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 can 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. Additionally, we are able to find, for the first time, the treewidth of several hard benchmark graphs.

1 Introduction

An effective technique for solving many combinatorial optimization problems is heuristic search through 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 managed. 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 uses a list to store expanded nodes. When a new node is generated, we 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. 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 may make a search inadmissible.

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 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 all of 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. Examples include bucket elimination for inference in Bayesian networks, and jointree clustering for constraint satisfaction (discussed in [Bodlaender, 2005; Dow and Korf, 2007]). 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].

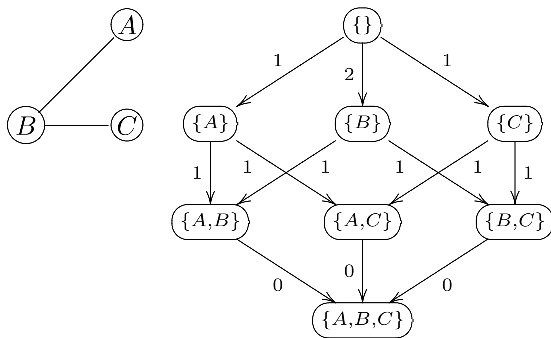

Figure 1: A graph we seek the treewidth of (left), and the corresponding search space (right).

2.2 The Vertex Elimination Order Search Space

A natural search space for treewidth involves constructing optimal vertex elimination orders. Consider searching for an optimal elimination order for the graph on the left in Figure 1; the corresponding search space is on the right. Eliminating a set of vertices results in the same 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 that a solution 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 [Dow and Korf, 2008].

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 depth-first search where solutions are only explored if their cost does not exceed the cutoff. If an iteration is completed without finding a solution, then all solutions must cost more than the cutoff, thus the cutoff is increased and the search is repeated. ID is effective for problems where the time required to search grows significantly with each increase of the cutoff.

ID is typically applied to problems with an additive cost function, where, in a single iteration, it may be necessary to expand the same node more than once. This is because, with additive costs, whether a solution is found from a node depends on the cost of the path taken to the node and the search space below that node. Thus, if a lower-cost path is found to a node that has already been expanded, the node must be expanded again to see if the new path leads to a solution.

In contrast to the case of additive costs, ID on a maximum edge cost search space never needs to expand the same node more than once per iteration. Notice that, with maximum edge costs, the cost of a path to a node doesn't matter, as long as it does not exceed the cutoff. If we generate a node that was previously expanded and we have not yet found a solution, then we know there is no valid path from that node to a goal node. Thus, even if there is a lower-cost path to that node, it also will not lead to a solution. Therefore, in the course of an iteration, if we ever generate a node that was previously expanded, we can prune 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 generated 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 generations of 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 expanded 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 expanded 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 expanded. 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 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. 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. Methods for identifying duplicate operator sequences and using them to avoid duplicate nodes are specific to a particular problem. Next, we will show how this is 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 redundant paths to a node. Gogate and Dechter [2004] present several rules that accomplish this in the elimination order search space for treewidth. These rules are based on the fact that eliminating

a vertex does not add or remove edges to vertices to which it is not adjacent. Thus, a pair of non-adjacent vertices can be eliminated in either order at the same cost. Suppose we have two non-adjacent vertices v and w, and an ordering over the vertices such that $v < w$. We will explore solutions where v is eliminated before w, but we will prune some solutions where w is eliminated before v. If w is eliminated, we will not allow v to be eliminated as long none of the vertices adjacent to v are eliminated in the interim. If w is eliminated, and then some vertex adjacent to v is eliminated, the effect of eliminating v is no longer the same as it was before we eliminated w. Thus, in this case, we will eliminate v after w.

We will refer to this duplicate avoidance process as the *Independent Vertex Pruning Rule (IVPR)*. It is described more formally as Theorems 6.1 and 6.2 of Gogate and Dechter [2004], where more details can be found. In related work, Bošnački *et al.* [2007] present a more general method for avoiding redundant paths to a node given an *independence relation* on operators or actions.

Our experiments (Section 8) show that IVPR eliminates a large number of duplicates, though many remain. The next section attempts to prune some of these remaining duplicates.

5 Inductive Duplicate Avoidance

As we saw in the previous section, IVPR avoids duplicates due to eliminating non-adjacent vertices because all permutations of those eliminations will have the same cost. Unfortunately, when considering vertices that are adjacent to each other, we no longer know, a priori, how the costs of the different permutations relate. At some point in an iteration of ID search, if the cost of one of these permutations exceeds the cutoff then that operator sequence isn't valid. If more than one don't exceed the cutoff, then they represent duplicates. In this section, we discuss a technique for recognizing valid duplicate operator sequences and avoiding all but one of them. We refer to this as *inductive* duplicate avoidance, because instead of relying on a priori knowledge about the search space, it avoids duplicates with information gathered during search.

