A Wide Branching Strategy for the Graph Coloring Problem

David R. Morrison, Jason J. Sauppe, Edward C. Sewell, Sheldon H. Jacobson

Abstract

Branch-and-price algorithms for the graph coloring problem use an exponentially-sized independent set based integer programming formulation to produce usually tight lower bounds to enable more aggressive pruning in the branch-and-bound tree. One major problem inherent to any branch-and-price scheme for graph coloring is that in order to avoid destroying the pricing problem structure during column generation, difficult-to-implement branching rules that modify the underlying graph must be used. This paper proposes an alternate branching strategy that does not change the graph in order to solve the pricing problem, but rather modifies the search tree to require fewer calls to difficult instances of the pricing problem. This approach, called **wide branching**, generates many subproblems at each node in the branch-and-price tree, which significantly reduces the length of any path through the search tree. In contrast, traditional **deep branching** only creates two subproblems per node, assigning a variable to either zero or one. A **delayed branching** procedure is introduced that prevents the branching factor at any particular node from growing too large in this scheme. Finally, computational results are presented that show the wide branching strategy to be competitive with state-of-the-art graph coloring solvers.

1 Introduction and Background

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). Therefore, it is important to derive new, efficient algorithms for this problem that are able to solve difficult instances exactly.

There is a substantial body of literature on this topic that covers both exact and heuristic methods that have been applied to the graph coloring problem. Galinier and Hertz (2006) provide a comparison of twenty different heuristic methods that have been applied to this problem; these methods include tabu search, simulated annealing, and steepest-descent algorithms. Malaguti and Toth (2010) describe a number of heuristic and exact approaches to graph coloring; a few of these exact methods are described in greater detail herein.

One of the first effective exact algorithms for graph coloring is the DSATUR (degree of saturation) algorithm of Brélaz (1979). This algorithm implicitly enumerates all possible colorings for the graph by choosing an uncolored vertex with the highest **degree of saturation**—that is, the vertex whose colored neighbors use the largest number of distinct colors—and assigning it an available color. If no previously-used color is available, a new color is introduced for this vertex. Despite its simplicity, it is an effective method for computing near-optimal colorings for many problem instances even if it is unable to prove optimality in a reasonable time. This algorithm was improved by Sewell (1996) by incorporating better upper bound heuristics and using branching rules to guide the search more intelligently.

A significant breakthrough for the graph coloring problem was made with the introduction of the well-known algorithm of Mehrotra and Trick (1996); this method operates on a reformulation of the graph coloring problem with an exponential number of variables. This formulation eliminates symmetries that obstruct the search process in DSATUR and its variants. However, due to the size of the reformulation, it is necessary to use branch-and-price to solve the model. Their algorithm was the first to use branch-and-price methods for the graph coloring problem; in fact, most exact graph coloring solvers today still use this fundamental idea. To avoid the destruction of the pricing problem structure during column generation, they propose a branching scheme that modifies the structure of the graph by adding edges or merging vertices.

Malaguti (2008) describes a new algorithm for graph coloring that combines the branch-and-price algorithm with an effective initialization heuristic. This heuristic method employs local search techniques together with an evolutionary algorithm to generate an initial solution as well as a good initial pool to seed the column generation procedure (Malaguti et al., 2008). Extensive computational results using this heuristic together with the standard branch-and-price algorithm are presented in Malaguti et al. (2011). The resulting algorithm is one of the best exact graph coloring algorithms available in the literature.

Similar work by Gualandi and Malucelli (2012) uses constraint programming techniques to solve the problem, both as a direct solution method and as a subroutine during column generation for a branch-and-price algorithm. They also present a suite of computational results for their algorithms that are comparable to those of Malaguti et al. (2011). Finally, Held et al. (2012) report new methods for calculating “safe” lower bounds in a branch-and-price setting. These bounds are safe in the sense that they are provably resilient to round-off error that may be introduced by finite-precision machine arithmetic.

The primary contribution of this paper is the introduction of an alternate branching strategy for the branch-and-price algorithm of Mehrotra and Trick (1996). This new strategy, termed **wide branching**, does not modify the underlying graph structure during column generation, but rather, rearranges and condenses the search tree to allow for fewer calls to difficult instances of the pricing problem. Additionally, a novel **delayed branching** technique is described that limits the branching factor at nodes in the branch-and-price tree to a manageable size. Finally, computational experiments are presented showing that an implementation of the wide branching strategy outperforms or is competitive with two different branch-and-price solvers using the traditional $0-1$ branching strategy (called **deep branching** in this paper).

Some similar work has been done in this area previously; Elhedhli et al. (2011) present a branch-and-price algorithm for the bin packing problem with conflicts that creates multiple branches at each node 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. Finally, Vanderbeck (2011) describes a new method for automatically determining branching rules in a generic branch-and-price solver so as to avoid destruction of the pricing problem.

The remainder of this paper is organized as follows: Section 2 describes the problem and the standard branch-and-price algorithm. Section 3 gives a theoretical motivation for the wide branching strategy, and describes its application to the graph coloring problem. In Section 4, additional implementation details are discussed that improve the performance of the wide branching solver, and Section 5 presents a computational comparison of an implementation of branch-and-price with wide branching to other methods in the literature. Finally, Section 6 provides some concluding remarks and directions for future research. Relevant notation used throughout the paper is given in Table 1.

