A Comparison of Exact Treewidth Implementations

Joey Eremondi
Utrecht University 15cu Experimentation Project
Supervisor: Hans Bodlaender
UU#: 4229924

November 10, 2015

Abstract

Treewidth-decompositions of graphs allow for some NP-hard operations to be performed efficiently if a graph has small enough treewidth. However, calculating the treewidth of a graph is itself NP-hard. Brute force search, or naive recursive algorithms, are prohibitively slow. More time-efficient dynamic programmign algorithms have been developed. However, these algorithms tend to run out of space on large inputs, making memory usage a problem before running time.

In this experiment, I examine different approaches to computing the exact treewidth of a graph, comparing different techniques with different space-time tradeoffs. I examine their time and memory usage when tested on a variety of graphs, and provide some discussion as to the ineffectiveness of the provided methods.

1 Background

1.1 Treewidth

Given a graph $G = (V, E)$, a tree-decomposition of that graph is a pair $(\{X_i \mid i \in I\}, T = (I, F))$ with each $X_i \subseteq V$ (called a "bag"), fulfulling the following conditions:

- If $v \in V$, then there exists $i \in I$ with $v \in X_i$
- If $\{v, w\} \in E$, then there exists $i \in I$ with $\{v, w\} \subseteq X_i$
- If $v \in V$, then $\{X_i \mid i \in I, v \in X_i\}$ forms a connected subtree of T.

The width of a decomposition is defined as $\max_{i \in I} |X_i| - 1$. The treewidth of a graph G, denoted $tw(G)$, is defined as the minimum width of any tree-decomposition of G.

The concept of treewidth is primarily used in solving NP-hard problems: when a graph can be decomposed with with k, often NP-hard problems on that graph can be solved in time $O(2^k P(|V|))$ for some polynomial P, giving us a polynomial-time algorithm for graphs of bounded treewidth. However, finding the treewidth of a graph is itself an NP-hard problem.

For $S \subseteq V$ and $v \notin S$, the set $Q(S, v) = \{w \in V \setminus (S \cup \{v\}) \mid v \text{ has a path to } w \text{ using only intermediate vertices from } S\}$. For a permutation π of V, and $v \in V$, we define $\pi(v, <) = \{w \in S \mid \pi(w) < \pi(v)\}$.

For a set $S \subseteq V$, we define $TW(S) = \min_{\pi \in \Pi(V)} \max_{v \in S} Q(\pi_{<,v}, v)$. In [5], it is shown that $tw(G) = TW(V) = \min_{\pi \in \Pi(V)} \max_{v \in V} Q(\pi_{<,v}, v)$.

This decomposition gives rise to a recurrence relation for finding $tw(G)$:

- $TW(\emptyset) = -\infty$
- $TW(S) = \min_{v \in S} \max(TW(S \setminus \{v\}), |Q(S \setminus \{v\}, v)|)$ for $S \neq \emptyset$
- $tw(G) = TW(V)$

1.2 Directed Acyclic Word Graphs

A *trie* is a tree structure, in which edges are marked with letters over some alphabet, and each path to a leaf node represents a stored word.

In a trie structure, many sub-trees may be identical. Merging identical subtrees gives way to Directed Acyclic Word Graphs, or DAWGs. A DAWG for a finite set of words is equivalent to the minimal finite automaton accepting the set of words. (Note that the resulting DAWG may be a multigraph, two nodes may be connected by multiple edges, each labeled with a different letter).

A trie containing n words contains at least n leaves, so it requires more space than, for example, a balanced binary tree. However, DAWGs can provide sub-linear storage in some cases. For example, the set of binary words of length n contains 2^n words, but can be represented as a DAWG with $n+1$ nodes and $2n$ edges.

A trie containing a set S of size n with a maximum word length k contains at most $O(kn)$ nodes, since there are exactly n simple paths to a leaf (since each defines a word), which contain at most k vertices. Since every DAWG contains no more nodes than an equivalent trie, we have an upper bound of $O(kn)$ for the size of a DAWG. In both a trie and a DAWG, lookup can be achieved in $O(k)$ time, assuming we can access the outgoing edges of a given node in $O(1)$ time. In practice, we reach performance near this bound by storing edges in a hashtable, which also saves memory in the case of sparse graphs.

2 Algorithms For Exact Treewidth

The recurrence relation described above lends itself naturally to dynamic-programming or memoized algorithms for computing exact treewidth. However, there are two distinct approaches to this, a *bottom-up* method, where the algorithm is solved recursively but values are cached, and a *bottom-up* method, where TW values for larger sets are built up from smaller ones.

