ORNL/TM-2012/176

INDDGO: Integrated Network Decomposition & Dynamic programming for Graph Optimization

October 2012

Chris Groër[1]
Blair D. Sullivan[1]
Dinesh Weerapurage[1]

[1]This work was supported by the Department of Energy Office of Science, Office of Advanced Scientific Computing Research

ORNL-2012/176

INDDGO: Integrated Network Decomposition & Dynamic programming for Graph Optimization

Chris Groër
Blair D. Sullivan
Dinesh Weerapurage

Date Published: October 2012

Prepared by
OAK RIDGE NATIONAL LABORATORY
P. O. Box 2008
Oak Ridge, Tennessee 37831-6285
managed by
UT-Battelle, LLC
for the
U. S. DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

Contents

List of Figures

Abstract

It is well-known that dynamic programming algorithms can utilize tree decompositions to provide a way to solve some *NP*-hard graph optimization problems where the complexity is polynomial in the number of nodes and edges in the graph, but exponential in the width of the underlying tree decomposition. However, there has been relatively little computational work done to determine the practical utility of such dynamic programming algorithms. We have developed software to construct tree decompositions using various heuristics and have created a fast, memory-efficient dynamic programming implementation for solving maximum weighted independent set. We describe our software and the algorithms we have implemented, focusing on memory saving techniques for the dynamic programming. We compare the running time and memory usage of our implementation with other techniques for solving maximum weighted independent set, including a commercial integer programming solver and a semi-definite programming solver. Our results indicate that it is possible to solve some instances where the underlying decomposition has width much larger than suggested by the literature. For certain types of problems, our dynamic programming code runs several times faster than these other methods.

1. Introduction

Tree decompositions were introduced by Robertson and Seymour in 1984 in one of their papers on structural graph theory [22]. These decompositions provide a combinatorial metric for the "distance" from a graph to a tree, known as the width of a decomposition. The minimal achievable width for a graph is its treewidth, and can be thought of as a measure of how "tree-like" a graph is. Although tree decompositions were introduced as tools for proving the Graph Minors Theorem [21], these mappings have gained importance in computational graph theory, as they allow numerous *NP*-hard graph problems to be solved in polynomial time for graphs with bounded treewidth [11]. We begin by giving some necessary definitions, then proceed to describe how to compute and use these decompositions in algorithms.

Formally, a *tree decomposition* of a graph $G = (V, E)$ is a pair (X, T), where $X = \{X_1, \ldots, X_n\}$ is a collection of subsets of V, and $T = (I, F)$ is a tree with $I = \{1, \ldots, n\}$, satisfying three conditions:

1. the union of the subsets X_i is equal to the vertex set V $(1 \leq i \leq n)$,
2. for every edge uv in G, $\{u, v\} \subseteq X_i$ for some $i \in \{1, 2, \ldots, n\}$, and
3. for every $v \in V$, if X_i and X_j contain v for some $i, j \in \{1, 2, \ldots, n\}$, then X_k also contains v for all k on the (unique) path in T connecting i and j. In other words, the set of nodes whose subsets contain v form a connected sub-tree of T.

The subsets X_i are often referred to as *bags* of vertices. The *width* of a tree decomposition $(\{X_1, \ldots, X_n\}, T)$ is the maximum over $i \in \{1, 2, \ldots, n\}$ of $|X_i| - 1$, and the *treewidth* of a graph G, denoted $\tau(G)$, is the minimum width over all tree decompositions of G. The negative one in the definition is purely cosmetic, and was chosen so that trees (and more generally, forests) have treewidth one. An *optimal tree decomposition* for a graph G is a decomposition (X, T) with width $\tau(G)$.

2. Constructing Tree Decompositions

Our primary interest in tree decompositions is to determine their practical utility for solving discrete optimization problems via dynamic programming. Dynamic programming recursions that exploit tree decompositions often require some kind of exhaustive search at each tree node, and this search is typically exponential in the size of the bags. Thus, it is very important for us

to be able to generate decompositions with as small a width as possible. Results of Seymour and Thomas [25] show that finding optimal decompositions is *NP*-hard, so we resort to heuristic methods for generating decompositions of "low" width.

Here we give a very brief overview of existing algorithms - a more comprehensive survey can be found in [16]. The algorithms for finding low-width tree decompositions are generally divided into two classes - "theoretical" and "computational." The former category includes, for example, the linear algorithm of Bodlaender [8] which checks if a tree decomposition of width at most k exists (for a fixed constant k), and the approximation algorithms of Amir [3]. These are generally considered computationally intractible due to very large hidden constants and complexity of implementation - e.g. Bodlaender's algorithm was shown by Röhrig [23] to have too high a computational cost even when $k = 4$. The approximation algorithms of Amir have been tested on graphs with up to 570 nodes, but require several hours of running time.

Most computational work has been done utilizing heuristics which offer no guarantee on their maximum deviation from optimality. One of the most common methods for constructing tree decompositions is based on known algorithms for decomposing *chordal graphs*, which are characterized by having no induced cycles of length greater than three. An *elimination ordering* is a permutation of the vertex set of a graph, commonly used to guide the addition of edges to make the graph chordal, a process known as *triangulation*. A valid tree decomposition for the triangulated graph can be formed with bags consisting of the sets of higher numbered neighbors for each vertex in the elimination ordering. Since a tree decomposition for a graph is valid for all subgraphs on the same vertex set, this simultaneously yields a decomposition for the original graph. The specifics of these kinds of procedures are well-known in the literature and we provide pseudocode of our implementations of two algorithms that create tree decompositions for a graph given an ordering of its vertices. We require a function $\textsc{GetNeighbors}(G, v, W)$ that returns the neighbors of vertex v in the graph that are also contained in the set of vertices W. Algorithm 1 describes the elimination of a vertex from a graph, and Algorithm 2 describes the details of our implementation of the tree decomposition construction algorithm outlined in [12]. We describe our implementation of the procedure given in [9] in Algorithm 3.

