

# The Cunningham-Geelen Method in Practice: Branch-Decompositions and Integer Programming

S. Margulies

Department of Mathematics, Pennsylvania State University, State College, Pennsylvania 16802, [margulies@math.psu.edu](mailto:margulies@math.psu.edu)

J. Ma

Department of Management Science and Engineering, Stanford University,  
Stanford, California 94305, [jm11.rice@gmail.com](mailto:jm11.rice@gmail.com)

I. V. Hicks

Department of Computational and Applied Math, Rice University, Houston, Texas 77251,  
[ivhicks@rice.edu](mailto:ivhicks@rice.edu)

In 2007, W. H. Cunningham and J. Geelen describe an algorithm for solving  $\max\{c^T x : Ax = b, x \geq 0, x \in \mathbb{Z}^n\}$ , where  $A \in \mathbb{Z}^{m \times n}$ ,  $b \in \mathbb{Z}^m$ , and  $c \in \mathbb{Z}^n$ , which utilizes a branch-decomposition of the matrix  $A$  and techniques from dynamic programming. In this paper, we report on the first implementation of the CG algorithm and compare our results with the commercial integer programming software GUROBI. Using branch-decomposition trees produced by heuristics and optimal trees produced by algorithms developed in our previous studies, we test both a memory-intensive and low-memory version of the CG algorithm on problem instances such as graph 3-coloring, set partition, market split, and knapsack. We isolate a class of set partition instances where the CG algorithm runs twice as fast as GUROBI, and demonstrate that certain infeasible market split and knapsack instances with width  $\leq 6$  range from running twice as fast as GUROBI, to running in a matter of minutes versus a matter of hours.

**Key words:** optimization; integer programming; branch-decompositions

**History:** Accepted by Karen Aardal, Area Editor for Design and Analysis of Algorithms; received April 2011; revised January 2012; accepted June 2012. Published online in *Articles in Advance*.

## 1. Introduction

The burgeoning area of branch-decomposition-based algorithms has expanded to include problems as diverse as ring-routing (Cook and Seymour 1994), travelling salesman (Cook and Seymour 2003) and general minor containment (Hicks 2004). In addition to being an algorithmic tool, branch-decompositions have been instrumental in proving such theoretical questions as the famous Graph Minors Theorem (proved in a series of 20 papers spanning 1983 to 2004), and also in identifying classes of problems that are solvable in polynomial time. For example, in Courcelle (1990) and Arnborg et al. (1991), the authors showed that several NP-hard problems (such as Hamiltonian cycle and covering by triangles) are solvable in polynomial time in the special case where the input graph has bounded tree-width or branch-width.

During its 50-year history, the approaches to solving integer programs have been as dissimilar as the industry applications modeled by the integer programs themselves. Branch-and-bound techniques, interior point methods and cutting plane algorithms

are only a few of the strategies developed to answer the question  $\max\{c^T x : Ax = b, x \geq 0, x \in \mathbb{Z}^n\}$ . However, a particularly interesting aspect of the Cunningham-Geelen (CG; Cunningham and Geelen 2007) algorithm is that it provides a first link between integer programming and the long list of research areas accelerated by branch-decompositions. How significant will this link between branch-decompositions and integer programming prove to be? Additionally, as with any new algorithm, there are a series of logical questions: How well does the CG algorithm work in practice? Are there classes of problems upon which it is efficient? How does it compare to commercial software? These are the questions explored in this paper.

We begin in §2 by recalling the relevant background and definitions (such as branch-decompositions, branch-width,  $T$ -branched sets, etc.). In §3, we describe the CG algorithm in detail, including a discussion of internal data-structures and a runtime analysis, and highlight the relevance of a particular vector space intersection to the workings of the algorithm. In §4, we describe three different methods

for calculating this particular intersection, and include experimental results for a comparison of the three methods. In §5, we display the computational results for the CG algorithm: we test on graph 3-coloring, set partition, market split, and knapsack instances, and also include a cross-comparison with the commercial integer programming software GUROBI (Bixby et al. 2010). We conclude in §5.4 with a brief discussion on the impact of different branch-decomposition trees on the runtime of the CG algorithm, and comment on future work in the conclusion.

## 2. Background and Definitions

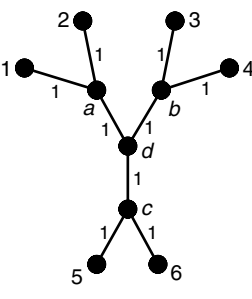
Given an  $m \times n$  matrix  $A$ , let  $E = \{1, \dots, n\}$  and  $X \subseteq E$ . A *branch-decomposition* of  $A$  is a pair  $(T, \nu)$ , where  $T$  is a cubic tree (all interior nodes have degree three), and  $\nu$  is a map from the columns of  $A$  to the leaves of  $T$ . The edges of  $T$  are weighted via a *connectivity function*  $\lambda_A$ . Specifically, for any edge  $e \in E(T)$ ,  $T - e$  disconnects the tree into two connected components. Since the leaves of the tree correspond to column indices, disconnecting the tree  $T$  is equivalent to partitioning the matrix into two sets of columns,  $X$  and  $E - X$ . Then, letting  $A|X$  denote the submatrix of  $A$  containing only the columns of  $X$ , we define the connectivity function

$$\lambda_A(X) = \text{rank}(A|X) + \text{rank}(A|(E - X)) - \text{rank}(A) + 1.$$

We note that the connectivity function is *symmetric* (since  $\lambda_A(X) = \lambda_A(E - X)$ ), and *submodular* (since  $\lambda_A(X_1) + \lambda_A(X_2) \geq \lambda_A(X_1 \cap X_2) + \lambda_A(X_1 \cup X_2)$  for all  $X_1, X_2 \subseteq E$ ). Finally, we define the *width* of a branch-decomposition  $(T, \nu)$  as the maximum over all the edge weights in  $T$ , and we define the *branch-width* of  $A$  as the minimum-width branch-decomposition over all possible branch-decompositions of  $A$ . In other words, let  $\text{BD}(A)$  denote the set of all possible branch-decompositions  $(T, \nu)$  of  $A$ , and thus, the

$$\text{branch-width of } A = \min_{(T, \nu) \in \text{BD}(A)} \left\{ \max_{e \in E(T)} \{\text{weight}(e)\} \right\}.$$

**EXAMPLE 1.** Consider the following matrix  $A$  and branch-decomposition  $(T, \nu)$ :

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 6 & 7 & 7 & 1 & 7 & 5 \\ 3 & 7 & 7 & 4 & 2 & 1 \\ 9 & 1 & 2 & 9 & 5 & 1 \\ 0 & 4 & 6 & 3 & 6 & 2 \\ 4 & 4 & 6 & 5 & 8 & 8 \\ 3 & 6 & 1 & 2 & 9 & 2 \end{bmatrix}$$


The weight of edge  $(a, d)$  is one, since  $T - (a, d)$  disconnects  $T$  into two components (labeled with columns  $X = \{1, 2\}$  and  $E - X = \{3, 4, 5, 6\}$ , respectively). Therefore, we see

$$\begin{aligned} \text{weight of edge } (a, d) &= \lambda_A(\{1, 2\}) \\ &= \text{rank}(A| \{1, 2\}) + \text{rank}(A| \{3, 4, 5, 6\}) - \text{rank}(A) + 1 \\ &= 2 + 4 - 6 + 1 = 1. \end{aligned}$$

Furthermore, the maximum over all the edge weights of  $T$  is one, which is the smallest possible width over all branch-decompositions of  $A$ . Thus, the branch-width of  $A$  is one, and the tree  $T$  is an optimal branch-decomposition of  $A$ .

