v_1, v_2, \ldots, v_n$ be the vertices of G in the order induced by the path decomposition. Finally, let $w_i(Z_{\mathscr{C}})$ be the number of nodes in $\ell_i(Z_{\mathscr{C}})$. To prove that $w_i(Z_{\mathscr{C}}) \leq F_i$, induction on i is used.

Note that $\ell_1(Z_{\mathscr{C}})$ has one node, the root. $\ell_2(Z_{\mathscr{C}})$ also contains one node since $v_1 \leftrightarrow v_2$. If $v_1 \not\leftrightarrow v_3$ and $v_2 \not\leftrightarrow v_3$, $\ell_3(Z_{\mathscr{C}})$ contains at most 3 nodes (if v_1 or v_2 must be in every maximal independent set of G, then $w_3(Z_{\mathscr{C}}) < 3$). In general, fix $i \in \{1, 2, \dots, n\}$, and assume that for any connected graph G with at least 3 vertices, and any ordering of $V(G)$ induced by a maximal path decomposition, $w_j(Z_{\mathscr{C}}) \le F_j$ for all $j < i$. It suffices to show that $w_i(Z_{\mathscr{C}}) \le F_i$.

To see that this is the case, consider two cases: first, v_i is not the first vertex in some path in the decomposition. In this case, $v_{i-1} \leftrightarrow v_i$, so the only edges that can exist between $\ell_{i-1}(Z_{\mathscr{C}})$ and $\ell_i(Z_{\mathscr{C}})$ are low edges. Thus, there are at most $w_{i-1}(Z_{\mathscr{C}})$ nodes in $\ell_i(Z_{\mathscr{C}})$ with parents in $\ell_{i-1}(Z_{\mathscr{C}})$. Additionally, there can be at most $w_{i-2}(Z_{\mathscr{C}})$ high edges from $\ell_{i-2}(Z_{\mathscr{C}})$ to $\ell_i(Z_{\mathscr{C}})$. If there are no edges from $\ell_k(Z_{\mathscr{C}})$ to $\ell_i(Z_{\mathscr{C}})$ for $k < i-2$, and there are no low edges from $\ell_{i-2}(Z_{\mathscr{C}})$, the result follows from the inductive hypothesis. On the other hand, suppose such an edge exists; call such an edge a **bad** edge. Since the bad edge skips $\ell_{i-1}(Z_{\mathscr{C}})$, there must exist $k < i-1$ such that $v_k \leftrightarrow v_{i-1}$. Delete all such edges in G; this increases the size of $\ell_{i-1}(Z_{\mathscr{C}})$ by at least one, so by the inductive hypothesis, $w_{i-1}(Z_{\mathscr{C}}) \le F_{i-1} - 1$ for the unmodified graph. By the same reasoning, if there are k bad edges, $w_{i-1}(Z_{\mathscr{C}}) \le F_{i-1} - k$. Therefore, the total number of nodes at level i is $w_i(Z_{\mathscr{C}}) \le F_{i-1} - k + F_{i-2} + k = F_i$, as desired.

In the second case, v_i is the first vertex along some path in the decomposition. In this case, all low edges from nodes in $\ell_{i-1}(Z_{\mathscr{C}})$ go to **0**, because v_{i-1} has no neighbor in $\{v_i, v_{i+1}, \dots, v_n\}$. A similar argument as in the first case then follows to show that $w_i(Z_{\mathscr{C}}) \le F_i$. ∎

Note that Theorem 4.3 is entirely a structural result—it makes no use of the equivalence conditions presented in Section 4.3.2. Therefore, there is the opportunity for this result to be significantly tighter, if many nodes in the ZDD are equivalent and can be merged. Furthermore, Theorem 4.3 implies that when using the maximal path decomposition ordering, the worst-case running time of Algorithm 4.3 is improved to $O(n\varphi^n)$, where $\varphi = (1+\sqrt{5})/2$, since $\sum_{i=1}^{n} F_i = F_{n+2} - 1$, and F_i is bounded by $O(\varphi^n)$ (Weisstein, 2013).

4.4 Computational Results

4.4.1 Graph Coloring

A B&P algorithm for the graph coloring problem (see Section 2.6.2) was implemented using a ZDD to solve the pricing problem, together with the CBFS strategy for subproblem exploration, and computational experiments were run on a subset of the instances from the DIMACS graph coloring challenge. This section describes some implementation details for this program, called `B&P+ZDD`, as well as discussing the results of these experiments and a comparison to the best algorithms in the literature.

Initialization and Preprocessing

To reduce the size of problem instances, `B&P+ZDD` uses a standard preprocessing technique: a search is done to find a large clique Q in the graph, and any vertex $v \in V$ with degree less than $|Q|$ is removed. Since a valid coloring for G must use at least $|Q|$ colors, at least one color exists in any proper coloring that is not assigned to any neighbor of v; thus, any proper coloring of $G-v$ can be extended to G without increasing the number of colors used (Méndez-Díaz and Zabala, 2006). A B&B search is employed in a heuristic manner to find an initial large clique. The clique Q can also be used to prove optimality—if a proper coloring of G is found that uses exactly $|Q|$ colors, this coloring must be optimal.

To initialize `B&P+ZDD`, a starting pool of independent sets needs to be generated. A modified version of the initialization procedure described in Malaguti (2008) is used for this purpose. Their algorithm employs a 2-phase approach to find good initial solutions. In the first phase, a genetic algorithm combined with a local search rule searches for valid k-colorings of the graph for some input parameter k. If a valid k-coloring is found, then the procedure is iteratively called with successively smaller values of k until a user-specified time limit is reached. The second phase takes the best solution found in phase 1 and applies a covering heuristic to improve the solution further. `B&P+ZDD` uses a similar procedure to generate its initial pool of independent sets for the RMP, which only runs the first phase of the algorithm described by Malaguti (2008).

Any column generated by the initialization routine can be added to the initial pool $\mathscr{C}'$ for the RMP. However, the initialization procedure often generates a large number of sets; thus, it is necessary to reduce the size of the initial pool. To this end, the RMP is solved once with only the sets used by the best available coloring to get initial dual prices. Only the generated sets with a price above 0.8 are included in $\mathscr{C}'$. This rule includes all sets with negative or close-to-negative reduced cost in $\mathscr{C}'$, since these sets are more likely to improve upon the LP solution to the RMP in early stages of the search.

Cyclic Best-First Search

As described in Section 2.5, when using standard integer branching in a B&P setting, the structure of the search tree can become extremely unbalanced. In particular, long chains of assignments that make no progress towards a solution exist, which (if explored) can dramatically increase the search time. Moreover, in many cases these long chains appear more promising than shorter chains which progress towards a solution. Note specifically that setting a variable y_C in (2.2) to 1 has the potential to satisfy many constraints, whereas if some variable y_C is set to zero, it does not change the structure of the integer program much. Furthermore, it was observed empirically that an assignment of the form $y_C = 0$ did not cause the LP relaxation to change substantially in most cases. Both of these facts lead to the aforementioned unbalanced search tree.

To combat this effect, the CBFS strategy described in Chapter 3 is used together with a contour definition that encourages the exploration of positive-assignment branches first. In particular, the positive assignment labeling function from Chapter 3, which assigns a subproblem to contour K_p if and only if there have been p branching decisions made of the form $y_C = 1$, was used for `B&P+ZDD`. Using this contour definition significantly restructures the order in which subproblems are selected for exploration, and counters the unbalanced search tree produced by the branching strategy (see Figure 3.2c).

ZDD Construction Details

Comparisons were done with both the recursive ZDD construction algorithm (Algorithm 4.3) and the merging ZDD construction algorithm described in Section 4.3 on a subset of the instances from the DIMACS graph coloring database to determine which one was more effective. The recursive construction algorithm was given a limit of 100 000 000 ZDD nodes; the merging algorithm had a node limit that changed with the size of the graph (since a larger graph requires a larger totally dominated and reduced eligibility set). Each algorithm used the maximal path decomposition ordering from Section 4.3.3. The results obtained from these tests are shown in Table A.2.

There are 50 instances for which $Z_{\mathscr{C}}$ could be constructed (or the node limits hit) within the time limit by one of the algorithms. Of these 50, the recursive algorithm significantly outperformed the merging algorithm in 28 cases. Conversely, the merging algorithm outperformed the recursive algorithm in 17 cases. Of the remaining 7 instances which failed to complete within the time limit by either algorithm, the recursive algorithm was able to construct a larger partial ZDD in 4 cases, and the merging algorithm was able to construct a larger partial ZDD in 3 cases.

This difference in performance can be explained by the fact that for some instances, the recursive algorithm needs to make many identical recursive calls to determine node equivalence in the ZDD. In these cases, memoizing the totally dominated set and reduced eligibility set allows the algorithm to short-cut these recursive calls. However, because computing T_z and $\overline{L}_z$ is itself computationally challenging, the recursive algorithm will outperform the merging algorithm for instances which do not require many identical recursive calls. Comparing the total number of maximal independent sets in the graph to the size of the ZDD yields an approximate measure of how much compression occurs in the ZDD—ZDDs that are small relative to the total number of maximal independent sets are able to perform many merges, whereas ZDDs that are larger than the total number of independent sets do not have as many similar paths that can be merged together. Based on these results, it was decided to use the recursive construction algorithm for `B&P+ZDD`, since it seems to perform slightly better than the merging algorithm for most problems.

Additionally, a study of the vertex orderings discussed in Section 4.3.3 was performed (using only the recursive construction algorithm). For each instance, a clique cover was computed by iteratively applying a B&B algorithm to search for maximal cliques, until all vertices are contained in some clique. At each iteration, the B&B algorithm terminated after exploring $2\Delta(G)$ states, and the best clique found was returned. Results from these tests are shown in Table A.3

For these tests, the maximal path decomposition ordering produced the minimally-sized ZDD in 17 cases. The clique cover ordering produced the smallest ZDD in 11 instances, and the degree list ordering and the reverse degeneracy ordering gave the smallest ZDD for 5 instances each. This difference in size appears to be related to the ease of finding large cliques in G; for instance, in the `DSJ` graphs and the `queen` graphs, large cliques can be easily found. In these cases, the clique cover ordering generally performs best. However, in instances that do not contain large cliques (such as the triangle-free `myciel` graphs), the maximal path ordering performs better. Since many of the hard graph coloring instances tend to contain no large cliques, the maximal path ordering was chosen for use by `B&P+ZDD`.

Results from the DIMACS Database

`B&P+ZDD` was implemented in C++ and used CPLEX 12.5 with default settings to solve the RMP; all computational experiments described in this section were performed on a desktop machine with an Intel Core i7-930 2.8GHz quad-core processor and 12 GB of available memory. The B&P algorithm utilized only a single processor core; however, CPLEX operates in parallel by default. All times reported here are aggregated over all cores. For the sake of comparison with the results obtained with the MMT algorithm, the *dfmax* benchmark program was run on the r500.5 instance provided by Trick (2005). The computer used for these experiments took 6.60s CPU time to solve this benchmark instance.

Comparisons were made against three different B&P algorithms available in the literature: first, Malaguti et al. (2011) give an exact algorithm for the graph coloring problem that uses an improved initialization heuristic, together with extensive computational results. These results were performed using standard $0-1$ branching instead of edge branching. Secondly, Gualandi and Malucelli (2012) describe a B&P solver for graph coloring that uses constraint programming techniques to solve the pricing problem; their implementation uses the edge branching rule. Finally, Held et al. (2012) provide a method for computing a numerically safe lower bound for graph coloring, which they embed inside a B&P solver. Using this algorithm, they are able to prove new lower bounds for a number of unsolved instances. Comparisons were also performed against the wide branching solver described in Chapter 5.

Experiments were run on 40 instances from the DIMACS instance database (Trick, 2005). Experiments

were not run on easy instances (those for which the lower bound at the root is sufficient to prove optimality), since these instances do not demonstrate the effectiveness of the ZDD data structure for solving the pricing problem in the presence of branching constraints. The remaining instances were chosen to span a range of difficulty, including ones that are easily solved to optimality by all algorithms in the literature, and others for which no algorithm has yet been able to verify optimality. In addition, experiments were run on 7 additional instances taken from Gualandi and Malucelli (2012). Raw data from these experiments are given in Table A.4.

A time limit of 10 hours was imposed for all experiments, and the ZDD size was limited to 100 000 000 nodes. The initialization procedure from Section 4.4.1 was run for 100 seconds for each instance to generate an initial pool; this did not contribute to the 10-hour time limit. Of the 40 instances tested, most were extremely difficult, and could not be solved by any algorithm within the 10-hour time limit.

`B&P+ZDD` was able to find and verify optimality for 15 of the 47 instances tested. On average, `B&P+ZDD` solves problems four times faster than the best previous algorithm in the literature, and in four cases, the improvement in speed is at least an order of magnitude. Additionally, `B&P+ZDD` is able to verify optimality for three new instances (`1-FullIns_4`, `r1000.5`, and `flat_300_0`) that have not been solved previously by B&P algorithms in the literature (however, in the case of `r1000.5`, the ZDD construction took longer than 10 hours). One other instance, `DSJC250.9`, has only been solved by the B&P solver of Held et al. (2012); their algorithm found a solution in 8685 (adjusted) CPU seconds.

It was observed that modifying the initial pool size can dramatically improve the running time of `B&P+ZDD`; for example, running the initialization procedure for 6100 seconds (the default initialization time limit in Malaguti et al. (2011)) allows `B&P+ZDD` to solve `DSJC125.5` in 31 seconds. Similarly, running the initialization procedure for only 3 seconds allows `B&P+ZDD` to solve `queen9_9` in 2.3 seconds. (this is explained by noting that a large initial pool can slow down the LP solver for the RMP).

Data were collected regarding the average length of time needed to solve the pricing problem for each instance, as well as the growth in size of the ZDD over the course of the algorithm. The average growth in size of a ZDD for any problem was 14%, with a standard deviation of 27%. In one case, the size of the ZDD nearly doubled, at 93% growth; however, even in this case, the length of time needed to solve the pricing problem was not impacted substantially. In most cases when the ZDD could be fully constructed, the length of time needed to solve one iteration of the pricing problem was under a second.

Finally, there are five instances which were solved substantially faster by the MMT graph coloring solver than by `B&P+ZDD`; however, four of these instances were solved at the root node by the MMT solver due to a better initialization procedure, and so do not provide a useful comparison against `B&P+ZDD`. This leaves

only one instance (`queen11_11`) for which some other algorithm substantially outperforms `B&P+ZDD`; for this instance, the lower bound is equal to the optimal objective value, which means the search can be terminated as soon as an optimal solution is found.

4.4.2 Computational Results for the Generalized Assignment Problem

A ZDD-based B&P solver was also implemented for the generalized assignment problem (see Section 2.6.3), and experiments were run against instances from the OR-Library collection. Comparisons were performed against the stabilized B&P algorithm of Pigatti et al. (2005) and the cut-and-branch algorithm of Avella et al. (2010) to determine the effectiveness of the ZDD approach for this problem.

Since the pricing problems for GAP are knapsack problems, the B&P solver first constructs a separate ZDD $Z^i_{\mathscr{C}}$ for each worker, corresponding to the family of valid schedules that can be assigned to the ith worker in the problem. The ZDD construction algorithm is a modification of the dynamic-programming-based merging algorithm given in Behle (2008), which keeps track of a lower and upper bounds for each node in the ZDD, as well as the current **slack**, defined as the remaining worker capacity after some assignments have been made. Behle (2008) proves that two nodes z, z' in a knapsack ZDD can be merged if the slack at z' is between the lower and upper bounds stored at z' (the result was originally proved for BDDs, but it translates easily to ZDDs as well). A maximum-weight assignment for each of the n workers can then be produced by the ZDD, which in turn yields negative-reduced-cost assignments or a proof of optimality for the RMP.

To handle the oscillation of the dual prices discussed in Section 2.6.3, Pigatti et al. (2005) present a stabilized B&P algorithm which imposes bounds on the dual prices; these bounds are slowly relaxed as column generation progresses. It was found that using this method dramatically improves the convergence of the column generation procedure, so it was incorporated into the ZDD-based B&P solver.

When compared to the B&P solver of Pigatti et al. (2005), the ZDD-based solver performed favorably. Instances were usually solved more quickly than in Pigatti et al. (2005), and in a few cases the improvement was an order of magnitude. These results validate the performance gains that can be obtained using ZDDs in conjunction with B&P. However, the cut-and-branch algorithm of Avella et al. (2010) significantly outperformed the ZDD-based solver in almost every case. It is believed that this is primarily due to the slow convergence of the column generation procedure for the generalized assignment problem. Interestingly, preliminary experiments indicated that using standard $0-1$ branching for the ZDD-based GAP solver decreased the convergence rate of the column generation procedure at subproblems in the search tree, though each individual iteration of column generation was substantially faster. However, further experiments must

be run to quantify and explain these results more thoroughly.

