Breadth-First Heuristic Search

Rong Zhou and Eric A. Hansen

Department of Computer Science and Engineering
Mississippi State University, Mississippi State, MS 39762
{rzhou,hansen}@cse.msstate.edu

Abstract

Recent work shows that the memory requirements of best-first heuristic search can be reduced substantially by using a divide-and-conquer method of solution reconstruction. We show that memory requirements can be reduced even further by using a breadth-first instead of a best-first search strategy. We describe optimal and approximate breadth-first heuristic search algorithms that use divide-and-conquer solution reconstruction. Computational results show that they outperform other optimal and approximate heuristic search algorithms in solving domain-independent planning problems.

Introduction

The A* graph-search algorithm and its variants are widely used for path planning, robot motion planning, and domain-independent STRIPS planning. But as is well-known, the scalability of A* is limited by its memory requirements. A* stores all explored nodes of a search graph in memory, using an Open list to store nodes on the search frontier and a Closed list to store already-expanded nodes. This serves two purposes. First, it allows the optimal solution path to be reconstructed after completion of the search by tracing pointers backwards from the goal node to the start node. Second, it allows nodes that have been reached along one path to be recognized if they are reached along another path, in order to prevent duplicate search effort. It is necessary to store all explored nodes in order to perform both functions, but not to perform just one. This leads to two different strategies for reducing the memory requirements of heuristic search: one strategy gives up duplicate elimination and the other gives up the traceback method of solution reconstruction.

Linear-space variants of A* such as IDA* (Korf 1985) and RBFS (Korf 1993) give up duplicate elimination. Instead of storing Open and Closed lists, they use a stack to organize the search. Since the current best solution path is stored on the stack, solution reconstruction by the traceback method is straightforward. But because they only store nodes on the current path, they are severely limited in their ability to recognize when newly-generated nodes have been previously explored. Essentially, linear-space search algorithms convert graph-search problems into tree-search problems. This

can lead to an exponential increase in the time complexity of search (measured by the number of node expansions) as the depth of the search increases, and for complex graph-search problems with many duplicate paths, IDA* and RBFS can perform very poorly, due to excessive node re-generations. Their performance can be improved by using available memory to store as many explored nodes as possible in order to check for duplicates (Reinefeld & Marsland 1994; Miura & Ishida 1998), but this requires as much memory as A* to eliminate *all* duplicate search effort.

A second strategy for reducing the memory requirements of search prevents duplicate search effort, but does not use the traceback method of solution reconstruction. It is based on the insight that it is not necessary to store all expanded nodes in order to prevent node re-generation. It is only necessary to store enough to form a *boundary* between the frontier and interior of the search graph. This strategy was introduced to the AI community in a pair of related search algorithms (Korf 1999; Korf & Zhang 2000), which in turn are related to earlier work on reducing the memory requirements of dynamic programming for sequence comparison (Hirschberg 1975). Instead of the traceback method, this strategy uses a divide-and-conquer method of solution reconstruction in which memory is saved by finding a node in the middle of an optimal path, instead of the complete optimal path, and then using the midpoint node to divide the original problem into two sub-problems. Each subproblem is solved recursively by the same algorithm until all nodes on the optimal path are identified.

The contribution of this paper is to show that when using divide-and-conquer solution reconstruction, a breadth-first search strategy is more memory-efficient than a best-first strategy. Although breadth-first search may lead to more node expansions, it reduces the size of the set of boundary nodes that need to be retained in memory. This allows larger problems to be solved. To substantiate this claim, our paper begins with a review of best-first search algorithms that use divide-and-conquer solution reconstruction. Then we introduce a family of breadth-heuristic search algorithms that includes Breadth-First Iterative-Deepening A* for optimal search, and Divide-and-Conquer Beam Search for approximate search. Computational results for the Fifteen Puzzle and for domain-independent STRIPS planning show the advantages of this approach.

Divide-and-conquer solution reconstruction

The divide-and-conquer strategy for solution reconstruction has long been used to reduce the space complexity of dynamic programming for sequence comparison (Hirschberg 1975; Myers & Miller 1988). It was recently introduced to the heuristic search community by Korf (1999). Korf and Zhang (2000) describe a version of A* that uses it. Zhou and Hansen (2003a; 2003b) introduce enhancements.

The strategy is based on recognition that it is not necessary to store all expanded nodes in a Closed list in order to prevent re-generation of already-expanded nodes. It is only necessary to store a subset of nodes that forms a *boundary* between the frontier and interior of the search graph. The concept of a "boundary" expresses the intuition that the set of explored nodes forms a "volume" that encompasses the start node and grows outward, and no unexpanded node outside the boundary can reach an already-expanded node without passing through some node in the boundary, as illustrated by Figure 1. Thus, storing only the boundary nodes is as effective as storing all expanded nodes with respect to preventing node re-generation.

Although nodes inside the boundary can be removed from memory without risking duplicate search effort, this means it is no longer possible to reconstruct a solution by the traditional traceback method. To allow divide-and-conquer solution reconstruction, each node stores information about a node along an optimal path to it that divides the problem in about half. Once the search problem is solved, information about this midpoint node is used to divide the search problem into two sub-problems: the problem of finding an optimal path from the start node to the midpoint node, and the problem of finding an optimal path from the midpoint node to the goal node. Each of these subproblems is solved by the original search algorithm, in order to find a node in the middle of their optimal paths. The process continues recursively until primitive subproblems (in which the goal node is an immediate successor of the start node) are reached, and all nodes on the optimal solution path have been identified. Since the time it takes to solve all subproblems is very short compared to the time it takes to solve the original search problem, this approach saves a great deal of memory in exchange for limited time overhead for solution reconstruction.

