On exact algorithms for treewidth*

Hans L. Bodlaender[1], Fedor V. Fomin[2], Arie M.C.A. Koster[3], Dieter Kratsch[4], and Dimitrios M. Thilikos[5]

[1] Institute of Information and Computing Sciences, Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, the Netherlands, hansb@cs.uu.nl

[2] Department of Informatics, University of Bergen, N-5020 Bergen, Norway, fomin@ii.uib.no

[3] Konrad-Zuse-Zentrum für Informationstechnik Berlin, Takustraße 7, D-14195 Berlin, Germany, koster@zib.de

[4] LITA, Université Paul Verlaine - Metz, 57045 Metz Cedex 01, France, kratsch@univ-metz.fr

[5] Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, Edifici Ω, c/ Jordi Girona 1-3, E-08034 Barcelona, Spain, sedthilk@lsi.upc.edu

Abstract. We give experimental and theoretical results on the problem of computing the treewidth of a graph by exact exponential time algorithms using exponential space or using only polynomial space. We first report on an implementation of a dynamic programming algorithm for computing the treewidth of a graph with running time $O^*(2^n)$. This algorithm is based on the old dynamic programming method introduced by Held and Karp for the Traveling Salesman problem. We use some optimizations that do not affect the worst case running time but improve on the running time on actual instances and can be seen to be practical for small instances. However, our experiments show that the space used by the algorithm is an important factor to what input sizes the algorithm is effective. For this purpose, we settle the problem of computing treewidth under the restriction that the space used is only polynomial. In this direction we give a simple $O^*(4^n)$ algorithm that requires *polynomial* space. We also prove that using more refined techniques with balanced separators, Treewidth can be computed in $O^*(2.9512^n)$ time and polynomial space.

1 Introduction

The use of treewidth in several application areas requires efficient algorithms for computing the treewidth and optimal width tree decompositions of given graphs.

* This research was partially supported by the project *Treewidth and Combinatorial Optimization* with a grant from the Netherlands Organization for Scientific Research NWO and by the Research Council of Norway and by the DFG research group "Algorithms, Structure, Randomness" (Grant number GR 883/9-3, GR 883/9-4). The research of the last author was supported by the Spanish CICYT project TIN-2004-07925 (GRAMMARS).

In the past years, a large number of papers appeared studying the problem to determine the treewidth of a graph, including both theoretical and experimental results, see e.g., [4] for an overview. Since the problem is NP complete [1], there is a little hope in finding an algorithm which can determine the treewidth of a graph in polynomial time. There are several exponential time (exact) algorithms known in the literature for the treewidth problem. (See the surveys [10, 21] for an introduction to the area of exponential algorithms.) Arnborg et al. [1] gave an algorithm that tests in $O(n^{k+2})$ time if a given graph has treewidth at most k. It is not hard to observe that the algorithm runs for variable k in $O^*(2^n)$ time[1]. See also [18]. In 2004, Fomin et al. [11] presented an $O(1.9601^n)$ algorithm to compute the treewidth based on minimal separators and potential maximal cliques of graphs, using the paradigms introduced by Bouchitté and Todinca [7, 6]. The analysis of the algorithm of Fomin et al. from [11] was improved by Villanger [20], who showed that the treewidth of a graph can be computed in $O(1.8899^n)$ time. While the algorithms from [11, 20] provide the best known running time, they are based on computations of potential maximal cliques and are difficult to implement.

In this paper we try another approach to compute the treewidth, which seems to be much more suitable for implementations. While TREEWIDTH is usually formulated as the problem to find a tree decomposition of minimum width, it is possible to formulate it as a problem to find a linear ordering of (the vertices of) the graph such that a specific cost measure of the ordering is as small as possible. Several existing algorithms and heuristics for treewidth are based on this linear ordering characterization of treewidth, see e.g., [2, 8, 12]. In this paper, we exploit this characterization again, and a lesser known property of the characterization. Thus, we can show that an old dynamic programming method, introduced by Held and Karp for the TRAVELING SALESMAN problem [15] in 1962 can be adapted and used to compute the treewidth of given graphs. Suppressing polynomial factors, time and space bounds of the algorithm for treewidth is the same as that of the algorithm of Held and Karp for TRAVELING SALESMAN: $O^*(2^n)$ running time and $O^*(2^n)$ space. The Held-Karp algorithm tabulates some information for pairs (S, v), where S is a subset of the vertices, and v is a vertex (from S); a small variation of the scheme allows us to save a factor $O(n)$ on the space for the problems considered in this paper: we tabulate information for all subsets $S \subseteq V$ of vertices.