Algorithm 1 Eliminate a vertex v from G

procedure ELIMINATE(G,v,W,del)▷ Graph $G = (V, E)$, $v \in V$, $W \subseteq V$
    Eliminate the vertex v from G and optionally delete it from G if del==`true`
    Set $N(v)$ = GETNEIGHBORS(G,v,W)
    for $u, w \in N(v)$ so that $u \neq w$ **do**
        $E = E \cup \{u, w\}$ ▷ Add the edge $\{u, w\}$ to the graph G
    end for
    if del==`true` **then**
        Delete the vertex v from the graph G
    end if
end procedure

Algorithm 2 Construct a tree decomposition using Gavril's algorithm

procedure GAVRIL(G,π) ▷ Graph $G = (V, E)$, π a permutation of V
    Constructs a tree decomposition $T = (X, (I, F))$ with X_i the bag of vertices for tree node $i \in I$ and (I, F) a tree
    Initialize $T = (X, (I, F))$ with $X = I = F = \emptyset$ and $k = 0$.
    $H = G$; $n = |V|$
    for $i = 1$ to $n - 1$ **do** ▷ Triangulate G
        H=ELIMINATE(H,π_i, $\{\pi_{i+1}, \ldots, \pi_n\}$, `false`)
    end for
    $k = 1$, $I = \{1\}$, $X_1 = \{\pi_n\}$, $t[\pi_n] = 1$ ▷ $t[]$ is an n-long array
    for $i = n - 1$ to 1 **do** ▷ Iterate backwards through π to construct T
        B_i = GETNEIGHBORS($H, \pi_i, \{\pi_{i+1}, \ldots, \pi_n\}$)
        Find $m = j$ such that $j \leq k$ for all $\pi_k \in B_i$
        if $B_i = X_{t[m]}$ **then**
            $X_{t[m]} = X_{t[m]} \cup \{\pi_i\}$
            $t[\pi_i] = t[m]$
        else
            $k = k + 1$
            $I = I \cup \{k\}$; $X_k = B_i \cup \{\pi_i\}$; $F = F \cup (k, t[m])$ ▷ Update T
            $t[\pi_i] = k$
        end if
    end for
    return $T = (X, (I, F))$
end procedure

Algorithm 3 Construct a tree decomposition using the Bodlaender-Koster algorithm

1: **procedure** BK(G,π) ▷ Graph $G = (V, E)$, π a permutation of V
2: Constructs a tree decomposition $T = (X, (I, F))$ with X_i the bag of vertices for tree node $i \in I$ and (I, F) a tree
3: $H = G$; $n = |V|$
4: Initialize $T = (X, (I, F))$ with $I = \{1, 2, \ldots, n\}$ and $X_i = \emptyset$ for all $i \in I$
5: **for** $i = 1$ to $n - 1$ **do** ▷ Triangulate G
6: H=ELIMINATE(H,π_i, $\{\pi_{i+1}, \ldots, \pi_n\}$, `false`)
7: **end for**
8: **for** $i = 0$ to $n - 1$ **do** ▷ Iterate through π to construct T
9: $B_i = \text{GETNEIGHBORS}(H, \pi_i, \{\pi_{i+1}, \ldots, \pi_n\})$
10: $X_{\pi_i} = \{\pi_i\} \cup B_i$ ▷ Construct the bag for tree node i
11: Find $m = \pi_j$ such that $j \leq k$ for all $\pi_k \in B_i$
12: $F = F \cup \{i, m\}$ ▷ Add the edge $(i, m\}$ to the tree
13: ELIMINATE(H,π_i, B_i, `false`) ▷ Add edges among π_i's forward neighbors
14: **end for**
15: **return** $T = (X, (I, F))$
16: **end procedure**

2.1 Elimination Ordering Heuristics

As our primary goal is to quickly generate tree decompositions of low width, we implemented a number of commonly used heuristics from the literature for generating elimination orderings. Since our purpose was to utilize and implement established heuristics, we do not describe the inner workings of each heuristic, but instead provide a reference describing each heuristic, and how to invoke it with our software.

All of the elimination ordering heuristics we implemented are available via a call to the function `find_elimination_ordering` which places the ordering in a user-provided location. The user can optionally provide a starting vertex for the heuristic and can also specify whether or not to add the edges produced during triangulation to the input graph.

We ran our algorithm on a set of more than 500 graphs generously provided to us by Hans Bodlaender. These are graphs that were previously available on the *TreewidthLIB* site [7]. We restrict our computational results to only those 248 graphs containing 100 nodes or more. We ran each heuristic on each of these graphs with a random starting vertex and a time

Heuristic [Reference]	Avg. Width	Best Width Rank	Worst Width Rank	Avg. Width Rank	Best Time Rank	Worst Time Rank	Time Rank	Avg. Time
MinDegree [13]	72.3	1	9	3.94	1	6	3.42	0.17
MCS [5]	78.95	1	10	7.38	4	10	6.81	205.1
MCS-M [5]	105.11	1	10	8.41	1	10	8.35	186.70
LEX-M [24]	99.4	1	10	8.37	1	7	9.52	232.8
MinFill [14]	66.7	1	7	1.53	4	10	7.83	21.4
MetisMMD [18, 20]	69.75	1	9	3.04	1	8	2.23	0.04
MetisNodeND [18]	71.91	1	10	5.12	1	8	2.56	0.05
AMD [2]	72.1	1	10	3.9	1	7	1.85	0.03
MMD [20]	72	1	10	4.07	1	6	3.93	0.18
MinMaxDegree	72	1	10	6.83	3	10	6.84	28.6

Table 1: A comparison of the performance of elimination ordering heuristics on a set of test graphs

limit of 7200 seconds. For each run, we recorded the running time (in seconds) and the width of the resulting tree decomposition. Table 1 presents the results of these computational experiments. For each heuristic, we provide a reference from the literature and its performance in terms of both width and running time. The average width and running times are computed across all 248 decompositions. For the width and running time rankings, we provide the best, worst, and average ranking of each heuristic in terms of both width and running time where a ranking of one indicates the heuristic is the top performer and a rank of ten means it is the worst. As an example of how to interpret the results, in terms of width, the *MinDegree* heuristic was the top-ranked heuristic on at least one problem, it was ranked as low as ninth on one instance, and its average width ranking was 3.94.

