Combining Breadth-First and Depth-First Strategies in Searching for Treewidth

Rong Zhou
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304
rzhou@parc.com

Eric A. Hansen
Dept. of Computer Science and Engineering
Mississippi State University
Mississippi State, MS 39762
hansen@cse.msstate.edu

Abstract

Breadth-first and depth-first search are basic search strategies upon which many other search algorithms are built. In this paper, we describe an approach to integrating these two strategies in a single algorithm that combines the complementary strengths of both. We show the benefits of this approach using the treewidth problem as an example.

1 Introduction

Breadth-first and depth-first search are basic search strategies upon which many other search algorithms are built. Given the very different way in which they order node expansions, it is not obvious that they can be combined in the same search algorithm. In this paper, we describe an approach to integrating these two strategies in a single algorithm that combines the complementary strengths of both. To illustrate the benefits of this approach, we use the treewidth problem as an example.

The treewidth of a graph (also known as the induced treewidth) is a measure of how similar the graph is to a tree, which has a treewidth of 1. A completely connected graph is least similar to a tree, and has a treewidth of $n - 1$, where n is the number of vertices in the graph. Most graphs have a treewidth that is somewhere in between 1 and $n - 1$.

There is a close relationship between treewidth and vertex elimination orders. Eliminating a vertex of a graph is defined as follows: an edge is added to every pair of neighbors of the vertex that are not adjacent, and all the edges incident to the vertex are removed along with the vertex itself. A vertex elimination order specifies an order in which to eliminate all the vertices of a graph, one after another. For each elimination order, the maximum degree (i.e., the number of neighbors) of any vertex when it is eliminated from the graph is defined as the width of the elimination order. The treewidth of a graph is defined as the minimum width over all possible elimination orders, and an optimal elimination order is any order whose width is the same as the treewidth.

Many algorithms for exact inference in Bayesian networks are guided by a vertex elimination order, including Bucket Elimination [Dechter, 1999], Junction-tree elimination [Lauritzen and Spiegelhalter, 1988], and Recursive Conditioning [Darwiche, 2001]. In fact, the complexity of all of these algorithms is exponential in the treewidth of the graph induced by the network. For these algorithms, use of a suboptimal elimination order leads to inefficiency, and improving an elimination order by even small amount can result in large computational savings. Solving the treewidth problem exactly, and finding an optimal elimination order, allows these algorithms to run as efficiently as possible.

2 Previous work

Finding the exact treewidth of a general graph is an NP-complete problem [Arnborg *et al.*, 1987]. One approach to finding the exact treewidth is depth-first branch-and-bound search in the space of vertex elimination orders [Gogate and Dechter, 2004]. However, Dow and Korf [2007] showed that best-first search can significantly outperform depth-first branch-and-bound search by avoiding repeated generation of duplicate search nodes.

In the search space of the treewidth problem, each node corresponds to an *intermediate graph* that results from eliminating a set of vertices from the original graph. Figure 1 shows the treewidth search space for a graph of 4 vertices. Each oval represents a search node that is identified by the set of vertices eliminated so far from the original graph. A path from the start node (which has an empty set of eliminated vertices) to the goal node (which has all vertices eliminated) corresponds to an elimination order, and there is a one-to-one mapping from the set of elimination orders to the set of paths from the start to the goal node. Although there are $n!$ different elimination orders for a graph of n vertices, there are only 2^n distinct search nodes. This is because different ways of eliminating the same set of vertices always arrive at the same intermediate graph [Bodlaender *et al.*, 2006], and there is only one distinct intermediate graph for each combination (as opposed to permutation) of the vertices. Depth-first branch-and-bound search treats the search space as a tree with $n!$ distinct states instead of a graph with only 2^n states. The faster performance of best-first treewidth search reflects the difference in size between a search tree and a search graph [Dow and Korf, 2007].

