Computing treewidth with LibTW

Thomas van Dijk
Jan-Pieter van den Heuvel
Wouter Slob

November 10, 2006

Abstract

There are many algorithms to compute an upperbound, a lowerbound or the exact treewidth of a graph. We have implemented a lot of upperbound and lowerbound heuristics and two exact algorithms (a Dynamic Programming and a Branch and Bound algorithm). This report compares the different kind of algorithms and shows that some algorithms are preferred.

From our results with the lowerbound algorithms we can conclude that the Least-C variant for Maximum Minimum Degree almost dominates the other algorithms. For the upperbounds, we conclude that Greedy-FillIn is best. TreewidthDP is quite fast on most of the tested graphs, but runs out of memory on large graphs. If TreewidthDP can not run with the available amount of memory one could use QuickBB, which is slower, but uses less memory.

We investigated the effects of the Memorization method on QuickBB suggested by Van Hoesel and found that it improved the algorithm with at least factor 15.

1 Introduction

Many NP-hard problems can be solved in polynomial time when the treewidth is bounded by a constant. Therefore, the need arises for fast ways to compute or approximate the treewidth and the matching tree decomposition. Unfortunately, Arnborg et al [1] proved that determining whether a graph G has a treewidth of at most k is NP-complete. There are heuristics for finding upperbounds and lowerbounds on the treewidth of a graph. And there are several (expensive) algorithms that compute the exact treewidth. However, it is not really clear which algorithm performs best.

To this end, we have written an experimentation framework to perform measurements on different algorithms. And we implemented a lot of algorithms which we compare in this report. The software, the LibTW library, is available under the LGPL on http://www.treewidth.com.

This report is organized as follows. In Section 2, we describe the experimental setup and something about the framework. A comparison between the different heuristics, both upperbound and lowerbound, can be found in Section 3. The results of experiments on and a comparison between exact algorithms are in Section 4. In section 5 we conclude with describing some improvements that can be made in the future.

2 Experimental setup

We ran our experiments with one test configuration: CentOS 4.4 (Linux) on a Dell Optiplex GX620. The machine has an Intel Pentium D 3.0 GHz Dual core CPU and 1024 MB RAM. Note that all implementations are single-threaded so we only use one core of the CPU. All algorithms were compiled and run with Sun Java 1.5.0_03.

3 Heuristics

Computing the treewidth of a graph is very expensive. This means that heuristics can be very useful. For example because a bound on the treewidth might be good enough, but also because they can be used to significantly speed up exact algorithms.

Another interesting situation arises when the lowerbounds and upperbounds are so good that they are equal. This seems to happen quite often on probabilistic networks: on 27 of the 58 graphs we tested for this some lowerbound was equal to some upperbound.

3.1 Lowerbounds

We have implemented several lowerbound algorithms, most of them with several variants. The basic list is as follows.

- Minimum Degree: the minimum degree over all vertices.
- Ramachandramurthi [7], also based on vertex degrees, but slightly smarter.
- Maximum Cardinality Search [2], based on triangulation algorithms.
- Maximum Minimum Degree [6], based on removing vertices.
- Minor-Min-Width [5], based on edge contractions.

Most of these algorithms are not entirely specific. Maximum Cardinality Search, for example, repeatedly selects a vertex of maximum weight and then changes some weights. But at any time there may be several vertices with the maximum weight. In particular, all vertices start out with weight zero and the result of the algorithm can vary wildly based on these choices. It is too expensive to branch on all such possibilities, but branching only on the first vertex to use gives nice results: it is not very expensive and, while it may miss the best value, it will often find a good value. We call this the All-Start variant.

All lowerbounds run pretty fast. For this reason we do not report on their runtimes; instead, we focus solely on the quality of the results. See Figures 1 and 2 for our results. (The raw data can be found in appendix A.) First of all, notice that the All-Start variants perform significantly better than the ordinary versions of the same algorithm. Also notice that the –Least-C and MinorMinWidth algorithms perform significantly better than any of the other algorithms. Because their runtime performance is similar to the others, we recommend always using one or both of these.

