

Duplicate Avoidance in Depth-First Search with Applications to 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

Depth-first search is effective at solving hard combinatorial problems, but if the problem space has a graph structure the same nodes may be searched many times. This may increase the size of the search exponentially. We explore two techniques that prevent this: duplicate detection and duplicate avoidance. We illustrate these techniques on the treewidth problem, a combinatorial optimization problem with applications to a variety of research areas. The bottleneck for previous treewidth algorithms is a large memory requirement. We develop a duplicate avoidance technique for treewidth and demonstrate that it significantly outperforms other algorithms when memory is limited.

1 Introduction

Single-agent pathfinding is a heuristic search problem solving technique where a solution is found as an agent seeks a path to a goal in an abstract problem space. The problem space can be represented by a graph, where nodes correspond to states and edges correspond to operations. The search graph may include many paths from the start to any single node. When the same node is reached from multiple paths in the search, we refer to it as a duplicate node. The existence of duplicates can lead to an exponential increase in the size of the search if not properly eliminated. Depth-first search is particularly prone to exploring a large number of duplicates, because, in its simplest form, it makes no effort at duplicate elimination.

Two methods of eliminating duplicate nodes are duplicate detection and duplicate avoidance. Duplicate detection relies on a list, where nodes that have already been encountered are stored. When a new node is encountered, we can check against the list to see if the node is a duplicate and whether it should be discarded. Duplicate detection is well studied and typically simple to implement.

Another type of duplicate elimination technique is duplicate avoidance. Whereas duplicate detection generates a node and checks it against a list, the purpose of duplicate avoidance is to prevent duplicates from being generated in the first

place. In this paper we describe methods for duplicate avoidance and give concrete examples of how to implement them. We show how duplicate avoidance techniques can be combined with other pruning techniques to reduce the size of the search, and we show how to avoid a pitfall in implementing duplicate avoidance that would invalidate a search algorithm.

Our discussion of duplicate elimination techniques will be in the context of a particular problem: finding exact treewidth. Treewidth is a fundamental property of a graph with significant implications for several areas of artificial intelligence research. A reason for focusing on treewidth is that a natural search space for it uses a maximum edge cost function. As we discuss in a later section, in an iterative-deepening search on a problem with a maximum edge cost function, every duplicate node can be discarded. This makes these problems well-suited for studying duplicate elimination techniques.

2 Treewidth and Maximum Edge Cost Search

2.1 Treewidth Definition and Applications

We present a definition of treewidth in terms of optimal vertex elimination orders. Note that the graphs discussed here are undirected and without self-loops.

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 edges incident to it. A *vertex elimination order* is a total order in which to eliminate 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 a graph's treewidth is central to many queries and operations in a variety of artificial intelligence research areas, including probabilistic reasoning, constraint satisfaction, and knowledge representation. These research areas generally use a graph to represent some information about the world and conduct various queries and operations on the graph. Knowing the treewidth of the graph and having an optimal elimination order can significantly speed up these queries. Some examples of this include bucket elimination for inference in Bayesian networks [Dechter, 1996] and jointree clustering for constraint satisfaction [Dechter and Pearl, 1989].

*This paper has been accepted for oral presentation at IJCAI-09.

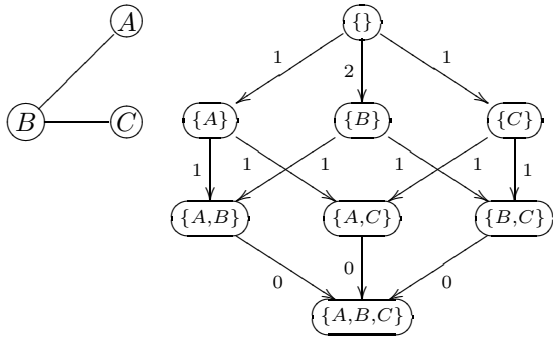


Figure 1: A graph we seek the treewidth of (left), and the corresponding search space (right).

While some classes of highly structured graphs have known treewidth [Bodlaender, 2005], determining the treewidth of a general graph is NP-complete [Arnborg *et al.*, 1987]. This inherent difficulty makes treewidth a natural candidate for heuristic search techniques.