We have carried out experiments that show that the method works well to compute the treewidth of graphs of size up to around forty to fifty. For larger graphs, the space requirements of the algorithm appear to be the bottleneck. Thus, this raises the question: are there polynomially space algorithms to compute the treewidth having running time of the form $O^*(c^n)$ for some constant c? In this paper we answer this question in the affirmative. We show that there is an algorithm to compute the treewidth that uses $O^*(4^n)$ time and only polynomial

[1] We sometimes use O^*-notation which is a modified O-notation introduced by Woeginger [21] suppressing all polynomially bounded factors.

space. It uses a simple recursive divide-and-conquer technique and is similar to the polynomial space algorithm of Gurevich and Shelah [14] for TSP.

Finally, we further provide theoretical results improving upon the running time for the polynomial space algorithm for TREEWIDTH. Using balanced separators, we obtain an algorithm for TREEWIDTH that uses $O^*(2.9512^n)$ time and polynomial space. As we expect that in practical cases, the algorithm will use too much time, we do not provide an experimental evaluation of the algorithm.

2 Algorithms for Treewidth

2.1 Definitions

We assume the reader to be familiar with standard notions from graph theory. Throughout this paper, $n = |V|$ denotes the number of vertices of graph $G = (V, E)$. A graph $G = (V, E)$ is *chordal*, if every cycle in G of length at least four has a chord, i.e., there is an edge connecting two non-consecutive vertices in the cycle. A *triangulation* of a graph $G = (V, E)$ is a graph $H = (V, F)$ that contains G as subgraph ($F \subseteq E$) and is chordal. $H = (V, F)$ is a *minimal triangulation* of $G = (V, E)$ if H is a triangulation of G and there does not exist a triangulation $H' = (V, F')$ of G with H' a proper subgraph of H.

Definition 1. *A* tree decomposition *of a graph $G = (V, E)$ is a pair $(\{X_i \mid i \in I\}, T = (I, F))$ with $\{X_i \mid i \in I\}$ a collection of subsets of V, called* bags*, and $T = (I, F)$ a tree, such that*

- *for all $v \in V$, there exists an $i \in I$ with $v \in X_i$,*
- *for all $\{v, w\} \in E$, there exists an $i \in I$ with $v, w \in X_i$,*
- *for all $v \in V$, the set $I_v = \{i \in I \mid v \in X_i\}$ forms a connected subgraph (subtree) of T.*

The width *of tree decomposition $(\{X_i \mid i \in I\}, T = (I, F))$ equals $\max_{i \in I} |X_i| - 1$. The* treewidth *of a graph G, $\mathbf{tw}(G)$, is the minimum width of a tree decomposition of G.*

2.2 Treewidth as a Linear Ordering Problem

It is well known that treewidth can be formulated as a linear ordering problem, and this is exploited in several algorithms for treewidth, see e.g., [2, 12, 8, 9].

A *linear ordering* of a graph $G = (V, E)$ is a bijection $\pi : V \to \{1, 2, \ldots, |V|\}$. For a linear ordering π and $v \in V$, we denote by $\pi_{<,v}$ the set of vertices that appear before v in the ordering: $\pi_{<,v} = \{w \in V \mid \pi(w) < \pi(v)\}$. Likewise, we define $\pi_{\leq,v}$, $\pi_{>,v}$, and $\pi_{\geq,v}$. A linear ordering π of G is a *perfect elimination scheme*, if for each vertex, its higher numbered neighbors form a clique, i.e., for each $i \in \{1, 2, \ldots, |V|\}$, the set $\{\pi^{-1}(j) \ : \ \{\pi^{-1}(i), \pi^{-1}(j)\} \in E \wedge j > i\}$ is a clique. A graph has a perfect elimination scheme, if and only if it is chordal, see [13, Chapter 4].

For arbitrary graphs G, a linear ordering π defines a triangulation H of G that has π as perfect elimination scheme. The *triangulation with respect to* π of G is built as follows: first, set $G_0 = G$, and then for $i = 1$ to n, G_i is obtained from G_{i-1} by adding an edge between each pair of non adjacent higher numbered neighbors of $\pi^{-1}(i)$. One can observe that the resulting graph $H = G_n$ is chordal, has π as perfect elimination scheme, and contains G as subgraph.

For our algorithms, we want to avoid working with the triangulation explicitly. The following predicate allows us to 'hide' the triangulation. For a linear ordering π, and two vertices $v, w \in V$, we say $P_\pi(v, w)$ holds, if and only if there is a path $v, x_1, x_2, \ldots, x_r, w$ from v to w in G, such that for each i, $1 \leq i \leq r$, $\pi(x_i) < \pi(v)$, and $\pi(x_i) < \pi(w)$. In other words, $P_\pi(v, w)$ is true, if and only if there is a path from v to w such that all internal vertices are before v and w in the ordering π. Note that the definition implies that $P_\pi(v, w)$ is always true when $v = w$ or when $\{v, w\} \in E$.