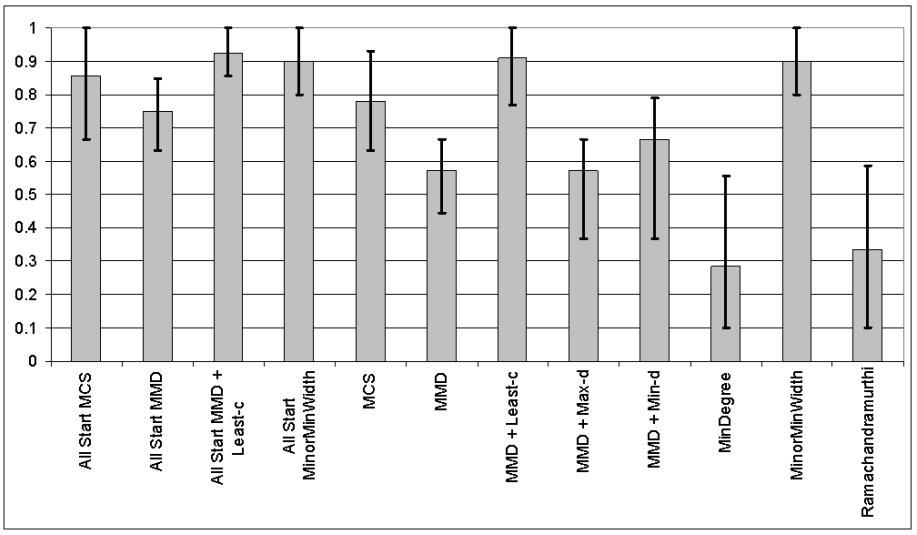

Figure 1: Lowerbounds: fraction of the actual treewidth. The bar gives the median value, the extensions are to first and third quartile. These results are for 31 assorted graphs from the TOL repository.

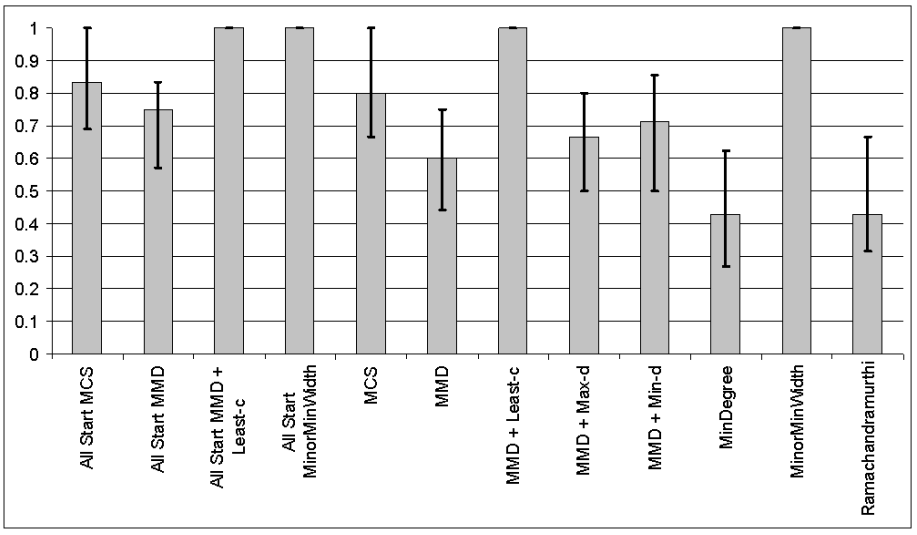

Figure 2: Lowerbounds: fraction of the best lowerbound found. The bar gives the median value, the extensions are to first and third quartile. These results are for 59 probabilistic networks from the TOL repository.

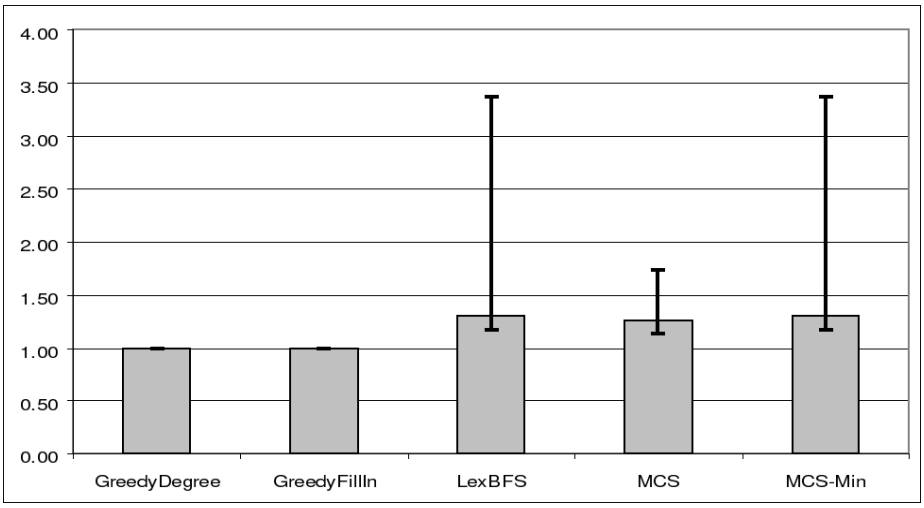

Figure 3: Upperbounds: fraction of the best upperbound found. The bar gives the median value, the extensions are to first and third quartile. These results are for 59 probabilistic networks from the TOL repository.