2 Graph Coloring with Branch-and-Price

Given a graph $G = (V, E)$, where $n = |V|$ and $m = |E|$, the graph coloring problem seeks a minimum **proper coloring** (that is, an assignment of colors to vertices such that no adjacent vertices receive the same color). The **chromatic number** χ of G is the minimum number of colors needed for a proper coloring. This problem has two standard integer programming representations; the first creates binary variables x_{vj} and q_j, where $v \in V$ and $j \in \{1, 2, ..., n\}$. Using these variables, the following integer program encodes the graph coloring problem (Mehrotra and Trick, 1996):

$$\begin{aligned} \text{minimize } & \sum_{j=1}^{n} q_j \\ \text{subject to } & \sum_{j=1}^{n} x_{vj} = 1, \ \forall \ v \in V \\ & x_{uj} + x_{vj} \leq q_j, \ \forall \ uv \in E, j \in \{1, 2, ..., n\} \\ & x_{vj}, q_j \in \{0, 1\}, \ \forall \ v \in V, j \in \{1, 2, ..., n\}. \end{aligned} \tag{1}$$

In an optimal solution to (1), if a variable x_{vj} is set to one, this means that vertex v should receive color j, and q_j is an indicator variable that is one if color j is assigned to some vertex. While the formulation in (1) has some nice properties (namely, relatively small size and ease of implementation), it has two significant disadvantages. First, this formulation has many symmetric solutions. By permuting color classes, one can transform a particular (partial) coloring into a different, equivalent coloring that must be considered by a solver, but provides it with no new information. Additionally, the LP relaxation of (1) is typically quite weak, and thus does not yield good bounds to effectively prune suboptimal regions of the search space. To address these issues, a different formulation is proposed by Mehrotra and Trick (1996) that uses binary variables x_S indexed by $S \in \mathcal{S}$, the set of maximal **independent sets** in G (an independent set S is a collection of nonadjacent vertices; it is maximal if every vertex of G is either in S or adjacent to a vertex in S):

$G = (V, E)$	the input graph with its vertex and edge sets
u, v	vertices of G
n, m	graph parameters: $n = \|V\|, m = \|E\|$
χ (χ_f)	the (fractional) chromatic number of G
x_S, x_{uv}, q_j	binary variables in the problem formulations
y_u	binary variables in the pricing problem
z	the objective function value for the RMP
$\pi(v)$	the value of the dual variable corresponding to vertex v
$\mathcal{S}; \mathcal{S}'$	the collection of all maximal independent sets of G; the restricted pool of maximal independent sets used in the RMP
S	a maximal independent set in G
T	a branch-and-price search tree
$ub(T)$	the value of the incumbent solution in T
a, b, c	nodes of T
$lb(a)$	the value of the lower bound at node $a \in T$
$X^1(a)$ ($X^0(a)$)	the collection of variables with positive (null) assignments at node $a \in T$
$d_{sat}(v)$	the "degree-of-saturation" function at vertex v (i.e., the number of differently-colored neighbors of v in some partial solution $\mathcal{S}^1_a$; the partial solution is usually implicit from context)

Table 1: A summary of notation used.

$$\begin{aligned} \text{minimize} \quad & \sum_{S \in \mathcal{S}} x_S \\ \text{subject to} \quad & \sum_{S : v \in S} x_S \geq 1, \ \forall \ v \in V \\ & x_S \in \{0, 1\}, \ \forall \ S \in \mathcal{S}. \end{aligned} \tag{2}$$

Since any proper coloring can be viewed as a partition of V into independent sets, the formulation in (2) also encodes the graph coloring problem. However, because independent sets are not explicitly assigned colors, this model eliminates the symmetry inherent in (1); additionally, in practice the LP relaxation of this model has demonstrated very tight bounds in conjunction with branch-and-price. The main disadvantage of this alternate formulation is that the number of maximal independent sets of G is generally exponential in the order of the graph, so it is infeasible to generate and store the entire integer program at once. Thus, it is necessary to resort to branch-and-price techniques to solve (2).

A branch-and-price algorithm is an extension of standard branch-and-bound algorithms for solving integer programs. At each node of the branch-and-price tree, the LP relaxation of (2) is solved to optimality to obtain the **fractional chromatic number** χ_f, which is a lower bound on the value of (2); if the LP relaxation produces an integer solution or if its objective value is worse than the incumbent's, the node can be pruned. Otherwise, subproblems (or branches) are created to force some fractional variables away from their current values, and these subproblems are re-optimized subject to those additional constraints.

Since the number of problem variables is large, solving the LP relaxation of (2) requires the use of column generation (Barnhart et al., 1998). This procedure explicitly maintains a restricted pool $\mathcal{S}'$ of maximal independent sets of G, and solves the LP relaxation of (2) over $\mathcal{S}'$ instead of $\mathcal{S}$. This restricted problem is known as the **restricted master problem** (RMP). While the LP relaxation over $\mathcal{S}'$ can be easily solved with any linear programming solver, it may not produce an optimal solution, so new "good" variables (corresponding to columns of the constraint matrix of (2)) must be introduced to the RMP via column generation.

Define $\{\pi\}$ to be the set of dual variables for the constraints in the RMP. The reduced cost of any independent set $S \in \mathcal{S}$ can be computed as $1 - \sum_{v \in S} \pi(v)$; if a set with negative reduced cost can be found, this set will improve the solution to the RMP, unless it leads to a degenerate pivot. Conversely, if no such independent set exists, the solution to the RMP is optimal. Thus, finding variables with negative reduced cost reduces to solving a separation problem (or **pricing problem**) in the dual space of the RMP. The integer program for the pricing problem is provided below:

$$\begin{aligned} \text{minimize } & 1 - \sum_{v \in V} \pi_v y_v \\ \text{subject to } & y_u + y_v \leq 1 \ \forall \ uv \in E \\ & y_v \in \{0,1\} \ \forall \ v \in V. \end{aligned} \tag{3}$$

The objective of (3) can be rewritten as $\max \sum_{v \in V} \pi_v y_v$ to see that the problem modeled by this integer program is a maximum-weight independent set problem. If the optimal solution to this problem is greater than one, a new variable has been found that can be added to $\mathcal{S}'$. On the other hand, if the optimal solution to the pricing problem has value less than or equal to one, the current solution to the RMP is optimal. Note that for graph coloring, the pricing problem is itself NP-hard (Garey and Johnson, 1979). Therefore, any fast branch-and-price algorithm for graph coloring must contain two important components:

(i) Fast column generation algorithms that can solve the RMP and the pricing problems quickly,

(ii) Intelligent tree-search and pruning methods for exploring the branch-and-price tree.

