Frontier Search

RICHARD E. KORF

University of California, Los Angeles, California

WEIXIONG ZHANG

Washington University, St. Louis, Missouri

AND

IGNACIO THAYER AND HEATH HOHWALD

University of California, Los Angeles, California

Abstract. The critical resource that limits the application of best-first search is memory. We present a new class of best-first search algorithms that reduce the space complexity. The key idea is to store only the Open list of generated nodes, but not the Closed list of expanded nodes. The solution path can be recovered by a divide-and-conquer technique, either as a bidirectional or unidirectional search. For many problems, frontier search dramatically reduces the memory required by best-first search. We apply frontier search to breadth-first search of sliding-tile puzzles and the 4-peg Towers of Hanoi problem, Dijkstra's algorithm on a grid with random edge costs, and the A* algorithm on the Fifteen Puzzle, the four-peg Towers of Hanoi Problem, and optimal sequence alignment in computational biology.

Categories and Subject Descriptors: I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Graph and tree search strategies*

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: A* algorithm, best-first search, bidirectional search, breadth-first search, Dijkstra's algorithm, heuristic search, sequence alignment, sliding-tile puzzles, Towers of Hanoi

---

This research was supported by the National Science Foundation (NSF) under grant No. EIA-0113313, by NASA and JPL under contract no. 1229784, and by the State of California MICRO grant no. 01-044 to R. Korf, and by NSF grant EIA-0113618 to W. Zhang.

Authors' addresses: R. E. Korf, Computer Science Department, University of California, Los Angeles, CA 90095, e-mail: korf@cs.ucla.edu; W. Zhang, Department of Computer Science, Washington University, Campus Box 1045, One Brookings Drive, St. Louis, MS 63130-4899, e-mail: zhang@cse.wustl.edu; I. Thayer, Google, Inc., 1600 Amphitheatre Parkway, Mountain View, CA 94043, e-mail: ignacio@google.com; H. Hohwald, 3 Fams Court, Jericho, NY 11753, e-mail: heathhohwald@gmail.com.

1. *Introduction*

Systematic graph-search algorithms can be classified into two general categories, best-first and depth-first searches. Best-first searches include breadth-first search (BFS), Dijkstra's single-source shortest-path algorithm (Dijkstra) [Dijkstra 1959], and A* [Hart et al. 1968]. Depth-first searches include depth-first search (DFS), depth-first iterative-deepening (DFID) [Korf 1985], iterative-deepening-A* (IDA*) [Korf 1985], and depth-first branch-and-bound (DFBnB).

1.1. BEST-FIRST SEARCH.

A best-first search algorithm maintains two lists of nodes, an Open list and a Closed list. The Closed list contains those nodes that have been expanded, by generating all their children, and the Open list contains those nodes that have been generated, but not yet expanded. At each cycle of the algorithm, an Open node of lowest cost is expanded, moved to the Closed list, and its children are added to the Open list.

Individual best-first search algorithms differ primarily in the cost function $f(n)$. If $f(n)$ is the depth of node n, best-first search becomes breadth-first search. If $f(n) = g(n)$, where $g(n)$ is the cost of the current path from the start state to node n, then best-first search becomes Dijkstra's single-source shortest-path algorithm [Dijkstra 1959]. If $f(n) = g(n) + h(n)$, where $h(n)$ is a heuristic estimate of the cost of reaching a goal from node n, then best-first search becomes the A* algorithm [Hart et al. 1968]. Breadth-first search can terminate as soon as a goal node is generated, while Dijkstra's algorithm and A* must wait until a goal node is chosen for expansion to guarantee optimality.

Every node generated by a best-first search is stored in either the Open or Closed lists, for two reasons. The first is to detect when the same state has previously been generated, to prevent expanding it more than once. The second reason is to generate the solution path once a goal is reached. This is done by saving with each node a pointer to its parent node along an optimal path to the node, and then tracing these pointers back from the goal state to the initial state.

The primary drawback of best-first search is its memory requirements. By storing all nodes generated, best-first search typically exhausts the available memory in a matter of minutes on most machines. For example, consider a current workstation with a two gigahertz processor, and two gigabytes of memory. Assume conservatively that a node can be stored in a four-byte word, and that it takes a thousand instructions to generate a node. Under these assumptions, memory would be exhausted in 250 seconds, or just over four minutes. Over time, memories become larger, but processors become faster as well. Disk storage cannot replace memory in the same algorithm, since in order to efficiently detect duplicate nodes, the nodes are normally stored in a hash table, and randomly accessed. Storing the hash table on disk is completely impractical due to disk latencies of about 10 milliseconds. However, see Korf [2004] for another way to use disk storage in best-first search.

1.2. DEPTH-FIRST SEARCH.

One solution to the memory limitation of best-first search is to use a depth-first search instead. Depth-first search stores only the current search path being explored, rather than Open and Closed lists. In a recursive implementation, this path is stored on the call stack. Once a goal is reached, the solution path is constructed by tracing back up the call stack. For example, depth-first iterative-deepening (DFID) [Korf 1985] simulates breadth-first search using memory that is only linear in the maximum search depth, while iterative-deepening-A*

[Korf 1985] simulates A* in linear space. Since their memory requirements are minimal, depth-first searches eliminate the space constraint of best-first search. In a tree, or a graph with very few cycles, a depth-first search is usually the best choice.

The primary drawback of depth-first searches is their time complexity in problem spaces with multiple paths to the same state. For example, consider a rectangular grid, where each node has four neighbors. If we eliminate the parent of a node as one of its children, we have a branching factor of three, and a depth-first search to a radius r would generate $O(3^r)$ nodes. A breadth-first search however, by detecting and eliminating duplicate nodes, would generate only $O(r^2)$ nodes.

While techniques have been developed to eliminate some duplicate nodes in a depth-first search [Taylor and Korf 1993], they are not completely general and only apply to problem spaces with certain properties.

2. *Overview of Article*

The main idea of this article is to reduce the memory requirement of best-first search by storing only the Open list, and deleting Closed nodes once they are expanded. We call this technique *frontier search* since the Open list represents the frontier of the explored region of the problem space. By storing with each Open node which of its neighbors have already been generated, and maintaining a solid search boundary, Closed nodes are never regenerated.

For some applications of breadth-first search, such as formal verification [Rudin 1987], only whether a given state is reachable or not is of interest. For other applications, such as generating pattern database heuristics [Culberson and Schaeffer 1998], the number of moves needed to reach a goal state is required. For many problems, a pattern database stores only the number of moves for each pattern, and not the pattern itself, relying instead on a function that maps each pattern to a unique index in the database. To generate the database, however, requires storing the state representation for each node. Furthermore, it is often effective to generate a much larger pattern database than can fit in memory, and then compress it into memory [Felner et al. 2004]. Thus, reducing the memory required to generate pattern databases is important.

For most applications of best-first search, however, the solution path is needed. Since frontier search doesn't store the interior nodes of the search, generating the solution path requires additional work. This can be done by finding an intermediate state in a solution path, and then recursively solving the problems of finding a path from the initial state to the intermediate state, and from the intermediate state to a goal state, using either unidirectional or bidirectional search.

At first, we ignore constructing the solution path, and describe frontier search on undirected graphs (Section 3.1). We then consider directed graphs (Section 3.2). Next we consider constructing the solution path, first using bidirectional search (Section 4.1), then using unidirectional search (Section 4.2), and finally using disk storage (Section 4.3). In Section 5 we consider the theoretical properties of frontier search, including its correctness (Section 5.2), memory savings (Section 5.3), and time cost (Section 5.4). We then consider several applications of the method. The first is breadth-first search of sliding-tile puzzles and the four-peg Towers of Hanoi (Section 6). The second is Dijkstra's algorithm on large grids with random edge costs (Section 7). Next we consider A* on the Fifteen Puzzle and the four-peg

Towers of Hanoi (Section 8). Finally, we consider optimal sequence alignment (Section 9), where we compare frontier-A* search to dynamic programming approaches. Section 10 presents other related work, followed by our conclusions in Section 11.

Generating a node means creating a data structure to represent the node, while *expanding* a node means generating all of its children. When a node n' is generated by expanding a node n, we say that node n is the parent, and node n' is the child. A problem *state* represents a particular configuration of a problem, while a *node* represents a problem state arrived at via a particular path. In general, there may be multiple nodes representing the same state, arrived at via different paths, which we refer to as *duplicate* nodes. In a directed graph, if there is a operator from node n to node n', we say that n' is a *successor* of n, and that n is a *predecessor* of n'. The *neighbors* of a node in a directed graph are the union of its predecessors and successors, and all adjacent nodes in an undirected graph.

Frontier search is a variation of best-first search, which includes breadth-first search, Dijkstra's algorithm, and A*. Thus, there are frontier search versions of each of these algorithms, referred to as breadth-first frontier search, frontier-Dijkstra's algorithm, and frontier-A*, respectively.

In all of our experiments, we find optimal solutions to our problems. However, frontier search is not restricted to optimal searches, but applies to any best-first search algorithm. For example, weighted-A* is a best-first search with cost $f(n) = g(n) + w * h(n)$ [Pohl 1970]. With $w > 1$, weighted-A* is not guaranteed to return optimal solutions, but frontier search can be applied to weighted-A* to reduce the number of nodes stored. In order to guarantee optimality, however, frontier search requires that the heuristic function be both admissible and consistent.

Our focus throughout this article is on searching implicit graphs that are too large to fit in memory or on disk. Implicit graphs are defined by an initial state and a successor function, rather than an explicit adjacency matrix.

Much of this work initially appeared in Korf [1999], Korf and Zhang [2000], and Hohwald et al. [2003]. Since this work first appeared, other work has built upon it, described in Section 10.

3. *Frontier Search*

Here we consider searching from the initial state to a goal state, ignoring the reconstruction of the solution path. We could view the task as finding the cost of an optimal solution, or determining if a goal state is reachable. We first work through an example in the simpler case of undirected graphs, and then consider directed graphs. Finally, we present the general algorithm.

3.1. UNDIRECTED GRAPHS.

As mentioned above, frontier search maintains only an Open list, and not a Closed list. With each node n we associate a state representation, and a cost value. In addition, we store with each node a vector of *used-operator* bits, one bit for each legal operator. This bit indicates whether the neighboring state reached via that operator has already been generated or not. For example, for a rectangular-grid problem space, where each node has four neighbors, we would store four bits with each node, one for each direction. When we expand a parent node, generating its children, in each child node we mark the operator that would regenerate the parent from that child as used.

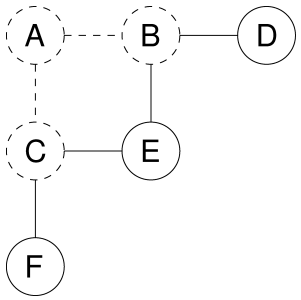

FIG. 1. Frontier search example on undirected graph.

For example, consider the graph fragment shown in Figure 1. The algorithm begins with just the start node on the Open list, say node A in this case, with no operators marked as used. Expanding node A will generate nodes B and C, and node A will be deleted from memory. The west operator from node B will be marked as used, as will the north operator from node C, since those operators would regenerate node A. Nodes B and C will be evaluated by the cost function, and placed on the Open list.

Assume that node B has lower cost and is expanded next. Expanding node B will generate nodes D and E, but not node A, since node B's west operator is used. Node B is then deleted. Node E's north operator, and node D's west operator will be marked as used. Nodes D and E will be evaluated, and placed on the Open list. Assume that node C is expanded next, and then deleted. Expanding node C will generate node F, with its north operator marked as used, and also generate another node corresponding to state E, with its west operator marked as used. Node A will not be generated, since node C's north operator was marked as used.

When the second copy of state E is generated, the first copy will be found on the Open list. Only the copy reached via the lower-cost path will be saved, with both the north and west operators marked as used. In general, when duplicate nodes representing the same state are generated, the operators marked as used in the saved copy are the union of the used operators of the individual nodes.

Figure 1 represents the state of memory at this point in the search, where solid nodes are stored, solid edges represent used operators, and dotted elements are no longer stored. The algorithm terminates when a goal node is chosen for expansion.

3.2. DIRECTED GRAPHS. To prevent the search from "leaking" back into the area already searched, the frontier of Open nodes must remain solid and unbroken. The simple algorithm described above works fine on undirected graphs, but not on a directed graph.