3.2 Upperbounds

The results from our upperbound implementations can be seen in Figure 3. (Again, the raw data can be found in appendix A.) For the same reasons as with the lowerbounds, we did not do runtime measurements.

Notice that GreedyDegree and GreedyFillIn are clearly the best. Actually, they were *always* at least as good as any of the others on the 58 probabilistic networks we tested on. On most graphs, GreedyDegree and GreedyFillIn give the same answer: this happened on 48 of the 58 test graphs. Sometimes –FillIn is better (this happened on the ten other graphs). It has occasionally happened that –Degree was better, but this didn't happen on our test graphs.

So again we have a clear winner: use GreedyFillIn.

4 Exact algorithms

There are several algorithms to compute the exact treewidth of a graph. We have implemented two kinds of exact algorithms, namely a Dynamic Programming algorithm (TreewidthDP) and a Branch and Bound algorithm (QuickBB).

4.1 TreewidthDP

As shown by Bodlaender et al. [4] treewidth can be seen as a Linear Ordering Problem. A linear ordering (permutation) defines a triangulation of the graph that has this ordering as a perfect elimination scheme. The triangulation with respect to a permutation 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.

The entries in the DP-table are the lowest treewidth computed so far when starting the permutation with a certain subset of the vertices. These values must be calculated with increasing size of the subset. The recursive formula for computing the treewidth is stated below. In this formula, S is a subset of the vertices V in G.

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

The treewidth of the whole graph is $TW(V)$. The Q-function is defined as follows.

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

For more detailed information, see [4].

In addition to the straightforward implementation, we've added two extras: upperbound and clique.

4.1.1 Implementation issues

By checking if a permutation so far yields a higher or equal treewidth than a known upperbound, we can skip this permutation because it will never lead to an improvement of the treewidth. The results of Bodlaender et al. [4] show that this significantly improves the running time and memory usage of the dynamic programming algorithm. During our implementation of this algorithm we also saw that this makes a lot of difference. If no upperbound is given, the algorithm starts with the trivial upperbound of $n - 1$.

As proved by Bodlaender et al. [4], for all cliques in the graph there always exists a permutation with optimal treewidth with the vertices of that clique as the last vertices to be eliminated. This means that we do not consider the vertices in a clique while DP-ing for the permutation.

For this to be as effective as possible, we choose the maximum clique. Finding the maximum clique is NP-hard, but for the graphs we are working on this can be computed very fast. We implemented this with a simple backtracking algorithm.

We've chosen to represent the subsets as BitSets. This is very efficient. All BitSets and the computed treewidth for that set are stored in a HashMap. That way we can find them in constant time. As we only need the computed treewidth of sets of size 1 smaller than the size of the current set, we can forget smaller sets to free up much needed memory.

4.1.2 Computational results

We have tested our implementation of the dynamic programming algorithm with our standard test configuration (see section 2). The results are displayed in Table 1. The upperbound was computed using the GreedyFillIn algorithm. The column CPU denotes the average running time over 10 runs, in milliseconds. However, the CPU times for TOL are not averages; we only ran it once and it reports the running time with a precision of 10 ms. The Mem column denotes the

memory usage in megabytes (rounded up). TOL computes this itself and for our implementation we did binary search on the minimum amount of heap memory the virtual machine required to complete the algorithm. This means that the memory usage reported is the absolute minimum needed by our implementation; giving the virtual machine more heap space might improve performance by stressing the garbage collector less.

On most graphs we tested the algorithm with and without Bodlaender's clique trick. We did not run without cliques on graphs that already swapped out or took a long time to compute or used a lot of memory. Where there is 'swapped out,' the operating system started using the hard disk as memory and CPU usage dropped to 1%. We then stopped the experiment.

On most of the graphs, the TOL implementation is somewhat faster (average ratio of 1.43), but on the queen7_7 graph, our implementation is a lot faster. The implementations differ in a lot of details and a part of the performance difference can be explained by the different programming language. TOL was written in C++ and our implementation in Java.

Our implementation sometimes uses more memory than the implementation in TOL. This is also due to implementation details. However on the queen7_7 graph, our implementation could DP with 49 megabytes of memory and TOL swapped out on our machine with 1 gigabyte of memory.