2.2 The Vertex Elimination Order Search Space

A natural search space for treewidth involves constructing optimal vertex elimination orders. For a simple example, consider searching for an optimal elimination order for the graph on the left in Figure 1. The corresponding search space is shown on the right in Figure 1. Eliminating a set of vertices from a graph leads to the same resulting graph, regardless of the order in which the vertices are eliminated. Thus, a state in this search space can be represented by the unordered set of eliminated vertices. At the root node, no vertices have been eliminated, and, at the goal node, all three vertices have been eliminated. To transition from one node to another, a vertex is eliminated from the graph. The cost of a transition, which labels the corresponding edge in the search space, is the degree of the vertex when it is eliminated. A solution path represents a particular elimination order, and the width of that order is the maximum edge cost on the solution path.

2.3 Maximum Edge Cost Iterative Deepening

A notable detail of the vertex elimination order search space is the fact that a solution path is measured by its maximum edge cost. Most problems in the search literature use an additive cost function, where a path is evaluated by summing its edge costs. In fact, a search algorithm may behave quite differently with a maximum versus an additive cost function.

Iterative deepening (ID) is a search technique where a search is conducted as a series of iterations with an increasing cutoff value [Korf, 1985]. Each iteration is a search where solutions are only explored if they have a cost that does not exceed the cutoff. If an iteration is completed without finding a solution, then any solution must cost more than the cutoff, thus the cutoff is incremented and the search is repeated. Iterative deepening is effective for problems where the time required to search increases significantly with each increment of the cutoff. In the past, it has been predominately applied to problems with an additive cost function. We consider how its behavior is different with a maximum edge cost function.

A helpful property of ID on a maximum edge cost search space is that no node need be explored more than once. In a single iteration of ID, once a node is expanded, we will either find a goal reachable from the node, or we will rule out every descendant node reachable from the node. Descendant nodes may be ruled out because an edge exceeds the cutoff, or as a result of other pruning methods. If a solution is found, then our search can terminate because we have found an optimal solution. If no solution is found, then we know that there is no path from this node to a goal node with cost less than the cutoff. Therefore, in the course of our search, if we ever encounter a node that was previously expanded, we know that it does not lead to a solution.

Notice that this is not the case for iterative deepening with an additive cost function. With additive costs, whether a node leads to a solution depends, not only on the search space below that node, but also on the cost of the path to it.

3 Techniques for Duplicate Elimination

The search space for treewidth and many other problems is a graph, therefore the same node can be reached from many different paths. A simple depth-first search of a graph will cause every node to be explored once for each path to it. In a search for the treewidth of a graph with N vertices, the search graph has 2^N unique nodes, while depth-first search would generate $O(N!)$ nodes. After a node has been generated once, we refer to any future encounters with that node as duplicates. In this section we discuss techniques for duplicate elimination.

Duplicate detection refers to methods for eliminating duplicate node expansions by caching previously encountered nodes and checking any new nodes against the cache. In a depth-first search, duplicate detection is typically accomplished with a hash table, referred to as a *transposition table*, into which every encountered node is inserted. In the case of treewidth, the key to the table is a bitstring of length $|V|$ with a bit set if the corresponding vertex has been eliminated from the graph. When a new node is generated, we check the transposition table to see if it includes the corresponding bitstring. If it does, then the node is pruned. On difficult problems there will not be enough memory for the transposition table to store every node that is encountered. When memory is exhausted, the algorithm can use a replacement scheme, such as least-recently used (LRU), to make room for new nodes.

Another technique for eliminating duplicate nodes involves recognizing when the search space includes multiple valid paths to the same node and preventing all but one of them from being explored. We refer to this as *duplicate avoidance*, because it prevents duplicate nodes from being generated in the first place. Duplicate avoidance techniques are based on identifying *sets of duplicate operator sequences*. Each sequence represents a different path between two nodes in the search space. Identifying duplicate operator sequences for a particular problem and using them to avoid duplicates requires knowledge about the problem space. Next, we will show how this can be done for treewidth. We distinguish between two types of duplicate avoidance: *deductive* duplicate avoidance, the existence of which can be deduced from a description of the problem space; and *inductive* duplicate

avoidance, which is discovered in the course of a search.

4 Deductive Duplicate Avoidance

Deductive duplicate avoidance involves examining the problem space for structure that allows us to avoid exploring redundant paths to a node. This is related to the well-studied idea of eliminating symmetry in a search space. In the case of treewidth, we consider partial elimination orders symmetric if they result in the same graph and incur the same cost. Gogate and Dechter [2004] developed the following pruning rule, which identifies symmetries that occur when a series of non-adjacent vertices are eliminated. Here we assume that there is some total ordering over the vertices in the graph.