Unfortunately, the scalability of best-first (treewidth) search is limited by its memory requirements, which tend to grow exponentially with the search depth. To improve scalability, Dow and Korf use a memory-efficient version of best-first search called *breadth-first heuristic search* [Zhou and Hansen, 2006], which, like *frontier search* [Korf *et al.*, 2005], only stores the search frontier in memory and uses a divide-and-conquer approach to reconstruct the solution path after

the goal is reached. In fact, they use a variant of breadth-first heuristic search, called *SweepA** [Zhou and Hansen, 2003], that exploits the fact that the search graph for the treewidth problem is a partially ordered graph. A *partially ordered graph* is a directed graph with a layered structure, such that a node in one layer can only have successors in the same layer or later layers. This allows layers to be removed from memory after all their nodes are expanded. SweepA* expands all the nodes in one layer before considering any nodes in the next layer, and uses an admissible heuristic and an upper bound to prune the search space.

Besides exploiting the layered structure of the search graph using SweepA*, there is another important way in which the search algorithm of Dow and Korf limits use of memory. Because the size of an intermediate graph can vary from several hundred bytes to a few megabytes, storing an intermediate graph at each search node is impractical for all but the smallest problems. Instead, Dow and Korf store with each search node only the set of vertices that have been eliminated so far. Each time a node is expanded, its corresponding intermediate graph is generated on-the-fly by eliminating from the original graph those vertices stored with the node. While this approach is space-efficient, it incurs the overhead of intermediate graph generation every time a node is expanded. For large graphs, this overhead is significant. In this paper, we describe a technique that eliminates much of this overhead.

3 Meta search space

To reduce the time overhead of intermediate graph generation, we describe a search algorithm that does not generate the intermediate graph from the original graph at the root node of the search space. Instead, it generates it from the intermediate graph of a close neighbor of the node that is being expanded. The advantage is that the intermediate graph of a close neighbor is already very similar, and so there is much less overhead in transforming it into a new intermediate graph. The simplest way to find the node's closest neighbor is by computing shortest paths from a node to all of its neighbors and picking the closest one. But at first, this does not seem to work for the treewidth problem, since its state space is a partially ordered graph in which the distance between any pair of nodes at the same depth is infinite.

Our idea is to measure the distance between a pair of nodes in a *meta search space*, instead of the original search space. A meta search space has exactly the same set of states as the original search space, but is augmented with a set of meta actions that can transform one node into another in ways not allowed in the original search space. For example, a meta action for the treewidth problem can be an action that "uneliminates" a vertex by reversing the changes made to a graph when the vertex was eliminated. For the treewidth problem augmented with the "uneliminate" meta action, its search graph is an undirected version of the graph shown in Figure 1. In this new graph, called a meta search graph, actions (i.e., edges) can go back and forth between a pair of adjacent nodes, and this allows us to generate the intermediate graph of a node from another node at the same depth. This is very useful for breadth-first heuristic search, which expands nodes in order of their depth in the search space.

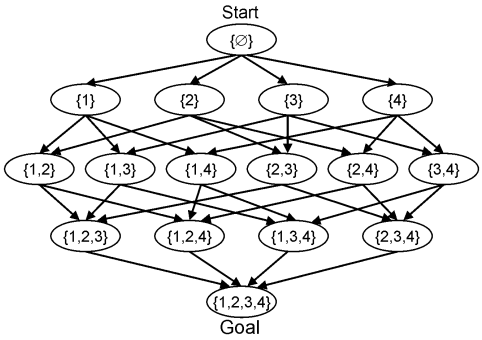

Figure 1: The search space of treewidth for a graph of 4 vertices. Each oval represents a search node identified by the set of vertices eliminated so far. The start node corresponds to the original graph with an empty set of eliminated vertices and the goal node is the one with all the vertices eliminated.