The deep branching strategy for solving integer programs (called **variable branching** by Malaguti et al. (2011)) chooses a fractional variable at the current node in the search tree, and creates two children, one in which the variable is forced to zero (called a **null assignment**), and the other in which it is forced to one (called a **positive assignment**). Unfortunately, this approach is not effective for the graph coloring problem, because the pricing problem becomes substantially more difficult to solve at nodes where null assignments have been made. In particular, note that if some variable x_S is set to zero, the pricing problem must now find a negative-reduced-cost independent set *that is not* S, or report that no such set exists. When one or more null assignments have been made, the corresponding pricing problem is referred to as the **constrained pricing problem**. Solving the constrained pricing problem typically requires significantly more computation power than the unconstrained pricing problem and is the primary bottleneck in algorithms using this branching method (Barnhart et al., 1998). In addition, when using this technique, the resulting search tree is unbalanced, because adding a null assignment does not constrain the problem significantly, and thus long chains exist in the search tree where no progress is made towards a solution.

Therefore, conventional wisdom for branch-and-price based graph coloring algorithms dictates the use of branching rules that do not destroy the structure of the pricing problem in this manner. The most well-known such branching rule, proposed by Mehrotra and Trick (1996) (called **edge branching** by Malaguti et al. (2011)), selects an independent set S that is fractionally used in the RMP, and two nonadjacent vertices $u, v \in S$, and creates two subproblems, one where u and v have been contracted to a single vertex (thus forcing them to each receive the same color) and another where an edge between u and v is added (thus forcing them to be colored differently). However, this method requires each node of the branch-and-price tree to operate on a different graph, which means that each node must explicitly store a list of modifications to G. In addition to increased memory overhead, this approach is often more difficult to implement.

The wide branching strategy takes an alternate approach to solving the problem. The fundamental observation driving this strategy is that the fragility in the pricing problem is asymmetric:

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. This can be seen by the fact that finding a maximal independent set in G, given that some independent set S has already been taken, is equivalent to finding a maximal independent set in $G - S$. The next section presents the wide branching strategy and describes its application to the graph coloring problem.

3 The Wide Branching Strategy

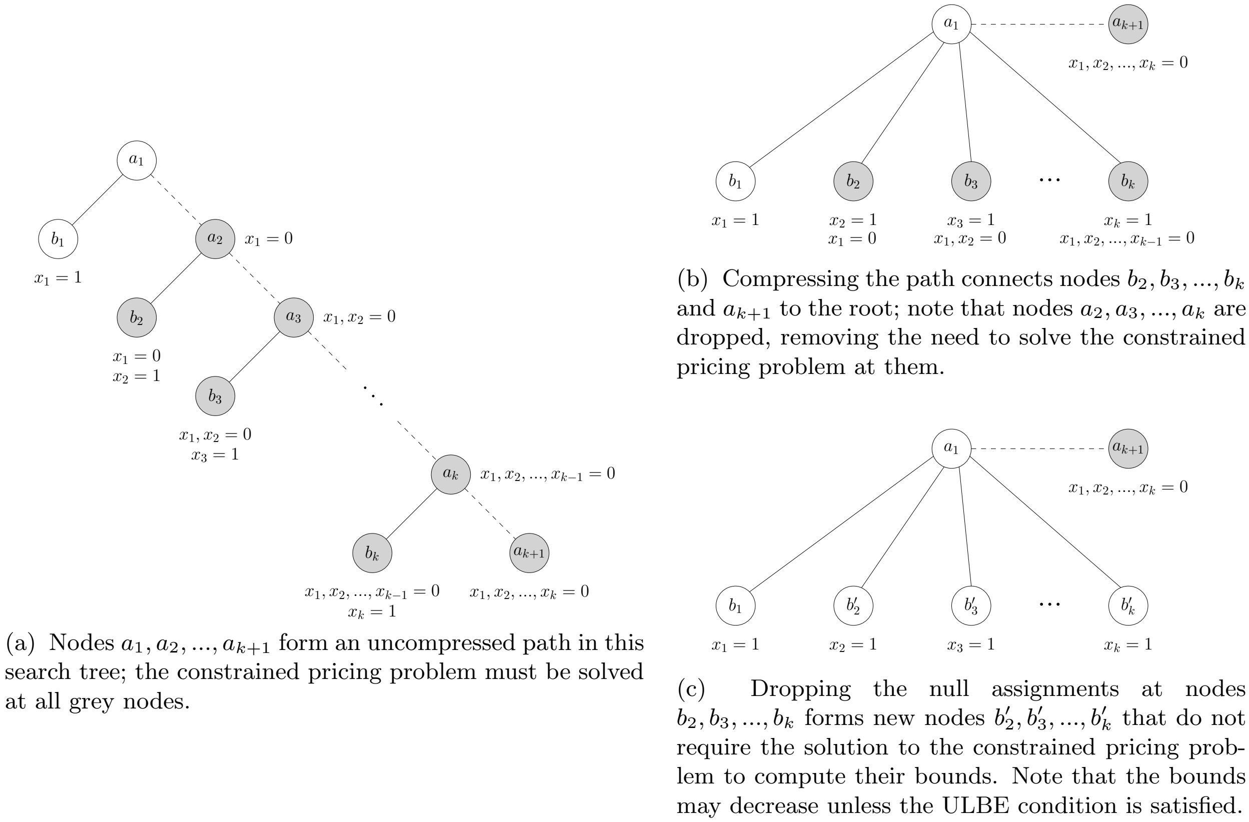
This section motivates the wide branching strategy by demonstrating how some restructuring operations transform a fully explored branch-and-price tree T into a related search tree T' that requires the solution of the constrained pricing problem at fewer nodes than T. A description of the restructuring operations (called **path compression** and **forgetful branching**) are presented here, along with theoretical results bounding the number of nodes 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 section, $\mathcal{P}$ is defined as a combinatorial optimization problem with an integer optimal solution, and $x_1, x_2, ..., x_n$ are binary variables in an integer programming formulation for $\mathcal{P}$. For a node a in a branch-and-price search tree T for $\mathcal{P}$, the partial assignment at node a is given by $X(a) = \big(X^1(a), X^0(a)\big)$, where $X^1(a)$ and $X^0(a)$ are the (disjoint) sets of variables that have been fixed to one and zero, respectively.