4.5 Conclusions

This chapter presents a framework for using standard integer branching in conjunction with B&P algorithms; this framework solves the pricing problem using a zero-suppressed binary decision diagram that is constructed during a preprocessing phase. When new columns are generated, they are restricted from generation by the ZDD a second time; this allows the constrained pricing problem to be solved exactly at every iteration of the algorithm. Using this technique combined with the positive assignment contour labeling function for CBFS, used to counterbalance the resulting lopsided search tree, the standard integer branching scheme can be used in conjunction with a B&P algorithm. This approach yields a faster and more direct solution method in many cases.

Computational results were presented showing that a B&P algorithm implementation for the graph coloring problem outperforms other B&P graph coloring solvers in the literature, in five cases by one or more orders of magnitude. Computational results for the generalized assignment problem also appear to support this result; however, B&P is not an effective algorithm for this problem, primarily due to convergence difficulties during column generation.

Since ZDDs are a generic method to solve the pricing problem, they can be used in conjunction with other B&P methods, even if these methods do not require the solution of the constrained pricing problem (for instance, the robust B&P-and-cut algorithm of de Aragão and Uchoa, 2003). In these settings, the ZDD does not need to have restrictions imposed via `RestrictSet` when a new variable is generated; however, they may still provide benefits, since the ZDD is able to produce a variable of most negative reduced cost at every iteration of column generation.

Chapter 5

A Wide Branching Strategy for Branch-and-Price

In this chapter, an alternate approach towards using integer branching rules in a B&P setting is presented. This approach uses a wide branching rule (Section 2.3.2) as opposed to a binary branching rule. By combining this rule with a novel delayed branching technique, the algorithm search tree is restructured so as to hopefully minimize the number of times the constrained pricing problem must be solved. Moreover, the wide branching method rebalances the search tree so that all subproblems are closer to the root of the tree. The fundamental observation driving this strategy is that the fragility in the pricing problem is asymmetric: often, while performing a null assignment to a variable requires the constrained pricing problem to be solved, performing a positive assignment to the variable does not.

Some similar work has been done in this area previously; Elhedhli et al. (2011) present a B&P algorithm for the bin packing problem with conflicts that creates multiple branches at each subproblem in the search tree; this is a similar method to the wide branching approach presented in this paper, however, it still applies specialized branching rules to maintain the structure of the pricing problem. In the approach presented herein, such rules are not necessary. Similarly, Lodi et al. (2011) discusses a method called **interdiction branching** which branches on multiple variables at once in a generic integer programming setting.

The remainder of this chapter is organized as follows: Section 5.1 gives a theoretical motivation for the wide branching strategy, and describes its application to the graph coloring problem. In Section 5.2, additional implementation details are discussed that improve the performance of the wide branching solver, and Section 5.3 presents a computational comparison of an implementation of B&P with wide branching to other methods in the literature for solving the graph coloring problem. Finally, Section 5.4 provides some concluding remarks and directions for future research.

5.1 The Wide Branching Strategy

This section motivates the wide branching strategy by demonstrating how some restructuring operations transform a fully explored B&P tree T into a related search tree $\hat{T}$ that requires the solution of the constrained

pricing problem at fewer subproblems than T. A description of the restructuring operations (called **path compression** and **forgetful branching**) are presented here, along with theoretical results bounding the number of subproblems in the restructured search tree. Finally, the section concludes with a discussion of the wide branching strategy applied to the graph coloring problem.

In the remainder of this chapter, let $\mathscr{P}$ be a combinatorial optimization problem with a binary integer programming formulation, and let $y_1, y_2, \ldots, y_n$ be binary variables in the IP formulation for $\mathscr{P}$. For a subproblem S in a B&P search tree T for $\mathscr{P}$, the partial assignment at subproblem S is given by $\left(S^1, S^0\right)$, where S^1 and S^0 are the (disjoint) sets of variables that have been fixed to one and zero at S, respectively.

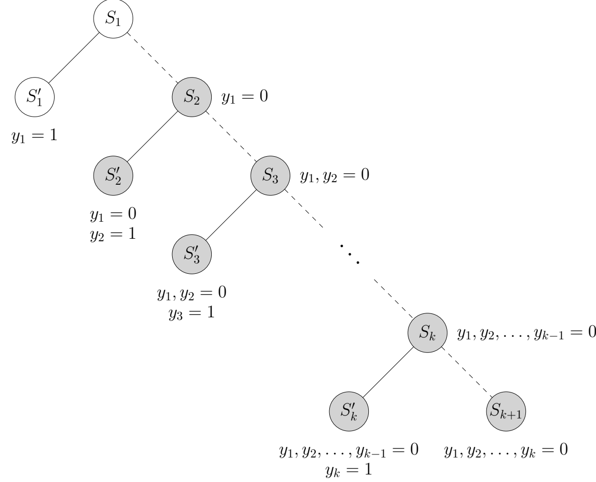
5.1.1 Path Compression and Forgetful Branching

Consider a B&P search tree T for $\mathscr{P}$ that uses the deep branching strategy. As observed in Section 2.5, long paths can exist in T in which every branching decision is a null assignment; a path in T whose branching decisions have at least two null assignments and no positive assignments is called an **uncompressed path**.

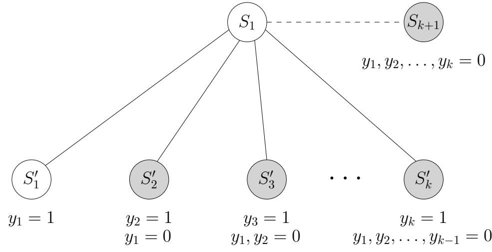
By removing uncompressed paths from T, the length of time needed to explore T can be reduced. To see this, consider an uncompressed path in T with length $k+1$. To fully explore this path and all its direct children, column generation must be performed at $2k+1$ subproblems, and the constrained pricing problem must be solved at $2k-1$ subproblems (see Figure 5.1a). However, observe that the amount of work done along this path can be reduced via the **path compression** operation. This operation takes a long chain of null assignments in T and collapses it to a single subproblem. After path compression, the root of the path has $k+1$ children, and the constrained pricing problem is only solved at k subproblems instead of the original $2k-1$ (Figure 5.1b).

To further reduce the number of subproblems requiring the use of the constrained pricing problem, any null assignments posted at children in which a positive assignment has been made can also be dropped. This operation is called **forgetful branching**, and allows these children to be treated as direct children of the root of the path, each formed by performing a single positive assignment (see Figure 5.1c).

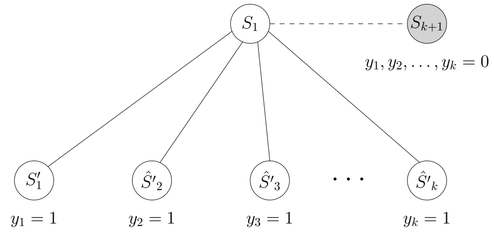
While forgetful branching reduces the number of children of S_1 that require the use of the constrained pricing problem, there are a few drawbacks to this strategy. First, the forgotten null assignments can slow the search process due to the relaxation of the LP solution (however, note that the forgetful branching process still produces a finite search tree, since at least one additional positive assignment is made at each child with dropped null assignments). Secondly, forgetful branching can create many redundant subproblems in the search tree. For instance, setting $y_i = 1$ and then $y_j = 1$ is equivalent to setting $y_j = 1$ and then $y_i = 1$. These two paths both lead to the same partial solution which is represented by two distinct subproblems in



(a) Subproblems $S_1, S_2, \ldots, S_{k+1}$ form an uncompressed path in this search tree; the constrained pricing problem must be solved at all grey subproblems.



(b) Compressing the path connects subproblems $S'_2, S'_3, \ldots, S'_k$ and S_{k+1} to the root; note that subproblems $S_2, S_3, \ldots, S_k$ are dropped, removing the need to solve the constrained pricing problem at them.



(c) Dropping the null assignments at subproblems $S'_2, S'_3, \ldots, S'_k$ forms new subproblems $\hat{S}'_2, \hat{S}'_3, \ldots, \hat{S}'_k$ that do not require the solution to the constrained pricing problem to compute their bounds. Note that the bounds may decrease unless the ULBE condition is satisfied.

Figure 5.1: Using path compression and forgetful branching can reduce the number of times the constrained pricing problem is solved.

the tree. However, if the algorithm keeps track of all generated subproblems, simple dominance rules can prevent identical subproblems from being explored multiple times.

By iteratively applying the path compression and forgetful branching operations to a B&P search tree T, a new search tree $\hat{T}$ can be created which requires the solution of the constrained pricing problem at fewer subproblems in the tree. This result is proved formally in the Wide Branching Theorem, below. Note, however, that the Wide Branching Theorem applies only in a very idealized setting. First, it requires the *a priori* knowledge of a complete tree T in order to construct the smaller tree $\hat{T}$. Secondly, the conditions that guarantee a reduction in the total number of subproblems are unlikely to hold in most practical cases. In practice, however, the computational savings that result from having to make fewer calls to the constrained pricing problem solver often outweigh the (potentially) increased search tree size.

It is assumed in what follows that if a variable y_i is given a positive assignment at a subproblem S in T, a branch is also created at S making a null assignment to y_i. In order to prove the Wide Branching Theorem, it is necessary to make an additional, stronger assumption about the structure of the problem being solved; intuitively, this assumption ensures that if a subproblem is pruned before the path compression and forgetful branching operations are applied, it can still be pruned afterwards. While this assumption is unlikely to be realistic in most settings, it is necessary to ensure that the reformulated tree does not grow in size.

Definition 5.1. *Let (S^1, S^0) be a partial assignment of values to variables at a subproblem S, let $f_{LP}(S)^*$ be the optimal solution value to the LP relaxation at S, and let $f_{LP}(S_\varnothing)^*$ be the optimal LP solution to the partial assignment $(S^1, \varnothing)$, that is, the partial assignment to $\mathscr{P}$ which matches all positive assignments of S, but has no null assignments. If for every pair of partial assignments S and $S_\varnothing$, $\lceil f_{LP}(S)^* \rceil = \lceil f_{LP}(S_\varnothing)^* \rceil$, then the problem $\mathscr{P}$ is said to satisfy the* **unconstrained lower bound equality** *(ULBE) condition.*

The ULBE condition ensures that the restructuring operations do not increase the size of the search tree by removing some restrictions that have been posted at a particular subproblem. Since the LP relaxation may become weaker when a constraint imposing a null assignment ($y_i = 0$) at a subproblem is removed, if the ULBE condition is not satisfied, it could be the case that the lower bound shrinks and no longer allows a subproblem to be pruned after restructuring. Thus, it is necessary to forbid this from occurring. In settings where the ULBE condition holds, the following theorem implies that any search tree can be reduced to a form with no uncompressed paths by repeatedly collapsing them:

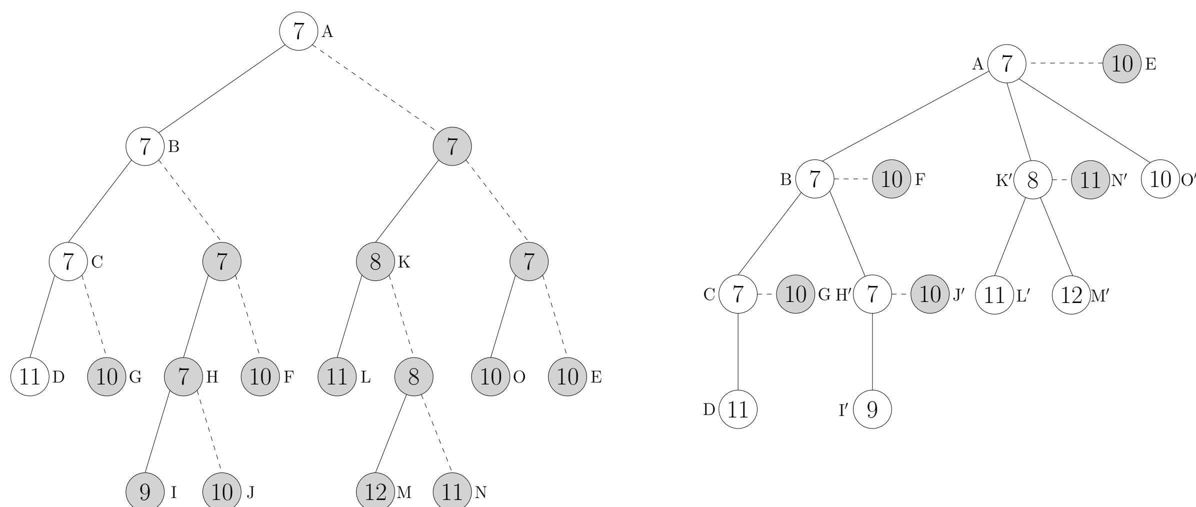
Theorem 5.1 (The Wide Branching Theorem). *Let $\mathscr{P}$ be a binary minimization problem with an integer optimal solution satisfying the ULBE condition. Given a B&P tree T for $\mathscr{P}$ that contains an uncompressed path P, there exists a B&P tree $\hat{T}$ that has strictly fewer subproblems by collapsing P into a single vertex.*

Proof. As in Figure 5.1, let $P = S_1 S_2 \dots S_{k+1}$ be a longest uncompressed path in T such that S_i ($i \in 1, 2, \dots, k$) has two child branches that fix variable y_i to either 0 or 1 in the LP relaxation, and S_{k+1} has no children (note that for notational convenience, the variables are indexed as they appear on the path, not as they are given in the problem formulation). Let S'_i be the subproblem in T hanging from P where a positive assignment has been made to y_i (see Figure 5.1a). To create $\hat{T}$, duplicate all branching decisions made in T except at subproblem S_1; at this point, create branches $\hat{S'}_1, \hat{S'}_2, \dots, \hat{S'}_k$ with a positive assignment made to y_i at branch $\hat{S'}_i$. Finally, create a single subproblem S_{k+1} that performs a null assignment for $y_1, y_2, \dots, y_k$ (see Figure 5.1b). Since there are no children at subproblem S_{k+1} in T, it must be the case that S_{k+1} is pruned in $\hat{T}$.

Furthermore, by the ULBE condition, note that the lower bound at S'_i equals the lower bound at $\hat{S'}_i$,

which means that subproblem $\hat{S'}_i$ is pruned in $\hat{T}$ if and only if S'_i is pruned in T. To complete exploration of $\hat{T}$, at each $\hat{S'}_i$, exactly duplicate the branching decisions made in the subtree rooted at S'_i in T; again by the ULBE condition, note that subproblems in these subtrees are pruned in $\hat{T}$ if and only if the corresponding subproblems are pruned in T.

Therefore, since no new subproblems are introduced into the B&P tree, and the path P has been compressed into a single subproblem S_{k+1}, the number of subproblems in $\hat{T}$ that require the solution of the constrained pricing problem are strictly fewer than in T. ∎



(a) A B&P tree produced by the standard deep branching rule; computed bounds are shown in the center of each subproblem. Grey subproblems require the solution of the constrained pricing problem.

(b) The resulting search tree after repeatedly applying path compression and forgetful branching operations; subproblems $\{H', I', \ldots, O'\}$ have dropped null assignments, and thus the bounds at these subproblems could change unless the ULBE condition is satisfied.

Figure 5.2: Repeated path compressions produce a minimal B&P tree

Repeated application of Theorem 5.1 produces a search tree containing no uncompressed paths (see Figure 5.2). Applying forgetful branching after each path compression operation (as in Figure 5.1) causes the number of subproblems requiring the solution of the constrained pricing problem to decrease. Thus, the B&P algorithm that obeys the branching decisions in $\hat{T}$ will solve $\mathscr{P}$ more quickly than the one that generates T.

Note that the ULBE condition is unlikely to hold at every subproblem in the search tree. In fact, it only needs to hold at subproblems which are pruned in T, but this is still unlikely to occur in practice. Consequently, this means that the number of subproblems in the search tree may increase after restructuring. However, the hope is that the time needed to explore the additional subproblems is substantially less than the time needed to solve the additional constrained pricing problems in T.

5.1.2 Wide Branching and Graph Coloring

The wide branching strategy for graph coloring attempts to replicate the branching decisions made in a tree containing no uncompressed paths. To do this, note that the path compression operation can be applied in an online fashion (that is, without knowing the full tree T) so long as the variables in the path are known along with their branching order. However, as this is not the case in most practical settings, a heuristic rule is used to guess a potential set of variables prior to path compression. One natural such rule is to create a branch for every fractional variable in the solution to the RMP at the subproblem. However, this will generally create a large number of branches; most of these will not lead to an optimal solution, but still require column generation to compute their lower bound.

Therefore, a different branching rule must be used; most rules currently in the literature are based either on information from the LP relaxation or on information about the problem structure (e.g., what vertices have an integral coloring). However, intuitively, both components appear to play a role in the branching choices made along an uncompressed path, so the following rule for the graph coloring problem is used which attempts to combine the two sources of information available at the current subproblem S in the search tree:

1. An independent set C_1 is chosen with the most fractional value (closest to 0.5) in the LP relaxation at subproblem S.

2. A vertex $v = \arg\max_{u \in C_1} d_{sat}(u)$ is selected with the highest degree of saturation in C_1 (that is, v has the most (integral) differently-colored neighbors among all vertices in C_1).