5.1 A General Procedure

Suppose that, at some point in an ID search, we recognize that we have applied some sequence of operators from a set of duplicate operator sequences. Since this 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, our search is ID and 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. We accomplish this by storing "good enough" sequences as they are discovered. As the search progresses, we can check whether any child of the current node would represent a duplicate of some "good enough" sequence. If it does we can prune that child, and avoid generating the corresponding duplicate.

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. In the case of treewidth, as we will see next, we limit it to only certain types of sequences.

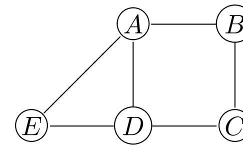

Figure 2: A graph we seek the treewidth of.

5.2 Inductive Duplicate Avoidance for Treewidth

Consider searching for the treewidth of the graph in Figure 2, with the following arbitrary ordering of the vertices: $A < B < C < D < E$. There are six possible duplicate sequences for eliminating vertices A, C, and E; and IVPR avoids four of them, leaving: (A, C, E) and (C, E, A). The duplicates that IVPR does not avoid are due to adjacent vertices, in this case A and E. Thus, the two remaining duplicate sequences represent the two permutations of the adjacent vertices. 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 an iteration of 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.

We will attempt to avoid the duplicate sequences left by IVPR. These sequences include permutations of adjacent vertices, therefore we will refer to this duplicate avoidance technique as the *Dependent Vertex Pruning Rule (DVPR)*. The key aspects of implementing this technique are (1) identifying and storing "good enough" sequences, (2) determining when eliminating a vertex will lead to a duplicate and should thus be pruned, and (3) determining when stored "good enough" sequences are valid.

The first task is to identify and store "good enough" sequences. Once a node n is generated by eliminating a vertex v, we examine the vertex eliminations that make up the path taken to node n. We identify the sequence of *dependent vertices*, i.e., v, the vertices adjacent to v, and the vertices adjacent to other dependent vertices. For example, say we just generated the node that follows from the elimination sequence (A, C, E) in Figure 2. We then identify (A, E) as the sequence of dependent vertices. We ignore the elimination of C because it had no impact on A or E. Once a dependent vertex sequence is identified, we consider it a "good enough" sequence for eliminating the included vertices, and we store it in a hash table. The key to the hash table is the set of vertices in the sequence, thus duplicate sequences have the same key.

The second task is to use the stored "good enough" sequences to avoid duplicates. If we have determined that some sequence is "good enough," then any other permutation of the included vertices will result in a duplicate. When expanding a node, before generating each of the children we check to see if they correspond to a duplicate sequence. For a child node that results from eliminating a vertex v, we append v to the end of the sequence of eliminations that make up the path to the current node. Then we find the sequence of depen-

dent vertices that include v, just as we discussed above. We then look up that sequence in the hash table. If a sequence with the same key is found, then it is either the same as the current sequence or a duplicate of the current sequence. If it is a duplicate, then we can prune the corresponding child node. Continuing the example above, say we have stored "good enough" sequence (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" sequence. On the other hand, if A is eliminated first, we will notice that eliminating E next is the same as the "good enough" sequence, 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" sequence is valid. We consider a sequence valid when we know that it can be executed without being pruned by the cutoff. Thus, a sequence is valid as long as the relevant parts of the graph are the same as they were when the sequence was discovered. Those parts include any vertex adjacent to a vertex in the sequence. If one of those vertices has been eliminated, excluding those in the sequence themselves, then the cost and effect of executing the sequence will have changed. When this occurs we no longer know if the sequence is "good enough," therefore we purge it from the hash table. If some vertex that is not adjacent to any in the "good enough" sequence is eliminated, then the sequence remains valid. Returning to the example above, say (A, E) is a stored "good enough" sequence, 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" sequence 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" sequence is no longer valid.

5.3 Related Work

Taylor and Korf [1993] describe a technique that uses a breadth-first search to discover duplicate operator sequences. They then construct a finite state machine that recognizes those sequences, such that, during a depth-first search, they can avoid executing them. They apply this technique to the sliding tile puzzle. This technique makes assumptions about the problem space that do not hold for treewidth, such as that operators have the same cost throughout the search. Thus, this technique does not directly apply to treewidth.

6 Duplicate Avoidance & Dominance Criteria

In a simple 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 some node leads to a solution that is no worse than any solution following some other node, i.e., the first node *dominates* the second 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, thus making the search inadmissible. 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. In our experiments, discussed in Section 8, we adapt IVPR and DVPR to incorporate this correction.