2.1 Top-down

The top-down algorithm roughly follows the following pseudocode:

Let $G = (V, E)$;
Let D be a dictionary mapping subsets of V to $\mathbb{N}$;
Function $TW(S)$**:**
  if $|S| = 0$ **then**
    return $-\infty$
  end
  else if D *contains* S **then**
    return $D[S]$
  end
  else
    $tw := \min_{v \in S} \max(TW(S \setminus \{v\}), |Q(S \setminus \{v\}, v)|)$;
    $D[S] := tw$;
    return tw;
  end
end
return $TW(V)$;

It works as one would expect a recursive algorithm to work, with values cached for efficiency.

2.2 Bottom-up

The bottom-up version, originally presented in [5], works somewhat differently. Instead of calculating the final TW value for each set, we calculate the "smallest" possible value found so far, and refine it by looking

at past values.

```
Let G = (V, E);
Let n = |V|;
For each i ∈ I = {0, ..., n}, let D[i] be a dictionary mapping subsets of V to ℕ;
D[0][∅] := −∞;
for i ∈ {1..., n} do
    for (S, r) ∈ D[i − 1] do
        for v ∈ V \ S do
            q := |Q(S, v)|;
            r′ = max(r, q);
            if D[i] contains S ∪ {v} then
                oldval := D[i][S ∪ {v}];
                D[i][S ∪ {v}] := min(oldval, r′);
            end
            else
                D[i][S ∪ {v}] := r′;
            end
        end
    end
end
return D[n][V];
```

3 Optimizations

The difference between these methods is only significant when combined with optimizations, which interact differently with each method.

3.1 Cliques

Its shown in [6] that if C is a clique in G, then either $tw(G) = |C| - 1$, or $tw(G) = TW(V \setminus C)$. We use this to our advantage, by computing the maximum clique in our graphs and removing those vertices from the initial set before beginning our bottom-up or top-down algorithms.

While the maximum-clique algorithm is technically exponential in time, the branching algorithm we use is has running-time which is negligible in comparison to the rest of the algorithm's running time.

3.2 Upper-bounds

We will use $UB(S)$ to denote the computed upper-bound for $TW(S)$.

Using local-search, we can estimate an upper-bound on the maximum value for $tw(G)$, or even for $TW(S)$ for some S.

Moreover, it is stated in [5] that for any $S \subseteq V$, $TW(V) \leq \max(TW * S, |V \setminus S| - 1)$. This allows us to continually lower our global upper-bound as we compute TW values for subsets.

In the bottom-up method, we can avoid inserting any value into some $D[i]$, if it is higher than our currently calculated upper-bound. If, when we reach the end, there are no values stored, we know that the graph has a treewidth equal to the upper bound. As a result of this, in practice, the bottom-up method often terminates well before reaching the final level.

In the top-down method, this allows us to avoid computing $TW(S \setminus \{v\})$ if $|Q(S \setminus \{v\}, v)| \geq UB(S)$.

The performance of the algorithms differs drastically based on these upper bounds. In the top-down versions, we recurse down from values which may never be reached in the bottom-up version.

3.3 Lower-bounds

We will use $LB(S)$ to denote the computed lower-bound for $TW(S)$.

Two lower bounds can be computed: a global lower bound on tw, and a local lower bound on $TW(S)$ for a specific S.

Global lower bounds have been studied extensively in [7]. In this experiment, we used the following lower bounds:

- δ_2-degeneracy (Algorithm 2 from [7])
- MMD+ heuristic for contraction degeneracy (Algorithm 3 from [7])

For local lower bounds, the above methods are not applicable. However, it is clear to see from our recurrence relation that for any S, $\min_{v \in S} |Q(S \setminus \{v\}, v)|$ provides a lower-bound on $TW(S)$. In practice, this means that whe computing $\max(TW(S \setminus \{v\}), |Q(S \setminus \{v\}, v)|$, we do not need to compute $\max(TW(S \setminus \{v\})$ if $|Q(S \setminus \{v\}, v|)$ exceeds some upper-bound we have already computed.

3.4 Simplicial vertices

For our top-down algorithm, we adapt a heuristic from [6]. A vertex w is simplicial with respect to a subset S if all neighbours of w in S form a clique. If a vertex w in S is simplicial, we know that $TW(S) = \min_{v \in S} \max(TW(S \setminus \{v\}), |Q(S \setminus \{v\}, v)|) = \max(TW(S \setminus \{w\}), |Q(S \setminus \{w\}, w)|)$. This allows us to reduce the number of branches we explore in certain cases.