3. For each of the k independent sets $C_1, C_2, \ldots, C_k$ which contain v and have $y_{C_i} > 0, i \in \{1, 2, \ldots, k\}$, create one child of S that performs a positive assignment to y_{C_i}. Additionally create one child of S that gives a null assignment to all sets $C_1, C_2, \ldots, C_k$, as well as any null assignments generated at a previous stage (see the discussion of delayed branching, below). This last subproblem is called the **delayed subproblem**, and is denoted $\bar{S}$.

The branching strategy above attempts to emulate the path compression and forgetful branching process by guessing columns $C_1, C_2, \ldots, C_k$ which compose some uncompressed path rooted at S. However, it may be the case that the entire uncompressed path could not be guessed by the above branching rule. In this case, $\bar{S}$ will not be pruned; instead of immediately guessing more subproblems that might lie along this uncompressed path, the search delays exploration of the remainder of the path until later. This process is known as **delayed branching**.

Later, when the algorithm returns to $\bar{S}$, it attempts to guess more subproblems along the uncompressed path rooted at S; in particular, to avoid repetition of previously-generated states, each of $y_{C_1}, y_{C_2}, \ldots, y_{C_k}$

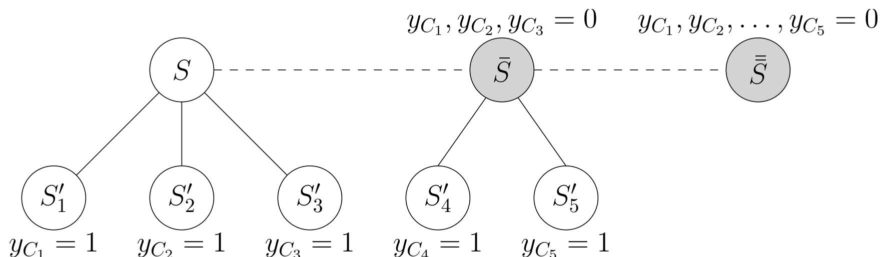


Figure 5.3: An example of the delayed branching process. y_{C_1}, y_{C_2}, and y_{C_3} are guessed as members of an uncompressed path rooted at S. A subproblem $\bar{S}$ restricts these sets from being generated, but $\bar{S}$ cannot be pruned. Later, y_{C_4} and y_{C_5} are generated at $\bar{S}$, and do not inherit the null assignments at $\bar{S}$, and a subproblem $\bar{\bar{S}}$ is generated with sets $y_{C_1}, y_{C_2}, \ldots, y_{C_5}$ restricted.

must not be generated as children of $\bar{S}$, so these variables are imposed as null assignments at $\bar{S}$. Suppose that some new independent sets $\bar{C}_1, \bar{C}_2, \ldots, \bar{C}_{\bar{k}}$ are generated as children of $\bar{S}$; since each of these subproblems are on an uncompressed path rooted at S, by the forgetful branching technique, they drop the null assignments posted at $\bar{S}$. Finally, a subproblem $\bar{\bar{S}}$ (the new delayed subproblem) is added. If $\bar{\bar{S}}$ is the end of an uncompressed path rooted at S, no further children need to be generated. However, if $\bar{\bar{S}}$ is *not* the end of such a path, new subproblems must be guessed that do not use $C_1, C_2, \ldots, C_k$ or $\bar{C}_1, \bar{C}_2, \ldots, \bar{C}_{\bar{k}}$. In other words, when children of a delayed subproblem are generated, all independent sets branched on at previous delayed subproblems must be restricted to zero (see Figure 5.3).

Finally, note that a slightly tighter bound at the delayed subproblems can be achieved as follows: let subproblem $\bar{S}$ be the delayed subproblem for S, and let subproblem S' be a child of $\bar{S}$. At the delayed subproblem of S' (that is, $\bar{S}'$), the children of S and S' both can be restricted at $\bar{S}'$. In other words, the forgetful branching rule is applied at all children of S *except* for its delayed subproblem. This does not affect algorithm performance, since the constrained pricing problem must be solved at $\bar{S}'$ regardless. For example, in Figure 5.3, the null assignments posted at subproblem $\bar{S}$ can be propagated to the delayed subproblems for S'_4 and S'_5. If a positive assignment and a null assignment conflict in this case, the positive assignment takes precedence.

In the worst case, only two sets C_1 and C_2 are generated at each subproblem in the search tree (since C_1 is chosen to be a fractionally-used independent set, there must be at least one other such set covering the chosen vertex v). However, by carefully choosing the branching rule, better performance may be obtained. One immediate improvement is apparent: the wide branching strategy generates a tree that is much more balanced; by trying to perform path compression, it eliminates long chains of null assignments, where no progress towards a solution has been made. This allows for large regions of the search space to be pruned higher in the tree.

Note that one disadvantage to the online wide branching strategy is that in order to fully explore the search tree, the RMP must potentially be solved at many more subproblems if the branching rule generates many additional children that do not lead to an optimal solution. Since column generation must be performed every time the RMP is solved, this could potentially lead to slower solution times. Thus, the wide branching strategy will be most effective when the constrained pricing problem is substantially more difficult to solve than the unconstrained problem. This observation is verified in the computational results.

5.2 Implementation Details

A version of the wide branching rule was developed for a graph coloring B&P solver called `B&P+Wide`. This implementation incorporates a number of additional features that demonstrate practical improvements in running time (though they do not change the theoretical complexity of the algorithm, which is still exponential). Pseudocode for `B&P+Wide` is given in Algorithm 5.1.

Algorithm 5.1: B&P+Wide(X, f)

1 Set $\mathscr{S} = \{X\}$
2 Initialize $\hat{x}$ and the initial RMP pool $\mathscr{C}'$
3 **while** $\mathscr{S} \neq \varnothing$**:**
4   Select a subproblem $S \in \mathscr{S}$ to explore
5   **if** a solution $\hat{x}' \in \{x \in S \mid f(x) < f(\hat{x})\}$ can be found**:** Set $\hat{x} = \hat{x}'$
6   **if** S cannot be pruned**:**
7   Generate sets $C_1, C_2, \ldots, C_k$ via the wide branching rule
8   **for** each $C_i \in \{C_1, C_2, \ldots, C_k\}$**:**
9   Create a child S_i' of S with $y_{C_i} = 1$
10   ⟪ *Column generation loop* ⟫
11   **while** $\exists\, C \in \mathscr{C} \setminus \mathscr{C}'$ with negative reduced cost**:** Add C to $\mathscr{C}'$
12   Compute a lower bound at S_i using added columns
13   ⟪ *Delayed branching: R is the set of other restrictions that should be imposed at $\bar{S}$* ⟫
14   Create a subproblem $\bar{S}$ with $\{y_{C_1} = 0, y_{C_2} = 0, \ldots, y_{C_k} = 0\} \cup R$
15   Insert $S_1, S_2, \ldots, S_r, \bar{S}$ into $\mathscr{S}$
16   Remove S from $\mathscr{S}$
17 return $\hat{x}$

5.2.1 Solving the Pricing Problem

Two different methods are used by `B&P+Wide` to solve the pricing problem: a fast heuristic solver called `HeurPrice` and a slower exact solver called `ExactPrice`. To generate new columns, the heuristic search method is called first, provided that the following conditions are met: (i) the most recent call to the exact solver took longer than 0.1s to complete, and (ii) the last column generated by either solver had a weight

larger than 1.02. The rationale behind condition (ii) is that there is generally no gain to calling the heuristic solver if the price of the best independent set is close to one because the heuristic is usually unable to find it.

The Heuristic Pricing Problem Solver

The heuristic search method, called `HeurPrice`, is an extension of the tabu search heuristic by Grosso et al. (2008) that can find maximum-weighted independent sets instead of just maximum independent sets. This algorithm maintains a partial independent set $\bar{C}$, as well as a tabu list R of vertices that cannot be added to $\bar{C}$. At each iteration of the algorithm, one of three types of local search moves is performed, either an **improvement move**, a **weighted swap move**, or an **unweighted swap move**. Note that Malaguti et al. (2011) also use a heuristic method based on the algorithm by Grosso et al. (2008); however, their version differs slightly from the one presented here. In particular, their heuristic method only considers improvement or swap moves with one or two participating vertices (called a $1-1$ **exchange** or a $2-1$ **exchange**), whereas `HeurPrice` performs local search moves that can have an arbitrary number of participating vertices.

An improvement move can be made if there exists a vertex $u \notin \bar{C} \cup R$ such that $\pi(u) > \sum_{v \in N(u) \cap \bar{C}} \pi(v)$. In this case, u is added to $\bar{C}$, and all its neighbors are removed from $\bar{C}$. A weighted swap move, on the other hand, can be made if there exists a vertex $u \notin \bar{C} \cup R$ such that $\pi(u) = \sum_{v \in N(u) \cap \bar{C}} \pi(v)$. In this case, again u is added to $\bar{C}$ and all u's neighbors are removed from $\bar{C}$. An unweighted swap move is precisely the same as in the unweighted algorithm by Grosso et al. (2008): if there is a vertex $u \notin \bar{C} \cup R$ such that $|N(u) \cap \bar{C}| = 1$, u and its neighbor in $\bar{C}$ are swapped. In each case, all of u's neighbors in R are added to the tabu list R to prevent cycling. Finally, if none of these moves can be performed but $\bar{C}$ is not maximal, the set is completed in a greedy fashion. On the other hand, if $\bar{C}$ is maximal, a partial random restart is performed: a vertex $u \in V$ is selected, and the independent set constructed in the next phase of the algorithm is initialized with $u \cup (\bar{C} - N(u))$. When such a restart occurs, the tabu list R is cleared.

At each iteration, `HeurPrice` performs one of the above search moves (if possible) until a dynamically determined iteration limit is reached. This iteration limit is based on how successful previous calls to `HeurPrice` have been: if the independent set returned by `HeurPrice` has price less than one (that is, if it does not improve the current solution to the RMP), the number of iterations taken during the next invocation of `HeurPrice` is doubled. On the other hand, if the independent set returned by `HeurPrice` does improve the value of the RMP, the number of iterations taken during the next call to `HeurPrice` is set to the average number of iterations over all calls to `HeurPrice`. The iteration limit is constrained to the range $[1000, 10000]$ (so it will never run for longer than 10000 iterations, for instance). If the value of the current solution found by `HeurPrice` ever exceeds 1.1, the routine aborts early and returns that solution.

The Exact Pricing Problem Solver

The exact pricing problem solver (called `ExactPrice`) is based on the fast branch-and-bound maximum-weighted independent set solver presented in Held et al. (2012). `ExactPrice` has been modified so that it also solves the constrained pricing problem by maintaining a current list of forbidden independent sets. When `ExactPrice` reaches a terminal state, it compares the current solution to each of the forbidden sets, in addition to computing its reduced cost. If the independent set is not forbidden and has negative reduced cost, the solver updates its incumbent; otherwise, it continues exploration. If no unrestricted solutions can be found with negative reduced cost, the solution to the RMP is optimal.

While this method allows the constrained pricing problem to be solved, the two pruning rules described in Held et al. (2012) must be disabled in `ExactPrice` when such restrictions are present. These pruning rules take advantage of structure in the unconstrained pricing problem to narrow the space that must be explored. For example, one such rule computes the **surplus** at available vertices in the graph, where the surplus of v is defined as $\pi(v) - \sum_{u \in N(v)} \pi(u)$, that is, the marginal gain achieved by taking v instead of all of its neighbors. If no restrictions are present, v can be automatically added to the current solution if it has positive surplus, since an independent set using v will strictly dominate any set that does not use v; however, in the presence of restrictions, it is likely that the independent sets containing v have already been found and restricted, so independent sets using neighbors of v must also be considered. The other pruning rule, the clique cover rule, must also be disabled for a similar reason. In addition, the branching rules used by `ExactPrice` in the unconstrained case may prune subproblems that should not be pruned in the presence of constraints, so a similar modification is made in this case.

However, one improvement can be made to the `ExactPrice` algorithm even in the constrained pricing problem setting. This improvement stores a list of **no-goods** – that is, sets of vertices that are not allowed to appear in any valid independent set with negative reduced cost. As the `ExactPrice` algorithm explores the search space, when the subtree at a subproblem has been exhausted, the currently-used vertices at that subproblem are appended to the no-good list. In the future, any subproblem whose independent set structure contains vertices in some no-good may be pruned, since that region has been explored previously.

Finally, if `ExactPrice` ever finds an independent set with price larger than 1.05, the solver immediately terminates and returns this solution. While this mechanism may slow down column generation convergence, in practice it is necessary because solving the pricing problem to optimality at every stage becomes prohibitively slow (though it must be solved to optimality in the last iteration).

5.2.2 Exploration Strategy

The cyclic best-first search (CBFS) strategy is used to select new subproblems for exploration, along with the depth contour rule described in Section 3.1. In addition to the other benefits of the CBFS strategy, in the wide branching context, CBFS yields a natural re-exploration rule for subproblems that have been re-inserted to the search tree by delayed branching. Specifically, the next time the contour containing a previously-visited subproblem is visited, if the re-inserted subproblem's bound is still the best in the contour, it is chosen for re-exploration (recall that when a subproblem is selected for re-exploration, all previously-generated sets are restricted; performing these restrictions may cause the LP solution value to increase, and possibly allow the subproblem to be pruned).

5.2.3 Dominance Checks

The nature of the wide branching strategy is such that two distinct branches of the search may be identical. In particular, if C_1 and C_2 are two independent sets under consideration, one branch could select C_1 first and then C_2, while a second branch could select them in the alternate order. However, the order in which these sets are selected has no impact on the solution; thus, to reduce unnecessary work in the search tree, `B&P+Wide` employs dominance checks to prevent this situation. Specifically, if the partial colorings given by two distinct subproblems in the search tree color the same set of vertices, and one uses at most the same number of colors as the other, the latter subproblem is pruned from exploration, because any complete solutions generated from it will also be explored by the other branch.

To check for dominance at a subproblem, `B&P+Wide` maintains a global hash table of previously-explored subproblems; any time a new subproblem $S_1 \in T$ is generated, a lookup is performed in the hash table to determine if another subproblem $S_2 \in T$ has been previously identified that covers the same set of vertices with fewer colors. The hash function used simply sums together the indices of vertices covered by sets indexed by variables in S_1^1 (the set of positive assignments at S_1); since this function is not necessarily robust to collisions, a secondary check must be performed to determine if the dominance condition is actually satisfied. However, the amount of time spent searching this hash table is generally small in comparison to the total running time of the algorithm.

5.2.4 Additional Improvements

A standard technique is used to generate multiple columns in between each iteration of column generation (see, e.g., Farley (1990)). After a new column is generated, the vertex weights of G are updated according to Equation (5.1), and the pricing problem is re-solved with the new weights. This enables multiple columns

to be generated with negative, or close-to-negative, reduced cost before the LP relaxation needs to be solved again. B&P+Wide generates up to five new columns using this method before re-solving the LP relaxation.

$$\pi'(u) = \begin{cases} \pi(u)/\pi(C) & \text{if } u \in C \\ \pi(u) & o.w. \end{cases} \tag{5.1}$$

Additionally, as the search process progresses, the length of time needed to generate new columns from the constrained pricing problem can increase dramatically. To maximize the amount of computation time that B&P+Wide spends exploring new regions of the search tree, instead of getting stuck searching for a particularly difficult-to-find independent set, a time limit is imposed that aborts the delayed branching procedure if it runs for more than five seconds. In this case, the subproblem's old lower bound is re-used, and CBFS drives the search process to a different area of the tree, with the hope that the next time the subproblem is re-explored, additional columns will have been added that can either prove optimality of the RMP or guide the pricing problem search more effectively.

5.3 Computational Experiments

B&P+Wide was implemented and tested on a subset of the graphs from the DIMACS implementation challenge testbed (see Section 2.6.2); for the sake of comparison, the deep branching strategy was also implemented (referred to as B&P+Deep). All computational experiments were performed using an Intel Core i7-930 2.8GHz quad-core processor with 12 GB of available memory. The B&P algorithm was implemented in C++ and used CPLEX 12.3 to solve the RMP. The B&P algorithm utilized only a single core of the processor; however, CPLEX operates in parallel by default. All times reported are aggregated over all cores. The same initialization and preprocessing measures described in Section 4.4.1 were used in B&P+Wide and B&P+Deep.

All improvements described in Section 5.2 were used in both the wide and deep versions of the B&P solver, with the exception of the dominance rules, which don't have an easily-implemented counterpart in B&P+Deep. This implementation uses depth-first search because of its low memory requirements. Additionally, to enable a fair comparison between the two strategies, B&P+Deep does not modify the graph structure using the edge branching rule. This also allows for a comparison against the results presented in Malaguti et al. (2011), as the most complete computational results reported in their paper use variable branching (they also report some limited tests with the edge branching rule, and do not see a significant difference in running times for this rule). The algorithm used in their paper is referred to herein as the MMT algorithm, and the initialization procedure used by B&P+Wide and B&P+Deep is called InitMMT.