Independent Vertex Pruning Rule (IVPR): If the current node was reached from the start node by the partial elimination order $(v_1 \dots v_d)$, prune all children that result from eliminating a vertex w such that there exists v_i , $1 \leq i \leq d$, where $v_i > w$ and w was not adjacent to any vertex v_j , $i < j \leq d$.

Our experiments (Section 8) show that IVPR eliminates a large number of duplicates, though many remain. The next section attempts to address the remaining duplicate nodes that we are unable to eliminate with deductive avoidance.

5 Inductive Duplicate Avoidance

Eliminating a set of vertices from a graph produces the same graph regardless of the order the vertices are eliminated. In the previous section, we saw how IVPR avoids duplicates by imposing a canonical order on the elimination of non-adjacent vertices. The other orders need not be explored because they would cost the same as the canonical order. Unfortunately, when considering duplicate sequences that include vertices that are adjacent to each other, we no longer know a priori how the costs of different orders will relate. In the context of an iteration of ID search, if the cost of one of these orders exceeds the cutoff then that operator sequence isn't valid. If more than one don't exceed the cutoff, then they represent redundant duplicates that could be avoided. In this section we will discuss a technique for recognizing valid duplicate operator sequences and avoiding all but one of them.

5.1 A General Procedure

Suppose that, at some point in an ID search for some problem, we recognize that we have applied some sequence of operators from a set of duplicate operator sequences. Since the current sequence has been successfully executed, its cost must not have exceeded the cutoff value. Although there may be other duplicate sequences in the set with a lesser cost, since our search is iterative deepening we only care that the cost does not exceed the cutoff. Therefore, we declare the current sequence "good enough", and, for the rest of the search, avoid all other duplicate sequences in the set.

To accomplish this, we propose storing "good enough" sequences as they are discovered. Then, as the search progresses, we can check the current node against that list to determine if any child node would represent a duplicate of some "good enough" sequence. If it does we can prune that child, and thereby avoid generating the corresponding duplicate node. Clearly the way these sequences are stored and

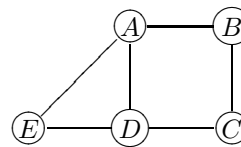


Figure 2: A graph we seek the treewidth of.

indexed will depend on the particular problem and will have a significant impact on the effectiveness of this technique.

Storing every "good enough" sequence may amount to explicitly storing the entire search space, which is infeasible on any interesting problem. This can be dealt with by limiting the scope of the duplicate avoidance. For example, we could limit the size of the sequences that are saved, or we could limit the number of sequences. Another option is to limit it to only certain types of duplicate operator sequences. This is the approach we take with treewidth in the next section.

5.2 Inductive Duplicate Avoidance for Treewidth

Consider a search for the treewidth of the graph in Figure 2, and assume the following ordering over the vertices: $A < B < C < D < E$. There are six possible duplicate orders for eliminating vertices A , C , and E ; and IVPR avoids four of them, leaving: (A, C, E) and (E, A, C) . Notice that the cost of the first sequence is $\max(3, 2, 2) = 3$, and the cost of the second is $\max(2, 2, 2) = 2$. In the context of an ID search, if cutoff = 2 then only the second sequence is explored and there is no duplicate. On the other hand, if the cutoff ≥ 3 then both sequences are valid and redundant. To avoid the duplicate we designate whichever sequence is explored first as "good enough", and use it to avoid exploring the other.

Since IVPR already eliminates duplicate nodes due to sets of non-adjacent vertices, here we will focus on sets of adjacent vertices. We will refer to this duplicate avoidance technique as the *Dependent Vertex Pruning Rule (DVPR)*, because it involves avoiding duplicates that arise from the subsequent elimination of adjacent vertices. The key aspects to implementing this technique are (1) identifying and storing "good enough" orders, (2) determining when eliminating a vertex will lead to a duplicate and should thus be pruned, and (3) determining when stored "good enough" orders are valid.

