

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

S. Margulies

Department of Computational and Applied Math, Rice University, Houston, Texas,
{susan.margulies@rice.edu}

J. Ma

Department of Management Science and Engineering, Stanford University, Palo Alto, California,
{jm11.rice@gmail.com}

I.V. Hicks

Department of Computational and Applied Math, Rice University, Houston, Texas,
{ivhicks@rice.edu}

Cunningham and Geelen [7] describe an algorithm for solving the integer program $\max\{c^T x : Ax = b, x \geq 0, x \in \mathbb{Z}^n\}$, where $A \in \mathbb{Z}_{\geq 0}^{m \times n}$, $b \in \mathbb{Z}^m$, 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 [3]. Using branch-decomposition trees produced by the heuristics developed by Ma et. al [12], and optimal trees produced by the algorithm designed by Hicks [10], 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 knapsacks instances with width ≤ 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: submitted April 2011.

1. Introduction

The burgeoning area of branch-decomposition-based algorithms has expanded to include problems as diverse as ring-routing [4], travelling salesman [5] and general minor containment [9]. 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 [6, 2], 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 various and 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) algorithm [7] is that it provides a first link between integer programming and the long list of research areas augmented by branch-decompositions. The question is, how significant will this link between branch-decompositions and integer programming be? How well does the CG algorithm work in practice? Are there classes of problems upon which is it efficient? Is it easy to implement? How does it compare to commercial software? These are the questions explored in this paper.

We begin in Section 2 by recalling the relevant background and definitions (such as branch-decomposition, branch-width, T -branched sets, etc.). In Section 3, we describe the CG algorithm in detail, including a discussion of internal data-structures and runtime, and highlight the relevance of a particular vector space intersection to the workings of the algorithm. In Section 4, we describe three different methods for calculating this particular intersection, and include experimental results for a comparison of the three methods. In Section 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 [3]. We conclude in Section 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, ν) , where T is a cubic tree (all interior nodes have degree three), and ν is a map from the columns of A to the leaves of T . The edges of T are weighted via a *connectivity function* λ_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 $\lambda_A(X_1) + \lambda_A(X_2) \geq \lambda(X_1 \cap X_2) + \lambda(X_1 \cup X_2)$ for all $X_1, X_2 \subseteq E$). Finally, we define the *width* of a branch-decomposition (T, ν) 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, ν) of A , and thus, the

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

Example 2.1 For example, consider the following matrix A and branch-decomposition (T, ν) :

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 1 & 0 \\ 3 & 1 & 2 & 3 \\ 4 & 2 & 1 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \quad \begin{array}{c} \bullet \\ \text{1} \\ \bullet \\ \text{2} \\ \bullet \\ \text{3} \\ \bullet \\ \text{4} \end{array}$$

In this example, we note that the weight of edge (v_1, v_2) is equal to one. This is because $T - (v_1, v_2)$ disconnects T into two components, the first labeled with the columns $X = \{1, 2\}$ and the second labelled with the columns $E - X = \{3, 4\}$. Thus,

$$\text{weight}(v_1, v_2) = \lambda_A(\{1, 2\}) = \text{rank}(A|\{1, 2\}) + \text{rank}(A|\{3, 4\}) - \text{rank}(A) + 1 = 1 .$$

We see that the maximum over all the edge weights in 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 . \square

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

Given a non-negative 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 non-negative matrix $A \in \mathbb{Z}^{m \times n}$, 2) a non-negative vector $b \in \mathbb{Z}^m$, 3) the objective function $c \in \mathbb{Z}^n$, and 4) a branch-decomposition (T, ν) of A with width k , and returns as **output** the optimal $x \in \mathbb{Z}^n$ which maximizes $c^T x$. The algorithm solves this maximization problem in $O((d+1)^{2k} mn + m^2 n)$ 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 a dynamic programming technique. 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, ν) , 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 $T - e$ (for example, in Example 2.1, we see that $X = \{1, 2\}$ is a *T-branched* set). Any *T-branched* set X where $|X| \geq 2$ can be partitioned into two smaller *T-branched* sets (X_1, X_2) . In this case, for $X = (X_1, X_2)$,