## 3. Overview of the Cunningham-Geelen (CG) Algorithm

Given a nonnegative matrix  $A \in \mathbb{Z}_{\geq 0}^{m \times n}$ , the CG algorithm solves the integer program

$$\max\{c^T x : Ax = b, x \geq 0, x \in \mathbb{Z}^n\}.$$

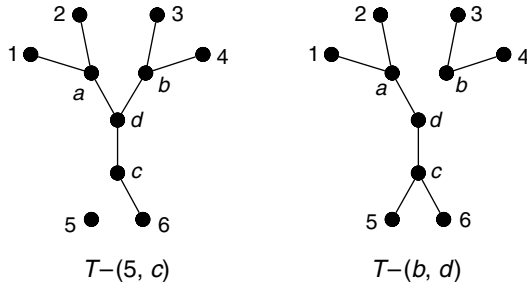
The algorithm takes four parameters as *input*: (1) a nonnegative matrix  $A \in \mathbb{Z}^{m \times n}$ , (2) a nonnegative vector  $b \in \mathbb{Z}^m$ , (3) the objective function  $c \in \mathbb{Z}^n$ , and (4) a branch-decomposition  $(T, \nu)$  of  $A$  with width  $k$ , and returns as *output*  $x \in \mathbb{Z}^n$  such that  $c^T x$  is maximized and  $Ax = b$ . The algorithm solves this maximization problem in  $O((d+1)^{2k}mn + m^2n)$ , where  $d = \max\{b_1, \dots, b_m\}$ . We note that for classes of matrices with branch-decompositions of constant width  $k$ , the CG algorithm runs *pseudopolynomial-time*, since the runtime is polynomial in the numeric entries of  $b$ .

The CG algorithm runs by combining a depth-first search of the tree  $T$  with dynamic programming. We begin by defining the internal data structures used within the algorithm, and then describe the algorithm itself. Let  $A' = [A \ b]$  (the matrix  $A$  augmented with the vector  $b$ ),  $E = \{1, \dots, n\}$ , and  $E' = \{1, \dots, n+1\}$ . For  $X \subseteq E$ , let

$$\begin{aligned} \mathcal{B}(X) = \{b' \in \mathbb{Z}^m : & \text{(i) } 0 \leq b' \leq b, \\ & \text{(ii) } \exists z \in \mathbb{Z}^{|X|}, z \geq 0 \text{ such that } (A|X)z = b', \text{ and} \\ & \text{(iii) } b' \in \text{span}(A'|(E' - X))\}. \end{aligned}$$

The integer program  $\max\{c^T x : Ax = b, x \geq 0, x \in \mathbb{Z}^n\}$  is *feasible* if and only if  $b \in \mathcal{B}(E)$ . Given a branch-decomposition  $(T, \nu)$ , there are particular sets  $X$  which can be calculated by combining elements from previously constructed sets, i.e., *T-branched* sets. A set  $X \subseteq E$  is *T-branched* if there is an edge  $e \in E(T)$  such that  $X$  is the label-set of one of the components of  $T - e$ .

EXAMPLE 2. For example, consider the two disconnected trees:



In the tree  $T - (5, c)$ , we see that the sets  $\{5\}$  and  $\{1, 2, 3, 4, 6\}$  are  $T$ -branched, and in tree  $T - (b, d)$ , we see that the sets  $\{1, 2\}$  and  $\{3, 4, 5, 6\}$  are  $T$ -branched. However, observe that the sets  $\{1, 2, 6\}$  and  $\{3, 4, 5\}$  are *not*  $T$ -branched, since there is no edge  $e$  such that  $T - e$  disconnects the graph into components labeled with either of those two sets.

We note that the set of leaves under a given parent is always a  $T$ -branched set. We will now relate  $T$ -branched sets to dynamic programming. If a given set  $X$  with  $|X| \geq 2$  is  $T$ -branched, then  $X$  can be partitioned into two sets,  $X_L$  and  $X_R$  (corresponding to the left and right children of a given interior node) such that  $X = X_L \cup X_R$ . In this case, we can redefine  $\mathcal{B}(X)$  in terms of the disjoint subsets  $X_L$  and  $X_R$ :

$$\begin{aligned} \mathcal{B}(X) = \{b' \in \mathbb{Z}^m: & \text{(i) } 0 \leq b' \leq b, \text{ (ii) } \exists b_L \in \mathcal{B}(X_L) \text{ and} \\ & b_R \in \mathcal{B}(X_R) \text{ such that } b' = b_L + b_R, \text{ and} \\ & \text{(iii) } b' \in \text{span}(A' | X) \cap \text{span}(A' | (E' - X))\}. \end{aligned}$$

Having defined  $T$ -branched sets and re-defined  $\mathcal{B}(X)$  when  $X$  is  $T$ -branched, we can now describe the precise steps of the CG algorithm. We determine the center of the tree, and then root the tree at its center (subdividing an edge and creating a new node if necessary). If the center node has degree three, then we arbitrarily choose an edge adjacent to the center and subdivide, ensuring that the root node always has exactly two children. We then walk the nodes of the tree in post-DEPTH-FIRST-SEARCH order. Since the tree  $T$  is cubic (with the exception of the root node), when considered in post-DEPTH-FIRST-SEARCH order, every internal node has two children and one parent. Thus, every  $T$ -branched set  $X$  with  $|X| \geq 2$  can be easily partitioned into two  $T$ -branched sets  $X_L$  and  $X_R$  corresponding to the two children. Then, we simply take linear combinations of the vectors in  $X_L$  and  $X_R$ , such that the conditions for inclusion in  $\mathcal{B}(X)$  are satisfied. When we reach the root, we check all feasible solutions in  $\mathcal{B}(E)$  and find the optimal according

**Algorithm:** Cunningham-Geelen Algorithm

**Input:** (1) A nonnegative matrix  $A \in \mathbb{Z}^{m \times n}$ ,  
(2) A nonnegative vector  $b \in \mathbb{Z}^m$ ,  
(3) The objective function  $c \in \mathbb{Z}^n$ , and  
(4) A branch-decomposition  $(T, \nu)$  of  $A$  with width  $k$ .  
**Output:**  $x \in \mathbb{Z}^n$  such that  $c^T x$  is maximized and  $Ax = b$ ,  
or INFEASIBLE.

```

1 Initialize root at center of  $T$  (subdividing edge if necessary).
2 for each  $v \in V(T)$  in post-DEPTH-FIRST-SEARCH order do
3   if  $v$  is a leaf of  $T$  then
4     Calculate  $\mathcal{B}(X)$  associated with leaf  $v$ .
5   else
6     Set  $X_L$  and  $X_R$  to sets associated with left and
       right children of  $v$ .
7      $\mathcal{B}(X) \leftarrow \emptyset$ .
8     for each  $b_L \in \mathcal{B}(X_L)$  and  $b_R \in \mathcal{B}(X_R)$  do
9       if  $b_L + b_R \leq b$  and
         $b_L + b_R \in \text{span}(A' | X) \cap \text{span}(A' | (E' - X))$  then
10         $\mathcal{B}(X) \leftarrow \mathcal{B}(X) \cup (b_L + b_R)$ .
11      end if
12    end for
13  end if
14 end for
15 return (max of  $c^T x$  over  $b \in \mathcal{B}(E)$ ) or INFEASIBLE.
```

Figure 1 The Pseudocode for the CG Algorithm

to the maximum of the objective function. The pseudocode is given in Figure 1.

In terms of actually implementing the CG algorithm, it is easy to see that the performance is greatly affected by the method of determining if a given vector is in the intersection of two vector subspaces (criteria (iii) for a vector  $b' \in \mathcal{B}(X)$ ). In the next section, we investigate several methods for answering that question, and display experimental results.

## 4. Intersecting Two Vector Spaces

Let  $A \in \mathbb{Z}^{m \times n}$  with  $m \leq n$ . As before, let  $E := \{1, \dots, n\}$ , and given  $X \subseteq E$ , let  $A | X$  denote the submatrix of  $A$  containing only the columns of  $X$ . Given  $X \subseteq E$ , let  $Y := E - X$ . Then, for any partition  $(X, Y)$  of  $E$ , let  $S_X := \text{span}(A | X) \cap \text{span}(A | Y)$ . The goal of this section is to describe three different ways of determining whether or not a given vector is in  $S_X$ . The first was described in Cunningham and Geelen (2007); the second is a well-known numerical algorithm from Golub and Van Loan (1996) for generally finding the intersection of two subspaces, and the third is a straightforward application of properties of  $\mathcal{B}(X)$ . In this section, we describe each of these methods, and then display a computational comparison. The methods are implemented in C++ with LAPACK for any standard linear algebra calls.

