Available online at www.sciencedirect.com
SCIENCE DIRECT®
ELSEVIER
Artificial Intelligence 170 (2006) 385–408
Artificial Intelligence
www.elsevier.com/locate/artint

Breadth-first heuristic search

Rong Zhou *, Eric A. Hansen

Department of Computer Science and Engineering, Mississippi State University, Mississippi State, MS 39762, USA

Received 1 December 2004; received in revised form 13 December 2005; accepted 13 December 2005

Abstract

Recent work shows that the memory requirements of A* and related graph-search algorithms can be reduced substantially by only storing nodes that are on or near the search frontier, using special techniques to prevent node regeneration, and recovering the solution path by a divide-and-conquer technique. When this approach is used to solve graph-search problems with unit edge costs, we show that a breadth-first search strategy can be more memory-efficient than a best-first strategy. We also show that a breadth-first strategy allows a technique for preventing node regeneration that is easier to implement and can be applied more widely. The breadth-first heuristic search algorithms introduced in this paper include a memory-efficient implementation of breadth-first branch-and-bound search and a breadth-first iterative-deepening A* algorithm that is based on it. Computational results show that they outperform other systematic search algorithms in solving a range of challenging graph-search problems.
© 2005 Elsevier B.V. All rights reserved.

Keywords: Heuristic search; Memory-limited graph search; Branch-and-bound search; Planning

1. Introduction

The A* algorithm [9], and related graph-search algorithms such as Dijkstra's single-source shortest-path algorithm [4] and breadth-first search, store all of the generated nodes of a search graph in memory, using an Open list to store nodes on the search frontier and a Closed list to store already-expanded nodes. As is well known, this creates a memory bottleneck that severely limits the scalability of these graph-search algorithms.

Storing all generated nodes serves two purposes. First, it allows states that have been reached along one path to be recognized if they are reached along another path, in order to prevent generation of duplicate nodes that represent the same state; this is called *duplicate detection*. Second, it allows the solution path to be reconstructed after completion of the search by the *traceback method*; each node stores a pointer to its parent node along the best path, and the solution path is recovered by tracing the pointers backwards from the goal node to the start node. Storing all generated nodes makes it possible to perform both of these functions. But it is not necessary to store all generated nodes in order to perform just one of these functions. This leads to two different strategies for reducing the memory requirements of A*: one strategy gives up duplicate detection and the other gives up the traceback method of solution reconstruction.

* Corresponding author.
E-mail addresses: rzhou@cse.msstate.edu (R. Zhou), hansen@cse.msstate.edu (E.A. Hansen).

0004-3702/$ – see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.artint.2005.12.002

Linear-space variants of A* such as recursive best-first search (RBFS) [14] and depth-first iterative-deepening A* (DFIDA*) [13] give up duplicate detection. Instead of storing Open and Closed lists, they use a stack to organize the search. Since the current best solution path is stored on the stack, solution reconstruction by the traceback method is straightforward. Because they only store nodes on the current path, however, these algorithms are severely limited in their ability to recognize when newly-generated nodes represent states that have already been reached by a different path. Essentially, linear-space search algorithms convert graph-search problems into tree-search problems. As the depth of the search increases, the number of node regenerations relative to the number of distinct nodes (i.e., the size of the search tree relative to the size of the search graph) can increase exponentially. For complex graph-search problems in which the same state can be reached along many different paths, DFIDA* and RBFS perform very poorly due to excessive node regenerations. Their performance can be improved by using available memory to store as many generated nodes as possible in order to check for duplicates [21,24]. But this requires as much memory as A* in order to eliminate *all* duplicate search effort.

A second strategy for reducing the memory requirements of A* prevents duplicate search effort, but does not use the traceback method of solution reconstruction. It is based on the development of techniques for duplicate detection that do not require storing all generated nodes, but only nodes that are on or near the search frontier. This strategy was introduced to the artificial intelligence community in the form of an algorithm called *frontier search*, which only stores the Open list and saves memory by not storing the Closed list [15,18,19]. A closely-related algorithm called *sparse-memory graph search* stores only a small part of the Closed list [26]. Instead of the traceback method, both algorithms use a divide-and-conquer technique to recover the solution path. In this technique, the search algorithm finds an intermediate node along an optimal path and uses it to divide the original problem into two subproblems – the problem of finding an optimal path from the start node to the intermediate node, and the problem of finding an optimal path from the intermediate node to the goal node. The subproblems are solved recursively by the same search algorithm until all nodes along an optimal solution path for the original problem are identified.