With $R_\pi(v)$, we denote the number of higher numbered vertices $w \in V$ for which $P_\pi(v, w)$ holds, i.e., $R_\pi(v) = |\{w \in V \mid \pi(w) > \pi(v) \wedge P_\pi(v, w)\}|$. The proof of the following proposition is an immediate consequence of a lemma of Rose et al. [17]. (See also [3, 8, 9].)

Proposition 1. *Let $G = (V, E)$ be a graph, and k a non-negative integer. The treewidth of G is at most k iff there is a linear ordering π of G, such that for each $v \in V$, $R_\pi(v) \leq k$.*

2.3 A Dynamic Programming Algorithm for Treewidth

The results of this section are based on the observation that the value $R_\pi(v)$ only depends on v, G, and the set of vertices left of v in π.

Let for sets S, Q, $Q \subseteq S$, $\Pi(S)$ be the set of all permutations of S, and $\Pi(S, Q)$ be the set of all permutations of S that end with the vertices in Q. For a set of vertices $S \subseteq V$ and a vertex $v \in V - S$, we define

$$Q(S, v) = |\{w \in V - S - \{v\} \mid \text{there is a path from } v \text{ to } w \text{ in } G[S \cup \{v, w\}]|\}$$

$$TW(S) = \min_{\pi \in \Pi(S)} \max_{v \in S} Q(\pi_{<,v}, v).$$

Let us note that $Q(S, v)$ can be computed in time $O(n + m)$ by checking for each $w \in V - S - \{v\}$ whether w has a neighbor in the component of $G[S \cup \{v\}]$ containing v, and that

$$\mathbf{tw}(G) = \min_{\pi \in \Pi(V)} \max_{v \in V} R_\pi(v) = \min_{\pi \in \Pi(V)} \max_{v \in V} Q(\pi_{<,v}, v) = TW(V)$$

Lemma 1. *For any graph $G = (V, E)$, and any set of vertices $S \subseteq V$, $S \neq \emptyset$,*

$$TW(S) = \min_{v \in S} \max\{TW(S - \{v\}), Q(S - \{v\}, v)\}.$$

Theorem 1. *The treewidth of a graph on n vertices can be determined in $O^*(2^n)$ time and $O^*(2^n)$ space.*

Proof. By Lemma 1, we almost directly obtain a Held-Karp-like dynamic programming algorithm for the problem. In order of increasing sizes, we compute for each $S \subseteq V$, $TW(S)$ using Lemma 1. The algorithm uses $O^*(2^n)$ time, as we do polynomially many steps per subset of V. The algorithm also keeps all subsets of V and thus uses $O^*(2^n)$ space. □

2.4 A Recursive Algorithm for Treewidth

For vertex subsets $L, S \subseteq V, S \cap L = \emptyset$ of a graph $G = (V, E)$ we define

$$\mathbf{t}(L, S) = \min_{\pi \in \Pi(S)} \max_{v \in S} Q(L \cup \pi_{<,v}, v).$$

The intuition behind $\mathbf{t}(L, S)$ is as follows: we investigate the resulting cost of the 'best' ordering of the vertices in S, assuming that all vertices in L are left of all vertices in S, and all vertices in $V - (L \cup S)$ are right of all vertices in S. We observe that if $S = \{v\}$, then $\mathbf{t}(L, S) = Q(L, v)$. Also, by definition, $\mathbf{t}(\emptyset, S) = TW(S)$ and therefore $\mathbf{tw}(G) = \mathbf{t}(\emptyset, V)$.

Lemma 2. *Let $G = (V, E)$ be a graph, let $S \subseteq V$, $|S| \geq 2$, $L \subseteq V$, $L \cap S = \emptyset$, $k = \lfloor |S|/2 \rfloor$. Then*

$$\mathbf{t}(L, S) = \min_{S' \subseteq S, |S'| = k} \max \{\mathbf{t}(L, S'), \mathbf{t}(L \cup S', S - S')\}$$

Theorem 2. *The treewidth of a graph on n vertices can be determined in $O^*(4^n)$ time and polynomial space.*

Proof. Lemma 2 is used to obtain Algorithm 1. This algorithm computes $\mathbf{t}(L, S)$ recursively. Algorithm 1 computes the treewidth of the graph G when calling Recursive-Treewidth(G,$\emptyset$,V). It is not hard to show that the algorithm uses polynomial space and $O^*(4^n)$ time. □

Algorithm 1 Recursive-Treewidth(Graph G, Vertex Set L, Vertex Set S)

```
if |S|=1 then
    Suppose S = {v}.
    return Q(L, v)
end if
Set Opt = ∞.
for all sets S' ⊆ S, |S'| = ⌊|S|/2⌋ do
    Compute v1 = Recursive-Treewidth(G, L, S');
    Compute v2 = Recursive-Treewidth(G, L ∪ S', S − S');
    Set Opt = min {Opt, max {v1, v2}};
end for
return Opt
```

3 Experimental Results

In this section, we comment on the experiments we have carried out for the dynamic programming algorithm for computing the treewidth of a given graph.