We can view the relative performance of each heuristic on this set of benchmark graphs by plotting the average width and running time rankings in the width-time plane. In Figure 1, we see that *Greedy Minimum Fill, Metis MMD,* and *Approximate Minimum Degree* (AMD) are not dominated by any other heuristic in either dimension. If one requires a decomposition with as small a width as possible, then *Greedy Minimum Fill* is a good choice. For larger graphs where running time becomes an issue, then the best choices are probably *Metis MMD* or *Approximate Minimum Degree.* While the slower, more complex heuristics such as *LEXM* performed poorly on many of the problems, we have seen cases where they dramatically outperforms all the other heuristics. For example, on the problem *1dc.256* from the DIMACS Independent Set Challenge [26], the *LEXM* heuristic produces a decomposition with width 78. However, *Greedy Minimum Fill, Metis MMD*, and *Approximate Minimum Degree* produce widths of 119, 132, and 128, respectively. Thus, while *Greedy Minimum Fill, Metis MMD,* and *Approximate Minimum Degree* are superior on most instances, exceptions exist where the other heuristics produce decompositions with much lower

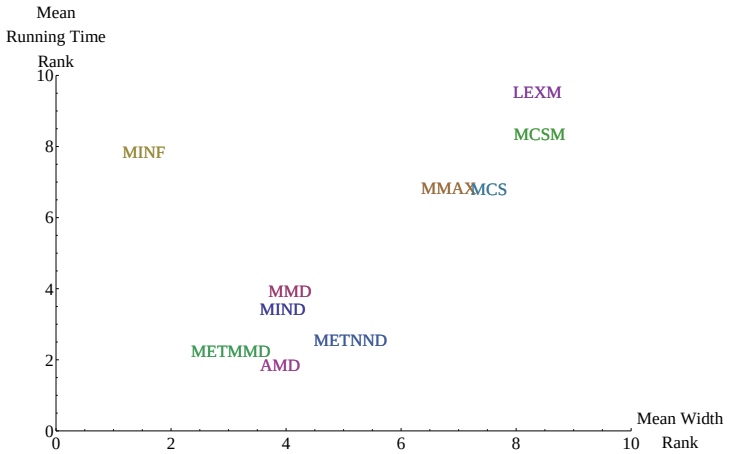

Figure 1: A comparison of the average width and average running time of various heuristics on a large set of benchmark problems

widths.

3. Solving Maximum Weighted Independent Set

Having described how to construct tree decompositions, we now describe how we use these decompositions as part of a dynamic programming algorithm to solve maximum weighted independent set (MWIS). The dynamic programming recursion for using tree decompositions to solve MWIS is well-known [6, 9]. The general idea is to root the tree decomposition and work upwards from the leaves, maintaining a dynamic programming table at each node in the tree. Given a tree node j, we denote its bag of vertices as X_j and let G_j denote the subgraph induced by all the vertices contained in bags at or beneath tree node j in the rooted tree decomposition. For each independent set $U \subseteq X_j$, there is a table entry with value $f_j(U)$ equal to the weight of the maximum weight independent set $S \subseteq G_j$ such that $S \cap X_j = U$. Since $G_r = G$, the largest value in the table for the root node r gives the weight of the maximum weight independent set in G. We now briefly describe the computation of $f_r(U)$ via dynamic programming.

For a leaf node l in the tree and an independent set $U \subseteq X_l$, the value of $f_l(U)$ is just the actual weight $w(U)$ of U since there are no vertices outside of $X_l \subseteq G_l$ to consider. For a non-leaf node j with d child nodes $c_1, c_2, \ldots, c_d$ and some independent set $U \subseteq X_j$, $f_j(U)$ can be calculated in terms of the

values stored at the child nodes via the dynamic programming equation

$$f_j(U) = w(U) + \sum_{i=1}^{d} \max\{f_{c_i}(V) - w(V \cap U) : V \cap X_j = U \cap X_{c_i}\}. \quad (1)$$

In other words, for every independent set $U \subseteq X_j$, we must look at the table for each child tree node c_i to find all independent sets $V \subseteq X_{c_i}$ that contain the same vertices as U from $X_{c_i} \cap X_j$. To compute the value of $f_j(U)$ we need to find the set V that has the largest value when one excludes their intersection.

There are two fundamental operations required in this dynamic programming recursion. First, we must have a fast method for finding all the independent sets in a bag of vertices. Second, for every independent set U that we find in a child node c_i, we must store it in such a way that the lookups required to compute equation (1) at the parent can be done very quickly. We describe the implementation of these operations in the next section.

3.1 Implementation Details

3.1.1 Set operations

The most important low-level operations in the dynamic programming algorithm are set operations: finding the intersection or union of two sets and checking if a particular vertex is in a set. For performance to be competitive with other methods on large problems, these operations must be performed quickly and in a memory-efficient manner. Since we are always dealing with subsets of a known set (typically a bag of vertices in the tree decomposition), we are able to represent subsets of vertices as bit vectors and perform the intersection, union, and containment operations via bitwise AND ($\wedge$) and OR ($\vee$) operations. As an example, suppose we have some bag of vertices $B = \{2, 4, 6, 8, 10, 11, 12, 15\}$ and two subsets $S, T \subset B$ with $S = \{2, 8, 10, 15\}$ and $T = \{4, 6, 10, 11, 15\}$. The set S is represented as `10011001` and T as `01101101`. To check if the i-th vertex of B is in S, we check to see if $S \wedge (2^i)$ is non-zero. To calculate the union $S \cup T = \{2, 4, 6, 8, 10, 11, 15\}$, we compute `10011001`$\vee$`01101101` = `11111101`.

When a tree decomposition has width less than the processor's word size (typically 32- or 64-bits), each of these bitwise operations can be done using a single CPU instruction. For larger width decompositions, we developed a `bigint_t` type to represent the required larger bitmasks as an array of 32-

or 64-bit words. The code for the `bigint_t` type is included in our software distribution and allows us to handle decompositions with arbitrarily large widths, as long as memory is available.

3.1.2 Finding all independent sets

A fundamental kernel in the dynamic programming algorithm for MWIS is finding all the independent sets in a graph. In particular, for each bag of vertices X_j in the tree decomposition, we must find all the independent sets in the subgraph induced by X_j. As there are $2^{|X_j|}$ sets to consider, it is clear that this search must be done efficiently, especially for larger widths. Algorithm 4 provides pseudocode for our implementation of this procedure, which takes advantage of the bitwise representation of subsets.

In Algorithm 4, we create a list S of all the independent sets contained in a graph $G = (V, E)$ with n vertices. The sets are represented by the bitmasks s that range from 0 (representing the empty set) up to $2^n - 1$ (representing all of V). In line 9 we check to see if there is some edge in the current set s that prevents it from being independent. When we find a non-independent set s', then in line 18 we are often able to advance the current value of s by a large amount. For example, if we know that the bitmask `1100100` does not represent an independent set, then any mask of the form `11001`$\cdot\cdot$ cannot represent an independent set. This allows us to advance to the bitmask `1101000` in the loop by adding `100` to `1100100`.