4 The Experiment

I implemented and compared three different approaches to computin exact treewidth, with a variety of parameter values.

The source code for this implementation can found in [9].

4.1 DAWG implementation

The first test used DAWGs in the implementation of the bottom-up method. The hypothesis here was that, there would be large groups of similar vertex-sets stored in each stage of the algorithm, that the groups of sets not eliminated by upper or lower bounds would be similar.

Consider $G = (V, E)$ and $V = \{v_1, \ldots, v_n\}$, and $S \subseteq V$ with $TW(S) = k$. We encode the set as a string over $\{0, 1, tw_0, \ldots, tw_n\}$, in the form $b_1 \cdots b_n tw_k$, where $b_i = 1$ if $v_i \in S, 0$ otherwise. No special selection method was used to choose the ordering of vertices, so choosing this ordering is a potential improvement for future work.

Because of the high cost of inserting into a DAWG, during the algorithm we inserted newly calculated TW values into a trie, called the "staging-area". There was a parameter $maxTransitions$: whenever the total number of transitions in the DAWG and its staging area exceeded $maxTransitions$, we calculated their union using the product construction for DFAs [10] (modified for union instead of intersection), then minimized the result using a linear-time acyclic-DFA minimization algorithm described in [8].

We tested our DAWG implementation with $maxTransition$ values of 100 million, 50 million, and 10 million.

4.2 Tree-based bottom-up

As our "control" implementation, I implemented the bottom-up algorithm. At each stage of the algorithm, we had an array of size $|V|$ storing a `std::set` (usually implemented in the standard library using balanced trees). If a set S has TW value k, we add S to the kth set.

The hash-table based `std::unordered_set` could have also been used, and while it tends to provide a speed increase, the hash table requires extra memory, and as the results show, the performance of `std::set` was quite satisfactory.

4.3 Hash-based top-down

I tested a "fixed-memory" version of the top-down algorithm. Vertex subsets are cached in a chaining hash-table. However, the chains for each hash value are of fixed-length, and the least-recently accessed values were discarded when chains reached their maximum length. Thus, there is an upper-bound on the memory used by the code.

For this test (and the hybrid test below), we used a hashtable of size 9999991, with at most 100 elements stored in each hash chain.

4.4 Hybrid approach

In an attempt to get the "best of both worlds" for top-down and bottom-up versions, I implemented a hybrid algorithm. This version carried out as many iterations of the bottom-up algorithm as it could before a set memory limit was reached. When the limit was reached, a top-down algorithm was started, but accessed the already-computed values from the final completed layer of the bottom-up iterations.

We tested our hybrid approach where at most 10000 values were stored in any layer of the bottom-up algorithm. While much higher values of this parameter were feasible, they usually resulted in the tw value being found in the bottom-up phase, providing very little useful information about the top-down approach. The top-down parameters were the same as in the previous test.

4.5 Common elements

My implementation uses C++11 features heavily, often omitting long-types with the *auto* declarations [1], and producing more readable code using range-based iterators [3].

I rely on several utilities from the boost libary. The `dynamic_bitset` library was used for a space-efficient representation of subsets of vertices, which also allowed for very fast intersection and union operations. The Boost Graph Library was used for graph operations, and Boost timers were used to evaluate the run-times of the program.

5 Results

The experiments were run on Ubuntu Linux 14.10, on a laptop with a dual-core 2.8 GHz i7 processor and 8GB RAM. The programs were compiled using the Clang C++ compiler with `-O2` selected as the otpimization level [2]. Run-times were measured with Boost timers, and memory-usage was profiled using the GNU-time utility [4].

The time is reported in seconds, and is the total CPU-time of the algorithm's run. The memory is reported in Kilobytes, and is specifically the "maximum resident set size of the process during its lifetime" [4].

The graphs were taken from a previous Utrecht treewidth project, libTW [11], though several graphs were also used in [5].

We summarize our results in Table 1.

6 Analysis

In general, the bottom-up tree-based implementation performed best in terms of both time and space.

6.1 DAWG Time Inefficiency

For any graphs that were non-trivial in size, the DAWG method was impractically slow, and the tests did not finish running after several hours. Even for those tests included in the results, the time required is usually much greater than the tree-based version. While much effort was put into making the implementation efficient, this is likely due to the time required for merging the staging area with the DAWG. If there are n transitions in a DAWG and m in a minimized staging area, then it takes $O(mn)$ time to compute their union. Inserting a word of length w directly takes $O(nw)$ time.