Search algorithms that use this strategy to reduce memory requirements differ in detail. In Korf and Zhang's (2000) Divide-and-Conquer Frontier Search (DCFA*), each node past the midpoint of the search stores (via propagation) all state information about a node along an optimal path to this node that is about halfway between the start and goal node. In Sparse-Memory A* (Zhou & Hansen 2003a), each node stores a pointer to its predecessor or to an intermediate node along an optimal path, called a *relay node*, that is retained in memory. Since storing pointers and relay nodes takes less space and allows faster solution reconstruction, we adopt that technique in this paper.

Algorithms also differ in how they distinguish the boundary from closed nodes that can be removed from memory. For example, DCFA* only stores the frontier nodes of the search (the Open List) and not the interior nodes (the Closed list). Already-closed nodes are prevented from being re-
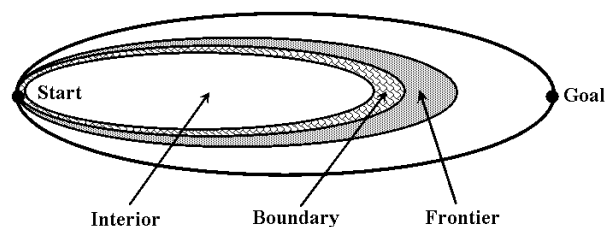

Figure 1: A set of boundary nodes separates the frontier from the interior of the search graph. (All nodes inside the boundary are closed and all nodes outside the boundary are open. The boundary itself may contain closed and/or open nodes.)

generated by storing a list of *forbidden operators* in each node. The list includes one operator (stored as a bit) for each potential successor of the node that has already been generated. Each time a node is expanded, each of its successor nodes is given a forbidden operator that blocks re-generation of the just-expanded node. In undirected graphs, this is sufficient to prevent re-generation of already-closed nodes. In directed graphs in which a node can have predecessors that are not also potential successors, an additional technique must be used. Each time a node is expanded, all of its predecessor nodes are generated as well as its successor nodes. If the search has not yet found a legal path to these predecessor nodes, they are assigned an infinite f-cost to prevent them from being expanded until a legal path is found. Note that these *virtual nodes* acquire an actual, finite cost once a path to them is found.

Zhou and Hansen (2003a) propose a different technique for preventing node re-generation that does not require forbidden operators or virtual nodes. Instead, they associate with each node a counter that is initially set to the number of potential predecessors of a node. Each time a node is expanded, the counter of each of its successors is decremented. Closed nodes can be removed from memory when their counter is equal to zero.

For the special case of multiple sequence alignment, Zhou and Hansen (2003b) propose an even simpler technique. Because the search graph of the multiple sequence alignment problem is a lattice, it can be decomposed into a sequence of layers such that each node in a layer can only have successors in the current layer or the next layer, but not in any previous layer. If all nodes in one layer are expanded before the next layer is considered, then all previously-expanded layers can be removed from memory without risking node re-generation. Because this technique does not require forbidden operators, virtual nodes or predecessor counters, it is much easier to implement. It also uses much less memory than DCFA* or Sparse-Memory A*.

The breadth-first heuristic search algorithms we introduce in this paper can be viewed as generalizations of this algorithm, in that they also expand the search graph on a layer-by-layer basis instead of in best-first order. However, layers are defined differently, and the algorithms we introduce are more general in the sense that they can search graphs with arbitrary structure, and not simply graphs that share the lattice structure of the multiple sequence alignment problem.

Breadth-First Heuristic Search

In the rest of this paper, we describe a family of heuristic search algorithms that use divide-and-conquer solution reconstruction in combination with a breadth-first strategy of node expansion. The only assumption we make about the graphs searched by these algorithms is that all edges have unit cost; in other words, we assume planning problems have unit-cost actions. This assumption allows a breadth-first search algorithm to guarantee that when a node is first generated, an optimal path to it has been found. Our approach can be extended to graphs that have varying and real-valued edge costs. But this extension requires some modifications of the algorithm, and we postpone discussion of it to a later paper.

A breadth-first search graph divides into layers, one for each depth. Since actions have unit cost, all nodes in the same layer have the same g-cost, which is identical to their depth in the graph. Although nodes are expanded in breadth-first order, we use a lower-bound heuristic function to limit exploration of the search space. As in A*, a lower-bound estimate of the cost of an optimal path through node n is given by a node evaluation function $f(g) = g(n) + h(n)$, where h is an admissible heuristic. No node is inserted into the Open list if its f-cost is greater than an upper bound on the cost of an optimal solution, since such nodes cannot be on an optimal path. We discuss how to obtain an upper bound later in this section.

So far, the algorithm we have described is essentially breadth-first branch-and-bound search. This search strategy is rarely used in practice because the number of nodes it expands is at least as great, and usually greater, than the number of nodes expanded by A*, which can be viewed as best-first branch-and-bound search. If all expanded nodes are stored in memory, breadth-first branch-and-bound uses as much or more memory than A*, and has no advantage.

But we propose a breadth-first branch-and-bound algorithm that uses divide-and-conquer solution reconstruction. Its memory requirements depend on the number of nodes needed to maintain a boundary between the frontier and interior of the search, and not the total number of nodes expanded. The central result of our paper is that when divide-and-conquer solution reconstruction is used, a breadth-first branch-and-bound search algorithm can be much more memory-efficient than a best-first algorithm such as DCFA* or Sparse-memory A*.