Since a node is uniquely identified by the set of vertices eliminated, we use the same lower-case letter (e.g., n, u, and v) to denote both a node and a set of eliminated vertices in the rest of this paper. To implement the "uneliminate" meta action, each edge of the meta search graph is labeled by a tuple $\langle u, v, \Delta E^+, \Delta E^- \rangle$, where u (v) is the set of vertices eliminated so far at the source (destination) node of the edge, and ΔE^+ (ΔE^-) is the set of edges added to (deleted from) the graph when the vertex in the singleton set $v \setminus u$ is eliminated. Let $G_n = \langle V_n, E_n \rangle$ be the intermediate graph associated with node n. The task of adding a previously-eliminated vertex back to the graph can be expressed formally as: given $G_v = \langle V_v, E_v \rangle$ and $e = \langle u, v, \Delta E^+, \Delta E^- \rangle$, how to compute $G_u = \langle V_u, E_u \rangle$? Since all the changes are recorded with the edge e, one can reconstruct $G_u = \langle V_u, E_u \rangle$ as follows,

$$V_u = V_v \cup v \setminus u \quad (1)$$

$$E_u = E_v \cup \Delta E^- \setminus \Delta E^+ \quad (2)$$

That is, by adding (deleting) the edges that have been previously deleted (added) to the graph, the "uneliminate" meta action can undo the effects of an elimination action in the original search space.

In general, adding meta actions can turn directed search graphs into undirected graphs. This guarantees that any changes made to the current state (e.g., the intermediate graph) is reversible, creating a graph with the following appealing property: for *any* two states reachable from the start state, there is *always* a path that maps one into the other. This property allows a search algorithm to generate the state representation of a node from *any stored node*, because if all actions are deterministic, then a state s' is uniquely identified by another state s plus a path from s to s'. If it takes less space to represent a path between s and s', then this approach to state encoding can save memory, although at the cost of some computational overhead.

For the treewidth problem, this means the intermediate graph of a node can be generated from any node instead of only from the node's direct ancestors, such as the start node. Thus, one only needs to maintain a single intermediate graph, which can be modified to become the intermediate graph for any node in the search space. An interesting question is how to minimize the overhead of generating the intermediate

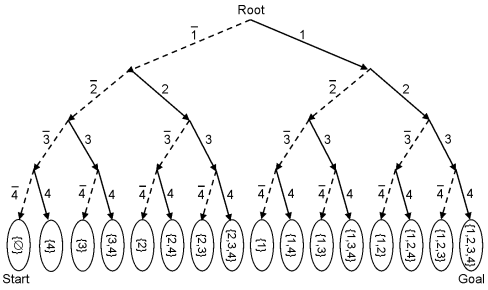

Figure 2: A binary decision tree in which frontier nodes are stored as leaves of the tree. For simplicity, non-leaf nodes are not shown.

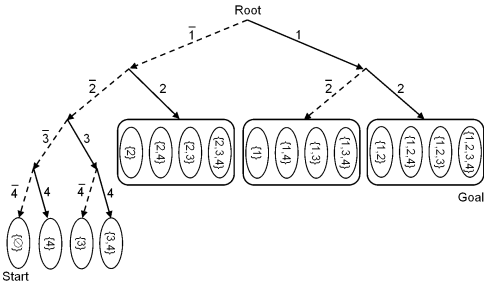

Figure 3: A partial binary decision tree in which the currently expanding nodes are stored as full-depth leaves and non-expanding nodes as shallow-depth leaves.

graph from one node to another. The answer depends on the search strategy, because ultimately our goal is to minimize the overhead of expanding not just a single node but a set of nodes.

4 Frontier decision tree

The solution we propose is to use an ordered decision tree to store the set of frontier nodes at the current depth of breadth-first heuristic search as the leaves of the decision tree. Unlike explicit-state search methods, this approach can be viewed as a variant of symbolic search in which the state-representation similarities among a set of nodes are retained in the decision tree and exploited by the search algorithm.

A decision tree is defined as a rooted tree in which every non-leaf node is a decision node that performs a test on a variable, the value of which is then used to determine recursively the next decision node until a leaf node is reached. A decision tree is commonly used to represent a discrete function over a set of variables. For the treewidth problem, these are Boolean variables, one for each vertex. A Boolean variable has the value of true if its corresponding vertex has been eliminated.