For practical considerations, we use a scheme that is slightly different than that of Theorem 1. We can note that it is not useful to perform computations with sets S for which $TW(S)$ is larger or equal than a known upper bound up on the treewidth of G: these cannot lead to a smaller bound on the treewidth of G. Thus, in order to save time and space in practice, we avoid handling some of such S. We compute collections $TW_1, TW_2, \ldots, TW_n$. Each collection TW_i ($1 \leq i \leq n$) contains pairs $(S, TW(S))$ with $|S| = i$. The collection for sets of size $i > 1$ is built as follows: for each pair $(S, r) \in TW_{i-1}$ and each $x \in V - S$, we compute $r' = \max(r, Q(S, x))$. If $r' < up$, then we check if there is a pair $(S \cup \{x\}, t)$ in TW_i for some t, and if so, replace it by $(S \cup \{x\}, \min(t, r'))$. If no such pair exists in TW_i, we insert $(S \cup \{x\}, r')$ in TW_i.

In our implementation, we use two additional optimizations that appeared to give significant savings in time and memory consumption. The following simple lemma gives the first idea.

Lemma 3. *Let $G = (V, E)$ be a graph, and let $S \subseteq V$. The treewidth of G is at most* $\max\{TW(S), n - |S| - 1\}$.

Lemma 3 shows correctness of the following rule that was used in the implementation: we keep an upper bound up for the treewidth of G, initially set by the user or set to $n-1$. Each time, we get a pair (S, r) in a collection TW_i, either by insertion, or by replacement of an existing pair, we set the upper bound up to the minimum of up and $n - |S| - 1 = n - i - 1$. Moreover, when handling a pair (S, r) from TW_{i-1}, it is first checked if r is smaller than up; if not, then this pair cannot contribute to an improvement of the upper bound, and hence is skipped. Our second optimization is stated in Lemma 4.

Lemma 4. *Let $G = (V, E)$ be a graph of treewidth k. Given a subset $Q \subset V$ inducing a clique in G, there exists a linear ordering π with $R_\pi(v) \leq k$ and $\pi(v) \geq |V| - |Q| + 1$ for all $v \in Q$.*

By Lemma 4, we can restrict the sets S to elements from $V \setminus Q$ for some clique Q; in particular for the maximum clique. Although it is NP-hard to compute the maximum clique in a graph, it can be computed extremely fast for the graphs considered. In our program, we use a simple combinatorial branch-and-bound is used to compute all maximum cliques. It recursively extends a clique by all candidate vertices once.

The algorithm was implemented in C++, using the Boost graph library, as part of the Treewidth Optimization Library TOL, a package of algorithms for the treewidth of graphs. The package includes preprocessing, upper bound, and lower bound algorithms for treewidth. Experiments were carried out on a number of graphs taken from applications; several were used in other experiments. See [19] for the used graphs, information on the graphs, and other results of experiments to compute the treewidth. The experiments were carried out on a Sun computer with 4 AMD Dualcore Opteron 875, 2.2 GHz processor and at most 20 GB of internal memory available. The program did not use parallelism.

In Table 1 the results of our experiments on a number of graphs are reported. Besides instance name, number of vertices, number of edges, and the computed treewidth, we report on the CPU time in seconds and the maximum number of sets (S, r), considered at once, $\max |TW| = \max_{i=0,\ldots,n} |TW_i|$ in a number of cases. First, we report on the CPU time and maximum number of sets for the case that no initial upper bound *up* is exploited. Next, we report on the case where we use an initial upper bound, displayed in the column *up*. The last two columns report on the experiments in which the algorithm is advanced by both an initial upper bound *up* and a maximum clique Q.

In several instances reported in [19], the best bound obtained from a few upper bound heuristics, and the lower bound obtained by the LBP+(MMD+) heuristic match, and then we have obtained in a relatively fast way an exact bound on the treewidth of the instance graph. In other cases, these bounds do not match. Then, when the graph is not too large, the dynamic programming algorithm can be of good use.

A nice example is the celar03 graph. This graph has 200 vertices and 721 edges. A combination of different preprocessing techniques yield an equivalent instance celar03-pp-001 which has 38 vertices and 238 edges. Existing upper bound heuristics gave a best upper bound of 15, while the lower bound of the LBP+(MMD+) heuristic was 13. With the dynamic programming algorithm with 15 as input for an upper bound, we obtained the exact treewidth of 14 for this graph, and hence also for celar03.

The algorithm can also be used as a lower bound heuristic: give the algorithm as 'upper bound' a conjectured lower bound ℓ: when it terminates, it either has found the exact treewidth, or we know that ℓ is indeed a lower bound for the treewidth of the input graph. In a few cases, we could thus increase the lower bound for the treewidth of considered instances, e.g., for the treewidth of the queen8-8 graph (the graph modeling the possible moves of a queen on an 8 by 8 chessboard) the lower bound could be improved from 27 to 35.