Figure 2 conveys an intuition of how breadth-first search results in a smaller set of boundary nodes. It shows that best-first node expansion "stretches out" the boundary, whereas breadth-first search does not and uses the upper bound to limit the width of the boundary. Although breadth-first search expands more nodes than best-first search, the memory it saves by storing a smaller boundary results in more efficient search, as our test results will show.

Before discussing details of the algorithm, we consider the question: how many layers of the breadth-first search graph need to be stored in memory to prevent duplicate search effort?
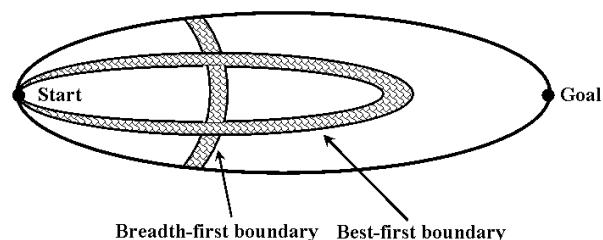

Figure 2: Comparison of best-first and breadth-first boundaries. The outer ellipse encloses all nodes with f-cost less than or equal to an (optimal) upper bound.

Duplicate elimination

When nodes are expanded in breadth-first order, the Open and Closed lists of the algorithm can be considered to have a layered structure, where $Open_\ell$ denotes the set of open nodes in layer ℓ and $Closed_\ell$ denotes the set of closed nodes in layer ℓ. As a result, we sometimes refer to the Open or Closed list of a particular layer, as if each layer has its own Open and Closed lists. Note that at any time, all open nodes are in the current layer or in the next layer, whereas closed nodes can be in the current layer or any previous layer.

Each time the algorithm expands a node in the current layer, it checks whether each successor node is a duplicate of a node that is in the Open list of the next layer, or whether it is a duplicate of a node in the Open or Closed list of the current layer. In addition, it checks whether it is a duplicate of a node that is in the Closed list of one or more previous layers. This raises the following crucial question: how many previous layers must be retained in memory and checked for duplicates to prevent re-generation of already-closed nodes? The answer determines when a closed layer of the search graph can be removed from memory. As it turns out, the answer depends on the structure of the graph.

Definition 1 *The* locality *of a breadth-first search graph is defined as*

$$\max_{n,n' \in N \ s.t. \ n \in pred(n')} \{ g^*(n) - g^*(n'), \ 0 \},$$

where N is the set of nodes, $g^*(n)$ is the length of a shortest path to node n (or equivalently, it is the layer in which node n first appears), and $pred(n)$ is the set of predecessors of n.

Note that $g^*(n)$ can never be less than $g^*(n')$ by more than one. But in general, there is no a priori limit on how much greater $g^*(n)$ can be than $g^*(n')$. In other words, the shortest path to a node n may be arbitrarily longer than the shortest path to its successor node n'. The locality of a graph determines the "thickness" of the set of boundary nodes needed to completely prevent duplicate nodes.

Theorem 1 *The number of previous layers of a breadth-first search graph that need to be retained to prevent duplicate search effort is equal to the locality of the search graph.*

Proof: First assume that the number of previous layers saved in memory is less than the locality k of the graph. To see how this can result in re-generation of a node, consider nodes n

and n' such that $g^*(n) - g^*(n') > k$. When node n is expanded, its successor n' is either in the previous k layers or not. If it is not, it is re-generated. If it is, it has been previously re-generated since it was first generated more than k layers before. In either case, there is a duplicate node.

Now assume the number of stored previous layers of a breadth-first search graph is equal to or greater than the locality of the graph. We prove by induction that this prevents node re-generation. The base step is obvious since for the first k layers of the graph, all previous layers are stored and re-generation of a duplicate node is impossible. For the inductive step, we assume that no duplicates are generated for the first m layers. When layer $m + 1$ is generated, no previously deleted node can be re-generated since the locality of the graph is less than or equal to the number of previous layers stored in memory. $\square$

In general, it is not easy to determine the locality of graph. But in the special case of undirected graphs, the locality is one and we have the following important result.

Corollary 1 *In undirected graphs, use of the immediate previous layer to check for duplicates is sufficient to prevent re-generation of closed nodes.*

Proof: This follows from the fact that the locality of any undirected graph is one. In undirected graphs, the set of predecessors of a node coincides with the set of successors. Therefore, the optimal g-cost of a predecessor is at most one greater than the optimal g-cost of a successor. $\square$

In graphs with a locality of one, such as undirected graphs, the number of layers the algorithm must keep in memory for the purpose of duplicate elimination is three; the previous layer, the currently-expanding layer, and the next layer. In general, the number of layers that need to be retained in memory to prevent duplicate search effort is equal to the locality of the graph plus two. (One additional "relay" layer is needed to allow divide-and-conquer solution reconstruction, as described later.)

Korf and Zhang (2000) use forbidden operators to prevent re-generation of closed nodes. It is easy to see that use of forbidden operators (without virtual nodes) has the same effect as storing one previous layer of the breadth-first search graph, since blocking a node from re-generating a predecessor has the same effect as storing the previous layer and checking for duplicates. But in graphs with locality greater than one, forbidden operators alone are not sufficient to prevent duplicate search effort. In this case, breadth-first search provides a simple alternative: store more than one previous layer of the search graph.