To make operations on decision trees more efficient, an ordering constraint is usually enforced that requires the order in which variables are tested to be the same on any path from the root to a leaf node. The resulting data structure is called an ordered binary decision tree, and an example is shown in Figure 2. In this example, variables are tested in increasing order of vertex number: 1, 2, 3, then 4. A solid (dashed) edge represents a truth (false) assignment to the variable being tested at the source node of the edge. A leaf node corresponds to a complete assignment to all the variables, and there is a one-to-one mapping from the set of leaf nodes shown in Figure 2 to the set of search nodes shown in Figure 1. From now on, we call a decision tree that stores the frontier nodes of a search graph in this way a *frontier decision tree*.

To support meta actions, each edge of the frontier decision tree stores "undo" information as needed by Equations (1) and (2). Note that for an undirected search space, there is no need to store such information.

Variable ordering We use a frontier decision tree in order to reduce the time overhead for regenerating the intermediate graph from the root node. But this must be weighed against the space overhead for storing the decision tree.

An important way to save space in a frontier decision tree is to find a good ordering of the variables, which affects the number of decision nodes needed to represent a set of frontier nodes. While finding an optimal ordering is a hard combinatorial optimization problem in itself, good orders can often be found quickly by using simple heuristics. We tried three variable-ordering heuristics, the details of which are described in the computational results section.

A decision node is useless if it does not lead to any leaf node. To prune these useless decision nodes, we store a leaf-node counter at each decision node. Each time a leaf node is deleted, all of its ancestor decision nodes decrease their leaf-node counters by one, and a decision node is deleted as soon as its leaf-node counter reaches zero. Once useless nodes are pruned, the (asymptotic) space complexity of a frontier decision tree is the same as that of an explicit-state representation of the same nodes. In practice, however, their actual memory requirements can differ by a constant, since a node in an explicit-state search can be compactly represented as a bit-vector; whereas a decision node needs more space to store, even though the total number of decision nodes is usually much less than the total number of bits needed to encode the same nodes, due to state aggregation in a decision tree.

Partial frontier decision tree To further reduce the space overhead of using a frontier decision tree, we introduce a hybrid data structure called a *partial frontier decision tree* that combines the complementary strengths of both symbolic and explicit-state representations. The idea is to generate the decision nodes of the tree on the fly as its leaf nodes are expanded, so that only a (small) subset of frontier nodes that are selected for expansion need to be represented in decision tree form. A partial decision tree has two kinds of leaves; a *full-depth* leaf, which is uniquely identified by a complete path whose length equals the number of vertices in the original graph, and a *shallow-depth* leaf, which is identified by an incomplete path. Figure 3 shows an example of a partial frontier decision tree in which only the currently expanding nodes are represented as full-depth leaves in the tree. Because an incomplete path can lead to a *set* of shallow-depth leaves, a bit-vector is stored at each frontier node to specify its "remaining" path.

Structured duplicate detection By itself, a partial frontier decision tree cannot significantly reduce the overhead of generating the intermediate graphs, because it still needs a search strategy that can exploit the similarities among a set of nodes. For this reason, we use a technique called *structured duplicate detection* (SDD) [Zhou and Hansen, 2004b]. Nodes in SDD are partitioned into buckets, one for each abstract state

defined by a state-space projection function. To exploit locality, SDD expands nodes in the same bucket consecutively. In the case of a partial decision tree, SDD expands full-depth leaves first, since their decision-tree representation is complete. Upon selecting a new bucket for expansion, SDD converts all its shallow-depth leaves into full-depth leaves before expanding them. Since the conversion is done automatically by SDD, the search algorithm has the "illusion" that it is working with the full (instead of partial) frontier decision tree. Thus, for clarity, we will ignore the difference between full and partial frontier decision trees, until we come back to it in the computational-results section.