3.1.3 Memory-efficient storage

We now describe our storage methods that allow for efficient lookups when performing the dynamic programming recursion. Since we root the tree prior to beginning the dynamic programming, the parent-child relationship of all nodes in the tree is completely known when we search for independent sets and update the tables. As all of the operations required by the dynamic programming equation (1) involve the intersection of an independent set with its parent tree node's bag, we do not need an entry in the table of tree node i for every independent set $U \subseteq X_i$. In particular, if j is the parent of tree node i, then we have an entry in i's table only for the independent sets in $X_i \cap X_j$, reducing the amount of storage required by the dynamic programming tables.

Nevertheless, when processing a tree node j that has child node c_i, for every independent set $U \subseteq X_j$, we must still quickly find the entry for $U \cap X_{c_i}$ in the table for tree node c_i. Since all sets are represented as bitmasks, the

Algorithm 4 Find all the weighted independent sets in a graph $G = (V, E)$

```
1: procedure FindAllWIS(G)
2:     Let A be the adjacency matrix of G so that that the row A[i] is
       a bitmask representing i's neighbors and let W[i] denote the weight of
       vertex i
3:     S = {(∅, 0)}; n = |V|; s = 1        ▷ Store the empty set with weight 0
4:     while s < 2^n − 1 do                ▷ Loop over all non-empty subsets
5:         is_independent=true; i = w = 0
6:         while i < n and is_independent do
7:             if bit i is set in s then   ▷ Vertex i is in the current set s
8:                 w = w + W[i] ▷ Update the weight w of the current set s
9:                 if s ∧ A[i] then        ▷ s contains some edge (i, ·)
10:                    is_independent=false
11:                end if
12:            end if
13:            i = i + 1                   ▷ Consider the next vertex in V
14:        end while
15:        if is_independent then
16:            S = S ∪ (s, w); s = s + 1   ▷ Add the set and consider next
       bitmask
17:        else                            ▷ Jump over 2^b non-independent sets
18:            s = s + 2^b (b the least significant bit of s)
19:        end if
20:    end while
21:    return S
22: end procedure
```

natural solution to this problem is to utilize hash tables. As the width of our decompositions grow, the dynamic programming tables can occupy a great deal of memory, and so it is essential for the hash table implementation to be fast and lightweight. Rather than attempt to write our own hash, we experimented with several well-known existing implementations, including the Standard Template Library (STL) `map` and `unordered_map` as well as the Boost `hashtable`. However, in our experience all of these consumed far too much memory to handle larger width graphs, and we settled upon a fast, open source, macro-based implementation called `uthash`[15]. This has proved to be much faster and very robust, allowing us to handle decompositions with much larger widths than alternative hash table implementations.

3.1.4 Dynamic programming implementation

We now provide the details of our dynamic programming implementation for solving MWIS. Algorithm 5 describes how we compute the dynamic programming tables at each node in the tree. In line 7, we generate a list of all the independent sets contained in the subgraph induced by X_k, the bag associated with tree node k. In practice, we do not actually store this list but instead process each set as it is encountered for a slight savings in memory. For each independent set s discovered, we compute its value in the dynamic programming table in line 16 and then incorporate s into the table D_k in lines 17-24. Note that for each independent set, we store the triple $(s_p, s, f_k(s_p))$ where s_p is the intersection of s with the parent bag, and $f_k(s_p)$ is the value associated with this set in the dynamic programming table. The bitmask for s is stored as it enables the reconstruction of the full solution. However, this storage is not necessary if one wishes to simply determine the optimal weight.

Having described how we compute the dynamic programming table for each tree node j, it is now straightforward to solve MWIS for an input graph. Algorithm 6 returns the weight m of the maximum weighted independent set in G. However, it typically only gives us limited information about the actual optimal solution discovered since the entry (s, s, m) in the table D_r only gives us the vertices in this solution that are contained in the bag X_r. The actual solution itself can be reconstructed very quickly by descending back down the tree starting at the root node. Given a pre-order walk σ of the rooted tree decomposition, and the set s that represents this solution's intersection with the root node's bag, Algorithm 7 describes how to reconstruct the corresponding actual optimal solution that has weight m.

3.2 Memory Usage

The obvious bottleneck for the dynamic programming algorithm is the storage of the hash tables D_k at each tree node k. As mentioned previously, one way we reduce the memory requirements is by storing a single entry $(s_p, s, f_k(s_p))$ for each independent set s_p in the intersection of a tree node's bag with its parent's. Nevertheless, the memory usage for our algorithm can still be extremely large. Below we describe another optimization that proved to be quite beneficial in practice.

Algorithm 5 Compute the dynamic programming table for a node in the tree decomposition

```
1: procedure ComputeDPTable(G, T,k)
2: ▷ Graph G, tree decomposition T = (X, (I, F)), node k
3:     Let c_1, c_2, ..., c_d denote the children of tree node k and let p denote the parent
4:     Let D_j be the dynamic programming hash table for tree node j
5:     Let f_j(s) be the value associated with a set s in the table
6:     Let H = G[X_k] be the subgraph of G induced by k's bag
7:     S = FindAllWIS(H)                ▷ S is a set of ordered pairs (s, w(s))
8:     for all s ∈ S do
9:         z = w(s)
10:        for i = 1 to d do
11:            t_i = s ∩ X_{c_i}        ▷ t_i is the part of s contained in child i's bag
12:            Look up the entry for t_i in the table D_{c_i}: (t_i, ·, f_{c_i}(t_i))
13:            z = z + f_{c_i}(t)
14:        end for
15:        Let s_p = s ∩ X_p
16:        f_k(s_p) = z − w(s_p)        ▷ Subtract the weight of the parent intersection
17:        if (s_p, ·, ·) ∉ D_k then    ▷ Check the hash table for the key s_p
18:            D_k = D_k ∪ (s_p, s, f_k(s_p))▷ Add a new entry to the hash table
19:        else                         ▷ The key s_p exists in the hash table
20:            Let (s_p, s', x) be the current entry stored in D_k for s_p
21:            if f_k(s_p) > x then
22:                Update the value for s_p in D_k: s' = s and x = f_k(s_p)
23:            end if
24:        end if
25:    end for
26:    return D_k
27: end procedure
```

3.2.1 Refining the tree decomposition

In order to run Algorithm 7, one must maintain the dynamic programming tables for all tree nodes in memory. However, if one wishes to just determine the weight of the maximum weighted independent set, then one can free tables from memory once the parent tree node is processed. When we reach the root node r in the tree, we know the weight of an optimal solution m and some set $s \subseteq X_r$ that represents the intersection of this optimal solution with