3.1 Path Compression and Forgetful Branching

Consider a branch-and-price search tree T for $\mathcal{P}$ that uses the deep branching strategy. As observed in Section 2, 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 $2k + 1$ nodes, and the constrained pricing problem must be solved at $2k - 1$ nodes (see Figure 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 node. After path compression, the root of the path has $k+1$ children, and the constrained pricing problem is only solved at k nodes instead of the original $2k-1$ (Figure 1b).



(a) Nodes $a_1, a_2, ..., a_{k+1}$ form an uncompressed path in this search tree; the constrained pricing problem must be solved at all grey nodes.

(b) Compressing the path connects nodes $b_2, b_3, ..., b_k$ and a_{k+1} to the root; note that nodes $a_2, a_3, ..., a_k$ are dropped, removing the need to solve the constrained pricing problem at them.

(c) Dropping the null assignments at nodes $b_2, b_3, ..., b_k$ forms new nodes $b'_2, b'_3, ..., b'_k$ 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 1: Using path compression and forgetful branching can reduce the number of times the constrained pricing problem is solved.

To further reduce the number of nodes 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 1c).

While forgetful branching reduces the number of children of a_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 nodes in the search tree. For instance, setting $x_i = 1$ and then $x_j = 1$ is equivalent to setting $x_j = 1$ and then $x_i = 1$. These two paths both lead to the same partial solution which is represented by two distinct nodes in the tree. However, if the algorithm keeps track of all generated subproblems, simple dominance rules can prevent identical subproblems from being explored multiple times.

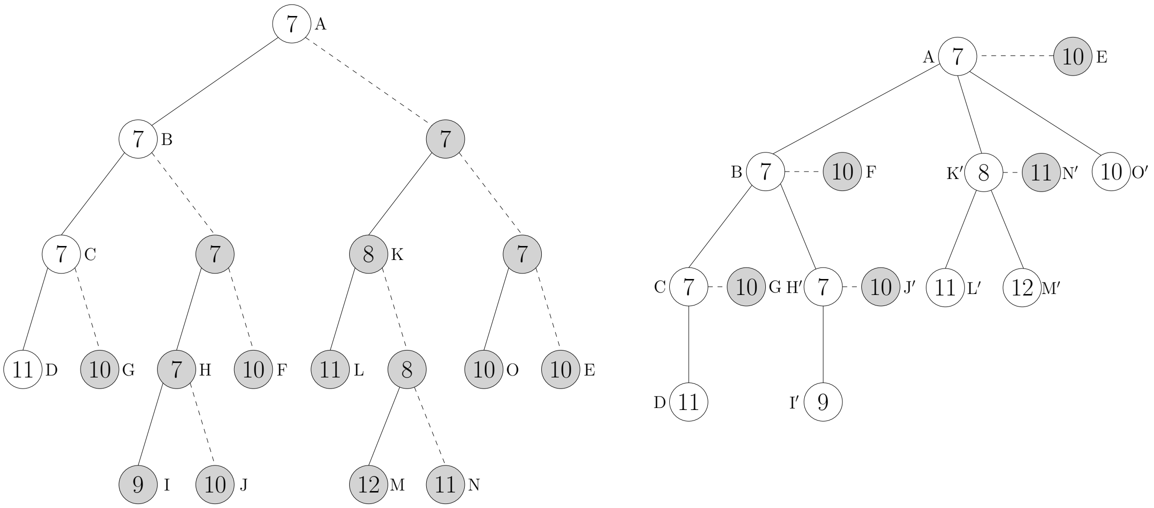
By iteratively applying the path compression and forgetful branching operations to a branch-and-price search tree T, a new search tree T' can be created which requires the solution of the constrained pricing problem at fewer nodes in the tree. This result is proved formally in the Wide Branching Theorem, 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 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 x_i is given a positive assignment at a node a in T, a branch is also created at a making a null assignment to x_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 node 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 1. *Let $X = (X^1, X^0)$ be a partial assignment of values to variables for a problem $\mathcal{P}$. Let z^*_X be the optimal solution value to the LP relaxation subject to the partial assignment X, and let $z^*_{\overline{X}}$ be the optimal LP solution to the partial assignment $\overline{X} = (X^1, \varnothing)$, that is, the partial assignment to $\mathcal{P}$ which matches all positive assignments of X, but has no null assignments. If for every pair of partial assignments X and $\overline{X}$, $\lceil z^*_X \rceil = \lceil z^*_{\overline{X}} \rceil$, then the problem $\mathcal{P}$ 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 node. Since the LP relaxation may become weaker when a constraint imposing a null assignment ($x_i = 0$) at a node is removed, if the ULBE condition is not satisfied, it could be the case that the lower bound shrinks and no longer allows a node 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 (proof given in Appendix A).