$$\begin{aligned} \mathcal{B}((X_1, X_2)) = \{ b' \in \mathbb{Z}^m : & \text{ i) } 0 \leq b' \leq b, \\ & \text{ ii) } \exists b'_1 \in \mathcal{B}(X_1) \text{ and } b'_2 \in \mathcal{B}(X_2) \text{ such that } b' = b'_1 + b'_2, \text{ and} \\ & \text{ iii) } b' \in \text{span}(A'|X) \cap \text{span}(A'|(E' - X)) \} . \end{aligned}$$

Having defined *T-branched* sets and $\mathcal{B}(X)$, 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). We then walk the nodes of the tree in post-depth-first-search order. Since the tree T is cubic, when considered in post-depth-first-search order, every internal node has two children and a parent. Thus, every *T-branched* set X where $|X| \geq 2$ can be easily partitioned into two *T-branched* sets X_1

```

*****
Algorithm:   Cunningham-Geelen Algorithm
Input:      1) A non-negative matrix  $A \in \mathbb{Z}^{m \times n}$ ,
                2) A non-negative 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:     1) A vector  $x \in \mathbb{Z}^n$  such that  $c^T x$  is maximized.
1 for each  $T$ -branched set  $X \subseteq E$  do
2   if  $|X| = 1$  then
3     Calculate  $\mathcal{B}(X)$ 
4   else
5     Partition  $X$  into two smaller  $T$ -branched sets  $(X_1, X_2)$ 
6      $\mathcal{B}(X) \leftarrow \emptyset$ 
7     for each  $b'_1 \in \mathcal{B}(X_1)$  and  $b'_2 \in \mathcal{B}(X_2)$  do
8        $b' \leftarrow b'_1 + b'_2$ 
9       if  $b' \leq b$  and  $b' \in S(A', X)$  then
10         $\mathcal{B}(X) \leftarrow \mathcal{B}(X) \cup b'$ 
11       end if
12     end for
13   end if
14 end for
15 for all  $b \in \mathcal{B}(E)$ 
16   Find the maximum of  $c^T x$ 
17 end for
18 return optimal  $x$ 
*****

```

Figure 1: The pseudocode for the CG algorithm.

and X_2 corresponding to the two children. Then, we simply take linear combinations of the vectors in X_1 and X_2 , 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 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 will be greatly affected by the method used in determining whether or not 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 above, 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 , and 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 (with a few notational errors, and without a rigorous proof) in [7], the second is a well-known numerical algorithm from [8] for finding the intersection of two subspaces, and the third is a straight-forward application of properties of $\mathcal{B}(X)$. In this section, we describe each of these methods, and then display a computational comparison.

The CG Intersection Method

The first algorithm for determining whether a given vector $v \in S_X$ is from [7] (with corrections). 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:

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

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 rows in each, denote the matrix $[V \ W]$ as the matrix consisting of all the columns and rows in both V and W . We can reorder the columns of A_X and A_Y such that $\overline{A_X} := [A|(B \cap X) \ A|(X - B)]$ and $\overline{A_Y} := [A|(B \cap Y) \ A|(Y - B)]$. In the case of A and X given as above,

$$\overline{A_X} = \begin{bmatrix} 0 & 0 & 7 & 6 \\ 1 & 0 & 7 & 1 \\ 0 & 0 & 2 & 1 \\ 0 & 1 & 6 & 4 \end{bmatrix} \text{ and } \overline{A_Y} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 4 \\ 0 & 1 & 4 \\ 0 & 0 & 6 \end{bmatrix}.$$

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 $\text{red}(\overline{A_X})$ and $\text{red}(\overline{A_Y})$ by cancelling the entries in the columns $X - B$ (or $Y - B$) that correspond to rows with non-zero 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,

$$\text{red}(\overline{A_X}) = \begin{bmatrix} 0 & 0 & 7 & 6 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 2 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix} \text{ and } \text{red}(\overline{A_Y}) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 4 \\ 0 & 1 & 0 \\ 0 & 0 & 6 \end{bmatrix}.$$