Algorithm 6 Solve maximum weighted independent set

```
1: procedure MWIS(G, T, σ)
2:     Generate an ordering π using find_elimination_ordering
3:     Create a tree decomposition T = (X, (I, F)) by running
       GAVRIL(G, π) or BK(G, π).
4:     Root the tree T at an arbitrary node r and construct a post-order
       walk σ of T
5:     for i = 1 to |X| do
6:         COMPUTEDPTABLE(G,T,σ[i])
7:     end for
8:     m = 0                                   ▷ m will hold the max weight
9:     for all entries (s, s, x) in the root table D_r do
10:        m = max(m, x)
11:    end for
12:    return m
13: end procedure
```

Algorithm 7 Reconstruct the MWIS

```
1: procedure RECONSTRUCTSOLUTION(T, σ, s)
2:     Let S = s                              ▷ S will hold the optimal solution
3:     Let opt_int_sets be an |X|-long vector of sets
4:     Let opt_int_sets[σ[0]] = s
5:     for i = 0 to |X| do
6:         for all child tree nodes j of σ[i] do
7:             t_p = opt_int_sets[σ[i]] ∩ X_j   ▷ t is the part of opt. sol. in X_j
8:             Let (t_p, t, f_j(t_p)) be the entry corresponding to t_p in table D_j
9:             S = S ∪ {t}                  ▷ Append the set t to the optimal solution
10:        end for
11:    end for
12:    return S
13: end procedure
```

X_r. In practice, the set s typically contains only a few vertices and is often empty. Nevertheless, s still gives us information about the optimal solution that we can exploit. In particular, we know that any vertex $v \in X_r \setminus \{s\}$ is not in the optimal solution, and we know that all vertices neighboring some vertex in s are not in the optimal solution. Therefore, all these vertices can be removed from the bags of the tree decomposition and we can re-run the dynamic programming algorithm on the *refined* tree. Since this new tree typically has smaller width, the running time of the dynamic programming

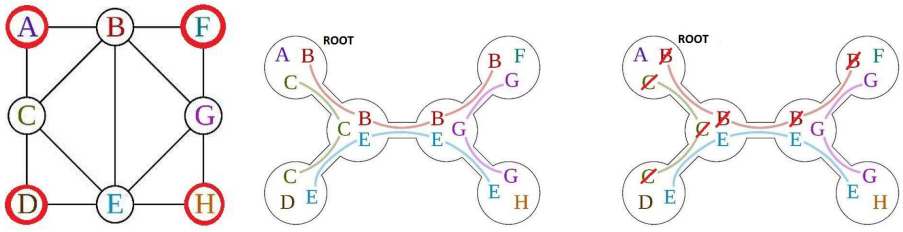

Figure 2: The maximum independent set in the graph is $\{A, D, F, H\}$, and A is the only vertex in this set also in the root node. Therefore, B and C can be removed from the tree decomposition in the refinement procedure.

algorithm on the new tree is exponentially smaller, and the tables require much less space, allowing us to store them in memory in order to reconstruct the solution.

3.2.2 Memory usage estimation

Even with the refined tree, the memory required to process a single tree node can still be too large, so we analyzed the memory consumption in more detail. Given a tree node i and its parent p, we have a single entry for each independent set contained in the intersection of $Y = X_i \cap X_p$. When the relevant subgraph induced by Y is very sparse, there can be $O(2^{|Y|})$ independent sets to store, and so the memory consumption can truly be exponential in the size of Y. However, the density of the subgraph plays a critical role in the actual expected number of independent sets contained in such a subgraph.

Under a few basic assumptions, we can estimate the expected total number of independent sets contained in a subgraph and use this to determine if a tree decomposition-based approach is tractable. Given some set Y as above, let H be the subgraph induced by Y. Denote the number of vertices in H as w and the number of edges as s and assume that the probability of any two vertices in H being joined by an edge is the same for all pairs of vertices. Then the probability of any two vertices u, v in H not being joined by an edge is $\rho = 1 - s/\binom{w}{2}$, and so the probability that some set of q vertices from Y represents an independent set is $\rho^{\binom{q}{2}}$. The expected number of independent sets in H is then

$$E[|\textsc{FindAllWIS}(H)|] = 1 + \sum_{k=1}^{w} \binom{w}{k} \rho^{\binom{k}{2}}. \tag{2}$$

We were unable to determine an asymptotic formula for equation (2), but we can compute the sum directly. Our software is able to compute this value exactly using multiple precision arithmetic, and in the next section we demonstrate that it is typically a very good estimate of the number of independent sets found and stored over the course of the dynamic programming algorithm implementation.

4. Computational Results

Our goals are to compare the overall performance of our dynamic programming algorithm with other well-established methods, to explore how our algorithm's performance scales as we alter various properties of the graphs, and to examine the traditional wisdom regarding the maximum width graphs that can be handled via tree decomposition-based dynamic programming. All of the experiments in this section were conducted using a standard Linux compute node equipped with 16GB of RAM and two quad-core AMD processors.

4.1 Partial k-trees

One of the challenges in analyzing the performance of our algorithms was finding a suitable set of graphs with a wide variety of sizes and densities with known upper bounds on the treewidth. Fortunately, it is straightforward to generate such graphs, using the definition of partial k-trees. The class of *k-trees* is defined recursively. In the smallest case, a clique on $k+1$ vertices is a k-tree. Otherwise, for $n > k$, a k-tree G on $n+1$ vertices can be constructed from a k-tree H on n vertices by adding a new vertex v adjacent to some set of k vertices which form a clique in H. A k-tree has treewidth exactly k (the bags of the optimal tree decomposition are the cliques of size $k+1$). The set of all subgraphs of k-trees is known as the *partial k-trees*. It easy to see that any partial k-tree has treewidth at most k (one can derive a valid tree decomposition of width k from that of the k-tree which contains it). Furthermore, any graph with treewidth at most k is the subgraph of some k-tree [19]. Thus the set of all graphs with treewidth at most k can be generated by finding all k-trees and their subgraphs, leading us to an easy randomized generator for graphs of bounded treewidth. The *INDDGO* software distribution includes an executable, *gen_pkt*, to produce randomly generated partial k-trees.

