Best-First Search for Treewidth

P. Alex Dow and **Richard E. Korf**
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095
alex_dow@cs.ucla.edu, korf@cs.ucla.edu

Abstract

Finding the exact treewidth of a graph is central to many operations in a variety of areas, including probabilistic reasoning and constraint satisfaction. Treewidth can be found by searching over the space of vertex elimination orders. This search space differs from those where best-first search is typically applied, because a solution path is evaluated by its maximum edge cost instead of the sum of its edge costs. We show how to make best-first search admissible on max-cost problem spaces. We also employ breadth-first heuristic search to reduce the memory requirement while still eliminating all duplicate nodes in the search space. Our empirical results show that our algorithms find the exact treewidth an order of magnitude faster than the previous state-of-the-art algorithm on hard benchmark graphs.

Introduction and Overview

Treewidth is a fundamental property of an undirected graph that is meant to measure how much like a tree a graph is. A tree has a treewidth of one, while a clique with n vertices has a treewidth of $n-1$. There are multiple equivalent definitions of treewidth that suggest unique ways of thinking about the problem (Kloks 1994; Bodlaender 2005). Our efforts are focused on finding treewidth in terms of optimal vertex elimination orders.

Eliminating a vertex from a graph is defined as the process of adding an edge between every pair of the vertex's neighbors that are not already adjacent, then removing the vertex and all incident edges from the graph. A *vertex elimination order* is a total order over the vertices in a graph. The *width* of an elimination order is defined as the maximum degree of any vertex when it is eliminated from the graph. Finally, the *treewidth* of a graph is the minimum width over all possible elimination orders, and any order whose width is the treewidth is an *optimal vertex elimination order*.

Finding the treewidth of an undirected graph is central to many queries and operations in a variety of areas, including probabilistic reasoning and constraint satisfaction. The predominant algorithms for exact inference on Bayesian networks, including Bucket Elimination (Dechter 1996), Jointree (Lauritzen & Spiegelhalter 1988), and Recursive Conditioning (Darwiche 2001), require an elimination order over the variables in a network. These algorithms have runtime complexity that is exponential in the width of the elimination order, therefore a slightly suboptimal order may make inference infeasible. An algorithm that finds the exact treewidth of a graph can dramatically improve the performance of exact inference and thereby increase the size of problems that can be solved in practice.

In this paper we propose several new algorithms for finding exact treewidth by searching the space of vertex elimination orders. This search space has many duplicate nodes, and we propose using algorithms, like best-first search, that detect and eliminate all of them. The version of best-first search most often used in practice is A^*, which applies to shortest-path problems. A solution path in the elimination order space is evaluated by its maximum elimination cost, as opposed to the sum of its elimination costs, therefore A^* does not apply. We show that best-first search is admissible on a max-cost problem space if the heuristic function satisfies a form of the triangle inequality that we refer to as max-consistency. Additionally, we show how to modify the predominant heuristic function for treewidth to make it max-consistent. We then use breadth-first heuristic search (Zhou & Hansen 2006) to reduce the memory requirements while still eliminating all duplicate nodes. We have included an empirical evaluation of our new algorithms and a comparison with the existing state-of-the-art. For the evaluation we used both random and benchmark graphs.

To avoid confusion, we refer to points in the search space as *nodes*, and points in a graph as *vertices*.

Prior Work

Determining the treewidth of a general graph is NP-complete (Arnborg, Corneil, & Proskurowski 1987). The current state-of-the-art algorithm, QuickBB (Gogate & Dechter 2004), uses depth-first branch-and-bound with a heuristic evaluation function, a vertex-ordering heuristic, and several pruning rules. QuickBB searches through the space of vertex elimination orders by eliminating one vertex at each stage and deriving the corresponding intermediate graph. An *intermediate graph* refers to the graph that results from eliminating some set of vertices from the original graph. In this search space a node is characterized by an *elimination order prefix*, i.e. a prefix of vertices that will be the first eliminated in any extension of the order. Each