**The CG Intersection Method.** The first algorithm for determining whether a given vector  $v \in S_X$  is from Cunningham and Geelen (2007). We will explicitly

describe a matrix  $M_X$  such that the column-span of  $M_X$  is equal to  $S_X$ . For the purpose of clearly explaining the notation, we will continuously work on the following example:

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 7 & 6 \\ 0 & 1 & 0 & 0 & 4 & 7 & 1 \\ 0 & 0 & 1 & 0 & 4 & 2 & 1 \\ 0 & 0 & 0 & 1 & 6 & 6 & 4 \end{bmatrix}, \quad X = \{2, 4, 6, 7\},$$

$$Y = \{1, 3, 5\}, \quad \text{and} \quad A|X = \begin{bmatrix} 0 & 0 & 7 & 6 \\ 1 & 0 & 7 & 1 \\ 0 & 0 & 2 & 1 \\ 0 & 1 & 6 & 4 \end{bmatrix}.$$

Let  $A_X := A|X$ ,  $A_Y := A|Y$ , and let  $B \subseteq E$  be a basis of the column-span of  $A$ . For example, in the matrix  $A$  given above,  $B = \{1, 2, 3, 4\}$ . Finally, given two matrices  $V$  and  $W$ , with an equal number of rows in each, denote the matrix  $[V \ W]$  as the matrix consisting of all the columns in both  $V$  and  $W$ . We can reorder the columns of  $A_X$  and  $A_Y$  such that  $\bar{A}_X := [A|(B \cap X) \ A|(X - B)]$  and  $\bar{A}_Y := [A|(B \cap Y) \ A|(Y - B)]$ . In the case of  $A$  and  $X$  given as earlier,

$$\bar{A}_X = \begin{bmatrix} B \cap X & X - B \\ 0 & 0 & 7 & 6 \\ 1 & 0 & 7 & 1 \\ 0 & 0 & 2 & 1 \\ 0 & 1 & 6 & 4 \end{bmatrix} \quad \text{and} \quad \bar{A}_Y = \begin{bmatrix} B \cap Y & Y - B \\ 1 & 0 & 1 \\ 0 & 0 & 4 \\ 0 & 1 & 4 \\ 0 & 0 & 6 \end{bmatrix}.$$

In our example, notice that the basis  $B$  is the standard basis, and the matrix  $A$  is already in standard form (in other words,  $A = [I \ N]$ ). By using elementary column operations (which do not change the range of the column-span), we create the partially reduced matrices  $\text{red}(\bar{A}_X)$  and  $\text{red}(\bar{A}_Y)$  by cancelling the entries in the columns  $X - B$  (or  $Y - B$ ) that correspond to rows with nonzero entries in the columns  $X \cap B$  (or  $Y \cap B$ ), respectively. Notice that these partially reduced matrices are different from the standard reduced row echelon form. For example,

$$\text{red}(\bar{A}_X) = \begin{bmatrix} B \cap X & X - B \\ 0 & 0 & 7 & 6 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 2 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix} \quad \text{and}$$

$$\text{red}(\bar{A}_Y) = \begin{bmatrix} B \cap Y & Y - B \\ 1 & 0 & 0 \\ 0 & 0 & 4 \\ 0 & 1 & 0 \\ 0 & 0 & 6 \end{bmatrix}.$$

We claim that  $M_X = [\text{red}(\bar{A}_X) \mid (X - B) \ \text{red}(\bar{A}_Y) \mid (Y - B)]$ . In our running example,

$$M_X = \begin{bmatrix} X - B & Y - B \\ 7 & 6 & 0 \\ 0 & 0 & 4 \\ 2 & 1 & 0 \\ 0 & 0 & 6 \end{bmatrix}.$$

Notice that the  $\dim(S_X) = 3$  and  $\text{rank}(M_X) = 3$  in our example. The following lemma is a correction to the claim in Section 3 of Cunningham and Geelen (2007). Following the notation of Cunningham and Geelen (2007), note that given  $U \subseteq B$  and  $V \subseteq E$ ,  $A[U, V]$  denotes the submatrix of  $A$  with rows indexed by  $U$  and columns indexed by  $V$ .

LEMMA 1. For any partition  $(X, Y)$  of  $E$ ,

$$\lambda_A(X) = \text{rank}(A[B - X, X - B]) + \text{rank}(A[B - Y, Y - B]) + 1.$$

Moreover,  $S_X$  is the column-span of the matrix

$$M_X = [\text{red}(\bar{A}_X) \mid (X - B) \ \text{red}(\bar{A}_Y) \mid (Y - B)].$$

PROOF. We must show that  $S_X \subseteq \text{span}(M_X)$  and  $\text{span}(M_X) \subseteq S_X$ . In the first case, consider a vector  $v \in S_X$ . Then  $v \in \text{span}(A|X)$  and  $v \in \text{span}(\text{red}(\bar{A}_X))$ , and likewise,  $v \in \text{span}(A|Y)$  and  $v \in \text{span}(\text{red}(\bar{A}_Y))$ . Thus,  $v$  can be written as a linear combination of columns in  $\text{red}(\bar{A}_X)$  (i.e.,  $v = \text{red}(\bar{A}_X)\alpha$ ), and  $v$  can also be written as a linear combination of the columns in  $\text{red}(\bar{A}_Y)$  (i.e.,  $v = \text{red}(\bar{A}_Y)\alpha'$ ).

We will construct an  $(|X - B| + |Y - B|)$  length vector  $\alpha''$  using entries from both  $\alpha$  and  $\alpha'$ . Set the first  $|X - B|$  entries in  $\alpha''$  to the last  $|X - B|$  entries in  $\alpha$ , and the second  $|Y - B|$  entries in  $\alpha''$  to the last  $|Y - B|$  entries in  $\alpha'$ . Then,  $v = M_X\alpha''$ , and  $S_X \subseteq \text{span}(M_X)$ .

Conversely, we must show that  $\text{span}(M_X) \subseteq S_X$ . Consider a vector  $v \in \text{span}(M_X)$  such that  $v = M_X\alpha''$  for some  $\alpha''$ . We will construct corresponding  $\alpha$ ,  $\alpha'$  such that  $v = \text{red}(\bar{A}_X)\alpha = \text{red}(\bar{A}_Y)\alpha'$ . Towards that end, set the first  $|X \cap B|$  entries in  $\alpha$  to the entries in  $v$  indexed by the  $X \cap B$  rows, and the last  $|X - B|$  entries in  $\alpha$  to the first  $|X - B|$  entries in  $\alpha''$ . To construct  $\alpha'$ , set the first  $|Y \cap B|$  entries in  $\alpha'$  to the entries in  $v$  indexed by the  $Y \cap B$  rows, and the last  $|Y - B|$

entries in  $\alpha'$  to the last  $|Y - B|$  entries in  $\alpha''$ . Then,  $v = \text{red}(\bar{A}_X)\alpha = \text{red}(\bar{A}_Y)\alpha'$  as desired, and  $\text{span}(M_X) \subseteq S_X$ .