Consider the graph fragment in Figure 2. In this case, the edges are directed, and the only legal operators are to go east or south from a given node. Assume that node A is the start node, and that the nodes are expanded in the order A, B, E. At that point, memory will contain Open nodes C, D, F and G. The inverses of the solid arrows represent the used operators, even though in this case they are not legal operators. If node C is expanded next, it will regenerate node E. Thus, node E will reenter the Open list, even though it has already been expanded. This wouldn't have happened in an undirected graph, since when node E was first expanded, another copy of node C would have been generated, with it's south operator marked as used.

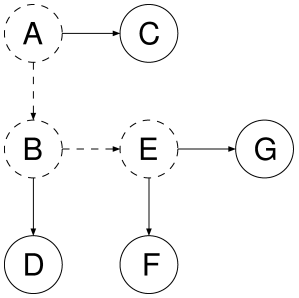

FIG. 2. Frontier search example on directed graph.

A solution to this problem is as follows: When expanding node E, we check to see if any of its predecessor states, nodes B or C, are on the Open list. Since node C is on Open, we mark its south operator as used, and thus when it is expanded later, it won't regenerate node E.

Assume that node G is expanded immediately after node E, and label its neighbor to the north as node H. In that case, when expanding node G, we generate a "dummy" node H, and mark the south operator from this node as used. The purpose of this dummy node is to prevent node G from being regenerated from node H after node G is deleted. The cost of the dummy node H is set to infinity, because it has not yet been reached via a legal path. When node C is subsequently expanded, another copy of node H will be generated, via a legal path with a finite cost. These copies will be merged to a single real copy with the finite cost, with its south and west operators marked as used.

This method requires that we be able to identify all predecessors of a node, including those that cannot be reached from the node via a legal path. The key idea is to guarantee that the frontier does not contain any gaps that can allow the search to leak back into the interior region.

3.3. GENERAL DESCRIPTION OF BEST-FIRST FRONTIER SEARCH. Here we provide a general description for a single-pass of frontier search from the initial state to a goal state, ignoring reconstruction of the solution path.

(1) Place the initial node on Open, with all its operators marked unused.
(2) If there are no nodes on Open with finite cost, terminate with failure.
(3) Else, choose an Open node n of lowest cost for expansion.
(4) If node n is a goal node, terminate with success.
(5) Else, expand node n, generating all successors n' reachable via unused legal operators, compute their costs, and delete node n.
(6) In a directed graph, also generate each predecessor node n' connected via an unused operator, and create dummy nodes for each with costs of infinity.
(7) For each neighboring node n', mark the operator from node n' to node n' as used.
(8) For each node n', if there is no copy of n' representing the same state already on Open, add it to Open.
(9) Else, save on Open only the copy of n' of lowest cost. Mark as used all operators marked as used in any of the copies.
(10) Go to (2)

4. *Reconstructing the Solution Path*

In a typical implementation of best-first search, each node on the Open and Closed lists keeps a pointer to its parent node. Once the search is completed, the pointers are traced back from the goal state to the initial state to construct the solution path. Since frontier search doesn't save the Closed list, an alternative method of recovering the solution path is needed.

The key idea is to identify an intermediate state on the solution path, and then recursively call frontier search to find a path from the initial state to the intermediate state, and from the intermediate state to the goal state.

If we want optimal solutions, the intermediate state must be on an optimal path. Assume that the cost of a solution is the sum of the edge costs on the path. In that case, we have the following principle of optimality: Given any intermediate state on an optimal solution path from an initial state to a goal state, we can construct an optimal solution path from the initial state to the goal state by concatenating any optimal solution path from the initial state to the intermediate state, with any optimal solution path from the intermediate state to the goal state.

We describe below two different methods to reconstruct the solution, one based on bidirectional search, and the other based on unidirectional search.

4.1. Divide-and-Conquer Bidirectional Frontier Search.

The main idea of bidirectional search is to simultaneously search forward from the initial state and backward from the goal state, until the two search frontiers meet [Pohl 1971]. The solution path is the forward search path concatenated with the inverse of the backward search path. Bidirectional search requires an explicit goal state, and that we be able to determine the predecessors of any given state. See Kaindl and Kainz [1997] for several enhancements to the original algorithm.

When a bidirectional frontier search reaches the same state from both directions, we have an intermediate node on a solution path from the initial state to the goal state. We can then recursively solve the subproblems of finding a path from the initial state to the intermediate state, and from the intermediate state to the goal state. The resulting algorithm is called *divide-and-conquer bidirectional frontier search*, or DCBFS for short [Korf 1999].

Given a non-overestimating cost function, we can terminate a unidirectional best-first search with an optimal solution whenever a goal node is chosen for expansion. For a bidirectional search, terminating with an optimal solution is more complex. Every time we reach a common state from from both directions, we have found a solution path. Let c be the cost of the lowest-cost solution path found so far.

For bidirectional Dijkstra's algorithm, the cost of node n is $g(n)$, the cost of the current path from the root to node n. Let $g1$ be the cost of the lowest-cost current Open node generated from the initial state, and $g2$ the cost of the lowest-cost current Open node generated from the goal state. We can terminate with an optimal solution when $c \leq g1 + g2$. The intuitive reason is that any additional solution must pass through an Open node in each direction, and hence its cost must be at least the sum of the minimum-cost Open nodes in each direction.

For bidirectional A*, the cost of a node is $f(n) = g(n) + h(n)$. If h is a non-overestimating heuristic estimate, we can terminate the search with an optimal solution when c is less than or equal to the minimum f cost of any current Open node in either direction [Pohl 1971]. The reason is that with a non-overestimating

heuristic function, the lowest f cost of any Open node is a lower bound on any solutions remaining to be found. Alternatively, we can terminate when $c \leq g1 + g2$.

There are several drawbacks of bidirectional search. First, it is more complex than unidirectional search. In addition to performing forward and backward searches, nodes on both searches must be compared. A consequence is that bidirectional search usually requires more time per node generation. A second drawback is that if the two searches are relatively balanced, and there is relatively little overlap between them, then each of the two searches may do as much work as a unidirectional heuristic search, causing bidirectional search to generate up to twice as many nodes as unidirectional search. Finally, by maintaining two search frontiers, bidirectional frontier search could use up to twice as much memory as unidirectional search. For all these reasons, we present a unidirectional version of frontier search next.

4.2. DIVIDE-AND-CONQUER UNIDIRECTIONAL FRONTIER SEARCH. Consider a problem space with a geometric structure, such as a rectangular grid, and assume we are searching from one corner to the opposite corner. If we draw a line dividing the rectangle into two equal rectangles, any solution path must cross this midline at some point. Furthermore, the point at which it crosses the midline is likely to be near the midpoint of the solution path.

Given such a midline, we perform a unidirectional frontier search until we reach the midline. With each node that is beyond the midline, we store the coordinates of its ancestor on the midline. When we merge duplicate nodes that represent the same state, we save the midpoint on the lowest-cost path to the given state. Once the search chooses a goal node for expansion, the midpoint associated with the goal node will be approximately halfway along an optimal solution path from the initial state to the goal state. The algorithm then recursively computes an optimal path from the initial state to the midpoint, and from the midpoint to the goal state.

Even in nongeometric problems, the midpoint of a solution may be apparent. In the Towers of Hanoi, for example, the middle of an optimal solution for moving all disks from one peg to another is when the largest disk moves to the goal peg.

In a problem space without well-defined midpoints, but with unit edge costs, such as the sliding-tile puzzles, a node n for which $g(n) = h(n)$ is likely to be approximately halfway from the initial state to the goal state, assuming an accurate heuristic function. These nodes could serve as midpoints for a unidirectional search.

In the most general case, we could complete the initial search, keeping track of the number of edges in the path to each node. This will give us the exact length of an optimal solution. We can then redo the initial search, using as midpoints those nodes reached by a path whose length is half the optimal solution length. These techniques for determining midpoints allow us to replace the more complex and expensive bidirectional search with a unidirectional frontier search.

Note that there is no need to continue recursive calls until the initial and goal states are adjacent. Rather, we can count the total number of nodes generated in any recursive call, identified by an initial and goal state. If this number is small enough so that we can store all these nodes in memory, we can use a standard best-first search to construct the path between this pair of initial and goal states.

4.3. USING DISK STORAGE TO RECOVER THE SOLUTION PATH. An alternative algorithm for reconstructing the solution makes use of secondary storage, which has much larger capacity than main memory. We implement a single pass of frontier search, and store with each child node the identity of its parent node along the

solution path to the child. When a node is expanded and closed, it is deleted from memory, but written to a secondary storage device such as a disk, along with the identity of its parent node. Then, once the search is completed, we scan the disk file backwards, looking for the parent of the goal node on an optimal solution path. Once it is found, we continue scanning backwards from that point, looking for the grandparent of the goal node, etc. Since a parent node will always be closed before any of its child nodes, a single backward scan of the disk file will yield the complete solution path.

Unfortunately, most disk files can't be read backwards. To approximate this behavior, we can break up the single file into separate files whose size is slightly less than the capacity of main memory. We can then read the files in the opposite order in which they were written, and scan their contents backward in main memory.

In any case, our analyses and experimental results below indicate that the time and space complexity of frontier search is dominated by the top-level call, and the cost of recovering the solution path is less significant.

5. *Theoretical Properties*

Here we show that a consistent cost function is required to guarantee optimal solutions, that frontier search never regenerates an expanded node in the same pass, and then consider its memory savings and time overhead.

5.1. Requirement for Consistent Heuristics.

Frontier search deletes a node once it is expanded, and never expands a node more than once. To guarantee the admissibility of frontier-A*, the heuristic function must be *consistent*, meaning that for any two nodes n and m, $h(n) \leq c(n, m) + h(m)$, where h is the heuristic function, and $c(n, m)$ is the cost of a lowest-cost path from node n to node m [Pearl 1984]. This is similar to the triangle inequality of metrics, and is stronger than the admissibility requirement that the heuristic not overestimate the actual cost. If the heuristic function is not consistent, then the first time a node is expanded we are not guaranteed to have a lowest-cost path to the node, and frontier-A* is not guaranteed to return an optimal solution. Fortunately, most naturally occurring admissible heuristics are consistent as well. A cost function $f(n) = g(n)$ with no heuristic is also consistent, since it is equivalent to a heuristic function that returns zero everywhere.

5.2. Correctness.

Here we show that frontier search never regenerates a node that has already been expanded, and that it faithfully simulates a standard best-first search with a consistent cost function. We focus here on a single *pass* of the algorithm, which is a call associated with a specific pair of initial and goal states, ignoring the recursive calls to reconstruct the solution path, since that is straightforward.

Theorem 5.1. *In a single pass, frontier search never regenerates a node that has already been expanded.*

Proof. Assume the converse, that there is a node that is expanded, and then later regenerated by the same pass of frontier search. Let n be the first such node, and let node n' be the neighbor of node n from which node n is later regenerated. If node n' was ever expanded before node n was expanded, then node n' would have

been deleted before node n was expanded, and must have been regenerated in order to exist after node n was expanded, in order to regenerate node n. But this couldn't have happened, since by assumption, node n is the first node to be regenerated after it is expanded. Thus, when node n was first expanded, its operator to node n' was not marked as used, and node n either created the first copy of node n', with the operator back to node n marked as used, or marked the operator from node n' to node n as used in the existing copy of node n'. Since node n' was not expanded and regenerated, when it is finally expanded, the operator from node n' to node n would still be marked as used, and node n' wouldn't generate another copy of node n. But this violates our assumption that node n' regenerates node n after node n was expanded. Thus, within a single pass, no expanded nodes are ever regenerated by frontier search. □

Next, we show that even without a Closed list, if the heuristic function is consistent, frontier search simulates standard best-first search.

THEOREM 5.2. *With a consistent heuristic function, or none, and the same tie-breaking rule, the first pass of frontier search expands the same nodes in the same order as standard best-first search.*

PROOF. To prove this, we will actually demonstrate a somewhat stronger result. In particular, after each node expansion, the set of nodes of finite cost on the Open list for frontier search, and their costs, is the same as for best-first search. We will prove this by induction on the number of node expansions. Both algorithms start with just the initial node on Open, which provides our base case. Assume that up through the kth node expansion, the Open list of each algorithm contains the same nodes with the same costs, except for dummy nodes of infinite cost on the Open list of frontier search, and that both algorithms have expanded the same nodes. Consider the next node expansion of each algorithm. Since they break ties identically, both algorithms will expand the same node next.