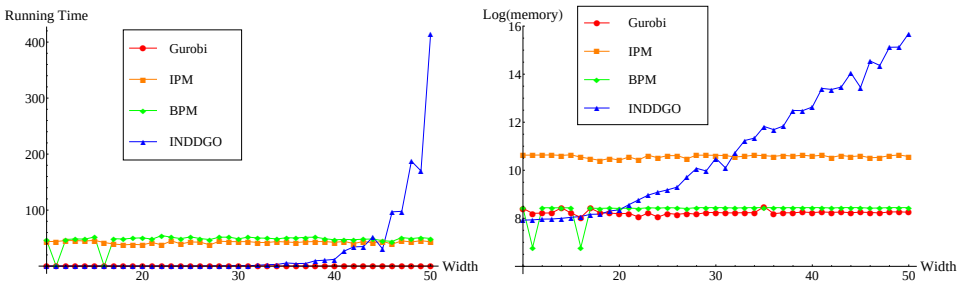

Figure 3: The running time and memory usage of the other methods remain roughly constant as the width varies while n and m remain constant. The tree decomposition based approach requires more memory as the width increases.

4.2 Comparison with other algorithms

We compared the runtime and memory usage of our algorithm against other well-known implementations: the commercial mixed integer programming solver, *Gurobi*, and two freely available branch and bound algorithms for MWIS based on the semi-definite programming (SDP) relaxation [10]. One of the SDP-based codes uses an interior point method (*IPM*) to solve the SDP, and the other uses a boundary point method (*BPM*).

For experiments with *Gurobi*, we formulate the MWIS instance as a pure 0/1 integer programming (IP) problem and then produce an input file that is read directly by *Gurobi*. The two implementations based on the SDP relaxation are able to read problem instances directly from so-called DIMACS files [26] so that no translation is necessary. Before presenting the results, we note that it is not our intention to claim superiority of any one implementation over another. Instead, we are primarily interested in the scaling behavior of each implementation in terms of the *size* of the instance, measured in terms of the number of nodes, number of edges, or *width* of a given tree decomposition.

In our first computational experiment, we generated a set of 40 partial k-trees. Each of these graphs has 256 nodes and approximately 2056 edges, with k running from 11 to 50 and p (the probability of keeping an edge in the k-tree) varying from 0.17 to 0.81. We created tree decompositions using the *Greedy Minimum Fill* heuristic and Gavril's algorithm, and ran our dynamic programming algorithm along with the SDP codes and *Gurobi*. In Figure 3, we see that the running time and memory usage of our dynamic programming are in line with the other methods until we reach the graphs

Implementation	Number Completed	Avg. Time	Max Time	Avg. Mem (GB)	Max Mem(GB)
Gurobi	80	26	499	0.18	1.21
BPM	51	53819	432226	0.18	0.87
IPM	32	58835	442509	3.35	16.01
TD	56	87	795	2.02	24.16
TD with refinement	62	134	2375	1.05	17.58

Table 2: A comparison of the performance of four different WIS implementations on a set of 80 partial k-trees. Average and maximum values apply only to completed graphs.

with width 40. At this point, both the memory usage and running times for our dynamic programming begin to increase very rapidly and require significantly more resources than the other methods. This supports the notion that, all other things being equal, the underlying treewidth of a graph does not affect the running time of the SDP- or IP-based methods whereas the running time and memory usage of the dynamic programming implementation are both very sensitive to the width of the tree decomposition.

In the next set of experiments, we generated 80 partial k-trees with the number of nodes $n \in \{1000, 2000, 4000, 8000\}$, $k \in \{15, 30, 60, 90, 120\}$, and keeping edges in the k-trees with probabilities $p \in \{0.2, 0.4, 0.6, 0.8\}$. We ran each of the four codes on all of the graphs, recording the running time and memory usage of each run. While *Gurobi* was able to produce an optimal solution for all 80 instances, the other methods met with varying degrees of success. When running our algorithm and keeping all of the dynamic programming tables in memory, we completed 56 of the 80 graphs and ran out of memory on the remainder. If we used the refinement procedure discussed in Section 3.2, we were able to complete 62 of the 80 graphs. The *BPM* and *IPM* implementations completed 51 of 80 and 32 of 80 graphs, respectively. Running time was typically the bottleneck for these methods (we imposed a limit of five days computing time). However, in fairness to these two methods, we note that neither was designed to handle graphs with large numbers of nodes or edges. In fact, these implementations can occasionally solve smaller instances that our dynamic programming implementation cannot. Some specific details summarizing the results of these computations are given in Table 2.

Since *Gurobi* is clearly the most successful of these methods for solving instances from our data set, we made some more detailed comparisons of the performance of our dynamic programming algorithm utilizing the tree refinement procedure. We find that this variant of our dynamic program-

		Gurobi		*Refined TD*	
n	m	time	memory	time	memory
1000	17721	0.543	29008	0.54	17528
2000	35721	1.684	52516	0.99	26104
4000	71721	4.423	98700	2.76	53040
8000	143721	19.618	194284	8.72	101236
1000	23628	0.764	38348	0.36	9416
2000	47628	2.35	70140	0.75	17096
4000	95628	10.424	133872	2.85	30664
8000	191628	44.986	254920	8.6	56176

Table 3: A comparison of the dynamic programming algorithm versus *Gurobi* on a particular family of partial k-trees with $k = 30$ and $p = .8$.

ming implementation can be up to 5 times faster than *Gurobi* on certain graphs. A more detailed inspection of the results shows that our implementation is faster on all partial k-trees in our test set with $k = 15$ or 30 and $p = 0.6$ or $p = 0.8$. These are lower width instances that share the majority of edges with the original k-tree. Since each bag in the tree decomposition of these graphs is somewhat dense, equation (2) implies that the dynamic programming tables remain small, leading to lower memory usage and faster running times. Table 3 contains some more detailed information regarding the instances with $k = 30$ and $p = 0.8$. It is worth noting that on all of these instances where our running time is lower, our memory usage is also substantially less than *Gurobi*'s.

4.3 Results on large width graphs

Due to the theoretical exponential growth in running time and memory usage, the traditional thinking has been that graphs with larger widths cannot be handled by dynamic programming based on tree decompositions. For example, Hüffner, et al, [17] state that "As a rule of thumb, the typical border of practical feasibility lies somewhere below a treewidth of 20 for the underlying graph." In this section, we run our algorithm on a particular class of partial k-trees where we increase the parameter k as much as possible. This allows us to find the optimal solution to weighted independent set instances on graphs with 10,000 nodes where the width of the underlying decomposition is as large as 708.