We claim that $M_X = [\text{red}(\overline{A_X})|(X - B) \quad \text{red}(\overline{A_Y})|(Y - B)]$. In our running example,

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

Notice that the $\dim(S_X) = 3$ and $\text{rank}(M_X) = 3$ in our example. The following lemma is a correction of claims in Section 3 of [7]. Following the notation of [7], 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 4.1 *For any partition (X, Y) of E ,*

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

Moreover, S_X is the column-span of the matrix

$$M_X = [\text{red}(\overline{A_X})|(X - B) \quad \text{red}(\overline{A_Y})|(Y - B)] .$$

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 $v \in \text{span}(\text{red}(\overline{A_X}))$, and likewise, $v \in \text{span}(A|Y)$ and $v \in \text{span}(\text{red}(\overline{A_Y}))$. Thus, v can be written as a linear combination of columns in $\text{red}(\overline{A_X})$ (i.e., $v = \text{red}(\overline{A_X})\lambda$), and v can also be written as a linear combination of the columns in $\text{red}(\overline{A_Y})$ (i.e., $v = \text{red}(\overline{A_Y})\lambda'$).

We will construct a $(|X - B| + |Y - B|)$ -length vector λ'' using entries from both λ and λ' . Set the first $|X - B|$ entries in λ'' to the last $|X - B|$ entries in λ , and the second $|Y - B|$ entries in λ'' to the last $|Y - B|$ entries in λ' . Then, $v = M_X\lambda''$, 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 $v = M_X\lambda''$ for some λ'' . We will construct corresponding λ, λ' such that $v = \text{red}(\overline{A_X})\lambda = \text{red}(\overline{A_Y})\lambda'$. Towards that end, set the first $|X \cap B|$ entries in λ to the first $|X \cap B|$ entries in v , and the last $|X - B|$ entries in λ to the first $|X - B|$ entries in λ'' . To construct λ' , set the first $|Y \cap B|$ entries in λ' to the first $|Y \cap B|$ entries in v , and the last $|Y - B|$ entries in λ' to the last $|Y - B|$ entries in λ'' . Then, $v = \text{red}(\overline{A_X})\lambda = \text{red}(\overline{A_Y})\lambda'$ as desired, and $\text{span}(M_X) \subseteq S_X$.

Finally, we must show that

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

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

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

When we inspect the matrix M_X (a basis for S_X), we see that any two vectors from the first $(X - B)$ columns and 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, when the rows of zeros are removed, we see that the $\text{rank}(M_X|(X - B)) = \text{rank}(A[B - (X \cap B), X - B])$ and $\text{rank}(M_X|(Y - B)) = \text{rank}(A[B - (Y \cap B), Y - B])$. 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, whenever we wish to 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 . Finally, we note that the runtime analysis of the CG algorithm assumes that 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 [8], Section 12.4.4 (pg. 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 $Q_A^T Q_B^T$. 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, which is on the same order as the runtime of the CG intersection method $O(m^2 n \log m)$.

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 $b' \in \text{span}(A_Y)$. 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

We test these three intersection algorithms on randomly generated set partition instances (see Section 5.3.2 for a complete description). 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 $\text{span}(Y)$).

<i>rows</i>	<i>columns</i>	CG	QR	$\text{span}(Y)$
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	2360	915	85
300	500	2381	1508	185
400	500	237	287	114
500	500	3	320	67
150	750	3565	1132	169
300	750	–	–	513
450	750	–	–	1057
600	750	–	–	621
750	750	9	–	403
200	1000	–	–	577
400	1000	–	–	1723
600	1000	–	–	3485

Table 1: Computational investigations on intersection algorithms.

