Solving the Pricing Problem in a Branch-and-Price Algorithm for Graph Coloring using Zero-Suppressed Binary Decision Diagrams

David R. Morrison, Edward C. Sewell, Sheldon H. Jacobson

Abstract

Branch-and-price algorithms combine a branch-and-bound search with an exponentially-sized LP formulation that must be solved via column generation. Unfortunately, the standard branching rules used in branch-and-bound for integer programming interfere with the structure of the column generation routine; therefore, most such algorithms employ alternate branching rules to circumvent this difficulty. This paper shows how a *zero-suppressed binary decision diagram* (ZDD) can be used to solve the pricing problem in a branch-and-price algorithm for the graph coloring problem, even in the presence of constraints imposed by branching decisions. This approach facilitates a much more direct solution method, and can improve convergence of the column generation subroutine.

1 Introduction

Branch-and-price algorithms are of increasing interest in many areas of operations research, including assignment and scheduling problems (Savelsbergh, 1997; Maenhout and Vanhoucke, 2010), vehicle routing problems (Fukasawa et al., 2006), graph coloring (Mehrotra and Trick, 1996; Malaguti et al., 2011), multicommodity flow problems (Barnhart et al., 2000), and cutting stock problems (Vance, 1998; Pisinger and Sigurd, 2007), among others. These algorithms combine a branch-and-bound search together with a tight linear programming relaxation having an exponential number of variables (such a formulation can be derived, for example, by the Dantzig-Wolfe decomposition method described in Dantzig and Wolfe, 1960). This LP relaxation, called the *master problem*, is used to produce good bounds that are used to prune suboptimal regions of the search space. Because the LP relaxation is too large to be stored in memory, it must be solved via *column generation*.

Let $\mathcal{S}$ be the set of variables for the LP relaxation; each of these variables is associated with a column of the master problem's constraint matrix (thus, the constraint matrix has an exponential number of columns). In the column generation method, a related LP called the *restricted master problem* (RMP) is built using a (small) subset $\mathcal{S}' \subseteq \mathcal{S}$ of variables. The RMP can be solved

efficiently by standard linear programming techniques; however, the solution to the RMP is not necessarily optimal for the master problem. Therefore, a subroutine called the *pricing problem* must be called to either produce a variable in $\mathcal{S} \setminus \mathcal{S}'$ that may improve the objective value of the RMP, or provide a guarantee that no such variable exists. If an improving variable is found, it is added to $\mathcal{S}'$, and the RMP is re-optimized. New columns are iteratively added to $\mathcal{S}'$ until the pricing problem reports that no (potentially) improving variables exist, at which point column generation is terminated and the solution to the RMP is provably optimal for the master problem.

In this paper, the pricing problem is assumed to be a weighted binary combinatorial optimization problem which is characterized by a family of "valid" subsets of some universe; in a slight abuse of notation, solutions to the pricing problem are interchangeably referred to as "variables", "columns", or "subsets", where the meaning is clear from context. The weights associated with the pricing problem are usually the optimal dual prices for the current solution to the RMP. Thus, the pricing problem returns a new variable with negative reduced cost if one exists; if such a variable exists, it may improve the value of the RMP (Bertsimas and Tsitsiklis, 1997). From this perspective, the pricing problem is a separation oracle for the dual of the RMP, since new variables for the RMP correspond to additional constraints in the dual.

Since the pricing problem is itself often NP-hard, and must be solved exactly, solving it is typically the most computationally-intensive part of a branch-and-price procedure. Moreover, when combined with the standard integer branching scheme used in most branch-and-bound algorithms, the structure of the pricing problem is destroyed (Barnhart et al., 1998). In such a branching scheme, a variable x_i with fractional value α is selected at a subproblem in the search tree, and two children are created with additional bounding constraints $x_i \leq \lfloor \alpha \rfloor$ and $x_i \geq \lceil \alpha \rceil$ (when all variables are binary, this is called $0-1$ *branching*). However, imposing these constraints changes the structure of the dual problem, which in turn means that a different separation oracle must be queried at each subproblem in order to generate new columns.

In effect, the pricing problem at these subproblems no longer seeks a variable of minimum reduced cost; it now must produce a variable with minimum reduced cost that respects the current branching decisions. This problem, called the *constrained pricing problem*, is much harder than the regular (or *unconstrained*) pricing problem, and is often related to the kth-shortest-path problem, which is well-known to be a challenging NP-hard problem (Garey and Johnson, 1979).

Moreover, many branch-and-price formulations have an inherent asymmetry due to the large number of variables in the formulation. This asymmetry can lead to extremely lopsided search trees if standard integer branching techniques are used. For example, in a problem with many covering constraints (of the form $\sum_i x_i \geq b$), fixing a variable to zero may not induce much change in the LP relaxation, but fixing a variable to 1 may immediately satisfy many constraints. Thus long paths in the search tree can exist where many variables are fixed to 0 but no progress towards a solution is made.

Therefore, most branch-and-price algorithms employ specialized branching rules or other techniques to avoid eliminating the pricing problem structure, as well as to maintain a more balanced search tree. For example, some branching rules modify the problem structure at each subproblem in the search tree (e.g., the graph coloring rule of Mehrotra and Trick, 1996); others branch on original (non-reformulated) problem variables, or problem constraints (Vanderbeck, 2011). A related scheme by Morrison et al. (2014a) uses a modified branching scheme called *wide branching*, which does not wholly eliminate calls to the constrained pricing problem, but restructures the search tree in an attempt to reduce the number of such calls.

An alternate approach, called *robust branch-and-cut-and-price* (BCP), eliminates calls to the constrained pricing problem by further modifying the RMP so that branching restrictions can be added without interfering with the pricing problem structure (de Aragão and Uchoa, 2003). This approach introduces additional variables and constraints into the RMP to form a linear program called the *explicit master*, which has the same objective value as the RMP. Furthermore, branching decisions made by the algorithm can be communicated to the pricing problem by imposing constraints on the reduced cost values in the dual of the explicit master. This approach has been used successfully in many variants of the capacitated vehicle routing problem (Pessoa et al., 2008; Fukasawa et al., 2006), as well as related problems such as the capacitated minimum spanning tree problem (Uchoa et al., 2008).

However, no algorithm in the literature has described a way to perform branch-and-price without using techniques like robust BCP or alternative branching rules, which often 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. The wide branching approach allows variables to be fixed explicitly, but to obtain good performance, it requires the derivation of a problem-specific branching rule.

Therefore, the primary contribution of this research is to establish an efficient method for solving the pricing problem in a branch-and-price algorithm for graph coloring that is directly compatible with the standard integer branching scheme for graph coloring. Two algorithmic ideas enable this result: the first 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, which allows the constrained pricing problem to be solved at every iteration of column generation. The second idea combines the above solution procedure with the cyclic best-first search (CBFS) strategy to overcome the lopsided search trees that can result when using standard integer branching with exponentially-sized problems. Computational results are presented 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 paper is organized as follows: Section 2 defines the ZDD data structure and shows how it can be used to solve the pricing problem for the graph coloring problem. This is done in three parts: first, Section 2.1 shows how ZDDs can be used to solve an arbitrary unconstrained pricing problem; secondly, Section 2.2 shows how to modify this ZDD to solve the constrained pricing problem; finally, Section 2.3 shows how to build a ZDD for the pricing problem in the graph coloring problem, namely, the maximal independent set problem. Next, Section 3 describes the cyclic best-first search strategy and how it is used to mitigate the effects of lopsided search trees. In Section 4, the computational results are given showing the effectiveness of the developed algorithm. Finally, Section 5 outlines several future research directions for this technique.

2 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 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. (2012a) 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. (2012b) describe how to use BDDs to compute upper and lower bounds to prune subproblems in a branch-and-bound algorithm.

Despite their success in these related areas, BDDs and ZDDs have not appeared in conjunction with branch-and-price 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 branch-and-price setting. Additionally, Behle (2007) uses BDDs to generate valid inequalities in a branch-and-cut algorithm to perform row generation instead of column generation.

The use of decision diagrams together with branch-and-price algorithms can provide substantial benefits to algorithm performance. This is because decision diagrams often yield a way to compactly (in practice) 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 branch-and-price 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 2.2, the set of valid pricing problem solutions can be modified in place, allowing branch-and-price algorithms using ZDDs to employ standard integer branching methods.