Graph						LibTW		TOL	
Name	$\|V\|$	$\|E\|$	TW	UB	Clique	CPU	Mem	CPU	Mem
alarm	37	65	4	4	5	533412	377	441840	277
mainuk	48	198	7	7	unused	62556	33	14530	7
					8	24066	17	12280	7
myciel3	11	20	5	5	unused	5	1	0	2
					2	5	1	0	2
myciel4	23	71	10	11	unused	1325	1	2280	7
					2	604	1	1700	4
myciel5	47	236	19	21	2	*swapped out*		*swapped out*	
oesoca+-pp	14	75	11	11	unused	3	1	0	2
					9	5	1	0	2
oow-trad	33	72	6	6	unused	112741	137	158610	101
					4	16007	23	20240	19
pathfinder-pp	12	43	6	6	unused	2	1	0	2
					6	3	1	0	2
queen5_5	25	160	18	18	unused	67	1	50	2
					5	24	1	20	2
queen6_6	36	290	25	26	unused	2518	3	2180	3
					6	829	1	700	2
queen7_7	49	476	35	37	unused	630109	261	*swapped out*	
					7	97894	49	*swapped out*	
ship-ship-pp	30	77	8	8	unused	366979	271	215470	173
					4	71745	77	49190	46
water	32	123	10	10	unused	17869	13	20050	13
					6	9573	11	2840	4

Table 1: Computational results TreewidthDP

4.2 QuickBB

The QuickBB algorithm proposed by Gogate and Dechter [5] is a Branch and Bound method for computing treewidth. The algorithm performs a search over permutations of the vertices.

The lowerbound algorithm used with QuickBB is the MinorMinWidth algorithm (as proposed in [5]). We have implemented a version of the MinorMinWidth algorithm which is optimized for use with QuickBB.

Using an upperbound at the beginning of the algorithm prevents the algorithm from doing a lot of work: if we can already conclude that upperbound is equal to lowerbound, we know the treewidth. Even if we are not that lucky, a good upperbound will always be useful. We have experimented with the GreedyDegree and the GreedyFillIn upperbound algorithms.

4.2.1 Implementation issues

Because the algorithm in its basic version is quite slow (just using MinorMinWidth, nothing else) we have implemented a method to prevent the algorithm from visiting equivalent nodes in the Branch and Bound tree more than once. A node in the Branch and Bound tree represents a certain graph. By eliminating the same vertices in a different order we end up with the same graph. This means that Branch and Bound nodes with the same vertices represent the same graph and will therefor return the same result. We use memorization on vertex sets to prevent branching on equivalent nodes more than once. This comes at a memory cost, of course, but it is quite modest.

We have also implemented the possibility to set the branching ordering for the vertices. This ordering can be retrieved from an upperbound algorithm.

To improve the running time of the algorithm the authors suggested a method to reduce the graph without affecting the treewidth. The method to reduce the graph is proposed by Bodlaender et al. [3] and is called 'the simplicial vertex rule' and 'the almost simplicial vertex rule'. In our implementation it is recommended to check only at the first step, because checking if a vertex is (almost) simplicial is quite computationally expensive.

4.2.2 Computational results

We have tested our implementation of QuickBB on several graphs, with different configurations. The results are shown in Table 2.

For some graphs (alarm, miles250, pathfinder, mainuk, mainuk-pp) we found already at the first step an upperbound equal to a lowerbound. Those graphs are not included in the result list, because they do not give much information about the performance of our implementation. Gogate and Dechter did report some results where the upperbound was equal to the lowerbound and where it still took them over a minute to compute the treewidth. This is remarkable, because if we know that the upperbound equals the lowerbound, we immediately know the exact treewidth.

From the computational results we can conclude that setting a branching order for the vertices is not very useful in our implementation. However, when testing with the implementation of Gogate & Dechter it actually makes quite a difference.