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. For example, on the 750×750 instance, the CG method finishes in 9 secs, while the QR method does not terminate, and the $\text{span}(Y)$ method takes 403 secs. In Section 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 or five 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 Section 4). In Section 5.1, we describe the nuances of our implementation. In Section 5.2, we describe the different types of trees we used as input. In Section 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 ≤ 6 . For a particularly demonstrative example, the CG algorithm runs the infeasible knapsack instance `ex6_Original_10000.lp` in $1188 \approx 19.8$ min while GUROBI runs in $13464 \approx 3.74$ 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. Towards 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 α 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 $\alpha v_1 + \beta v_2 \notin \text{span}(W)$.
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 4×30

market split instance (see Section 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 .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 450×588 infeasible graph 3-coloring instance (see Section 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:

$$\text{hash}(v) = \sum_{i=1}^m (i + 65599)v_i .$$

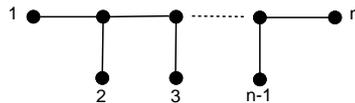
Obviously, two different vectors can “hash” to the same value. In this case, we must do a direct component-by-component comparison to determine if the given combination does indeed sum to the vector b . However, the “hash” function often spares us the expensive direct comparison. Furthermore, the “hash” 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 .

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, ν) 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 in [10] with code provided by the author. However, we were unable to obtain trees for many of our larger test cases, which is expected since finding an optimal branch-decomposition is NP-hard [16, 11].

The heuristically-derived trees were computed via the algorithms described in [12] with code provided by the authors. In [12], 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. 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, surprising 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 dual-Core AMD Opteron processor

with 3 GZ clock speed, 4 GB of RAM and 2 GB of swap space. We tested our code against GUROBI [3], which is well-known commercial software for solving integer programs (GUROBI is considered competitive with CPLEX on integer programming performance benchmarks [13]).

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.

In [14], the authors 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 in [14], the authors 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

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

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.

We tested these instances with the memory-intensive implementation, 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 131×171 instance, although the width of both trees is 42, the heuristic tree

<i>rows</i>	<i>cols</i>	<i>max(b)</i>	<i>cat width</i>	<i>sec</i>	<i>heuristic width</i>	<i>sec</i>	<small>GUROBI</small> <i>sec</i>
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	–	–	–	–	1873
772	1008	1	–	–	–	–	9676
836	1092	1	–	–	–	–	16033

Table 2: Infeasible graph 3-coloring instances.

runs in 2 seconds, but the caterpillar tree *does not terminate*. We provide an explanation for this discrepancy in Section 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 $Ax = \mathbf{1}$, 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 = double(rand(m, n) > .20)`.

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 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 Section 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

<i>rows</i>	<i>cols</i>	cat width	<i>sec</i>	heuristic width	<i>sec</i>	GUROBI <i>sec</i>
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	1133	151	85	16
300	750	301	7094	299	317	16
450	750	302	5846	302	735	3
600	750	102	1168	152	842	6
750	750	2	1547	2	1104	7

Table 3: Randomly-generated infeasible set partition instances.

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.

<i>rows</i>	<i>cols</i>	cat width	<i>sec</i>	GUROBI <i>sec</i>
250	250	2	0	2
1000	1000	2	23	46
1500	1500	2	80	154
2000	2000	2	190	570
2500	2500	2	372	1047
3000	3000	2	858	2024
3500	3500	2	1063	3226
4000	4000	2	1528	4900
4500	4500	2	2588	6960
5000	5000	2	3378	9589
5500	5500	2	5208	12626

Table 4: Randomly-generated square infeasible set partition instances.

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 ≤ 6 .

The following instances are the Cornuéjols-Dawande *market split* problems (see [1] 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.

<i>rows</i>	<i>cols</i>	<i>max(b)</i>	<i>cat width</i>	<i>sec</i>	<i>heuristic width</i>	<i>sec</i>	<i>opt width</i>	<i>sec</i>	^{GUROBI} <i>sec</i>
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	1067	6	1885	6	1900	6	*	2235
6	50	1264	7	–	7	–	7	–	5631

Table 5: Infeasible market split instances.