2.1 Definitions and Notation

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 branch-and-price setting than ordinary BDDs.

Formally, a ZDD Z is defined as follows. Let $\mathcal{E}$ be an ordered set of n elements $(e_1, e_2, \ldots, e_n)$; then Z is a directed acyclic graph 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 r.

2. Every node $a \in Z - \{\mathbf{1}, \mathbf{0}\}$ 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 a is denoted $\text{hi}(a)$ ($\text{lo}(a)$). The true and false nodes have no outgoing edges. The *indegree* of a, denoted $\delta^-(a)$, is the number of incoming edges to a; thus, $\delta^-(r) = 0$.

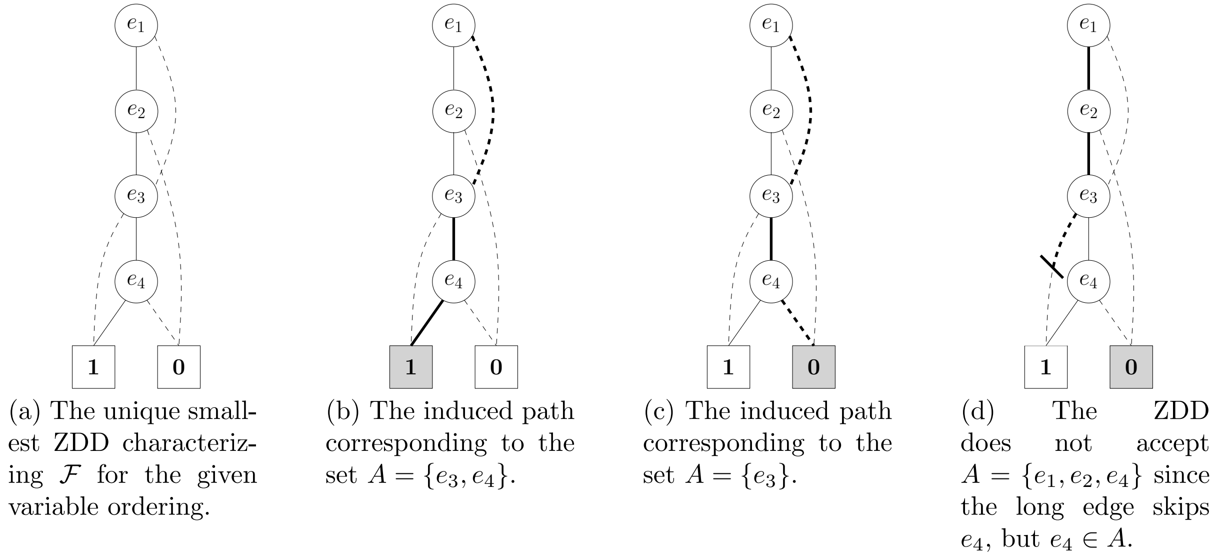
3. Every node $a \in Z - \{\mathbf{1}, \mathbf{0}\}$ is associated with some element $e_i \in \mathcal{E}$; the index of the associated element for a is given by $\text{var}(a)$, that is, $\text{var}(a) = i$. By convention, $\text{var}(\mathbf{1}) = \text{var}(\mathbf{0}) = n + 1$. Finally, if $\text{var}(a)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

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

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

For an arbitrary family $\mathcal{F}$ and an arbitrary vertex ordering, the size of $Z_{\mathcal{F}}$ (that is, the number of nodes and edges in the graph, denoted $|Z_{\mathcal{F}}|$) may be exponential in n. However, Bryant (1986)

Figure 1: Let $\mathcal{E} = (e_1, e_2, e_3, e_4)$, and $\mathcal{F} = \{\varnothing, \{e_1, e_2\}, \{e_3, e_4\}, \{e_1, e_2, e_3, e_4\}\}$ (Andersen, 1997). Solid lines represent high edges, and dashed lines represent low edges; all edges are directed downwards. Grey nodes indicate whether $Z_\mathcal{F}$ accepts A.



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

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

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

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

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 $\mathcal{F}$ can be defined as a boolean function. One way to construct the unique smallest ZDD characterizing $\mathcal{F}$ is to first construct the BDD for $\mathcal{F}$'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 $Z_\mathcal{F}$ directly (Knuth, 2008).

Note that the choice of ordering on the elements of $\mathcal{E}$ 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 branch-and-price algorithm, let $\mathcal{F}$ be the set of all valid solutions to the pricing problem. Then, using the technique of Hadžić and Hooker (2008), assign weights to the edges of $Z_\mathcal{F}$ and compute the longest path or shortest path in $Z_\mathcal{F}$ from the root to **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 $\mathcal{E}$; set the weight of edge $(a, b) \in Z_{\mathcal{F}}$ to $\pi_{\text{var}(a)}$ if $b = \text{hi}(a)$, and 0 otherwise. Then, finding the longest or shortest path with respect to $\{\pi\}$ from the root of $Z_{\mathcal{F}}$ to **1** can be found in $O(|Z_{\mathcal{F}}|)$ time using dynamic programming (Sedgewick and Wayne, 2011). The resulting path corresponds to the optimal solution to the pricing problem (see Figure 2).

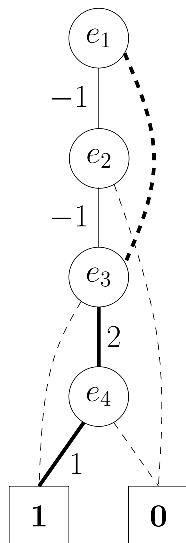


Figure 2: The ZDD from Figure 1a with weights given by the objective function $\max\,[-e_1 - e_2 + 2e_3 + e_4]$; the bold path corresponds to the maximum-weight valid set, that is $\{e_3, e_4\}$. Weights not shown are 0.

2.2 The ZDD Restriction Algorithm

In order to use standard integer branching methods in a branch-and-price 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 $\mathcal{S}'$ for the RMP; to see this, observe that if any variable in $\mathcal{S}'$ 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 $\mathcal{S}'$, 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 $\mathcal{F}$ be the family of valid solutions to the pricing problem, where each $A \in \mathcal{F}$ is a subset of $\mathcal{E} = (e_1, e_2, \ldots, e_n)$, and let $Z_\mathcal{F}$ be the ZDD characterizing $\mathcal{F}$. The restriction algorithm for ZDDs, called `RestrictSet`, takes as input a set $A \in \mathcal{F}$, and builds a new ZDD $Z_{\mathcal{F}'}$ that accepts $\mathcal{F}' = \mathcal{F} - A$. 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 $Z_\mathcal{F}$ by at most n nodes and $2n$ edges (and often by much less).

Intuitively, the `RestrictSet` algorithm identifies the path P_A in $Z_\mathcal{F}$ corresponding to the set A, and updates this path so that it ends at the false node instead of the true node. However, if there exists $A' \neq A$ such that P_A and $P_{A'}$ overlap, this update could also restrict A'. Therefore, `RestrictSet` duplicates the portion of P_A 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_A with indegree greater than one, referred to as the *split node*, 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 1; this algorithm makes use of a function called $Z_\mathcal{F}$.`insert`(i, a_1, a_2), which takes as input an index $i \in \{1, 2, \ldots, n\}$ and pointers to two pre-existing nodes $a_1, a_2 \in Z_\mathcal{F}$. The function inserts a new node into $Z_\mathcal{F}$ associated with element e_i, with low edge pointing to a_1 and high edge pointing to a_2, and returns a pointer to the newly-inserted node. It also updates the indegrees of the high and low children. $Z_\mathcal{F}$.`insert` can be implemented in (average) constant time (see Andersen, 1997 for details).

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 1a is given in Figure 3.

Theorem 1. *Given a ZDD $Z_\mathcal{F}$ describing a family of subsets $\mathcal{F}$ of an ordered set $\mathcal{E}$ with n elements, together with a set $A \in \mathcal{F}$, the* `RestrictSet` *algorithm modifies $Z_\mathcal{F}$ in $O(n)$ time to produce a new ZDD called $Z_{\mathcal{F}'}$ such that $\mathcal{F}' = \mathcal{F} - A$. Furthermore, $|Z_{\mathcal{F}'}| \leq |Z_\mathcal{F}| + 3n$.*

Proof. First, note that `RestrictSet` visits each node along P_A at most twice, and P_A 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 c is at most the root of $Z_\mathcal{F}$, at most $|\mathcal{E}|$ nodes are added to $Z_\mathcal{F}$ to form $Z_{\mathcal{F}'}$, and each new node has two outgoing edges.