			LibTW										Gogate & Dechter		
graph	$\|V\|$	$\|E\|$	Init perm	UB Alg	Init Simp	Branch Simp	LB	UB	TW	Runtime	Created	Skipped	Time	Time[a]	Time[b]
anna-pp	22	148	-	GD	-	-	11	12	12	132	520	497	71	41	
			-	GD	(A)S	-	11	12	12	134	520	497			
			-	GD	(A)S	(A)S	11	12	12	344	669	590			
			GD	GD	-	-	11	12	12	150	605	393			
			GD	GD	(A)S	-	11	12	12	151	605	497			
			GD	GD	(A)S	S	11	12	12	345	789	712			
			GD	GD	(A)S	(A)S	11	12	12	413	789	609			
			GD	GD	(A)S	-	11	12	12	151	605	609			
			-	GF	(A)S	-	11	12	12	144	520	701			
			GF	GF	(A)S	S	11	12	12	373	840	712			
			GF	GF	-	-	11	12	12	112	408	497			
			GF	GF	-	-	11	12	12	3.771	not used				
myciel3	11	20	-	GD	-	-	4	5	5	6	6	5	63	10	59
myciel4	23	71	-	GD	-	-	10	8	11	7.283	33.347	50.706	199	205	
			-	GD	(A)S	-	10	8	11	7.315	33.347	50.706			
			GD	GD	-	-	10	8	11	7.302	33.161	50.522			
			GD	GD	(A)S	-	10	8	11	7.327	33.161	50.522			
			GD	GD	(A)S	S	10	8	11	10.402	36.053	51.439			
			GD	GD	(A)S	(A)S	10	8	11	12.247	36.053	50.439			
			-	GF	(A)S	-	10	8	11	7.539	33.347	50.706			
			GF	GF	-	-	10	8	11	7.750	34.535	53.927			
			GF	GF	(A)S	S	10	8	11	10.166	35.522	50.090			
myciel5	57	236	GF	GF	(A)S	S	14	21	19[c]	-	-	-	OoM[d]	OoM[d]	112.120
oesoca+	67	208	GD	GD	(A)S	-	-	-	15	26	25[c]	-	110	108	2.787
queen5_5	26	60	-	GD	-	-	12	18	18	3.690	8.907	12.020	2.034	2.006	5.409
			-	GD	-	-	12	18	18	3.715	8.978	12.156			
			-	GD	(A)S	-	12	18	18	3.691	8.907	12.020			
			GD	GD	(A)S	-	12	18	18	3.716	8.978	12.156			
			GD	GD	(A)S	S	12	18	18	9.502	10.018	10.595			
			GD	GD	(A)S	(A)S	12	18	18	12.047	10.018	10.595			
			-	GF	-	-	12	18	18	4.013	9.450	13.044			
			-	GF	(A)S	-	12	18	18	3.716	8.907	12.020			
			GF	GF	(A)S	S	12	18	18	11.583	11.664	13.378			
			-	GF	-	-	12	18	18	52.359	not used				
queen6_6	36	290	-	GD	-	-	15	26	25	191.739	173.869	271.332	65.361	73.892	81.320
			GD	GD	(A)S	-	15	26	25	185.754	167.952	261.695			
			GF	GF	(A)S	S	15	26	25	444.856	187.145	237.570			
DSJR500.1c	500	121.275	GF	GF	(A)S	-	474	485	485[c]	-	-	-	8.515	104.348	656.198
DSJC125.9	125	6.691	GF	GF	(A)S	-	104	119	119	735.202	23.096	24.544	13.134	13.179	260.879
DSJC125.9	125	736	GD	GD	(A)S	-	20	67	65[c]	-	-	-			

Table 2: Computational results QuickBB

[a]Time when using -lb and –min-fill-ordering options
[b]Running time reported in [5]
[c]Best upperbound found within time or memory limit
[d]Out of memory (1 GB)

The check for simplicial vertices does not improve the running time of the algorithm with our implementation, even though it quite often finds simplicial vertices and reduces the graph. Therefore the check for simplicial vertices is probably useful on some graphs and because it is not expensive it is recommended to do it at least at the first step.

The check for almost simplicial vertices is quite expensive if we do it in every branching node. The number of almost simplicial vertices found is often zero, so checking for almost simplicial vertices is not very useful on the tested graphs, but can probably be useful on a graph with a lot of almost simplicial vertices.

Preventing the algorithm from exploring Branch and Bound nodes it already explored seems to be very useful. During the implementation we already noticed it made quite a difference. We have tested this on 2 graphs (anna-pp and queen5_5) and it speeds up the algorithm by at least factor 15. The number of stored and skipped nodes can be found in Table 2: columns 'Created' and 'Skipped.'

We also ran the implementation of Gogate and Dechter on the same system and included their results also in Table 2. The first column (Time) contains the result of running without any options. The second column contains the result of running the algorithm with the options -LB and -MIN-FILL-ORDERING. The third column contains the results reported in [5]. Unfortunately we often did not even get close to the running times they achieve. The running times we achieved with their implementation is sometimes as much as times faster then they reported in [5] (on comparable systems).