Graph	$\|V\|$	$\|E\|$	LB	UB	tw	DAWG 100M	DAWG 50M	DAWG 10M	Bottom-up	Hybrid	Top-down
ana-pp	22	148	5	13	12	**0.02 s**	0.03 s	**0.02 s**	**0.02 s**	0.06 s	0.26 s
						4256 kb	4456 kb	4424 kb	**4060 kb**	160264 kb	160084 kb
barley-pp	26	78	4	10	7	31.6 s	31.41 s	30.71 s	10.08 s	4.26 s	**4.22 s**
						90632 kb	90628 kb	90684 kb	**23952 kb**	178316 kb	178112 kb
david-pp	29	191	4	18	13	5.7 s	6 s	5.75 s	2.19 s	0.78 s	**0.68 s**
						29616 kb	29488 kb	29664 kb	**9344 kb**	161436 kb	160800 kb
mainuk-pp	9	28	2	6	6	**0 s**	**0 s**	**0 s**	**0 s**	0.02 s	0.02 s
						4028 kb	4048 kb	4032 kb	**4016 kb**	159940 kb	160080 kb
MCSTestGraph	7	11	3	3	3	**0 s**	**0 s**	**0 s**	**0 s**	0.04 s	0.05 s
						4048 kb	3956 kb	3956 kb	**3828 kb**	160164 kb	159936 kb
MCSTestGraph2	9	13	1	3	3	**0 s**	**0 s**	**0 s**	**0 s**	0.02 s	s
						3968 kb	4052 kb	3880 kb	**3824 kb**	159908 kb	160160 kb
myciel3	11	20	2	5	5	**0 s**	**0 s**	**0 s**	**0 s**	0.02 s	0.03 s
						3968 kb	3968 kb	**3920 kb**	4024 kb	159996 kb	160076 kb
myciel4	23	71	4	10	10	0.32 s	0.32 s	0.32 s	**0.11 s**	0.12 s	16.83 s
						8216 kb	8044 kb	8072 kb	**4348 kb**	160076 kb	204952 kb
oesoca+-pp	14	75	2	11	11	**0 s**	**0 s**	**0 s**	**0 s**	0.03 s	0.02 s
						4096 kb	4208 kb	4100 kb	**3928 kb**	160084 kb	160096 kb
pathfinder-pp	12	43	3	7	6	**0 s**	**0 s**	**0 s**	**0 s**	0.02 s	0.03 s
						3992 kb	4144 kb	4064 kb	**3988 kb**	159940 kb	160044 kb
queen5_5	25	160	9	19	18	0.07 s	0.06 s	0.06 s	**0.02 s**	0.03 s	55.04 s
						5216 kb	5416 kb	5364 kb	**3980 kb**	160140 kb	258912 kb
queen6_6	36	290	12	27	25	2.7 s	2.74 s	2.73 s	**0.72 s**	*DNF*	*DNF*
						30348 kb	30288 kb	30356 kb	**5480 kb**	*DNF*	*DNF*
ship-ship-pp	30	77	4	9	8	436.97 s	443.85 s	487.61 s	96.01 s	52.95 s	**52.02 s**
						1027740 kb	1028028 kb	1141488 kb	**151560 kb**	303132 kb	302972 kb
water	32	123	6	11	9	27.01 s	26.82 s	26.78 s	**7.17 s**	12.25 s	12.24 s
						68604 kb	68696 kb	68512 kb	**14936 kb**	195840 kb	195632 kb

Table 1: Time and Space usage of exact treewidth algorithms

Because calculating exact treewidth is NP-hard, there can be an exponential number of words stored in a DAWG at any time, relative to the size of our original graph. Thus, merging the staging area is an operation, performed multiple times, which itself can require exponential time. This is in contrast to the tree-based approach, in which insertions are always $O(\log n)$ when n words are stored. Thus, even when there are an exponential number of values in the tree, lookups are at most linear in the size of our original graph.

6.2 DAWG Space Inefficiency

Memory usage of DAWGs was generally low, but almost always greater than the tree implementation. In terms of asymptotic complexity, when storing n words, the tree and DAWG versions each require $O(n)$ space. However, the complexity of storing an automaton may provide a much larger constant-factor for the DAWGs. Additionally, hash-tables were used in the DAWG implementation, as storing transitions in trees was prohibitively slow, and using arrays would have negated much of the space-efficiency, due to the sparseness of the automata. This also may have provided constant overhead for memory usage.

It is possible that, for larger examples, the compression provided by DAWGs would manifest and provide better space efficiency. However, this was impossible to test because of how prohibitively slow the DAWG implementation was.