Algorithm 1: RestrictSet($Z_\mathcal{F}, A$)

input: A ZDD $Z_\mathcal{F}$ and a characteristic vector $(\alpha_1, \alpha_2, \ldots, \alpha_n)$ for a set $A \in \mathcal{F}$
output: A modified ZDD $Z_{\mathcal{F}'}$ such that $\mathcal{F}' = \mathcal{F} - A$

1 ⟪ *Find the first node on P_A with indegree higher than 1* ⟫
2 $a = \text{root}(Z_\mathcal{F})$; $b = -1$
3 **while** $\delta^-(a) < 2$ and $a \notin \{\mathbf{1}, \mathbf{0}\}$ **:**
4     $i = \text{var}(a)$; $b = a$
5   **if** α_i **:** $a = \text{hi}(a)$
6   **else:** $a = \text{lo}(a)$
7 ⟪ *The node c is the "split" node, and b is its parent* ⟫
8 $c = a$
9 ⟪ *Make copies of all remaining nodes on P_A and point to* $\mathbf{0}$ ⟫
10 list = ()
11 **while** $a \notin \{\mathbf{1}, \mathbf{0}\}$ **:**
12   list.`append`(a)
13   **if** $\alpha_{\text{var}(a)}$ **:** $a = \text{hi}(a)$
14   **else:** $a = \text{lo}(a)$
15 $a' = \mathbf{0}$
16 ⟪ *Insert the duplicated nodes into $Z_\mathcal{F}$* ⟫
17 **for each** $a \in$ list (in reverse order) **:**
18   **if** $\alpha_{\text{var}(a)}$ **:** $a' = Z_\mathcal{F}$.`insert`$(\text{var}(a), \text{lo}(a), a')$
19   **else:** $a' = Z_\mathcal{F}$.`insert`$(\text{var}(a), a', \text{hi}(a))$
20 ⟪ *Point the correct edge of the parent node b to the root of the duplicated path* ⟫
21 **if** $\alpha_{\text{var}(b)}$ **:** $\text{hi}(b) = a'$
22 **else:** $\text{lo}(b) = a'$
23 return $Z_\mathcal{F}$

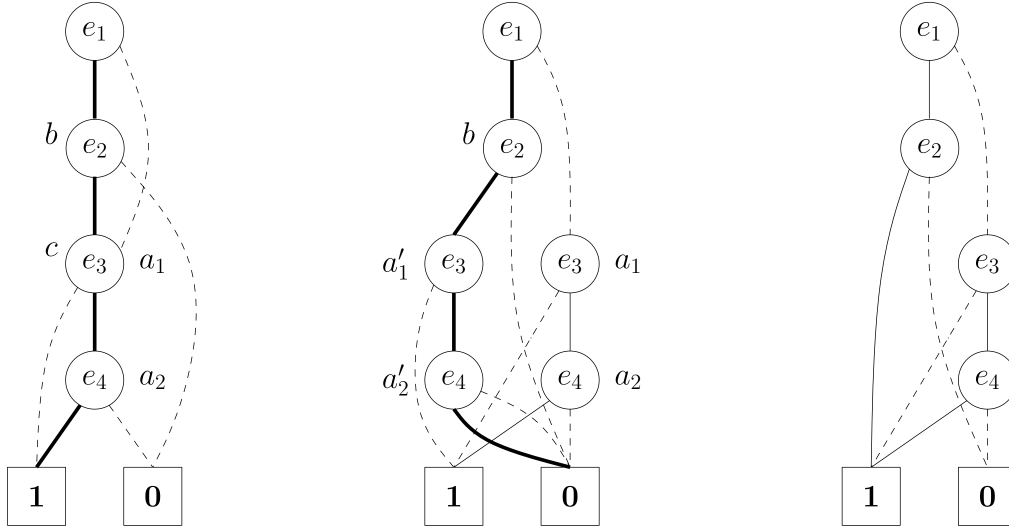
To prove that $Z_{\mathcal{F}'}$ has the desired properties, let $a'_1, a'_2, \ldots, a'_l$ be the nodes added to $Z_\mathcal{F}$ in lines 18-19 (Algorithm 1), in increasing order of depth. Consider some set $A' \subseteq \mathcal{E}$; if $A' = A$, the path from the root of $Z_{\mathcal{F}'}$ to the bottom of the ZDD is the same as the path from the root of $Z_\mathcal{F}$ up to the parent b of the split node c. By construction, the next node visited in $Z_{\mathcal{F}'}$ is a'_1 (lines 21 and 22, Algorithm 1). Then, the remainder of the path in $Z_{\mathcal{F}'}$ follows the added nodes; at each a'_i, the high and low children are constructed to agree with the values of A. Finally, the last node along this path is $\mathbf{0}$ (line 15, Algorithm 1), so $Z_{\mathcal{F}'}(A') = 0$.

Furthermore, if $A' \neq A$, then consider the first index i where the characteristic vectors of A and A' differ; if $i < \text{var}(c)$, then the modifications to $Z_{\mathcal{F}'}$ have no effect on whether A' is accepted, since the only nodes added to $Z_{\mathcal{F}'}$ appear at depths greater than or equal to that of c. However, if $i \geq \text{var}(c)$, the path will follow along the newly added nodes $a'_1, a'_2, \ldots, a'_j$, where $\text{var}(a'_j) = i$. At

this point, A and A' differ, and by construction, the path returns to the original node in $Z_{\mathcal{F}}$ and never returns to the newly-added nodes. Therefore, $Z_{\mathcal{F}'}(A') = Z_{\mathcal{F}}(A')$, as desired. ■

To reduce the size of $Z_{\mathcal{F}'}$, 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 3c). Finally, note that the ZDD produced by the `RestrictSet` is no longer necessarily minimal with respect to $\mathcal{F}'$. In particular, in the worst case, if all 2^n subsets of $\mathcal{E}$ 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 $Z_{\mathcal{F}}$ that can be merged.

Figure 3: The result of applying the `RestrictSet` algorithm to $Z_{\mathcal{F}}$ from Figure 1a with $A = \{e_1, e_2, e_3, e_4\}$. The final ZDD accepts $\mathcal{F}' = \{\varnothing, \{e_1, e_2\}, \{e_3, e_4\}\}$.



(a) The path P_A is in bold; the split node c is the first node along this path with indegree larger than 1. The parent of the split node is b.

(b) Copies of nodes a_1 and a_2 are created, and the high edge from b points to this new path. The new path points to **0**, thus restricting the set A.

(c) The new ZDD $Z_{\mathcal{F}-A}$; since both high and low edges of a'_2 point to **0**, it can be suppressed. a'_1 is also suppressed to satisfy the zero-suppressed property.

Using the `RestrictSet` procedure, a branch-and-price algorithm can be developed that uses traditional integer branching. This branch-and-price 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 $n \ll |Z_{\mathcal{F}}|$, the increase in size of the ZDD over the course of the branch-and-price 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 branch-and-price search is given in Algorithm 2.