We conjecture that for many directed graphs, it is sufficient to store one previous layer to prevent re-generation of most, if not all, duplicate nodes. Even if the number of stored layers is less than the locality of the graph, an important result is that in the worst case, the number of times a node can be re-generated is at most linear in the depth of the search. This is in sharp contrast to the potentially exponential number of node re-generations for linear-space search algorithms that rely on depth-first search.

Theorem 2 *In breadth-first heuristic search, the worst-case number of times a node n can be re-generated is bounded by*

$$\left\lfloor \frac{f^* - g^*(n)}{\textit{number of saved layers}} \right\rfloor.$$

Proof: Let $\Delta \geq 2$ be the total number of layers saved by the algorithm. Obviously, no duplicate nodes can exist in these Δ layers, because the algorithm always checks for duplicates in all saved layers before inserting any newly-generated node into the Open list for the next layer. Therefore, the earliest time for a node n to be re-generated is $g^*(n) + \Delta$ and the earliest time for the same node to be re-generated twice is $g^*(n) + 2\Delta$, and so on. Since the total number of layers is bounded by the length of the shortest solution path (f^*), the number of times a node n is re-generated cannot exceed the bound stated in the theorem. $\square$

Use of bounds to prune the search graph further reduces the chance of re-generating already closed nodes. Because nodes are expanded in breadth-first order, it is impossible to improve on the g-cost of a node after it is generated. It follows that any node with an f-cost equal to the upper bound will not be re-generated, since it will have a greater g-cost in a subsequent layer, and thus an f-cost greater than the upper bound, causing it to be pruned. From this and the fact that the breadth-first algorithm stores one or more previous layers of the search graph, we have the following optimization that can further improve space efficiency.

Theorem 3 *In breadth-first heuristic search, any node in the k-th previous layer whose f-cost is greater than or equal to the upper bound minus k cannot be re-generated and can be removed from memory.*

If only one previous layer of the search graph is stored, this means that any node in the immediate previous layer whose f-cost is one less than the upper bound can be removed from memory. (This optimization is not included in our pseudocode, although it is included in our implementation.)

Algorithm

Figure 3 gives the pseudocode of the basic breadth-first heuristic search algorithm. The main algorithm, *BFHS*, is the same as A* except that the Open and Closed lists are indexed by layers, previous layers are deleted to recover memory, and the solution is reconstructed by the divide-and-conquer method once the goal node is selected from the Open list. The procedure *ExpandNode* works in the usual way except that it uses an upper bound U to prune nodes that cannot be on an optimal path. (Note that for subproblems solved during divide-and-conquer solution reconstruction, the upper bound is optimal since the optimal cost of the overall solution is determined before beginning solution reconstruction.)

To allow divide-and-conquer solution reconstruction, each node must store information about an intermediate node along an optimal solution path. We use the method described by Zhou and Hansen (2003a), in which each node stores a pointer to either its predecessor node or to an ancestor node (called a relay node) along an optimal path. For simplicity, the pseudocode assumes that a single intermediate layer of the breadth-first search graph (called a *relay*

Procedure *DeletePreviousLayer* (Integer ℓ, $relay$; Node $start$)
1   **if** $\ell \leq relay$ **then**
2   **for each** $n \in Closed_\ell$ **do**
3           $ancestor(n) \leftarrow start$
4   **else**
5   **for each** $n \in Closed_\ell$ **do**
6           $\alpha \leftarrow ancestor(n)$
7   **while** $\alpha \notin Closed_{relay}$ **do**   /* find relay node for n */
8               $\alpha \leftarrow ancestor(\alpha)$
9           $ancestor(n) \leftarrow \alpha$
10 **for each** $n \in Closed_{\ell-1}$ **do**   /* delete previous layer */
11      $Closed_{\ell-1} \leftarrow Closed_{\ell-1} \setminus \{n\}$
12   **delete** n

Procedure *ExpandNode* (Node n; Integer ℓ, U)
13 **for each** $n' \in Successors(n)$ **do**
14   **if** $g(n) + 1 + h(n') > U$ **continue**   /* prune */
15   **if** $n' \in Closed_{\ell-1} \cup Closed_\ell$ **continue**   /* duplicate */
16   **if** $n' \in Open_\ell \cup Open_{\ell+1}$ **continue**   /* duplicate */
17      $g(n') \leftarrow g(n) + 1$
18      $ancestor(n') \leftarrow n$
19      $Open_{\ell+1} \leftarrow Open_{\ell+1} \cup \{n'\}$

Algorithm *BFHS* (Node $start$, $goal$; Integer U)
20 $g(start) \leftarrow 0$
21 $ancestor(start) \leftarrow$ **nil**
22 $Open_0 \leftarrow \{start\}$, $Open_1 \leftarrow \emptyset$
23 $Closed_0 \leftarrow \emptyset$
24 $\ell \leftarrow 0$ /* ℓ = layer */
25 $relay \leftarrow \lfloor U/2 \rfloor$ /* $relay$ = relay layer */
26 **while** $Open_\ell \neq \emptyset$ **or** $Open_{\ell+1} \neq \emptyset$ **do**
27   **while** $Open_\ell \neq \emptyset$ **do**
28          $n \leftarrow \arg\min_n\{g(n) \mid n \in Open_\ell\}$
29          $Open_\ell \leftarrow Open_\ell \setminus \{n\}$
30          $Closed_\ell \leftarrow Closed_\ell \cup \{n\}$
31   **if** n is $goal$ **then**   /* solution reconstruction */
32              $middle \leftarrow ancestor(n)$
33   **if** $g(middle) = 1$ **then**   /* recursion ends */
34                  $\pi_0 \leftarrow \langle start, middle \rangle$
35   **else**
36                  $\pi_0 \leftarrow BFHS(start, middle, g(middle))$
37   **if** $g(n) - g(middle) = 1$ **then** /* recursion ends */
38                  $\pi_1 \leftarrow \langle middle, n \rangle$
39   **else**
40                  $\pi_1 \leftarrow BFHS(middle, n, g(n) - g(middle))$
41   **return** $Concatenate\,(\pi_0, \pi_1)$
42          $ExpandNode\,(n, \ell, U)$ /* U = upper bound */
43   **if** $1 < \ell \leq relay$ **or** $\ell > relay + 1$ **then**
44          $DeletePreviousLayer\,(\ell, relay, start)$
45      $\ell \leftarrow \ell + 1$ /* move on to next layer */
46      $Open_{\ell+1} \leftarrow \emptyset$
47      $Closed_\ell \leftarrow \emptyset$
48 **return** $\emptyset$

