A COLUMN GENERATION APPROACH FOR GRAPH COLORING

Anuj Mehrotra
Department of Management Science
School of Business Administration
University of Miami
Coral Gables, FL 33124-8237
e-mail: anuj@nirvana.bus.miami.edu

Michael A. Trick
Graduate School of Industrial Administration
Carnegie Mellon University
Pittsburgh, PA 15213-3890
e-mail: trick+@cmu.edu

April 11, 1995

Abstract

We present a method for solving the independent set formulation of the graph coloring problem (where there is one variable for each independent set in the graph). We use a column generation method for implicit optimization of the linear program at each node of the branch-and-bound tree. This approach, while requiring the solution of a difficult subproblem as well as needing sophisticated branching rules, solves small to moderate size problems quickly. We have also implemented an exact graph coloring algorithm based on DSATUR for comparison. Implementation details and computational experience are presented.

1 INTRODUCTION

The graph coloring problem is one of the most useful models in graph theory. This problem has been used to solve problems in school timetabling [10], computer register allocation [7, 8], electronic bandwidth allocation [11], and many other areas. These applications suggest that effective algorithms for solving the graph coloring problem would be of great importance.

Despite this relevance, there are relatively few methods available for solving graph coloring instances exactly. Those that are available ([21, 22]) are limited to solving small instances. In contrast, heuristic techniques have been designed that solve instances with hundreds or thousands of vertices ([14, 15, 25]) at the cost of regularly suboptimal solutions.

We suggest an approach based on an integer programming formulation of the graph coloring problem. This formulation, called the *independent set formulation*, has a variable for each independent set in the graph. While this formulation is well known, the enormous number of variables has apparently discouraged use of it as a computational method. We show that it is possible to develop an effective column generation technique for this problem while still ensuring integrality with appropriate branching rules. This method is tested on a variety of instances and is shown to be robust and more effective than previous techniques in solving moderately sized instances.

In section 2, we develop the independent set formulation of the graph coloring problem and discuss various advantages of the formulation. In section 3, we discuss techniques for generating columns in this formulation and outline one method for such generation. In section 4, we develop alternative branching rules and discuss their characteristics. In section 5, we describe our implementation details. In section 6, we present the computational results, and in the final section, we give some directions for future exploration.

2 A COLUMN GENERATION MODEL

Let $G = (V, E)$ be an undirected graph on V, the set of vertices, with E being the set of of edges. Let $|V| = n$ and $|E| = m$.

A *coloring* of G is an assignment of labels to each vertex such that the endpoints of any edge have different labels. A *minimum coloring* of G is a coloring with the fewest different labels among all possible colorings.

An *independent set* of G is a set of vertices such that there is no edge in E connecting any pair. Clearly, in any coloring of G, all vertices with the same label comprise an independent set. A *maximal independent set* is an independent set that is not strictly included in any other independent set.

The problem of finding a minimum coloring in a graph can be formulated in many ways. For instance, the problem of determining if K colors suffice can be formulated as follows. Let x_{ik}, $i \in V$, $1 \le k \le K$ be a binary variable that is 1 if vertex i is assigned label k and 0 otherwise. The problem is then to determine if the following system (denoted (VC) for vertex–color) has a feasible solution:

$$\begin{aligned} x_{ik} + x_{jk} &\le 1 \qquad \forall\, (i,j) \in E,\ \forall\, k \\ \sum_k x_{ik} &= 1 \qquad \forall\, i \\ x_{ik} &\in \{0,1\}. \end{aligned}$$

The minimum graph coloring problem can then be solved by doing binary search on K to find the minimum value for which the above system has a feasible solution. This formulation, while correct, is difficult to use in practice. One obvious problem is the size of the formulation. Since K can be as large as n, the formulation can have up to n^2 variables and $nm + n$ constraints. Given the need to enforce integrality, this formulation becomes computationally intractable for all except the smallest of instances. This is especially true because the linear programming relaxation is extremely fractional. To see this, note that x_{ik} $= 1/K$ for every i, k is feasible whenever $K \ge 2$.

A second, less obvious, problem involves the symmetry of the formulation. The variables for each k appear in exactly the same way. This means that it is difficult to enforce integrality in one variable without problems showing up in the other variables because any solution to the linear relaxation has an exponential number (as a function of K) of representations. Therefore, branching on x_{i1} to take on integral values does little good because it results in another representation of the same fractional solution in which x_{i2} takes on the old value of x_{i1} and vice-versa.

We consider a formulation with far fewer constraints that does not exhibit the same symmetry problems as our first formulation. Let S be the set of all maximal independent sets of G. We create a formulation with binary variables x_s for each $s \in S$. $x_s = 1$ implies that independent set s will be given a unique label, while $x_s = 0$ implies that the set does not require a label. The minimum coloring problem is then the following (denoted (IS)):

$$\begin{array}{llcll} \textit{Minimize} & \sum_s x_s & & & \\ \textit{Subject to} & \sum_{\{s: i \in S\}} x_s & \geq & 1 & \forall\, i \in V \\ & x_s & \in & \{0,1\} & \forall\, s \in S. \end{array}$$

This formulation can also be obtained from the first formulation by using a suitable decomposition scheme as explained in [16, 24] in the context of general mixed integer programs. The formulation (IS) has only one constraint for each vertex, but can have a tremendous number of variables. Note that a feasible solution to (IS) may assign multiple labels to a vertex. This can be corrected by using any one of the multiple labels as the label for the vertex. The alternative would be to allow non–maximal sets in S and to require equalities in (IS). In view of the ease of correcting the problem versus the great increase in problem size that would result from increasing S, we choose the formulation given.

This formulation exhibits much less symmetry than (VC): vertices are combined into independent sets and forcing an independent set to 0 means that no color can correspond to that independent set. Furthermore, it is easy to show [24] that the bound provided by the linear relaxation of (IS) will at least be as good as the bound provided by the linear relaxation of (VC).