5 Depth-first search in frontier decision tree

The purpose of using an ordered binary decision tree to store the set of frontier nodes is twofold. First, it reveals the similarities among the frontier nodes, because nodes with the same prefix (according to the test ordering) share the same ancestor node in the decision tree. For example, because nodes $\{1,2,3\}$ and $\{1,2,3,4\}$ share the same prefix $\{1,2,3\}$, they have the same parent node in the decision tree. On the other hand, because nodes $\{\emptyset\}$ and $\{1,2,3,4\}$ have nothing in common, their common ancestor is only the root node. Second, the tree topology guarantees there is a unique path from the root to a leaf node. This facilitates the use of a tree-search algorithm such as depth-first search to determine the order in which frontier nodes are expanded.

It is well-known that depth-first search has excellent memory-reference locality. This is particularly well suited for decision trees, since a depth-first search of a decision tree always visits nodes with the same prefix before visiting nodes with different prefixes, and the *longer* the prefix shared by two nodes, the *closer* they will be visited in depth-first search. For the treewidth problem, this means that if two nodes have similar intermediate graphs, they will be expanded close to each other, and the more similar their intermediate graphs, the closer together they will be expanded. To minimize the intermediate-graph generation overhead for the entire set of frontier nodes at the current search depth, we use depth-first traversal of the decision tree, which visits all leaf nodes of the decision tree. Thus, our treewidth algorithm adopts a hybrid search strategy that uses depth-first traversal in a symbolic (e.g., decision-tree) representation of the (meta) search graph to determine the order of node expansions for the current depth of breadth-first heuristic search (or nodes with the same minimum f-cost in the case of A*). The depth-first search aspect essentially serves as a tie-breaking strategy in breadth-first (or best-first) search to improve its memory-reference locality; in the case of treewidth computation, it also reduces the overhead of generating the intermediate graphs.

Example Figure 4 shows an example how depth-first search can reduce the intermediate-graph generation overhead in breadth-first heuristic search for treewidth. Suppose the three leaves shown in the figure are the frontier nodes of breadth-first heuristic search, and the intermediate graph has already been generated for node $\{1,3,4\}$. Depth-first search will visit node $\{1,2,4\}$ next, and then node $\{1,2,3\}$. A sequence of dark (solid and dashed) arrows represents the order in which

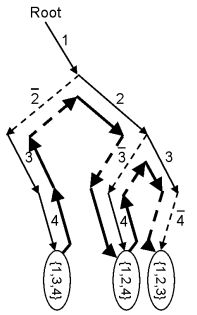

Figure 4: Depth-first search in a binary decision tree can be used to order node expansions in breadth-first treewidth computation.

actions are taken to "migrate" the intermediate graph from node $\{1,3,4\}$ to node $\{1,2,4\}$ and then to node $\{1,2,3\}$. A solid dark arrow moving towards the leaf (root) represents an action that eliminates (uneliminates) a vertex. Dashed arrows represent no-op actions that simply move an intermediate graph around in the decision tree without changing its content. The action sequence shown in Figure 4 starts with a meta action (shown as a dark, upward arrow from node $\{1,3,4\}$) that "uneliminates" vertex 4 from the intermediate graph $G_{\{1,3,4\}}$ in order to generate $G_{\{1,3\}}$. Then vertex 3 is "uneliminated" to generate $G_{\{1\}}$. Next, vertex 2 is eliminated from the intermediate graph to generate $G_{\{1,2\}}$, and then vertex 4 is eliminated to arrive at $G_{\{1,2,4\}}$. To migrate from node $\{1,2,4\}$ to node $\{1,2,3\}$, vertex 4 is "uneliminated" and then vertex 3 is eliminated to generate the intermediate graph $G_{\{1,2,3\}}$. Because an "uneliminate" meta action does not need to check for connectivity between all possible pairs of a vertex's neighbors, it is usually much cheaper than eliminating a vertex. Thus, we only count the number of times an elimination action is performed as overhead. In this example, there are altogether 3 elimination actions. For comparison, generating the intermediate graphs for nodes $\{1,2,3\}$ and $\{1,2,4\}$ from the root node would require 6 elimination actions (3 for node $\{1,2,3\}$ and 3 for node $\{1,2,4\}$), which is (almost) twice as expensive.