For larger graphs, the above idea can be combined by an idea exploited earlier in various papers. Given a graph G and a minor G' of G, $\mathbf{tw}(G') \leq \mathbf{tw}(G)$. In [5, 12, 16], a lower bound on $\mathbf{tw}(G')$ is computed to obtain a lower bound for G. With the dynamic programming algorithm, we can compute $\mathbf{tw}(G')$ exactly to obtain a lower bound for $\mathbf{tw}(G)$. For the 1024 vertices graph pignet2-pp, we have generated a sequence of minors by repeatedly contracting a minimum degree vertex with a neighbor with least number of common neighbors (see [5]). Figure 1 shows the treewidth (right y-scale) for the minors with 70 to 79 vertices. Moreover, the maximum number of sets for three different upper bounds is reported (left y-scale, logarithmic). If the used upper bound is less than or equal to the treewidth, no feasible solution is found in the end. The best known lower bound for pignet2-pp is increased from 48 to 59 by the treewidth of the

instance	$\|V\|$	$\|E\|$	**tw**	no *up*, no Q CPU	max $\|TW\|$	*up*	with *up*, no Q CPU	max $\|TW\|$	with *up*, w Q CPU	max $\|TW\|$
myciel3	11	20	5	0.00	240	5	0.00	35	0.00	21
myciel4	23	71	10	7.64	296835	10	0.14	4422	0.12	4064
queen5-5	25	160	18	0.15	18220	18	0.02	944	0.02	392
queen6-6	36	290	25	36.43	2031716	26	1.16	18872	0.36	6994
queen7-7	49	476	35	-	-	37	1012.12	96517095	248.03	24410915
pathfinder-pp	12	43	6	0.00	107	6	0.00	1	0.00	1
oesoca+-pp	14	75	11	0.00	48	11	0.00	5	0.00	5
fungiuk	15	36	4	0.07	4713	4	0.00	4	0.00	4
weeduk	15	49	7	0.02	2906	7	0.00	35	0.00	35
munin-kgo-pp	16	41	5	0.11	6892	5	0.00	2	0.00	2
wilson	21	27	3	14.44	350573	3	0.08	2412	0.06	2342
water-pp	22	96	9	1.60	77286	10	0.04	816	0.01	475
oow-trad-pp	23	54	6	42.91	1065120	6	0.09	2953	0.05	1895
barley-pp	26	78	7	349.31	6110572	7	0.61	13597	0.29	7971
oow-bas	27	54	4	1579.38	19937301	4	0.01	303	0.00	111
oow-solo-pp	27	63	6	1059.50	17048070	6	0.91	22484	0.30	9426
ship-ship-pp	30	77	8	-	-	9	291.20	3062863	50.75	820910
water	32	123	9	-	-	10	12.59	127545	1.53	25874
oow-trad	33	72	6	-	-	6	129.55	1162650	14.55	178846
mildew	35	80	4	-	-	4	2.98	33045	0.35	5431
mainuk	48	198	7	-	-	8	-	-	2251.97	11748147
celar03-pp-001	38	238	14	-	-	15	121.29	911918	4.36	55504

Table 1. Experimental results for some DIMACS vertex coloring graphs, some probabilistic networks and celar03-pp-001

79 vertex-minor. Figure 1 shows one more time the impact of the upper bound on the memory consumption (and time consumption) of the algorithm.

4 Improved Polynomial Space Algorithms for Treewidth

In this section, we give a faster exponential time algorithm with polynomial space for TREEWIDTH. The algorithm is based on results of earlier sections combined with techniques based upon balanced separators.

Lemma 5. *Let $S \subseteq V$ be a set of vertices, such that the treewidth of G is equal to the treewidth of the graph $G' = (V, E \cup \{\{v, w\} \mid v, w \in S, v \neq w\})$ obtained from G by turning S into a clique. Then there is a linear ordering $\pi \in \Pi(V)$, that ends with a permutation of S, with $\mathbf{tw}(G) = \max_{v \in V} R_\pi(v)$, Moreover, $\mathbf{tw}(G) = \max\{TW(V-S), TWR(V-S, S)\}$.*

Lemma 6. *Let $G = (V, E)$ be a graph with treewidth at most k. There is a set $S \subseteq V$ with $|S| = k+1$, such that each connected component of $G[V-S]$ contains at most $(|V|-k)/2$ vertices and the graph $G' = (V, E \cup \{\{v, w\} \mid v, w \in S, v \neq w\})$ obtained from G by turning S into a clique has treewidth at most k.*

Lemma 6 is a variant on a folklore result on balanced separators in graphs of small treewidth. Lemmas 5 and 6 are used to show Lemma 7.