The fact remains, however, that (IS) can have far more variables than can be reasonably attacked directly. We resolve this difficulty by using only a subset of the variables and generating more variables as needed. This technique, called *column generation*, is well known for linear programs and has recently emerged as a viable technique for some integer programming problems [4, 17]. The need to generate dual variables (which requires something like linear programming) while still enforcing integrality makes column generation procedures nontrivial for integer programs. The procedures need to be suitably developed and their effectiveness is usually dependent on cleverly exploiting the characteristics of the problem.

The following is a brief overview of the column generation technique in terms of (IS). Begin with a subset $\bar{S}$ of independent sets. Solve the linear relaxation (replace the integrality constraints on x_s with nonnegativity) of (IS) restricted to $s \in \bar{S}$. This gives a feasible solution to the linear relaxation of (IS) and a dual value π_i for each constraint in (IS). Now, determine if it would be useful to expand $\bar{S}$. This is done by solving the following *weighted independent set* problem (MWIS):

$$
\begin{array}{llll}
Maximize & \sum_{i \in V} \pi_i z_i & & \\
Subject\ to & z_i + z_j \leq 1 & & \forall\, (i,j) \in E \\
& z_i \in \{0,1\} & & \forall\, i \in V.
\end{array}
$$

If the optimal solution to this problem is more than 1, then the z_i with value 1 correspond to an independent set that should be added to $\bar{S}$. If the optimal value is less than or equal to 1, then there exist no improving independent sets: solving the linear relaxation of (IS) over the current $\bar{S}$ is the same as solving it over S.

This process is repeated until there is no improving independent set. If the resulting solution to the linear relaxation of (IS) has x_s integer for all $s \in \bar{S}$, then that corresponds to an optimal solution to (IS) over S. When some of the x_s are not integer, though, we are faced with the problem of enforcing integrality.

To complete this algorithm, then, we need to do two things. First, since (MWIS) is itself a difficult problem, we must devise techniques to solve it that are sufficiently fast to be able to be used over and over. Second, we must find a way of enforcing integrality if the solution to the linear relaxation of (IS) contains fractional values. Standard techniques of enforcing integrality (cutting planes, fixing variables) make it difficult or impossible to generate improving independent sets. We discuss these two problems in the next two sections.

3 SOLVING THE MAXIMUM WEIGHTED INDEPENDENT SET PROBLEM

The maximum weighted independent set problem is a well studied problem in graph theory and combinatorial optimization (though often under the name of maximum weighted clique, where a clique is an independent set in the complement of a graph). Various solution approaches have been tried, including implicit enumeration [6], integer programming with branch and bound [2, 3], and integer programming with cutting planes [1, 27]. In addition a number of heuristics have been developed [28] and combined with general heuristic methods such as simulated annealing [13]. In this section, we outline a simple recursive algorithm based on the work of [20] and describe a simple greedy heuristic that can be used to reduce the need for the recursive algorithm.

The basic algorithm for finding a maximum weight independent set (MWIS) is based on the following insight: Given a graph G and a vertex $i \in V$, the MWIS in G is either the MWIS in G restricted to $V/\{i\}$ or it is i together with the MWIS in AN(i), where AN(i) is the *anti–neighbor* set of i: the set of all vertices j in V where there is not $(i, j) \in E$. This insight, first examined by [20] for the unweighted case, leads to the following recursion which can be turned into a full program:

$$\text{MWIS}(G \cup \{i\}) = \max(\text{MWIS}(G),\text{MWIS}(\{i\} \cup \text{AN}(i))).$$

While this approach is reasonably effective for not–too–sparse graphs, it can be improved by appropriately ordering the vertices. The following have been shown to be effective in reducing the computational burden of the recursion:

1. Begin with a good MWIS. Note that if G itself is an independent set then adding an independent vertex to it will require the resolution of the current MWIS. This can be avoided by starting with a good MWIS. Then adding a vertex will necessarily involve solving a new problem.

2. Order the remaining vertices in order of degree from lowest to highest. During the final stages of the recursion, it is important to keep the anti–neighbor set small in order to solve the MWIS on as small a graph as possible. Since vertices with high degree have small anti–neighbor sets, those should be saved for the end.

3. Try to determine if a branch of the recursion can possibly return a MWIS better than the incumbent. For instance, if the total weight of the set examined is less than the incumbent, the incumbent is necessarily better, so it is unnecessary to continue the recursion.

4. Use a faster code for smaller problems. It appears that a weighted version of the method of Carraghan and Pardalos [6] is faster for smaller problems, particularly when it is able to terminate when it is clear that no independent set is available that is better than the incumbent. In our tests, which use relatively small graphs, we use a variant of Carraghan and Pardalos for all except the first level of recursions, which echos the results of Khoury and Pardalos in the unweighted case.

In the context of our column generation technique, it is not critical that we get the best (highest weight) maximal independent set: it is sufficient to get any set with weight over 1. This suggests that a heuristic approach for finding an improving column may suffice in many cases. It is only when it is necessary to prove that no set exists with weight over 1 (or when the heuristics fail) that it is necessary to resort to the recursion. There are many heuristics for weighted independent sets. The simplest is the greedy heuristic: begin with (one of) the highest weighted vertices. Add vertices in non-increasing order of their weight making certain that the resulting set remains an independent set.

This heuristic, in addition to being simple, is very fast, and seems to work reasonably well. The resulting independent set can either be added directly to (IS) (if it has value over 1) or can be used as a starting point for the recursion. We will examine the value of this heuristic in the computational results.

4 BRANCHING RULE

A difficult part about using column generation for integer programs is the development of branching rules to ensure integrality. Rules that are appropriate for integer programs where the entire set of columns is explicitly available do not fit in well with restricted integer programs where the columns are generated by implicit techniques. Consider, for instance, the rule of branching on a fractional variable, where the variable is set to 1 in one subproblem and set to 0 in the other. The former subproblem causes no problem for (IS): setting an independent set variable to 1 corresponds to applying a single label to those vertices. Those vertices can then be removed from the graph. The other subproblem is more difficult. Setting a variable to 0 corresponds to not permitting the use of that independent set. How can this information be passed to the subproblem (that generates maximum weight independent sets)? What if the maximum weight independent set is set to 0? How can it be checked if there is another independent set with value more than 1? This seems to involve finding the second, third, and so on highest weight independent sets. This is a much more expensive operation than simply finding the highest weight set (consider how complicated the recursion in the previous section would have to be).