The difference between their implementation and ours is caused by several factors: we used Java instead of C++ and they've implemented more ideas to reduce the graph while running the algorithm.

4.3 Comparison of TreewidthDP and QuickBB

So, which one is better: TREEWIDTHDP or QUICKBB? The answer is not that simple. QUICKBB can handle much larger graphs and uses less memory than TREEWIDTHDP, but TREEWIDTHDP is much faster. The processing time and memory usage of both QUICKBB and TREEWIDTHDP heavily depend on the quality of the computed lowerbound and upperbound and on the graph itself.

If it does not run out of memory you will prefer TREEWIDTHDP, but as noted above, its memory usage is hard to predict. On the graphs we have tested on we were able to handle graphs with about 40 to 50 vertices on our machine with 1 GB of memory.

5 Further work

- Multi-threading will become much more important in the future. (Recently Intel predicted that within 5 years we would have processors with as many as 80 cores.) Several algorithms, QUICKBB and TREEWIDTHDP in particular, could benefit enormously from using more cores. But this does require a multi-threaded implementation and doing this well is non-trivial.

- Another idea not yet used is to include the clique trick in the QUICKBB algorithm. We think that will lead to a substantial improvement.

- In some graphs of practical problems, for example the queen graphs, there is a lot of symmetry. Probably the running time and memory usage of the QuickBB and TreewidthDP algorithms will improve if we do not consider symmetric cases. This can be done by hand, but it would be nicer if this is done automatically. However, this involves solving graph automorphism kind of problems, which are hard. On the other hand, we are already solving an NP-complete problem as preprocessing (Maximum Clique) so perhaps this is feasible as well.

- An interesting possibility is to use the Dynamic Programming algorithm until you run out of memory and then use the computed treewidths in a Branch and Bound algorithm. That way you would benefit from both the speed of Dynamic Programming and the low memory usage of Branch and Bound.

References

[1] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Algebraic Discrete Methods*, 8(2):277–284, 1987.

[2] Anne Berry, Jean R. S. Blair, and Pinar Heggernes. Maximum cardinality search for computing minimal triangulations. In *WG '02: Revised Papers from the 28th International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 1–12, London, UK, 2002. Springer-Verlag.

[3] H. Bodlaender, A. Koster, F. van den Eijkhof, and L. van der Gaag. Preprocessing for triangulation of probabilistic networks. In *Uncertainty in Artificial Intelligence (UAI2001)*, pages 32–39. Morgan Kaufmann, 2001.

[4] Hans L. Bodlaender, Fedor V. Fomin, Arie M. C. A. Koster, Dieter Kratsch, and Dimitrios M. Thilikos. On exact algorithms for treewidth. Technical Report UU-CS-2006-032, Institute of Information and Computing Sciences, Utrecht University, 2006.

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

[6] Arie M.C.A. Koster, Hans L. Bodlaender, and Stan P.M. van Hoesel. Treewidth: Computational experiments. Research Memoranda 001, Maastricht : METEOR, Maastricht Research School of Economics of Technology and Organization, 2002. available at http://ideas.repec.org/p/dgr/umamet/2002001.html.

[7] Siddharthan Ramachandramurthi. The structure and number of obstructions to treewidth. *SIAM Journal on Discrete Mathematics*, 10(1):146–157, 1997.