For this experiment, we generated partial k-trees with 10,000 nodes, $p = 0.9$ and $k \in \{100, 200, \ldots, 700\}$. The resulting seven graphs have up to

m	width	running time	Total DP table entries	memory(GB)
895455	103	112	6,395,909	0.32
1781910	205	790	29,554,406	1.14
2659365	306	3638	78,361,001	3.00
3527820	405	14353	165,086,196	7.70
4387275	505	33223	289,374,021	12.80
5237730	615	93450	478,397,538	20.92
6079185	708	168411	715,022,103	36.99

Table 4: Details of our algorithm's performance on a set of high width graphs with 10,000 nodes and up to 6 million edges.

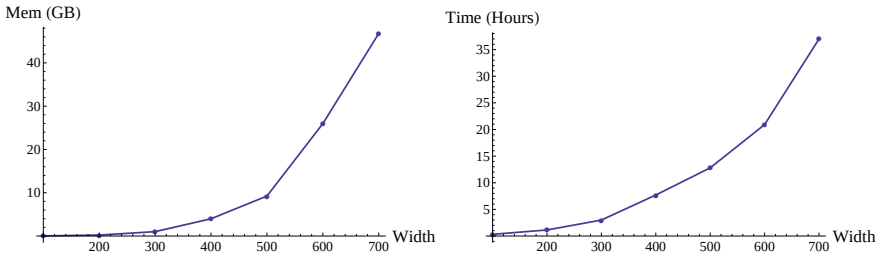

Figure 4: The memory usage and running time of our dynamic programming algorithm on large-width graphs.

six million edges and we solved them to optimality on a server with 512 GB RAM. Table 4 presents the running time and memory usage of our procedure on these graphs, and these values are plotted in Figure 4

4.4 Estimating the Memory Usage

Virtually all of the storage required by our dynamic programming algorithm is related to the storage of the tables at each tree node. Given a tree node k with bag X_k and its parent's bag X_p, recall that for each independent set s contained in the subgraph induced by X_k, there is an entry in the dynamic programming table for $s \cap X_p$ (see line 15 of Algorithm 5). Thus, in order to estimate the number of dynamic programming table entries required for tree node k, we apply our estimate from equation (2) to the subgraph induced by $X_k \cap X_p$.

We ran an experiment on a set of graphs to test the empirical accuracy of formula (2) as a tool for estimating the total number of table entries necessary for the dynamic programming. The first set of graphs consisted

Graph	(n, m)	width	# of DP entries	Est. # of DP entries
ch150.tsp	(150,432)	19	7456	18825.8
celar07pp	(162,764)	18	1677	1763.4
a280.tsp-pp-003	(169,490)	19	7262	37576.9
a280.tsp-pp	(261,749)	19	9065	39847.2
celar08pp	(265,1029)	18	3321	3591.1
a280.tsp	(280,788)	20	10532	79877.1

Table 5: A comparison of the accuracy of our formula for estimating the number of dynamic programming table entries required by our algorithm

purely of partial k-trees. On these graphs, our estimate was within 10% of the actual value in every case. However, since the procedure that we use to randomly generate partial k-trees adheres closely to the assumptions behind equation (2), we also ran the same procedure on a set of 48 graphs taken from Hans Bodlaender's *TreewidthLib* [7] that contained anywhere between 50 and 1290 nodes. For these graphs, our estimate of the total number of independent sets was within an average of 87% of the actual value, typically over-estimating the required number of entries. Details of this comparison for some of these graphs are shown in Table 5

5. Obtaining and Running the Code

The latest stable version of our tree decomposition and dynamic programming code is freely available via `github` at `https://github.com/bdsullivan/INDDGO`. One can download a compressed archive of the code and build it following the instructions contained therein.

5.1 Algorithm options

During the course of developing our code for solving MWIS via tree decompositions, we experimented with many different options for various stages of the procedure: the heuristic used to determine the elimination ordering, the algorithm used to construct a tree decomposition from an elimination ordering, different techniques to root the tree, different methods to process a tree node, and so on. We also experimented with so-called *nice* tree decompositions which allow a particularly simple dynamic programming algorithm for MWIS [9]. While dynamic programming using nice decompositions was generally slower in our experiments, our software allows one to experiment with these types of decompositions. It is not possible to give an exhaustive

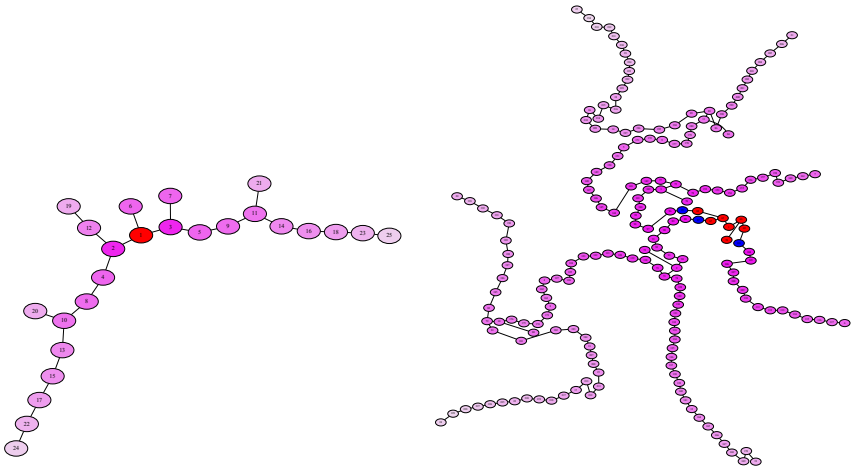

Figure 5: The figure on the left shows a diagram of a width 35 decomposition where tree nodes are red if their bag has 36 vertices, blue if 35 vertices, and gradations of purple for smaller bag sizes. The diagram on the right shows the same decomposition after it is transformed into a nice decomposition with the `make_nice` option.

comparison of all the options provided by our software, but we give a brief description of many of the implemented options in Table 6. The main binary created by compiling *INDDGO* is `serial_wis` and all of the options listed below are accessible via command line switches.

5.2 Example Usage

A typical use of the code is to create a decomposition using some elimination ordering heuristic and then solve MWIS. The following command does this for a small sample graph included with the distribution, using the *AMD* heuristic for the elimination ordering. We also generate a file that can be processed by Graphviz [1] for visualization. Some sample visualizations of the decomposition are given in Figure 5.

```
serial_wis -f ../sample_graphs/1dc.64.dimacs -gviz 1dc.64.gviz -amd -gavril
file n m w obj
1dc.64.dimacs 64 543 35 10
```

Additional details on the dynamic programming can be produced by adding the `-v` option.