6.3 Top-down inefficiency

Because the same TW values are calculated in top-down and bottom-up algorithms without optimizations, the main difference here is upper-bounds. With the bottom-up algorithm, in many cases there were few or no elements left in our TW set, allowing us to effectively skip many layers. In the case where no elements are left, we can conclude that the upper-bound is in fact the exact width. Once we've determined $TW(S) \geq UB$, we will never look at any sets including S again.

In the top-down version, the knowledge that $TW(S) \geq UB$ comes "too late:" once we know $TW(S)$, we have already fully evaluated the $TW(S')$ for every $S' \subseteq S$. Knowing that it exceeds our upper bound does not prune away search paths as it does in the bottom-up version.

When computing $TW(S \cup \{x\})$, we can eliminate the branch computing $TW(S)$ if we know that $UB(S) < LB(S \cup \{x\})$ or $LB(S) > UB(S \cup \{x\})$, since in either case we know that $TW(S)$ cannot contribute to the final value. However, fewer branches were pruned this way. Lower bounds were often much below the actual treewidth, and in practice the value of $|Q(S, x)|$ was usually the best estimate for the lower-bound on $TW(S \cup \{x\})$. This optimization was effective, but not enough to provide satisfactory performance.

6.4 Hybrid methods

The tests with the hybrid method provided very little interesting information. Generally, performance was very similar to top-down. When testing on larger examples, I found that if we allowed high memory usage by the bottom-up check, it would completely solve the instance, or the process as a whole would run out of memory. However, when restricted in memory usage, the speed benefits from the hybrid methods were negligible, and couldn't compute any of the examples for which top-down was too slow.

7 Future Work

Both bottom-up and top-down algorithms were implemented using Object-Oriented principles, and are based on classes which abstract over their storage methods. Thus, it is should be very easy to extend my code for future tests.

A potential future investigation would be exploring heuristics better suited to the top-down algorithms, since the top-down approach gives us more fine-grained control over our memory usage.

In the current bottom-up version, the absence of a subset S after completing the ith iteration means that $TW(S)$ cannot contribute to the final treewidth value. As a result, we are not allowed to "forget" values to save space as we are in the top-down version, since a potentially relevant value might be omitted. An alternate approach to future work would be devising a bottom-up algorithm which allows values to be dropped from our cache, allowing others to be re-computed in a time-space tradeoff. Such an algorithm would

potentially be able to use upper-bound herustics effectively, while avoiding the memory-usage bottleneck that the current bottom-up approach faces.

8 Conclusion

I have compared the DAWG and tree-based methods of computing treewidth bottom-up, as well as fixed-space and hybrid versions of computing it top-down. Tree-based methods appear to show the best performance on small graphs for both time and space. The undesirable time performance of the DAWG and top-down methods made it difficult to determine their space-efficiency on larger graph instances. The memory usage issues of exact algorithms in [5] remain the largest barriers to computing exact treewidth values for larger graphs.

References

[1] auto specifier (since C++11) - cppreference.com. `http://en.cppreference.com/w/cpp/language/auto`.

[2] "clang" C Language Family Frontend for LLVM. `http://clang.llvm.org/`.

[3] Range-based for loop (since C++11) - cppreference.com. `http://en.cppreference.com/w/cpp/language/range-for`.

[4] Ubuntu Manpage: time. `http://manpages.ubuntu.com/manpages/maverick/man1/time.1.html`.

[5] Hans L. Bodlaender, Fedor V. Fomin, Arie M. C. A. Koster, Dieter Kratsch, and Dimitrios M. Thilikos. On exact algorithms for treewidth. *ACM Trans. Algorithms*, 9(1):12:1–12:23, December 2012.

[6] Hans L. Bodlaender and Arie M.C.A. Koster. Treewidth computations i. upper bounds. *Information and Computation*, 208(3):259 – 275, 2010.

[7] Hans L. Bodlaender and Arie M.C.A. Koster. Treewidth computations ii. lower bounds. *Information and Computation*, 209(7):1103 – 1119, 2011.

[8] Johannes Bubenzer. Cycle-aware minimization of acyclic deterministic finite-state automata. *Discrete Appl. Math.*, 163:238–246, January 2014.

[9] Joey Eremondi. Github: treewidth-memoization, final-submission branch. `https://github.com/JoeyEremondi/treewidth-memoization/tree/final-submission`.

[10] J E Hopcroft and J D Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA, 1979.

[11] Thomas van Dijk, Jan-Pieter van den Heuvel, and Wouter Slob. Treewidth.com: libtw. `http://treewidth.com/treewidth/index.html`.