Lemma 7. *Let $G = (V, E)$ be a graph with treewidth at most k. Let $k + 1 \leq r \leq n$. There is a set $W \subseteq V$, with $|W| = r$, such that each connected component of $G[V - W]$ contains at most $(|V| - r + 1)/2$ vertices, and $\mathbf{tw}(G) = \max\{TWR(\emptyset, V - W), TWR(V - W, W)\}$.*

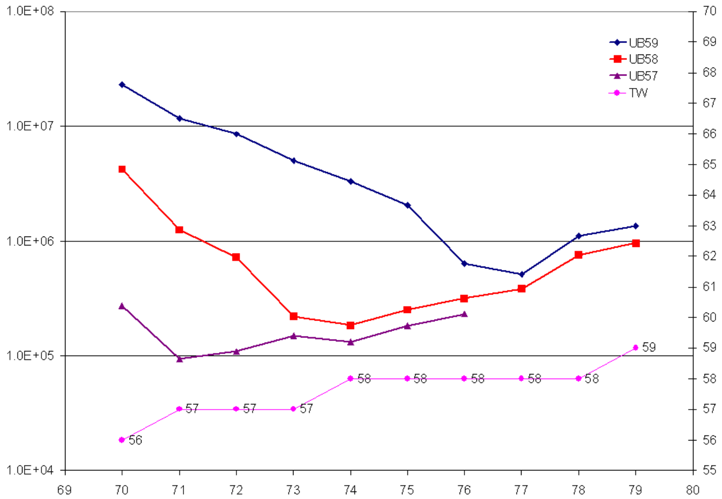

Fig. 1. Maximum number of subsets S during algorithm for different upper bounds.

For a graph $G = (V, E)$, and a set W, let $G^+[W]$ be the fill-in graph, obtained by eliminating the vertices in $V - W$, i.e., $G^+[W] = (W, F)$, with for all $v, w \in W$, $v \neq w$, we have that $\{v, w\} \in F$, if and only if there is a path from v to w that uses only vertices in $V - W$ as internal vertices. The next lemma formalizes the intuition behind $TWR(V - W, W)$: when computing $TWR(V - W, W)$, we look for the best ordering of the vertices in W, after all vertices in $V - W$ are eliminated — i.e., in the graph $G^+[W]$.

Lemma 8. *Let $G = (V, E)$ be a graph, and $W \subseteq V$ a set of vertices. Then $\mathbf{tw}(G^+[W]) = TWR(V - W, W)$.*

Lemma 9. *Let $G = (V, E)$ be a graph, and let $S = S_1 \cup S_2 \subseteq V$. Suppose $S_1 \cap S_2 = \emptyset$, and that there is no edge between a vertex in S_1 and a vertex in S_2. Then $TW(S) = \max\{TW(S_1), TW(S_2)\}$.*

The lemmas above are summarized in the following result, which gives a main idea of the improved recursive algorithm.

Corollary 1. *Let $G = (V, E)$ be a graph, and let k, r be integers, $0 \leq k < r \leq |V|$. The treewidth of G is at most k, if and only if there is a set of vertices $S \subseteq V$, with $|S| = r$, such that each connected component of $G[V - S]$ contains at most $(|V| - r + 1)/2$ vertices, for each connected component W of $G[V - S]$, $TWR(\emptyset, W) \leq k$, and the treewidth of $G^+[S]$ is at most k.*

We now present the main result of this section.

Theorem 3. *The treewidth of a graph G on n vertices can be computed in polynomial space and time $O^*(2.9512^n)$.*

Proof. We describe a decision algorithm for treewidth: given a graph G, and an integer k, it decides whether the treewidth of G is at most k. Of course, an algorithm that, given a graph G, computes $\mathbf{tw}(G)$ can be constructed at the cost of an additional multiplicative factor $O(\log n)$. Correctness of the algorithm follows from Corollary 1. Let $\gamma = 0.4203$.

The algorithm works as follows. If $|V| \leq k+1$, then the treewidth of G is at most $|V| - 1 \leq k$, so the algorithm returns true.

Otherwise, the algorithm checks if $k \leq 0.25 \cdot |V|$ or $k \geq \gamma \cdot |V|$. If this is the case, then we search for a set S, as implied by Corollary 1 when we take $r = k+1$. I.e., we enumerate all sets S of size $k+1$. For each such S, we check if all connected components of $G = (V, E)$ have size at most $(|V| - |S| + 1)/2$. If so, we use the algorithm of Theorem 2 (Algorithm 1 Recursive-Treewidth) to compute for each connected component W the value $TWR(W, \emptyset)$. If for each such component W, $TWR(\emptyset, W)$, then the algorithm returns true: as $G^+(W)$ has $k+1$ vertices, its treewidth is trivially at most k, and hence all conditions of Corollary 1 are fulfilled, so G has treewidth at most k. If no set S of size $k+1$ yields true, then the algorithm returns false.