Finally, we must show that

$$\lambda_A(X) = \text{rank}(A[B - X, X - B]) \\ + \text{rank}(A[B - Y, Y - B]) + 1.$$

Since it is well known that  $\lambda_A(X) = \dim(S_X) + 1$ , it suffices to show that

$$\dim(S_X) = \text{rank}(A[B - X, X - B]) \\ + \text{rank}(A[B - Y, Y - B]).$$

When we inspect the matrix  $M_X$  (a basis for  $S_X$ ), we see that any two vectors, one from the first  $(X - B)$  columns and one from the last  $(Y - B)$  columns, are linearly independent (since the zero entries always appear in different rows). Thus,  $\text{rank}(M_X) = \text{rank}(M_X | (X - B)) + \text{rank}(M_X | (Y - B))$ . However, recall that the two submatrices,  $M_X | (X - B)$  and  $M_X | (Y - B)$ , each contain rows of zeros where the entries were cancelled during the conversion of  $\bar{A}_X$  to  $\text{red}(\bar{A}_X)$ , and  $\bar{A}_Y$  to  $\text{red}(\bar{A}_Y)$ , respectively. Therefore, when the rows of zeros are removed from the submatrices  $M_X | (X - B)$  and  $M_X | (Y - B)$ , we see that the  $\text{rank}(M_X | (X - B)) = \text{rank}(A[B - X, X - B])$  and  $\text{rank}(M_X | (Y - B)) = \text{rank}(A[B - Y, Y - B])$ . Since the  $\text{rank}(M_X) = \dim(S_X)$ , this concludes our proof.  $\square$

When the matrix  $A$  is not in standard form, we rewrite  $A$  as  $[A_B \ N]$ , and calculate the matrix  $A_B^{-1}$  such that  $[A_B \ N] \cdot A_B^{-1} = [I \ N]$ . Then, when we determine if a given vector  $v \in S_X$ , we test if  $A_B^{-1}v \in \text{span}(A_B^{-1}M_X)$ . We note that finding  $A_B^{-1}$  is basically a free computation within the algorithm for finding the maximal set of linearly independent columns  $B$ . Additionally, when  $X$  is a  $T$ -branched set, we see from Cunningham and Geelen (2007) that dynamic programming techniques can be used to calculate a basis of  $S_X$  via the previously calculated bases of  $S_{X_L}$  and  $S_{X_R}$ , yielding a runtime of  $O(k^2m)$  per basis. In the final runtime analysis of the CG algorithm, this is the method used for all  $S_X$  inclusion tests.

**The Numerical Method.** The second method is a well-known numerical algorithm for finding a basis for the intersection of two subspaces, and is described in detail in Golub and Van Loan (1996, §12.4.4, p. 604). To briefly summarize, given  $A \in \mathbb{R}^{m \times p}$  and  $B \in \mathbb{R}^{m \times q}$  (with  $p \geq q$ ), we calculate the two QR factorizations  $A = Q_A R_A$  and  $B = Q_B R_B$ , and then take the singular value decomposition of  $Q_A^T Q_B$ . Using this singular value decomposition, we can explicitly write down a numerical basis of  $\text{span}(A) \cap \text{span}(B)$ . This algorithm requires approximately  $4m(q^2 + 2p^2) + 2pq(m + q) + 12q^3$  flops, as compared to the  $O(k^2m)$  runtime of the CG intersection method.

**The  $\text{span}(Y)$  Method.** Finally, we notice that if a given vector  $b'$  is in  $\mathcal{B}(X)$ , by criteria (ii), this implies that there exists a  $z \in \mathbb{Z}^{|X|}$  with  $z \geq 0$  such that  $A_X z = b'$ . In other words, we already know that  $b' \in \text{span}(A_X)$ , and we only need to check if  $b' \in \text{span}(A_Y)$ . In particular, in line 8 of the pseudocode for the CG algorithm, we observe that it is already known that  $b_L \in \mathcal{B}(X_L)$  and  $b_R \in \mathcal{B}(X_R)$ . Therefore, in line 9, we already know that  $b_R + b_L \in \text{span}(A' | X)$ , and we only need to check if  $b_R + b_L \in \text{span}(A' | Y)$ . Since the matrix  $A_Y$  is easy to construct (as compared to the computational complexity inherent in the other methods), it is worthwhile to investigate how this simplified method behaves in practice.

**Computational Results for Intersection Tests.** The computational results for the intersection test comparisons are displayed in Table 1. We test these three intersection algorithms on randomly generated set partition instances (see §5.3.2 for a complete description) on a AMD Opteron dual-core processor with 3 GHz clock speed, 4 GB of RAM and 2 GB of swap space. These instances are infeasible. We denote the three methods as CG (for the Cunningham-Geelen matrix construction algorithm), QR (for the numerical, QR-factorization-based algorithm) and finally,  $\text{span}(Y)$  (to denote we are only checking the  $\text{span}(Y)$ ). The methods are each implemented in C++ with LAPACK for any standard linear algebra calls.

In general, the  $\text{span}(Y)$  method is the most practical, and it will become the default method for our computational investigations. However, we note that the CG intersection method is extremely fast on square matrices, which have lower width. For example, on the  $750 \times 750$  instance, the CG method finishes in nine seconds, while the QR method does not terminate, and the  $\text{span}(Y)$  method takes 403 seconds.

**Table 1** Computational Investigations on Intersection Algorithms

Rows	Columns	CG (seconds)	QR (seconds)	$\text{Span}(Y)$ (seconds)
50	250	17	6	3
100	250	80	28	5
150	250	77	51	11
200	250	7	13	6
250	250	1	15	4
100	500	488	142	29
200	500	2,360	915	85
300	500	2,381	1,508	185
400	500	237	287	114
500	500	3	320	67
150	750	3,565	1,132	169
300	750	—	—	513
450	750	—	—	1,057
600	750	—	—	621
750	750	9	—	403
200	1,000	—	—	577
400	1,000	—	—	1,723
600	1,000	—	—	3,485

In §5.3.2, we investigate this discrepancy further, and propose a class of square, infeasible set partition instances for testing with the CG algorithm based on this result. The “—” signifies that the computation was terminated after four hours.

## 5. Computational Results for the CG Algorithm

In this section, we summarize the computational results for our implementation of the CG method. We experimented with two different implementations (a low-memory version and a memory-intensive version), and three different methods of computing the intersection  $S_X$  (previously described in §4). In §5.1, we describe the nuances of our implementation. In §5.2, we describe the different types of trees we used as input. In §5.3, we describe our computational investigations on graph 3-coloring, set partition, market split, and knapsack instances. To summarize, the CG method was not successful (as currently implemented) on the graph 3-coloring instances, partially successful on the set partition instances, partially successful on the feasible knapsack instances, and very successful on the infeasible market split and knapsack instances with width  $\leq 6$ . For a particularly demonstrative example, CG runs the infeasible knapsack instance `ex6_Original_10000.lp` in 1,188 seconds  $\approx$  19.8 minutes while GUROBI runs in 13,464 seconds  $\approx$  3.74 hours.

### 5.1. Implementation Details

The two most significant challenges faced during the implementation of the CG algorithm were the challenge of managing the memory of the  $\mathcal{B}(X)$  sets, and the challenge of quickly iterating the combinations. Toward managing the memory of the  $\mathcal{B}(X)$  sets, we store only a scalar/vector pair  $\{\alpha, b'\}$ , where the vector  $b' \in \mathcal{B}(X)$ , and the scalar  $\alpha$  is the largest multiplier such that  $\alpha b' \leq b$ . In our memory-intensive implementation, given two sets of scalar/vector pairs  $\{\alpha, b_1\}, \{\beta, b_2\}$ , we compute  $b' = \alpha' b_1 + \beta' b_2$ , where  $0 \leq \alpha' \leq \alpha$  and  $0 \leq \beta' \leq \beta$  and determine whether or not  $b' \in S_X$  for each linear combination. In the low-memory implementation, we rely on the following three observations. Given vectors  $v_1, v_2$  and a vector space  $W$ ,

1. if  $v_1, v_2 \in \text{span}(W)$ , then  $\alpha v_1 + \beta v_2 \in \text{span}(W)$ .
2. if  $v_1 \in \text{span}(W)$ , but  $v_2 \notin \text{span}(W)$ , then  $\alpha v_1 + \beta v_2 \notin \text{span}(W)$  (for  $\beta \neq 0$ ).
3. if  $v_1, v_2 \notin \text{span}(W)$ , then  $\alpha v_1 + \beta v_2$  may or may not be in  $\text{span}(W)$ , and the combination must be explicitly checked.