Algorithm 2: Branch-and-Price with ZDDs

1 Construct $Z_{\mathcal{F}}$, where $\mathcal{F}$ is the set of valid columns
2 Compute an initial pool $\mathcal{S}'$ of columns for the RMP
3 **for each** $A \in \mathcal{S}'$**:** $Z_{\mathcal{F}} = \texttt{RestrictSet}(Z_{\mathcal{F}}, A)$
4 Initialize the branch-and-price search tree T
5 ⟪ *Main branch-and-price loop* ⟫
6 **while** T has an unexplored subproblem**:**
7   Select a subproblem s that has not been explored
8   Generate children of s according to branching rule
9   **for each** child of s**:**
10   ⟪ *Column generation loop* ⟫
11   **while** $\exists$ a variable in $\mathcal{S} \setminus \mathcal{S}'$ with negative reduced cost**:**
12   Use $Z_{\mathcal{F}}$ to generate a variable $A \in \mathcal{S} \setminus \mathcal{S}'$ with negative reduced cost
13             $\texttt{RestrictSet}(Z_{\mathcal{F}}, A)$
14   Add A to $\mathcal{S}'$ and re-optimize the RMP
15   Apply pruning rules to delete child, or insert child into T
16   Mark s as explored
17 return the best solution found in T

2.3 The Maximal Independent Set ZDD

The graph coloring problem is a classic NP-hard problem (Garey and Johnson, 1979); given a graph $G = (V, E)$, the objective is to find a minimum *proper coloring* of vertices (i.e., a coloring in which no adjacent vertices share colors). The *chromatic number* χ of G is the minimum number of colors required in any proper coloring. For a vertex v, the *neighborhood* of v, denoted by $N(v)$, is the set of vertices adjacent to v. For a subset $S \subseteq V$, the *induced subgraph* $G[S]$ is the subgraph of G with vertex set S that has an edge between $u, v \in S$ if and only if (u, v) is an edge in G. A set $S \subseteq V$ is

an *independent set* if $G[S]$ has no edges, and S is a *maximal* independent set if there is no vertex $v \in V \setminus S$ such that $S + v$ is independent. For any set of vertices S, a vertex v is *covered* if $v \in S$ or $v \in N(S)$. Finally, a set $S \subseteq V$ is a *clique* if all pairs of vertices in S are adjacent.

The integer programming formulation for graph coloring used in most state-of-the-art solvers, initially proposed by Mehrotra and Trick (1996), is as follows:

$$\begin{aligned} \text{minimize} \; & \sum_{S \in \mathcal{S}} x_S \\ \text{subject to} \; & \sum_{S : v \in S} x_S \geq 1 \;\forall\; v \in V \\ & x_S \in \{0,1\}. \end{aligned} \tag{1}$$

In this formulation, $\mathcal{S}$ is the (exponential) 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 x_S indicate whether the maximal independent set S is used in the coloring, and the constraints ensure that each vertex in the graph appears in some color class.

The pricing problem for the graph coloring problem as formulated in (1) is a maximum-weight maximal independent set problem, where the weights on the vertices are given by the values of the dual variables of the RMP. If a maximal independent set S with weight larger than 1 is found, then variable x_S has negative reduced cost, which means that x_S is a candidate to improve the solution value of the RMP and can be added to $\mathcal{S}'$.

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). However, in this paper, a ZDD is built characterizing the family of maximal independent sets of G, and is used with a standard $0-1$ branching method (called *variable branching* by Malaguti et al., 2011).

To build a ZDD for the pricing problem of formulation (1), fix an ordering $\{v_1, v_2, \ldots, v_n\}$ on the vertices of the graph, and for some vertex $u \in V$, let $u-1$ be the vertex immediately preceding u in this ordering (in this setting, the vertex set V plays the role of $\mathcal{E}$). This section describes

a recursive algorithm for building the minimal ZDD (with respect to this vertex ordering) that characterizes $\mathcal{F} = \{A \mid A \text{ is a maximal independent set in } G\}$.

The maximal independent set ZDD is stored as a lookup table; if a is an index in this table, the lookup table stores var(a), lo(a), hi(a), and $\delta^-(a)$. In addition, to facilitate the merging of isomorphic regions of the ZDD, a reverse lookup table is stored that maps a tuple (i, b, c) to an index a such that $\text{var}(a) = i$, $\text{lo}(a) = b$, and $\text{hi}(a) = c$, if such an a exists in $Z_{\mathcal{F}}$. This reverse table is implemented as a hash table which allows for average constant insertion and lookup time.

A recursive construction algorithm for $Z_{\mathcal{F}}$, called `MakeIndSetZDD`, can be formulated following the general approach given in Knuth (2008). At each stage, a set U of k vertices and an index $i \leq k + 1$ is given as input to `MakeIndSetZDD`; U is the set of uncovered vertices for some (not necessarily maximal) independent set $R \subseteq \{v_1, v_2, \ldots, u_i - 1\}$. Here, vertices $\{u_i, u_{i+i}, \ldots, u_k\}$ can still be added to this (hypothetical) set R. To construct the ZDD node corresponding to U and i, the high child b_h and low child b_ℓ must first be constructed. To do this, vertex u_i and all its neighbors are removed from U to form a set U_H, and h is set to the index (in U_H) of the first vertex appearing after u_i in the vertex ordering, or $|U_H| + 1$, if no such vertex exists. Then, to compute b_h, `MakeIndSetZDD`(U_H, h) is called; this mimics the addition of vertex u_i to an independent set R at the current node. Conversely, to compute the low child, `MakeIndSetZDD`$(U, i+1)$ is called, which forbids vertex u_i from being used in R. To ensure minimality of $Z_{\mathcal{F}}$, before a node corresponding to some set U and index i is inserted, the algorithm checks to see if any node a exists in the ZDD with $\text{var}(a) = i$, $\text{hi}(a) = b_h$, and $\text{lo}(a) = b_\ell$. If such a node exists, the index of that node is returned; otherwise a new node is inserted.

In the base case, $i = k + 1$ (that is, no more vertices can be added to an independent set from the current node). If U is empty, all vertices in G are covered by an independent set, so the algorithm returns **1**; if U is not empty, there is some uncovered vertex in U, so the algorithm returns **0** (in fact, a tighter base case can be developed by observing that if there is some vertex in $\{u_1, u_2, \ldots, u_{i-1}\}$ that is not adjacent to any vertex in $\{u_i, u_{i+1}, \ldots, u_k\}$, it is impossible to build a maximal independent set from the current ZDD node).

Pseudocode for `MakeIndSetZDD` is given in Algorithm 3; to build $Z_{\mathcal{F}}$, `MakeIndSetZDD`(V, v_1) is called. An example of running `MakeIndSetZDD` on the graph in Figure 4 is given in Figure 5.

Note that `MakeIndSetZDD` does not actually maintain the vertices that are used in a current

Algorithm 3: MakeIndSetZDD(U, i)

input: A set $U = \{u_1, u_2, \ldots, u_k\}$ of uncovered 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** $G[U]$ cannot be covered by taking all vertices in $\{u_i, u_{i+1}, \ldots, u_k\}$**:** return **0**
2 **if** $U == \varnothing$**:** return **1**
3 $U_H = U - u_i - N(u_i)$ ⟪ *Use vertex u_i; remove it and its neighbors from U* ⟫
4 $h = \min\{j \mid u_j > u_i$ and $u_j \in U_H\}$ or $|U_H| + 1$ if no such j exists
5 $b_h =$ `MakeIndSetZDD`(U_H, h)
6 $b_\ell =$ `MakeIndSetZDD`$(U, i + 1)$
7 **if** $b_h == \mathbf{0}$**:** return b_l
8 ⟪ *Use reverse lookup table* ⟫
9 **if** $\exists\, a \in Z_{\mathcal{F}}$ s.t. $\text{var}(a) = i, \text{lo}(a) = b_\ell,$ and $\text{hi}(a) = b_h$**:** return a
10 **else:** return $Z_{\mathcal{F}}$.`insert`(i, b_ℓ, b_h)

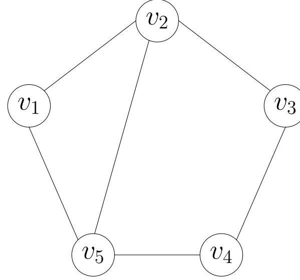


Figure 4: An example graph with a vertex ordering.