Theorem 1 (The Wide Branching Theorem)**.** *Let $\mathcal{P}$ be a binary minimization problem with an integer optimal solution satisfying the ULBE condition. Given a branch-and-price tree T for $\mathcal{P}$ that contains an uncompressed path P, there exists a branch-and-price tree T' that has strictly fewer nodes by collapsing P into a single vertex.*



(a) A branch-and-price tree produced by the standard deep branching rule; computed bounds are shown in the center of each node. Grey nodes require the solution of the constrained pricing problem.

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

Figure 2: Repeated path compressions produce a minimal branch-and-price tree

Repeated application of Theorem 1 produces a search tree containing no uncompressed paths (see Figure 2). Applying forgetful branching after each path compression operation (as in Figure 1) causes the number of nodes requiring the solution of the constrained pricing problem to decrease.

Thus, the branch-and-price algorithm that obeys the branching decisions in T' will solve $\mathcal{P}$ more quickly than the one that generates T.

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

3.2 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 node. 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 node a in the search tree:

1. An independent set S_1 is chosen with the most fractional value (closest to 0.5) in the LP relaxation at node a.

2. A vertex $v = \arg\max_{u \in S_1} d_{sat}(u)$ is selected with the highest degree of saturation in S_1 (that is, v has the most (integral) differently-colored neighbors among all vertices in S_1).

3. For each of the k independent sets $S_1, S_2, ..., S_k$ which contain v and have $x_{S_i} > 0, i \in \{1, 2, ..., k\}$, create one child of a that performs a positive assignment to x_{S_i}. Additionally create one child of a that gives a null assignment to all sets $S_1, S_2, ..., S_k$, as well as any null assignments generated at a previous stage (see the discussion of delayed branching, below). This last node is called the **delayed node**, and is denoted $\bar{a}$.

The branching strategy above attempts to emulate the path compression and forgetful branching process by guessing variables $S_1, S_2, ..., S_k$ which compose some uncompressed path rooted at a. However, it may be the case that the entire uncompressed path could not be guessed by the above branching rule. In this case, $\bar{a}$ 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**.

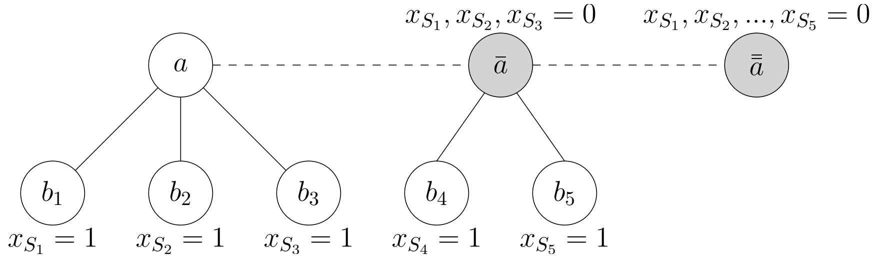


Figure 3: An example of the delayed branching process. x_{S_1}, x_{S_2}, and x_{S_3} are guessed as members of an uncompressed path rooted at a. A node $\bar{a}$ restricts these sets from being generated, but $\bar{a}$ cannot be pruned. Later, x_{S_4} and x_{S_5} are generated at $\bar{a}$, and do not inherit the null assignments at $\bar{a}$, and a node $\bar{\bar{a}}$ is generated with sets $x_{S_1}, x_{S_2}, ..., x_{S_5}$ restricted.

Later, when the algorithm returns to $\bar{a}$, it attempts to guess more nodes along the uncompressed path rooted at a; in particular, to avoid repetition of previously-generated states, each of $x_{S_1}, x_{S_2}, ..., x_{S_k}$ must not be generated as children of $\bar{a}$, so these variables are imposed as null assignments at $\bar{a}$. Suppose that some new independent sets $\bar{S}_1, \bar{S}_2, ..., \bar{S}_{\bar{k}}$ are generated as children of $\bar{a}$; since each of these nodes are on an uncompressed path rooted at a, by the forgetful branching technique, they drop the null assignments posted at $\bar{a}$. Finally, a node $\bar{\bar{a}}$ (the new delayed node) is added. If $\bar{\bar{a}}$ is the end of an uncompressed path rooted at a, no further children need to be generated. However, if $\bar{\bar{a}}$ is *not* the end of such a path, new nodes must be guessed that do not use $S_1, S_2, ..., S_k$ or $\bar{S}_1, \bar{S}_2, ..., \bar{S}_{\bar{k}}$. In other words, when children of a delayed node are generated, all independent sets branched on at previous delayed nodes must be restricted to zero (see Figure 3).

Finally, note that a slightly tighter bound at the delayed nodes can be achieved as follows: let node $\bar{a}$ be the delayed node for a, and let node b be a child of $\bar{a}$. At the delayed node of b (that is, $\bar{b}$), the children of a and b both can be restricted at $\bar{b}$. In other words, the forgetful branching rule is applied at all children of b *except* for its delayed node. This does not affect algorithm performance, since the constrained pricing problem must be solved at $\bar{b}$ regardless. For example, in Figure 3, the null assignments posted at node $\bar{a}$ can be propagated to the delayed nodes for b_4 and b_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 S_1 and S_2 are generated at each node in the search tree (since S_1 is chosen to be a fractionally-used independent set, there must be at least one other such set covering the chosen node 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 nodes 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.