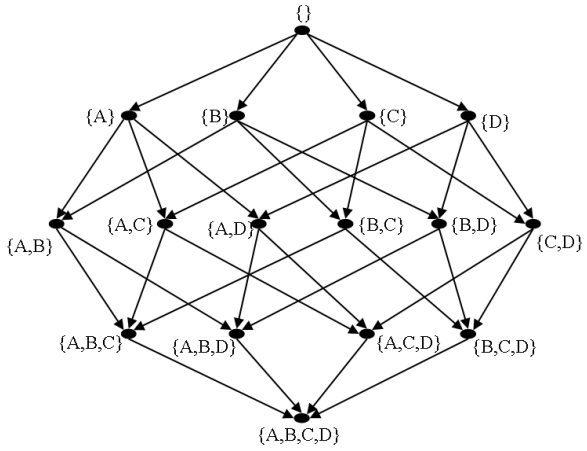

Figure 1: 2^n states in the elimination order search space.

child of a node is generated by eliminating a vertex from the parent's intermediate graph and appending the vertex to the parent's elimination order prefix.

As a tree-search algorithm, QuickBB treats elimination order prefixes as states by effectively assuming that each one results in a unique intermediate graph. In fact, a set of vertices eliminated from the original graph in any order results in the same intermediate graph (Bodlaender *et al.* 2006). Figure 1 shows the state space for a graph with four vertices, where each state is labeled with the unordered set of vertices eliminated from the graph. *Duplicate nodes* are a set of distinct nodes that correspond to different paths to the same state. QuickBB searches the state space as if it were a tree, therefore it cannot know if a newly generated node is a duplicate of a node that was previously encountered. The effect of this is that, given a graph of n vertices the state space has only 2^n distinct states, but QuickBB will search the tree with over $n!$ nodes. QuickBB does eliminate some duplicate nodes with a small set of pruning rules, although as the problem size grows this small set of rules fails to prune an increasing percentage of the duplicate nodes.

QuickBB actually only needs to search to depth $n - k$, where k is the treewidth of the graph. Nodes below that depth correspond to intermediate graphs with less that k vertices, thus they can be eliminated in any order. This implies that the worst-case running time of QuickBB is $O(n!/k!)$.

Best-First Search

Since the number of states in the elimination order search space, 2^n, is much smaller than the number of nodes, over $n!$, detecting and discarding duplicate nodes will make the search much more efficient. For that we turn to best-first search methods that avoid duplicates by saving a list of the states they have encountered so far. Best-first search uses a *Closed list* to store nodes that have been expanded, and an *Open list* to store nodes that have been generated but not yet expanded. An *evaluation function*, denoted $f(n)$, is used to form an optimistic estimate of the cost of any complete solution path through a node n.

Best-first search progresses by choosing a node with the *best* f-value in the Open list for expansion. The node is removed from the Open list and each of its children are generated. For each child node that is generated, best-first search checks the Open and Closed lists to ensure that it is not a duplicate of a previously generated node or, if it is, that it represents a better path than the previously generated duplicate. If that is the case, then the node is inserted into the Open list. After every child node has been generated, the parent node is inserted into the Closed list and a new node is chosen for expansion. This continues until a goal node is chosen for expansion.

Different best-first search algorithms are constructed by choosing different evaluation functions. Frequently the evaluation function is constructed by combining a measurement of the cost incurred by a partial solution path, denoted by the function $g(n)$, and an optimistic estimate of the cost of any path that follows from a node, denoted by the function $h(n)$. The well-known best-first search algorithm A^* uses the evaluation function $f(n) = g(n) + h(n)$. In the elimination order search space, the cost of a complete solution path is the width of the corresponding order. It is computed by taking the maximum elimination cost along the solution path, where the cost of a particular elimination is the degree of the vertex being eliminated. A reasonable g-function would thus be the maximum cost incurred in reaching a node. A reasonable h-function would be an optimistic estimate of the maximum cost along some path from the current state to a goal state. These can be combined into the following evaluation function:

$$f(n) = \max\left(g(n), h(n)\right). \tag{1}$$

Max-Cost Best-First Search

To demonstrate that best-first search with (1) as an evaluation function will find the exact treewidth of a graph we must show that it is *complete*, i.e. it will always terminate with a solution, and *admissible*, i.e. the solution it returns is guaranteed to be optimal. For the elimination order search space, showing completeness is trivial, because, as can be seen in Figure 1, the search space is finite with no loops.

To show admissibility several definitions are required.

Definition An evaluation function f is *order preserving* if, for any two paths P_1 and P_2 from the start state to some other state n, and any path P_3 from state n to some other state m, the following holds:

$$f(P_1) \geq f(P_2) \Rightarrow f(P_1P_3) \geq f(P_2P_3) \tag{2}$$

Let $C(\omega)$ denote the cost function that returns the actual cost of a complete solution path ending at goal node ω. The following conditions are sufficient for a best-first search algorithm to be admissible (Dechter & Pearl 1985):

- The evaluation function must be optimistic
- $f(\omega) = C(\omega)$ for any goal node ω
- The evaluation function must be order-preserving.

(1) meets the first two conditions if the heuristic function, $h(n)$, is admissible, i.e. an optimistic lower-bound on paths that follow a node.

To make (1) order-preserving, we define a form of the triangle inequality with respect to the max cost function:

Definition A heuristic function $h(n)$ is *max-consistent* if, for all pairs of states n,m, where m is a descendant of n, the following holds:

$$h(n) \leq \max\left(k(n,m), h(m)\right) \tag{3}$$

where $k(n,m)$ denotes the maximum edge cost on any path from n to m.

Proposition 1 *If the heuristic function h is max-consistent, then (1) is order-preserving.*

Proof We need to prove the following:

$$\begin{aligned}\max(g(P_1), h(n)) \geq \max(g(P_2), h(n)) \Rightarrow \\ \max(g(P_1), k(n,m), h(m)) \geq \\ \max(g(P_2), k(n,m), h(m))\end{aligned}$$

There are two disjoint cases in which the left side is true: $g(P_1) \geq g(P_2)$, and $g(P_1) < g(P_2) \leq h(n)$. In the first case, the condition clearly holds. In the second case, we see that the condition holds if $g(P_2) \leq \max(k(n,m), h(m))$. This holds by the definition of a max-consistent heuristic, because, in this case, $g(P_2) \leq h(n)$. ∎

It follows that max-cost best-first search with a max-consistent heuristic is admissible.

Incorporating a Non-Max-Consistent Heuristic

The predominant lower-bound heuristics for treewidth are referred to as MMD+ and were developed independently by Bodlaender, Koster, & Wolle (2004) and by Gogate & Dechter (2004). These heuristics are based on contracting edges in a graph, an operation that never increases a graph's treewidth. We focus on MMD+(min-d), which is computed by continually contracting an edge between a min-degree vertex and a min-degree neighbor. After the graph has been contracted down to a single vertex, the degree of the maximum min-degree vertex encountered in the process is a lower bound on the treewidth of the original graph. QuickBB uses this heuristic.

Different tie-breaking procedures can lead MMD+(min-d) to return different results on the same graph. A counterexample shows that breaking ties in a specific manner causes the heuristic to violate the definition of max-consistency (see Appendix). Since we have no tie-breaking procedure guaranteed to make the heuristic max-consistent, we break ties arbitrarily and assume it is not max-consistent.

In order to incorporate MMD+(min-d) into best-first search and maintain admissibility, we construct a max-consistent heuristic function out of an admissible heuristic function. This technique is similar to those used by Mero (1984) for B', and Dechter & Pearl (1985) for A^{**}.