Figure 3: Pseudocode for *BFHS* (Breadth-First Heuristic Search).

layer) is preserved in memory for use in divide-and-conquer solution reconstruction. It also assumes the relay layer is approximately in the middle of the search graph, since equal-sized sub-problems are easier to solve. When the procedure *DeletePreviousLayer* is invoked to delete a previously-expanded layer of the breadth-first search graph that is no longer needed for duplicate detection, it adjusts the ancestor pointers in its successor layer so that they point to nodes in the relay layer, if the deleted layer comes after the relay layer (lines# 4-9), or else to the start node, if the deleted layer comes before the relay layer (lines# 1-3).

The divide-and-conquer method can be implemented in a more sophisticated and efficient way than presented in the pseudocode. Since the middle layer is typically the largest layer of the graph (see Figure 4), we have found that in practice, it is more memory-efficient to save the layer that is at the 3/4 point in the graph, for example. This reduces the peak memory requirements of the algorithm, and in practice, increases the time overhead of solution reconstruction by an almost negligible amount.

The time efficiency of the algorithm can be improved significantly by not using the divide-and-conquer method when there is enough memory to solve a problem, or one of the recursive subproblems, by keeping all the layers of the search graph in memory and using the traceback method to recover the solution path. After one level of divide-and-conquer recursion, for example, there is often enough memory to solve the resulting subproblems without deleting any layers, and without needing to continue the divide-and-conquer recursion. In an efficient implementation of the algorithm, a lazy approach to deleting previous layers of the search graph is adopted, in which previous layers are deleted only when memory is close to full.

Breadth-First Iterative-Deepening A*

Our breadth-first algorithm uses an upper bound on the cost of an optimal solution to prune the search space, and the quality of the upper bound has a significant effect on the efficiency of the algorithm. The better the upper bound, the fewer nodes are expanded and stored. (In fact, given an optimal upper bound, the algorithm does not expand any more nodes than A*, disregarding ties.)

An upper bound can be obtained by finding an approximate solution to the search problem. There are many possible ways to quickly compute an approximate solution in order to obtain an upper bound. An obvious method is to use weighted A* search. Below, we describe a beam search algorithm that is also very effective.

Here, we point out that it is possible to define a version of breadth-first heuristic search that does not need a previously-computed upper bound. Instead, it uses an iterative-deepening strategy to avoid expanding nodes that have an f-cost greater than a hypothetical upper bound. The algorithm first runs breadth-first heuristic search using the f-cost of the start node as an upper bound. If no solution is found, it increases the upper bound by one (or to the least f-cost of any unexpanded nodes) and repeats the search. Because of the similarity of this algorithm to Depth-First Iterative-Deepening A* (Korf 1985), we call it *Breadth-First Iterative-Deepening A** (BFIDA*). The amount of memory it uses is the same as the amount of memory BFHS would use given an optimal upper bound. However, BFIDA* may run more slowly than BFHS with a previously-computed upper bound, because running multiple iterations of BFHS takes extra time.
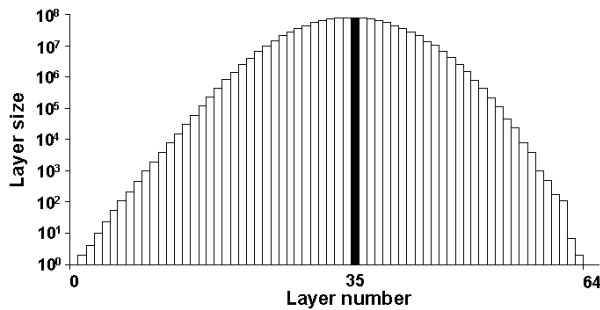
Figure 4: Size of layer (logarithmically scaled) as function of search depth for Korf's most difficult 15-puzzle instance (No. 88).

#	Len	Stored	A* Exp	BFIDA* Exp
17	66	16,584,444	218,977,081	279,167,411
49	59	21,177,925	243,790,912	345,700,085
53	64	12,753,096	177,244,033	224,545,853
56	55	13,066,308	141,157,391	208,900,977
59	57	13,974,753	158,913,130	228,900,723
60	66	56,422,199	767,584,679	978,804,885
66	61	21,435,302	275,076,045	368,138,264
82	62	46,132,337	549,557,759	765,608,989
88	65	77,547,650	999,442,569	1,360,582,446
92	57	12,591,419	151,699,572	213,871,768

Table 1: Performance of BFIDA* on the 10 most difficult instances of Korf's 100 random instances of the 15-puzzle. Columns show the instance number (#); solution length (Len); peak number of nodes stored (Stored); number of node expansions in the last iteration, which is equal to the number of nodes that A* must expand (A* Exp); and the total number of node expansions (BFIDA* Exp).