Standard best-first search will expand node n, generating and computing the cost of each of its child nodes. These nodes will then be compared to those on the Open and Closed lists. There are four different types of child nodes that could be generated, which we will represent by nodes, n'_1 through n'_4. n'_1 is not found on either the Open or Closed list. This node will be added to the Open list. A copy of n'_2 is on the Open list, with an equal or lower cost than that of the newly-generated copy. Thus, the new copy of n'_2 will be discarded. A copy of n'_3 is already on the Open list, but with a greater cost than the newly-generated copy. In that case, the lower-cost newly generated node n'_3 will replace the existing copy on Open. A copy of n'_4 is on the Closed list. With a consistent or no heuristic function, when a node is expanded, it has been reached via a lowest-cost path [Pearl 1984], and hence the newly generated copy of node n'_4 can only have equal or greater cost. Thus, it will be discarded. Thus, the net effect on the Open list of these four types of children is to move the parent node n from Open to Closed, add to Open node n'_1, replace the existing copy of node n'_3 with the newly generated copy of lower cost, and discard the newly generated nodes n'_2 and n'_4.

Frontier search will also expand the same node n, and delete it from Open. Since node n'_4 would be on the Closed list of best-first search, it was previously expanded by best-first search, and by our inductive hypothesis has also been expanded by frontier search as well. Since frontier search never regenerates a node that has already

been expanded in the same pass, we know that node n'_4 will not be regenerated. All other neighbors of node n will be generated. Node n'_1 will either not be found on Open, or it will be found with a cost of infinity. In the former case, it will simply be added to Open, and in the latter case it will replace the infinite-cost copy, after unioning their used-operator bits. Node n'_2 will be found on Open with a lower cost than that of the newly generated copy. Thus, the new copy will be discarded, after merging their used-operator bits. Node n_3 will also be found on Open, but with a greater cost than the newly generated copy. Thus, the new copy will replace the old copy, after merging their used-operator bits. In addition, in a directed graph, additional dummy nodes may be generated and stored on Open, but with infinite costs. If we ignore these nodes, the net effect is to remove the parent node n from Open and delete it, and add nodes n'_1 and n'_3 to Open, which is the same change to the Open list made by best-first search. Thus, after the next node generation, the Open lists of the two search algorithms are still identical, if we ignore the nodes of infinite cost on the Open list of frontier search.

Since the Open lists remain the same, with the exception of dummy nodes of infinite cost, and both algorithms always choose an Open node of lowest-cost to expand next, and frontier search terminates when all Open nodes have infinite cost or when a goal node of finite cost is chosen for expansion, both algorithms expand the same nodes in the same order. □

5.3. MEMORY SAVINGS. How much memory does frontier search save compared to standard best-first search? Since frontier search doesn't save the Closed list, the savings it produces is the size of the Closed list divided by the sum of the sizes of the Open and Closed lists, assuming that a node in each algorithm occupies the same amount of memory, an issue we address below. The magnitude of these savings depends on the structure of the graph searched. For example, in a directed acyclic graph, or an undirected graph with cycles, the more paths there are between a pair of nodes, the greater the size of the Closed list relative to the Open list. The worst case occurs in a tree-structured graph, where there is a unique path between every pair of nodes. In a tree with branching factor b, frontier search saves a fraction of $1/b$ of the total nodes stored, relative to standard best-first search.

To show this, we begin with the following very simple fact:

THEOREM 5.3. *In a tree with i interior nodes, in which every interior node has b children, the total number of interior and leaf nodes is $bi + 1$.*

PROOF. The proof is by induction on the number of interior nodes. In the base case, there are no interior nodes, the tree is just a single leaf node, and $1 = b \cdot 0 + 1$.

For the induction step, assume that in any tree with $i \leq k$ interior nodes and N total nodes, $N = bi + 1$. Now consider a tree T with $k + 1$ interior nodes. Given any such tree, there must be at least one interior node n, all of whose children are leaf nodes. To see this, imagine that the tree is generated by expanding leaf nodes one at a time. The children of the last node expanded must all be leaf nodes. Thus, we can generate a tree T' with k interior nodes by deleting all b children of node n. By the induction hypothesis, T' will have $N' = b \cdot k + 1$ total nodes. To generate tree T from tree T', we simply expand node n, generating b children. Thus, tree T will have exactly b more total nodes than tree T', and one more interior node. Thus, $N' = bk + 1 \Rightarrow N = N' + b = bk + 1 + b = b(k + 1) + 1$. □

An immediate corollary of this theorem is that the ratio of interior nodes to total nodes, i/N, equals $i/(bi+1) \approx i/bi = 1/b$. Thus, as N goes to infinity, the ratio of interior nodes i to total nodes N approaches $1/b$. In a tree, the interior nodes are the Closed nodes. Thus, on a tree, frontier search will save a fraction of $1/b$ of the nodes needed for standard best-first search.

In a tree with fixed branching factor, this result is independent of the shape and size of the tree explored. Thus, a breadth-first frontier search with no heuristic saves the same fraction of memory as a frontier-A* search with an accurate heuristic function. The absolute savings will be greater for the breadth-first frontier search, but the relative savings will be the same.

In an undirected graph with cycles, such as a grid, or a directed graph with multiple paths to the same node, the memory savings of frontier search will be greater than on a tree. For example, in a brute-force search of a grid to a radius r, a standard breadth-first search would store $O(r^2)$ nodes, while breadth-first frontier search would store only $O(r)$ nodes. For a general graph, the ratio of the size of the interior to the total size of a bounded region depends on the structure of the graph and on the shape of the boundary. We will see examples of these savings in our experimental results.

We can summarize these results in the following general theorem:

THEOREM 5.4. *In a tree, frontier search will save a fraction of* $1/b$ *of the nodes stored by best-first search, where b is the branching factor of the tree. This is also the worst-case savings for frontier search in any undirected graph.*

In a general directed graph, however, frontier search could store more nodes than breadth-first search, because of dummy nodes. It is difficult to give a worst-case bound on this excess, since the number of dummy nodes can always be increased by adding additional nodes n, such that there is a directed edge from n to a node of the original graph, and there is either no path to node n, or the only path has a very large cost, guaranteeing that node n will never be generated via a legal path, and hence the associated dummy node will persist once it is generated.

In a search where the in-degree of the nodes expanded equals their out-degree, the number of dummy nodes will not exceed the number of real nodes. In practice, it will be many fewer, since most dummy nodes will eventually be arrived at via a legal path, and hence be replaced by real nodes.

The best-case is a simple linear chain of nodes, in which frontier search only has to store a single node. While this may seem unrealistic, the problem-space graph for the famous Chinese Rings Puzzle has exactly this structure. What makes the problem difficult for people is figuring out what the operators are.

What about the size of a node? In addition to the information that must be stored with each node in a standard best-first search, frontier search must also store a single bit for each operator. For problems with a small number of operators, this is not significant. For example, in the sliding-tile puzzles, there are at most four legal operators, requiring four bits. For the Towers of Hanoi problem, only one bit is needed for each peg, to indicate whether the disk on top of that peg has moved from that state. For problems with a large number of legal operators, however, such as the multiple sequence alignment problem described in Section 9, this additional space can be significant. When aligning d sequences, there are $2^d - 1$ legal operators, each requiring a used-operator bit.

5.4. TIME COST. How fast is frontier search compared to standard best-first search? If we don't reconstruct the solution path, the first pass of frontier search will expand the same nodes as standard best-first search. On an undirected graph, the time per node expansion will generally be less for frontier search than for standard best-first search. The reason is that standard best-first search generates all successors of a given node, and checks each against the Open and Closed lists for duplicates. Frontier search, however, does not generate neighboring nodes that are reached via used operators. In fact, this used-operator mechanism could be used to optimize standard best-first search as well. In a directed graph, however, frontier search may generate dummy nodes, which will increase the time per node expansion.

To reconstruct the solution path, frontier search must expand additional nodes in the recursive calls. The number of such expansions depends on how fast the search space grows with depth, but is usually limited to a small multiplicative factor.

For example, consider a breadth-first search of a space that grows quadratically with depth. In other words, the number of nodes expanded at depth d is proportional to d^2. This is in fact the case for the pairwise sequence-alignment problem described in Section 9. In that case, the number of nodes expanded by standard breadth-first search, as well as the first pass of frontier search, will be proportional to d^2. After the first pass, two recursive searches will be made, one from the initial state to a middle state, and the other from the middle state to the goal state, each of approximately half the original search depth. Thus, the number of nodes expanded in these two searches will be approximately $2(d/2)^2 = 2(d^2/4) = d^2/2$. At the next level of recursion, four searches will be made, each to approximately $1/4$ of the depth of the original search, or $4(d/4)^2 = 4(d^2/16) = d^2/4$. Continuing in this fashion results in the summation $d^2 + d^2/2 + d^2/4 + \cdots$. This sum is bounded by the sum of the corresponding infinite series, which is $2d^2$. Thus, in this case, the total number of node expansions by frontier search is no more than twice the number of node expansions for standard breadth-first search.

The above analysis assumes that the search space grows quadratically with depth. A higher rate of growth results in even lower overhead for frontier search. For example, if the search space grows as the cube of the search depth, as with three-way sequence alignment, then frontier search expands only 1.333 times as many nodes as standard best-first search. These analyses are supported by empirical results on sequence alignment in Section 9.5.4.

If the search space grows exponentially with depth, with a branching factor greater than one, then the overhead as a fraction of the total time goes to zero as the depth increases.

We next consider several versions of frontier search, including breadth-first frontier search, frontier-Dijkstra's algorithm, and frontier-A*, applied to the domains of sliding-tile puzzles, the four-peg Towers of Hanoi problem, finding shortest paths in very large grids, and multiple sequence alignment.

6. *Breadth-First Frontier Search*

6.1. APPLICATIONS OF BREADTH-FIRST SEARCH. Breadth-first search is an important algorithm for several applications. In a problem where any given pair of nodes may have many paths between them, and there is no useful heuristic evaluation function, breadth-first search is the best method for finding optimal solutions.

A second application is the construction of a pattern database heuristic, which stores the number of moves from the goal for each state in a subspace or abstraction of the original problem space [Culberson and Schaeffer 1998]. For example, a pattern database for the sliding-tile puzzles stores the number of moves required to get a subset of tiles to their goal positions. This requires an exhaustive search of the space of different possible permutations of the subset of tiles. A third application is hardware or protocol verification, where we want to fully explore a finite set of states to determine if any error states are reachable from an initial state, or if a particular goal state is reachable [Rudin 1987]. Another application is finding the longest shortest path between any pair of states in a problem space, also known as the diameter of the search space. If all states are equivalent, this can be done by a breadth-first search from any state. For example, finding the diameter of the Rubik's Cube problem space graph is the most famous open problem concerning the Cube, and beyond the ability of breadth-first search on current machines. Finally, breadth-first heuristic search, described in Section 10.3.2 is probably the most memory-efficient heuristic search algorithm for graphs with unit edge costs, and is based on frontier search.

To compute pattern databases, determine whether particular states are reachable, or compute the diameter of a problem space, we only have to perform the first pass of frontier search. To reconstruct the solution path to a given state, we have to execute the recursive calls as well.

Frontier search is not very useful on a tree with a large branching factor, since the memory savings is the reciprocal of the branching factor. Most problem spaces are actually graphs rather than trees, however. In a grid, for example, the number of nodes grows polynomially with depth, and hence the search frontier is asymptotically smaller than the interior. Furthermore, when performing a complete breadth-first search of a finite problem space, as in computing a pattern database or the diameter of a space, frontier search is often very effective. While the size of the space may grow exponentially initially, eventually the number of nodes at a given depth reaches a maximum value, known as the *width* of the problem space, and then decreases until all nodes have been generated. The space complexity of a complete breadth-first frontier search is proportional to the width of the problem space, while the space complexity of standard breadth-first search is proportional to the number of nodes in the space, which is often much larger, as we will see.