The contribution of this paper is to show that when this second strategy for reduced-memory graph search is adopted, a breadth-first branch-and-bound search algorithm can be more memory-efficient than A*. This appears counter-intuitive because breadth-first branch-and-bound typically requires more node expansions than A*. However, we show that a breadth-first search strategy can reduce the number of nodes that need to be stored in memory in order to perform duplicate detection, and this allows larger problems to be solved in the same amount of memory. In addition, a breadth-first strategy allows a technique for duplicate detection that is easier to implement and can be applied more widely.

The paper is organized as follows. Section 2 reviews previous work on A* search using this divide-and-conquer strategy for reducing memory requirements. Section 3 describes how the same approach can be used to reduce the memory requirements of breadth-first branch-and-bound search. Section 4 uses this memory-efficient breadth-first branch-and-bound algorithm to create a breadth-first iterative-deepening A* algorithm. Section 5 reports computational results that show the advantages of the breadth-first approach in solving the Fifteen Puzzle problem, the 4-peg Towers of Hanoi problem, and STRIPS planning problems. Potential extensions of this work are discussed in Section 6.

We focus on graph-search problems with unit (or uniform) edge costs, which are best-suited for breadth-first search. In Section 6.1, we briefly discuss how to extend the breadth-first approach to find optimal solutions to graph-search problems with non-uniform edge costs. However, we leave the details of this extension for future work.

2. Background

2.1. Graph-search preliminaries

State-space search is a widely-used problem-solving framework in artificial intelligence. A state space consists of a set of states and a set of operators for transforming one state into another. A state space can be represented by a weighted graph $G = (N, E, c)$ in which each node $n \in N$ corresponds to a problem state, each edge $(n, n') \in E$ corresponds to a state transition, and a cost function $c : E \to \Re^+$ associates a non-negative cost $c(n, n')$ with each transition (n, n'). A *problem instance* is a state-space graph with a start node, $s \in N$, and a set of goal (or terminal) nodes, $T \subseteq N$. A solution is a path from the start node s to a goal node $t \in T$. An optimal solution is a minimum-cost

path, where the cost of a path is the sum of the costs of its edges. In this paper, we assume that all edges have unit cost. In this case, a minimum-cost path is also a shortest path from the start node to a goal node.

Typically, a graph is represented implicitly by a procedure for generating the successors of a node, which is referred to as *expanding* the node. This procedure allows parts of the graph to be generated and stored in memory, possibly deleted from memory, and then possibly regenerated later, in the course of the search for a solution path. The set of nodes and edges that is stored in memory at any time is referred to as the *explicit graph*, in contrast to the entire graph, which is called the *implicit graph*. Implicit representation of a graph is useful since the entire graph does not usually need to be stored in memory, or even generated, in order to solve a particular problem instance.

There are two general approaches to systematic graph search. One relies on a stack to organize the search and determine the next node to expand; it includes depth-first search, depth-first branch-and-bound search, depth-first iterative-deepening A* [13], and recursive best-first search [14]. The other relies on a priority queue to organize the search and determine the next node to expand; it includes breadth-first search, Dijkstra's single-source shortest-path algorithm [4], and the best-first search algorithm A* [9]. The priority queue is called an Open list and contains the frontier nodes of the explicit graph, that is, the nodes that have been generated but not yet expanded. In breadth-first search, the Open list is ordered by the depth of each node n; in Dijkstra's single-source shortest-path algorithm, it is ordered by the cost of a best path from the start node to each node n, denoted $g(n)$; in A*, or best-first search, it is ordered by an estimate of the cost of the best solution path that goes through each node n, denoted $f(n) = g(n) + h(n)$, where $h(n)$ is an estimate of the cost of a best path from node n to a goal node.

If $h(n)$ never over-estimates the cost of a best path from node n to a goal node, for any node n, it is said to be an *admissible heuristic* (or equivalently, a lower-bound function). Using an admissible heuristic, A* is guaranteed to find minimum-cost paths. A heuristic is said to be *consistent* if $h(n) \leqslant c(n, n') + h(n')$ for all n and n'. Consistency implies admissibility. Using a consistent heuristic, A* is guaranteed to expand the fewest nodes of any algorithm for finding a minimum-cost solution that uses the same heuristic, up to ties [3]. Breadth-first search and Dijkstra's single-source shortest-path algorithm are also guaranteed to find optimal solution paths, under appropriate assumptions.

In graph search, there are multiple paths to a node, and in order to avoid duplicate search effort, it is important to recognize when an already generated node has been reached along a different path; this is called duplicate detection. The traditional approach to duplicate detection is to store all generated nodes in memory, typically in a hash table, and to check these stored nodes before adding a new node to the explicit graph. This prevents the search algorithm from storing more than one node that represents the same problem state. If the search algorithm finds another path to a node with a lower g-cost, it updates the cost information associated with the node already in the explicit graph, as well as the pointer to its parent node along a best path. When A* uses a consistent heuristic, the g-cost of a node is guaranteed to be optimal once the node is expanded.