Divide-and-Conquer Beam Search

Breadth-first heuristic search can significantly reduce the memory requirements of search, while still eliminating all duplicate search effort. But it can still run out of memory if the number of nodes in any layer becomes too large. Typically, the largest layer is in the middle of the search graph, since layers close to the start node are relatively small due to reachability constraints, and layers close to the goal node are relatively small because of the strength of the heuristic close to the goal, which makes it possible to prune most nodes. Figure 4 illustrates this and shows how the size of each layer varies with the depth of the search for a difficult instance of the 15-puzzle.

If the largest layer of the breadth-first search graph does not fit in memory, one way to handle this follows from recognition that breadth-first heuristic search is very closely-related to beam search. Instead of considering all nodes in a layer, we propose a beam-search variant of breadth-first heuristic search that considers the most promising nodes until memory is full (or reaches a predetermined bound). At that point, the algorithm recovers memory by pruning the least-promising nodes (i.e., the nodes with the highest f-cost) from the Open list. Then it continues the search.

Aside from pruning the least-promising open nodes when memory is full, the algorithm is identical to breadth-first heuristic search. The difference from traditional beam search is that it uses divide-and-conquer solution reconstruction to reduce memory requirements. But this is an important difference since it allows it to use a much larger beam width in order to improve performance. We call the resulting algorithm Divide-and-Conquer Beam Search, and report impressive empirical results later in the paper.

Computational results

We first consider the performance of Breadth-First Heuristic Search on the Fifteen-puzzle, since this makes for easier comparison to the best-first alternatives of DCFA* and Sparse-Memory A*. Then we consider its performance on a range of difficult STRIPS planning problems from the Planning Competition.

Fifteen-puzzle

We tested BFIDA* on the same 100 instances of the 15-puzzle used as a test set by Korf (1985). For the 15-puzzle, our implementation of BFIDA* uses forbidden operators to block re-generation of nodes in the previous layer, rather than storing the previous layer, since this saves space and is easily implemented for this problem. Table 1 shows results for the ten most difficult instances. BFIDA* solves all 100 instances using no more than 1.3 gigabytes of memory. Given 4 gigabytes of memory, neither DCFA* nor Sparse-memory A* can solve more than 96 instances; the instances they cannot solve are numbers 17, 60, 82, and 88.[1] For the 96 solvable instances, DCFA* stores 5 times more nodes than BFIDA*, and Sparse-Memory A* stores 3.4 times more nodes. This clearly shows the advantage of breadth-first over best-first divide-and-conquer heuristic search.

Based on the number of nodes that need to be expanded to solve these 15-puzzle instances, A* would need between 12 and 16 times more memory than BFIDA* just to store the Closed list. Although BFIDA* must re-expand some nodes due to iterative deepening and divide-and-conquer solution reconstruction, the last column of Table 1 shows that it only expands from 30% to 40% (on average) more nodes than A* would expand in solving these 15-puzzle instances. IDA* expands many more nodes than this, but still runs much faster in solving the 15-puzzle due to lower node-generation overhead and the fact that the number of duplicate paths does not grow too fast with the depth of the search, for this problem. (For the planning problems considered next, IDA* loses its advantage.)

Domain-independent STRIPS planning

Over the past several years, the effectiveness of heuristic search for domain-independent STRIPS planning has be-

---

[1]DCFA* and Sparse-memory A* can solve instance 49 but not instance 17, even though BFIDA* requires less memory to solve 17 than 49. The explanation is that the best-first boundary for instance 17 has more nodes than the best-first boundary for instance 49, although the breadth-first boundary for instance 17 is smaller than for instance 49.

		A*			BFHS		
Instance	Len	Stored	Exp	Secs	Stored	Exp	Secs
blocks-14	38	735,905	252,161	12.5	228,020	863,495	37.9
gripper-6	41	2,459,599	2,436,847	35.1	1,529,307	11,216,130	157.1
satellite-6	20	3,269,703	2,423,288	177.6	1,953,396	3,750,119	257.4
elevator-11	37	3,893,277	3,884,960	181.1	1,144,370	8,678,466	433.9
depots-3	27	> 6,100,806	> 3,389,343	> 112.2	4,841,706	8,683,716	270.3
driverlog-10	17	> 7,626,008	> 1,484,326	> 95.1	6,161,424	10,846,888	560.7
freecell-4	27	> 6,990,507	> 3,674,734	> 432.9	5,891,140	17,140,644	1,751.0

Table 2: Comparison of A* and BFHS (using an upper bound found by beam search) on STRIPS planning problems. Columns show optimal solution length (Len); peak number of nodes stored (Stored); number of node expansions (Exp); and running time in CPU seconds (Secs). The > symbol indicates that A* ran out of memory before solving the problem.