Proposition 2 *Given an admissible heuristic function $h(n)$, a parent node n_p, one of its child nodes n_c, and the edge cost between them c, then the following is a max-consistent heuristic function for node n_c:*

$$\tilde{h}(n_c) = \begin{cases} h(n_c) & \text{if } c \geq \tilde{h}(n_p) \\ \max(h(n_c), \tilde{h}(n_p)) & \text{if } c < \tilde{h}(n_p) \end{cases} \tag{4}$$

Proof In (3), let n and m refer to n_p and n_c, respectively. Clearly max-consistency holds in this case. Induction on n_c shows that max-consistency holds for n_p and all of its descendants, thus $\tilde{h}(n)$ is a max-consistent heuristic. ∎

An additional benefit of this heuristic function is that it can improve the quality of the admissible heuristic used as a subroutine. The $\tilde{h}$-value of an inferior duplicate node may be greater than that of others if it inherits its parent's $\tilde{h}$-value. Before the inferior node is discarded, the superior duplicate can take its $\tilde{h}$-value.

Best-First Treewidth

Here we discuss the details of our application of best-first search to the elimination order search space, which we call BestTW. Like QuickBB, BestTW only needs to search to depth $n - k$, where k is the treewidth of the graph. We can bound the worst-case running time of BestTW by $O(2^n)$, though if $k > n/2$ then $O((n-k)\binom{n}{k})$ is a tighter bound.

Graph Reduction Techniques

Bodlaender *et al.* (2001) developed several "safe" rules for eliminating vertices from a graph without affecting the treewidth. Briefly, a vertex is *simplicial* if its neighbors form a clique, and a vertex is *almost simplicial* if all but one of its neighbors form a clique. The *simplicial rule* states that if a vertex v is simplicial, then the treewidth of the graph is the greater of the degree of v and the treewidth of the graph that results from eliminating v. The *almost simplicial rule* states that if a vertex v is almost simplicial and the degree of v is no more than a lower bound, lb, on the treewidth of the original graph, then the treewidth of the current graph is no greater than lb or the treewidth of the graph that results from eliminating v. In BestTW, a node's f-value is used for lb.

When one of these reduction rules applies to a vertex in an intermediate graph, then we know that the child that is generated by eliminating that vertex is the only child that needs to be explored. Thus we can prune all but that one child from the search space. QuickBB uses both reduction rules in the same manner.

Efficient Node Representation

In best-first search, the primary factor limiting the size of solvable problems is the large number of nodes that must be stored. To restrict the size of a single node we use a compact representation of the state. Recall that the state is uniquely determined by the set of vertices that have been eliminated from the original graph. The state can thus be represented by an n-bit vector, where n is the number of vertices in the original graph, and a bit is set if the corresponding vertex has been eliminated.

While compact, this representation adds a computational overhead to node expansion, because the corresponding intermediate graph must be derived by eliminating the appropriate vertices from the original graph.

Pruning with an Upper Bound

The width of a suboptimal elimination order can be used as an upper bound to prune any nodes in the Open List that

have an f-value that meets or exceeds it. BestTW uses the *Minimum Fill-in (min-fill)* heuristic (see Bodlaender (2005)) to compute an upper bound before starting to search.

Breadth-First Heuristic Search

The primary bottleneck in BestTW is the large number of nodes it needs to store. To reduce this number and still eliminate all duplicate nodes, we employ breadth-first heuristic search (Zhou & Hansen 2006). Breadth-first heuristic search is a frontier search (Korf *et al.* 2005) variant that expands nodes in order of their depth in the search space. It stores fewer nodes than best-first search, because all nodes at a previously explored depth can be discarded. An admissible heuristic and an upper bound are used to prune nodes as they are generated, and therefore further restrict the number of nodes that must be stored at any given time. A divide-and-conquer approach is used for solution reconstruction.