These observations allow us to significantly reduce the number of span computations, while calculating a “generating set” of vectors for  $\mathcal{B}(X)$ . For

example, when running the  $4 \times 30$  market split instance (see §5.3.3 for a complete description), we see that the memory-intensive implementation runs for over an hour while consuming 4 GB of RAM and 2 GB of swap space before being killed remotely by the system. Additionally, the last  $\mathcal{B}(X)$  set calculated before the process was terminated contained 20,033,642 vectors. By contrast, the low-memory version runs in 2.88 seconds, consumes under 0.1% of memory, and has a root  $\mathcal{B}(X)$  set consisting of only 30 vectors.

The drawback of the low-memory implementation is that we are no longer required to only consider pairwise-linear combinations: we must also consider triples and quadruples, etc., to ensure that we are not missing any essential vectors. This particular aspect of our code has not been fully optimized. For example, on the  $450 \times 588$  infeasible graph 3-coloring instance (see §5.3.1 for a complete description), one of the  $\mathcal{B}(X)$  sets contain over 200 vectors. This is trivial to manage for the memory-intensive implementation, and the code terminates with the infeasible answer in 230 seconds. But iterating the  $\binom{220}{2}, \binom{220}{3}, \binom{220}{4}$ , etc. combinations (24,090, 1,750,540 and 94,966,795, respectively) and computing the necessary span checks runs overnight without finishing. Thus, as currently implemented, sometimes the memory-intensive version is more efficient, and sometimes the low-memory version is more efficient. In our experimental results, we will use both.

Finally, when iterating through the combinations at the root node of the branch-decomposition tree, we must quickly determine if a given combination of scalar/vector pairs sums to the vector  $b$ . Thus, we create a “hash” value for each vector (and for the vector  $b$ ), which is quickly computable and easily comparable. Through trial and error, we settled on the following “hash” function:

$$\text{hash}(v) = \sum_{i=1}^m (i + 65,599) v_i.$$

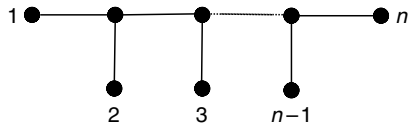
Obviously, two different vectors can “hash” to the same value. In this case, we must do a direct component-by-component comparison to determine if a given combination does indeed sum to the vector  $b$ . However, the “hash” function often spares us the expensive direct comparison. Furthermore, since the  $v_i$  values are 0/1 in our problem instances, the “hash” function allows us to sort the vectors in increasing value, which allows us to use binary search techniques to isolate combinations that sum to the vector  $b$ . An obvious avenue for further optimization would be to explore the practical behavior of different hash functions that still permit this sorting property on 0/1 vectors.

## 5.2. Trees

The CG algorithm takes as input the normal parameters for an integer program (the matrix  $A$ , right-hand side vector  $b$ , and objective function  $c$ ), but it also takes as input a branch-decomposition  $(T, \nu)$  of the matrix  $A$ . In our computational investigations, we often tested the same integer program input with several different branch-decompositions. These trees are optimal branch-decompositions, branch-decompositions derived heuristically, and “worst-case” branch-decompositions. The optimal branch-decompositions were constructed via the algorithm described by Hicks (2005) with code provided by the author. However, we were unable to obtain optimal trees for many of our larger test cases, which is expected since finding an optimal branch-decomposition is NP-hard (Seymour and Thomas 1994, Hicks and McMurray 2007).

The heuristically derived trees were computed via the algorithms described in Ma et al. (2012) with code provided by the authors. In Ma et al. (2012), the authors describe two different methods for finding near-optimal branch-decompositions of linear matroids, based on classification theory and max-flow algorithms, respectively. The authors introduce a “measure” which compares the “similarity” of elements of the linear matroid, which reforms the linear matroid into a similarity graph. The method runs in  $O(n^3)$  time, and is implemented in MATLAB. This runtime is *not* included in our tables, both because it is not an optimized implementation, and also because the CG algorithm specifies the tree  $T$  as input. All heuristic trees used in our experiments were derived using the max-flow algorithm.

Finally, we experimented with “worst-case” caterpillar trees. A *caterpillar* tree is formed by taking the  $n$  columns of a matrix and distributing them across the legs of a long tree such that every interior node (with the exception of the two ends) is adjacent to a single leaf, and the two end interior nodes are each adjacent to two leaves.



The width of a caterpillar tree may be the worst possible, since the columns are always assigned to leaves in order 1 to  $n$ , regardless of the linear independence of the columns. However, constructing such trees is fast and trivial, and as we will see in our experimental investigations, surprisingly useful.

## 5.3. Experimental Results

We tested our code on several different types of problems: graph 3-coloring, set partition, market split, and

knapsack. We ran our instances on a AMD Opteron dual-core processor with 3 GHz clock speed, 4 GB of RAM and 2 GB of swap space. We tested our code against GUROBI, version 3.0 (Bixby et al. 2010), which is a well-known commercial software for solving integer programs. We note that GUROBI is considered competitive with CPLEX on integer programming performance benchmarks (Mittelmann 2012). We also comment that GUROBI has an option to turn off multi-threading, which we used to ensure a more fair competition between GUROBI and our single-threaded implementation of the CG algorithm. Additionally, since finding a branch-decomposition is a different problem from using a branch-decomposition to solve an integer program with the CG algorithm, we do *not* include the time to find the branch-decomposition in any of the experimental tables in this section. The time is negligible in any case, since we use heuristics to find the branch-decompositions.

**5.3.1. Graph 3-Coloring Instances.** In this section, we describe the experimental results on a particular class of graph 3-coloring instances. In this case, the CG algorithm was not competitive with GUROBI.

Mizuno and Nishihara (2008) describe a randomized algorithm based on the Hajós calculus for generating infinitely large instances of quasi-regular, 4-critical graphs. When testing these graphs for 3-colorability, Mizuno and Nishihara (2008) experimented with numerous algorithms and software platforms, but always found exponential growth in the runtime for larger and larger instances. Based on these experimental observations, the authors propose these graphs as “hard” examples of 3-colorability. When converted to an integer program, there are three variables per vertex,  $x_{iR}, x_{iG}, x_{iB} \in \{0, 1\}$  and three slack variables per edge,  $s_{ijR}, s_{ijG}, s_{ijB} \in \{0, 1\}$ . There is one constraint per vertex  $x_{iR} + x_{iG} + x_{iB} = 1$ , and three constraints per edge:

$$x_{iR} + x_{jR} + s_{ijR} = 1, \quad x_{iG} + x_{jG} + s_{ijG} = 1, \quad \text{and} \\ x_{iB} + x_{jB} + s_{ijB} = 1.$$

Since these graphs are non-3-colorable, the corresponding integer programs are infeasible. We include these examples in our benchmark suite because they are infeasible, because they are purported to be hard for constraint-satisfaction software programs, and because the  $\max(b_i) = 1$ . Since the runtime of the CG algorithm in general is  $O((d+1)^{2k}mn + m^2n)$ , in this case, the runtime is  $O(2^{2k}mn + m^2n)$ . However, the CG algorithm is not competitive with GUROBI on these instances because the width is still too high.

The computational results for the graph 3-coloring instances are displayed in Table 2. We tested these instances with the memory-intensive implementation,

**Table 2** Infeasible Graph 3-Coloring Instances

Rows	Cols	Max(b)	Cat width	Sec.	Heuristic width	Sec.	GUROBI sec
64	84	1	22	36	22	0	0
131	171	1	42	—	42	2	1
195	255	1	62	—	62	5	0
255	333	1	80	—	80	10	1
450	588	1	140	—	140	—	16
510	666	1	158	—	158	—	112
652	852	1	—	—	—	—	682
719	939	1	—	—	—	—	1,873
772	1,008	1	—	—	—	—	9,676
836	1,092	1	—	—	—	—	16,033

and we see widths ranging from 22 to 158. The “—” signifies that the algorithm runs overnight without terminating. We note that GUROBI is faster than the CG algorithm (as currently implemented) on these instances. We next observe a surprising fact: the width of the caterpillar tree and the heuristic tree are the same. Since we were unable to obtain an optimal tree for these instances (84 columns is too large for the existing code), we do not know how close these widths are to optimal. Furthermore, despite identical widths, the runtime of the CG algorithm with the heuristic tree as compared to the caterpillar tree is dramatically different. For example, on the  $131 \times 171$  instance, although the width of both trees is 42, the heuristic tree runs in two seconds, but the caterpillar tree *does not terminate*. We provide an explanation for this discrepancy in §5.4.