The first task is to identify and store "good enough" orders. Once a node n is generated by eliminating a vertex v , we examine the eliminations that led to node n and identify any sequence of subsequently eliminated adjacent vertices that ends with vertex v . For example, say we just generated the node that follows from the elimination order (A, C, E) in Figure 2. We then identify (A, E) as a sequence of subsequent adjacent vertices ending in vertex E . Once such a sequence is identified, we consider it a "good enough" order for eliminating those vertices, and we will store it.

The second task is to use the stored "good enough" orders to avoid duplicates. If we have determined that some order is "good enough," then any other ordering of the included vertices will result in a duplicate. Suppose we have stored some "good enough" order, and all but one vertex in that order has been eliminated. If the remaining vertex is not the last in

the order, then eliminating it would generate a duplicate node and we can prune. If the remaining vertex is the last in the order, then eliminating that vertex corresponds to exploring the “good enough” order. Continuing the example above, say we have stored “good enough” order (A, E) and the current node in the search corresponds to the graph in Figure 2 with no vertices eliminated. If we eliminate vertex E , then we will notice that eliminating A next is redundant with the “good enough” order. On the other hand, if A is eliminated first, we will notice that eliminating E next corresponds to the “good enough” order, thus we can go ahead and eliminate it.

The final task necessary for our description of DVPR is to determine when a stored “good enough” order is valid. We consider an order valid when we know that it can be executed without being pruned by the cutoff. An order becomes invalid when the graph structure has changed such that we no longer know that following that sequence of eliminations will not exceed the cutoff. We determine the validity of a “good enough” order by monitoring the included vertices as the graph changes. We know that the cost of an order will not change as long as no vertex adjacent to those in the order, besides those in the order themselves, are eliminated. If some vertex that is not adjacent to any in the “good enough” order is eliminated, then the order remains valid. But if some vertex is eliminated that is not in the order but is adjacent to some vertex that is, then the order is no longer valid. Returning to the example above, say (A, E) is a stored “good enough” order, and we eliminate E and then C from the graph in Figure 2. Since C is adjacent to neither A nor E when eliminated, the “good enough” order remains valid and will be used to avoid eliminating A next. On the other hand, given the graph in Figure 2, if we eliminate B , then the neighborhood of A has changed and the “good enough” order is no longer valid.

6 Duplicate Avoidance & Dominance Criteria

In an ID search, any technique that eliminates all duplicates, whether duplicate detection or duplicate avoidance, will expand the same set of nodes, though they may be reached by different paths. If the ID search is enhanced with other pruning rules then duplicate avoidance may actually expand fewer nodes than duplicate detection. One such type of pruning rule is a *dominance criterion*, which says that, when expanding a node, one or more child nodes will always lead to a solution that is at least as good as any solution following from some other child node. The dominated node can then be pruned. When combined with a dominance criterion, duplicate avoidance can lead to expanding fewer nodes than would duplicate detection with the same dominance criterion. This is a result of the fact that duplicate avoidance prevents duplicates from being generated, while duplicate detection does not. Consider some node n that can be generated by two potential parents, p_1 and p_2 , and assume that both parents are generated and expanded. Suppose that the dominance criterion causes n to be pruned when generated by p_1 , but not when generated by p_2 . Duplicate detection will certainly still generate n via p_2 . Since duplicate avoidance only generates n from a single parent, if that parent is p_1 then it will not generate n at all.

If not implemented correctly this combination may lead to

overzealous pruning. A dominance criterion states that a node can be pruned because there is another node that is at least as good. Duplicate avoidance states that a node can be pruned because there is another path to it that either was or will be explored. These pruning rules can conflict if a node is not generated because the dominance criterion pruned the parents that duplicate avoidance would allow to generate it. This can be fixed by ensuring that duplicate avoidance never prevents a node from generating a child if the parent that it would allow to generate that child was pruned by the dominance criterion.

We should note that this conflict was not mentioned by Gogate and Dechter [2004] and neither was any method of correcting for it. This omission means that their algorithm, which combined IVPR with several dominance criteria to find treewidth, would have incorrectly pruned too many nodes. In our experiments, discussed in Section 8, we adapt IVPR and DVPR to incorporate the correction mentioned above.

7 Existing Tools for Treewidth Search

A useful pruning rule, the *Adjacent Vertex Dominance Criterion (AVDC)*, is based on a result attributed to Dirac (see Gogate and Dechter [2004]). It states that every graph has an optimal vertex elimination order with no adjacent vertices appearing consecutively until the graph becomes a clique. Thus, if the current node was reached by eliminating a vertex v , we can prune all children that result from eliminating any vertex w that is adjacent to v , unless the graph is a clique.