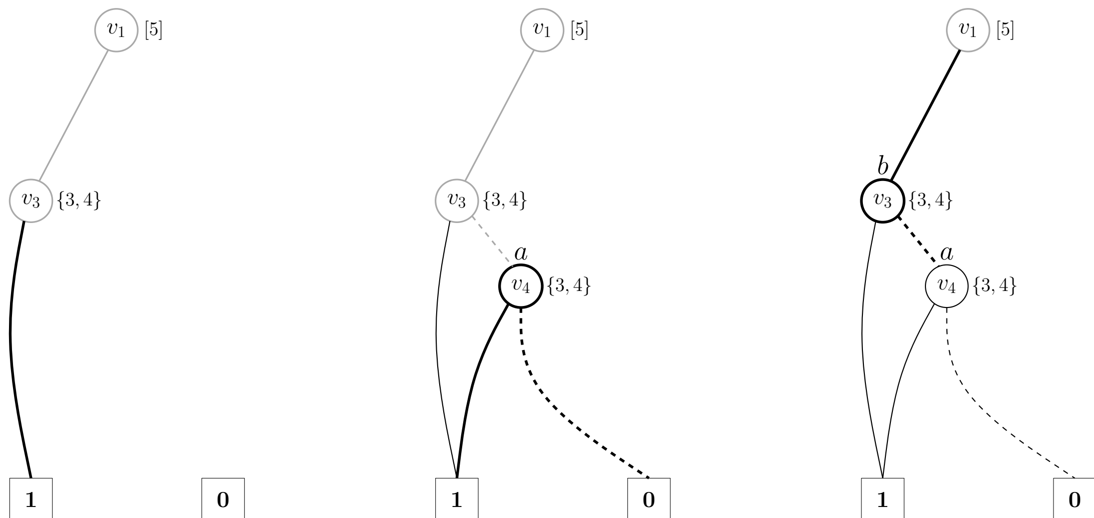
independent set R during the ZDD construction. It is sufficient to maintain a list of vertices that are left uncovered by *some* independent set, since many independent sets may yield the same set of uncovered vertices.

A number of different ordering heuristics can be applied to the vertex set of G to derive ZDDs of varying size. The rule that was empirically found to produce the smallest ZDDs is the maximal path decomposition rule, which computes a set of paths $P_1, P_2, \ldots, P_q$ such that P_i is maximal in $G[V - \bigcup_{j=1}^{i-1} P_j]$. The vertices are then ordered as

$$v_1^1, v_2^1, \ldots, v_{l_1}^1, v_1^2, v_2^2, \ldots, v_{l_2}^2, v_1^q, v_2^q, \ldots, v_{l_q}^q,$$

where v_i^j is the ith vertex along the path P_j, and l_j is the length of path P_j. Morrison et al. (2014c) show that the number of nodes associated with the kth vertex in this ordering is bounded by the

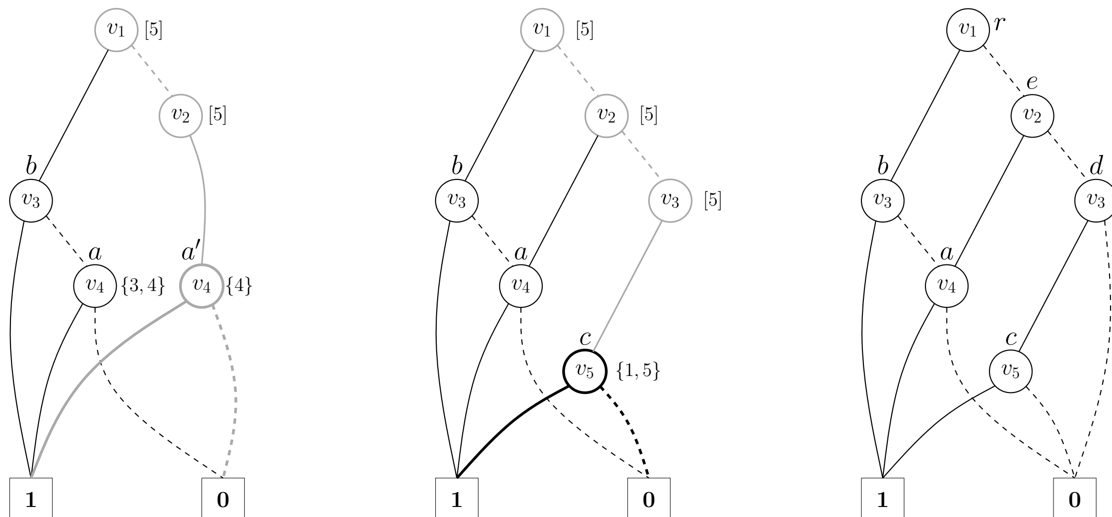
Figure 5: A visualization of the steps taken by `MakeIndSetZDD` to build $Z_{\mathcal{F}}$ for the example graph given in Figure 4. Grey nodes and edges have been visited by a recursive call, but are not yet stored in the ZDD. Black nodes and edges have been stored in the ZDD's lookup table. Bold elements have been inserted in the most recent step of the algorithm; nodes are labeled 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 a is the first node inserted in the ZDD.

(c) Both children of b, the high branch of the root, have been computed, so b 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 a' is computed as the high child, but is not inserted because it is a duplicate of node a.

(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 $\text{lo}(d) = \mathbf{0}$. All branches are now complete and the algorithm terminates.

k^{th} Fibonacci number F_k.

3 Cyclic Best-First Search

As described in Section 1, when using standard integer branching in a branch-and-price 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. For instance, in a problem with many covering constraints of the form $\sum_i x_i \geq b$, setting a variable $x_i = 0$ generally does not change the lower bound much, nor does it restrict the solution considerably since many other unfixed variables could also satisfy the constraint.

Therefore, standard search strategies such as depth-first search (DFS) or best-first search (BFS) do not perform particularly well in this setting. If DFS gets unlucky, it can start exploring some long chain early in the process which does not improve the incumbent solution but requires a large amount of search time. On the other hand, a strategy like BFS which relies on the lower bound to perform node selection will also perform poorly, since the lower bounds along the long branches will often be smaller than lower bounds in other parts of the tree.

Historically, the iterative deepening depth-first search (IDFS) strategy (Korf, 1985) has been used in such settings to prevent the exploration of long chains that do not make progress towards better incumbents. However, this paper proposes the use of the cyclic best-first search (CBFS) strategy as an alternative search strategy that enables the use of lower bound information during subproblem selection. The CBFS strategy, originally proposed by Kao et al. (2009) (and called distributed best-first search in their paper), has since been used successfully in a number of additional settings including two different scheduling problems (Morrison et al., 2014b; Sewell et al., 2012).

This search strategy can be thought of as a hybrid algorithm between DFS and BFS; the algorithm uses a measure-of-best function μ to select the next subproblem to explore (as in BFS), but repeatedly cycles through a set of labeled *contours* (i.e., a collection of subproblems), selecting one subproblem from each contour to explore before advancing to the next contour. The cyclic behavior can be thought of as a variant of backtracking in DFS. The contour labels are simply

taken from $\mathbb{N}_0$ as a way to order the set of contours. For example, the levels of the search tree provide a natural contour definition, where subproblems are grouped by their distance from the root of the tree (see Figure 6a).

Let C_i denote the contour with label i; Once a subproblem has been explored from C_i, CBFS chooses the next subproblem for exploration from C_{i+p}, where $p = \min\{p' \in \mathbb{Z}^+ \mid C_{i+p'} \neq \varnothing\}$, and index addition is done modulo K (the largest contour label currently in use). The subproblem chosen from this contour is one that minimizes the measure-of-best function μ. In contrast, BFS always chooses the subproblem with the best (global) value of μ to explore. Pseudocode for the CBFS strategy is given in Algorithm 4; this code is called in Line 7 of Algorithm 2 to select a new subproblem for exploration.