A Computational results for the heuristics

Graph	All-Start MCS	All-Start MMD	All-Start MMD Least-C	All-Start MinorMinWidth	MCS	MMD	MMD + Least-C	MMD + Max-D	MMD + Min-D	MinDegree	MinorMinWidth	Ramachandramurthi
eil51.tsp	4	4	5	5	4	4	5	4	5	4	5	4
queen5_5	12	12	12	12	12	12	12	12	12	12	12	12
LabeledTest	3	2	3	3	2	2	3	3	3	2	3	3
celar02	9	9	10	9	9	3	10	4	4	1	9	1
queen6_6	15	15	15	15	15	15	15	15	15	15	15	15
queen7_7	18	18	20	18	18	18	19	18	18	18	18	18
barley-pp	6	5	6	6	6	4	6	5	5	4	6	4
alarm	4	4	4	4	4	2	4	2	2	1	4	1
celar07	12	11	15	13	11	6	15	6	6	1	13	1
celar09pp	7	7	7	7	7	4	3	3	3	3	7	3
myciel3	3	3	5	4	3	3	4	3	4	3	4	3
myciel4	5	5	8	8	5	5	8	5	8	4	8	4
myciel5	8	8	14	14	8	8	14	8	13	5	14	5
munin2-wpp	4	3	6	6	4	3	6	3	4	2	6	2
barley	5	5	6	6	5	4	6	4	4	2	6	2
water	8	6	8	7	7	6	8	6	7	1	7	1
david	10	10	12	11	10	6	12	6	6	1	11	1
MCSTestGraph	3	2	3	3	2	2	3	3	3	2	3	3
miles250	8	7	9	9	7	4	0	0	0	0	9	0
david-pp	11	10	12	11	11	8	12	8	11	7	1 1	7
huck	10	10	10	10	10	7	1	1	1	1	10	1
anna-pp	10	9	11	11	10	9	11	9	11	8	11	9
MCSTestGraph2	3	2	3	3	2	2	3	3	3	1	3	2
miles500	21	19	22	22	21	10	22	11	13	3	22	3
anna	8	7	11	10	7	3	0	0	0	0	10	0
pathfinder	6	5	6	6	6	2	6	2	2	1	6	1
jean	9	9	9	9	9	4	0	0	0	0	9	0
mildew	3	3	4	4	3	3	4	3	3	1	4	2
oesoca+	9	9	9	9	9	3	9	3	3	1	9	1
oesoca	3	3	3	3	3	2	3	2	2	1	3	1
oesoca+-pp	10	9	9	10	10	9	9	9	10	7	10	10
munin1	4	4	10	8	4	2	10	3	3	1	8	1
oesoca42	3	3	3	3	3	2	3	2	2	1	3	1

Table 3: Computational results for the lowerbounds on 31 assorted graphs.

Graph	All-Start MCS	All-Start MMD	All-Start MMD Least-C	All-Start MinorMinWidth	MCS	MMD	MMD + Least-C	MMD + Max-D	MMD + Min-D	MinDegree	MinorMinWidth	Ramachandramurthi
link-wpp	6	5	11	8	5	5	11	5	6	4	8	4
mildew	3	3	4	4	3	3	4	3	3	1	4	2
munin2-pp	5	4	6	6	5	4	6	4	4	4	6	4
munin1-wpp	5	4	10	8	4	3	10	4	4	3	8	3
mildew-wpp	4	3	4	4	3	3	4	3	4	3	4	3
oesoca+-hugin	9	9	9	9	9	3	9	3	3	1	9	1
ship-ship-pp	5	4	6	6	4	4	6	4	5	4	6	4
oesoca	3	3	3	3	3	2	3	2	2	1	3	1
ship-ship	5	4	6	6	4	3	6	3	4	2	6	2
fungiuk	4	4	4	4	4	4	4	4	4	2	4	3
mainuk-pp	6	5	6	6	5	5	6	5	6	5	6	6
diabetes	4	3	4	4	3	2	4	3	3	2	4	2
oesoca+-hugin-pp	10	9	9	10	10	9	9	9	10	7	10	10
munin_kgo_complete	5	5	5	5	5	2	5	2	2	1	5	1
munin_kgo_complete-pp	5	4	4	4	5	4	4	5	4	4	4	4
munin3	4	3	7	6	4	2	7	2	2	1	6	1
pathfinder-pp	6	5	6	6	6	5	6	5	6	5	6	6
oow_solo-pp	4	4	5	5	4	4	5	4	5	4	5	4
oow_bas-wpp	3	3	4	4	3	3	4	3	3	3	4	3
mainuk	7	7	7	7	7	6	7	6	7	2	7	2
oow_bas	3	3	4	4	3	3	4	3	3	2	4	2
pigs-pp	5	4	7	7	4	4	7	4	5	4	7	4
vsd-hugin	4	4	4	4	4	2	4	2	2	1	4	1
munin_kgo_complete-wpp	4	4	5	5	4	3	5	4	4	3	5	3
alarm	4	4	4	4	4	2	4	2	2	1	4	1
pathfinder	6	5	6	6	6	2	6	2	2	1	6	1
oow_solo	4	3	5	5	4	3	5	4	4	1	5	2
oow_solo-wpp	4	4	5	5	4	4	5	4	5	4	5	4
munin3-pp	5	4	7	7	5	4	7	5	5	4	7	4
barley	5	5	6	6	5	4	6	4	4	2	6	2
ship-ship-wpp	5	4	6	6	4	3	6	3	4	3	6	3
oow-trad-pp	4	4	5	5	4	4	5	4	5	4	5	4
wilson-hugin	3	2	3	3	2	2	3	2	2	1	3	1
oow-trad-wpp	4	3	5	5	4	3	5	4	4	3	5	3
oesoca42	3	3	3	3	3	2	3	2	2	1	3	1
link-pp	6	6	11	8	6	5	11	6	6	5	8	5
water-wpp	8	6	8	8	7	6	8	7	8	5	8	6
munin2-wpp	4	3	6	6	4	3	6	3	4	2	6	2
water	8	6	8	7	7	6	8	6	7	1	7	1
munin1-pp	5	4	10	8	4	4	10	4	5	4	8	4
oow-trad	4	3	5	5	4	3	5	4	4	2	5	2
pigs	3	3	7	6	3	2	7	2	2	2	6	2
oesoca+-hugin-wpp	9	8	9	9	9	6	9	6	8	2	9	3
munin4	5	4	7	7	4	2	7	2	2	1	7	1
weeduk	7	7	7	7	7	7	7	7	7	3	7	3
barley-wpp	5	5	6	6	5	4	6	5	5	3	6	4
munin4-pp	5	5	8	7	5	4	8	4	5	4	7	4
munin4-wpp	5	4	8	7	4	3	8	4	4	3	7	3
diabetes-pp	4	4	4	4	4	4	4	4	4	4	4	4
pigs-wpp	3	3	7	6	3	3	7	3	3	3	6	3
boblo	3	3	3	3	3	2	3	2	2	1	3	1
munin3-wpp	4	4	7	7	4	3	7	4	4	3	7	3
barley-pp	6	5	6	6	6	4	6	5	5	4	6	4
munin1	4	4	10	8	4	2	10	3	3	1	8	1
diabetes-wpp	4	3	4	4	3	3	4	3	3	3	4	3
water-pp	8	6	8	8	7	6	8	7	8	5	8	6
link	4	4	11	8	4	4	0	0	0	0	8	0
munin2	4	3	6	6	4	2	6	2	2	1	6	1
munin2	4	3	6	6	4	2	6	2	2	1	6	1