Another useful pruning rule is the *Graph Reduction Dominance Criteria (GRDC)*. Bodlaender *et al.* [2001] developed several criteria for identifying when a graph can be reduced by immediately eliminating some set of vertices without raising its treewidth. When these criteria, known as the *simplicial* and *almost simplicial* rules, identify a vertex for elimination, we can prune all other children of the current node.

The existing state-of-the-art algorithm for treewidth is Breadth-First Heuristic Treewidth (BFHT), developed by Dow and Korf [2007], and enhanced with a faster method of deriving the graph associated with a node, developed by Zhou and Hansen [2008]. A computational bottleneck in BFHT is that each time a node is expanded it must derive the corresponding graph. The original algorithm always derived the node’s graph from the original graph. Zhou and Hansen observed that it could be derived from the graph associated with the last expanded node. They use a large data structure to ensure that nodes with similar graphs are expanded consecutively. Unfortunately, this increases the algorithm’s memory requirement “up to ten times.” We discard this data structure and instead sort the nodes by their state representation: a bit vector where a bit is set if the corresponding vertex has been eliminated. This method is as fast as the memory-based method and without the large memory requirement. This claim is supported by the fact that the performance results reported here on benchmark graphs are comparable to those published by Zhou and Hansen [2008], though the computers and implementations used do differ. Note that BFHT uses GRDC, but it is not obvious how to incorporate AVDC.

The final piece of existing treewidth research that we will employ is a heuristic lower bound. We use the MMD+(least-

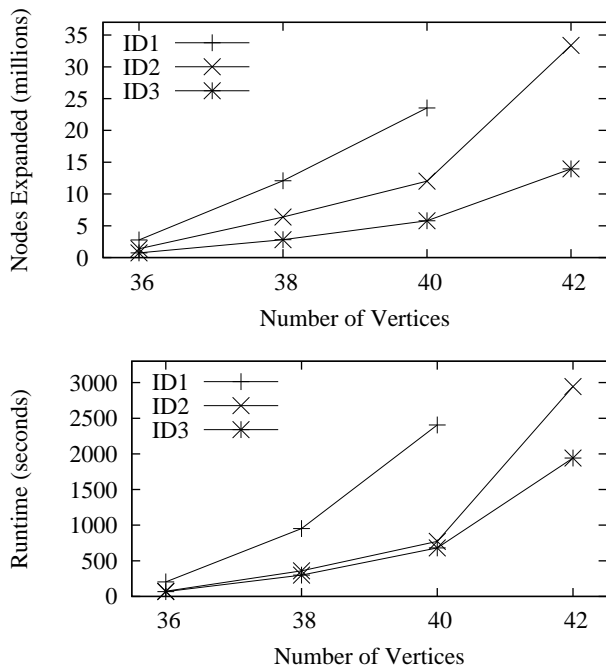


Figure 3: Average nodes expanded (top) and running time (bottom) by ID algorithms on random graphs.

c) heuristic of Bodlaender *et al.* [2004], and we refer readers to the original publication for more details.

8 Empirical Results

Here we evaluate the effectiveness of the techniques discussed in the paper on the treewidth problem. Experiments are conducted on random graphs and benchmark graphs. The random graphs are generated with the following parametric model: given V , the number of vertices, and E , the number of edges, generate a graph by choosing E edges uniformly at random from the set of all possible edges. All of the random graphs used in these experiments have an edge density of 0.2, i.e., $0.1 * V * (V - 1)$ edges. Testing suggested that this density produced the most difficult graphs for various numbers of vertices. The benchmark graphs include Bayesian networks and graph coloring instances. Many of these have been used to evaluate previous treewidth algorithms in the literature.

All algorithms were implemented in C++ and data was gathered on a Macbook with a 2 GHz Intel Core 2 Duo processor running Ubuntu Linux. Although the processor has two cores, the algorithm implementations are single-threaded. The system has 2 GB of memory, but to prevent paging we limited all algorithms to at most 1800 MB.

Our experiments include several variations on the ID search algorithm discussed in this paper. These algorithms employ AVDC, GRDC, and the MMD+(least-c) lower bound.