For the sake of comparison with the results obtained with the MMT algorithm, the *dfmax* benchmark program was run on the r500.5 instance provided by Trick (2005). The systems used for these experiments took 6.60s user time to solve this benchmark instance, which is only slightly faster than the results obtained by Malaguti et al. (2011) (7s user time). Thus, the times reported by Malaguti et al. (2011) are treated as roughly equivalent to `B&P+Wide` and `B&P+Deep`. The complete set of computational data obtained from these experiments is given in Table A.5.

In an attempt to narrow the source of the improvements, a smaller set of tests was also run with deep branching with CBFS and deep branching with BFS. These search strategies performed slightly worse than DFS on the selected set of problems; furthermore, many B&P algorithms use DFS due to its low memory requirements; therefore the reported results here are from deep branching with DFS. Additionally, a limited set of tests was done that branched on the variable closest to 1.0 instead of 0.5, similarly to some rules proposed by Mitra (1973). In this setting, using the "closest to 1.0" rule performed slightly worse than the "closest to 0.5" rule. Finally, no comparison was done using CPLEX as the pricing problem solver, as it was observed by Held et al. (2012) and in preliminary experiments that it was substantially slower than the fast branch-and-bound solver described in Section 5.2.1.

Of the fourteen instances tested for which an optimal solution can be verified by at least one solver (excluding `myciel3`, which is too easy to provide any meaningful information), seven are solved faster by `B&P+Wide` than by the MMT algorithm. These problems are `DSJC125.5`, `DSJC125.9`, `DSJR500.1c`, `queen9_9`, `queen10_10`, `myciel4`, and `myciel5`. In almost every instance, `B&P+Wide` is able to solve the problem at least an order of magnitude faster than the MMT algorithm; furthermore, in each case the initial solutions found by `InitMMT` are equal to or worse than the initial solutions found by the MMT algorithm—in other words, despite starting with an inferior initial solution, `B&P+Wide` is substantially faster at finding and verifying the optimal solution than the MMT algorithm. Additionally, note that for one of these problems, `myciel5`, `B&P+Wide` is able to verify optimality of the solution, whereas the MMT algorithm could not establish optimality within the 10-hour time limit.

There are six problem instances that are solved faster by the MMT algorithm than `B&P+Wide`; in addition, for many problems that do not terminate within the 10-hour time limit, the MMT algorithm is able to find a better solution than `B&P+Wide`. However, for every problem considered, the initial solution found by the MMT algorithm is at least as good as the initial solution found by `InitMMT`; therefore, it should be expected that `B&P+Wide` would require additional time to find solutions with smaller chromatic numbers. Furthermore, in many cases the initial solution found by the MMT algorithm equals the MMT lower bound, which allows the MMT algorithm to terminate without needing to branch. Thus, these problem instances

are not particularly useful in determining the effectiveness of the wide branching rule. If the initial solutions produced by the MMT algorithm were fed into B&P+Wide, the algorithm is expected to perform better (recall that the MMT algorithm employs a 2-phase initialization procedure, whereas InitMMT only implements the first phase).

Since a number of algorithmic components differ between B&P+Wide and the MMT algorithm, a comparison was also done between B&P+Wide and B&P+Deep. For this comparison, problems were run with the same initial solution and the same initial random seed. Of the 9 problem instances solved to optimality by B&P+Wide, all except queen11_11 were also solved by B&P+Deep. In addition, DSJC250.9 was solved to optimality by B&P+Deep but not B&P+Wide. Malaguti et al. (2011) report that DSJC250.9 has an unknown chromatic number, but Held et al. (2012) state that their solver was able to prove optimality for this problem in 11094 seconds (they proved this value by improving the value of the lower bound to 72, whereas B&P+Deep used a lower bound of 71 and exhausted the search space). Of the eight instances solved by both B&P+Wide and B&P+Deep, five (DSJR500.5, queen9_9, queen10_10, myciel4, and myciel5) are solved faster by B&P+Wide than B&P+Deep, and in one case the improvement is an order of magnitude. The remaining three problems are solved faster by B&P+Deep.

A comparison was also done with the recent results reported by Gualandi and Malucelli (2012); this paper employs three different methods for solving the graph coloring problem. The first two, CP-UB and CP-LB, employ constraint programming techniques to solve the graph coloring problem directly (without using B&P). The third method, called CG-CP, uses constraint programming to solve the pricing problem in a B&P algorithm. For comparison, they report a running time of 8.74s on the r500.5 instance; thus it is estimated that the machine used for B&P+Wide is about 25% faster than for Gualandi and Malucelli (2012). The CP-UB and CP-LB algorithms perform very well on a subset of the problems considered herein (most notably, myciel4, myciel5, and myciel6, which can all be solved in under 175 seconds). However, queen9_9 is only solved in 113 seconds, which is about 85 seconds when differences in machine speed are taken into account. A fairer comparison with B&P+Wide considers their B&P implementation; in this setting, B&P+Wide runs significantly (at least an order of magnitude) faster than CG-CP in all but one case.

Finally, for one problem, latin_square_10, note that both B&P+Wide and B&P+Deep are able to improve the upper bound for the problem, compared to the best solution found by the MMT algorithm, though B&P+Wide is able to find a better solution than B&P+Deep.

5.3.1 Analysis of Wide versus Deep Branching

To understand why `B&P+Deep` outperforms `B&P+Wide` on certain problems, and in order to gain a more detailed understanding of how the deep and wide versions of the branch-and-bound solver perform, more fine-grained statistics were collected for each of these problems. Some interesting observations from these statistics are summarized below; raw data are included in Tables A.6 and A.7.

Firstly, since the principal advantage of wide branching is to allow the unconstrained pricing problem to be solved more often, data were collected on the total number of columns generated over the course of the algorithm. To allow for a comparison across different running times, the number of columns generated was divided by the total length of time taken by the algorithm to determine the average number of columns generated per second of running time. These values were then averaged across all tested instances. Problems that ran out of memory before the time limit was hit were terminated by the operating system before detailed statistics could be collected.

On average, `B&P+Wide` was able to generate 5.0 columns per second of CPU time, whereas `B&P+Deep` was only able to generate 3.8 columns per second of computation time. However, problems that could be solved in under 2s of computation time tended to generate a disproportionately large number of columns per second, and some problems generated very few columns due to difficulty. When these problem instances were removed, `B&P+Wide` generated on average 2.7 columns per second of computation time, whereas the deep solver was only able to generate 1.0 columns per second. These data provide empirical evidence that, by making positive assignments to variables along every branch, and only solving the constrained pricing problem during the delayed branching procedure, wide branching is able to generate more columns in the same amount of computation time than deep branching. This is a desirable property because it suggests that the RMP can be solved more quickly, and thus that more subproblems can be identified in the search space.

Furthermore, this analysis demonstrates why the deep branching strategy is more effective than the wide branching strategy in some cases. For the 4 problems that `B&P+Deep` solved more quickly than `B&P+Wide` (including `DSJC250.9`, which `B&P+Wide` was unable to solve), the number of columns generated per second is about the same or greater for `B&P+Deep`. For the problems which `B&P+Wide` solves more quickly than `B&P+Deep`, the number of columns generated per second by `B&P+Wide` is an order of magnitude greater than by `B&P+Deep`. This implies that the specific branching rule for `B&P+Wide` described in Section 5.1.2 is not effectively emulating the behavior of the ideal search tree guaranteed by the Wide Branching Theorem in all cases, and thus is performing worse than expected.

Finally, to gain insight into the behavior of `B&P+Wide` as it explores the search space, statistics on the

maximum and average branching factors were collected. The maximum branching factor varied significantly by problem instance, with the lowest value of 3, and the largest value of 113, with an average maximum branching factor of 19.6. However, in most cases, the maximum branching factor occurred at the root of the search tree, and decreased substantially at child subproblems. Across all problem instances, the average branching factor was 13.1, and for problems which explored more than one subproblem, the average branching factor was 10.3, indicating that as the search with the wide branching rule progresses, the branching factor at subproblems is substantially reduced.

5.4 Conclusion

This chapter describes an implementation of a wide branching strategy for B&P algorithms, and computational results are discussed for the graph coloring problem. The Wide Branching Theorem is proved, which shows how to take a fully-explored search tree T and transform it into a smaller tree that requires fewer calls to the constrained pricing problem solver. While the Wide Branching Theorem cannot be directly applied to B&P algorithms, a heuristic rule is provided that attempts to duplicate the results of this theorem in an online fashion. This rule has been shown to be competitive with the state-of-the-art graph coloring solvers in terms of computational running time.

In particular, computational results show that wide branching for graph coloring is able to generate substantially more columns than the deep branching solver, which implies that it is able to identify and possibly prune more of the search space in the same amount of time. For most problems in which wide branching was able to prove optimality, the solution was reached substantially faster than the times reported by Malaguti et al. (2011), one of the best algorithms available in the literature. Additionally, the wide branching strategy outperforms a comparable implementation of B&P with deep branching for many problems.

Additional work needs to be done to determine a better wide branching rule for the graph coloring problem that yields better performance for the instances in which `B&P+Deep` performs better than `B&P+Wide`. An analysis of how the LP changes as branching decisions are made in deep branching may lead to better heuristics and branching rules that more closely emulate the ideal tree described by the Wide Branching Theorem. It is also beneficial to perform a computational analysis for a large number of different problems to determine how frequently the ULBE condition holds for various instances.

Chapter 6

Conclusion

In this dissertation, three new ideas are presented that can be used with branch-and-bound or branch-and-price algorithms to obtain faster and more effective algorithms in practice. The first of these methods uses a recent search strategy called cyclic best-first search (CBFS). It is shown that this strategy is the most general search strategy that can be defined, in the sense that CBFS can emulate any other search strategy by constructing an appropriate contour definition. Bounds are also proven relating CBFS and BFS, and the relative ordering of subproblems that is obtained by different contour labeling functions is studied.

The second method described in this dissertation is an extension to branch-and-price algorithms that uses a data structure called a zero-suppressed binary decision diagram (ZDD). This data structure is used to characterize all of the valid inputs to a pricing problem in a B&P framework, and the `RestrictSet` algorithm is presented to allow the ZDD to solve the constrained pricing problem at subproblems in the search tree. The use of ZDDs together with the CBFS strategy gives a straightforward way to use standard integer branching in a B&P setting, while avoiding the difficulty of the constrained pricing problem or the issues occurring when dealing with unbalanced search trees. Computational results are presented showing the effectiveness of this approach.

Finally, this dissertation describes the wide branching algorithm, which is an alternative framework for solving problems using B&P. In this method, instead of trying to speed up the solution times for the constrained pricing problem, the search tree is restructured using a wide branching method instead of a binary branching method, with the goal of reducing the number of times the constrained pricing problem needs to be solved. A delayed branching method is proposed to limit the branching factor at nodes in the search tree, and forgetful branching is used to drop branching constraints that lead to hard instances of the pricing problem.

The research in this dissertation opens up a number of future research directions that show promise. Firstly, while the CBFS strategy has been quite successful in a large number of different settings, the reasons behind its performance on any particular problem instance are often difficult to understand. In some cases, such as for the graph coloring B&P algorithm, it is relatively clear why a particular contour

labeling function performs well. In other cases, most notably on the library of mixed-integer programming problems, there is no clear relationship between problem structure and choice of labeling function. In fact, the labeling function that performs best in these cases appears to be highly dependent on the specific instance under consideration! Therefore, one significant direction for future research will be to obtain a better understanding of what labeling functions should be used for what problems.

One approach towards answering this question would use machine learning techniques in conjunction with B&B to learn contour labeling functions that work well for particular problem instances or classes of problems. Such a task is non-trivial, as it is unclear exactly how a learning algorithm could be trained, and how to avoid over-fitting the search strategy to a particular problem instance. Another approach here would be to use meta-optimization techniques to develop a contour labeling function for a problem; for instance, when a new subproblem is generated, an auxiliary MIP could be solved to determine the contour label for that subproblem, in the spirit of the local branching (Fischetti and Lodi, 2003) or backdoor branching (Fischetti and Monaci, 2011) algorithms.

Another approach towards answering this question would explore the relationship between the labeling function and the measure-of-best. By choosing an appropriate labeling function, CBFS can sometimes override the choice of a subproblem with a good value of μ in favor of a less-desirable subproblem. Understanding the interplay between these two axes of flexibility available to algorithm designers may provide insights into the performance of CBFS. In particular, are there any potential gains that can be made using a non-admissible or probabilistic μ (as in Shi and Ólafsson (2000) or Dür and Stix (2005)) with CBFS? Some interesting preliminary work has been done using a "measure-of-worst" function, which always explores the subproblem in the current contour with the worst value of μ, or even selecting a subproblem from the current contour at random.

A second open question raised by this dissertation is how ZDDs can more effectively be applied to B&P algorithms, particularly in cases where the full ZDD characterizing the pricing problem cannot be stored in memory. One potentially promising approach here involves the use of **approximate** ZDDs, described in Bergman et al. (2013). An approximate ZDD is a width-constrained ZDD that does not eliminate any valid solutions to the pricing problem, but may accept some invalid solutions. Given an approximate ZDD for a pricing problem, a post-generation step is required during column generation which checks the validity of the produced column, and queries the ZDD again if the column was found to be invalid.

This approach is somewhat challenging, as it is often difficult to determine how to build an approximate ZDD that does not discard any valid solutions to the pricing problem. An alternate approach instead identifies a set of "complicating" variables in the pricing problem, in the sense that their removal results

in a drastically smaller decision diagram. If a small set of complicating variables can be identified ahead of time, a brute-force algorithm could be applied at every iteration of column generation to identify the best assignment to these variables, and the ZDD could then be used to identify the best assignment to the remaining variables. As before, machine learning or meta-optimization techniques may be helpful in the identification of such a set.

A final open question raised by the discussion of wide branching with B&P algorithms is how branching affects algorithm performance. In particular, as observed in Chapter 5, the choice of branching rule can have significant impacts on algorithm performance, even when the search strategy and pruning rules are fixed. For example, suppose that for subproblems $S_1, S_2, \ldots, S_k$, it is possible to prune $S_2, S_3, \ldots, S_k$ immediately if S_1 is explored first, but otherwise the remaining subproblems must be explored. Then, naturally the branching rule that generates S_1 before any of the other subproblems will lead to faster performance. This effect is exacerbated in a B&P setting, where column generation must be performed at every subproblem before exploration or pruning can occur.

However, it is not known how to tailor the branching rules used to achieve the best algorithm performance; the heuristic rule for `B&P+Wide` combines knowledge from the LP structure and the graph structure together to moderate effect, but this rule does not generalize easily. It is likely that answering this question will rely on domain-specific or problem-instance-specific knowledge; however, answering it may provide significant benefits for some classes of problems.

Appendix A

Data Tables

This appendix provides the raw data for the computational experiments performed in Chapters 3-5. The following table presents a list of column headings and their meanings used in Tables A.1-A.7.

Table A.1 - CBFS results for MIPLIB benchmark problems	
min(CPX,BFS)	The best performance by either CPX or BFS
$\max_{\kappa_{p,n}}$	The worst performance over all 11 CBFS variants
$\min_{\kappa_{p,n}}$	The best performance over all 11 CBFS variants
$\text{mean}_{\kappa_{p,n}}$	The average performance over all 11 CBFS variants
$\arg\min_{\kappa_{p,n}}$	The best-performing contour labeling function parameters
Tables A.2 and A.3 - Comparison of ZDD construction algorithms and vertex orderings	
n, m	The number of vertices and edges in the instance
$\lvert\mathscr{C}\rvert$	The number of maximal independent sets in the instance, if known
$Z_{\mathscr{C}}$ size	The number of nodes in the maximal independent set ZDD for the instance
CPU time	The total time to construct $Z_{\mathscr{C}}$ for the instance, in CPU seconds
Table A.4 - `B&P+ZDD` results for DIMACS instances	
n, m	The number of vertices and edges in the instance
χ	The chromatic number of the instance, if known
LB, UB	Lower and upper bounds on χ produced by `B&P+ZDD`
Time ($Z_{\mathscr{C}}$)	CPU time needed for `B&P+ZDD` to build the ZDD
Time (B&P)	CPU time needed for `B&P+ZDD` to find and verify the chromatic number
exp/id	Number of subproblems explored and identified by `B&P+ZDD`
$Z_{\mathscr{C}}$ size (start,end)	The number of nodes in the ZDD at the beginning and end of the algorithm
% change	The change in size of the ZDD over the course of the algorithm
MMT	The CPU time taken by the MMT algorithm adjusted by the output of *dfmax*
Wide	The CPU time taken by the wide branching solver (Chapter 5)
CG-CP	The CPU time by CP-B&P (Gualandi and Malucelli, 2012) adjusted by *dfmax*
Tables A.5-A.7 - Results from `B&P+Wide` and `B&P+Deep` on DIMACS instances	
n, m	The number of vertices and edges in the instance
χ	The chromatic number of the instance, if known
LB, UB	Lower and upper bounds on χ produced by `B&P+ZDD`
Time	CPU time taken
TTB	CPU time to the best coloring
exp/id	Number of subproblems explored and identified by `B&P+ZDD`
$\lvert\mathbf{0}\rvert$	Number of times constrained pricing problem solved
$\lvert\text{cols}\rvert$	Total number of columns generated
$\lvert\text{cols}_0\rvert$	Number of columns generated from the constrained pricing problem
cols/sec	Number of columns generated per CPU seconds
$\max_{br}$	The maximum branching factor at any subproblem
avg_{br}	The average branching factor over all subproblems

Table A.1: A comparison of the 11 different CBFS variants on each of the 87 different MIPLIB benchmark instances. An entry of 'X' indicates that the search strategy did not solve the instance. The first column in each set shows the best performance for CPX or BFS; the next three show the maximum, minimum, and average performance for CBFS over all labeling functions. The final column shows which labeling function parameters performed best. 'ALL' indicates that all search strategies performed equally well.