Cutting planes, another technique for forcing integrality, are also difficult to fit into a column generation framework. For instance, consider the graph in Figure 1.

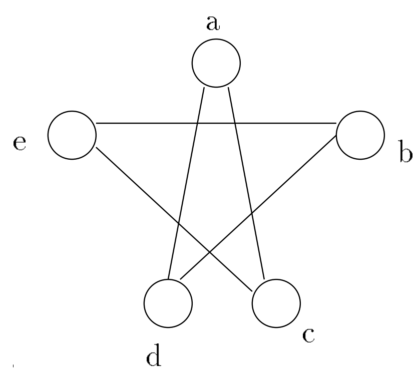

Figure 1: Star Graph

Such a graph has maximal independent sets $\{a, b\}, \{b, c\}, \{c, d\}, \{d, e\}$ and $\{e, a\}$. Applying weights of 1/2 to each set results in a feasible solution to the linear relaxation of (IS), with objective 2.5. We would like to add a constraint that it takes three independent sets to cover these vertices. While it is easy to add such a constraint to (IS), it is not clear as to how this can be accomplished while not complicating the subproblem.

Our approach to the integrality problem is to use a branch-and-bound method without increasing the complexity of the subproblems. We accomplish this by devising special branching rules that ensure that the subproblem to be solved for each branch is itself a graph coloring problem without any additional constraints and can be solved by our column generation methodology. Additionally, the optimal integer solution to (IS) lies in exactly one branch. This implies that the algorithm we use for the maximum weight independent set can be used to generate columns for the problem at every node of the branch-and-bound tree.

Define the following operations on a graph coloring problem: SAME(S) requires that the set S all have the same label, and DIFFER(i,j) requires that nodes i and j have different labels. These operations can be implemented by changing the graph on which the coloring is done. SAME(S) can be enforced by collapsing the set S into a single vertex. A vertex outside of S has an edge to it if and only if it has an edge to any member of S in the original graph. DIFFER(i,j) is even easier: it is only necessary to add an edge between i and j.

Consider a fractional solution to the linear relaxation of (IS). It is easy to see that there exist two sets S_1 and S_2, and vertices i, j, such that $i \in S_1 \cap S_2$, and $j \in S_1 \setminus S_2$, and at least one of x_{s_1} or x_{s_2} is fractional. Create the subproblems:

DIFFER(i,j)

SAME(i,j).

Any feasible coloring must occur in exactly one of the two sets. Furthermore, the independent sets that make up the current fractional solutions are not feasible for the two subproblems. This approach has the advantage of creating only 2 subproblems like traditional branching schemes.

5 IMPLEMENTATION DETAILS

5.1 Column Generation Methodology

The methodology has been implemented on a DEC ALPHA 3000 (Model 300) workstation using CPLEX version 2.1 as the linear programming solver and MINTO version 1.5 [29] as the integer programming solver.

Currently, we generate a feasible initial coloring using a greedy heuristic (essentially applying the greedy MWIS heuristic repeatedly until all nodes are colored). This gives us an initial solution to the coloring problem as well as a number of columns to add to our linear program. We then generate columns to improve the linear program. The following discussion on generation of columns to improve the linear program is valid at each node of the branch-and-bound tree.

5.1.1 Improving the Linear Program

Improving Column As mentioned earlier, any solution to the MWIS with value greater than 1 represents an improving column for the linear program. In our current implementation, we set a target to 1.1 and our MWIS algorithm either returns the first such solution it finds, failing which, it finds the exact solution. We have also experimented with changing this target value to a higher number initially (an approach to find a *good* set of columns as fast as possible) and then decreasing its value later on in the column generation. This has mixed results. While, it tends to optimize the linear program at any node of the branch-and-bound tree faster, the resulting number of nodes to explore rises. We have noticed that the effort required to solve some difficult problems can be substantially reduced by suitably altering this target value.

Ordering the Nodes The order in which the nodes are to be considered can be specified in our MWIS algorithm. We have found that ordering the nodes in order of nonincreasing weights or in order of nonincreasing degree are not as efficient as ordering them by considering both at the same time. In our experiments we order the nodes in nonincreasing values of square root of the degree of the node times the weight of the node.

Column Management Another approach to optimizing the linear program faster is to generate several columns at every iteration rather than a single column [4]. For example, one could use improvement algorithms that take existing columns with reduced cost equal to zero and try to construct columns that might improve the linear program. In our experiments, we generated more candidates for improving independent sets by deleting a node from an improving independent set that was found by the MWIS procedure, and constructing other independent sets from the remaining set. This approach did not improve our results: while the total number of iterations of column generation went down, the time to generate and optimize the linear programs between iterations offset any savings in time because of reduced number of iterations. When the number of columns gets large, it is useful to implement column management schemes that either delete the nonbasic variables with very negative reduced costs or keep them in a temporary pool that is checked every few iterations for any improving columns. Since, the total number of columns generated in our experiments was modest at best, we did not implement such schemes.

Early Branching At any node in the branch-and-bound tree, when there are more improving independent sets, we either get an integer solution (representing an upper bound on the number of colors required) or branch using method 1 if the linear program is fractional. In the latter case, the linear programming solution provides a lower-bound on the best possible coloring available from that node. However, in our implementation, we do not wait for the linear program at a node to be optimized before branching. Rather, we optimize the linear program only at the root node of the branch-and-bound tree to obtain a lower bound on the number of colors required. Then at any other node of the branch-and-bound tree, we stop generating columns as soon as the restricted linear programming relaxation objective value goes below the best possible coloring value determined by rounding up the objective value at the root node. This tended to reduce the number of columns generated and resulted in exploration of fewer nodes in our experiments. The resulting decrease in overall computation time by not solving the linear programming relaxation to optimality before branching has also been experienced on other combinatorial problems [4, 31].