4 Implementation Details

A version of the wide branching rule was developed for a graph coloring branch-and-price 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 1.

Algorithm 1: B&P+Wide

```
Initialize $\mathcal{S}'$ and $UB$ by calling InitMMT ⟪ Section 4.4 ⟫
while unexplored nodes remain:
    Select best node $a$ from current level to explore ⟪ Section 4.2 ⟫
    Generate sets $S_1, S_2, ..., S_k$ via the wide branching rule ⟪ Section 3.2 ⟫
    for each $S_i \in \{S_1, S_2, ..., S_k\}$:
        Create a child $b_i$ of $a$ with $x_{S_i} = 1$
        if $b_i$ is dominated by some other node: prune $b_i$ and continue ⟪ Section 4.3 ⟫
        Compute $lb(b_i)$ by solving RMP with column generation techniques ⟪ Sections 4.1,
          4.5 ⟫
        if $\lceil lb(b_i) \rceil \geq UB$: prune $b_i$ and continue
        else if $b_i$ has a better integral solution than the incumbent: update $UB$
        else: Insert $b_i$ into the search tree
    ⟪ Delayed branching, Section 3.2; $R$ is the set of other restrictions that should be
      imposed at $\bar{a}$ ⟫
    Create a node $\bar{a}$ with $\{x_{S_1}, x_{S_2}, ..., x_{S_k} = 0\} \cup R$
    Compute $lb(\bar{a})$ by solving RMP with constrained pricing problem
    Prune $\bar{a}$ via LP bounds, or insert into the tree otherwise
return $UB$
```

4.1 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.

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

4.1.2 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 nodes 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 node has been exhausted, the currently-used vertices at that node are appended to the no-good list. In the future, any node 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).

4.2 Exploration Strategy

The cyclic best-first search (CBFS) strategy is used to select new nodes for exploration. This strategy, described in Sewell et al. (2012), operates in a similar manner to best-first search (BFS) in that it uses heap data structures to store information about promising regions of the search space. Unlike BFS, however, CBFS maintains a different heap for each level of the search tree, and it repeatedly cycles through the levels of the tree, selecting the next node to explore as the best node from among all nodes at the current level of the tree, using the efficient heap operations to store and retrieve nodes. Specifically, once a node has been explored at level i, the next node chosen for exploration comes from level $i + 1 \pmod{D}$, where D is the current depth of the search tree. In contrast, BFS always chooses the best known (global) node to explore.

CBFS has two primary advantages over other search methods. The first is that, unlike breadth-first search (BrFS) or BFS, it is able to produce complete, often good, solutions early in the search process. This is beneficial, because the earlier a good incumbent can be found, the more of the search tree can be pruned. Additionally, unlike BFS and depth-first search (DFS), the cyclic nature of the algorithm prevents the search process from getting stuck in poor regions of the search space that do not lead to good solutions, and has been shown to be quite effective in several settings. In contrast, DFS often exhibits **thrashing**, where many similar, bad solutions need to be pruned before the search can move to a different region of the space; in BFS, many nodes in middle levels of the tree will have similar scores, and thus complete solutions may not be generated often.

Additionally, in the wide branching context, CBFS yields a natural re-exploration rule for nodes that have been re-inserted to the search tree by delayed branching. Specifically, the next time the level containing a previously-visited node is visited, if the re-inserted node's bound is still the best in the level, it is chosen for re-exploration (recall that when a node 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 node to be pruned).

4.3 Dominance Checks

The nature of the wide branching strategy is such that two distinct branches of the search may be identical. In particular, if S_1 and S_2 are two independent sets under consideration, one branch could select S_1 first and then S_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 nodes 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 node 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 node, B&P+Wide maintains a global hash table of previously-explored nodes; any time a new node $a \in T$ is generated, a lookup is performed in the hash table to determine if another node $b \in T$ 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 $X^1(a)$ (the set of positive assignments at a); 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.

4.4 Preprocessing and Initialization

To reduce the size of problem instances, B&P+Wide uses a standard preprocessing technique: a search is done to find a large clique C in the graph. Then any vertex $v \in V$ whose degree is less than $|C|$ can be removed, since a valid coloring for G uses at least $|C|$ colors, which means that there will be at least one color not assigned to any neighbor of v. Thus, v can be colored without needing to increase the number of colors used (Méndez-Díaz and Zabala, 2006). A branch-and-bound search using CBFS is employed in a heuristic manner to find such a large clique.

To start the branch-and-price algorithm, an initial pool of independent sets needs to be generated. A good method for creating this pool is described in Malaguti et al. (2008); this procedure employs a 2-phase approach to find a good initial solution. In the first phase, a genetic algorithm combined with a local search rule searches for valid k-colorings of the graph, given an input k. If a valid k-coloring is found, 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 one and applies a covering heuristic to improve the solution further. A similar initialization procedure, called `InitMMT`, is used by `B&P+Wide` to produce good initial solutions. This procedure only runs the first phase of the algorithm described in Malaguti et al. (2008); the covering heuristic is not used to improve the results.

Any column generated by `InitMMT` can be taken to be in the initial pool $\mathcal{S}'$ of independent sets used by `B&P+Wide`; however, since `InitMMT` can generate a large number of independent sets, the RMP is solved once to get an initial set of dual prices, and only sets generated by `InitMMT` with price above a threshold value of 0.8 are included in $\mathcal{S}'$, since these sets are more likely to improve upon the LP solution to the RMP, at least in early stages of the search. Moreover, note that `InitMMT` actually generates a good feasible solution to the problem; this feasible solution is taken as the initial upper bound in the search tree, and all sets contained in this feasible solution are included in $\mathcal{S}'$.