The remaining case is that $0.25 \cdot |V| < k < \gamma \cdot |V|$. Now, we search for a set S as implied by Corollary 1 when taking $r = \lceil \gamma \cdot |V| \rceil$. I.e., we enumerate all sets S of size $r = \lceil \gamma \cdot |V| \rceil$. For each we check if all connected components W of $G[V-S]$ have size at most $(|V| - r + 1)/2$. If so, we use Algorithm 1 Recursive-Treewidth for deciding if all connected components W of $G[V - S]$ fulfill $TWR(\emptyset, W) \leq k$. We also recursively call the algorithm on $G^+(W)$ to decide if this graph has treewidth at most k. If all these checks succeed, the algorithm returns true. If no S of size r made the algorithm return true, the algorithm returns false.

We now analyze the running time of the algorithm. Write $\alpha = k/|V|$.

We start with analyzing the case where $k \leq 0.25 \cdot |V|$ or $\gamma \cdot |V| \leq k$. We have $\alpha \leq 0.25$ or $\alpha \geq \gamma$. The number of subsets of size $\alpha \cdot n$ of a set of size n is known to be of size $O^*((\alpha^{-\alpha} \cdot (1-\alpha)^{\alpha-1})^n)$. Write $f(\alpha) = \alpha^{-\alpha} \cdot (1-\alpha)^{\alpha-1} \cdot 2^{1-\alpha}$. Each connected component W of $G[V - S]$ for which the algorithm calls Recursive-Treewidth has size at most $(|V| - \alpha \cdot |V| + 1)/2$, thus we use at most $O^*(4^{(|V| - \alpha \cdot |V| + 1)/2}) = O^*(2^{(1-\alpha)|V|}$ time for one such component. Thus, the total time in this case is bounded by $O^*(f(\alpha)^n))$. f monotonically increases in the interval $(0, \frac{1}{3})$, and monotonically decreases in the interval $(\frac{1}{3}, 1)$. As $f(0.25) < 2.9512$, and $f(\gamma) < 2.9512$, we have for all α with $0 < \alpha \leq 0.25$ or $\gamma \leq \alpha < 1$, that $f(\alpha) < 2.9512$, and hence that the algorithm uses $O^*(2.9512^n)$ time.

We now look at the case where $0.25 \cdot |V| < k < \gamma \cdot |V|$, i.e., where $0.25 < \alpha < \gamma$. As in the previous case, the time for all computations of $TWR(\emptyset, W)$ for all connected components of $G[V - S]$ over all sets $S \subseteq V$ of size r is bounded by $O^*((\gamma^{-\gamma} \cdot (1-\gamma)^{\gamma-1} \cdot 2^{1-\gamma})^n = O^*(f(\gamma)^n) = O^*(f(0.4203)^n) < O^*(2.9512^n)$.

We have to add to this time the total time over all recursive calls to the algorithm with graphs of the form $G^+(S)$. Note that the recursion depth is at most 1: in the recursion, the value of k is unchanged, while we now have a graph with $|S| = \lceil \gamma \cdot |V| \rceil$ vertices. So, in the recursive call, $k > 0.25 \cdot |V| > \gamma \cdot |S|$, and the algorithm executes the first case. From the analysis above, it follows that each recursive call of Improved-Recursive-Treewidth on a graph $G^+[S]$ costs $O^*((\beta^{-\beta} \cdot (1-\beta)^{\beta-1} \cdot 2^{1-\beta})^{\gamma \cdot n})$ time, with $\beta = \alpha/\gamma$. Write

$$g(\alpha) = \gamma^\gamma \cdot (1-\gamma)^{\gamma-1} \cdot \left((\alpha/\gamma)^{-\alpha/\gamma} \cdot (1-\alpha/\gamma)^{\alpha/\gamma-1} \cdot 2^{1-\alpha/\gamma}\right)^\gamma$$

As there are $O^*((\gamma^\gamma \cdot (1-\gamma)^{\gamma-1})^n)$ vertex sets $S \subseteq V$ of size γn, the total time of all calls of Improved-Recursive-Treewidth with graphs of the form $G^+[S]$ is bounded by $O^*(g(\alpha)^n)$. On the interval $[0.25, \gamma]$, the function g is monotonically decreasing, with $g(0.25) < 2.9511$. Thus, the total time over all calls of Improved-Recursive-Treewidth for graphs $G^+[S]$ is bounded by $O^*(2.9511^n)$, and the total time of the algorithm is bounded by $O^*(2.9512^n)$. □

We conjecture that with a more detailed analysis and more levels of recursion, small improvements to the running time are possible.