In this case, we see that the CG method is significantly faster than GUROBI on instances with relatively small widths (≤ 6), but that GUROBI scales better with instance size. For example, while the CG method runs twice as fast as GUROBI on the market split 5×40 instance, on the 6×50 instance, the CG method ran overnight without terminating while GUROBI simply doubled in time. We note that we were unable to obtain an optimal tree for the 5×40 instance, which is why a “*” appears, rather than the customary non-termination “_”.

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”. In [15], the authors describe hard, infeasible knapsack instances (posted online at <http://www.unc.edu/~pataki/instances/marketshare.htm>). In these infeasible knapsack instances, the

matrices A are 5×40 , which implies that these are multiple-knapsack problems, i.e., the goal is to use 40 items to pack 5 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 5×40 Pataki matrices, which allows these instances to become feasible. The computational results from these constructed feasible knapsack instances are described below in Table 6.

<i>name</i>	<i>rows</i>	<i>cols</i>	$\max(b)$	row set	<i>cat width</i>	<i>sec</i>	GUROBI SEC
ex1_Original_10000.lp	2	40	101,368	{1, 2}	3	2792	534
ex2_Original_10000.lp	2	40	110,947	{1, 2}	3	1676	144
ex3_Original_10000.lp	2	40	119,606	{1, 4}	3	1658	< 15848
ex4_Original_10000.lp	2	40	95,708	{1, 4}	3	2599	1491
ex5_Original_10000.lp	2	40	104,334	{1, 2}	3	2207	1321

Table 6: Feasible knapsack instances created from infeasible instances in [15].

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 3325 sec \approx 55 min, we could not run GUROBI to termination on our machine. Indeed, after over 15848 sec \approx 4.5 hours, GUROBI used up 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 (≤ 6), the CG method is significantly faster than the more traditional integer programming methods of GUROBI.

<i>name</i>	<i>rows</i>	<i>cols</i>	<i>max(b)</i>	<i>heuristic width</i>	<i>CG sec</i>	<i>GUROBI sec</i>
ex1_Original_10000.lp	5	40	112,648	6	534 \approx 8.9 min	4969 \approx 1.38 hour
ex2_Original_10000.lp	5	40	110,947	6	1681 \approx 28 min	11257 \approx 3.13 hour
ex3_Original_10000.lp	5	40	119,606	6	343 \approx 5.7 min	4122 \approx 1.15 hour
ex4_Original_10000.lp	5	40	116,946	6	1880 \approx 31.3 min	10388 \approx 2.89 hour
ex5_Original_10000.lp	5	40	104,334	6	2644 \approx 44 min	4570 \approx 1.23 hour
ex6_Original_10000.lp	5	40	108,565	6	1188 \approx 19.8 min	13464 \approx 3.74 hour
ex7_Original_10000.lp	5	40	105,870	6	1340 \approx 23 min	4819 \approx 1.34 hour
ex8_Original_10000.lp	5	40	106,495	6	925 \approx 15 min	10085 \approx 2.8 hour
ex9_Original_10000.lp	5	40	112,366	6	1969 \approx 32 min	11584 \approx 3.22 hour
ex10_Original_10000.lp	5	40	102,170	6	2189 \approx 36 min	6140 \approx 1.71 hour

Table 7: Infeasible knapsack instances from [15].

5.4 Branch-decompositions and Edge-weight Dispersion

In Sections 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, although the width of the heuristic and caterpillar trees was virtually identical, the runtime of the CG algorithm on the heuristic trees was significantly faster than the runtime on the caterpillar trees.