While breadth-first heuristic search is designed to avoid duplicate nodes by carefully searching a graph with unit edge costs, Zhou & Hansen (2003) describe a variant that searches partially ordered graphs with arbitrary edge costs. A partially ordered graph is a directed graph whose nodes are divided up into disjoint subsets, called layers, which are ordered such that each node in a given layer has descendants only in the same or later layers. This guarantees that the nodes in a single layer can be discarded once they have all been expanded. Depending on the structure of the graph, only a small number of layers need to be stored at any given time. Zhou & Hansen's algorithm is called Sweep A*, because nodes in a single layer must be expanded in a best-first order among the nodes in that layer. Thayer (2003) proposes an algorithm, bounded diagonal search, that works for the special case of partially ordered graphs where none of the descendants of the nodes in a single layer are in that same layer. For these graphs the nodes in a single layer can be expanded in an arbitrary order and discarded immediately after expansion.

Breadth-First Heuristic Treewidth

As can be seen in Figure 1, the elimination order state space is a partially ordered graph with layers that correspond to depth in the search space. We propose an algorithm, *breadth-first heuristic treewidth (BFHT)*, that searches the elimination order state space by expanding nodes by depth. Since all the descendants of a node lie in deeper layers, BFHT can expand nodes within a single layer in an any order and discard each node after it is expanded. The size of the largest layer, $\binom{n}{n/2}$, is an upper bound on the number of nodes that need to be stored in memory at any given time.

BFHT uses min-fill to compute an upper bound solution at the beginning of the search. Using the same evaluation function and heuristic as BestTW, BFHT can prune nodes that meet or exceed this upper bound. Since BFHT eliminates all duplicate nodes, it has the same worst-case performance as BestTW. Nevertheless, a loose initial upper bound will cause it to expand more nodes than BestTW.

Like other versions of frontier search (Korf *et al.* 2005; Zhou & Hansen 2006), the time BFHT spends on divide-and-conquer solution reconstruction is insignificant. Any significant time difference between BestTW and BFHT is attributed to BFHT expanding more nodes in its initial search.

Results

Here we empirically demonstrate the effectiveness of our algorithms by comparing their performance to that of QuickBB on a variety of graphs. Our algorithms are written in C++, and they are compared to a C++ implementation of QuickBB made available by Vibhav Gogate. The tests were conducted with a 1.8GHz Intel Pentium 4 processor and one gigabyte of RAM. For some benchmark graphs we have included two timing results for QuickBB, because there is a discrepancy between the performance reported by Gogate & Dechter (2004) and the performance of the implementation they made available to us for our experiments.

An additional feature of QuickBB is that it is an anytime algorithm. An anytime algorithm can be stopped at any point in its execution, and it will return a solution that may or may not be optimal. QuickBB can only guarantee that the solution it returns is optimal if it runs to completion. Since we are concerned with finding exact treewidth, we disregard solutions returned by QuickBB before it completes.

Random Graphs

The graphs used in these tests were generated with the following parametric model. Given V, the number of vertices, and E, the number of edges, we generate a graph by choosing E edges uniformly at random from the set of $V*(V-1)/2$ possible edges. We generated 100 graphs with each of nine different parameters sets (# vertices, # edges): (25,50), (25,100), (25,200), (30,60), (30,120), (30,240), (35,70), (35,140), (35,280). Each algorithm was given one hour and 800MB of memory to find the treewidth of each graph. Eight hundred of the 900 graphs were solved by all three algorithms. QuickBB was unable to solve 7 graphs from the (35,70) set, and 93 from the (35,140) set. BestTW exhausted memory on 3 of the (35,140) graphs. BFHT was able to solve all 900 graphs. Figure 2 shows the ratio of the runtime of BFHT and QuickBB to the runtime of BestTW for the graphs that all three algorithms solved. The data suggests that as graphs take longer for BestTW to solve, the rate of improvement over QuickBB increases. This is consistent with our analysis of the number of nodes searched by each algorithm. The data also shows that BFHT pays an approximately constant time-penalty over BestTW. This penalty is the result of a loose upper bound that caused BFHT to expand some nodes with f-values greater than the treewidth.