5.1.2 Branching Methodology

In our implementation, we use a depth-first-search(DFS) strategy in choosing the node to evaluate. We have also experimented with the idea of choosing the node with the best-bound and of switching from a DFS strategy to the best-bound strategy after updating the upper bound (from the one provided by the initial solution). The DFS strategy seems to work best in our implementation, both from a point of view of number of nodes explored and in cpu-time overall.

For implementing our branching, we first determine the most fractional column (s_1). Then we find the first row i covered by this column and determine another column(s_2) that covers row i. Then we find row j such that only one of the columns s_1 or s_2 cover row j. We have also experimented with choosing the first fractional column as s_1 instead of finding the most fractional one. This tends to increase the overall effort.

5.2 DSATUR

In addition to our column generation approach to graph coloring, we have implemented an exact graph coloring algorithm based on the DSATUR algorithm first developed by Brélaz [5]. We will use this algorithm as a comparison code to determine the effectiveness of column generation.

DSATUR works by dividing a graph coloring instance into a series of subproblems. A subproblem in DSATUR is a partial coloration of the graph. At each step of the algorithm, there is an upper bound UB on the number of colors required for the graph. If a subproblem uses k colors, and k is at least UB, clearly the subproblem can be fathomed. Alternatively, if every node in the graph is colored, and $k <$ UB, then a better coloring has been found and UB can be set to k.

If the graph is not completely colored, and the number of colors used is less than UB, new subproblems are created. Some uncolored node i is chosen for branching. For each feasible color for i (out of the k used in the subproblem), a new subproblem is created assigning i that color. In addition, a subproblem is created with i receiving color $k + 1$.

The algorithm terminates when there are no subproblems left. At this point, UB give the coloring number of the graph.

The choice of branch node i can have a large effect on the algorithm. In a heuristic, Brèlez [5] suggests choosing the node adjacent to the largest number of differently colored nodes. This has the effect of reducing the number of subproblems created at each branch. Ties can be broken by choosing a node with highest degree in the uncolored subgraph. Korman [19] recommended using this within the optimization routine, and Kubale and Jackowski [21] confirm that this is an effective choice in their experiment.

A further modification is suggested by Sewell [30]. If the first k nodes colored form a clique, then it is clear that they will never be recolored. This suggests that it would be useful to find a maximum clique in the graph and color those nodes first. This approach is a large improvement when the clique value and the coloring number of a graph are close, and seems a good idea for many instances.

Our version of DSATUR attempts to find the maximum clique in the graph using the unweighted versions of the clique finding algorithms previously presented. Since it is not critical to prove the optimality of the clique, we terminate the clique search once 10,000 clique subproblems have been generated. In the vast majority of the instances we solved, the optimal clique was found in far fewer subproblems.

The rest of the nodes are dynamically ordered in terms of the number of adjacent colors and subproblems are created as in the basic DSATUR algorithm. Subproblems are solved in a depth–first search manner.

6 COMPUTATIONAL RESULTS

6.1 Instance Description

In our computational experiments, we use instances drawn from a large number of sources. Our goal is to determine the robustness of the approaches. For some of these graphs, the coloring problem has no real interpretation. We use these graphs as examples of structured graphs, rather than just experimenting on random graphs. Here we briefly describe the instance classes.

Random Graphs Random graphs are ubiquitous in computational experiments for graph coloring. $G(n, p)$ is formed by generating a graph on n vertices, where each edge occurs independently with probability p.

Register Graphs One standard application of graph coloring is *register allocation*. In this problem, a compiler is attempting to assign variables to registers. Two variables can be assigned to the same register if they are not both required at the same time in a code fragment. Condon [9] has developed a program that takes code fragments and generates the corresponding graph coloring problem.

Geometric Graphs A different type of random graph is created by generating $2n$ random numbers in the range (0,1) and treating pairs number as coordinates in the Euclidean unit square. A node is placed at each coordinate, and two nodes are connected by an edge if and only if the Euclidean distance between them is less than some cutoff value. Alternatively, a *reverse geometric graph* results when edges correspond to nodes that are more than some distance apart.

Book Graphs Given a work of literature, a graph is created where each node represents a character. Two nodes are connected by an edge if the corresponding characters encounter each other in the book. Knuth [18] creates the graphs for five classic works: Tolstoy's *Anna Karenina* (`anna`), Dicken's *David Copperfield* (`david`), Homer's *Iliad* (`homer`), Twain's *Huckleberry Finn* (*huck*), and Hugo's *Les Misérables* (`jean`).

Game Graphs A graph representing the games played in a college football season can be represented by a graph where the nodes represent each college team. Two teams are connected by an edge if they played each other during the season. Knuth [18] gives the graph for the 1990 college football season.

Miles Graphs These graphs are similar to geometric graphs in that nodes are placed in space with two nodes connected if they are close enough. These graphs, however, are not random. The nodes represent a set of United States cities and the distance between them is given by by road mileage from 1947. These graphs are also due to Kuth [18].

Queen Graphs Given an n by n chessboard, a queen graph is a graph on n^2 nodes, each corresponding to a square of the board. Two nodes are connected by an edge if the corresponding squares are in the same row, column, or diagonal. Unlike some of the other graphs, the coloring problem on this graph has a natural interpretation: Given such a chessboard, is it possible to place n sets of n queens on the board so that no two queens of the same set are in the same row, column, or diagonal? The answer is yes if and only if the graph has coloring number n. Gardner [12] states without proof that this is the case if and only if n is not divisible by either 2 or 3. In all cases, the maximum clique in the graph is no more than n, and the coloring value is no less than n.