**5.3.2. Set Partition Instances.** In this section, we describe the experimental results for randomly generated infeasible instances of set partition. Although the CG method is not generally competitive with GUROBI here, we isolate a special class of square infeasible instances where the CG algorithm runs twice as fast as GUROBI.

In the set partition problem, the  $A$  matrices are 0/1 matrices, and the right-hand side vector  $b$  contains only ones. Thus, an instance of set partition is  $Ax = 1$ , where  $A(i, j) = 1$  if and only if the integer  $i$  appears in the set  $M_j$ . The instance is feasible if there is a collection of sets  $M_{i_1}, \dots, M_{i_k}$  such that the intersection of any two sets is empty, and the union is the entire set of integers  $1, \dots, m$ . These randomly generated instances are infeasible, and were generated with the MATLAB command  $A = \text{double}(\text{rand}(m, n) > 0.20)$ .

The computational results for the randomly generated infeasible instances are displayed in Table 3, and the results for the infeasible square instances are displayed in Table 4. We tested these instances with the memory-intensive implementation, and we see widths ranging from 1 to 302 on these instances. We note that GUROBI is faster than the CG algorithm (as currently implemented) on these instances. We also

**Table 3** Randomly Generated Infeasible Set Partition Instances

Rows	Cols	Cat width	Sec.	Heuristic width	Sec.	GUROBI sec
50	250	51	6	51	4	0
100	250	101	28	98	4	1
150	250	102	27	94	8	0
200	250	52	9	52	7	1
250	250	2	9	2	10	0
100	500	101	143	101	45	2
200	500	201	908	185	55	4
300	500	202	749	202	107	1
400	500	102	166	102	130	2
500	500	2	177	2	177	2
150	750	151	1,133	151	85	16
300	750	301	7,094	299	317	16
450	750	302	5,846	302	735	3
600	750	102	1,168	152	842	6
750	750	2	1,547	2	1,104	7

see that although the width of the caterpillar trees is similar to the width of the trees produced by the heuristic, the runtime of the heuristic trees is significantly faster than the runtime of the caterpillar trees (discussed in §5.4). We also note that the full-rank, square set partition instances have branch-width one. We recall the results from the intersection testing in Table 1, which demonstrate that the CG intersection method was significantly faster than the other methods on square, full-rank matrices. It is therefore logical to combine instances with minimal branch-width and minimal right-hand side entries, and our next test is square, infeasible set partition instances and the CG method for testing the intersection.

The strength of the CG algorithm is shown in Table 4: the method is twice as fast as GUROBI on these low branch-width, low right-hand side examples.

**5.3.3. Market Split: Cornuéjols-Dawande Instances.** In this section, we describe the experimental results for the market-split instances, where we see that the CG method is competitive with GUROBI for widths  $\leq 6$ .

The following instances are the Cornuéjols-Dawande *market split* problems (see Aardal et al. 1999

**Table 4** Randomly Generated Square Infeasible Set Partition Instances

Rows	Cols	Cat width	Sec.	GUROBI sec.
250	250	2	0	2
1,000	1,000	2	23	46
1,500	1,500	2	80	154
2,000	2,000	2	190	570
3,000	3,000	2	858	2,024
3,500	3,500	2	1,063	3,226
4,000	4,000	2	1,528	4,900
4,500	4,500	2	2,588	6,960
5,000	5,000	2	3,378	9,589
5,500	5,500	2	5,208	12,626



**Table 5** Infeasible Market Split Instances

Rows	Cols	Max(b)	Cat width	Sec.	Heuristic width	Sec.	Opt width	Sec.	GUROBI sec
2	10	278	3	0	3	0	3	0	0
3	20	572	4	1	4	0	4	0	3
4	30	941	5	2	5	3	5	2	37
5	40	1,067	6	1,885	6	1,900	6	*	2,235
6	50	1,264	7	—	7	—	7	—	5,631

and references therein). In this case, the matrix  $A$  is  $m \times 10(m-1)$  with entries drawn uniformly at random from the interval  $[1, 99]$ . The right-hand side vector  $b$  is defined as  $b_i = \lfloor \frac{1}{2} \sum_{j=1}^n a_{ij} \rfloor$ . These instances are infeasible, and we tested with the low-memory version of the CG algorithm. The computational results are displayed in Table 5.

In this case, the CG method is significantly faster than GUROBI on instances with relatively small widths ( $\leq 6$ ), but GUROBI scales better with instance size. For example, while the CG method runs faster than GUROBI on the market split  $5 \times 40$  instance, on the  $6 \times 50$  instance, the CG method runs overnight without terminating while GUROBI simply doubles in time. We note that we were unable to obtain an optimal tree for the  $5 \times 40$  instance, which is why a “\*” appears, rather than the customary nontermination “—.”

**5.3.4. Knapsack Instances.** In this section, we describe the experimental results for knapsack instances, where we see that the CG method is significantly faster than GUROBI on the infeasible instances.

A “knapsack” problem is a “packing” problem represented by  $Ax = b$ , where  $x \in \{0, 1\}$ , the matrix  $A$  consists of a single row, and the variables  $x_i$  are items that can be “packed” or “left behind.” Thus, the coordinate  $A_{1i}$  represents the weight of the item  $x_i$ , and  $b$  (a single integer) represents the total weight that can be carried in the “knapsack.” Pataki et al. (2010) describe hard, infeasible knapsack instances (available online at <http://www.unc.edu/~pataki/instances/marketshare.htm>). In these infeasible knapsack instances, the matrices  $A$  are  $5 \times 40$ , which implies that these are multiple-knapsack problems, i.e., the goal is to use 40 items to pack five knapsacks. If an item appears in one knapsack, it must appear in all knapsacks.

To be thorough, we test both feasible and infeasible knapsack instances. The low-memory version of the CG algorithm is always used. While GUROBI is faster than CG on the feasible instances, CG is significantly faster than GUROBI on the infeasible knapsack instances. To construct a feasible instance, we remove three rows from each of the  $5 \times 40$  Pataki matrices, which allows these instances to become feasible. The computational results from these constructed feasible knapsack instances are described below in Table 6.

**Table 6** Feasible Knapsack Instances Created from Infeasible Instances from Pataki et al. (2010)

Name	Rows	Cols	Max(b)	Row set	Cat width	Sec.	GUROBI sec
ex1_Original_10000.lp	2	40	101,368	{1, 2}	3	2,792	534
ex2_Original_10000.lp	2	40	110,947	{1, 2}	3	1,676	144
ex3_Original_10000.lp	2	40	119,606	{1, 4}	3	1,658	> 15,848
ex4_Original_10000.lp	2	40	95,708	{1, 4}	3	2,599	1,491
ex5_Original_10000.lp	2	40	104,334	{1, 2}	3	2,207	1,321

Although GUROBI is almost always faster than the CG algorithm on these instances, the behavior of GUROBI on the Pataki example ex3\_Original\_10000.lp (with rows one and four removed) is worth noting. While the CG algorithm terminated with an optimal solution in 1,658 seconds  $\approx 28$  minutes, we could not run GUROBI to termination on our machine. Indeed, after over 15,848 seconds  $\approx 4.5$  hours, GUROBI used 4 GB of RAM and 2 GB of swap space on our machine and the process was killed remotely by the system.

In Table 7, we see the computational results for the infeasible Pataki knapsack instances. In these instances, the strength of the CG algorithm is truly shown: the CG method runs in minutes, whereas GUROBI runs in hours. However, we note that, in each of these knapsack instances, both feasible and infeasible, the width of the trees is equal to the rank of the matrix plus one. Thus, these trees do not allow the CG method to filter out any excess vectors. However, in the infeasible case, because the width of these trees is comparatively low ( $\leq 6$ ), the CG method is significantly faster than the more traditional integer programming methods of GUROBI.