ID1 ID with a transposition table and LRU replacement.

ID2 Same as ID1 with the addition of IVPR. This is an iterative deepening version of the branch-and-bound algorithm of Gogate and Dechter [2004] with the addition of

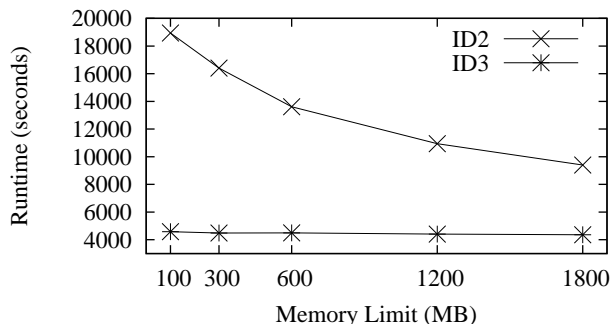


Figure 4: Running time of ID2 and ID3 with various memory limitations on a randomly generated graph with 42 vertices.

the transposition table, the correction to IVPR discussed in Section 6, and the MMD+(least-c) lower bound.

ID3 Same as ID2 with the addition of DVPR.

Since all algorithms used in our experiments (including BFHT) employ iterative deepening, performance data is only presented for the final iteration before the optimal solution is found. This iteration proves that there is no elimination order with width less than the treewidth. The vast majority of the search effort is expended on this iteration.

The first experiments show the effect described in Section 6 where dominance criteria combined with duplicate avoidance result in fewer expanded nodes. Figure 3 shows the average performance of our ID algorithms on ten random graphs with each of 36, 38, 40, and 42 vertices. Because of exceedingly long runtimes, ID1 was not run on the 42-vertex graphs.

The first three data points in Figure 3 show that adding IVPR to ID1 halves the number of nodes expanded, and adding DVPR as well halves them again. These reductions can be attributed to combining duplicate avoidance with dominance criteria. We also see that ID2 and ID3 are three times faster than ID1. ID2 expands half as many nodes as ID1 but only requires a third of the time, because duplicate avoidance prevents duplicate generation as well as expansion. Finally, while ID3 expanded half as many nodes as ID2, it required about the same amount of time to complete. This is because DVPR adds extra processing and, though fewer nodes are expanded, it ends up taking the same time.

The 42-vertex graphs in Figure 3 show a different dynamic between ID2 and ID3. This set includes several graphs where the transposition table was unable to hold every encountered node in the allocated 1800 MB, thus requiring the LRU replacement scheme. The data shows that the addition of DVPR led ID3 to expand a third of the nodes expanded by ID2. Apparently this was enough to make up for the extra computational overhead, because ID3 then runs for two thirds of the time ID2 runs. As one would expect, once the transposition table is unable to prevent all duplicate expansions, the addition of DVPR makes a significant difference.

Our next set of experiments investigates this behavior further. Figure 4 demonstrates the runtime of ID2 and ID3 on a single random graph with various memory constraints. This graph is the hardest instance from the 42-vertex random graph

Graph	Time (sec)			tw
	BFHT	ID2	ID3	
queen6-6	1.3	0.8	2.8	25
queen7-7	109.5	69.2	62.6	35
queen8-8	6941.6	3422.8	2833.1	45
myciel5	155.3	34.5	41.3	19
pigs	mem	*23668.6	*12292.1	9
B_diagnose	28.1	7.3	9.4	13
BN_78	1.5	0.7	0.7	13
BN_79	1.5	0.7	0.7	13
bwt3ac	1.4	0.2	0.2	16
depot01ac	71.4	17.2	25.7	14
driverlog01ac	175.1	12.9	23.7	9

Table 1: Running time on benchmark graphs. ‘mem’ denotes algorithm required > 1800MB of memory and did not complete. ‘*’ denotes that algorithm utilized all 1800MB.

set, and the only instance for which both ID2 and ID3 utilized the entire 1800 MB of available memory. In this experiment, we ran both algorithms while limiting memory usage to 1200, 600, 300, and 100 MB. The figure shows that when restricted to 1800 MB of memory ID3 is twice as fast as ID2, when restricted to 600 MB of memory ID3 is three-times as fast, and when restricted to 100 MB of memory ID3 is four-times as fast. These experiments are not meant to suggest that someone would try to solve treewidth with only 300 or 100 MB of memory. Instead, these experiments demonstrate that DVPR helps our search scale much better when the transposition table can only hold a fraction of the encountered nodes. As we try to find the treewidth of ever larger graphs, we can expect the addition of DVPR to help significantly.