6.2. PROBLEM DOMAINS. We describe here three sets of experiments: complete searches of sliding-tile puzzles and four-peg Towers of Hanoi problems, and half-depth searches of the Towers of Hanoi, for moving all disks from one peg to another.

The well-known sliding-tile puzzles, such as the 4×4 Fifteen Puzzle, consist of a rectangular frame filled with numbered square tiles, except for one empty position, called the "blank". Any tile horizontally or vertically adjacent to the blank can be slid into the blank position. The task is to rearrange the tiles from some initial configuration to a particular goal configuration.

In the original three-peg Towers of Hanoi problem, there are three pegs and a set of disks all of different size, initially all located on one peg. The task is to move all the disks to a different goal peg, with the constraints that only one disk can be moved at a time, and a larger disk may never be placed on top of a smaller disk. For the three-peg version, there is a well-known strategy that moves all the disks in a minimum number of moves.

The four-peg Towers of Hanoi, or Reve's Puzzle, adds one more peg. While there exists a strategy that is conjectured to require a minimum number of moves [Frame 1941; Stewart 1941], this conjecture remains unproven [Dunkel 1941], even though the problem was posed in 1908 [Dudeney 1908]. For n disks, the "presumed optimal solution" requires $S(n)$ moves, where $S(n)$ is defined recursively as $S(1) = 1$ and $S(n) = \min\{2S(k) + 2^{n-k} | k \in \{1, \ldots, n-1\}\}$ [Klavzar and Milutinovic 2002]. Best-first search is useful for testing this conjecture for various numbers of disks.

6.3. WHY NOT USE DEPTH-FIRST SEARCH INSTEAD? Since breadth-first search is memory limited, why not use depth-first search (DFS), or depth-first iterative-deepening (DFID) [Korf 1985] instead, which requires memory that is only linear in the search depth? The reason, as explained in Section 1.2, is that a depth-first search can generate many more nodes than breadth-first search, because most duplicate nodes cannot be detected.

For example, the branching factor of the 4×4 Fifteen Puzzle is about 2.13 [Korf et al. 2001], assuming we never generate the parent of a node as one of its children, and its maximum search depth is 80 moves [Brungger et al. 1999]. A depth-first search to that depth would generate $2.13^{80} \approx 1.864 \times 10^{26}$ nodes, compared to only 10^{13} states in the problem space. As another example, the branching factor of the four-peg Towers of Hanoi problem is about 3.766 if the same disk is never moved twice in a row, and the solution depth of the 6-disk problem is 17 moves. A depth-first search to this depth would generate over $3.766^{17} \approx 6 \times 10^9$ nodes, even though there are only $4^6 = 4096$ unique states in the space.

6.4. STANDARD IMPLEMENTATIONS OF BREADTH-FIRST SEARCH. The resource that limits the size of a breadth-first search is memory to store the states. Normally, states are stored in a hash table to detect duplicate nodes efficiently. A hash table must be stored in memory, since randomly accessing data stored on disks is prohibitively expensive due to long latencies. Recent work [Korf 2003; Korf 2004] presents a method to use disk storage for the states, and also applies to frontier search, but is beyond the scope of this article.

The Open list can be represented by a linked list threaded through the hash table entries. This requires an additional word in each entry to store the link. Alternatively, we could store with each state in the hash table the depth at which it first appears. Then, for each level of the search, we scan the table linearly to find and expand the nodes at the current depth. This normally only requires one more byte to store the depth of the state. In this scheme, each level of the search will require time proportional to the size of the hash table, and the first and last few levels will spend most of their time scanning the table, but most of the work is done in the intermediate levels, when the table is relatively full.

6.4.1. *One Bit per State*. For problems where the number of states in the problem is no greater than the number of bits of memory available, we may be able to represent each state with a single bit. For problems such as the sliding-tile puzzles and the Towers of Hanoi, for example, there exist indexing functions that map each state to a unique integer from zero to the number of states minus one. For example, to specify a state of the Towers of Hanoi, we only need to specify the location of each disk, since all the disks on any peg must be stacked in decreasing order of size. Since we can specify one of four disks in two bits, every legal state in a four-peg n-disk problem can be uniquely specified with $2n$ bits. Furthermore, every

$2n$-bit word represents a unique state of the problem. A similar, but more complex indexing function can be constructed to map each solvable sliding-tile puzzle state to a unique integer in the range zero to $(n + 1)!/2 - 1$, where n is the number of tiles [Korf 2005]. The factor of two is due to the fact that only half the possible permutations are solvable [Johnson and Storey 1879].

Given such an indexing function, and sufficient memory, we can allocate an array with one bit per state, where the index in the array uniquely identifies a state. Initially, all bits except the one corresponding to the initial state are set to zero, and as each new state is generated, its corresponding bit is set to one. In addition, we need to maintain an Open queue. Since a queue is accessed first-in first-out, it can be stored on disk. In particular, at each level of the search we maintain two files. One contains the states at the current depth, which we read from, and the other contains the states at the next depth, which we write to. As each state is expanded, generating its children, the index of each child in the bit array is computed, and the corresponding bit is read. If the state has already been generated, it is discarded, but if it hasn't, its corresponding bit is set to one, and the state is written to the output file. Once the input file is exhausted, a level of the search is complete, and the output file becomes the input file for the next level. The algorithm continues until no new states are generated.

This method requires an indexing function, and one bit of memory for each state in the problem space, whether it is generated or not, plus disk files for the FIFO queue. It cannot be used for even partial breadth-first searches in a problem as large as the Fifteen Puzzle, since at one bit per state, the Fifteen Puzzle would require over a terabyte of memory.

6.5. IMPLEMENTATIONS OF BREADTH-FIRST FRONTIER SEARCH. In a breadth-first frontier search, we only store the Open nodes, which are at the current depth or the next greater depth, in a hash table. For each state, this requires enough memory for the state representation, plus a constant number of used-operator bits. The maximum number of states that must be stored is the width of the problem space, rather than the size of the space.

As described above, the nodes at the current depth can be distinguished by threading them on a linked list, or by storing with each node its depth, and linearly scanning the hash table for the nodes at the current depth.

In frontier search, after a node is expanded, it is deleted. Depending on the hash table implementation, this may involve additional work. Collisions in a hash-table are typically handled in one of two ways. Entries that hash to the same location can be chained together on a linked list, in which case deleting entries is straightforward. The drawback of this scheme is the memory required for the links.

Alternatively, with open addressing, an entry that hashes to an occupied location is stored in the next available empty location. When searching for an entry, we start with the target location, and search subsequent locations until either the entry is found, or an empty location is reached. The drawback of this scheme is that a significant fraction of the table must remain empty to keep these searches fast.

To delete an entry in an open-addressed hash table at a given location x, we must examine each occupied location from x forward to the next empty location. If the target index of an entry at location y is less than or equal to x, it must be moved back to location x. This process is then repeated starting with location y, until an

TABLE I. COMPLETE SEARCHES OF SLIDING-TILE PUZZLES

Size	Tiles	Total States	Radius	Width	Depth	Ratio
2×2	3	12	6	2	3	6.000
2×3	5	360	21	44	14	8.182
2×4	7	20,160	36	1,999	24	10.085
3×3	8	181,440	31	24,047	24	7.545
2×5	9	1,814,400	55	133,107	36	13.631
2×6	11	239,500,800	80	13,002,649	49	18.419
3×4	11	239,500,800	53	21,841,159	36	10.966
2×7	13	43,589,145,600	108	1,862,320,864	66	23.406
3×5	14	653,837,184,000	84	45,473,143,333	52	14.379
2×8	15	10,461,394,944,000	140	367,084,684,402	85	28.499
4×4	15	10,461,394,944,000	80	784,195,801,886	53	13.340

empty location is reached. Calculating the relative positions in the hash table is complicated by a circular hash table.

Below, we report the results of complete breadth-first searches on sliding-tile puzzles and four-peg Towers of Hanoi problems.

6.6. SLIDING-TILE PUZZLES. Schofield [1967] performed a complete breadth-first search of a subspace of 20,160 states of the Eight Puzzle, in which the blank is in the center. Reinefeld [1993] computed the optimal solutions for all $9/2 = 181{,}440$ Eight Puzzle states, using IDA* [Korf 1985]. Using breadth-first frontier search, we performed complete breadth-first searches of all sliding-tile puzzles up to the 4×4 and 2×8 Fifteen Puzzles. Table I shows the results.

The first column gives the dimensions of the puzzle, and the second column the number of tiles, which is the product of the dimensions minus one. The third column is the number of states in the complete problem space, which is $(n + 1)!/2$, where n is the number of tiles. The column labeled "Radius" is the maximum distance of any state from the initial state, via a shortest path. In all cases, the blank started in a corner position. The fifth column shows the width of the problem space, which is the maximum number of states at any depth, and the next column gives the corresponding depth. The last column gives the ratio of the number of states divided by the width of the space. The horizontal line below the Eleven Puzzles indicates the largest problem that could be completely searched with previously existing techniques. The Thirteen, Fourteen, and Fifteen Puzzles were searched using the disk-based technique described in Korf [2005].

While the memory required by standard breadth-first search is proportional to the size of the problem space, the memory requirement of breadth-first frontier search is proportional to the width of the space. The ratio of these two values determines the memory savings of frontier search. For the 2×8 Fifteen Puzzle, for example, breadth-first frontier search saves a factor of about 28.5 in memory relative to standard breadth-first search.

For the sliding-tile puzzles, frontier search saves more memory on those problems with a larger aspect ratio. For example, there are two Eleven Puzzles and two Fifteen Puzzles. As expected, the narrower puzzles have a lower branching factor and larger maximum search depth. Thus, the same number of states are spread over a greater depth, resulting in a smaller maximum width.

6.7. FOUR-PEG TOWERS OF HANOI PROBLEM. We also implemented breadth-first frontier search on the four-peg Towers of Hanoi problem. We performed

TABLE II. COMPLETE SEARCHES OF FOUR-PEG TOWERS OF HANOI PROBLEMS

Disks	Total States	Radius	Width	Depth	Ratio
1	4	1	3	1	1.333
2	16	3	6	2	2.666
3	64	5	30	4	2.133
4	256	9	72	7	3.555
5	1,024	13	282	10	3.631
6	4,096	17	918	14	4.462
7	16,384	25	2,568	19	6.341
8	65,536	33	9,060	25	7.234
9	262,144	41	31,638	32	8.286
10	1,048,576	49	109,890	41	9.542
11	4,194,304	65	335,292	52	12.509
12	16,777,216	81	1,174,230	64	14.288
13	67,108,864	97	4,145,196	78	16.190
14	268,435,456	113	14,368,482	94	18.682
15	1,073,741,824	130	48,286,104	111	22.237
16	4,294,967,296	161	162,989,898	134	26.349
17	17,179,869,184	193	572,584,122	160	30.004
18	68,719,476,736	225	1,994,549,634	188	34.454
19	274,877,906,944	257	6,948,258,804	218	39.561
20	1,099,511,627,776	294	23,513,260,170	251	46.761

complete searches of the problem space, and also took advantage of symmetry between the standard initial and goal states to find optimal solutions to those instances while searching to only half the solution depth.

6.7.1. *Complete Searches.* We performed complete breadth-first searches on problems with up to twenty disks. The results are shown in Table II. The first column shows the number of disks. The second column gives the total number of states in the space, which is 4^n, where n is the number of disks. The third column gives the radius of the problem space, from an initial state with all disks on the same peg. The fourth column shows the maximum width of the search space, and the next column gives the depth at which that occurs. The last column is the total number of states divided by the maximum width, which is the memory savings of breadth-first frontier search over standard breadth-first search. For example, for 20 disks, standard breadth-first search would require 46 times more memory than frontier search. The horizontal line below the 16-disk problem indicates the largest problem that could have been solved with previously existing techniques. The remaining problems required the disk-based technique described in Korf [2004].