	Iterations to best incumbent					Total tree size				
Instance	min(CPX,BFS)	$\max_{\kappa_{p,n}}$	$\min_{\kappa_{p,n}}$	$\text{mean}_{\kappa_{p,n}}$	$\arg\min_{\kappa_{p,n}}$	min(CPX,BFS)	$\max_{\kappa_{p,n}}$	$\min_{\kappa_{p,n}}$	$\text{mean}_{\kappa_{p,n}}$	$\arg\min_{\kappa_{p,n}}$
30n20b8	19000	82400	14000	37779	(3,1)	62977	886433	73358	224454	(1,0)
acc-tight5	470	1573	167	578	(1,-1)	471	1574	168	579	(1,-1)
aflow40b	36219	315062	9000	137467	(1,-3)	103413	320287	82637	176312	(1,-3)
air04	11	144	14	56	(1,-1)	103	251	118	172	(1,-1)
app1-2	5457	106797	186	21394	(1,-3)	5460	106798	449	21500	(3,-1)
ash608gpia-3col			infeasible			0	0	0	0	ALL
bab5	X	115500	70000	97180	(0,1)			time limit reached		
beasleyC3	10000	114000	9000	42364	(3,-1)	324601	537905	288829	397786	(1,-3)
biella1	36000	X	X	X	X	36582	X	X	X	X
bienst2	37069	84318	6298	32730	(1,1)	62677	135497	62033	81284	(0,1)
binkar10_1	1746	11970	3346	6763	(-1,1)	2881	12118	4553	7368	(-1,1)
bley_xl1	0	0	0	0	ALL	0	0	0	0	ALL
bnatt350	X	9042	9042	9042	(1,-1)	0	9043	9043	9043	(1,-1)
core2536-691	2891	X	X	X	X	4073	X	X	X	X
cov1075	100	1110	100	315	(1,0) (1,-3) (3,1) (3,-1)			time limit reached		
csched010	316000	X	X	X	X			time limit reached		
danoint	140	569600	63000	390225	(3,1)			time limit reached		
dfn-gwin-UUM	5857	100845	19000	60596	(1,0)	30180	128623	50659	89016	(1,1)
eil33-2	7529	12599	1870	7499	(0,1) (-1,1)	11796	13302	7898	10450	(1,0)
eilB101	3487	19600	110	13550	(1,0)	9222	22195	4981	16420	(1,0)
enlight13	23385	170934	4167	72278	(1,1)	807266	2015305	119110	750537	(1,0)
enlight14			infeasible			907744	597144	155441	300786	(1,0)
ex9	0	0	0	0	ALL	0	0	0	0	ALL
glass4	X	1304200	1304200	1304200	(3,-1)			time limit reached		
gmu-35-40	X	851900	851900	851900	(-1,3)			time limit reached		
iis-100-0-cov	0	X	X	X	X	317192	X	X	X	X
iis-bupa-cov	10080	180900	104500	137634	(1,1)			time limit reached		
iis-pima-cov	15499	108060	11000	58447	(1,0)	21610	108073	21207	62449	(1,0)
lectsched-4-obj	3000	164495	1397	35949	(1,-3)	3000	164496	1398	35949	(1,-3)
m100n500k4r1	4060	93723	2350	39032	(3,1)			time limit reached		
macrophage	4043	316800	180900	243500	(3,1)			time limit reached		
map18	260	1830	160	1067	(1,1)	693	2032	749	1315	(1,1)
map20	300	2570	30	1424	(1,3)	534	2684	494	1636	(1,3)
mcsched	144703	350004	50929	192530	(-3,1)	265059	702922	291866	447875	(0,1)
mik-250-1-100-1	100	159	100	111	(0,1) (-1,1) (3,1) (-1,3) (1,3) (-3,1)	581380	3525905	625019	1451031	(-1,1)
mine-166-5	3000	18190	9147	13022	(1,-1)	5091	18354	9978	13905	(1,-1)
mine-90-10	45000	240000	66590	138863	(1,-1)	83714	396794	91024	212456	(1,-1)
msc98-ip	X	7000	7000	7000	(0,1)			time limit reached		
mspp16	9	297	3	99	(1,-1) (1,0) (1,-3) (1,3) (3,-1)			time limit reached		
mzzv11	170	2033	190	1406	(3,1)	2067	2756	1344	2200	(3,1)
n3div36	X	429200	45000	259488	(3,1)			time limit reached		
n3seq24	96	5400	166	1552	(1,0)			time limit reached		

Table A.1: (con't) A comparison of the 11 different CBFS variants on the MIPLIB benchmark instances.

	Iterations to best incumbent					Total tree size				
Instance	min(CPX,BFS)	$\max_{\kappa_{p,n}}$	$\min_{\kappa_{p,n}}$	$\text{mean}_{\kappa_{p,n}}$	$\arg\min_{\kappa_{p,n}}$	min(CPX,BFS)	$\max_{\kappa_{p,n}}$	$\min_{\kappa_{p,n}}$	$\text{mean}_{\kappa_{p,n}}$	$\arg\min_{\kappa_{p,n}}$
n4-3	3450	160022	22920	82458	(0,1)	36079	165179	63821	99224	(0,1)
neos-1109824	2958	5807	47	1598	(1,1)	14814	42106	4452	15019	(1,1)
neos13	41	2972	1336	2137	(1,0)	1771	4048	2373	3004	(3,1)
neos-1337307	4934	41982	3540	17277	(1,0)	time limit reached				
neos-1396125	4307	55122	3364	18769	(-3,1)	6115	80519	29509	56013	(-3,1)
neos-1601936	14356	33100	20000	26550	(1,1)	14357	33100	20000	26550	(1,1)
neos18	0	0	0	0	ALL	3416	10213	3737	6468	(-1,1)
neos-476283	241	4248	359	2497	(1,1)	260	4271	455	2534	(1,1)
neos-686190	4213	34238	13931	23603	(1,0)	5565	35223	15434	24431	(1,0)
neos-849702	1076	140732	68161	104447	(3,1)	1077	140733	68162	104448	(3,1)
neos-916792	72473	161179	37407	94818	(-1,1)	114283	191313	88436	134276	(1,-1)
neos-934278	560	290	37	171	(1,-3)	time limit reached				
net12	60	5785	50	1320	(1,3)	1834	11048	1632	5416	(1,0)
netdiversion	60	470	9	87	(1,0) (1,-3) (1,-1) (3,-1)	93	483	30	111	(1,-1)
newdano	X	230900	163000	196950	(1,-1)	time limit reached				
noswot	680	13000	10000	11500	(3,1)	6971832	3050345	2199651	2624998	(3,1)
ns1208400	11220	X	X	X	X	30242	X	X	X	X
ns1688347	79	806	34	303	(1,-3)	86	825	53	337	(1,-3)
ns1758913	0	0	0	0	ALL	0	0	0	0	ALL
ns1766074	infeasible					829348	898690	866322	881103	(1,-1)
ns1830653	5146	74429	35189	51627	(-1,1)	15682	75686	38638	53464	(-1,1)
opm2-z7-s2	1130	12460	4030	7718	(-3,1)	1224	12638	4160	7873	(-3,1)
pg5_34	1147040	577470	577470	577470	(1,-1)	time limit reached				
pigeon-10	50	95	40	58	(-1,1) (-3,1)	5454064	5746066	5565966	5652175	(-1,1)
pw-myciel4	0	0	0	0	ALL	111259	638763	40895	175848	(1,-1)
qiu	4240	39360	120	10013	(3,1)	11669	70561	12820	27391	(-1,3)
rail507	16033	77200	6000	33904	(-1,1)	time limit reached				
ran16x16	15515	293077	73957	136397	(1,-1)	36652	304181	77495	149973	(1,1)
reblock67	622171	975000	473000	724000	(1,-1)	692781	986004	542826	764415	(1,-1)
rmatr100-p10	919	2327	170	941	(3,1)	1237	2342	1081	1389	(-3,1)
rmatr100-p5	683	382	34	197	(-1,1)	859	720	517	600	(-1,1)
rmine6	37000	102600	36970	62283	(-1,1)	time limit reached				
rocII-4-11	126554	316478	64610	137911	(1,0)	244865	320978	69947	176755	(1,0)
rococoC10-001000	47101	709720	21000	244048	(1,3)	47102	819412	36447	288663	(3,1)
roll3000	392312	1200900	530058	865479	(0,1)	434908	1242867	586962	914915	(0,1)
satellites1-25	9000	20450	1157	7327	(-1,1)	58562	62804	17144	35312	(-1,1)
sp98ic	62138	X	X	X	X	106982	X	X	X	X
sp98ir	2613	14750	3342	7461	(1,1)	4297	15286	4691	8002	(1,1)
tanglegram1	17	37	10	31	(-1,3)	33	55	41	48	(-3,1)
tanglegram2	14	39	13	23	(1,0)	39	55	27	46	(1,-1)
timtab1	2389267	2198646	1447034	1746179	(-1,3)	2493833	2489954	1874161	2338877	(0,1)
triptim1	0	0	0	0	ALL	0	0	0	0	ALL
unitcal_7	1000	2849	721	1539	(3,1)	1551	3915	1646	2440	(-1,1)
vpphard	8883	27000	7860	12177	(3,1)	time limit reached				
zib54-UUE	X	54600	2550	28575	(3,1)	0	55068	48570	51819	(3,1)

Table A.2: A comparison of the recursive ZDD construction algorithm (Algorithm 4.3) and the bottom-up variant of Algorithm 4.6 for a subset of the DIMACS graph coloring database. Grey cells indicate the faster algorithm, or the algorithm that generated more nodes if both took more than 2 hours. Instances for which the algorithms were unable to construct the ZDD in the two-hour time limit report the number of nodes inserted at termination in columns 5 and 7.

				Recursive alg.		Merging alg.	
Instance	n	m	$\lvert\mathscr{C}\rvert$	$Z_{\mathscr{C}}$ size	CPU time	$Z_{\mathscr{C}}$ size	CPU time
DSJC125.5	125	3891	43268	48328	0.61	48328	14.55
DSJC125.9	125	6961	524	623	0.01	623	0.02
DSJC250.5	250	15668	1470363	1476916	34.50	>1153590	>2hrs
DSJC250.9	250	27897	2580	2893	0.04	2893	0.36
DSJC500.5	500	62624	91664597	83467418	3818.16	>1057125	>2hrs
DSJC500.9	500	112437	14560	15397	0.39	15397	7.03
DSJC1000.5	1000	249826	?	$>10^8$	6937.39	>716976	>2hrs
DSJC1000.9	1000	449449	100389	102909	5.77	102909	184.60
DSJR500.1	500	3555	?	>72383	>2hrs	>592268	>2hrs
DSJR500.1c	500	121275	643	2443	0.15	2443	0.46
DSJR500.5	500	58862	38551855	1809872	711.83	>1057125	>2hrs
queen8_8	64	728	10188	9951	0.08	9951	1.35
queen8_12	96	1368	334806	221524	3.42	221524	476.55
queen9_9	81	1056	57600	50746	0.60	50746	32.44
queen10_10	100	2940	376692	295493	5.27	295493	1078.65
queen11_11	121	3960	2640422	1870782	42.95	>778484	>2hrs
queen12_12	144	5192	19469324	12443637	368.98	>777125	>2hrs
queen13_13	169	6656	151978440	88885235	3351.61	>709664	>2hrs
queen14_14	196	8372	?	$>10^8$	3795.47	>673151	>2hrs
queen15_15	225	10360	?	$>10^8$	3740.66	>664525	>2hrs
queen16_16	256	12640	?	$>10^8$	3972.67	>564070	>2hrs
myciel3	11	23	16	29	0	29	0.00
myciel4	20	71	79	152	0	152	0.00
myciel5	47	236	857	1429	0	1429	0.04
myciel6	95	755	49049	40191	0.99	40191	18.96
myciel7	191	2360	75511755	7191878	2635.97	>317430	>2hrs
1-Insertions_4	67	232	56641	85122	0.93	85122	155.42
3-Insertions_3	56	110	228439	81050	2.46	81050	146.24
4-Insertions_3	79	156	37833929	5518516	430.29	>522455	>2hrs
1-FullIns_4	93	593	129042	137761	8.78	137761	881.28
2-FullIns_3	52	201	15966	7975	0.25	7975	3.26
3-FullIns_3	80	346	1454750	363408	55.76	>251538	>2hrs
4-FullIns_3	114	541	?	>4527300	>2hrs	>250095	>2hrs
5-FullIns_3	154	792	?	>6042790	>2hrs	>212485	>2hrs
fpsol2.i.1	496	11654	1.67×10^{14}	>521	>2hrs	3969	1.59
fpsol2.i.2	451	8691	8.49×10^{18}	>174	>2hrs	16923	8.51
fpsol2.i.3	425	8688	7.43×10^{18}	>375	>2hrs	17405	8.73
latin_square_10	900	307350	30240	52742	35.93	52742	134.38
school1	385	19	?	>608862	>2hrs	>93887	>2hrs
school1_nsh	352	14612	?	>2148287	>2hrs	>473049	>2hrs
mulsol.i.1	197	3925	98404	644	1.37	664	0.04
mulsol.i.2	188	3885	2669597327	>1650	>2hrs	2021	0.15
mulsol.i.3	184	3916	2669597327	>1663	>2hrs	2029	0.16
mulsol.i.4	185	3946	4650922127	>1665	>2hrs	2033	0.15
mulsol.i.5	185	3973	3330038927	>1828	>2hrs	2196	0.17
miles250	128	774	?	>275	>2hrs	>7552	>2hrs
miles500	128	2340	?	>53123	>2hrs	>85851	>2hrs
miles750	128	4226	33208742	6112	111.8	6112	1.39
miles1000	128	6342	775281	8520	4.64	8520	1.80
miles1500	128	10396	7802	1695	0.06	1695	0.07
anna	138	986	?	>9316	>2hrs	23296	4.52
david	87	812	44149508	6901	174.35	6901	2.53
jean	80	508	1251960	1360	2.33	1360	0.06
huck	74	602	7272300	283	15.25	27	0.01
zeroin.i.1	211	4100	79170	731	0.66	731	0.05
zeroin.i.2	211	3541	18189098	1114	138.59	1114	0.11
zeroin.i.3	206	3540	12912650	1112	111.29	1112	0.12

Table A.3: A comparison of the maximal independent set ZDD size with the vertex orderings from Section 4.3.3. Grey cells indicate the smallest ordering. Not shown are the reverse degree ordering and the forward degeneracy list, since they did not outperform other orderings for any instances.