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 bitstring where a bit is set if the corresponding vertex is eliminated. This method is as fast as the memory-based method and without the large memory requirement. Note that BFHT

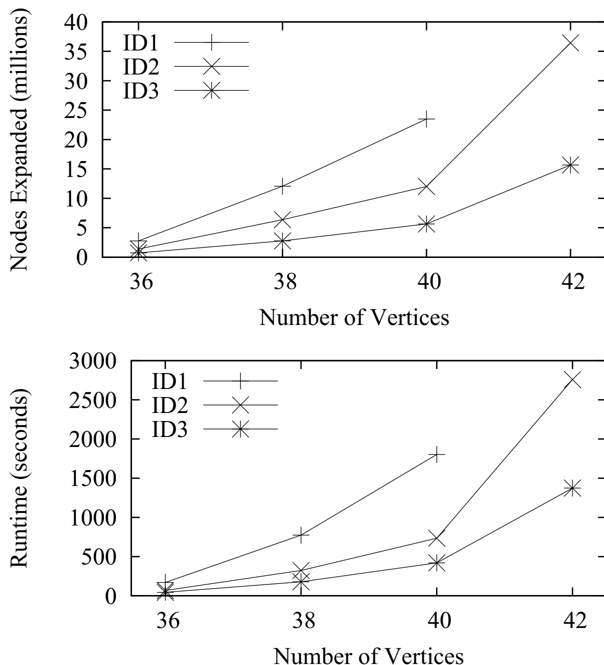

Figure 3: Average nodes expanded (top) and running time (bottom) by ID algorithms on random graphs.

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-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 this 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.

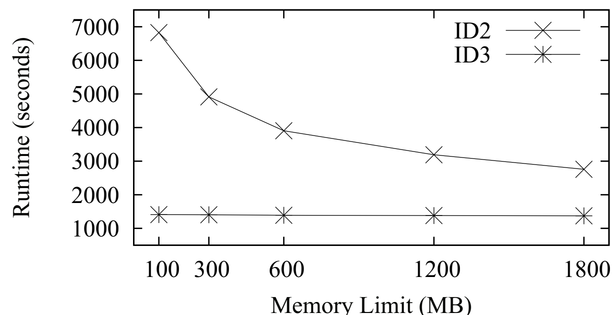

Figure 4: Average running time of ID2 and ID3 with various memory limitations on 42-vertex random graphs.

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 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.

Figure 3 shows 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 is about 2.5 times faster than ID1, which is a larger improvement than the number of node expansions. This is because duplicate avoidance also decreases the number of node generations. The improvement in running time from ID2 to ID3 is not as large as the improvement in node expansions, because the addition the DVPR adds extra computational overhead to each expansion. In the set of 42-vertex graphs, we see that the runtime improvement of ID3 over ID2 is more significant than in the smaller graphs. This can be attributed to the fact that this set includes several graphs where the transposition table was unable to hold every expanded node in the allocated 1800 MB, thus requiring the LRU replacement scheme. The next set of experiments investigates how these algorithms scale when memory is limited.

Figure 4 demonstrates the runtime of ID2 and ID3 on the set of 42-vertex graphs with various memory constraints. We ran both algorithms while limiting memory usage to 1800, 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 almost three

Graph	Time (sec)			tree-width
	BFHT	**ID2**	**ID3**	
queen6-6	1.3	0.8	0.9	25
queen7-7	109.5	66.5	62.9	35
queen8-8	6941.6	3291.9	2854.8	**45**
myciel5	155.3	33.8	33.3	19
pigs	mem	*17038.4	*5289.1	**9**
B_diagnose	28.1	7.5	6.6	13
bwt3ac	1.4	0.1	0.1	16
depot01ac	71.4	17.1	11.8	14
driverlog01ac	175.1	12.6	10.5	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.

times as fast, and when restricted to 100 MB of memory ID3 is almost five times as fast. These experiments are not meant to suggest that we want to solve treewidth with only 300 or 100 MB of memory. Instead, they demonstrate that DVPR helps our search scale much better when the transposition table can only hold a fraction of the expanded nodes. As we try to find the treewidth of larger graphs, we 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 graphs[1] 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 four graphs[2] 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 are several times faster than BFHT on all graphs. 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 ID3 is a little faster than ID2, 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 utilized all 1800 MB and, as is consistent with the random graph results, ID3 outperformed ID2 by a factor of three.

9 Conclusion

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 based on discovering duplicate operator sequences during search. 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. This has allowed us to, for the first time, find the exact treewidth of two hard benchmark graphs.

[1] `http://www.cs.uu.nl/~hansb/treewidthlib`.
[2] Available at `http://graphmod.ics.uci.edu/uai08`.

Acknowledgments

This research was supported by NSF grant No. IIS-0713178 to Richard Korf.

References

[Arnborg *et al.*, 1987] Stefan Arnborg, Derek Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J Algbrc & Dscrt Mthds*, 8(2), 1987.

[Bodlaender *et al.*, 2001] Hans Bodlaender, Arie Koster, Frank van den Eijkhof, and Linda van der Gaag. Preprocessing 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 Wolle. 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.

[Bošnački *et al.*, 2007] Dragan Bošnački, Edith Elkind, Blaise Genest, and Doron Peled. On commutativity based edge lean search. In *ICALP*, pages 158–170, 2007.

[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.

[Dow and Korf, 2008] P. Alex Dow and Richard E. Korf. Best-first search with a maximum edge cost function. In *Proc. 10th Int'l Sym on AI and Math*, Ft Lauderdale, 2008.

[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.

[Taylor and Korf, 1993] Larry A. Taylor and Richard E. Korf. Pruning duplicate nodes in depth-first search. In *AAAI*, pages 756–761, 1993.

[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.