Option	Description
`-gavril`	Uses Gavril's algorithm [12] to construct a tree decomposition from an elimination ordering
`-bk`	Uses the algorithm of Bodlaender-Koster [9] to construct a tree decomposition from an elimination ordering
`-make_nice`	Transforms the decomposition into a *nice* decomposition
`-w <file>`	Writes the decomposition to a file in a DIMACS-like format
`-t <file>`	Reads the decomposition from a file
`-gviz <file>`	Writes the decomposition in .dot format for Graphviz
`-root <v>`	Roots the tree at node v
`-asp_root`	Uses the algorithm of Aspvall, et al,[4] to determine a suitable root node
`-child_root`	Considers the structure of the parent/child relationship when rooting the tree
`-dfs`	Uses a depth-first search to generate the post order walk (breadth-first is default)
`-nonniceDP`	Uses non-nice dynamic programming routines for a nice decomposition
`-del_ch`	Deletes the dynamic programming tables once they have been processed
`-pc`	Searches for independent sets by trying to modify an existing child's table
`-split_bag`	Divides a tree node's bag into two parts prior to searching for ind. sets
`-async_tbl`	Does the DP operation one child node at a time rather than all at once
`-mem_est`	Uses equation (2) to estimate the memory usage of the algorithm

Table 6: Some of the options available with `serial_wis` in *INDDGO*

6. Conclusion

In this paper, we have described an efficient and freely available software library for generating tree decompositions and running dynamic programming to solve weighted independent set instances. Our software offers easy-to-use implementations of many well-known heuristics for producing elimination orderings that lead to low-width tree decompositions. Our dynamic programming code is particularly memory efficient and computational experiments indicate that our implementation is competitive and even superior to state of the art methods for certain types of MWIS instances. While sparse graphs with large widths present memory consumption difficulties that our code is currently unable to handle, we have nevertheless demonstrated that dynamic programming on decompositions with very large widths can be feasible in some cases, casting doubt on the conventional wisdom. While our code currently is able to solve only weighted independent set, our software framework is designed so that other dynamic programming algorithms can be incorporated in a modular fashion. Finally, we mention that we have created a parallel version of our software that is able to generate tree decompositions and solve weighted independent set instances using distributed memory architectures [27].

7. Acknowledgments

The authors thank the support of the Department of Energy Applied Mathematics Program for supporting this work. Additionally, we thank Hans Bodlaender for sending us a large set of test graphs, Arie Koster for sending us his source code for elimination orderings, Brian Borchers for assistance with building and running the IPM and BMP codes, interns Gloria D'Azevedo and Zhibin Huang for early participation in the project, and Josh Lothian for helping to organize the code and build procedures.

References

[1] `Graphviz`, graph visualization software.

[2] P. Amestoy, T. Davis, and I. Duff. "an approximate minimum degree ordering algorithm". *"SIAM J. Matrix Analysis and Applications"*, 17:886–905, 1996.

[3] Eyal Amir. Approximation algorithms for treewidth. *Algorithmica*, page in press, 2008.

[4] Bengt Aspvall, Andrzej Proskurowski, and Jan Arne Telle. Memory requirements for table computations in partial k-tree algorithms. In *Algorithm Theory - SWAT'98*, pages 222–233. Springer-Verlag, 1998.

[5] Anne Berry, Jean Blair, Pinar Heggernes, and Barry Peyton. Maximum cardinality search for computing minimum triangulations of graphs. In *ALGORITHMICA*, pages 1–12. Springer-Verlag, 2002.

[6] A. Bockmayr and K. Reinert. Discrete math for bioinformatics, 2010.

[7] H. Bodlaender. `uthash`, a benchmark for algorithms for treewidth and related graph problems, 2011.

[8] H. L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal of Computing*, 25:1305–1317, 1996.

[9] H. L. Bodlaender and A. M. C. A. Koster. Combinatorial optimization on graphs of bounded treewidth. *The Computer Journal*, 51(3):245–269, 2008.

[10] B. Borchers and A. Wilson. Branch and bound code for maximum independent set. `http://euler.nmt.edu/~brian/mis_bpm/`, 2009.

[11] Bruno Courcelle. Graph rewriting: An algebraic and logic approach. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B)*, pages 193–242. 1990.

[12] Fanica Gavril. The intersection graphs of subtrees in trees are exactly the chordal graphs. *Journal of Combinatorial Theory, Series B*, 16:47–56, 1974.

[13] A. George and J. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Review*, 31:1–19, 1989.

[14] Vibhav Gogate and Rina Dechter. A complete anytime algorithm for treewidth. In *Proceedings of the 20th conference on Uncertainty in artificial intelligence*, UAI '04, pages 201–208, Arlington, Virginia, United States, 2004. AUAI Press.

[15] T. Hanson. `uthash`, a hash table for *c* structures, http://uthash.sourceforge.net/, 2011.

[16] I. V. Hicks, A. M. C. A. Koster, and E. Kolotoglu. Branch and tree decomposition techniques for discrete optimization. *Tutorials in Operations Research, INFORMS–New Orleans*, 2005.

[17] Falk Huffner, Rolf Niedermeier, and Sebastian Wernicke. Developing fixed-parameter algorithms to solve combinatorially explosive biological problems. 453, May 2008.

[18] G. Karypis. *METIS* - serial graph partitioning and fill-reducing matrix ordering, 2011.

[19] T. Kloks. *Treewidth: Computations and Approximations.* Lecture Notes in Computer Science. Springer, 1994.

[20] Joseph Liu. Modification of the minimum-degree algorithm by multiple elimination. *ACM Trans. Math. Software*, 11:141–153, 1985.

[21] N. Robertson and P. D. Seymour. Graph minors XX: Wagner's conjecture. *Journal of Combinatorial Theory, Series B*, 92:325–357, 2004.

[22] Neil Robertson and Paul D. Seymour. Graph minors III: Planar tree-width. *Journal of Combinatorial Theory, Ser. B*, 36(1):49–64, 1984.

[23] H. Röhrig. Tree decomposition: A feasibility study. Master's thesis, Max-Planck-Institut füur Informatik, Saarbrüucken, Germany, 1998.

[24] D. Rose, R. Tarjan, and G. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comput.*, 5:266–283, 1976.

[25] P. D. Seymour and R. Thomas. Call routing and the ratcatcher. *Combinatorica*, 14(2):217–241, 1994.

[26] N. Sloane. Challenge problems: Independent sets in graphs, 2011.

[27] Blair D. Sullivan, Dinesh Weerapurage, and Chris Groër. Parallel algorithms for graph optimization using tree decompositions. *ORNL/TM-2012/194*, 2012.