We conclude by commenting that these knapsack instances were deliberately designed by Pataki and Krishnamoorthy (2009) and Pataki et al. (2010) to be difficult to solve by conventional methods, but with the multiplication by a unimodular matrix, the instances are reformulated and then can be solved in seconds by software such as GUROBI.

**Table 7** Infeasible Knapsack Instances from Pataki et al. (2010)

Name	Rows	Cols	Max(b)	Heuristic width	CG sec.	GUROBI sec
ex1_Original_10000.lp	5	40	112,648	6	534	4,969
ex2_Original_10000.lp	5	40	110,947	6	1,681	11,257
ex3_Original_10000.lp	5	40	119,606	6	343	4,122
ex4_Original_10000.lp	5	40	116,946	6	1,880	10,388
ex5_Original_10000.lp	5	40	104,334	6	2,644	4,570
ex6_Original_10000.lp	5	40	108,565	6	1,188	13,464
ex7_Original_10000.lp	5	40	105,870	6	1,340	4,819
ex8_Original_10000.lp	5	40	106,495	6	925	10,085
ex9_Original_10000.lp	5	40	112,366	6	1,969	11,584
ex10_Original_10000.lp	5	40	102,170	6	2,189	6,140

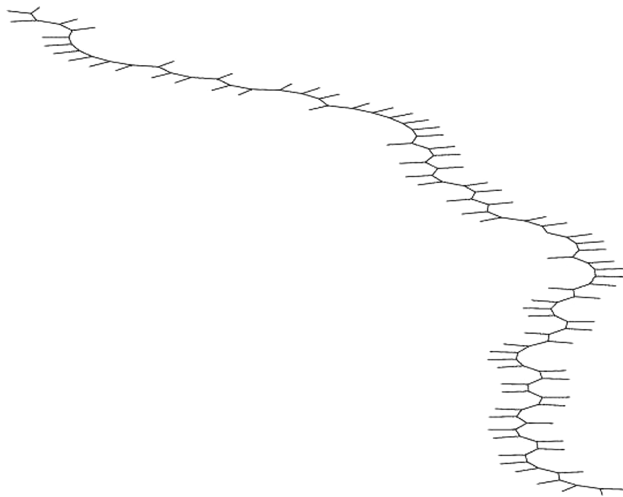


Figure 2 Caterpillar Tree

#### 5.4. Branch-Decompositions and Edge-Weight Dispersion

In §§5.3.1 and 5.3.2, we investigated the behavior of the CG algorithm on graph 3-coloring and set partition instances. In both cases, we observed that despite similar widths, the runtime of the CG algorithm on the heuristic trees was significantly faster than the runtime on the caterpillar trees. For example, in Table 2, we see that the  $64 \times 84$  graph 3-coloring instance runs in 36 seconds versus 0 seconds, even though both trees have width 22. This is surprising in light of the  $O((d+1)^{2k}mn + m^2n)$  runtime of the CG algorithm, where  $d = \max\{b_1, \dots, b_m\}$  and  $k$  is the width of the tree. In this section, we analyze the internal structure of these two trees and also the internal data of the CG algorithm on these two instances in order to explain the 36 seconds versus 0 seconds runtime difference.

In Figures 2 and 3, we see a dramatic difference in the structure of the caterpillar and heuristic trees

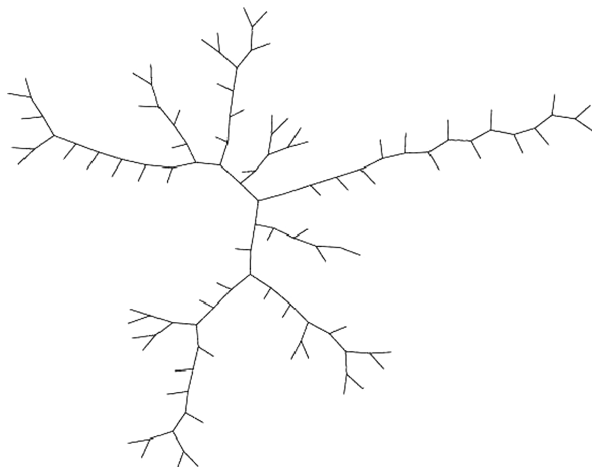


Figure 3 Heuristic Tree

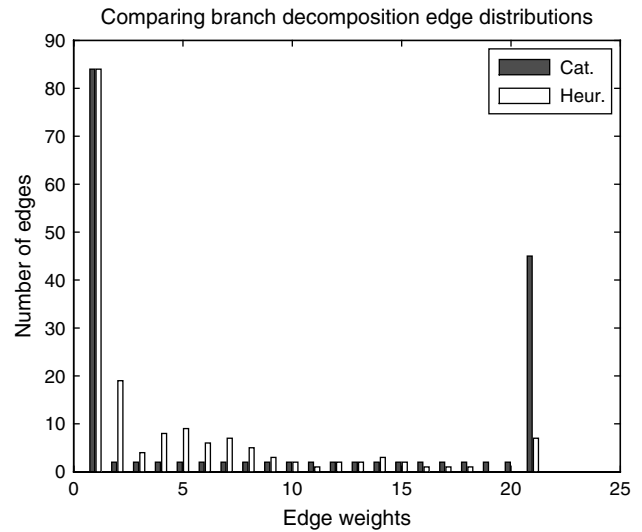


Figure 4 Distribution of Edge Weights between Caterpillar and Heuristic Trees

of the  $64 \times 84$  graph 3-coloring instance. The caterpillar tree is by definition a long and straight “caterpillar,” but the heuristic tree has a “star-like” structure with many different branches and sub-branches. We will see that this “star-like” structure, coupled with the edge weight distribution, allows the CG algorithm to filter out a significantly larger number of vectors before creating the search space  $\mathcal{B}(X)$  of the root node.

In Figure 4, we compare the edge weight distributions of the caterpillar and heuristic trees of the  $64 \times 84$  graph 3-coloring instance. Although both trees have width 22, we see that the number of edges with larger edge weights is significantly higher in the caterpillar tree than in the heuristic tree. For example, we see that there are 45 edges with edge weight 21 in the caterpillar tree, and only seven edges with edge weight 21 in the heuristic tree. While both trees have 84 edges with edge weight 1, the heuristic tree has 19 edges with weight 2, and the caterpillar tree has only two edges with weight 2. We will see that the performance increase on the heuristic tree is *not* due to a difference in width, but rather due to the fact that there are significantly *more* edges with *less* weight in the heuristic tree.

In Figure 5, we compare the internal data of the CG algorithm on the caterpillar and heuristic trees. On the upper and lower left, we track the dimension of the vector space  $S_X$  versus the internal nodes of the caterpillar and heuristic trees. Both trees have 83 internal nodes which are ordered by the post-DFS of the CG algorithm, and the 83rd node of both trees is the root node. Since both trees have a width of 22, both trees attain the same maximum dimension of 21. However, from Figure 2, we see that the caterpillar tree has only two branches, and from Figure 4,