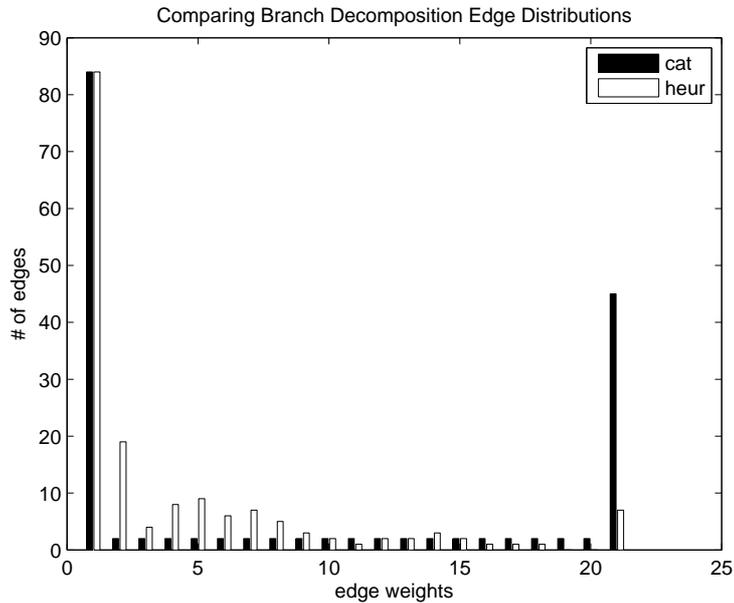


Figure 2: Distribution of edge weights between caterpillar and heuristic trees.

In Figure 2, we compare the edge weight distribution of a caterpillar and a heuristic tree on the first MUG instance (the matrix has size 64×84 , and 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 7 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 2 edges with weight 2. Thus, we can see 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. This observation should encourage research into branch-decomposition heuristics that emphasize equal focus on both lower widths and less 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 ≤ 6 , the CG algorithm runs in minutes while the commercial software GUROBI [3] 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 algorithms 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.

Acknowledgements

The authors would like to thank Mark Embree for his support and feedback on this project, and also Jon Lee for facilitating use of the IBM Yellowzone machines. Finally, we acknowledge the support of NSF DMS-0729251, NSF-CMMI-0926618 and DMS-0240058.

References

- [1] K. Aardal, R. E. Bixby, C. A. J. Hurkens, A. K. Lenstra, and J. W. Smeltink. Market split and basis reduction: Towards a solution of the Cornuéjols-Dawande instances. *Lecture Notes in Computer Science*, 1610:1–16, 1999.
- [2] S. Arnborg, J. Lagergren, and D. Seese. Easy problems for tree-decomposable graphs. *Journal of Algorithms*, 12:308–340, 1991.
- [3] R. Bixby, Z. Gu, and E. Rothberg. Gurobi optimization. Available at <http://gurobi.com>.
- [4] W. Cook and P.D. Seymour. An algorithm for the ring-routing problem. *Bellcore technical memorandum*, 1994.
- [5] W. Cook and P.D. Seymour. Tour merging via branch-decomposition. *INFORMS Journal on Computing*, 15(3):233–248, 2003.
- [6] B. Courcelle. The monadic second-order logic of graphs I: Recognizable sets of finite graphs. *Information and Computation*, 85:12–75, 1990.
- [7] W.H. Cunningham and J. Geelen. On integer programming and the branch-width of the constraint matrix. *Lecture Notes in Computer Science*, 4513:158–166, 2007.
- [8] G. Golub and C. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore and London, 3rd edition, 1996.
- [9] I.V. Hicks. Branch decompositions and minor containment. *Networks*, 43:1–9, 2004.
- [10] I.V. Hicks. Graphs, branchwidth, and tangles! Oh my! *Networks*, 45:55–60, 2005.
- [11] I.V. Hicks and N. McMurray. The branchwidth of graphs and their cycle matroids. *Journal of Combinatorial Theory Series B*, 97:681–692, 2007.
- [12] J. Ma, S. Margulies, I.V. Hicks, and E. Goins. Branch-decomposition heuristics for linear matroids. Manuscript in preparation.
- [13] H. Mittelmann. Mixed integer linear programming benchmark (parallel codes). Available at <http://plato.asu.edu/ftp/milpc.html>.

- [14] K. Mizuno and S. Nishihara. Constructive generation of very hard 3-colorability instances. *Discrete Applied Mathematics*, 156(2):218–229, 2008.
- [15] G. Pataki, M. Tural, and E. B. Wong. Basis reduction and the complexity of branch-and-bound. In *SODA '10 Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA, 2010.
- [16] P.D. Seymour and R. Thomas. Call routing and the ratcatcher. *Combinatorica*, 14(2):217–241, 1994.