	Degree List		Rev. Degeneracy Ordering		Clique Cover		Maximal Path Decomp.		Random	
Instance	$Z_{\mathscr{C}}$ size	CPU time	$Z_{\mathscr{C}}$ size	CPU time	$Z_{\mathscr{C}}$ size	CPU time	$Z_{\mathscr{C}}$ size	CPU time	$Z_{\mathscr{C}}$ size	CPU time
DSJC125.5	47183	0.50	44809	0.51	45559	0.55	48328	0.61	46445	0.55
DSJC125.9	627	0.00	622	0.01	607	0.00	623	0.01	627	0.01
DSJC250.5	1416093	27.69	1419868	28.89	1410240	30.40	1476916	34.50	1447760	30.53
DSJC250.9	2880	0.04	2852	0.04	2866	0.07	2893	0.04	2913	0.04
DSJC500.5	79745416	3054.26	80620093	3290.64	80307678	3378.64	83467418	3818.16	81549931	3310.20
DSJC500.9	15252	0.39	15271	0.37	15279	0.54	15397	0.39	15329	0.39
DSJC1000.5	$>10^8$	6371.72	78359919	>2hrs	$>10^8$	6652.68	$>10^8$	6937.39	87381935	>2hrs
DSJC1000.9	102060	5.34	102584	5.12	102368	6.67	102909	5.77	102451	5.96
DSJR500.1c	2437	0.15	2085	0.13	2012	0.24	2443	0.15	2138	0.14
DSJR500.5	612313	313.21	1832145	308.96	1188614	373.67	1809872	711.83	2169788	380.22
queen8_8	9611	0.08	9689	0.09	9341	0.09	9951	0.08	11056	0.09
queen8_12	209878	3.35	220198	3.06	169104	3.31	221524	3.42	253668	2.97
queen9_9	47591	0.55	49159	0.56	46899	0.60	50746	0.60	61160	0.54
queen10_10	280640	4.35	281622	4.35	272460	4.76	295493	5.27	345768	4.34
queen11_11	1715945	44.24	1728852	35.72	1678570	38.29	1870782	42.95	2223191	35.18
queen12_12	11474606	322.40	11488955	310.23	11097792	379.80	12443637	368.98	15285013	290.45
queen13_13	80618162	2791.45	79625893	3093.27	77364686	3228.99	88885235	3351.61	$>10^8$	2277.94
queen14_14	$>10^8$	3412.03	$>10^8$	4122.76	$>10^8$	4280.20	$>10^8$	3795.47	$>10^8$	2346.78
queen15_15	$>10^8$	3355.18	$>10^8$	4795.93	$>10^8$	4176.67	$>10^8$	3740.66	$>10^8$	2530.21
queen16_16	$>10^8$	3817.44	$>10^8$	5416.81	$>10^8$	4913.85	$>10^8$	3972.67	$>10^8$	2712.34
myciel3	38	0.00	30	0.00	33	0.00	29	0.00	35	0.00
myciel4	230	0.00	176	0.00	210	0.00	152	0.00	202	0.00
myciel5	3435	0.02	1734	0.03	2948	0.02	1429	0.00	3479	0.03
myciel6	221879	4.33	97346	4.38	193969	4.15	40191	0.99	177746	10.68
myciel7	$>10^8$	5461.37	2297922	>2hrs	$>10^8$	6644.03	7191878	2635.97	19031053	>2hrs
1-Insertions_4	227317	2.04	58411	2.02	85456	1.61	85122	0.93	181447	4.38
3-Insertions_3	592429	5.42	32975	3.52	52489	2.70	81050	2.46	260534	4.84
4-Insertions_3	43549657	1322.11	586435	665.40	773005	544.35	5518516	430.29	22883116	1606.48
1-FullIns_4	294052	5.80	142075	7.67	304315	8.96	137761	8.78	402067	19.97
2-FullIns_3	11759	0.18	11161	0.27	20161	0.27	7975	0.25	20473	0.37
3-FullIns_3	814178	28.22	739391	49.65	1206277	36.32	363408	55.76	1420292	61.18
latin_square_10	93203	27.72	110236	47.06	113209	34.22	52742	35.93	162995	32.92
mulsol.i.1	1012	18.40	804	17.68	1447	21.63	644	1.37	63963	15.20
miles750	22994	146.39	21626	172.69	19078	160.43	6112	111.80	463524	160.68
miles1000	4435	3.09	9618	3.28	4367	3.20	8520	4.64	62315	3.12
miles1500	717	0.04	1584	0.03	991	0.03	1695	0.06	2402	0.05
david	33493	1279.88	23687	1753.32	11057	1252.25	6901	174.35	608664	727.55
jean	6534	46.41	6471	75.11	3760	50.32	1360	2.33	80955	29.73
huck	4553	69.90	2478	182.87	2576	78.13	283	15.25	56630	63.37
zeroin.i.1	1129	16.68	830	21.84	2504	21.02	731	0.66	116632	17.09
zeroin.i.2	87372	3556.30	3297	4646.50	68695	4178.08	1114	138.59	6921096	6342.30
zeroin.i.3	79356	2323.45	3227	3339.63	58670	2858.85	1112	111.29	2110951	2245.66

Table A.4: A comparison of `B&P+ZDD` to other algorithms in the literature. Cells highlighted in grey show the fastest algorithm. Entries labeled "init" indicate that the initial upper bound equaled the root lower bound; cells labels "oom" indicate that the algorithm ran out of memory. All running times are scaled based on the reported values of the *dfmax* benchmarking utility.

Instance	n	m	χ	LB	UB	Time ($Z_{\mathscr{C}}$)	Time (B&P)	exp/id	$Z_{\mathscr{C}}$ size (start)	$Z_{\mathscr{C}}$ size (end)	% change	MMT	Wide	CG-CP
DSJC125.5	125	3891	17	16	17	0.47	284.92	599/1199	48367	52207	7.9	17019.33	225.21	14372.75
DSJC125.9	125	6961	44	43	44	0	0.21	55/111	623	627	0.6	3674.22	1.02	33.23
DSJC250.5	250	15668	?	26	28	29.55	>10hrs	5122/10244	1476916	1511171	2.3	>10hrs	>10hrs	>10hrs
DSJC250.9	250	27897	72	71	72	0.04	649.9	16411/32823	2893	2960	2.3	>10hrs	>10hrs	>10hrs
DSJC500.5	500	62624	?	43	53	3458.5	>10hrs	25/50	83507135	83509619	0.003	>10hrs	>10hrs	-
DSJC500.9	500	112437	?	123	129	0.38	>10hrs	46634/93268	15397	15913	3.4	>10hrs	>10hrs	>10hrs
DSJC1000.5	1000	249826	?	10	108	6608.86	oom	0	partial	-	-	>10hrs		-
DSJC1000.9	1000	449449	?	215	228	5.64	>10hrs	6466/12932	102909	105366	2.4	>10hrs	>10hrs	-
DSJR500.1c	500	121275	85	85	85	0.14	0.1	18/37	2443	2447	0.2	272.01	1.29	0.60
DSJR500.5	500	58862	122	122	122	689.48	102.75	74/149	1809872	2222124	22.8	322.6	6862.28	>10hrs
le450_25c	450	17343	25	25	28	19889.92	oom	0	partial	-	-	init	>10hrs	-
le450_25d	450	17425	25	25	28	16262.45	oom	0	partial	-	-	init	>10hrs	-
queen9_9	81	1056	10	9	10	0.44	9.33	11/23	50719	54727	7.9	34.51	20.48	74
queen10_10	100	2940	11	10	11	4.09	140.76	76/153	295500	308571	4.4	647.65	587.32	26393.2
queen11_11	121	3960	11	11	11	33.1	14354.16	1902/3805	1867378	1923347	3	1759.08	19208.45	21858.5
queen12_12	144	5192	12	12	13	297.91	>10hrs	1224/2448	12426874	12526852	0.8	>10hrs	>10hrs	-
queen13_13	169	6656	13	13	14	2739.39	>10hrs	98/195	88797420	88874820	0.09	>10hrs	>10hrs	-
queen14_14	196	8372	14	14	15	3139.2	>10hrs	3/6	partial	-	-	>10hrs	>10hrs	-
queen15_15	225	10360	15	15	16	3235.39	>10hrs	0	partial	-	-	>10hrs	>10hrs	-
queen16_16	256	12640	16	16	18	3527.63	>10hrs	0	partial	-	-	>10hrs	>10hrs	-
myciel3	11	23	4	3	4	0	0	4/9	29	29	0	0.00	0.01	-
myciel4	20	71	5	4	5	0	0.09	191/383	152	169	11.2	111.26	0.47	-
myciel5	47	236	6	4	6	0.01	392.21	160622/321245	1429	2188	53.1	-	3207.63	-
myciel6	95	755	7	4	7	0.76	>10hrs	94395/188789	40191	70326	75.0	>10hrs	>10hrs	-
myciel7	191	2360	8	5	8	2198	>10hrs	7816/15632	7191878	7344883	2.1	>10hrs	>10hrs	-
1-Insertions_4	67	232	5	3	5	0.69	>10hrs	72106/144211	85112	106550	25.2	>10hrs	>10hrs	-
1-Insertions_5	202	1227	?	2	6	23708.86	>10hrs	0/0	partial	-	-	>10hrs	>10hrs	-
2-Insertions_4	149	541	?	2	5	>10hrs	-	-	-	-	-	>10hrs	>10hrs	-
3-Insertions_3	56	110	4	3	4	1.8	>10hrs	29128/58256	94885	183834	93.7	>10hrs	>10hrs	-
3-Insertions_4	281	1046	?	2	5	22684.73	>10hrs	0/0	partial	-	-	>10hrs	>10hrs	-
4-Insertions_3	79	156	4	3	4	314.85	>10hrs	14177/28353	6585989	6693239	1.6	>10hrs	>10hrs	-
1-FullIns_4	93	593	5	4	5	8.76	122.58	7422/14845	148275	155237	4.7	>10hrs	>10hrs	-
1-FullIns_5	282	3247	6	3	6	>10hrs	-	-	-	-	-	>10hrs	>10hrs	-
2-FullIns_4	212	1621	6	0	0	>10hrs	-	-	-	-	-	>10hrs	>10hrs	-
2-FullIns_5	852	12201	7	4	7	>10hrs	-	-	-	-	-	>10hrs	>10hrs	-
3-FullIns_4	405	3524	7	5	7	>10hrs	-	-	-	-	-	>10hrs	>10hrs	-
4-FullIns_4	690	6650	8	0	0	>10hrs	-	-	-	-	-	>10hrs	>10hrs	-
latin_square_10	900	307350	?	90	100	36.4	>10hrs	5265/10529	52807	52807	0	>10hrs	>10hrs	-
qg.order30	900	26100	30	0	0	>10hrs	-	-	-	-	-	0.19	>10hrs	-
wap06	947	43571	40	0	0	>10hrs	-	-	-	-	-	165	>10hrs	-
r250.5	250	14849	65	65	65	14.63	7.15	25/51	137683	266893	93.8	-	-	6.8
r1000.1c	1000	485090	?	96	98	2.29	>10hrs	215082/430163	11762	11997	2	-	-	>10hrs
r1000.5	1000	238267	234	234	234	43020.09	4690.16	192/385	37664084	38165484	1.3	-	-	>10hrs
flat300_28_0	300	21695	28	28	28	144.57	19883.45	1004/2009	5817662	5858003	0.7	-	-	>10hrs
flat1000_50_0	1000	245000	?	10	106	6795.93	>10hrs	0/0	partial	-	-	-	-	>10hrs
flat1000_60_0	1000	245830	?	10	105	6368.64	>10hrs	0/0	partial	-	-	-	-	>10hrs
flat1000_76_0	1000	246708	?	12	106	6628.52	>10hrs	0/0	partial	-	-	-	-	>10hrs

Table A.5: Comparison of the wide and deep branch-and-price solvers (Chapter 5) with the MMT branch-and-price solver. Grey cells indicate the solver with the fastest time to verify an optimal solution. "oom" indicates the program ran out of memory before the time limit was reached.

Graph Info				Initialization		MMT			Wide					Deep (DFS)			
Name	n	m	χ	TTB	Time	LB	UB	Time	LB	UB	Time	TTB	exp/id	UB	t	TTB	exp/id
DSJC125.5	125	3891	17	361.16	6100	17	17	18050.8	16	17	225.21	*init*	67/473	17	101.50	*init*	58/117
DSJC125.9	125	6961	44	20.06	6100	44	44	3896.9	43	44	1.74	*init*	49/190	44	1.25	*init*	58/117
DSJC250.5	250	15668	?	223.7	6100	20	28	$> 10hrs$	26	29	$> 10hrs$	*init*	334/2638	29	$> 10hrs$	*init*	6027/12053
DSJC250.9	250	27897	72	3032.73	6100	71	72	$> 10hrs$	71	72	$> 10hrs$	83.54	53220/238887	72	19733.62	17168.35	79423/158847
DSJC500.5	500	62624	?	1301.9	6100	16	48	$> 10hrs$	43	50	$> 10hrs$	*init*	1/10	50	$> 10hrs$	*init*	1/1
DSJC500.9	500	112437	?	2185.1	6100	123	127	$> 10hrs$	123	128	$> 10hrs$	3204.31	11909/57010	129	$> 10hrs$	314.98	39684/79368
DSJC1000.9	1000	449449	?	5015.35	6100	51	224	$> 10hrs$	215	228	$> 10hrs$	14798.83	1106/6005	229	$> 10hrs$	8681.37	5632/11263
DSJR500.1c	500	121275	85	1970.62	6100	85	85	288.5	85	85	1.29	1.29	8/27	85	1.03	0.94	12/25
DSJR500.5	500	58862	122	582.62	6100	122	122	342.2	122	122	6862.28	6862.27	78/341	122	7211.25	7209.48	82/165
le450_25c	450	17343	25	96.89	6100	25	25	*init*	25	27	$> 10hrs$	*init*	1/1	27	$> 10hrs$	*init*	1/1
le450_25d	450	17425	25	12.44	6100	25	25	*init*	25	27	$> 10hrs$	*init*	1/1	27	$> 10hrs$	*init*	1/1
queen9_9	81	1056	10	0.36	3	10	10	36.6	9	10	20.48	*init*	9/48	10	32	*init*	11/23
queen10_10	100	2940	11	1.55	3	11	11	686.9	10	11	587.32	*init*	44/335	11	1181.18	*init*	70/141
queen11_11	121	3960	11	0.26	3	11	11	1865.7	11	12	19210.3	19210.3	409/3658	12	$> 10hrs$	303.53	341/682
queen12_12	144	5192	12	1.07	3	12	13	$> 10hrs$	12	13	$> 10hrs$	1604.25	469/4740	13	$> 10hrs$	1102.35	55/110
queen13_13	169	6656	13	15.42	100	13	14	$> 10hrs$	13	14	$> 10hrs$	*init*	59/611	14	$> 10hrs$	*init*	11/22
queen14_14	196	8372	14	29.19	100	14	15	$> 10hrs$	14	15	$> 10hrs$	*init*	10/105	15	$> 10hrs$	*init*	7/14
queen15_15	225	10360	15	8.74	100	15	16	$> 10hrs$	15	16	$> 10hrs$	*init*	1/1	16	$> 10hrs$	*init*	1/1
queen16_16	256	12640	16	18.99	100	16	17	$> 10hrs$	16	17	$> 10hrs$	*init*	1/1	17	$> 10hrs$	*init*	1/1
myciel3	11	23	4	0	3	3	4	0	3	4	0.01	*init*	3/11	4	0	*init*	4/9
myciel4	20	71	5	0	3	4	5	118	4	5	0.47	0.46	83/360	5	0.81	*init*	206/413
myciel5	47	236	6	0	3	4	6	$> 10hrs$	4	6	3207.63	324.82	26085/111751	6	31967.88	*init*	86256/172513
myciel6	95	755	7	0	3	4	7	$> 10hrs$	4	7	$> 10hrs$	*init*	79/917	7	$> 10hrs$	*init*	21/41
myciel7	191	2360	8	0.01	3	5	8	$> 10hrs$	5	8	$> 10hrs$	*init*	38/609	8	$> 10hrs$	*init*	1/1
1-Insertions_4	67	232	5	0	3	3	5	$> 10hrs$	3	5	$> 10hrs$	*init*	659/8123	5	$> 10hrs$	*init*	652/1303
1-Insertions_5	202	1227	?	0	3	3	6	$> 10hrs$	3	6	$> 10hrs$	*init*	3/105	6	$> 10hrs$	*init*	3/5
2-Insertions_4	149	541	?	0	3	3	5	$> 10hrs$	3	5	$> 10hrs$	*init*	10/435	5	$> 10hrs$	*init*	2/3
3-Insertions_3	56	110	4	0	3	3	4	$> 10hrs$	3	4	$> 10hrs$	*init*	485/10198	4	10902.43(oom)	*init*	3687/7375
3-Insertions_4	281	1046	?	0	3	3	5	$> 10hrs$	3	5	$> 10hrs$	*init*	1/1	5	$> 10hrs$	*init*	1/1
4-Insertions_3	79	156	4	0	3	3	4	$> 10hrs$	3	4	$> 10hrs$	*init*	212/6934	4	$> 10hrs$	*init*	434/868
1-FullIns_4	93	593	5	0	3	4	5	$> 10hrs$	4	5	$> 10hrs$	*init*	4/36	5	$> 10hrs$	*init*	20/39
1-FullIns_5	282	3247	6	0	3	4	6	$> 10hrs$	4	6	12098.11(oom)	*init*	4/36	6	$> 10hrs$	*init*	2/3
2-FullIns_4	212	1621	6	0	3	5	6	$> 10hrs$	5	6	$> 10hrs$	*init*	1/1	6	$> 10hrs$	*init*	1/1
2-FullIns_5	852	12201	7	0.01	3	5	7	$> 10hrs$	5	7	$> 10hrs$	*init*	3/25	7	$> 10hrs$	*init*	1/1
3-FullIns_4	405	3524	7	0.01	3	6	7	$> 10hrs$	6	7	$> 10hrs$	*init*	1/1	7	$> 10hrs$	*init*	1/1
4-FullIns_4	690	6650	8	0.01	3	7	8	$> 10hrs$	7	8	$> 10hrs$	*init*	1/1	8	$> 10hrs$	*init*	1/1
latin_square_10	900	307350	?	3560.94	6100	90	102	$> 10hrs$	90	100	$> 10hrs$	11058.5	880/6343	101	$> 10hrs$	2686.27	6453/12906
qg.order30	900	26100	30	0.03	3	30	30	*init*	30	31	$> 10hrs$	*init*	1/1	31	$> 10hrs$	*init*	1/1
wap06	947	43571	40	54.9	170	40	40	*init*	40	43	$> 10hrs$	*init*	1/1	43	$> 10hrs$	*init*	1/1

Table A.6: An analysis of the wide branching solver for selected graph coloring instances. “oom” indicates that the solver ran out of memory.