Mycielski Graphs Given a graph G with vertex set $\{v_1, v_2, \ldots, v_n\}$, we can get the Mycielski transformation [26] $\mu(G)$ of G by creating a graph with vertex set

$$\{x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_n, z\}$$

and edges $x_i x_j$ if and only if $v_i v_j \in E(G)$, $x_i y_j$ if and only if $v_i v_j \in E(G)$ and $y_i z$ for all i. Larson, Propp, and Ullman [23] show that as long as G has at least one edge, the size of the largest clique in the graph is not affected by this transformation. They also show that the coloring number goes up by one. Finally, they show that if the solution to the linear relaxation of the coloring problem (IS) for G is k, then the corresponding solution for $\mu(G)$ is $k + 1/k$.

If we let G_1 be the graph with two nodes and a single edge, and recursively define $G_{i+1} = \mu(G_i)$, then each G_i is triangle–free (the maximum clique is 2), has coloring number $i+1$, and has linear relaxation of IS value in between those two values. As such, these graphs seem difficult to solve since neither the clique nor the linear relaxation of IS can provide tight bounds.

6.2 Summary of Results

The results on various coloring instances are summarized in Tables 1– 4. The following results are presented:

- Size: Lists the number of nodes and the number of edges in the graph.
- GR: The value of the feasible coloring returned by the greedy heuristic. This heuristic generates the starting solution for LPCOLOR, the column generation based procedure.
- Bounds: These provide the lower bounds on the optimal coloring value for the graph. CL represents the bound provided by the size of the maximum clique in the graph. When the maximum clique used for providing the CL bound is not confirmed to be maximum, an asterisk is placed to indicate that. The LP lower bound is the value of

the optimal linear programming objective value at the root node of the branch-and-bound tree in our methodology and LP_{up} is the lower bound provided by rounding up the LP bound.

- Optimal: Lists the optimal coloring value of the graph.
- DSATUR: Lists the cpu seconds and the number of explored in the search for the optimal solution in our implementation based on DSATUR.
- LPCOLOR: Lists the cpu seconds and the number of evaluated nodes in the branch-and-bound tree for the column generation methodology.

The time limit for these experiments was one hour of cpu time. A missing entry in the tables indicates that the problem could not be solved within this time limitation. In some cases, we present averages over 5 instances. When the average is over less than 5 instances (because the others could not be solved within the time limits), we indicate the number of instances solved in brackets. We omit any results if the number of solved instances is less than 3.

The first type of coloring instances solved are those on random graphs, $G(n, p)$. We generated graphs with 30, 50 and 70 nodes with edge probabilities .1, .3, .5, .7, and .9. The results over average of 5 instances are shown in Table 1.

Graph	Size		GR	Bounds			Optimal	DSATUR		LPCOLOR	
	nodes	*edges*		CL	LP	LP_{up}		time	nodes	time	nodes
G(30,.1)	29.8	38.2	3.2	3.0	3.0	3.0	3.0	0.0	27.8	0.0	1.0
G(30,.3)	30.0	131.6	5.8	4.6	4.7	5.0	5.0	0.0	38.2	0.4	4.8
G(30,.5)	30.0	212.8	7.8	6.0	6.6	7.0	7.0	0.0	41.6	0.2	2.4
G(30,.7)	30.0	301.0	10.8	8.8	9.3	9.8	9.8	0.0	64.2	0.0	6.0
G(30,.9)	30.0	374.0	15.0	14.4	14.7	14.8	14.8	0.0	43.4	0.0	1.0
G(50,.1)	50.0	120.8	4.0	3.0	3.3	4.0	4.0	0.0	57.6	3.2	1.0
G(50,.3)	50.0	364.8	7.4	5.2	5.7	6.2	6.4	0.0	942.6	30.2	33.2
G(50,.5)	50.0	600.2	11.4	7.6	8.6	9.0	9.4	0.8	46991.0	13.8	50.8
G(50,.7)	50.0	814.6	16.2	12.4	13.5	14.2	14.2	1.4	3594.0	1.4	8.8
G(50,.9)	50.0	1096.8	24.0	20.6*	21.9	22.2	22.2	2.6	239.6	0.0	1.6
G(70,.1)	70.0	235.8	5.0	3.2	3.6	4.0	4.0	0.0	95.0	73.0	15.0
G(70,.3)	70.0	736.8	9.8	5.6	6.8	7.6	7.8	7.6	102753.6	585.0	167.6
G(70,.5)	70.0	1218.6	14.6	8.4	10.7	11.4	11.6	535.0	3692013.0	35.4	48.4
G(70,.7)	70.0	1678.6	19.8	12.6*	16.2	16.6	17.0	408.8	4253916.0	14.2	38.0
G(70,.9)	70.0	2173.6	31.4	24.0*	28.1	28.6	28.6	13.4	89594.2	1.0	8.0

Table 1: Results on Random Graphs (Average of 5 instances)

There are a few things to note for the LPCOLOR results:

1. The lower bound provided by rounding up objective value of linear programming solution at the root node of the branch-and-bound tree is usually within 1 of the optimal value (this was true for instances we have solved). This is evidence of the strength of the formulation.

2. Instances with densities in the range of .1–.5 are most difficult for this method in terms of computation time and the number of nodes evaluated. Also, the GR bound (and hence the starting solution) is not a good bound as the problems get more difficult.

3. The number of evaluated nodes is very small, suggesting the strength of this branching scheme.

4. While 70 nodes does not seem so large, some instances are taking a fair amount of time already. However, the average timings above can be deceptive. For instance, in many cases, it is usually only one or two problems that raise the average cpu time and the average number of evaluated nodes. So more often than not, the problems are solved in a reasonable amount of time.

In comparison, the performance of DSATUR based algorithm indicates that the densities of 0.5 and 0.7 are most difficult both in computational time and the number of explored nodes. Typically, both methods find the optimal solution quickly and spend a long time in verifying optimality.

The next set of instances come from register allocation. These graphs seem to be quite easy for many algorithms: there is an easily found large clique that defines the coloring number. The results are contained in Table 2. In all cases, the simple heuristic was able to find the correct coloring and column generation quickly proved optimality. Similarly, the DSATUR based algorithm had no difficulties with these problems.