Note that the benefit of our approach increases as the search frontier moves further away from the root node. For example, if the three leaf nodes in Figure 4 are a hundred elimination steps away from the root node, then it will take about 200 elimination actions to regenerate the intermediate graphs for nodes $\{1,2,4\}$ and $\{1,2,3\}$; whereas it still takes the same number of elimination actions (3) with our approach, no matter how deep these nodes are, as long as the intermediate graph has just been generated for a node ($\{1,3,4\}$ in this example) that is close by. Moreover, the overhead of elimination actions can differ significantly depending on the size and topology of the intermediate graph. Because intermediate graphs have fewer vertices as the search goes deeper, elimination actions tend to become cheaper as frontier nodes move further away from the root. In other words, the overhead of the 3 elimination actions needed in our approach is probably cheaper than 1.5% (i.e., $3/200$) of the overhead incurred by generating the intermediate graphs from the root, if the overhead differences in elimination actions are accounted for.

6 Computational results

Both random graphs and benchmark graphs were used in our experiments. Given the number of vertices V, a random graph is generated by selecting a fixed number of edges E uniformly from the set of $V(V-1)/2$ possible edges. All experiments were run on a machine with two Intel 2.66 GHz Xeon dual-core processors and 8 GB of RAM, although the search algorithm never used more than 4 GB of RAM; no multi-threading parallelization was used. As in previous studies [Gogate and Dechter, 2004; Dow and Korf, 2007], the same admissible heuristic called MMD+(least-c) [Bodlaender *et al.*, 2004] was used in all our experiments.

Recall that the order in which decision variables are tested can affect the size of an ordered decision tree. We tested three variable-ordering heuristics: (a) a random ordering heuristic, (b) a minimum-degree-vertex-first heuristic, which orders the variables in increasing degrees of their corresponding vertices, and (c) a maximum-degree-vertex-first heuristic, which does the opposite. Results indicate that the random ordering and minimum-degree-vertex-first heuristics store on average 40% and 135% more decision nodes than the maximum-degree-vertex-first heuristic in solving random graphs, respectively. Thus, the maximum-degree-vertex-first heuristic was used for all the experimental results reported here.

Next we studied two different strategies for caching the "undo" information at the edges of the frontier decision tree. The *cache-until-removal* strategy stores "undo" information for every edge of the decision tree until the edge is removed due to the pruning of some decision node. The *cache-until-backtrack* strategy stores "undo" information until the depth-first traversal of the decision tree backtracks from the edge to the source decision node of that edge. In other words, it only stores "undo" information along the current "stack" of the depth-first traversal. Thus, the maximum number of edges for which "undo" information is stored cannot exceed the depth of the decision tree, which is bounded by the number of vertices in the original graph. Because the memory requirements depend on the complexity of the "undo" information measured in terms of the size of ΔE^+ and ΔE^-, our implementation keeps track of the maximum number of edges included in all such ΔE sets, which reflects accurately the total amount of memory used for storing "undo" information over the entire decision tree. With the cache-until-removal strategy, the average peak number of ΔE edges cached is 7,253,520 edges. This number decreased to about 405 edges when the cache-until-backtrack strategy was used, reducing the number of ΔE edges by a factor of over 17,900 times! Surprisingly, this has little effect on the average running time of the algorithm; using the cache-until-backtrack strategy increased the average running time by less than 1.7%, which is hardly noticeable. We also compared the treewidth solution and the number of node expansions of the two caching strategies for all random graphs we tested to make sure the results are correct. Results in the rest of this section were obtained by using the cache-until-backtrack strategy only.

We used breadth-first heuristic search as the underlying search algorithm, which needs an upper bound to prune nodes with an f-cost greater than or equal to the upper bound. While an upper bound for treewidth can be quickly computed by using the minimum fill-in (min-fill) heuristic, in our experiments we used divide-and-conquer beam search [Zhou and Hansen, 2004a] which can usually find tighter upper bounds. A beam-search variant of breadth-first heuristic search, divide-and-conquer beam search (DCBS) limits the maximum size of a layer in the breadth-first search graph. When memory is full (or reaches a predetermined bound), DCBS recovers memory by pruning the least-promising nodes (i.e, the nodes with the highest f-cost) from the Open list before it continues the search.