Name	n	m	χ	UB	Time	TTB	exp/id	$\lvert\mathbf{0}\rvert$	$\lvert$cols$\rvert$	$\lvert$cols$_0\rvert$	cols/sec	max$_{br}$	avg$_{br}$
DSJC125.5	125	3891	17	17	224.00	*init*	67/473	20	1021	64	4.56	11	6.04
DSJC125.9	125	6961	44	44	1.84	*init*	49/190	3	108	31	58.70	5	2.86
DSJC250.5	250	15668	?	29	$> 10hrs$	*init*	334/2638	69	14095	721	0.39	14	6.90
DSJC250.9	250	27897	72	72	$> 10hrs$	83.54	53220/238887	2124	734	159	0.02	10	3.49
DSJC500.5	500	62624	?	50	$> 10hrs$	*init*	1/10	0	871	2	0.02	12	12.00
DSJC500.9	500	112437	?	128	$> 10hrs$	3190.80	11909/57010	132	4900	1058	0.14	11	3.79
DSJC1000.9	1000	449449	?	228	$> 10hrs$	14257.98	1106/6005	24	9055	1516	0.25	24	4.43
DSJR500.1c	500	121275	85	85	1.09	1.09	8/27	0	63	10	57.80	3	2.25
DSJR500.5	500	58862	122	122	6860.51	6860.51	78/341	0	55	2	0.01	6	3.36
le450_25c	450	17343	25	27	$> 10hrs$	*init*	1/1	0	0	0	0	6	6.00
le450_25d	450	17425	25	27	$> 10hrs$	*init*	1/1	0	0	0	0	6	6.00
queen9_9	81	1056	10	10	20.36	*init*	9/48	5	947	48	46.51	7	4.22
queen10_10	100	2940	11	11	586.86	*init*	44/335	13	6280	237	10.70	10	6.59
queen11_11	121	3960	11	11	19208.45	1117.60	409/3658	256	40162	1819	2.09	23	7.94
queen12_12	144	5192	12	13	$> 10hrs$	1599.44	469/4740	444	30769	1110	0.85	18	9.13
queen13_13	169	6656	13	14	$> 10hrs$	*init*	59/611	48	9294	57	0.26	16	9.41
queen14_14	196	8372	14	15	$> 10hrs$	*init*	10/105	2	4440	34	0.12	14	10.50
queen15_15	225	10360	15	16	$> 10hrs$	*init*	1/1	0	0	0	0	8	8.00
queen16_16	256	12640	16	17	$> 10hrs$	*init*	1/1	0	0	0	0	13	13.00
myciel3	11	23	4	4	0.01	0.01	3/11	2	1	0	–	3	2.33
myciel4	20	71	5	5	0.47	0.46	83/360	53	1	1	2.13	9	3.33
myciel5	47	236	6	6	3207.61	324.80	26085/111751	20751	344	340	0.11	17	3.28
myciel6	95	755	7	7	$> 10hrs$	*init*	79/917	47	590	462	0.02	27	10.67
myciel7	191	2360	8	8	$> 10hrs$	*init*	38/609	14	1205	743	0.03	48	15.29
1-Insertions_4	67	232	5	5	$> 10hrs$	*init*	7706/104811	7019	6982	6750	0.19	28	11.33
1-Insertions_5	202	1227	?	6	$> 10hrs$	*init*	3/105	0	2504	158	0.07	68	42.67
2-Insertions_4	149	541	?	5	$> 10hrs$	*init*	10/435	4	1394	234	0.04	72	46.10
3-Insertions_3	56	110	4	4	$> 10hrs$	*init*	485/10198	484	9667	9520	0.27	27	20.08
3-Insertions_4	281	1046	?	5	$> 10hrs$	*init*	1/1	0	2662	1	0.07	113	113.00
4-Insertions_3	79	156	4	4	$> 10hrs$	*init*	212/6934	211	6925	6796	0.19	47	31.85
1-FullIns_4	93	593	5	5	$> 10hrs$	*init*	147/921	109	640	538	0.02	10	5.31
1-FullIns_5(oom)	282	3247	6	6	12098.11	*init*	4/36	–	–	–	–	–	–
2-FullIns_4	212	1621	6	6	$> 10hrs$	*init*	1/1	0	0	0	0	4	4.00
2-FullIns_5	852	12201	7	7	$> 10hrs$	*init*	3/25	0	252	10	0.01	12	10.00
3-FullIns_4	405	3524	7	7	$> 10hrs$	*init*	1/1	0	1	0	0.00	3	3.00
4-FullIns_4	690	6650	8	8	$> 10hrs$	*init*	1/1	0	3	0	0.00	4	4.00
latin_square_10	900	307350	?	100	$> 10hrs$	11052.21	880/6343	206	3857	445	0.11	13	6.21
qg.order30	900	26100	30	31	$> 10hrs$	*init*	1/1	0	0	0	0	16	16.00
wap06	947	43571	40	43	$> 10hrs$	*init*	1/1	0	0	0	0	10	10.00

Table A.7: An analysis of the deep branching solver for selected graph coloring problems. "oom" indicates that the solver ran out of memory.

Name	n	m	χ	UB	Time	TTB	exp/id	$\lvert\mathbf{0}\rvert$	|cols|	cols/sec
DSJC125.5	125	3891	17	17	101.50	*init*	58/117	56	410	4.04
DSJC125.9	125	6961	44	44	1.25	*init*	58/117	54	88	70.4
DSJC250.5	250	15668	?	29	$> 10hrs$	*init*	6027/12053	6007	5909	0.16
DSJC250.9	250	27897	72	72	19733.62	17168.35	79423/158847	79403	740	0.4
DSJC500.5	500	62624	?	50	$> 10hrs$	*init*	1/1	0	302	0.01
DSJC500.9	500	112437	?	129	$> 10hrs$	314.98	39684/79368	39604	1841	0.05
DSJC1000.9	1000	449449	?	229	$> 10hrs$	8681.37	5632/11263	5462	3839	0.11
DSJR500.1c	500	121275	85	85	1.03	0.94	12/25	2	42	40.78
DSJR500.5	500	58862	122	122	7211.25	7209.48	82/165	0	0	0
le450_25c	450	17343	25	27	$> 10hrs$	*init*	1/1	0	0	0
le450_25d	450	17425	25	27	$> 10hrs$	*init*	1/1	0	0	0
queen9_9	81	1056	10	10	31.88	*init*	11/23	9	321	10.06
queen10_10	100	2940	11	11	1180.72	*init*	70/141	69	1740	1.47
queen11_11	121	3960	11	12	$> 10hrs$	303.53	341/682	332	10746	0.30
queen12_12	144	5192	12	13	$> 10hrs$	1102.35	55/110	44	4462	0.12
queen13_13	169	6656	13	14	$> 10hrs$	*init*	11/22	6	929	0.03
queen14_14	196	8372	14	15	$> 10hrs$	*init*	7/14	1	796	0.02
queen15_15	225	10360	15	16	$> 10hrs$	*init*	1/1	0	0	0
queen16_16	256	12640	16	17	$> 10hrs$	*init*	1/1	0	0	0
myciel3	11	23	4	4	0.00	*init*	4/9	3	0	–
myciel4	20	71	5	5	0.81	*init*	206/413	204	6	7.41
myciel5	47	236	6	6	31967.88	*init*	86256/172513	86253	417	0.01
myciel6	95	755	7	7	$> 10hrs$	*init*	21/41	17	24	0.001
myciel7	191	2360	8	8	$> 10hrs$	*init*	1/1	0	40	0.001
1-Insertions_4	67	232	5	5	$> 10hrs$	*init*	652/1303	650	508	0.01
1-Insertions_5	202	1227	?	6	$> 10hrs$	*init*	3/5	0	468	0.01
2-Insertions_4	149	541	?	5	$> 10hrs$	*init*	2/3	0	364	0.01
3-Insertions_3(oom)	56	110	4	4	$> 10hrs$	10902.43	3687/7375	–	–	–
3-Insertions_4	281	1046	?	5	$> 10hrs$	*init*	1/1	0	2468	0.07
4-Insertions_3	79	156	4	4	$> 10hrs$	*init*	434/868	433	596	0.01
1-FullIns_4	93	593	5	5	$> 10hrs$	*init*	20/39	19	41	0.001
1-FullIns_5	282	3247	6	6	$> 10hrs$	*init*	2/3	0	10	0.00
2-FullIns_4	212	1621	6	6	$> 10hrs$	*init*	1/1	0	0	0
2-FullIns_5	852	12201	7	7	$> 10hrs$	*init*	1/1	0	31	0.001
3-FullIns_4	405	3524	7	7	$> 10hrs$	*init*	1/1	0	2	0.00
4-FullIns_4	690	6650	8	8	$> 10hrs$	*init*	1/1	0	11	0.00
latin_square_10	900	307350	?	101	$> 10hrs$	2686.27	6453/12906	6362	672	0.02
qg.order30	900	26100	30	31	$> 10hrs$	*init*	1/1	0	0	0
wap06	947	43571	40	43	$> 10hrs$	*init*	1/1	0	0	0

References

T. Achterberg. Forum post, 2009. URL `https://www.ibm.com/developerworks/community/forums/html/topic?id=77777777-0000-0000-0000-000014396633`. Last accessed March 2014.

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

T. Achterberg, T. Berthold, T. Koch, and K. Wolter. Constraint integer programming: A new approach to integrate CP and MIP. In L. Perron and M. A. Trick, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, number 5015 in Lecture Notes in Computer Science, pages 6–20. Springer Berlin Heidelberg, Jan 2008.

S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, 100(6):509–516, 1978.

H. R. Andersen. An introduction to binary decision diagrams, 1997. URL `www.itu.dk/courses/AVA/E2005/bdd-eap.pdf`. Last accessed March 2014.

S. Arora, B. Bollobas, and L. Lovász. Proving integrality gaps without knowing the linear program. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings*, pages 313–322, 2002.

A. Atamtürk and M. W. P. Savelsbergh. Integer-programming software systems. *Annals of Operations Research*, 140(1):67–124, Nov 2005.

P. Avella, M. Boccia, and I. Vasilyev. A computational study of exact knapsack separation for the generalized assignment problem. *Computational Optimization and Applications*, 45(3):543–555, 2010.

L. Babel. A fast algorithm for the maximum weight clique problem. *Computing*, 52(1):31–38, Mar 1994.

E. Balas. Disjunctive programming. In P. L. Hammer, E. Johnson, and B. Korte, editors, *Annals of Discrete Mathematics*, volume 5 of *Discrete Optimization II Proceedings of the Advanced Research Institute on Discrete Optimization and Systems Applications of the Systems Science Panel of NATO and of the Discrete Optimization Symposium co-sponsored by IBM Canada and SIAM Banff, Aha. and Vancouver*, pages 3–51. Elsevier, 1979.

E. Balas and M. Perregaard. A precise correspondence between lift-and-project cuts, simple disjunctive cuts, and mixed integer gomory cuts for 0-1 programming. *Mathematical Programming*, 94(2-3):221–245, Jan 2003.

E. Balas, S. Ceria, and G. Cornuéjols. A lift-and-project cutting plane algorithm for mixed 0–1 programs. *Mathematical Programming*, 58(1-3):295–324, Jan 1993.

E. Balas, S. Ceria, and G. Cornuéjols. Mixed 0-1 programming by lift-and-project in a branch-and-cut framework. *Management Science*, 42(9):1229–1246, Sep 1996a.

E. Balas, S. Ceria, G. Cornuéjols, and N. Natraj. Gomory cuts revisited. *Operations Research Letters*, 19 (1):1–9, Jul 1996b.

C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance. Branch-and-price: Column generation for solving huge integer programs. *Operations Research*, 46(3):316–329, May 1998.

C. Barnhart, C. A. Hane, and P. H. Vance. Using branch-and-price-and-cut to solve origin-destination integer multicommodity flow problems. *Operations Research*, 48(2):318–326, 2000.

C. Barnhart, A. M. Cohn, E. L. Johnson, D. Klabjan, G. L. Nemhauser, and P. H. Vance. Airline crew scheduling. In R. W. Hall, editor, *Handbook of Transportation Science*, number 56 in International Series in Operations Research & Management Science, pages 517–560. Springer US, Jan 2003.

O. Battaïa and A. Dolgui. A taxonomy of line balancing problems and their solution approaches. *International Journal of Production Economics*, 142(2):259–277, Apr 2013.

I. Baybars. A survey of exact algorithms for the simple assembly line balancing problem. *Management science*, 32(8):909–932, 1986.

E. M. L. Beale and J. J. H. Forrest. Global optimization using special ordered sets. *Mathematical Programming*, 10(1):52–69, 1976.

E. M. L. Beale and J. A. Tomlin. Special facilities in a general mathematical programming system for non-convex problems using ordered sets of variables. In *Fifth annual international conference on operational research*, volume 69, pages 447–454. Tavistock Publications, 1970.

J. E. Beasley. Or-library, Jun 2012. URL `http://people.brunel.ac.uk/~mastjjb/jeb/info.html`. Last accessed March 2014.

M. Behle. *Binary Decision Diagrams and Integer Programming*. PhD thesis, Universistät des Saarlands, 2007.

M. Behle. On threshold BDDs and the optimal variable ordering problem. *Journal of Combinatorial Optimization*, 16(2):107–118, Aug 2008.

M. Behle and F. Eisenbrand. 0/1 vertex and facet enumeration with BDDs. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007.

J. Beliën and E. Demeulemeester. A branch-and-price approach for integrating nurse and surgery scheduling. *European journal of operational research*, 189(3):652–668, 2008.

R. Bellman. The theory of dynamic programming. *Bulletin of the American Mathematical Society*, 60(6): 503–515, 1954.

J. F. Benders. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, 4(1):238–252, Dec 1962.

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

D. Bergman, A. A. Cire, W. J. V. Hoeve, and J. N. Hooker. Variable ordering for the application of BDDs to the maximum independent set problem. In N. Beldiceanu, N. Jussien, and É. Pinson, editors, *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems*, number 7298 in Lecture Notes in Computer Science, pages 34–49. Springer Berlin Heidelberg, Jan 2012.

D. Bergman, A. A. Cire, W. J. V. Hoeve, and J. N. Hooker. Optimization bounds from binary decision diagrams. *INFORMS Journal on Computing*, Articles in Advance, Aug 2013.

D. Bertsimas and J. N. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, Feb 1997.

R. E. Bixby, M. Fenelon, Z. Gu, E. Rothberg, and R. Wunderling. MIP: theory and practice— closing the gap. In M. J. D. Powell and S. Scholtes, editors, *System Modelling and Optimization*, number 46 in The International Federation for Information Processing, pages 19–49. Springer US, Jan 2000.

B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002, Sep 1996.

C. Bron and J. Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, Sep 1973.

R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug 1986.

R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)*, 24(3):293–318, 1992.

Y. Caseau and F. Laburthe. Improving branch and bound for jobshop scheduling with constraint propagation. In M. Deza, R. Euler, and I. Manoussakis, editors, *Combinatorics and Computer Science*, number 1120 in Lecture Notes in Computer Science, pages 129–149. Springer Berlin Heidelberg, Jan 1996.

S. Choi, M. Shin, and J. Cha. Loss reduction in distribution networks using cyclic best first search. In M. L. Gavrilova, O. Gervasi, V. Kumar, C. J. K. Tan, D. Taniar, A. Laganá, Y. Mun, and H. Choo, editors, *Computational Science and Its Applications*, number 3984 in Lecture Notes in Computer Science, pages 312–321. Springer Berlin Heidelberg, Jan 2006.

A. Cire, D. Bergman, J. N. Hooker, and W. J. V. Hoeve. A branch and bound algorithm based on approximate binary decision diagrams. In *INFORMS Annual Conference*, Oct 2012.

J. Clausen. Branch and bound algorithms—principles and examples. *Department of Computer Science, University of Copenhagen*, pages 1–30, 1999.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, Jul 2009.

G. Cornuéjols. Valid inequalities for mixed integer linear programs. *Mathematical Programming*, 112(1): 3–44, Mar 2008.

C. D'Ambrosio and A. Lodi. Mixed integer nonlinear programming tools: a practical overview. *4OR*, 9(4): 329–349, Dec 2011.

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

M. P. de Aragão and E. Uchoa. Integer program reformulation for robust branch-and-cut-and-price algorithms. In *Mathematical Program in Rio: a Conference in Honour of Nelson Maculan*, pages 56–61, 2003.

I. de Farias, Jr., E. Johnson, and G. Nemhauser. A generalized assignment problem with special ordered sets: a polyhedral approach. *Mathematical Programming*, 89(1):187–203, 2000.

R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of A*. *Journal of the ACM*, 32(3):505–536, Jul 1985.