Table 4: Computational results for the lowerbounds on 59 probabilistic networks.

Graph	GreedyDegree	GreedyFillIn	LexBFS	MCS	MCS-Min
munin_kgo_complete-wpp	5	5	6	6	6
munin1-pp	11	11	26	18	26
barley-pp	7	7	10	8	10
oow-trad	6	6	7	6	7
alarm	4	4	4	4	4
water-pp	11	10	11	11	11
oow_solo	6	6	8	8	8
oesoca+-hugin-wpp	11	11	14	14	14
ship-ship	8	8	10	9	10
vsd-hugin	4	4	5	5	5
fungiuk	4	4	5	4	4
oesoca	3	3	7	3	7
oesoca+-hugin-pp	11	11	11	11	11
barley	7	7	8	8	8
diabetes	4	4	87	9	87
water	11	10	13	11	13
mainuk	7	7	9	8	9
munin4-pp	8	8	21	15	21
boblo	3	3	21	4	21
link	19	15	89	26	89
oow-trad-wpp	6	6	7	6	7
oow_solo-wpp	6	6	7	7	7
munin1	11	11	24	22	24
munin2	7	7	16	13	16
ship-ship-wpp	8	8	10	9	10
oow-trad-pp	6	6	6	6	6
munin3	7	7	53	40	53
munin4	8	8	42	22	42
link-pp	19	15	64	26	64
oow_solo-pp	7	6	7	7	7
pigs-pp	10	10	17	18	17
mildew	4	4	5	5	5
ship-ship-pp	8	8	9	9	9
barley-wpp	7	7	10	8	10
wilson-hugin	3	3	3	3	3
pathfinder	7	6	14	7	14
diabetes-wpp	4	4	91	28	91
munin3-pp	7	7	38	45	38
water-wpp	11	10	11	11	11
oow_bas	4	4	5	5	5
munin_kgo_complete-pp	5	5	6	6	6
diabetes-pp	5	4	49	26	49
link-wpp	19	15	84	26	84
mainuk-pp	6	6	7	7	7
pigs	10	10	40	20	40
munin1-wpp	11	11	28	18	28
munin2-wpp	7	7	24	15	24
munin3-wpp	7	7	53	39	53
munin_kgo_complete	5	5	27	8	27
munin4-wpp	8	8	34	19	34
weeduk	7	7	7	7	7
mildew-wpp	4	4	5	5	5
munin2-pp	7	7	9	10	9
oesoca+-hugin	11	11	12	14	12
oow_bas-wpp	4	4	5	5	5
pathfinder-pp	7	6	7	8	7
pigs-wpp	10	10	32	16	32
oesoca42	3	3	6	4	4

Table 5: Computational results for the upperbounds on 58 probabilistic networks.