Graph	Size		GR	Bounds			Optimal	DSATUR		LPCOLOR	
	nodes	*edges*		CL	LP	LP_{up}		time	nodes	time	nodes
mulsol.i.1	197	3925	49	49	49	49	49	0	149	2	1
mulsol.i.2	188	3885	31	31	31	31	31	0	158	1	1
mulsol.i.3	184	3916	31	31	31	31	31	0	154	3	1
mulsol.i.4	185	3946	31	31	31	31	31	0	155	1	1
mulsol.i.5	186	3973	31	31	31	31	31	0	156	1	1
zeroin.i.1	211	4100	49	49	49	49	49	0	163	2	1
zeroin.i.2	211	3541	30	30	30	30	30	0	182	2	1
zeroin.i.3	206	3540	30	30	30	30	30	0	177	2	1
inithx.i.1	864	18707	54	54	54	54	54	1	432	31	1
inithx.i.2	645	13979	31	31	31	31	31	1	422	19	1
inithx.i.3	621	13969	31	31	31	31	31	1	396	14	1
fpsol2.i.1	496	11654	65	65	65	65	65	0	811	10	1
fpsol2.i.2	451	8691	30	30	30	30	30	2	615	9	1
fpsol2.i.3	425	8691	30	30	30	30	30	2	591	16	1

Table 2: Register Allocation Graphs

Our third set of graphs are geometric graphs. The `gn.d` graphs are geometric graphs with n nodes and cutoff d; the `grn.d` graphs are reverse geometric graphs with n nodes and cutoff d. The results are shown in Table 3. The results on these graphs are quite striking in

Graph	Size		GR	Bounds			Optimal	DSATUR		LPCOLOR	
	nodes	*edges*		CL	LP	LP_{up}		time	nodes	time	nodes
g100.1	99.8	153.2	5.2	5.2	5.2	5.2	5.2	0.0	95.4	0.2	1.0
g150.1	150.0	318.8	6.2	6.2	6.2	6.2	6.2	0.0	144.8	1.0	1.0
g200.1	200.0	588.2	7.8	7.8	7.8	7.8	7.8	0.0	241.6	4.6	1.0
g250.1	250.0	884.0	9.0	8.8	9.0	9.0	9.0	0.0	242.2	4.0	1.0
g100.5	100.0	2275.4	30.4	29.2	29.2	29.2	29.2	3.2	29225.8	7.8	11.6
g150.5	150.0	4064.8	43.2	38.4	38.4	38.8	38.8	-	-	94.8	32.6
g200.5	200.0	9335.8	60.8	54.6	54.6	54.6	54.6	156.2(4)	559811.1 (4)	110.0	29.0
g250.5	250.0	15041.4	71.0	63.8	64.4	64.4	64.4	-	-	475.8	41.4
g100.9	100.0	4475.0	67.8	67.4	67.4	67.4	67.4	0.6	449.8	0.1	1.0
g150.9	150.0	10137.0	100.0	100.0	100.0	100.0	100.0	4.4	1405.4	1.0	1.0
g200.9	200.0	18142.8	134.8	133.2	133.2	133.2	133.2	60.5 (4)	98179.2 (4)	2.4	1.0
g250.9	250.0	28841.2	171.4	170.4*	170.6	170.6	170.6	-	-	4.2	1.0
gr100.1	100.0	4797.0	43.6	43.2*	43.6	43.6	43.6	12.8	57.8	0.0	1.0
gr150.1	150.0	10856.2	55.6	53.8*	54.8	54.8	54.8	21.8	110.8	1.0	1.0
gr200.1	200.0	19311.8	62.4	58.4*	61.0	61.0	61.2	39.8	26463.6	2.0	2.0
gr250.1	250.0	30215.4	66.4	62.0*	64.6	64.6	64.6	50.4	21825.4	5.2	1.0
gr100.5	100.0	2654.0	6.8	5.6*	6.0	6.0	6.0	2.0	126.2	0.2	1.0
gr150.5	150.0	5946.0	6.8	5.2*	6.4	6.8	6.8	3.0	332.2	1.2	1.0
gr200.5	200.0	10551.2	7.0	5.2*	6.4	6.8	6.8	3.8	492.0	2.6	1.0
gr250.5	250.0	15934.8	7.2	5.0*	6.7	7.0	7.0	4.8	561.8	6.2	1.4
gr100.9	100.0	475.0	3.4	3.2	3.2	3.2	3.2	0.0	97.8	0.0	1.0
gr150.9	149.6	1038.0	3.8	3.2	3.3	3.4	3.4	0.2	147.4	0.8	1.0
gr200.9	199.6	1757.2	4.0	3.2	3.4	3.6	3.6	1.0	197.6	2.2	1.0
gr250.9	249.8	2411.2	4.0	3.4	3.6	3.8	3.8	1.0	247.4	3.4	1.0

Table 3: Geometric Graphs (Average of Five Instances)

that LPCOLOR provided much more robust results, solving many instances that DSATUR was unable to handle.

Finally, Table 4 presents the results for the other graph classes we have experimented with. All of these graphs were relatively easy to solve except for the 8×8 (and larger) queen problems. DSATUR was unable to solve a couple of these queen problems. Mycielski graphs seem to be very difficult for either method. DSATUR could solve mycielski–5 graph that LPCOLOR could not even though the starting solution for LPCOLOR was an optimal one. This is clearly because of the large gap between the LP and Optimal values. These gaps continue to be large even deeper down in the branch-and-bound tree making it difficult to prove optimality for LPCOLOR. Larger queen and mycielski graphs could not be solved by either program.