5 Conclusions

In this paper, we have given dynamic programming and recursive algorithms to compute the treewidth. Similar results can be obtained for related graph parameters, like minimum fill-in. The dynamic programming algorithm for the treewidth problem has been implemented; for small instances (slightly below 50 vertices), the algorithm appears to be practical. On a more theoretical side, we gave the first exponential time algorithms for TREEWIDTH with a running time of the type $O^*(c^n)$ for some constant c that use polynomial space and we reduced the running time of the algorithm with polynomial space to $O^*(2.9512.)$.

A comparison of the dynamic programming algorithm for TREEWIDTH with other algorithms, (e.g., a branch and bound algorithm as in [12] or the algorithm of Shoikhet and Geiger [18]) would be very interesting.

References

1. S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Alg. Disc. Meth.*, 8:277–284, 1987.
2. E. H. Bachoore and H. L. Bodlaender. New upper bound heuristics for treewidth. In S. E. Nikoletseas, editor, *Proceedings of the 4th International Workshop on Experimental and Efficient Algorithms WEA 2005*, pages 217–227. Springer-Verlag, Lecture Notes in Computer Science, vol. 3503, 2005.
3. H. L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theor. Comp. Sc.*, 209:1–45, 1998.
4. H. L. Bodlaender. Discovering treewidth. In P. Vojtáš, M. Bieliková, and B. Charron-Bost, editors, *SOFSEM 2005: Theory and Practive of Computer Science: 31st Conference on Current Trends in Theory and Practive of Computer Science*, pages 1–16. Springer-Verlag, Lecture Notes in Computer Science 3381, 2005.

5. H. L. Bodlaender, A. M. C. A. Koster, and T. Wolle. Contraction and treewidth lower bounds. In S. Albers and T. Radzik, editors, *Proceedings 12th Annual European Symposium on Algorithms, ESA2004*, pages 628–639. Springer, Lecture Notes in Computer Science, vol. 3221, 2004.
6. V. Bouchitté and I. Todinca. Treewidth and minimum fill-in: Grouping the minimal separators. *SIAM J. Comput.*, 31:212–232, 2001.
7. V. Bouchitté and I. Todinca. Listing all potential maximal cliques of a graph. *Theor. Comp. Sc.*, 276:17–32, 2002.
8. F. Clautiaux, A. Moukrim, S. Négre, and J. Carlier. Heuristic and meta-heuristic methods for computing graph treewidth. *RAIRO Operations Research*, 38:13–26, 2004.
9. N. D. Dendris, L. M. Kirousis, and D. M. Thilikos. Fugitive-search games on graphs and related parameters. *Theor. Comp. Sc.*, 172:233–254, 1997.
10. F. V. Fomin, F. Grandoni, and D. Kratsch. Some new techniques in design and analysis of exact (exponential) algorithms. *Bulletin of the EATCS*, 87:47–77, 2005.
11. F. V. Fomin, D. Kratsch, and I. Todinca. Exact (exponential) algorithms for treewidth and minimum fill-in. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming, ICALP 2004*, pages 568–580, 2004.
12. V. Gogate and R. Dechter. A complete anytime algorithm for treewidth. In *Proceedings of the 20th Annual Conference on Uncertainty in Artificial Intelligence UAI-04*, pages 201–208, Arlington, Virginia, USA, 2004. AUAI Press.
13. M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.
14. Y. Gurevich and S. Shelah. Expected computation time for Hamiltonian path problem. *SIAM J. Comput.*, 16:486–502, 1987.
15. M. Held and R. Karp. A dynamic programming approach to sequencing problems. *J. SIAM*, 10:196–210, 1962.
16. A. M. C. A. Koster, T. Wolle, and H. L. Bodlaender. Degree-based treewidth lower bounds. In S. E. Nikoletseas, editor, *Proceedings of the 4th International Workshop on Experimental and Efficient Algorithms WEA 2005*, pages 101–112. Springer-Verlag, Lecture Notes in Computer Science, vol. 3503, 2005.
17. D. J. Rose, R. E. Tarjan, and G. S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comput.*, 5:266–283, 1976.
18. K. Shoikhet and D. Geiger. A practical algorithm for finding optimal triangulations. In *Proc. National Conference on Artificial Intelligence (AAAI '97)*, pages 185–190. Morgan Kaufmann, 1997.
19. Treewidthlib. http://www.cs.uu.nl/people/hansb/treewidthlib, 2004.
20. Y. Villanger. Improved exponential-time algorithms for treewidth and minimum fill-in. In *Proceedings of the 7th Latin American Theoretical Informatics Symposium (LATIN 2006)*, volume 3887 of *LNCS*, pages 800–811. Springer-Verlag, Berlin, 2006.
21. G. J. Woeginger. Exact algorithms for NP-hard problems: A survey. In *Combinatorial Optimization: "Eureka, you shrink"*, pages 185–207, Berlin, 2003. Springer Lecture Notes in Computer Science, vol. 2570.