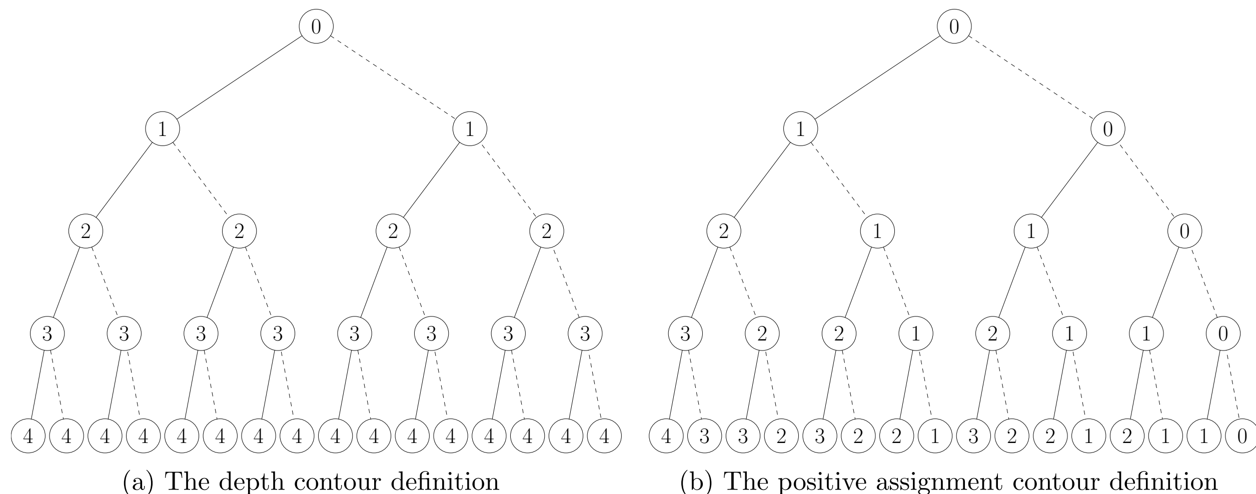
Previous implementations of CBFS have only used the depth-based contour definition; however, this contour definition does not produce better performance than DFS or BFS, for the same reason that BFS performs poorly. In fact, CBFS can produce worse performance than DFS in some instances of the graph coloring problem, for instance if DFS gets lucky and finds a good incumbent early. The key insight provided here is that other, more complicated, contour definitions are possible which may dramatically improve the search process for branch-and-price algorithms for graph coloring.

In particular, consider the following contour definition (called the *positive assignment* definition), which assigns a subproblem to contour ℓ if and only if there have been ℓ branching decisions made of the form $x_i = 1$ (called a positive assignment). Using this contour definition significantly restructures the order in which subproblems are selected for exploration (see Figure 6b). In particular, using this definition prohibits the immediate exploration of a child subproblem which assigns $x_i = 0$, even if the lower bound at this child is better than the lower bound at the $x_i = 1$ child. In this way, the search is weighted towards exploration of subproblems that make positive assign-

Algorithm 4: CBFS

```
1 Let i be the label of the contour containing the last explored subproblem
2 if ∃ C_j ≠ ∅ with j > i:
3     Let j be the first index larger than i of a non-empty contour
4 else:
5     Let j be the first index in {0, 1, ..., i} of a non-empty contour
6 return s ∈ arg min_{s'∈C_j} μ(s')
```

Figure 6: Subproblems in a search tree with two different contour functions. Dashed lines indicate an assignment of 0 to the variable at that subproblem. The number in the center of each node is the label of the contour that subproblem is assigned to.



(a) The depth contour definition

(b) The positive assignment contour definition

ments, which serves to counterbalance the effects of a lopsided search tree. In essence, the positive assignments can be thought of as discrepancies in limited discrepancy search (Harvey and Ginsberg, 1995; Korf, 1996): most of the maximal independent sets in G will not be used, but a few, the discrepancies, will be.

4 Computational Results

A branch-and-price algorithm for the graph coloring problem 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 (Johnson and Trick, 1996; Trick, 2005). This section describes some implementation details for this program, called `B&P+ZDD`, and discusses the results of these experiments and a comparison to the best algorithms in the literature.

4.1 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 C in the graph, and any vertex $v \in V$ with degree less than $|C|$ is removed. Since a valid coloring for G must use at least $|C|$ 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 branch-and-bound search is employed in a heuristic manner to find an initial large clique. The clique C can also be used to prove optimality – if a proper coloring of G is found that uses exactly $|C|$ 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 et al. (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 et al. (2008).

Any column generated by the initialization routine can be added to the initial pool $\mathcal{S}'$ 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 $\mathcal{S}'$. This rule includes all sets with negative or close-to-negative reduced cost in $\mathcal{S}'$, since these sets are more likely to improve upon the LP solution to the RMP in early stages of the search.

4.2 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 branch-and-price 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 four different branch-and-price 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. The compared results were obtained using standard $0-1$ branching instead of edge branching. Secondly, Gualandi and Malucelli (2012) describe a branch-and-price solver for graph coloring that uses constraint programming techniques to solve the pricing problem; their implementation uses the edge branching rule. Morrison et al. (2014a) provide extensive computational results using a wide branching technique, which modifies the standard $0-1$ branching rule to allow multiple children to be generated from a subproblem in the search tree. Finally, Held et al. (2012) provide a method for computing a numerically safe lower bound for graph coloring, which they embed inside a branch-and-price solver. Using this algorithm, they are able to prove new lower bounds for a number of unsolved instances.

Instance	Name of the tested instance
n	Number of vertices in the instance
m	Number of edges in the instance
χ	Chromatic number of the instance, if known
LB	Lower bound found by `B&P+ZDD`
UB	Best solution found by `B&P+ZDD` (if $t < 10hrs$, this value is optimal)
t_Z	Time needed to construct the maximal independent set ZDD
t	Time spent in the branch-and-price phase of the algorithm
$t_{\text{MMT-init}}$	Total initialization time used by Malaguti et al. (2011)
t_{MMT}	Adjusted time to verify optimality by Malaguti et al. (2011)
t_{wide}	Adjusted time to verify optimality by the wide branching solver of Morrison et al. (2014a)
$t_{\text{CP-BnP}}$	Adjusted time to verify optimality by the branch-and-price solver of Gualandi and Malucelli (2012)
exp	Number of nodes explored in the search tree
id	Number of nodes identified in the search tree
Z_i	Initial size of the ZDD
Z_f	Final size of the ZDD
% change	Percent change in size of the ZDD over the course of the algorithm
|col|	Number of columns generated over the course of the algorithm
t_{price}	Time spent solving all pricing problems over the course of the algorithm

Table 1: Notation used for computational results data (Tables 2 and 3).

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

periments 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).

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 4.1 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. Data for all 47 instances are shown in Table 2, and notation used in this table is given in Table 1. To provide the most fair comparison between different computational platforms, all running times are scaled according to the benchmark value of the *dfmax* utility reported. In most cases, Held et al. (2012) is concerned with computing lower bounds instead of total solution times, so data from this paper are omitted from Table 2 (though they are still discussed in the sequel).

Computational comparisons across differing models, source code, and platforms is notoriously difficult, and these data are no exception. The MMT algorithm uses an initialization procedure with a variable running time, depending on the difficulty of the problem. However, `B&P+ZDD` is given a flat 100 seconds of initialization, plus the length of time required to build the ZDD; moreover, the initialization procedure used by `B&P+ZDD` is strictly weaker than that of the MMT algorithm, since it only uses the first phase of a two-phase procedure. This choice was made to highlight the performance benefits of using ZDDs; in principle, if `B&P+ZDD` had access to the full initialization procedure used by the MMT algorithm, its results would be even better. Therefore, to produce the fairest comparison between the algorithms, the compared running times are solely the time spent on the branch-and-price algorithm, not including the initialization time.

`B&P+ZDD` was able to find and verify optimality for 15 of the 47 instances tested. In four cases, `B&P+ZDD` is able to find and verify optimality at least an order of magnitude faster than any competing algorithm. Additionally, `B&P+ZDD` is able to verify optimality for three new instances