6.7.2. *Half-Depth Searches.* The standard initial and goal states of the Towers of Hanoi have all disks stacked on one peg. The symmetry between these two states allows us to solve these problem instances much more efficiently [Bode and Hinz 1999]. To move the largest disk from the initial peg to the goal peg, at some point all disks but the largest must be distributed among the two intermediate pegs. We refer to such a state as a *middle* state. Once we have reached a middle state, we can move the largest disk to the goal peg, and then apply the sequence of moves used to reach the middle state in reverse order, interchanging the initial peg with the goal peg. This will have the effect of moving all but the largest disk from the intermediate pegs to the goal peg, solving the problem. If k moves are required to reach a middle state, then the entire problem can be solved in $2k + 1$ moves. Thus,

TABLE III. HALF-DEPTH SEARCHES OF FOUR-PEG TOWERS OF HANOI PROBLEMS

Disks	Optimal	Depth	States Generated	States Stored	Ratio
1	1	0	0	1	0.000
2	3	1	1	1	1.000
3	5	2	4	3	1.333
4	9	4	10	6	1.667
5	13	6	52	30	1.733
6	17	8	148	66	2.242
7	25	12	370	126	2.937
8	33	16	1,690	462	3.658
9	41	20	5,518	1,044	5.285
10	49	24	16,444	3,528	4.661
11	65	32	39,754	9,108	4.365
12	81	40	197,182	36,984	5.332
13	97	48	709,090	116,052	6.110
14	113	56	1,918,594	246,468	7.784
15	129	64	5,073,850	451,236	11.244
16	161	80	12,901,264	1,464,336	8.810
17	193	96	64,287,334	5,778,096	11.126
18	225	112	255,103,138	19,862,262	12.844
19	257	128	813,860,860	59,394,084	13.703
20	289	144	2,133,548,752	135,479,616	15.748
21	321	160	5,468,859,016	262,009,302	20.873
22	385	192	13,561,169,770	781,533,102	17.352
23	449	224	68,858,251,762	3,234,292,848	21.290
24	513	256	291,931,945,152	11,845,070,940	24.646

for the standard initial and goal states, we can solve the problem by searching to only half the solution depth.

Using this technique, we performed breadth-first frontier search from an initial state with all disks on one peg, to the first middle state encountered. Table III shows the results. The first column gives the number of disks. The second column gives the optimal solution length. The third column shows the depth at which a middle state first appears. Multiplying these values by two and adding one gives the optimal solution length. In each case, the optimal solution length matches the presumed optimal solution length, verifying the conjecture for up to 24 disks.

The fourth column gives the total number of states generated by the search, and the fifth column shows the maximum number of states stored. In each case, the maximum number of states stored occurred at the maximum search depth shown in the third column. Finally, the last column shows the number of states generated divided by the states stored, indicating the memory savings of breadth-first frontier search compared to standard breadth-first search. For example, with 24 disks, standard breadth-first search would use more than 24 times the memory used by breadth-first frontier search. For 19 through 22 disks, the disk-based technique described in Korf [2004] was used.

For the three-peg Towers of Hanoi Problem, the number of states with n disks is 3^n. The maximum width of the search space is only 2^n, however. Thus, frontier search would yield an exponential savings in memory for this problem. Of course, search is unnecessary for the three-peg problem, since a simple deterministic algorithm is known to yield an optimal solution.

6.7.3. *Symmetry Among Noninitial Pegs.* Each of these experiments started with all disks on the initial peg. The problems of moving the disks to any of the other

pegs are equivalent, since the three remaining pegs are symmetric. This allows us to reduce the number of nodes generated and stored by a factor of six, by representing each state by a canonical state in which the three noninitial pegs are sorted by the largest disk occupying them [Bode and Hinz 1999]. For ease of comparison, however, the node numbers reported above do not reflect this optimization.

6.7.4. *Previous Work.* Previously, Bode and Hinz [1999] verified the presumed optimal solution length for up to 17 disks. Their algorithm stores only the nodes at the current depth and the previous depth. Each newly generated node is compared against both these node lists to detect duplicates. In contrast, used-operator bits allow frontier search to only store the nodes at the current search depth. They also didn't address the reconstruction of the solution path.

6.7.5. *A Very Surprising Anomaly.* The third column in Table II represents the radius of the problem space, or the maximum depth one must search to generate all states, starting with all disks on one peg. The second column of Table III represents the optimal solution length for transferring all disks from one peg to another. An astute reader might notice that in all cases except 15 and 20 disks, these values are the same. For 15 disks, the goal state is reached from the initial state in 129 moves, but there are still 588 states that are not generated until depth 130. For 20 disks, the goal state is reached in 289 moves, but five more moves are required to generate 11,243,652 additional states.

This result is completely unexpected. For the three-peg Towers of Hanoi problem, it is easy to show that the diameter of the problem space is always the same as the optimal solution length for the standard initial and goal states, and this was believed to be true for the four-peg problem as well. We have no explanation for this anomaly at present.

7. *Frontier–Dijkstra's Algorithm*

If different edges have different costs, then breadth-first search can be generalized to Dijkstra's single-source shortest path algorithm [Dijkstra 1959]. Dijkstra's algorithm is a best-first search where the cost of a node n is $g(n)$, the cost of the current path from the start to node n. While it normally finds a shortest path in a graph from an initial state to all other states, we terminate the algorithm when it finds a shortest path to a particular goal state.

Normally, Dijkstra's algorithm is run on a graph that is explicitly stored in memory. Indeed, if there is sufficient memory available to store the entire graph, Dijkstra's algorithm is perfectly adequate. The problem occurs with an implicitly specified graph that is too big to fit in memory.

To generate such graphs, we chose a square grid, which looks like a square sheet of graph paper. The task is to find a lowest-cost path from the upper left corner to the lower-right corner, where the legal moves from any position are up, down, left, and right. We assign a random cost to each edge of the graph, and compute the cost of a path as the sum of its edge costs. As a result, the lowest-cost paths are almost never shortest paths, in terms of number of edges, but rather include some up and left edges which move away from the goal state.

For grids that fit in memory, we generate and store with each edge a random cost. For grids that are too large to fit in memory, we can't generate a new random value

every time we access an edge, because we have to associate the same cost with a given edge every time we access it. To do this, each edge of the grid is assigned a unique index in the sequence of values produced by a pseudo-random number generator. The cost of an edge is the random number in the sequence at the position that corresponds to its index. Given the index of an edge, we need to determine its cost efficiently, without regenerating the entire pseudo-random sequence. This is done using a technique for efficiently jumping around in a pseudo-random sequence, described in Korf and Chickering [1996].

The space complexity of Dijkstra's algorithm is $O(n^2)$, where n is the length of the side of the square grid. On a machine with a gigabyte of memory, Dijkstra's algorithm is able to solve this shortest path problem on grids up to about 7500 by 7500, which contain over 56 million nodes.

The application of frontier search to Dijkstra's algorithm, called frontier-Dijkstra, is straightforward. Its space complexity is only $O(n)$. On this problem, it stores less than $4n$ nodes. On a one-gigabyte machine, frontier-Dijkstra wouldn't run out of memory on problems with over 12 million nodes on a side. However, since such grids contain 144 trillion nodes, they are not practical due to time considerations. Thus, frontier-Dijkstra completely removes the space constraint on this problem, allowing us to solve problems as large as we have time for.

The bidirectional version of frontier-Dijkstra generates about three times as many nodes as Dijkstra's algorithm on this problem, but its running time is less than twice as long [Korf 1999]. The reason is that frontier search doesn't regenerate nodes that have already been expanded, nor look them up in the hash table, making it faster per node expansion than Dijkstra's algorithm.

8. *Frontier-A* Algorithm*

In addition to breadth-first search and Dijkstra's algorithm, frontier search can also be applied to A* [Hart et al. 1968]. We applied frontier-A* to three different domains, the 4×4 Fifteen Puzzle, the four-peg Towers of Hanoi problem, and optimal sequence alignment, which is described in Section 9.

8.1. 4×4 Fifteen Puzzle.

For the Fifteen Puzzle, approximate midpoints on a solution path can be identified by choosing nodes n for which $g(n)$ is equal to $h(n)$, the heuristic estimate of the distance from node n to the goal. A* is limited by the amount of memory available. For example, with a gigabyte of memory, we can store about 30 million nodes. A* with the Manhattan distance heuristic function is only able to solve 79 of the 100 random problems in Korf [1985] before running out of memory. Using frontier-A* with the same amount of memory allows us to solve 94 of the 100 problems.

The fraction of memory saved, which is the number of nodes expanded divided by the number of nodes generated, is about 57% for all 100 problems. Thus, frontier search more than doubles the effective memory on this problem. The branching factor of the 4×4 Fifteen Puzzle is about $b = 2.13$. Thus, if this problem space were a tree instead of a graph, the memory savings would be only $1/b = 1/2.13 \approx 47\%$, as described in Section 5.3.

8.2. Four-Peg Towers of Hanoi.

The best method for finding an optimal solution for transferring all disks from one peg to another in the four-peg Towers

of Hanoi problem is a breadth-first frontier search from the initial state to a middle state where all but the largest disk are distributed over the two intermediate pegs, as described in Section 6.7.2. A* is not directly applicable to this problem instance, because we don't know the goal state, or exactly how the disks should be distributed over the intermediate pegs. The technique described in Section 6.7.2 is not applicable to finding a shortest solution path between arbitrary initial and goal states, however, but A* is.

We applied Frontier-A* to this problem using various heuristic functions, ranging from one based on a relaxation of the problem with an infinite number of pegs, to different pattern database heuristics. Using these heuristics, we were able to solve up to the 17-disk, four-peg problem. The memory savings due to frontier-A* compared to standard A* ranged from a factor of six to a factor of over 13. On the 17-disk problem, with the most accurate heuristic function, frontier-A* saved an order of magnitude in memory. See Felner et al. [2004] for a full discussion of these experiments.

9. *Optimal Sequence Alignment*

Sequence alignment is the most important practical problem to which frontier search has been applied so far. Given a set of character strings, which may represent sequences of DNA base pairs or amino acid sequences in a protein, the alignment task is to insert gaps in the strings, in order to maximize the number of matches between corresponding positions of the resulting strings. One application of this problem, for example, might be to determine regions of the human and mouse genomes that most closely resemble each other. The simplest case is pairwise alignment, in which two strings are aligned.

9.1. PAIRWISE ALIGNMENT. For example, consider two DNA sequences `ACGTACGTACGT` and `ATGTCGTCACGT`. If we insert a gap in the first string after the eighth character, and insert a gap in the second string after the fourth character, then all the letters in corresponding positions are the same, except for the substitution of `T` for `C` in the second position, as shown below.

```
ACGTACGT-ACGT
ATGT-CGTCACGT
```

The quality of an alignment is defined by a cost function. For example, we might charge a penalty of one unit for a mismatch or substitution between characters, and two units for a gap in either string. The cost of an alignment is the sum of the individual substitution and gap costs for each pair of characters. With this cost function, the alignment above has a cost of five, since there is one substitution and two gaps. An optimal alignment of a pair of strings is an alignment with the lowest cost, and the above alignment is optimal. Real cost functions are more complex, involving different substitution penalties for different pairs of characters, and more complex gap functions, such as a fixed cost for each gap plus an additional cost related to its length.

This problem can be mapped to the problem of finding a lowest-cost corner-to-corner path in a two-dimensional grid [Needleman and Wunsch 1970]. One sequence is placed on the horizontal axis from left to right, and the other sequence on the vertical axis from top to bottom. An alignment is represented by a path

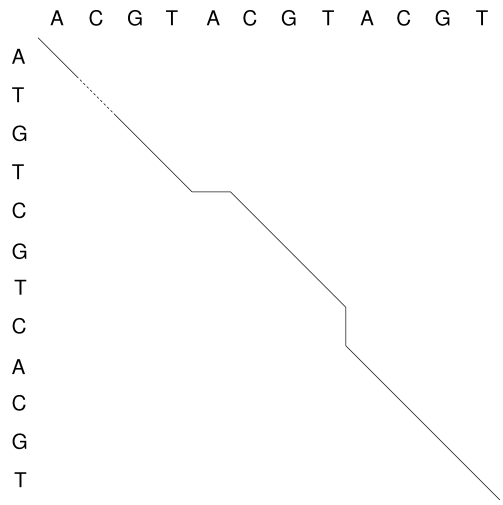

FIG. 3. Sequence alignment as path-finding in a grid.

from the upper-left corner of the grid to the lower-right corner. Figure 3 shows the path that represents our example alignment. If there is no gap in either string at a given position, the path moves diagonally down and right, since this consumes both characters. The cost of such a move is zero if the corresponding characters match, or the substitution penalty if they differ. A gap in the vertical string is represented by a horizontal move right, since the gap consumes a character in the horizontal string, but leaves the position in the vertical string unchanged. Similarly, a gap in the horizontal string is represented by a vertical move down. Horizontal and vertical moves are charged the gap penalty. Given this mapping, the problem of finding an optimal sequence alignment corresponds to finding a lowest-cost corner-to-corner path in the grid, where the legal moves at each point are down, right, and diagonally down and right.