Benchmark Graphs

Table 1 shows a comparison of the algorithms on graphs from real-world problems. The first three graphs are moralized Bayesian networks, and the middle 13 are DIMACS graph coloring instances.[1] These two sets include the graphs used to evaluate QuickBB by Gogate & Dechter (2004), excluding those that took both BestTW and QuickBB less

[1] Both sets are available at the TreewidthLib website: `http://www.cs.uu.nl/~hansb/treewidthlib/`.

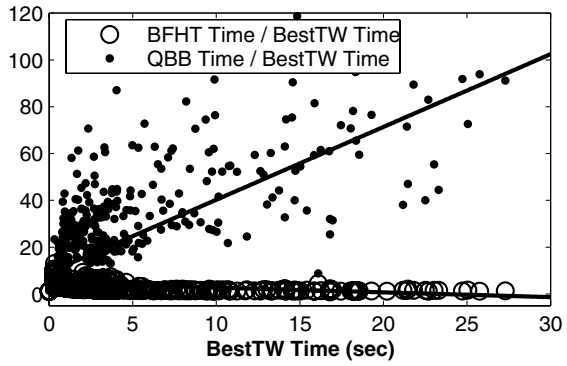

Figure 2: Ratio of BFHT and QuickBB runtime to BestTW runtime on 800 random graphs. Lines show linear least squares fit of data.

than one second to complete. The last 10 graphs are moralized Bayesian networks from the Evaluation of Probabilistic Inference Systems at UAI 2006.[2]

Each algorithm was given five hours and 800MB of memory to solve each graph. The majority of the results are consistent with the random graph results, where BestTW's improvement factor over QuickBB grows with the difficulty of the problem. Both BestTW and BFHT frequently find the treewidth an order of magnitude faster than QuickBB.

There are several graphs where QuickBB outperforms our algorithms. The only graph we have encountered that QuickBB could solve yet caused BestTW and BFHT to exhaust memory was miles500. There are four other graphs that QuickBB was able to solve faster that our algorithms: mulsol.i.5 and the inithx.i graphs. With hundreds of vertices, these were some of the larger graphs we solved. Our algorithms have setup costs and node overheads that QuickBB does not. Typically they outperform QuickBB by expanding many fewer nodes, a difference that subsumes any overhead costs. These graphs were very easy, requiring only hundreds of node expansions, thus they were not able to make up that cost. Clearly this will not be an issue on hard instances.

Comparing Memory Usage

Figure 3 shows the reduction in memory requirements for BFHT over BestTW on the 897 random graphs both algorithms solved. Although there is a lot of variance in the easy graphs on the left, it is clear that BFHT consistently requires 20–30% of the memory required by BestTW.

Conclusions

In recent years, the dominant approach for finding the exact treewidth of a general graph has been heuristic search over the space of vertex elimination orders. The existing state-of-the-art algorithm uses depth-first branch-and-bound to search the space of elimination order prefixes. We have

[2]Networks available at `http://ssli.ee.washington.edu/~bilmes/uai06InferenceEvaluation/`.