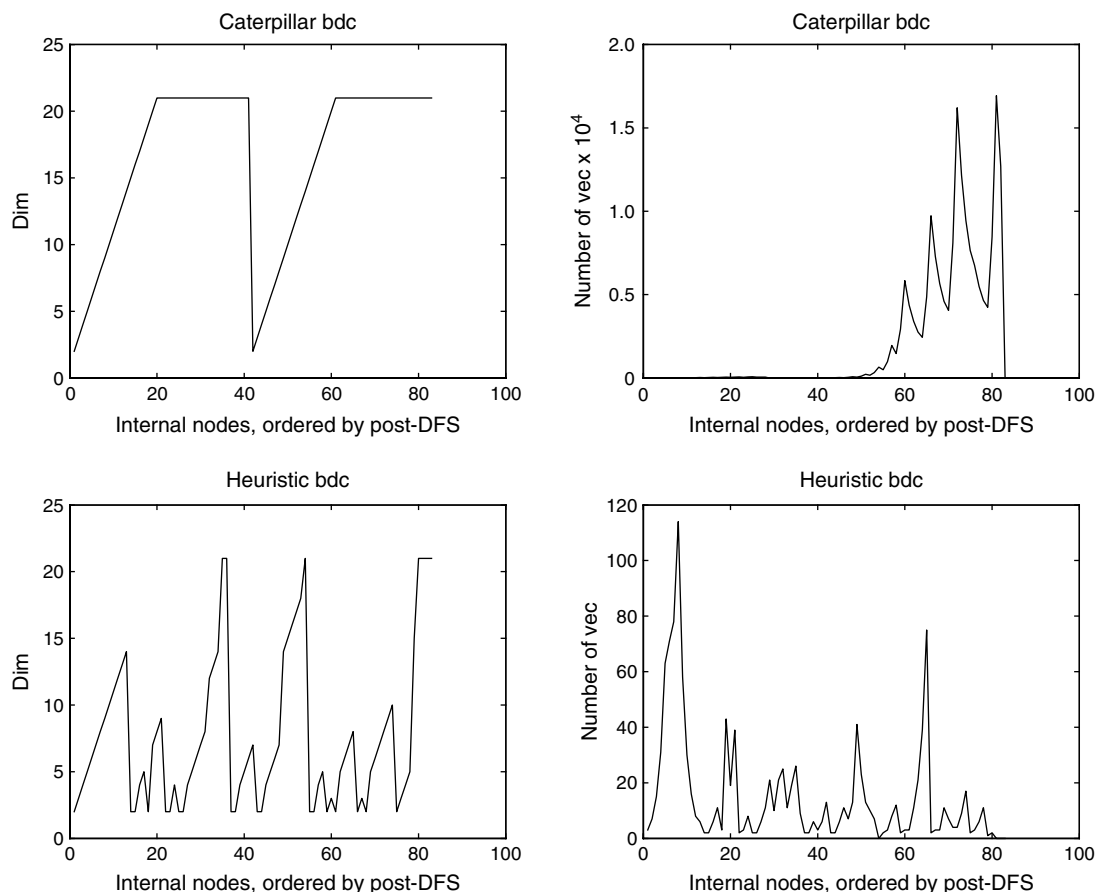


Figure 5 Internal Data from the CG Algorithm for the  $64 \times 84$  MUG Instance

we see that the caterpillar tree has 45 nodes with  $S_X$  dimension 21. This explains the two flat peaks in the caterpillar dimension graph (Figure 5, upper left). On the other hand, we see from Figure 3 that the heuristic tree has *many* branches, and from Figure 4, only seven nodes with  $S_X$  dimension 21. Thus, in the lower left of Figure 5, we see a graph with a series of jagged peaks, attaining the maximum dimension in several places, including a brief flat peak at the final root node.

However, the right side of Figure 5 displays the truly important data: the number of vectors stored in the set  $\mathcal{B}(X)$  associated with the internal nodes. For the heuristic tree, the maximum number of vectors (114) is attained at the *beginning* of the post-DFS, and although the number of vectors rises and falls throughout the algorithm, by the time the root node is searched, even though the *dimension* of the space is 21, enough vectors have been filtered out that there are only *two* vectors left in the set. By dramatic contrast, for the caterpillar tree, the leaves are ordered in such a way that the last 20 nodes have  $S_X$  dimension 21. Thus, comparatively few vectors are filtered out, and number of vectors climbs steadily from 3 to 16,927 as the second branch is processed. Here is the reason for the difference in computation time: for the heuristic

tree, the internal node with the largest number of vectors is the 9th node processed, with a dimension of 9 and a size of 114. However, for the caterpillar tree, the internal node with the largest number of vectors is the 81st node processed, with a dimension of 21 and a size of 16,927.

This analysis of the internal workings of the CG algorithm emphasizes the importance of the edge weight distribution as well as the maximum width, and will hopefully encourage research into branch-decomposition heuristics that produce trees with both lower width and fewer edges with larger weights.

## 6. Conclusion

In this paper, we demonstrate a specific niche for the CG algorithm. On infeasible market split and knapsack problems with branch-width  $\leq 6$ , the CG algorithm runs in minutes while the commercial software GUROBI (Bixby et al. 2010) runs on the order of hours. Additionally, on one particular feasible knapsack instance, the low-memory implementation of the CG algorithm finds an optimal solution in under an hour, while GUROBI runs for several hours, consumes 4 GB of RAM and 2 GB of swap space,

before being killed remotely by the system. Finally, we demonstrate that the CG algorithm runs almost twice as fast as GUROBI on a particular class of square, infeasible set partition instances.

For future work, we intend to continue optimizing the low-memory implementation and researching faster methods of calculating the intersection  $S_X$ . Additionally, searching for problems that are hard for GUROBI and yet have low enough branch-width to be practical with the CG algorithm will be an active area of interest. Finally, the CG algorithm readily lends itself to parallelization in a way that the simplex algorithm does not. This will be the first priority for our next investigation.

### Acknowledgments

The authors thank Mark Embree for his support and feedback on this project, and also Jon Lee for facilitating the use of the IBM Yellowzone machines. We are also deeply indebted to the three anonymous reviewers for their prompt and thorough reviews. Their suggestions greatly improved the content and presentation of the paper. Finally, we acknowledge the support of the National Science Foundation [DMS-0729251], [NSF-CMMI-0926618], and [DMS-0240058].

### References

- Aardal K, Bixby RE, Hurkens CAJ, Lenstra AK, Smeltink JW (1999) Market split and basis reduction: Towards a solution of the Cornuéjols-Dawande instances. Cornuéjols G, Burkard R, Woeginger G, eds. *Integer Programming and Combinatorial Optimization*, Lecture Notes Computer Science, Vol. 1610 (Springer, Berlin), 1–16.
- Arnborg S, Lagergren J, Seese D (1991) Easy problems for tree-decomposable graphs. *J. Algorithms* 12(2):308–340.
- Bixby R, Gu Z, Rothberg E (2010) Gurobi optimization. Accessed April 2011, <http://gurobi.com>.
- Cook W, Seymour P (1994) An algorithm for the ring-routing problem. Technical report, Bellcore Technical Memorandum.
- Cook W, Seymour P (2003) Tour merging via branch-decomposition. *INFORMS J. Comput.* 15(3):233–248.
- Courcelle B (1990) The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Inform. Comput.* 85(1):12–75.
- Cunningham WH, Geelen J (2007) On integer programming and the branch-width of the constraint matrix. Fischetti M, Williamson D, eds. *Integer Programming and Combinatorial Optimization*, Lecture Notes Computer Science, Vol. 4513 (Springer, Berlin), 158–166.
- Golub G, Van Loan C (1996) *Matrix Computations*, 3rd ed. (Johns Hopkins University Press, Baltimore).
- Hicks IV (2004) Branch decompositions and minor containment. *Networks* 43(1):1–9.
- Hicks IV (2005) Graphs, branchwidth, and tangles! Oh my! *Networks* 45(2):55–60.
- Hicks IV, McMurray N (2007) The branchwidth of graphs and their cycle matroids. *J. Combin. Theory Ser. B* 97(5):681–692.
- Ma J, Margulies S, Hicks IV, Goins E (2012) Branch-decomposition heuristics for linear matroids. Unpublished manuscript.
- Mittelman H (2012) Mixed integer linear programming benchmark (parallel codes). Accessed April 2012, <http://plato.asu.edu/ftp/milpc.html>.
- Mizuno K, Nishihara S (2008) Constructive generation of very hard 3-colorability instances. *Discrete Appl. Math.* 156(2):218–229.
- Pataki G, Krishnamoorthy B (2009) Column basis reduction and decomposable knapsack problems. *Discrete Optim.* 6(3):242–270.
- Pataki G, Tural M, Wong EB (2010) Basis reduction and the complexity of branch-and-bound. *SODA '10 Proc. Twenty-First Annual ACM-SIAM Sympos. Discrete Algorithms* (Society for Industrial and Applied Mathematics, Philadelphia), 1254–1261.
- Seymour PD, Thomas R (1994) Call routing and the ratcatcher. *Combinatorica* 14(2):217–241.