9.2. MULTIPLE SEQUENCE ALIGNMENT. This problem readily generalizes to aligning multiple strings simultaneously. To align three strings, we find a lowest-cost path in a cube from one corner to the opposite corner. The cost of a multiple alignment is usually computed as the sum-of-pairs cost, or the sum of each of the different pairwise alignments induced by the multiple alignment [Setubal and Meidanis 1997]. For example, if we exhibit a three-way alignment by writing each string directly above the next so that corresponding characters are aligned vertically, then this also exhibits an alignment between each pair of strings as well. Equivalently, we can score each character position by summing the cost associated with each pair of characters. For example, if we have a `C`, a `G`, and a gap at one position, the cost at that position is two gap penalties (C- and G-) plus a substitution penalty (CG). If we have two gaps and a character at one position, the cost is two gap penalties, since no cost is charged for two gaps opposite each other.

9.3. RELATED WORK ON SEQUENCE ALIGNMENT. Before applying frontier search to this problem, we review previous work on sequence alignment. For simplicity, we describe this work in the context of pairwise alignment, but all the algorithms generalize to multiple alignments as well. We only consider here algorithms that return optimal alignments.

9.3.1. *Dynamic Programming.* Sequence alignment is a special case of the general shortest-path problem in a grid. The only legal operators are to move right, down, or diagonally down and right. Thus, we can a priori divide the neighbors of a node into its predecessors and successors. For any node in the grid, its predecessors must be to the left of it, or above it, or both, and its successors must be to right of it, or below it, or both. As a result, this problem is simpler than the general shortest-path problem described in Section 7, and we can apply dynamic programming to it, rather than Dijkstra's more general algorithm.

Dynamic programming scans the grid from left to right and from top to bottom, storing with each node the cost of a lowest-cost path from the start node to that node. For each node, we add the gap cost to the cost of the node immediately to its left, and to the cost of the node immediately above it. To the cost of the node diagonally above and to the left, we add either the substitution cost, in the case of a mismatch, or no cost, in the case of a match. We then store the smallest of these three sums as the cost of the current node, and also store an indication of which parent node gave rise to the lowest cost. Once we reach the bottom right corner, we trace back through these indicators to recover a lowest-cost solution path, which corresponds to an optimal alignment of the two strings.

This algorithm generalizes to multiple alignments. To align d strings, we search for a lowest-cost corner to corner path in a d-dimensional hypercube.

The time complexity of this algorithm is $O(n^d)$, where n is the length of the strings, and d is the number of strings being aligned. The space complexity is also $O(n^d)$, because we store every node of the grid in memory. Thus, in practice, this algorithm is severely space bound. In a gigabyte of memory, dynamic programming can only align pairs of strings of about thirty thousand characters each, triples of strings of about a thousand characters each, and quadruples of strings of about 175 characters each. These are larger problems than can be solved with Dijkstra's algorithm, because dynamic programming requires less memory per node.

9.3.2. *Hirschberg's Algorithm.* Hirschberg [1975] presented an algorithm for the related problem of computing a maximal common subsequence of two character strings in linear space. His algorithm is based on a two-dimensional grid, with each of the strings placed along one axis. A node of the grid corresponds to a pair of initial substrings of the original strings, and contains the length of a maximal common subsequence of the substrings.

The standard dynamic programming algorithm for this problem requires $O(n^2)$ time and $O(n^2)$ space, for two strings of length n. To compute an element of the grid, however, we only need the values to its immediate left and immediately above it. Thus, we only need to store two rows at a time, deleting each row as soon as the next row is completed. In fact, only one row is needed, since we can replace elements of the row as soon as they are used. Unfortunately, this only yields the length of a maximal common subsequence, and not the subsequence itself.

Hirschberg's algorithm computes the first half of this grid from the top down, and the second half from the bottom up, storing only one row at a time. Then, given the two middle rows, one from each direction, it finds a column for which the sum of the two corresponding elements is a maximum, since this is a maximization problem. This point splits the original strings in two parts, and the algorithm is then called recursively on the initial substrings, and on the final substrings. Hirschberg's algorithm is easily generalized to solve the sequence-alignment problem, in two or more

dimensions. This reduces the asymptotic space complexity of the d-dimensional alignment problem from $O(n^d)$ to $O(n^{d-1})$. The additional cost in time is only a constant factor of two in two dimensions, and even less in higher dimensions.

In fact, bidirectional frontier search can be viewed as a generalization of Hirschberg's algorithm from the special case of dynamic programming to the more general case of best-first search. In addition, Hirschberg [1997] credits David Eppstein with suggesting a unidirectional version of his dynamic programming algorithm, which can be viewed as a special case of unidirectional frontier search. The key difference between dynamic programming and best-first search is that dynamic programming processes the nodes of the graph in a fixed order, and hence has to distinguish a priori the parents of a node from its children. Best-first search does not require this. For example, dynamic programming can be used to find lowest-cost paths in the directed grids generated by sequence alignment problems, but a best-first search such as Dijkstra'a algorithm or A* must be used to find lowest-cost paths in the more general undirected grids described in Section 7 above.

9.3.3. *Bounded Dynamic Programming (BDP).* Hirschberg's algorithm can be improved by using upper and lower bounds on the solution cost [Spouge 1989]. For pairwise alignments, a lower bound is the number of vertical or horizontal moves that must be made to reach the lower right corner of the grid, times the gap penalty.

For multiple sequence alignment, the usual cost function is the sum of the costs of the induced pairwise alignments. In general, an optimal alignment of multiple strings may not induce an optimal alignment of any of the pairs of strings. However, since the cost of an optimal alignment must be less than or equal to the cost of any alignment, the sum of the costs of the optimal pairwise alignments must be a lower bound on the cost of an optimal simultaneous alignment of the group of strings.

A simple upper bound on the cost of an optimal alignment is the cost of aligning the strings with no gaps, but this is not very accurate. Instead, for the top-level call, we perform a series of iterations with gradually increasing upper bounds, starting with a lower bound on the alignment cost, until a solution is found [Ukkonen 1985]. We refer to this algorithm as iterative-deepening bounded dynamic programming (IDBDP). For each recursive call to construct the actual alignment, we use the known exact optimal solution cost from the parent call as the upper bound.

Given an upper bound on the optimal alignment cost, and a lower bound on the cost of aligning any pair of substrings, we can limit the dynamic programming algorithm to those nodes for which the cost to reach them, plus the estimated cost to reach the goal, is less than the upper bound.

9.3.4. *A* Algorithm.* A* has also been applied to optimal sequence alignment by several authors, including Araki et al. [1999], Ikeda and Imai [1999], and Lermen and Reinert [2000]. All these implementations are constrained by the memory requirements of A*, which are much greater than that of Hirschberg's algorithm. It is only feasible for highly correlated strings.

9.4. DIVIDE-AND-CONQUER FRONTIER SEARCH. As an alternative to dynamic programming, we can use best-first frontier search for optimal sequence alignment. We implemented both bidirectional and unidirectional divide-and-conquer frontier search for this problem. Since the problem is modelled by searching for a lowest-cost path in a directed graph, we used the directed-graph version, requiring dummy nodes as described in Section 3.2.

The particular best-first search we used is based on A*. We refer to the unidirectional frontier-search version of A* that reconstructs the solution path as divide-and-conquer frontier A*, or DCFA*, and the bi-directional version as divide-and-conquer bidirectional A* or DCBDA*. Each node n of the search represents an alignment of a set of initial substrings of the given strings. For example, in a two-way alignment, the node n whose coordinates are (x, y) represents an alignment of the first x characters of the string on the horizontal axis, and the first y characters of the string on the vertical axis. The $g(n)$ value of that node is the cost of the best alignment of those two initial substrings found so far. For two-way alignments, the heuristic function $h(n)$ is the number of vertical or horizontal moves that must be made to reach the lower-right corner of the grid, times the gap penalty.

When aligning more than two sequences, the heuristic used is the sum of the costs of the optimal pairwise alignments. At a given node n, representing an alignment of the corresponding string prefixes, we want $h(n)$ to estimate the lowest-cost alignment of the remaining string suffixes. To do this, for each pair of strings, we run the dynamic programming algorithm backwards from the goal state in a preprocessing phase, storing in a separate matrix for each pair of strings the cost of the optimal pairwise alignment of each pair of suffix strings. Then, during the actual search, we lookup the corresponding $h(n)$ value for each pair of remaining suffix strings, and sum them to get a lower-bound heuristic on the cost of the remaining alignment.

The particular cost function we used charges no penalty for a match, one unit for a mismatch, and a two units for a gap in either string. In the case of pairwise alignments induced from multiple alignments, if there is a gap in both strings at the same position, no penalty is charged. There is nothing special about this cost function, and other cost functions could be substituted instead.

9.5. EXPERIMENTAL RESULTS. We performed numerous experiments with both randomly generated strings and real protein sequences, aligning between two and five sequences at a time. We compared Hirschberg's algorithm, the best dynamic programming approach (IDBDP), DCBDA*, and DCFA*. We describe these experiments here. Full details can be found in Hohwald et al. [2003], Korf and Zhang [2000], and Thayer [2003].

For pairwise alignment, Hirschberg's algorithm runs faster than DCFA*. Its speed is a consequence of the fact that almost all of its memory access is sequential, resulting in excellent cache performance. Best-first search algorithms, on the other hand, access their memory in a much more random pattern, resulting in many cache misses. For pairwise alignment with either Hirschberg's algorithm or frontier search, the space complexity is $O(n)$ compared to a time complexity of $O(n^2)$, and hence memory is not a practical constraint, but rather time is the limiting resource.

9.5.1. *Three-Dimensional Alignments.* For aligning three strings simultaneously, the sum of the optimal pairwise alignments is an effective heuristic function, and Hirschberg's brute-force algorithm is not competitive. Rather, we use bounded dynamic programming (BDP) or iterative-deepening BDP (IDBDP). The heuristic function is precomputed and stored in lookup tables, and the space complexity for these tables and the search nodes is $O(n^2)$, making memory the limiting factor.

Table IV [Korf and Zhang 2000] shows the results for aligning triples of strings. For each problem instance, we optimally aligned three random strings, from an alphabet of 20 characters to simulate amino acid sequences. The cost function

TABLE IV. 3-WAY ALIGNMENT OF RANDOM 20-CHARACTER STRINGS

	DCBDA*			DCFA*			IDBDP		
N	Nodes	Mb	Sec	Nodes	Mb	Sec	Nodes	Mb	Sec
1	4,385,088	48	5	2,428,388	15	2	1,478,166	20	2
2	45,956,680	119	50	24,147,383	56	17	12,419,072	78	16
3	184,459,312	248	212	94,778,522	117	78	46,080,688	173	56
4	464,723,712	472	666	236,315,529	221	192	111,106,632	306	137
5	979,234,880	575	1398	496,715,891	272	499	237,084,672	478	275
6				883,732,606	463	897	466,050,656	689	556

charges nothing for a match, one unit for a substitution, and two units for a gap. Each data point is the average of 100 problem instances. We ran three different algorithms, all of which return optimal alignments. For each algorithm we give the average number of nodes generated, the amount of memory used in megabytes (MB), and the average running time per problem instance in seconds, on a 440 Megahertz Sun Ultra 10 workstation with 640 MB of memory. The first column gives the length of each string, in thousands of characters.

Hirschberg's algorithm is not competitive for more than two strings, since it doesn't use a lower bound, and generates every node at least once. For example, for three strings of length 1000, it generates 1.3 billion nodes, and takes 405 seconds to run, compared to a few seconds for the other algorithms.