Instance	n	m	χ	LB	UB	t_Z	t	$T_{\text{MMT-init}}$	t_{MMT}	t_{wide}	$t_{\text{CP-BnP}}$
DSJC125.5	125	3891	17	16	17	0.47	284.92	6100	17019.33	225.21	14372.75
DSJC125.9	125	6961	44	43	44	0	0.21	6100	3674.22	1.02	33.23
DSJC250.5	250	15668	?	26	28	29.55	>10hrs	6100	>10hrs	>10hrs	>10hrs
DSJC250.9	250	27897	72	71	72	0.04	649.90	6100	>10hrs	>10hrs	>10hrs
DSJC500.5	500	62624	?	43	53	3458.5	>10hrs	6100	>10hrs	>10hrs	-
DSJC500.9	500	112437	?	123	129	0.38	>10hrs	6100	>10hrs	>10hrs	>10hrs
DSJC1000.5	1000	249826	?	10	108	6608.86	oom	6100	>10hrs		-
DSJC1000.9	1000	449449	?	215	228	5.64	>10hrs	6100	>10hrs	>10hrs	-
DSJR500.1c	500	121275	85	85	85	0.14	0.10	6100	272.01	1.29	0.60
DSJR500.5	500	58862	122	122	122	689.48	102.75	6100	322.6	6862.28	>10hrs
le450_25c	450	17343	25	25	28	19889.92	oom	6100	init	>10hrs	-
le450_25d	450	17425	25	25	28	16262.45	oom	6100	init	>10hrs	-
queen9_9	81	1056	10	9	10	0.44	9.33	3	34.51	20.48	74.00
queen10_10	100	2940	11	10	11	4.09	140.76	3	647.65	587.32	26393.20
queen11_11	121	3960	11	11	11	33.1	14354.16	3	1759.08	19208.45	21858.50
queen12_12	144	5192	12	12	13	297.91	>10hrs	3	>10hrs	>10hrs	-
queen13_13	169	6656	13	13	14	2739.39	>10hrs	100	>10hrs	>10hrs	-
queen14_14	196	8372	14	14	15	3139.2	>10hrs	100	>10hrs	>10hrs	-
queen15_15	225	10360	15	15	16	3235.39	>10hrs	100	>10hrs	>10hrs	-
queen16_16	256	12640	16	16	18	3527.63	>10hrs	100	>10hrs	>10hrs	-
myciel3	11	23	4	3	4	0	0	3	0	0.01	-
myciel4	20	71	5	4	5	0	0.09	3	111.26	0.47	-
myciel5	47	236	6	4	6	0.01	392.21	3	-	3207.63	-
myciel6	95	755	7	4	7	0.76	>10hrs	3	>10hrs	>10hrs	-
myciel7	191	2360	8	5	8	2198	>10hrs	3	>10hrs	>10hrs	-
1-Insertions_4	67	232	5	3	5	0.69	>10hrs	3	>10hrs	>10hrs	-
1-Insertions_5	202	1227	?	2	6	23708.86	>10hrs	3	>10hrs	>10hrs	-
2-Insertions_4	149	541	?	2	5	>10hrs	-	3	>10hrs	>10hrs	-
3-Insertions_3	56	110	4	3	4	1.8	>10hrs	3	>10hrs	>10hrs	-
3-Insertions_4	281	1046	?	2	5	22684.73	>10hrs	3	>10hrs	>10hrs	-
4-Insertions_3	79	156	4	3	4	314.85	>10hrs	3	>10hrs	>10hrs	-
1-FullIns_4	93	593	5	4	5	8.76	122.58	3	>10hrs	>10hrs	-
1-FullIns_5	282	3247	6	3	6	>10hrs	-	3	>10hrs	>10hrs	-
2-FullIns_4	212	1621	6	0	0	>10hrs	-	3	>10hrs	>10hrs	-
2-FullIns_5	852	12201	7	4	7	>10hrs	-	3	>10hrs	>10hrs	-
3-FullIns_4	405	3524	7	5	7	>10hrs	-	3	>10hrs	>10hrs	-
4-FullIns_4	690	6650	8	0	0	>10hrs	-	3	>10hrs	>10hrs	-
latin_square_10	900	307350	?	90	100	36.4	>10hrs	6100	>10hrs	>10hrs	-
qg.order30	900	26100	30	0	0	>10hrs	-	3	0.19	>10hrs	-
wap06	947	43571	40	0	0	>10hrs	-	170	165.00	>10hrs	-
r250.5	250	14849	65	65	65	14.63	7.15	-	-	-	6.80
r1000.1c	1000	485090	?	96	98	2.29	>10hrs	-	-	-	>10hrs
r1000.5	1000	238267	234	234	234	43020.09	4690.16	-	-	-	>10hrs
flat300_28_0	300	21695	28	28	28	144.57	19883.45	-	-	-	>10hrs
flat1000_50_0	1000	245000	?	10	106	6795.93	>10hrs	-	-	-	>10hrs
flat1000_60_0	1000	245830	?	10	105	6368.64	>10hrs	-	-	-	>10hrs
flat1000_76_0	1000	246708	?	12	106	6628.52	>10hrs	-	-	-	>10hrs

Table 2: Results from computational experiments with B&P+ZDD, 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. Entries of 0.00 indicate that the length of time is lower than the precision of the timer, and a dash indicates that the information is not available.

(1-FullIns_4, r1000.5, and flat_300_0) that have not been solved previously by branch-and-price 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 branch-and-price 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) and Morrison et al. (2014a)) 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.

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.

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 94% 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.

Details regarding these data are shown in Table 3, with column headings again given in Table 1. Here, note that the number of nodes explored in the tree (column 4) is the total number of nodes at which children were generated. The number of nodes identified in the tree (column 5) includes those nodes for which the RMP was solved, but could be pruned away before generating children. Finally, the last column in this table shows the total CPU time taken to solve every pricing problem for each instance. As the bulk of the work to solve an instance is in solving the RMP and solving the pricing problem, the time needed to solve the RMP for a particular instance is approximately

Instance	n	m	exp	id	Z_i	Z_f	% change	$\lvert\text{col}\rvert$	t_{price}
DSJC125.5	125	3891	599	1199	48367	52207	7.9	3462	7.9
DSJC125.9	125	6961	55	111	623	627	0.6	81	0
DSJC250.5	250	15668	5122	10244	1476916	1511171	2.3	37404	4108.62
DSJC250.9	250	27897	16411	32823	2893	2960	2.3	808	2.88
DSJC500.5	500	62624	25	50	83507135	83509619	0.003	4519	30791.1
DSJC500.9	500	112437	46634	93268	15397	15913	3.4	5259	25.07
DSJC1000.5	1000	249826	-	-	-	-	-	-	-
DSJC1000.9	1000	449449	6466	12932	102909	105366	2.4	24892	141.52
DSJR500.1c	500	121275	18	37	2443	2447	0.2	68	0.03
DSJR500.5	500	58862	74	149	1809872	2222124	22.8	8	14.77
le450_25c	450	17343	-	-	-	-	-	-	-
le450_25d	450	17425	-	-	-	-	-	-	-
queen9_9	81	1056	11	23	50719	54727	7.9	216	0.34
queen10_10	100	2940	76	153	295500	308571	4.4	1088	11.99
queen11_11	121	3960	1902	3805	1867378	1923347	3	16394	2138.55
queen12_12	144	5192	1224	2448	12426874	12526852	0.8	24743	18255.37
queen13_13	169	6656	98	195	88797420	88874820	0.09	4985	33826.38
queen14_14	196	8372	3	6	100000008	100000008	0	5	37809.97
queen15_15	225	10360	-	-	-	-	-	-	-
queen16_16	256	12640	-	-	-	-	-	-	-
myciel3	11	23	4	9	29	29	0	0	0
myciel4	20	71	191	383	152	169	11.2	0	0.02
myciel5	47	236	160622	321245	1429	2188	53.1	26	35.7
myciel6	95	755	94395	188789	40191	70326	75	3658	357.24
myciel7	191	2360	7816	15632	7191878	7344883	2.1	3742	8653.59
1-Insertions_4	67	232	72106	144211	85112	106550	25.2	12596	443.94
1-Insertions_5	202	1227	-	-	-	-	-	-	-
2-Insertions_4	149	541	-	-	-	-	-	-	
3-Insertions_3	56	110	29128	58256	94885	183834	93.7	38580	394.57
3-Insertions_4	281	1046	-	-	-	-	-	-	-
4-Insertions_3	79	156	14177	28353	6585989	6693239	1.6	12115	12196.88
1-FullIns_4	93	593	7422	14845	148275	155237	4.7	1376	51.53
1-FullIns_5	282	3247	-	-	-	-	-	-	-
2-FullIns_4	212	1621	-	-	-	-	-	-	-
2-FullIns_5	852	12201	-	-	-	-	-	-	-
3-FullIns_4	405	3524	-	-	-	-	-	-	-
4-FullIns_4	690	6650	-	-	-	-	-	-	-
latin_square_10	900	307350	5265	10529	52807	52807	0	13310	26.35
qg.order30	900	26100	-	-	-	-	-	-	-
wap06	947	43571	-	-	-	-	-	-	-
r250.5	250	14849	25	51	137683	266893	93.8	0	0.57
r1000.1c	1000	485090	215082	430163	11762	11997	2	1298	165.64
r1000.5	1000	238267	192	385	37664084	38165484	1.3	1317	3741.64
flat300_28_0	300	21695	1004	2009	5817662	5858003	0.7	24665	8699.34
flat1000_50_0	1000	245000	-	-	-	-	-	-	-
flat1000_60_0	1000	245830	-	-	-	-	-	-	-
flat1000_76_0	1000	246708	-	-	-	-	-	-	-

Table 3: Detailed statistics for the computational experiments with B&P+ZDD. Cells with dashes indicate that the information is not available due to time limits or memory constraints.

t (column 8, Table 2) minus t_{price}.