For graph-search algorithms that use an Open list to store frontier nodes that have been generated but not yet expanded, nodes that have been expanded are stored in a Closed list. The memory needed to store all open and closed nodes is the bottleneck of these algorithms. Although some depth-first search algorithms can find an optimal solution without storing all generated nodes, their inability to detect duplicates typically leads to an exponential increase in time complexity that can make their performance unacceptably slow in practice. In the rest of this paper, we consider an approach to reducing the memory requirements of graph search that does not give up duplicate detection.

2.2. Reduced-memory graph search

The strategy for reduced-memory graph search that we adopt in this paper was introduced to the AI heuristic search community by Korf [15]. (A similar, although less general, strategy was previously used to reduce the memory requirements of dynamic-programming algorithms for sequence comparison [12,22].) Korf and Zhang [18] describe a version of A* that uses this strategy. Korf also uses this strategy in Dijkstra's single-source shortest-path algorithm [15], bidirectional A* [15], and (uninformed) breadth-first search [16]. Zhou and Hansen [26,27] introduce enhancements.

The strategy is based on the insight that it is not necessary to store all expanded nodes in a Closed list in order to perform duplicate detection. Often, duplicate detection only requires storing nodes that are on or near the search frontier. The intuition is that the set of generated nodes forms a "volume" that encompasses the start node and grows outward as the search frontier is expanded. If the graph has a particular structure, or if special techniques are used by

the search algorithm, nodes on the frontier cannot regenerate nodes in the interior of the search volume. Therefore interior nodes do not need to be stored in memory in order to perform duplicate detection.

If nodes in the interior of the search volume are removed from memory, however, this prevents recovery of the solution path by the traditional traceback method. Therefore search algorithms that use this memory-saving technique rely on a divide-and-conquer technique of solution recovery. Each node n stores information about an intermediate node along the best path to n from the start node. It could be a node in the middle of the search space, but does not have to be in the middle. (Whether a node is in the middle of the search space, or at some other position, can be estimated in various ways. For example, its position can be estimated by comparing its g-value to its f-value, or by comparing its depth to an estimate of solution length.) Once the search problem is solved, information about this intermediate node is used to divide the search problem into two subproblems: the problem of finding an optimal path from the start node to the intermediate node, and the problem of finding an optimal path from the intermediate node to the goal node. Each of these subproblems is solved by the same search algorithm in order to find an intermediate node along their optimal paths. The process continues recursively until primitive subproblems (in which the optimal path consists of a single edge) are reached, and all nodes on an optimal solution path for the original search problem have been identified. The time it takes to solve all of these subproblems is usually very short compared to the time it takes to solve the original search problem, and this approach can save a substantial amount of memory in exchange for very limited time overhead for solution recovery.

Search algorithms that use this memory-reduction strategy differ in detail. We briefly review some of the differences by considering, in turn, the two key issues in this strategy: duplicate detection and divide-and-conquer solution reconstruction.

2.2.1. Duplicate detection

Korf et al. [15,18,19] describe a memory-efficient approach to graph search called *divide-and-conquer frontier search* or simply *frontier search*. Frontier search only stores nodes on the search frontier (the Open List) and not nodes in the search interior (the Closed list). Closed nodes are prevented from being regenerated by storing a list of *used-operator bits* in each open node. The list has one bit for each operator; the bit indicates whether the neighboring node reached by that operator has already been expanded. Each time a node is expanded, only unused legal operators are used to generate successor nodes. If a newly-generated node could have the just-expanded node as a successor (as in undirected graphs), a used-operator bit in the successor node is set to block later regeneration of the just-expanded node. When a node is generated that is a duplicate of a node already stored in the Open list, the operators marked in the saved node are the union of the used operators of the individual nodes. In undirected graphs, used-operator bits are sufficient to prevent regeneration of already-closed nodes. In directed graphs in which a node can have predecessors that are not also potential successors, an additional technique must be used. Each time a node is expanded, frontier search not only generates its successor nodes, it generates its predecessor nodes to which a legal path has not yet been found (i.e., they have not yet been inserted in the Open list). Such nodes are assigned an infinite f-cost to prevent them from being expanded until a legal path is found. Note that these *dummy nodes* acquire an actual, finite cost once a path to them is found.