Graph	Time (sec)				tw
	BestTW	BFHT	QBB	QBB*	
diabetes	0.4	3.7	13.5	206.0	4
barley	0.5	3.8	2.0	48.8	7
munin3	1.1	2.3	9.5	109.3	7
david	0.5	3.5	5.1	77.7	13
queen5-5	0.6	1.8	2.8	5.4	18
miles1500	1.0	2.2	0.5	6.8	77
DSJC125.9	3.7	8.5	30.9	260.9	119
queen6-6	4.7	8.0	101.8	81.3	25
mulsol.i.5	5.2	495.3	0.3	3.7	31
inithx.i.2	9.6	14.0	1.9	1.1	31
inithx.i.3	9.8	14.3	1.8	1.0	31
DSJR500.1c	74.6	144.1	180.7	656.2	485
inithx.i.1	309.1	514.2	39.4	26.3	56
myciel5	550.7	1071.3	6699.6	112.1	19
queen7-7	779.0	1872.9	$>$ 5hrs	543.3	35
miles500	x	x	149.7	1704.6	22
BN_91	2.2	4.4	67.3	-	22
BN_88	3.4	6.6	75.6	-	22
BN_87	3.7	6.8	67.4	-	22
BN_89	3.7	6.5	69.0	-	22
BN_93	4.1	7.0	73.7	-	22
BN_78	7.3	26.6	1131.1	-	13
BN_92	8.5	13.6	70.6	-	22
BN_90	8.7	14.2	70.6	-	22
BN_79	8.9	12.7	1180.2	-	13

Table 1: Performance comparison on benchmark graphs. QBB* denotes runtime for QuickBB as previously published by Gogate & Dechter (2004). 'x' denotes algorithm required $>$ 800MB of memory.

demonstrated that such a search does not recognize the large number of duplicate nodes present in the search space. Best-first search automatically detects and eliminates all duplicate nodes, but this problem uses a different cost function than is typically used in best-first search. We showed that best-first search with a max-consistent heuristic is admissible on a max-cost problem like finding optimal elimination orders. We have also shown how to modify a popular elimination order heuristic to make it max-consistent. Our empirical results show that best-first search finds exact treewidth an order of magnitude faster than the existing state-of-the-art on hard instances.

We have also shown that a variant of breadth-first heuristic search can be used to decrease the memory requirement of our best-first search by 70-80% while still solving problems an order of magnitude faster than the depth-first algorithm.

Future Work

The primary goal of future work will be to increase the size of problems that can be solved in a reasonable amount of time. One approach to doing this is to continue to investigate algorithms that eliminate all duplicate nodes, specifically disk-based search (Korf 2004) These methods greatly

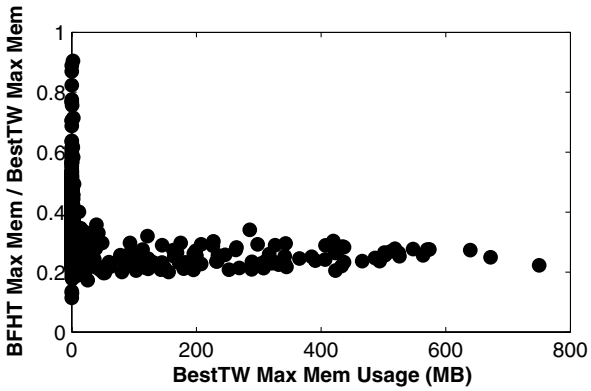

Figure 3: Ratio of BFHT's memory requirement to BestTW's memory requirement.

increase the amount of available memory by utilizing external disk storage while limiting the added costs in terms of running time.

Another approach to solving larger problems involves constant-space algorithms. The idea behind these methods is to use all the available memory to eliminate as many duplicates as possible. This can be done with a memory-bounded version of best-first search (Zhou & Hansen 2002), or by adding a transposition table to depth-first branch-and-bound.

Finally, since anytime algorithms are useful in some contexts it will be interesting to evaluate anytime variants of the algorithms developed in this paper. This can be done by finding a min-fill path from interior search nodes.

Acknowledgments

Thanks to Adnan Darwiche for helpful discussions, and to Vibhav Gogate for answering questions about QuickBB and providing the QuickBB code.