	Full decision tree			Partial decision tree		
Tw	Dnode	Exp	Sec	Dnode	Exp	Secs
14	799K	160K	9.0	117K	160K	12.2
14	998K	322K	15.7	123K	322K	18.3
15	710K	327K	13.3	97K	326K	16.2
15	875K	334K	14.2	111K	333K	17.2
15	1,442K	517K	25.0	148K	515K	28.9
16	5,701K	3,629K	193.9	270K	3,630K	202.2
16	7,055K	3,826K	211.8	415K	3,819K	223.7
16	6,181K	4,431K	215.8	397K	4,424K	227.0
16	6,900K	4,677K	230.8	389K	4,672K	243.1
17	9,619K	6,232K	342.5	367K	6,231K	350.8

Table 1: Comparison of full and partial frontier decision trees. Columns show the treewidth (Tw), peak number of decision nodes stored in thousands (Dnode), number of nodes expanded in thousands (Exp), and running time in CPU seconds (Sec). The horizontal line separates 5 easiest instances from the 5 hardest in a set of 100 random graphs.

Graph	Ub	Lb	Tw	Stored	Exp	Sec
queen5_5	18	12	18	961	1,294	0.1
david	13	10	13	483	2,009	0.4
queen6_6	25	15	25	11,995	13,353	1.6
miles500	22	21	22	2	2	2.3
inithx.i.1	56	55	56	209	370	30.8
queen7_7	35	18	35	597,237	935,392	149.6
myciel5	19	14	19	678,540	3,418,309	192.3

Table 3: Performance of breadth-first heuristic search on benchmark graphs. Columns show the upper bound found by divide-and-conquer beam search (Ub), the heuristic value for the start node (Lb), the treewidth (Tw), the peak number of frontier nodes stored (Stored), the number of node expansions (Exp), and running time in CPU seconds (Sec).

Using a memory bound of 64 nodes, DCBS finds the exact treewidth for 97 out of the 100 random graphs. For the remaining three graphs, its solutions are very close to the exact treewidth. When averaged over all 100 graphs, the solution found by DCBS is within 0.2% of optimality.

Table 1 shows the reduction in peak number of decision nodes stored in a partial decision tree against a full tree. For this experiment, we used a set of 100 random graphs, generated by using $V = 35$ and $E = 140$, which correspond to the most difficult set of random graphs used in [Dow and Korf, 2007]. As can be seen, there is more reduction in the peak number of decision nodes as problem difficulty increases. Further reductions are possible if a finer-grained state-space projection function is used to partition the search frontier nodes into smaller buckets in structured duplicate detection.

In our next experiment, we used $V = 40$ and $E = 120$

	Start node			Neighbor			
Tw	Node	Exp	Sec	Node	Dnode	Exp	Sec
13	430,184	1,494,554	84.9	485,174	354,235	1,444,525	74.9
13	608,958	1,915,962	123.5	602,720	436,021	1,842,721	105.9
13	2,191,918	7,449,878	631.2	2,129,098	789,137	7,306,455	422.0
12	1,947,561	7,625,206	486.7	1,700,106	1,132,437	7,479,034	353.8
13	2,138,996	9,866,471	553.7	2,228,288	1,599,155	9,705,659	441.2
14	33,603,321	124,616,891	26,943.7	27,195,887	4,205,060	123,068,347	10,144.5
14	44,158,361	150,735,512	67,360.6	42,147,733	6,562,130	145,508,098	19,917.0
14	47,553,089	169,672,080	69,420.6	40,197,720	5,243,087	166,965,285	20,800.5
14	44,831,166	176,838,188	32,372.6	34,552,465	6,077,206	175,624,948	12,779.3
15	73,850,566	248,633,008	114,623.7	57,612,526	4,748,778	243,467,972	30,687.2