Graph	Size		GR	Bounds			Optimal	DSATUR		LPCOLOR	
	nodes	*edges*		CL	LP	LP_{up}		time	nodes	time	nodes
anna	138	493	11	11	11	11	11	0	128	0	1
david	87	406	11	11	11	11	11	0	77	0	1
homer	561	1629	13	13	13	13	13	1	549	24	1
huck	74	301	11	11	11	11	11	0	64	0	1
jean	80	254	10	10	10	10	10	0	71	0	1
games120	120	638	9	9	9	9	9	0	112	1	1
miles250	128	387	8	8	8	8	8	0	121	2	1
miles500	128	1170	20	20	20	20	20	0	109	1	1
miles750	128	2113	32	31	31	31	31	0	98	1	3
miles1000	128	3216	42	42	42	42	42	0	87	0	1
miles1500	128	5198	73	73	73	73	73	0	56	1	1
queen5.5	25	160	7	5	5	5	5	0	21	0	1
queen6.6	36	290	9	6	7	7	7	0	1865	1	4
queen7.7	49	476	11	7	7	7	7	0	6849	4	1
queen8.8	64	728	12	8	8.4	9	9	-	-	19	18
queen9.9	81	2112	13	9	9	9	10	-	-	515	77
queen8.12	96	1368	14	12	12	12	12	0	164	79	42
myciel2	5	5	3	2	2.5	3	3	0	4	0	1
myciel3	11	20	4	2	2.9	3	4	0	27	0	9
myciel4	23	71	5	2	3.2	4	5	0	848	9	303
myciel5	47	236	6	2	3.5	4	6	18	378311	-	-

Table 4: Other Miscellaneous Graphs

7 CONCLUSIONS

We have developed an algorithm for coloring based on the independent set formulation. The success of this algorithm hinges on fast algorithms for finding maximum weighted independent sets. It appears that for the instances solved here, the independent set formulation gives a very good lower bound on the number of colors required. In fact, this observation leads to a good heuristic procedure for graphs that are too large or too difficult to solve exactly. In the heuristic procedure, we do not generate any columns after optimizing the root node linear program. Instead, we find the best integer solution from among the columns accumulated at the root node of the branch-and-bound tree. We refer to this restricted program at the root node as the restricted integer program (RIP). This type of heuristic has been successfully used for clustering problems in [17].

The strength of this branching method is another important observation. Very few nodes are explored in the branch-and-bound tree. To study this strength, we solved the RIP (without generating additional columns at subsequent nodes of the branch-and-bound tree) by two methods: the Standard method refers to the branching method of fixing a fractional variable to 0 or a 1. The New Branching refers to the branching rule developed in this paper. Note that the values obtained by the new branching are not necessarily the same as that in the standard branching because the new branching does not guarantee finding the best possible solution for RIP without the ability to generate additional columns at each node of the resulting tree. The results are shown in Table 5, where we give the results for the G(70,.) graphs including the value of the solutions, the number of nodes evaluated and the cpu time it takes to solve the linear program at the root node as an integer program. Once again, these are averaged over five instances. The speed with which we solve problems to optimality seems to be due to the strength of the branching rule, since the time per node is roughly comparable for the two approaches. It is interesting to note that it takes roughly the same amount of time to solve RIP to optimality with the standard branching rule as it takes LPCOLOR to solve the entire problem to optimality with the new branching rule (Table 1).

Density	Standard			New Branching		
	Value	Nodes	sec	Value	Nodes	sec
.1	4.8	214.8	117.2	5.2	71.6	46.2
.3	8.8	3333.0	500.6	8.8	525.0	119.6
.5	12.6	590.0	72.4	12.6	296.0	45.8
.7	17.4	79.6	8.4	17.4	51.0	7.8
.9	28.6	3.0	0.8	28.6	7.2	1.0

Table 5: Solving the restricted integer program

Improvements to the column generation implementation would be more robust heuristics for the maximum weighted clique problem and more intelligent rules for determining when to terminate the recursion for exactly finding the MWIS. Also, implementing other branching rules and studying their efficiency would be interesting. For example, the following branching rule could be used:

Consider a fractional solution to the linear relaxation of (IS). As before, there exist two sets S_1 and S_2, and vertices i, j, such that $i \in S_1 \cap S_2$, and $j \in S_1 \setminus S_2$, and at least one of x_{s_1} or x_{s_2} is fractional. Earlier we created the subproblems SAME(i,j), and DIFFER(i,j). To implement DIFFER(i,j), consider the following: If i is an isolated vertex, there is no reason to look at DIFFER(i,j): the solution can be no better than SAME(i,j). Generalizing this, if i and j are to differ and have the solution not be equivalent to SAME(i,j), one of the neighbors of i must receive the same color as j. So, we can create the subproblems

SAME(i_1, j)
SAME(i_2, j)
...
SAME(i_k, j)

where $\{i_1, i_2, ..., i_k\}$ are neighbors of i. This choice of branching ensures that each node of the branch-and-bound tree is of similar difficulty and that the depth of the resulting tree is no more than the number of nodes in the graph. This approach however can have the optimal solution occuring in more than one branches. To prevent that, the DIFFER(i, j) could be implemented as follows:

SAME(i_1, j)
DIFFER(i_1, j), SAME(i_2, j)
DIFFER(i_1, j), DIFFER(i_2, j), SAME(i_3, j)
...
DIFFER(i_1, j), DIFFER(i_2, j), ...DIFFER(i_{k-1}, j), SAME(i_k, j).

When it is not crucial to generate a provably optimal coloring, or when the time available is insufficient for our procedure to complete, simple modifications can be used to generate good solutions rather quickly. For example, a target value for the coloring can be prespecified and column generation at the nodes of the branch-and-bound tree can be stopped as soon as the objective value of the linear program falls below this target value. This can be useful especially when a good starting solution is already available and one is seeking a better solution. The branching algorithm can then be terminated as soon as an improving solution is found. In table 6, we give results for the G(.,.5) graphs. For these experiments, our algorithm is modified to stop when either a solution that is guaranteed to be within 1 of the best possible solution as determined by rounding up the optimal linear programming objective value is found or when 100 nodes of the branch-and-bound tree have been explored. Results are averaged over five instances in each case. The Initial value is the upper bound corresponding to the starting solution found by the greedy procedure, the LP bound is obtained by rounding up the linear programming objective value at the root node and the best solution found with the termination criterion as stated is given by Best Found.