In order to assess the efficacy of the search strategy, additional computational experiments were run against the 12 instances from the DIMACS database that `B&P+ZDD` was able to solve within the time limit. `B&P+ZDD` was modified for these tests to use DFS instead of CBFS, and the results are presented in Table 4.

Instance	t_{DFS}	t_{CBFS}
`DSJC125.5`	7689.02	284.92
`DSJC125.9`	0.17	0.21
`DSJC250.9`	1587.94	649.90
`DSJR500.1c`	0.07	0.10
`DSJR500.5`	103.70	102.75
`queen9_9`	11.31	9.33
`queen10_10`	134.07	140.76
`queen11_11`	32057.94	14354.16
`myciel4`	0.08	0.09
`myciel5`	244.37	392.21
`1-FullIns_4`	59.04	122.58

Table 4: A comparison of the running times of `B&P+ZDD` using DFS and CBFS.

In these experiments, there are 3 instances in which CBFS (with the positive assignment rule) significantly outperforms DFS, and 2 instances in which DFS outperforms CBFS. For the remaining 7 instances, both algorithms perform within a few seconds of each other. For the three instances where CBFS outperformed DFS, the improvement was an order of magnitude in one case, and over twice as fast in the other two cases. Therefore, it was concluded that in general, CBFS with the positive assignment rule is a better choice of branching strategy for the graph coloring problem than DFS.

5 Conclusions and Future Work

This paper presents a framework for using standard integer branching in conjunction with branch-and-price 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 com-

bined with a new contour definition for the cyclic best-first search strategy to counterbalance the resulting lopsided search tree, the standard integer branching scheme can be used in conjunction with a branch-and-price algorithm, which yields a much faster and more direct solution method in many cases. Computational results were presented showing that a branch-and-price algorithm implementation for the graph coloring problem in some cases outperforms other branch-and-price graph coloring solvers in the literature. In several cases, this performance is an improvement of an order of magnitude or more, though an exact comparison is difficult due to differences in initialization.

A number of future research directions exist for this method; firstly, this paper proposes a ZDD algorithm for the graph coloring problem. However, there is nothing specific to graph coloring in Sections 2.1 and 2.2; in fact, the techniques described here are general, and could be applied to other problems. Some preliminary results using ZDDs with the generalized assignment problem are described in Morrison (2014), but more work must be done to show their effectiveness on other types of problems. Moreover, ZDDs can also be used even if the branch-and-price solver does not require the solution of the constrained pricing problem (for instance, the robust branch-and-price-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. Thus, additional research can be done to study how ZDDs interact with other established branch-and-price methods.

Secondly, research can be performed to determine the best way to use ZDDs when the entire data structure will not fit in memory; one proposed idea uses approximate ZDDs (described in Bergman et al., 2012b) to solve the pricing problem in these settings. An approximate ZDD is a width-constrained ZDD that does not eliminate any valid solutions to the pricing problem, but may accept some inputs that are not valid solutions to the pricing problem. In this setting, a post-generation check can be performed to see if the ZDD produced a set that is a valid solution to the pricing problem; if not, the `RestrictSet` routine can be called with the erroneous solution to prevent it from being generated again. If the approximate ZDD can be constructed in an appropriate fashion, it is hypothesized that invalid solutions will be generated relatively infrequently, and the algorithm will not suffer much loss of efficiency.

A final important question addresses the addition and removal of restrictions from the pricing problem ZDD. Many standard branch-and-price algorithms will generate multiple columns in between each intermediate solution of the RMP. It has been observed that this can improve algorithm performance; thus, an interesting research direction may be to modify the ZDD algorithm to generate multiple columns in a single pass through the data structure. Moreover, in many cases the variable pool for branch-and-price algorithms may grow quite large over the course of the algorithm. Since the size of this pool directly impacts the solution time for the RMP, and most of the elements of this pool are never used in any optimal solutions, most branch-and-price solvers will prune the pool by deleting variables with very large positive cost. In such a setting, it is necessary to modify $Z_{\mathcal{F}}$ to allow removed variables to be generated again; these variables can be stored in an auxiliary pool to be scanned before the ZDD is queried. Thus, future research should analyze the effects of such a modification.

Acknowledgments

The computational results reported were obtained at the Simulation and Optimization Laboratory at the University of Illinois, Urbana-Champaign. This research has been supported in part by the Air Force Office of Scientific Research (FA9550-10-1-0387, FA9550-15-1-0100), 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(s) and do not necessarily reflect the views of the United States Government. The authors would also like to thank Jason Sauppe for conversations and help regarding the branch-and-price implementation, as well as Enrico Malaguti for the source code used in our initialization procedure. Finally, the authors thank the two anonymous referees and the associate editor for comments which resulted in a much-improved draft of this paper.

References

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

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

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

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

Behle, M. 2007. Binary decision diagrams and integer programming. Ph.D. thesis, Universistät des Saarlands.

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

Bergman, D., A. Cire, W. V. Hoeve, J. Hooker. 2012a. Optimization bounds from binary decision diagrams. Tech. Rep. 1378, Tepper School of Business.

Bergman, D., A. A. Cire, W. V. Hoeve, J. N. Hooker. 2012b. Variable ordering for the application of BDDs to the maximum independent set problem. N. Beldiceanu, N. Jussien, . Pinson, eds., *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems*. No. 7298 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 34–49.

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

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

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

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

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

Dantzig, G. B., P. Wolfe. 1960. Decomposition principle for linear programs. *Operations Research* **8** 101–111.

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

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

Garey, M. R., D. S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. First edition ed. W. H. Freeman.

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

Hadžić, T., J. Hooker. 2008. Postoptimality analysis for integer programming using binary decision diagrams. Tech. Rep. 167, Tepper School of Business.

Harvey, W. D., M. L. Ginsberg. 1995. Limited discrepancy search. *IJCAI (1)*. 607–615.

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

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

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

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

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

Korf, R. E. 1996. Improved limited discrepancy search. *AAAI/IAAI, Vol. 1*. 286–291.

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

Maenhout, B., M. Vanhoucke. 2010. Branching strategies in a branch-and-price approach for a multiple objective nurse scheduling problem. *Journal of Scheduling* **13** 77–93.

Malaguti, E., M. Monaci, P. Toth. 2008. A metaheuristic approach for the vertex coloring problem. *INFORMS Journal on Computing* **20** 302 –316.

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

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

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

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

Morrison, D. R. 2014. New methods for branch-and-bound algorithms. Ph.D. thesis, University of Illinois, Urbana-Champaign.

Morrison, D. R., J. J. Sauppe, E. C. Sewell, S. H. Jacobson. 2014a. A wide branching strategy for the graph coloring problem. *INFORMS Journal on Computing* **26** 704–717.

Morrison, D. R., E. C. Sewell, S. H. Jacobson. 2014b. An application of the branch, bound, and remember algorithm to a new simple assembly line balancing dataset. *European Journal of Operational Research* **236** 403–409.

Morrison, D. R., E. C. Sewell, S. H. Jacobson. 2014c. Characteristics of the maximal independent set ZDD. *Journal of Combinatorial Optimization* **28** 121–139.

Pessoa, A., M. P. de Aragao, E. Uchoa. 2008. Robust branch-cut-and-price algorithms for vehicle routing problems. *The vehicle routing problem: Latest advances and new challenges*. Springer, 297–325.

Pisinger, D., M. Sigurd. 2007. Using decomposition techniques and constraint programming for solving the two-dimensional bin-packing problem. *INFORMS Journal on Computing* **19** 36–51.

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

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

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

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

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

Vance, P. H. 1998. Branch-and-price algorithms for the one-dimensional cutting stock problem. *Computational Optimization and Applications* **9** 211–228.

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