Table 2: Comparison of different approaches to generating the intermediate graphs. The column labeled "Start node" corresponds to the approach of generating the intermediate graph by eliminating vertices from the original graph; the column labeled "Neighbor" corresponds to the approach of modifying the intermediate graph of a neighboring node. The horizontal line separates the 5 easiest instances from the 5 hardest in a set of 30 random graphs solvable by both approaches.

to generate a set of more difficult random graphs than previously used. The results are shown in Table 2, which compares two different approaches to generating the intermediate graphs, one by eliminating vertices from the original graph, as in [Dow and Korf, 2007], and the other by modifying the intermediate graph of a neighboring node. Both approaches use BFHS as their underlying search algorithm. Note that the ratio by which the second approach improves on the first one increases with the hardness of the instance. For example, the average speedup ratio for the 5 easiest instances (shown above the horizontal line) is 1.3; for the 5 hardest instances (shown below the horizontal line), it is 3.1. The reason our hybrid algorithm performs even better on hard instances is that the more nodes are expanded, the easier it is to find a neighbor whose intermediate graph closely resembles the one to be generated next.

Table 3 shows the performance of our algorithm on benchmark graphs for DIMACS graph coloring instances. Compared to published results [Dow and Korf, 2007; Gogate and Dechter, 2004], Table 3 shows improvement over the state of the art. The running time shown in Table 3 includes the CPU seconds for computing the upper bound using DCBS.

7 Conclusion

We have presented a novel combination of breadth-first and depth-first search that allows a single search algorithm to possess the complementary strengths of both. While our paper focuses on the treewidth problem, many of the ideas have the potential to be applied to other search problems, especially graph-search problems with large encoding sizes, for which memory-reference locality is the key to achieving good performance. Possibilities include model checking [Clarke *et al.*, 2000], where a large data structure that represents the current state is typically stored with each search node. As long as the similarities among different search nodes can be captured in a form that allows depth-first search to exploit the state-representation locality in node expansions, the approach we have described could be effective.

Finally, since our approach to reducing the memory requirements of frontier decision tree uses structured duplicate detection, it can be easily combined with an external-memory graph-search algorithm, for which time rather than memory is likely to be the main bottleneck.

References

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

[Bodlaender *et al.*, 2004] H. Bodlaender, A. Koster, and T. Wolle. Contraction and treewidth lower bounds. In *Proc. of the 12th European Symposium on Algorithms*, pages 628–639, 2004.

[Bodlaender *et al.*, 2006] H. Bodlaender, F. Fromin, A. Koster, D. Kratsch, and D. Thilikos. On exact algortithms for treewidth. In *Proc. of the 14th European Symposium on Algorithms*, pages 672–683, 2006.

[Clarke *et al.*, 2000] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 2000.

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

[Dechter, 1999] R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113(1-2):41–85, 1999.

[Dow and Korf, 2007] P. Alex Dow and R. Korf. Best-first search for treewidth. In *Proceedings of the 22nd National Conference on Artificial Intelligence*, pages 1146–1151, 2007.

[Gogate and Dechter, 2004] V. Gogate and R. Dechter. A complete anytime algorithm for treewidth. In *Proc. of the 20th Conference on Uncertainty in Artificial Intelligence*, pages 201–208, 2004.

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

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

[Zhou and Hansen, 2003] R. Zhou and E. Hansen. Sweep A*: Space-efficient heuristic search in partially ordered graphs. In *Proc. of 15th IEEE International Conf. on Tools with Artificial Intelligence*, pages 427–434, 2003.

[Zhou and Hansen, 2004a] R. Zhou and E. Hansen. Breadth-first heuristic search. In *Proc. of the 14th International Conference on Automated Planning and Scheduling*, pages 92–100, 2004.

[Zhou and Hansen, 2004b] R. Zhou and E. Hansen. Structured duplicate detection in external-memory graph search. In *Proc. of the 19th National Conference on Artificial Intelligence*, pages 683–688, 2004.

[Zhou and Hansen, 2006] R. Zhou and E. Hansen. Breadth-first heuristic search. *Artificial Intelligence*, 170(4-5):385–408, 2006.