Finally, it would be interesting to study the algorithm's performance on the various classes of instances in more detail. Random instances seem to be difficult (at least for some densities) with graphs as small as 70 nodes. Register allocation graphs seem to be very easy. Geometric graphs seem markedly different whether one places edges for long distances or short ones. It would be useful to understand where these differences come from.

Size	Value			Time
Size	Initial	LP Bound	Best Found	sec
70	16.8	11.4	12.4	18.8
80	18.2	12.0	13.2	48.2
90	20.6	13.0	14.2	99.4
100	21.6	13.8	15.6	170.2
110	22.6	14.6	16.8	248.6
120	24.0	15.8	17.6	308.4

Table 6: Results when algorithm is terminated early

References

[1] Egon Balas and H. Samuelsson. A node covering algorithm. *Naval Research Logistics Quarterly*, 24(2):213–233, 1977.

[2] Egon Balas and Jue Xue. Minimum weighted coloring of triangulated graphs, with application to maximum weight vertex packing and clique finding in arbitrary graphs. *SIAM Journal on Computing*, 20(2):209–221, 1991.

[3] Egon Balas and Chang Sung Yu. Finding a maximum clique in an arbitrary graph. *SIAM Journal on Computing*, 15(4):1054–1068, 1986.

[4] Cynthia Barnhart, Ellis L. Johnson, George L. Nemhauser, Martin W. P. Savelsbergh, and Pamela H. Vance. Branch-and-Price: Column Generation for Huge Integer Programs. *School of Industrial & Systems Engineering, Georgia Institute of Technology, Atlanta, GA 30332-0205*, 1994.

[5] D. Brélaz. New Methods to color the vertices of a graph. *Communications of the ACM*, 22:251–256, 1979.

[6] R. Carraghan and P. M. Pardalos. An exact algorithm for the maximum clique problem. *Operations Research Letters*, 9:375–382, 1990.

[7] Fred C. Chow and John L. Hennessy. Register allocation by priority–based coloring. In *Proceedings of the ACM SIGPLAN 84 Symposium on Compiler Construction*, pages 222–232, New York, NY, 1984. ACM.

[8] Fred C. Chow and John L. Hennessy. The priority–based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, 1990.

[9] Anne Condon. *DIMACS Challenge* 1994.

[10] D. De Werra. An introduction to timetabling. *European Journal of Operations Research*, 19:151–162, 1985.

[11] Andreas Gamst. Some lower bounds for a class of frequency assignment problems. *IEEE Transactions of Vehicular Echnology*, 35(1):8–14, 1986.

[12] Martin Gardner. *The Unexpected Hanging and Other Mathematical Diversions*. Simon and Schuster, New York, 1969.

[13] Mark Jerrum. Large cliques elude the metropolis process. *Random Structures and Algorithms*, 3(4):347–360, 1992.

[14] David S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.

[15] David S. Johnson. Worst–case behavior of graph coloring algorithms. In *Proceedings 5th Southeastern Conference on Combinatorics, Graph Theory, and Computing*, pages 513–527, Winnipeg, Canada, 1974. Utilitas Mathematica Publishing.

[16] Ellis L. Johnson. Modeling and strong linear programs for mixed integer programming. S.W. Wallace (ed.). *Algorithms and Model Formulations in Mathematical Programming*, NATO ASI Series 51, 1989.

[17] Ellis L. Johnson, Anuj Mehrotra, and George L. Nemhauser. Min-cut clustering. *Math Programming*, 62:133-151, 1993.

[18] Donald E. Knuth. *The Stanford GraphBase*. ACM Press, Addison Wesley, New York, 1993.

[19] S.M. Korman. The graph–coloring problem. In N. Christophides, P. Toth, and C. Sandi, editors, *Combinatorial Optimization*, 211–235, Wiley, New York, 1979.

[20] Bassam N. Khoury and Panos M. Pardalos. An algorithm for finding the maximum clique on an arbitrary graph. *DIMACS Challenge*, 1993.

[21] M. Kubale and B. Jackowski. A generalized implicit enumeration algorithm for graph coloring. *Communications of the ACM*, 28:412–418, 1985.

[22] M. Kubale and E. Kusz. Computational experience with implicit enumeration algorithms for graph coloring. In M. Nagl and J. Perl, editors, *Proceedings of the WG'83 International Workshop on Graphtheoretic Concepts in Computer Science*, pages 167–176, Linz, 1983. Trauner Verlag.

[23] Michael Larsen, James Propp, and Daniel Ullman. The fractional chromatic number of a graph and a construction of Mycielski. preprint, 1994.

[24] Anuj Mehrotra. Constrained Graph Partitioning: Decomposition, Polyhedral Structure and Algorithms. *Ph.D. Thesis, Georgia Institute of Technology, Atlanta, GA*, 1992.

[25] Craig Morgenstern and Harry Shapiro. Coloration neighborhood structures for general graph coloring. In *First Annual ACM–SIAM Symposium on Discrete Algorithms*, 1990.

[26] J. Mycielski. Sur le coloriage des graphes. *Colloquim Mathematiques*, 3:161–162, 1955.

[27] George L. Nemhauser and Les E. Trotter. Vertex packings: Structural properties and algorithms. *Mathematical Programming*, 8:232–248, 1975.

[28] B. Pittel. On the probable behaviour of some algorithms for finding the stability number of a graph. *Mathematical Proceedings of the Cambridge Philosophical Society*, 92:511–526, 1982.

[29] Martin W.P. Savelsbergh and George L. Nemhauser. Functional description of MINTO, a Mixed INTeger Optimizer. *School of Industrial & Systems Engineering, Georgia Institute of Technology, Atlanta, GA 30332-0205*, 1993.

[30] Edward C. Sewell. An Improved Algorithm for Exact Graph Coloring. To appear, *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, 1995.

[31] P.H. Vance, C. Barnhart, E.L. Johnson, and G.L. Nemhauser. Solving Binary Cutting Stock Problems by Column Generation and Branch-and-Bound. *Computational Optimization and Applications*, to appear, 1994.