		IDA*			BFIDA*		
Instance	Len	Stored	Exp	Secs	Stored	Exp	Secs
blocks-12	34	5,015	180,305	50.3	6,354	34,687	1.6
blocks-14	38	94,011	51,577,732	20,631.0	224,058	1,324,320	44.4
logistics-4	20	2,289	45,194,644	621.7	1,730	16,077	0.5
depots-2	15	2,073	227,289	31.1	1,923	8,139	0.5
gripper-2	17	1,769	16,381,009	312.7	1,398	17,281	0.3
gripper-6	41	-	-	-	1,848,788	85,354,245	1,029.3
satellite-4	17	-	-	-	70,298	303,608	8.2

Table 3: Comparison of IDA* (using transposition table) and BFIDA* on STRIPS planning problems. Columns show optimal solution length (Len); peak number of nodes stored (Stored); number of node expansions (Exp); and running time in CPU seconds (Secs). IDA* could not solve the last two problems after 12 hours of CPU time.

come widely-recognized. A* and IDA* are used to find optimal plans, given an admissible heuristic (Haslum & Geffner 2000). Weighted A* is used to find approximate plans for difficult planning problems, guided by an informative though usually non-admissible heuristic (Bonet & Geffner 2001b). Heuristic-search planners have performed very well in the biennial planning competitions, and the problems used in the competition provide a good test set for comparing graph-search algorithms since they give rise to a variety of search graphs with different kinds of structure, and memory is a limiting factor in solving many of the problems.

For domain-independent STRIPS planning, breadth-first search has an important advantage over best-first search when divide-and-conquer solution reconstruction is used: it is *much* easier to implement. Implementing DCFA* or Sparse-memory A* for STRIPS planning problems presents several difficulties. Implementing forbidden operators in a domain-independent way can increase node size substantially, since every possible operator instantiation must be considered. When STRIPS operators are only conditionally reversible, forbidden operators are also difficult to implement because it is impossible to determine reversibility in advance. For STRIPS planning problems that correspond to directed graphs, implementing virtual nodes (as in DCFA*) or even predecessor counting (as in Sparse-memory A*) is challenging given the difficulty of identifying all predecessors of a node, especially since the number of potential predecessors is exponential in the size of the Add list of an operator, and all operators must be considered. Since many potential predecessors may not even be reachable in the search graph, the boundary could become cluttered with nodes that will never be removed.

By contrast, implementation of a breadth-first algorithm that uses divide-and-conquer solution reconstruction is straightforward. Because of the layered structure of the search graph, there is no need for forbidden operators, virtual nodes or predecessor counters. The algorithm simply stores one or more previous layers and checks for duplicates. Given the difficulty of implementing DCFA* or Sparse-memory A* for domain-independent planning, we compare our breadth-first divide-and-conquer algorithms to A*, IDA*, and weighted A*.

The following experiments were performed on a Pentium IV 2.4 GHz processer with 512 megabytes of RAM. We used the HSPr planning system of Blai Bonet as a foundation for implementing our algorithms (Bonet & Geffner 2001a). Although divide-and-conquer search algorithms implemented using relay nodes can use extra memory to reduce the overhead of solution reconstruction, we ran our algorithms to minimize memory use. We also stored only a single previous layer in checking for duplicates. Although two of the planning domains among our eight test domains correspond to directed graphs for which graph locality is not obvious (*freecell* and *satellite*), we found empirically that storing a single previous layer was sufficient to eliminate all duplicates in both cases. The other six planning problems correspond to undirected graphs.

BFHS vs. A* Table 2 shows representative results from a comparison of A* to Breadth-First Heuristic Search (using an upper bound computed by Divide-and-Conquer Beam

| | Weighted A* (W = 2.0) | | | | | Divide-and-Conquer Beam Search | | | | |
Instance	Len	LB	Stored	Exp	Secs	Len	LB	Stored	Exp	Secs
blocks-10	60	24	59,250	47,017	1.0	36	34	20,000	144,655	4.2
blocks-20	-	36	> 3,195,661	> 1,710,952	> 149.0	60	48	50,000	1,058,716	103.0
depots-4	-	14	> 5,162,221	> 2,519,509	> 151.7	30	17	120,000	547,622	33.1
depots-5	-	14	> 4,793,491	> 1,571,290	> 114.4	45	20	100,000	1,248,798	91.8
depots-8	-	10	> 4,473,925	> 812,213	> 90.0	34	12	50,000	239,361	30.3
depots-10	27	10	286,657	30,155	3.2	25	13	50,000	147,951	14.4
depots-11	-	14	> 3,728,271	> 823,102	> 148.2	49	17	500,000	5,522,175	1,073.5
depots-14	-	11	> 3,532,046	> 200,226	> 68.9	29	13	800,000	1,706,529	383.9
depots-16	29	9	2,041,919	190,157	29.4	26	12	50,000	133,046	17.0
driverlog-12	52	12	27,740	8,158	0.7	38	16	5,000	47,944	3.9
driverlog-15	64	11	1,890,205	235,462	41.6	36	13	50,000	295,044	48.3
freecell-2	17	11	734,703	1,878,993	102.2	17	12	5,000	11,460	1.0
freecell-3	27	13	282,548	486,796	25.3	21	14	10,000	33,944	2.6
freecell-4	-	17	> 5,592,406	> 10,815,416	> 878.1	27	18	10,000	46,116	6.4

Table 4: Comparison of weighted A* (with weight of 2.0) and Divide-and-Conquer Beam Search on STRIPS planning problems. Columns show solution length (Len); provable lower bound (LB); peak number of nodes stored (Stored); number of node expansions (Exp); and running time in CPU seconds (Secs). The > symbol indicates that weighted A* ran out of memory before solving the problem.