Finally, we compare the performance of our algorithms to the existing state-of-the-art algorithm, BFHT, on benchmark graphs. Table 1 shows the runtime of BFHT, ID2, and ID3 on various benchmark graphs. The first five graphs¹ are graph coloring instances and Bayesian networks used to evaluate previous treewidth search algorithms [Gogate and Dechter, 2004; Dow and Korf, 2007; Zhou and Hansen, 2008]. Note that this is the first time that the exact treewidth for queen8-8 and pigs has been reported. The last six graphs² are graphical models from the Probabilistic Inference Evaluation held at UAI’08. The graphs included here were chosen because they had less than 100 vertices, they were not trivial to solve, and at least one algorithm successfully found the treewidth.

The table shows that ID2 and ID3 outperform BFHT on all graphs by up to an order of magnitude. It also shows that for all but one of the included graphs both ID2 and ID3 were able to store every encountered node in the transposition table. As is consistent with the random graph experiments, when memory is sufficient ID2 and ID3 perform about the same, though on some graphs ID2 is much faster. The most difficult graph is clearly the pigs Bayesian network. The treewidth of this graph was not previously known, and BFHT was unable to complete because of insufficient memory. ID2 and ID3 uti-

¹<http://www.cs.uu.nl/~hansb/treewidthlib>.

²Available at <http://graphmod.ics.uci.edu/uai08>.

lized all 1800 MB and, as is consistent with the random graph results, ID3 outperformed ID2 by a factor of two.

9 Conclusion

Heuristic search on a graph-structured problem space can lead to an exponential increase in the size of the search versus the size of the problem space. In this paper, we have discussed several techniques for duplicate elimination that can prevent this increase. We have shown that duplicate avoidance, as opposed to duplicate detection, can lead to a substantial decrease in the number of expanded nodes when combined with dominance criteria. Additionally, we have presented inductive duplicate avoidance, a duplicate avoidance technique that does not require the a priori identification of symmetries in the search space. We have demonstrated how to implement this technique on the treewidth problem. Our experimental results show that adding duplicate elimination techniques to an iterative deepening search for treewidth consistently outperforms the existing state-of-the-art on hard benchmark graphs. Existing techniques are limited by their memory requirement, and we have shown that our method scales well when memory is limited.

References

- [Arnborg *et al.*, 1987] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic and Discrete Methods*, 8(2):277–284, April 1987.
- [Bodlaender *et al.*, 2001] Hans Bodlaender, Arie Koster, Frank van den Eijkhof, and Linda van der Gaag. Pre-processing for triangulation of probabilistic networks. In *Proc. 17th UAI*, pages 32–39, San Francisco, CA, 2001.
- [Bodlaender *et al.*, 2004] Hans L. Bodlaender, Arie M. C. A. Koster, and Thomas Wollé. Contraction and treewidth lower bounds. In *Proc. 12th European Symposium on Algorithms (ESA-04)*, pages 628–639, January 2004.
- [Bodlaender, 2005] Hans L. Bodlaender. Discovering treewidth. *LNCS*, 3381:1–16, January 2005.
- [Dechter and Pearl, 1989] Rina Dechter and Judea Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38(3):353–366, 1989.
- [Dechter, 1996] Rina Dechter. Bucket elimination: A unifying framework for probabilistic inference. In *Proc. 12th UAI*, pages 211–19, San Francisco, CA, 1996.
- [Dow and Korf, 2007] P. Alex Dow and Richard E. Korf. Best-first search for treewidth. In *Proc. 22nd AAAI*, pages 1146–1151, Vancouver, British Columbia, Canada, 2007.
- [Gogate and Dechter, 2004] Vibhav Gogate and Rina Dechter. A complete anytime algorithm for treewidth. In *Proc. 20th UAI*, pages 201–20, Arlington, Virginia, 2004.
- [Korf, 1985] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.
- [Zhou and Hansen, 2008] Rong Zhou and Eric A. Hansen. Combining breadth-first and depth-first strategies in

searching for treewidth. In *Symp. on Search Techniques in AI and Robotics*, Chicago, Illinois, USA, July 2008.