4.5 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 (4), 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.

$$\pi'(u) = \begin{cases} \pi(u)/\pi(S) & \text{if } u \in S \\ \pi(u) & o.w. \end{cases} \tag{4}$$

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 node'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 node 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 Computational Experiments

`B&P+Wide` was implemented and tested on a subset of the graphs from the DIMACS implementation challenge testbed (Johnson and Trick, 1996; Trick, 2005); 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 branch-and-price algorithm was implemented in C++ and used CPLEX 12.3 to solve the RMP. The branch-and-price algorithm utilized only a single core of the processor; however, CPLEX operates in parallel by default. All times reported are aggregated over all cores.

All improvements described in Section 4 were used in both the wide and deep versions of the branch-and-price 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.

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`.

Held et al. (2012) observed that an important issue when solving graph coloring problems with a branch-and-price method is round-off error. In particular, when computing lower bounds for these problems, errors in machine precision could cause the lower bound to be greater than the actual value; this effect is amplified by the fact that the ceiling of the lower bound is used for pruning purposes. A number of methods have been proposed in the literature to counteract this effect, including the safe lower bound methods of Held et al. (2012). For simplicity, however, a more naïve approach is used in `B&P+Wide` and `B&P+Deep`: before the ceiling of the lower bound is computed, a tolerance value ϵ is subtracted from the solution value of the RMP; this may cause more work to be done than necessary, but should ensure that no node is pruned in error.

Due to implementation and memory requirements, only results for graphs with 1000 vertices or fewer are reported. To compare with the results reported by Malaguti et al. (2011), both the wide and deep solvers are allowed to run for 10 hours; additional time for the `InitMMT` initialization procedure is given that does not count towards the 10-hour limit.

5.1 Algorithm Comparisons

Results comparing `B&P+Wide`, `B&P+Deep`, and the MMT branch-and-price solvers are presented in Table 3. The first four columns describe characteristics of the problem instance; the next group of columns reports statistics about the initialization procedure. Finally, the last three groups of columns report results from the MMT algorithm, `B&P+Wide`, and `B&P+Deep`, respectively. Detailed information about each of the columns in this table are given in Table 2.

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 branch-and-price 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 4.1.2.

Note that Table 3 does not contain all problem instances in the DIMACS testbed; in particular, many of the instances in that dataset are too easy (either the `InitMMT` initialization procedure finds a coloring with chromatic number equal to the size of the large clique found in the preprocessing stage, or the lower-bound solution at the root of the search tree is sufficient to prove optimality). A few problem instances were also too difficult, in that the RMP at the root node could not be solved within the 10-hour time limit. These results do not demonstrate the usefulness of the wide branching strategy. Thus, only instances for which at least one node was explored are presented.

Name	Name of the problem instance
n	Number of vertices in the instance
m	Number of edges in the instance
χ	Chromatic number (if known)
t_{init}	Time (in CPU seconds) for `InitMMT` to find the initial upper bound solution branch-and-price algorithm
t_{lim}	Time (in CPU seconds) allocated to the initialization procedure; these values were chosen to be consistent with the time bounds used by Malaguti et al. (2011), and do not count toward the 10-hour time limit given for the branch-and-price portion of the algorithm.
UB	Value of best solution found by the algorithm
LB	Best lower bound on the optimal solution (taken from Malaguti et al. (2011) for the MMT algorithm; only reported for `B&P+Wide`, because `B&P+Deep` has the same root node)
t	Time (in CPU seconds) to solve the problem instance; if t is $> 10hrs$, the problem could not be solved to optimality by the algorithm within the time limit. In this case, the value in column *UB* is an upper bound on the optimal value. Otherwise, the value in *UB* is the optimal solution.
t_{best}	Time (in CPU seconds) for the algorithm to find a solution with value equal to the upper bound. If this column reads *init*, the solution was found by the initialization procedure. Malaguti et al. (2011) do not report this data.
exp/id	Number of nodes explored by the algorithm, versus number of nodes identified. Malaguti et al. (2011) do not report this data.

Table 2: Meaning of column headers in Table 3.

Of the remaining problem instances, most take more than the 10-hour time limit for all three solvers; this suggests that this set of problems is particularly difficult to solve. In particular, many of these graphs are relatively sparse and quite large, both characteristics which tend to prevent fast solutions, because many very similar independent sets will need to be generated before nodes can be pruned.

Of the fourteen problems 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 can be 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 branch-and-price). The third method, called CG-CP, uses

Graph Info				Initialization		MMT			Wide					Deep (DFS)			
Name	n	m	χ	t_{init}	t_{lim}	*LB*	*UB*	t	*LB*	*UB*	t	t_{best}	exp/id	*UB*	t	t_{best}	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†	*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†	*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 3: Comparison of the wide and deep branch-and-price solvers with the MMT branch-and-price solver. Grey cells indicate the solver with the fastest time to verify an optimal solution. †indicates the program ran out of memory before the time limit was reached.

constraint programming to solve the pricing problem in a branch-and-price 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 branch-and-price 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.

5.2 Analysis of Wide versus Deep Branching

UB	value of best solution found (if $t < 10$ hours, this is the optimal solution value)
t	total running time
t_{best}	time to best solution found
exp/id	number of explored nodes versus number of identified nodes
$\lvert 0 \rvert$	number of times the delayed branching procedure is called
$\lvert \text{cols} \rvert$	total number of generated columns
$\lvert \text{cols}_0 \rvert$	total number of columns generated from a constrained pricing problem
cols/sec	number of columns generated per second of computation time
max_{br}	maximum branching factor (number of children generated at a node)
avg_{br}	average branching factor

Table 4: Explanation of column headings for Tables 6 and 5.