The first algorithm listed in Table IV is divide-and-conquer bidirectional A* (DCBDA*), and the second is divide-and-conquer frontier A* (DCFA*). DCFA* generates about half the nodes of DCBDA*, and runs more than twice as fast, due to lower overhead per node. 640 megabytes of memory was exhausted by DCBDA* on strings of length 6000.

The last algorithm in the table is iterative-deepening bounded dynamic programming (IDBDP). IDBDP generates fewer nodes than DCFA*, because DCFA* generates dummy nodes to keep the search from leaking into the closed region. IDBDP also runs faster than DCFA* on the longer strings, but our implementation used more memory than DCFA*. Thus, for small problems where sufficient memory is available to run both algorithms, IDBDP is preferred, but DCFA* can solve larger problems with the same memory.

The absolute performance of these algorithms is sensitive to the cost function, the size of the alphabet, and the correlation of the strings. Thus, these results can only be used for relative comparison of the different algorithms.

9.5.2. *Optimal Alignment of Four and Five Random Strings.* We also experimented with simultaneous optimal alignment of four and five random strings, chosen from an alphabet of four characters, simulating DNA sequences. These experiments were run on a 1.8-gigahertz Pentium 4 machine with a gigabyte of RAM. Full details can be found in Korf [2003]. To simulate more realistic problems, we generated sequences with varying degrees of similarity, as follows: We first generated a reference string randomly and uniformly from the alphabet. Each character in each of the strings to be aligned was the same as the corresponding character in the reference string with probability λ, and randomly chosen among all the characters, including the reference character, with probability $1 - \lambda$. Thus, strings generated with $\lambda = 1$ are all identical, and those generated with $\lambda = 0$ are uncorrelated.

Since DCFA* dominated DCBDA* in our previous experiments, we did not consider it further. To be conservative, we compared DCFA* to bounded dynamic

TABLE V. 4-WAY ALIGNMENT OF RANDOM 4-CHARACTER STRINGS OF LENGTH 400

	BDP-PB		DCFA*	
λ	Nodes	Time	Nodes	Time
1.00	77,653	59.82	5,601	0.06
.75	77,690	59.83	5,601	0.07
.50	220,262	60.06	32,971	0.27
.25	21,109,562	78.12	762,740	38.20
0.00	34,838,768	89.79	1,286,730	57.90

TABLE VI. 5-WAY ALIGNMENT OF RANDOM 4-CHARACTER STRINGS OF LENGTH 90

	BDP-PB		DCFA*	
λ	Nodes	Time	Nodes	Time
1.00	33,896	15.41	2,701	0.016
.75	33,896	15.39	2,701	0.015
.50	39,199	15.33	5,246	0.022
.25	351,878	16.08	97,408	0.780
0.00	1,009,293	17.61	110,742	1.460

programming with perfect bounds (BDP-PB), which is BDP with the optimal solution cost as an upper bound on its first iteration. This algorithm is not feasible in practice, since the optimal solution cost isn't known a priori, but it provides the best possible performance for BDP.

Table V shows the results for four strings of length 400, and Table VI shows the results for five strings of length 90. Each data point is an average of 50 problem instances. For each algorithm and each value of λ, we show the average number of nodes generated, and the average running time in seconds.

The easiest strings to align are identical strings ($\lambda = 1$), and the hardest are those that are completely uncorrelated ($\lambda = 0$). This difference is particularly pronounced for DCFA*, where the difference is almost three orders of magnitude for aligning four strings. This explains why performance results on random uncorrelated strings often look much worse than on real strings, which are often highly correlated.

DCFA* runs faster than BDP-PB, and generates fewer nodes, requiring less memory. The memory difference is about an order of magnitude. The time difference ranges from about 50% for four uncorrelated strings, to three orders of magnitude for identical strings. The time difference is more pronounced for five strings than four, suggesting that DCFA* may outperform BDP-PB for more strings as well.

9.5.3. *Optimal Alignment of Real Proteins.* Finally, we ran a similar set of experiments on amino acid sequences of real proteins from the BAliBASE database of benchmark alignments [Thomson et al. 1999]. We used groups of five or fewer sequences from the first and third sets of BAliBASE (ref1 and ref3), since neither algorithm could solve problems from the other sets, which involved very large numbers of sequences. The results of these experiments are presented in Table VII, where the last two columns give the running times of their respective algorithms in seconds. As in the case of random sequences, DCFA* is faster than BDP-PB for four and five dimensional problems. A number of the real sequence sets were solvable by DCFA* but not by BDP-PB because of memory constraints.

In four and five dimensions, DCFA* outperforms BDP-PB in both time and space. The main reason is that in higher dimensions, bounded dynamic programming

TABLE VII. 4-WAY AND 5-WAY ALIGNMENT OF REAL PROTEINS

Sequence Set	Number	Length	BDP-PB	DCFA
1uky	4	186	3.35	2.14
3grs	4	201	4.35	2.65
1zin	4	206	4.55	2.35
2hsdA	4	223	6.66	3.55
451c	5	70	4.84	0.18
1plc	5	88	13.55	0.40
9rnt	5	96	—	0.41
2mhr	5	110	—	0.75
5ptp	5	222	—	9.57
1ton	5	224	—	40.12
2cba	5	237	—	17.91
kinase	5	263	—	24.23
gal4	5	335	—	76.42
arp	5	380	—	655.47
glg	5	438	—	280.80
1ajs (ref3)	4	365	44.38	15.78
1pamA (ref3)	4	404	65.92	23.31
1pamA (ref3)	5	63	3.20	0.13

becomes very difficult to implement, and this complexity results in significant time and space overheads. The complexity comes from bounding the region of the cube or hypercube that must be examined, based on the lower and upper bounds on solution cost. This bounding is essential, since otherwise the entire solution space would have to be searched, which is prohibitive for the larger problems.

9.5.4. *Overhead to Reconstruct the Solution Path.* In Section 5.4, we argued that the time overhead of frontier search to reconstruct the solution path was only a small multiplicative constant. Our sequence-alignment experiments confirm this. For three random uncorrelated strings of length 8000, the node expansion overhead of the recursive calls to compute the actual alignment in DCFA* is about 25%, compared to just running the first pass of the algorithm. As predicted, this decreases with increasing numbers of strings. For four strings of length 2000, the overhead is about 7%; for five strings of length 1000, it is about 2%; and for six, seven, or eight strings, it is less than 1%.

For real protein sequences, the overhead is higher, ranging from 8% to 48%. The reason is that many of these sequences are highly correlated, producing a search volume that is highly elongated along the diagonal axis of the hypercube. As a result, the volumes searched by the recursive calls are a higher percentage of the volume searched on the first pass.

To be conservative, in these experiments we continued the recursion all the way to adjacent nodes, rather than running A* at the point where there was sufficient memory to do so. Note that bounded dynamic programming incurs a similar overhead to reconstruct the actual alignment.

In summary, unidirectional frontier-A* search outperforms both bidirectional frontier search, and the best existing competitor, bounded dynamic programming, for optimally aligning four or five strings. In three dimensions, DCFA* runs a little slower, but uses less memory than BDP, and hence can solve larger problems in practice. We believe DCFA* will also perform well for even more strings, but at some point the memory required for storing the used-operator bits becomes

significant. Thus, optimal sequence alignment remains an important real-world application of frontier search.

10. *Other Related Work*

Since the purpose of frontier search is to reduce the memory required by best-first search, we consider here other approaches to this problem, in roughly chronological order. We divide these into earlier approaches, recent approaches motivated by sequence alignment, and subsequent work that builds upon frontier search.

10.1. EARLIER APPROACHES. Depth-first algorithms, such as depth-first iterative-deepening (DFID) and Iterative-Deepening-A* (IDA*) [Korf 1985], and Recursive Best-First Search (RBFS) [Korf 1993], only require memory that is linear in the maximum search depth. As a result, they cannot detect most duplicate nodes, and can be extremely inefficient on graphs with multiple paths to the same node, as explained in Sections 1.2 and 6.2.

MREC [Sen and Bagchi 1989] is a hybrid combination of best-first and depth-first search. It executes a best-first search until memory is almost exhausted. Then, it executes a series of iterations of IDA* on each of the Open nodes, until the goal is reached. Unfortunately, these depth-first iterations suffer the same duplicate-explosion problem of other depth-first searches.

MA* [Chakrabarti et al. 1989] is an elegant algorithm designed to use any amount of memory available for a best-first search. Initially, it behaves as a standard best-first search until the available memory is almost exhausted. Then, in order to reclaim memory to expand the best Open node, it identifies a set of sibling nodes on the Open list of highest cost, backs up their minimum cost to their parent node, deletes the child nodes from memory, and moves their parent node back to the Open list. The algorithm continues retracting the worst nodes, and expanding the best nodes, until a goal is chosen for expansion.

Unfortunately, there are several serious drawbacks to this algorithm. One is that many nodes are retracted and expanded multiple times. Another is that when a node is generated, there is no guarantee that duplicate nodes that have previously been generated will still be in memory, subjecting it to the duplicate-explosion problem of depth-first search. Finally, the constant-factor overhead of this algorithm is significantly greater than for standard best-first search. At least two subsequent algorithms have been proposed to address this constant overhead, including simplified MA* (SMA*) [Russell 1992] and ITS [Ghosh et al. 1994]. Despite these attempts, however, these algorithms are not competitive on real problems.

10.2. APPROACHES MOTIVATED BY SEQUENCE ALIGNMENT. A simple idea that reduces the space required by best-first search is given an upper bound on the cost of an optimal solution, there is no need to store nodes whose cost exceeds that upper bound, since they can't be on an optimal solution path. This was proposed by Spouge [1989] in the context of dynamic programming for sequence alignment, and applied to A* for sequence alignment by Ikeda and Imai [1999]. Zhou and Hansen [2002] use this idea in conjunction with weighted-A* to get better upper bounds. This idea can be applied to frontier search as well to reduce the size of the Open list.

A* with partial expansion (PEA*) [Yoshizumi et al. 2000] significantly extends this idea. Full expansion of a node means generating and storing all its children.

In PEA*, only those children whose cost is below some threshold are stored, in the hope that the remaining children will never be needed. The parent node is also saved, allowing subsequent children to be generated later, if the cost threshold increases. This is particularly useful in multiple sequence alignment, since the branching factor of a node is $2^d - 1$, where d is the number of sequences being simultaneously aligned. PEA* also stores an expansion priority with each node, requiring additional memory per node.

This idea can also be combined with frontier search, although it reduces the effectiveness of frontier search, since a node cannot be deleted from memory until it has been fully expanded. In our experiments [Thayer 2003] on sequences of length 4, 5, and 6, the combination of frontier search with partial expansion generally stored the fewest nodes, but simple PEA* or frontier-A* often consumed less memory.

10.3. APPROACHES THAT BUILD UPON FRONTIER SEARCH. Since this work initially appeared [Korf 1999], and was extended to sequence alignment [Korf and Zhang 2000], Zhou and Hansen have developed several algorithms that build on frontier search, including *sparse-memory graph search*, *sweep-A**, and *breadth-first heuristic search*.

10.3.1. *Sparse-Memory Graph Search.* Instead of storing just the Open nodes, sparse-memory graph search [Zhou and Hansen 2003a] stores some of the Closed nodes as well. In particular, a node is not deleted from memory until all its neighbors have been expanded. This replaces the need for dummy nodes in directed graphs, but at the cost of approximately doubling the number of nodes that must be stored. This also allows frontier search to be more easily combined with other enhancements, such as pruning Open nodes whose cost exceeds an upper bound.

By storing nodes until all their neighbors have been expanded, sparse-memory graph search eliminates the need to store a used bit for each operator, since a neighbor generated via a used operator will still be in memory. Rather, one can simply store the number of operators that have already been used. When merging duplicate nodes, we simply add their numbers of used operators. When all the operators of a node have been used, it can be deleted from memory, even if it hasn't been expanded yet. This is particularly useful when aligning large numbers of sequences, since the number of possible operators is as $2^d - 1$, where d is the number of sequences being aligned.