E. Demeulemeester, B. D. Reyck, and W. Herroelen. The discrete time/resource trade-off problem in project networks: a branch-and-bound approach. *IIE Transactions*, 32(11):1059–1069, Nov 2000.

J. Desrosiers, R. Jans, and Y. Adulyasak. Improved column generation algorithms for the job grouping problem. Technical Report G–2013–26, Les Cahiers du GERAD, 2013.

M. Dür and V. Stix. Probabilistic subproblem selection in branch-and-bound algorithms. *Journal of Computational and Applied Mathematics*, 182(1):67–80, Oct 2005.

K. Easton, G. Nemhauser, and M. Trick. Solving the travelling tournament problem: A combined integer programming and constraint programming approach. In E. Burke and P. D. Causmaecker, editors, *Practice and Theory of Automated Timetabling IV*, number 2740 in Lecture Notes in Computer Science, pages 100–109. Springer Berlin Heidelberg, Jan 2003.

S. Elhedhli, L. Li, M. Gzara, and J. Naoum-Sawaya. A branch-and-price algorithm for the bin packing problem with conflicts. *INFORMS Journal on Computing*, 23(3):404–415, Jun 2011.

D. Eppstein and D. Strash. Listing all maximal cliques in large sparse real-world graphs. In P. M. Pardalos and S. Rebennack, editors, *Experimental Algorithms*, number 6630 in Lecture Notes in Computer Science, pages 364–375. Springer Berlin Heidelberg, Jan 2011.

T. Fahle. Simple and fast: Improving a branch-and-bound algorithm for maximum clique. In R. Möhring and R. Raman, editors, *Algorithms ESA 2002*, number 2461 in Lecture Notes in Computer Science, pages 485–498. Springer Berlin Heidelberg, Jan 2002.

Fair Isaac Corporation (FICO). *Xpress Optimization Suite*, 2014.

A. A. Farley. A note on bounding a class of linear programming problems, including cutting stock problems. *Operations Research*, 38(5):922–923, Oct 1990.

M. Fischetti and A. Lodi. Local branching. *Mathematical Programming*, 98(1-3):23–47, Sep 2003.

M. Fischetti and M. Monaci. Backdoor branching. In O. Günlük and G. J. Woeginger, editors, *Integer Programming and Combinatoral Optimization*, number 6655 in Lecture Notes in Computer Science, pages 183–191. Springer Berlin Heidelberg, Jan 2011.

M. Fischetti and D. Salvagnin. Pruning moves. *INFORMS Journal on Computing*, 22(1):108–119, 2010.

R. Fukasawa, H. Longo, J. Lysgaard, M. P. D. Aragão, M. Reis, E. Uchoa, and R. F. Werneck. Robust branch-and-cut-and-price for the capacitated vehicle routing problem. *Mathematical Programming*, 106 (3):491–511, May 2006.

K. Fukunaga and P. M. Narendra. A branch and bound algorithm for computing k-nearest neighbors. *IEEE Transactions on Computers*, C-24(7):750–753, Jul 1975.

P. Galinier and A. Hertz. A survey of local search methods for graph coloring. *Computers & Operations Research*, 33(9):2547–2562, Sep 2006.

M. Gendreau, J. Ferland, B. Gendron, N. Hail, B. Jaumard, S. Lapierre, G. Pesant, and P. Soriano. Physician scheduling in emergency rooms. In *Practice and Theory of Automated Timetabling VI*, pages 53–66. Springer, 2007.

B. Gendron, P. Khuong, and F. Semet. A Lagrangian-based branch-and-bound algorithm for the two-level uncapacitated facility location problem with single-assignment constraints. Technical Report CIRRELT-2013-21, CIRRELT, Apr 2013.

A. M. Geoffrion. An improved implicit enumeration approach for integer programming. *Operations Research*, 17(3):437–454, May 1969.

A. M. Geoffrion. Generalized Benders decomposition. *Journal of Optimization Theory and Applications*, 10 (4):237–260, Oct 1972.

A. Gilpin and T. Sandholm. Information-theoretic approaches to branching in search. *Discrete Optimization*, 8(2):147–159, May 2011.

F. Glover. Tabu search: A tutorial. *Interfaces*, 20(4):74–94, Jul 1990.

S. W. Golomb and L. D. Baumert. Backtrack programming. *Journal of the ACM*, 12(4):516–524, Oct 1965.

R. E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, 64(5):275–278, 1958.

S. Görtz and A. Klose. A simple but usually fast branch-and-bound algorithm for the capacitated facility location problem. *INFORMS Journal on Computing*, 24(4):597–610, Sep 2012.

A. Grosso, M. Locatelli, and W. Pullan. Simple ingredients leading to very efficient heuristics for the maximum clique problem. *Journal of Heuristics*, 14(6):587–612, Dec 2008.

S. Gualandi and F. Malucelli. Exact solution of graph coloring problems via constraint programming and column generation. *INFORMS Journal on Computing*, 24(1):81–100, Feb 2012.

Gurobi Optimization, Inc. *Gurobi Optimizer 5.6*, 2014.

M. Guzelsoy, G. Nemhauser, and M. Savelsbergh. Restrict-and-relax search for 0-1 mixed-integer programs. *EURO Journal on Computational Optimization*, 1(1-2):201–218, May 2013.

T. Hadžić and J. Hooker. Postoptimality analysis for integer programming using binary decision diagrams. Technical Report 167, Tepper School of Business, Apr 2008.

S. Held, W. Cook, and E. C. Sewell. Maximum-weight stable sets and safe lower bounds for graph coloring. *Mathematical Programming Computation*, 4(4):363–381, Dec 2012.

H. Hernández-Pérez and J. Salazar-González. A branch-and-cut algorithm for a traveling salesman problem with pickup and delivery. *Discrete Applied Mathematics*, 145(1):126–139, Dec 2004.

T. Ibaraki. Theoretical comparisons of search strategies in branch-and-bound algorithms. *International Journal of Computer & Information Sciences*, 5(4):315–344, Dec 1976.

T. Ibaraki. The power of dominance relations in branch-and-bound algorithms. *Journal of the ACM*, 24(2): 264–279, Apr 1977.

IBM Corp. *IBM ILOG CPLEX Optimization Studio V12.5*, 2014.

D. S. Johnson and M. A. Trick. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, October 11-13, 1993*. American Mathematical Society, Jan 1996.

M. Jünger, G. Reinelt, and S. Thienel. Practical problem solving with cutting plane algorithms in combinatorial optimization. In W. Cook, L. Lovász, and P. Seymour, editors, *Combinatorial Optimization*, volume 20 of *DIMACS series in Discrete Mathematics and Theoretical Computer Science*, pages 111–152. American Mathematical Society, 1995.

G. K. Kao, E. C. Sewell, and S. H. Jacobson. A branch, bound, and remember algorithm for the $1|r_i|\sum t_i$ scheduling problem. *Journal of Scheduling*, 12(2):163–175, 2009.

F. K. Karzan, G. L. Nemhauser, and M. W. P. Savelsbergh. Information-based branching schemes for binary linear mixed integer problems. *Mathematical Programming Computation*, 1(4):249–293, Dec 2009.

H. Kim and J. N. Hooker. Solving fixed-charge network flow problems with a hybrid optimization and constraint programming approach. *Annals of Operations Research*, 115(1-4):95–124, Sep 2002.

D. Knuth. Fun with zero-suppressed binary decision diagrams, Dec 2008. URL `http://myvideos.stanford.edu/player/slplayer.aspx?coll=ea60314a-53b3-4be2-8552-dcf190ca0c0b&co=af52aca1-9c60-4a7b-a10b-0e543f4f3451&o=true`. Last accessed March 2014.

T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R. E. Bixby, E. Danna, G. Gamrath, A. M. Gleixner, S. Heinz, A. Lodi, H. Mittelmann, T. Ralphs, D. Salvagnin, D. E. Steffy, and K. Wolter. MIPLIB 2010. *Mathematical Programming Computation*, 3(2):103–163, 2011.

W. H. Kohler and K. Steiglitz. Characterization and theoretical comparison of branch-and-bound algorithms for permutation problems. *Journal of the ACM*, 21(1):140–156, Jan 1974.

P. J. Kolesar. A branch and bound algorithm for the knapsack problem. *Management Science*, 13(9): 723–735, May 1967.

Konrad-Zuse-Zentrum für Informationstechnik Berlin. *SCIP Optimization Suite 3.0.1*, 2014. URL `http://scip.zib.de/scip.shtml`. Last accessed March 2014.

R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1): 97–109, 1985.

V. Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, 13(1):32, Mar 1992.

L. Ladanyi, T. Ralphs, M. Guzelsoy, and A. Mahajan. *SYMPHONY 5.5.0*, 2014. URL `https://projects.coin-or.org/SYMPHONY`. Last accessed March 2014.

A. H. Land and A. G. Doig. An automatic method for solving discrete programming problems. *Econometrics*, pages 497–520, 1960.

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

C. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38(4):985–999, 1959.

A. N. Letchford and A. Lodi. Strengthening Chvátal–Gomory cuts and gomory fractional cuts. *Operations Research Letters*, 30(2):74–82, Apr 2002.

C. M. Li, F. Manyà, and J. Planes. Exploiting unit propagation to compute lower bounds in branch and bound max-SAT solvers. In P. V. Beek, editor, *Principles and Practice of Constraint Programming*, number 3709 in Lecture Notes in Computer Science, pages 403–414. Springer Berlin Heidelberg, Jan 2005.

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

LINDO Systems, Inc. *LINDO API 8.0*, 2014.

A. Lodi, T. K. Ralphs, F. Rossi, and S. Smriglio. Interdiction branching. Technical report, COR@L Laboratory, Lehigh University, 2011.

L. Lovász and A. Schrijver. Cones of matrices and set-functions and 0–1 optimization. *SIAM Journal on Optimization*, 1(2):166–190, May 1991.

M. E. Lübbecke and J. Desrosiers. Selected topics in column generation. *Operations Research*, 53(6):1007–1023, Nov 2005.

E. Malaguti. The vertex coloring problem and its generalizations. *4OR*, 7(1):101–104, Mar 2008.

E. Malaguti and P. Toth. A survey on vertex coloring problems. *International Transactions in Operational Research*, 17(1):1–34, Jan 2010.

E. Malaguti, M. Monaci, and P. Toth. An exact approach for the vertex coloring problem. *Discrete Optimization*, 8(2):174–190, May 2011.

S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley and Sons, 1990.

A. Mehrotra and M. A. Trick. A column generation approach for graph coloring. *INFORMS Journal on Computing*, 8(4):344–354, 1996.

I. Méndez-Díaz and P. Zabala. A branch-and-cut algorithm for graph coloring. *Discrete Applied Mathematics*, 154(5):826–847, Apr 2006.

P. Meseguer. Interleaved depth-first search. In *IJCAI*, volume 97, pages 1382–1387, 1997.

S. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *30th Conference on Design Automation*, pages 272–277, 1993.

J. E. Mitchell. Branch-and-cut algorithms for combinatorial optimization problems. *Handbook of Applied Optimization*, pages 65–77, 2002.

G. Mitra. Investigation of some branch and bound strategies for the solution of mixed integer linear programs. *Mathematical Programming*, 4(1):155–170, Dec 1973.

T. Nazareth, S. Verma, S. Bhattacharya, and A. Bagchi. The multiple resource constrained project scheduling problem: A breadth-first approach. *European Journal of Operational Research*, 112(2):347–366, Jan 1999.

G. Nemhauser and L. Wolsey. *Integer and combinatorial optimization*, volume 18. Wiley New York, 1988.

F. Ortega and L. A. Wolsey. A branch-and-cut algorithm for the single-commodity, uncapacitated, fixed-charge network flow problem. *Networks*, 41(3):143–158, 2003.

J. Ostrowski, J. Linderoth, F. Rossi, and S. Smriglio. Orbital branching. *Mathematical Programming*, 126 (1):147–178, Jan 2011.

A. Otto, C. Otto, and A. Scholl. Systematic data generation and test design for solution algorithms on the example of SALBPGen for assembly line balancing. *European Journal of Operational Research*, 228(1): 33–45, Jul 2013.

M. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33(1):60–100, Mar 1991.

C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Courier Dover Publications, Jan 1998.

P. Pardalos, T. Mavridou, and J. Xue. The graph coloring problem: A bibliographic survey. In D. Du and P. M. Pardalos, editors, *Handbook of combinatorial optimization*, volume 2, pages 331–395. Kluwer Academic Publishers, 1998.

D. T. Phan. Lagrangian duality and branch-and-bound algorithms for optimal power flow. *Operations Research*, 60(2):275–285, 2012.

A. Pigatti, M. P. De Aragão, and E. Uchoa. Stabilized branch-and-cut-and-price for the generalized assignment problem. *Annals of GRACO*, 5:389–395, 2005.

J. Pryor and J. W. Chinneck. Faster integer-feasibility in mixed-integer linear programs by branching to force change. *Computers & Operations Research*, 38(8):1143–1152, Aug 2011.

D. P. Ranjan, D. Tang, and S. Malik. A comparative study of 2QBF algorithms. In *SAT*, pages 292–305, 2004.

F. Rossi, P. V. Beek, and T. Walsh. *Handbook of Constraint Programming.* Elsevier Science, first edition, Oct 2006.

T. Sandholm and R. Shields. Nogood learning for mixed integer programming. In *Workshop on Hybrid Methods and Branching Rules in Combinatorial Optimization, Montral*, 2006.

B. R. Sarker and H. Pan. Designing a mixed-model, open-station assembly line using mixed-integer programming. *The Journal of the Operational Research Society*, 52(5):545–558, May 2001.

M. Savelsbergh. A branch-and-price algorithm for the generalized assignment problem. *Operations Research*, 45(6):831–841, 1997.

A. Scholl and C. Becker. State-of-the-art exact and heuristic solution procedures for simple assembly line balancing. *European Journal of Operational Research*, 168(3):666–693, 2006.

A. Scholl and R. Klein. SALOME: a bidirectional branch-and-bound procedure for assembly line balancing. *INFORMS Journal on Computing*, 9(4):319–334, 1997.

A. Scholl and R. Klein. Balancing assembly lines effectively—a computational comparison. *European Journal of Operational Research*, 114(1):50–58, Apr 1999.

R. Sedgewick and K. Wayne. *Algorithms*. Addison-Wesley Professional, Feb 2011.

E. C. Sewell and S. H. Jacobson. A branch, bound, and remember algorithm for the simple assembly line balancing problem. *INFORMS Journal on Computing*, 24(3):433–442, 2012.

E. C. Sewell, J. J. Sauppe, D. R. Morrison, S. H. Jacobson, and G. Kao. A BB&R algorithm for minimizing total tardiness on a single machine with sequence dependent setup times. *Journal of Global Optimization*, 54(4):791–812, Dec 2012.

H. D. Sherali and C. H. Tuncbilek. A global optimization algorithm for polynomial programming problems using a reformulation-linearization technique. *Journal of Global Optimization*, 2(1):101–112, Mar 1992.

L. Shi and S. Ólafsson. Nested partitions method for global optimization. *Operations Research*, 48(3): 390–407, May 2000.

D. J. Slate and L. R. Atkin. CHESS 4.5—The northwestern university chess program. In P. W. Frey, editor, *Chess Skill in Man and Machine*, pages 82–118. Springer New York, Jan 1983.

R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, Jun 1972.

M. Tawarmalani and N. V. Sahinidis. Global optimization of mixed-integer nonlinear programs: A theoretical and computational study. *Mathematical Programming*, 99(3):563–591, Apr 2004.

E. Tomita, A. Tanaka, and H. Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theoretical Computer Science*, 363(1):28–42, Oct 2006.

M. A. Trick. Computational series: Graph coloring and its generalizations, Mar 2005. URL `http://mat.gsia.cmu.edu/COLOR02/`. Last accessed March 2014.

E. Uchoa, R. Fukasawa, J. Lysgaard, A. Pessoa, M. P. de Aragão, and D. Andrade. Robust branch-cut-and-price for the capacitated minimum spanning tree problem over a large extended formulation. *Mathematical Programming*, 112(2):443–472, 2008.

P. van Beek. Backtracking search algorithms. In F. Rossi, P. van Beek, and T. Walsh, editors, *Foundations of Artificial Intelligence*, volume 2 of *Handbook of Constraint Programming*, chapter 4, pages 85–134. Elsevier, 2006.

F. Vanderbeck. Branching in branch-and-price: a generic scheme. *Mathematical Programming*, 130(2): 249–294, Dec 2011.

M. Vilà and J. Pereira. A branch-and-bound algorithm for assembly line worker assignment and balancing problems. *Computers & Operations Research*, 44:105–114, Apr 2014.

E. W. Weisstein. Binet's fibonacci number formula—from Wolfram MathWorld, 2013. URL `http://mathworld.wolfram.com/BinetsFibonacciNumberFormula.html`. Last accessed March 2014.

D. Zuckerman. Linear degree extractors and the inapproximability of max clique and chromatic number. In *Proceedings of the Thirty-eighth Annual ACM Symposium on Theory of Computing*, STOC '06, pages 681–690, New York, NY, USA, 2006. ACM.