Search).[2] Although BFHS expands more nodes than A*, it uses less memory and thus can solve larger problems. The memory savings for these planning problems is not as great as for the 15-puzzle because the best available admissible heuristic for domain-independent planning, the *max-pair* heuristic of Haslum and Geffner (2000), is very weak. In general, the more informative a heuristic, the greater the advantage of a breadth-first divide-and-conquer strategy, for two reasons. First, the more informative a heuristic, the more it "stretches out" a best-first boundary, whereas a weak heuristic results in a boundary that is more similar to the boundary of breadth-first search. Second, a more informed heuristic "narrows" the breadth-first boundary because it prunes more nodes.

BFIDA* vs. IDA* Table 3 shows representative results from a comparison of Haslum's implementation of IDA* using a transposition table, as described in his paper (Haslum & Geffner 2000), with Breadth-First Iterative-Deepening A*. IDA* performs much worse than A* due to excessive node re-generations. This clearly illustrates that the problem of duplicate paths is much more severe for the Planning Competition problems than for the 15-puzzle, and effective duplicate elimination is essential for good performance.

Divide-and-Conquer Beam Search vs. weighted A* Table 4 shows representative results from a comparison of weighted A* (using a weight of 2.0) to Divide-and-Conquer Beam Search. Since we are willing to accept approximate solutions in order to solve larger problems, we use the more informative, although non-admissible, *additive heuristic* to guide the search (Bonet & Geffner 2001b). However, we also use the admissible *max-pair* heuristic to evaluate each node, since this lets us compute a lower bound on the optimal solution length; the lower bound is the least admissible

f-cost of any unexpanded node. Since the solution found is an upper bound on the optimal solution length, the lower bound lets us bound the sub-optimality of the solution.

It is interesting that the beam search algorithm consistently finds better lower bounds than weighted A*, even when it expands far fewer nodes. This is due to the fact that it distributes search effort more evenly among layers, and fully expands early layers of the search graph, whereas weighted A* tends to focus search effort near the goal.

In the results for the beam search algorithm, the peak number of stored nodes corresponds to a predetermined bound. We adjusted the bound for different problems based on the difficulty of the problem, in order to adjust a time-quality tradeoff. The peak number of stored nodes is roughly equal to the size of a layer times four, since Divide-and-Conquer Beam Search only stores four layers in memory; the current, previous, and next layer, and the relay layer used for solution reconstruction. Because a divide-and-conquer implementation of beam search is more memory-efficient than ordinary beam search, it allows larger beam widths that lead to better solutions. For some of the test problems (e.g., *depot-11* and *depot-14*), traditional beam search (without divide-and-conquer solution reconstruction) would run out of memory if it used the same beam width.

An interesting side-effect of the divide-and-conquer strategy, when used with beam search, is that the length of the initial solution found by beam search can be improved on during solution reconstruction, since better solutions to sub-problems can be found in the divide-and-conquer phase. This occurred for four of the examples in Table 4, and resulted in a reduction of between 2% and 6% in the length of the initial solution found by beam search.

Although the beam search algorithm uses much less memory than weighted A*, it solves larger problems and finds much higher-quality solutions. Given the extensive use of weighted A* in the Planning Competition, the fact that Divide-and-Conquer Beam Search performs so much better is very encouraging and suggests that this is a promising direction for continued research.

Conclusion

Best-first search is traditionally considered more efficient than breadth-first search because it minimizes the number of node expansions. The contribution of this paper is to show that when divide-and-conquer solution reconstruction is used to reduce the memory requirements of search, breadth-first search becomes more efficient than best-first search because it needs less memory to prevent regeneration of closed nodes.

We have described a family of algorithms that use a breadth-first heuristic search strategy, including algorithms that find optimal and approximate solutions. We have shown that this strategy outperforms A*, IDA*, and weighted A* in solving STRIPS planning problems, and that it outperforms DCFA* and Sparse-Memory A* on the 15-puzzle. Because a breadth-first strategy is simpler to implement, it is also practical for a wider class of problems than the best-first strategy of DCFA* and Sparse-memory A*.

In adopting breadth-first search, we made the limiting assumption that all actions have unit cost. In future work, we will show how to relax this assumption and extend this approach to planning problems in which actions have varying or real-valued costs, including metric planning problems.

References

Bonet, B., and Geffner, H. 2001a. Heuristic search planner 2.0. *AI Magazine* 22(3):77–80.

Bonet, B., and Geffner, H. 2001b. Planning as heuristic search. *Artificial Intelligence* 129(1):5–33.

Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. In *Proc. of the 5th International Conference on AI Planning and Scheduling*, 140–149.

Hirschberg, D. S. 1975. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM* 18(6):341–343.

Korf, R. 1985. Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence* 27:97–109.

Korf, R. 1993. Linear-space best-first search. *Artificial Intelligence* 62:41–78.

Korf, R. 1999. Divide-and-conquer bidirectional search: First results. In *Proc. of the 16th International Joint Conference on Artifiicial Intelligence (IJCAI-99)*, 1184–1189.

Korf, R., and Zhang, W. 2000. Divide-and-conquer frontier search applied to optimal sequence alignment. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-00)*, 910–916.

Miura, T., and Ishida, T. 1998. Stochastic node caching for memory-bounded search. In *Proc. of the 15th National Conference on Artificial Intelligence (AAAI-98)*, 450–456.

Myers, E., and Miller, W. 1988. Optimal alignments in linear space. *Computer Appl. in the Biosciences* 4:11–17.

Reinefeld, A., and Marsland, T. 1994. Enhanced iterative-deepening search. *IEEE Trans. on Pattern Analysis and Machine Intelligence* 16:701–710.

Zhou, R., and Hansen, E. 2003a. Sparse-memory graph search. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, 1259–1266.

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