10.3.2. *Sweep-A*, Bounded Diagonal Search, Breadth-First Heuristic Search, and Breadth-First Iterative-Deepening-A*.* Zhou and Hansen also developed two similar algorithms, called *Sweep-A** for the special-case of lattice graphs [Zhou and Hansen 2003b], and *Breadth-First Heuristic Search* for the special-case of graphs with unit edge costs [Zhou and Hansen 2004]. Thayer [2003] independently developed an algorithm similar to Sweep-A*, called *Bounded Diagonal Search*, for sequence alignment. We describe breadth-first heuristic search here, since it is the simplest of these algorithms. Frontier search stores in memory the entire Open list at any given time. Since the Open list will often have an irregular shape and include nodes at different depths, Zhou and Hansen observed that the maximum size of the Open list is often much greater than the maximum width of the graph, which is the maximum number of nodes at any single depth. Given an upper bound on total cost, Breadth-First Heuristic Search performs a Sparse-Memory Graph Search, pruning nodes whose cost exceeds the upper bound. This guarantees that the

maximum memory requirement will be no more than twice the width of the graph. To turn this into a best-first search, they perform a series of iterations with increasing upper bounds, resulting in an algorithm they call *Breadth-First Iterative-Deepening-A**.

11. *Conclusions*

We have presented a new, general-purpose best-first search algorithm. Frontier search saves only the Open list, and not the Closed list, thus reducing the memory required by standard best-first search. Frontier search can be applied to any best-first search, including breadth-first search, Dijkstra's algorithm, or the A* algorithm. If the cost function is consistent, including most heuristic functions in practice, then frontier search faithfully simulates best-first search, and guarantees finding optimal solutions if they exist. The solution path can be reconstructed by a divide-and-conquer technique, using either bidirectional or unidirectional search.

The amount of memory saved depends on the problem space. In the worst case, a tree, frontier search saves a fraction of $1/b$ of the memory required by standard best-first search, where b is the branching factor of the tree. In directed acyclic graphs, or undirected graphs with cycles, the savings are greater. For example, when searching a complete d-dimensional hypercube with sides of length n, frontier search reduces the asymptotic memory requirement of best-first search from $O(n^d)$ to $O(n^{d-1})$. The time overhead to reconstruct the solution path is a small multiplicative constant, never more than two in any of our analyses or experiments.

We experimented in four different domains. One was finding lowest-cost paths in a grid with random edge costs, using frontier-Dijkstra's algorithm. In this domain, frontier search eliminates the space constraint, increasing the size of problems that could be solved before memory is exhausted from grids of length 7500 on a side, to over 12 million nodes on a side in principle, making time the limiting resource.

The next domain was the well-known sliding-tile puzzles. We completed exhaustive breadth-first frontier searches of all sliding-tile puzzles up to the Fifteen puzzles, which contain over 10^{13} nodes. On the 2×8 Fifteen Puzzle, frontier search saves a factor of over 28 in space, compared to standard breadth-first search. We also applied frontier-A* with the Manhattan distance heuristic to the 4×4 Fifteen Puzzle, reducing the memory required by 57% compared to A*, and increasing the number of problems from a standard set that could be solved from 79% to 94%.

Next we considered the four-peg Towers of Hanoi problem. We performed complete breadth-first frontier searches on problems with up to 20 disks, saving a factor of over 46 in storage on the 20-disk problem compared to standard breadth-first search. We also discovered a surprising anomaly concerning the radius of the 15 and 20 disk problems. Taking advantage of the symmetry between the initial and goal states, we were able to verify conjectured but unproven optimal solution lengths for up to 24 disks, saving a factor of over 24 in space. Previously, this had only been done for up to 17 disks. We also applied frontier-A* to the 17-disk problem, saving an order of magnitude in memory over standard A*.

Finally, we considered multiple sequence alignment, where the problem of finding an optimal alignment of d sequences can be mapped to finding a lowest-cost corner-to-corner path in a d dimensional hypercube. In this case, frontier search reduces the asymptotic space complexity from $O(n^d)$ to $O(n^{d-1})$, where n is the

length of the sequences. We showed that frontier-A* outperforms dynamic programming, the standard algorithm for this problem, in memory for three or more sequences, and in time and memory for four or five sequences.

We conclude that frontier search is a general and effective method for significantly reducing the memory required by best-first search.

ACKNOWLEDGMENTS. Thanks to Teresa Breyer for the data on the overhead of reconstructing the solution path, and to her and Alex Dow for helpful discussions concerning this research. Thanks to Hermann Kaindl for information on bidirectional search.

REFERENCES

ARAKI, S., GOSHIMA, M., MORI, S., NAKASHIMA, H., AND TOMITA, S. 1999. Application of parallelized DP and A* algorithm to multiple sequence alignment. In *Proceedings of the Genome Informatics Workshop IV*, 94–102.

BODE, J.-P., AND HINZ, A. 1999. Results and open problems on the Tower of Hanoi. In *Proceedings of the 30th Southeastern International Conference on Combinatorics, Graph Theory, and Computing* (Boca Raton, FL). Congressus Numerantium, Vol. 139. 112–122.

BRUNGGER, A., MARZETTA, A., FUKUDA, K., AND NIEVERGELT, J. 1999. The parallel search bench ZRAM and its applications. *Ann. Oper. Res. 90*, 45–63.

CHAKRABARTI, P., GHOSE, S., ACHARYA, A., AND DE SARKAR, S. 1989. Heuristic search in restricted memory. *Artif. Intell. 41*, 2 (Dec.), 197–221.

CULBERSON, J., AND SCHAEFFER, J. 1998. Pattern databases. *Computat. Intell. 14*, 3, 318–334.

DIJKSTRA, E. 1959. A note on two problems in connexion with graphs. *Numer. Math. 1*, 269–271.

DUDENEY, H. 1908. *The Canterbury Puzzles (and Other Curious Problems)*. E.P. Dutton, New York.

DUNKEL, O. 1941. Editorial note concerning advanced problem 3918. *Amer. Math. Month. 48*, 219.

FELNER, A., KORF, R., AND HANAN, S. 2004. Additive pattern database heuristics. *J. Artif. Intell. Res. 22*, 279–318.

FELNER, A., MESHULAM, R., HOLTE, R., AND KORF, R. 2004. Compressing pattern databases. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-04)* (San Jose, CA). 638–643.

FRAME, J. 1941. Solution to advanced problem 3918. *Amer. Math. Month. 48*, 216–217.

GHOSH, S., MAHANTI, A., AND NAU, D. 1994. ITS: An efficient limited-memory heuristic tree search algorithm. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)* (Seattle, WA). 1353–1358.

HART, P., NILSSON, N., AND RAPHAEL, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cyber. SSC-4*, 2 (July), 100–107.

HIRSCHBERG, D. 1975. A linear space algorithm for computing maximal common subsequences. *Commun. ACM 18*, 6 (June), 341–343.

HIRSCHBERG, D. 1997. Serial computations of levenshtein distances. In *Pattern Matching Algorithms*, A. Apostolic and Z. Galil, Eds. Oxford University Press, Oxford, England, 123–141.

HOHWALD, H., THAYER, I., AND KORF, R. 2003. Comparing best-first search and dynamic programming for optimal multiple sequence alignment. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)* (Acapulco, Mexico). 1239–1245.

IKEDA, T., AND IMAI, H. 1999. Enhanced A* algorithms for multiple alignments: Optimal alignments for several sequences and k-opt approximate alignments for large cases. *Theoret. Comput. Sci. 210*, 341–374.

JOHNSON, W., AND STOREY, W. 1879. Notes on the 15 Puzzle. *Amer. J. Math. 2*, 397–404.

KAINDL, H., AND KAINZ, G. 1997. Bidirectional heuristic search reconsidered. *J. Artif. Intell. Res. 7*, 283–317.

KLAVZAR, S., AND MILUTINOVIC, U. 2002. Simple explicit formulas for the Frame-Stewart numbers. *Ann. Combinat. 6*, 157–167.

KORF, R. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artif. Intell. 27*, 1 (Sept.), 97–109.

KORF, R. 1993. Linear-space best-first search. *Artif. Intell. 62*, 1 (July), 41–78.

KORF, R. 1999. Divide-and-conquer bidirectional search: First results. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)* (Stockholm, Sweden). 1184–1189.

KORF, R. 2003. Delayed duplicate detection: Extended abstract. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)* (Acapulco, Mexico). 1539–1541.

KORF, R. 2004. Best-first frontier search with delayed duplicate detection. In *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-2004)* (San Jose, CA). 650–657.

KORF, R. 2005. Large-scale parallel breadth-first search. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI-2005)* (Pittsburgh, PA), 1380–1385.

KORF, R., AND CHICKERING, D. 1996. Best-first minimax search. *Artif. Intell. 84*, 1-2 (July), 299–337.

KORF, R., REID, M., AND EDELKAMP, S. 2001. Time complexity of Iterative-Deepening-A*. *Artif. Intell. 129*, 1–2 (June), 199–218.

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)* (Austin, TX). 910–916.

LERMEN, M., AND REINERT, K. 2000. The practical use of the A* algorithm for exact multiple sequence alignment. *J. Computat. Biol. 7*, 5, 655–671.

NEEDLEMAN, S., AND WUNSCH, C. 1970. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *J. Molec. Biol. 48*, 443–453.

PEARL, J. 1984. *Heuristics*. Addison-Wesley, Reading, MA.

POHL, I. 1970. Heuristic search viewed as path finding in a graph. *Artif. Intell. 1*, 193–204.

POHL, I. 1971. Bi-directional search. In *Machine Intelligence 6*, B. Meltzer and D. Michie, Eds. American Elsevier, New York, 127–140.

REINEFELD, A. 1993. Complete solution of the Eight Puzzle and the benefit of node ordering in IDA*. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI-93)* (Chambery, France). 248–253.

RUDIN, H. 1987. Network protocols and tools to help produce them. In *Annual Review of Computer Science*, J. Traub, B. Grosz, B. Lampson, and N. Nilsson, Eds. Annual Reviews Inc., Palo Alto, CA, 291–316.

RUSSELL, S. 1992. Efficient memory-bounded search methods. In *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI-92)* (Vienna, Austria). 1–5.

SCHOFIELD, P. 1967. Complete solution of the Eight Puzzle. In *Machine Intelligence 3*, B. Meltzer and D. Michie, Eds. American Elsevier, New York, 125–133.

SEN, A., AND BAGCHI, A. 1989. Fast recursive formulations for best-first search that allow controlled use of memory. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI-89)* (Detroit, MI). 297–302.

SETUBAL, J., AND MEIDANIS, J. 1997. *Introduction to Computational Molecular Biology*. PWS Publishing, Boston, MA.

SPOUGE, J. 1989. Speeding up dynamic programming algorithms for finding optimal lattice paths. *SIAM J. Appl. Math. 49*, 5, 1552–1566.

STEWART, B. 1941. Solution to advanced problem 3918. *Amer. Math. Monthly 48*, 217–219.

TAYLOR, L., AND KORF, R. 1993. Pruning duplicate nodes in depth-first search. In *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI-93)* (Washington, DC). 756–761.

THAYER, I. 2003. Methods for optimal multiple sequence alignment. M.S. dissertation. Computer Science Department, University of California, Los Angeles, CA.

THOMSON, J., PLEWNIAK, F., AND POCH, O. 1999. BAliBASE: A benchmark alignment database for the evaluation of multiple alignment programs. *Bioinformatics 15,* 1, 87–88.

UKKONEN, E. 1985. Algorithms for approximate string matching. *Info. Contr. 64*, 100–118.

YOSHIZUMI, T., MIURA, T., AND ISHIDA, T. 2000. A* with partial expansion for large branching factor problems. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-00)* (Austin, TX). 923–929.

ZHOU, R., AND HANSEN, E. 2002. Multiple sequence alignment using anytime A*. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI-02)* (Edmonton, Alberta, Canada). 975–976.

ZHOU, R., AND HANSEN, E. 2003a. Sparse-memory graph search. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)* (Acapulco, Mexico). 1259–1266.

ZHOU, R., AND HANSEN, E. 2003b. Sweep-A*: Space-efficient heuristic search in partially ordered graphs. In *Proceedings of the 15th International Conference on Tools with Artificial Intelligence (ICTAI-03)* (Sacramento, CA). 427–434.

ZHOU, R., AND HANSEN, E. 2004. Breadth-first heuristic search. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS-04)* (Whistler, British Columbia, Canada). 92–100.

RECEIVED OCTOBER 2003; REVISED MAY 2005; ACCEPTED MAY 2005