Appendix

The following graph is provided as a counterexample to the max-consistency of the MMD+(min-d) heuristic. The graph $G = (V, E)$ has 8 vertices, and the edges are listed as pairs of vertices: $E = \{(v_1, v_4), (v_1, v_7), (v_1, v_8), (v_2, v_4), (v_2, v_6), (v_2, v_7), (v_2, v_8), (v_3, v_4), (v_3, v_5), (v_3, v_8), (v_4, v_8), (v_5, v_6), (v_5, v_7)\}$

When an edge is contracted, two vertices are combined to create a new vertex. We assume that the new vertex takes the label of the lesser of the two combined vertices, i.e. edge (v_i, v_j) is contracted to form a new vertex labeled v_i if $i < j$. The MMD+(min-d) heuristic value of G may be computed by the following ordered series of edge contractions: (v_5, v_6), (v_1, v_7), (v_3, v_5), (v_1, v_2), (v_1, v_3), (v_1, v_4), (v_1, v_8). The heuristic returns a lower bound of 4.

v_5 in graph G has degree 3. Let G' be the graph produced by eliminating v_5 from G. The MMD+(min-d) value of G' is 3 if the following ordered series of edge contractions is used: (v_1, v_4), (v_3, v_6), (v_2, v_7), (v_1, v_2), (v_1, v_3), (v_1, v_8).

In (3) state n corresponds to graph G and state m corresponds to graph G'. As stated, $h(G) = 4$, $h(G') = 3$, and $k(G, G') = 3$. Clearly this violates max-consistency.

References

Arnborg, S.; Corneil, D. G.; and Proskurowski, A. 1987. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic and Discrete Methods* 8(2):277–284.

Bodlaender, H.; Koster, A.; van den Eijkhof, F.; and van der Gaag, L. 2001. Pre-processing for triangulation of probabilistic networks. In *Proc. 17th UAI*, 32–39.

Bodlaender, H. L.; Fromin, F. V.; Koster, A.; Kratsch, D.; and Thilikos, D. M. 2006. On exact algorithms for treewidth. In *Proc. 14th European Symp. on Algorithms*.

Bodlaender, H. L.; Koster, A. M. C. A.; and Wolle, T. 2004. Contraction and treewidth lower bounds. In *Proc. 12th European Symposium on Algorithms (ESA-04)*, 628–639.

Bodlaender, H. L. 2005. Discovering treewidth. In *Proc. 31st Current Trends in Theory and Practice of Computer Science*, volume 3381 of *LNCS*, 1–16. Springer.

Darwiche, A. 2001. Recursive conditioning. *Artificial Intelligence* 126(1-2):5–41.

Dechter, R., and Pearl, J. 1985. Generalized best-first search strategies and the optimality of A*. *Journal of the ACM* 32(3):505–536.

Dechter, R. 1996. Bucket elimination: A unifying framework for probabilistic inference. In *Proc 12th UAI*, 211–19.

Gogate, V., and Dechter, R. 2004. A complete anytime algorithm for treewidth. In *Proc 20th UAI*, 201–20.

Kloks, T. 1994. *Treewidth, Computations and Approximations*. Berlin: Springer-Verlag.

Korf, R. E.; Zhang, W.; Thayer, I.; and Hohwald, H. 2005. Frontier search. *Journal of the ACM* 52(5):715–748.

Korf, R. E. 2004. Best-first frontier search with delayed duplicate detection. In *AAAI-04*, 650–657.

Lauritzen, S. L., and Spiegelhalter, D. J. 1988. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of Royal Statistics Society, Series B* 50(2):157–224.

Mero, L. 1984. A heuristic search algorithm with modifiable estimate. *Artificial Intelligence* 23(1):13–27.

Thayer, I. E. 2003. Methods for optimal multiple sequence alignment. Master's thesis, UCLA.

Zhou, R., and Hansen, E. A. 2002. Memory-bounded A* graph search. In *Proc 15th FLAIRS*, 203–209.

Zhou, R., and Hansen, E. A. 2003. Sweep A*: Space-efficient heuristic search in partially ordered graphs. In *Proc 15th ICTAI*, 427.

Zhou, R., and Hansen, E. A. 2006. Breadth-first heuristic search. *Artificial Intelligence* 170:385–408.