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; Table headings for these results are given in Table 4, with raw data included in Tables 5 and 6. First, 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

Name	n	m	χ	UB	t	t_{best}	exp/id	$\|\mathbf{0}\|$	$\|\text{cols}\|$	$\|\text{cols}_0\|$	$cols/sec$	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†	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 5: An analysis of the wide branching solver for selected graph coloring instances. † indicates that the solver ran out of memory.

Name	n	m	χ	UB	t	t_{best}	exp/id	$\|\mathbf{0}\|$	|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†	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

Table 6: An analysis of the deep branching solver for selected graph coloring problems. † indicates that the solver ran out of memory.

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 nodes 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 3.2 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 nodes. Across all problem instances, the average branching factor was 13.1, and for problems which explored more than one node, the average branching factor was 10.3, indicating that as the search with the wide branching rule progresses, the branching factor at nodes is substantially reduced.

6 Conclusion

This paper describes a branching strategy for branch-and-price algorithms that solve the graph coloring problem. This branching strategy, instead of choosing a single fractional variable to set to either 0 or 1, selects many children at a node, eliminating restrictions that have been imposed to allow for the solution of the unconstrained pricing problem (which is much easier to solve). A theoretical result called the Wide Branching Theorem 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 branch-and-price 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 branch-and-price with deep branching for many problems.

There are a number of possible directions for future research on this problem; first, 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.

Moreover, note that the wide branching strategy is applicable in many different settings, not just graph coloring. Therefore, it will be interesting to determine how it performs for different problems; it is also important to understand how these results extend for problems that use cutting plane methods to improve LP bounds, or for problems that have general integer variables.

Finally, an important research direction will be to characterize the types of problems for which wide branching is most effective, as well as to prove theoretical guarantees on the performance of the strategy. In particular, it is conjectured that wide branching will be extremely effective for problems in which the unconstrained pricing problem can be solved in polynomial time, or for problems which do not have easy alternative branching rules that maintain pricing problem structure.

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), the Department of Defense (DoD) through the National Defense Science & Engineering Graduate Fellowship (NDSEG) Program, and the National Science Foundation through the Graduate Research Fellowship Program. Additionally, the fourth author was supported in part by (while serving at) the National Science Foundation. 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 Enrico Malaguti for making available the source code for the `InitMMT` initialization procedure, as well as two anonymous referees and the editors for their careful reviews and suggestions, which resulted in a much improved version of this paper.

A Proof of the Wide Branching Theorem

Proof. As in Figure 1, let $P = a_1a_2...a_{k+1}$ be a longest uncompressed path in T such that a_i ($i \in 1, 2, ..., k$) has two child branches that fix variable x_i to either 0 or 1 in the LP relaxation, and a_{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 b_i be the node in T hanging from P where a positive assignment has been made to x_i (see Figure 1a). To create T', duplicate all branching decisions made in T except at node a_1; at this point, create branches $b'_1, b'_2, ..., b'_k$ with a positive assignment made to x_i at branch b'_i. Finally, create a single node a_{k+1} that performs a null assignment for $x_1, x_2, ..., x_k$ (see Figure 1b). Since there are no children at node a_{k+1} in T, it must be the case that a_{k+1} is pruned in T'.

Furthermore, by the ULBE condition, note that $lb(b_i) = lb(b'_i)$, which means that node b'_i is pruned in T' if and only if b_i is pruned in T. To complete exploration of T', at each b'_i, exactly duplicate the branching decisions made in the subtree rooted at b_i in T; again by the ULBE condition, note that nodes in these subtrees are pruned in T' if and only if the corresponding nodes are pruned in T.

Therefore, since no new nodes are introduced into the branch-and-price tree, and the path P has been compressed into a single node a_{k+1}, the number of nodes in T' that require the solution of the constrained pricing problem are strictly fewer than in T. ■

References

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. *Oper. Res.* **46** 316–329.

Brélaz, D. 1979. New methods to color the vertices of a graph. *Comm. of the ACM* **22** 251–256.

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

Farley, A. A. 1990. A note on bounding a class of linear programming problems, including cutting stock problems. *Oper. Res.* **38** 922–923.

Galinier, P., A. Hertz. 2006. A survey of local search methods for graph coloring. *Computers & Oper. Res.* **33** 2547–2562.

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

Grosso, A., M. Locatelli, W. Pullan. 2008. Simple ingredients leading to very efficient heuristics for the maximum clique problem. *J. of Heuristics* **14** 587–612.

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

Held, S., W. Cook, E. C. Sewell. 2012. Maximum-weight stable sets and safe lower bounds for graph coloring. *Math. Programming Comput.* **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.

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

Malaguti, E. 2008. The vertex coloring problem and its generalizations. *4OR* **7** 101–104.

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

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

Malaguti, E., P. Toth. 2010. A survey on vertex coloring problems. *Internat. Trans. in Oper. Res.* **17** 1–34.

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

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

Mitra, G. 1973. Investigation of some branch and bound strategies for the solution of mixed integer linear programs. *Math. Programming* **4** 155–170.

Pardalos, P., T. Mavridou, J. Xue. 1998. The graph coloring problem: A bibliographic survey. D. Du, P. M. Pardalos, eds., *Handbook of Combin. Optim.*, vol. 2. Kluwer Academic Publishers, 331–395.

Sewell, E. 1996. An improved algorithm for exact graph coloring. D. S. Johnson, M. A. Trick, eds., *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, vol. 26. American Mathematical Society, 359–372.

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. *J. of Global Optim.* **54** 791–812.

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

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

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