Zhou and Hansen [26] describe a closely-related approach to reducing the memory requirements of graph search, called *sparse-memory graph search*, that prevents node regeneration without used-operator bits or dummy nodes. Instead, each node has a counter (called a *predecessor counter*) that is initially set to the number of predecessors of that node in the implicit graph. Each time a node is expanded, the counter of each of its successor nodes is decremented by one. Unlike frontier search, sparse-memory graph search uses a Closed list. However, closed nodes can be removed from memory once their counter is equal to zero, since this ensures they cannot be regenerated. Although this technique delays removal of closed nodes from memory, which is a disadvantage compared to frontier search, it has advantages that make up for this. One advantage is that it allows use of an upper bound on the cost of an optimal solution to prune any open node with an f-cost greater than the bound. Such nodes will never be expanded and memory is saved by not storing them. By contrast, frontier-A* cannot remove sub-optimal nodes from the Open list because doing so discards used-operator bits; if it did so and the same node was later regenerated via a shorter path, loss of the used-operator bits could allow already-closed nodes to be regenerated.

Sparse-memory graph search has some other advantages over frontier search. For problems with a very high branching factor, storing a counter in each node takes much less memory than storing used-operator bits, one for each operator. Moreover, in directed graphs, the sparse-memory approach does not generate dummy nodes. On the other

hand, frontier search has some advantages over sparse-memory graph search; in particular, it saves space by not storing *any* closed nodes. In practice, whether frontier search or sparse-memory graph search is more memory-efficient is problem-dependent.

2.2.2. Divide-and-conquer solution reconstruction

In Korf et al.'s [18,19] implementation of frontier search, each node past the middle of the search space stores (via propagation from its parent node) all state information about a node along the best path to it that is about halfway between the start and goal nodes. After a goal node is expanded, the midpoint node identified by this information is used as a pivot point for divide-and-conquer solution recovery.

In sparse-memory graph search, Zhou and Hansen [26] use a different technique for keeping track of an intermediate node along a best path. Each node stores a pointer to the intermediate node, which is called a *relay node*, and these relay nodes are saved in memory. Although the count of nodes in memory is increased by storing relay nodes, overall memory use can be decreased because the node data structure is smaller; it contains a pointer instead of all state information about an intermediate node. In fact, this is the same node data structure used by A*, since the pointer stored in each node could be a pointer to either a parent node or a relay node. If enough memory is available to solve a search problem (or subproblem) using A*, this makes it possible to avoid the overhead of divide-and-conquer solution reconstruction by retaining all nodes in memory, letting each node store a pointer to its parent, and using the traceback method of solution recovery. The technique of pointers and relay nodes can also be used in frontier search, and usually improves its performance. In the rest of this paper, all of the algorithms described will use pointers and relay nodes for divide-and-conquer solution reconstruction.

3. Memory-efficient breadth-first branch-and-bound search

The frameworks of frontier search and sparse-memory graph search have been used to implement A* in a memory-efficient way. Because both frontier-A* and sparse-memory A* are best-first search algorithms, their memory requirements depend on the size of a best-first frontier.

In this section, we consider the potential advantages of a breadth-first approach to heuristic search that adopts a similar strategy for reducing memory requirements. First we consider the possibility that a breadth-first frontier could be smaller than a best-first frontier, requiring less memory for duplicate detection. Then we discuss other potential advantages. In particular, for problems where frontier search and sparse-memory graph search are too complex to implement, we show that breadth-first search allows a simpler method of duplicate detection.

By *breadth-first heuristic search*, we mean a search algorithm that expands nodes in breadth-first order, but uses the same heuristic information as A* in order to prune the search space. For any node n, a lower-bound estimate of the cost of an optimal path through the node is given by the node evaluation function $f(n) = g(n) + h(n)$, where $h(n)$ is an admissible heuristic; this is the same node evaluation function used by A*. In breadth-first heuristic search, no node is inserted into the Open list if its f-cost is greater than an upper bound on the cost of an optimal solution, since such nodes cannot be on an optimal path. We discuss how to obtain an upper bound later.

This simple search algorithm, which combines breadth-first search with pruning using upper and lower bounds, is *breadth-first branch-and-bound search* (abbreviated BFBnB). This search strategy has been rarely used in practice because the number of nodes it expands is at least as great, and usually greater, than the number of nodes expanded by the best-first search algorithm A*. Given a perfect upper bound, BFBnB expands the same nodes as A*, disregarding ties. (For a discussion of tie-breaking, see Section 4.2.) If the upper bound is not perfect, BFBnB expands more nodes than A*. Therefore, if all expanded nodes are stored in memory, breadth-first branch-and-bound search uses as much *or more* memory than A*, and has no advantage.

However we propose a breadth-first branch-and-bound algorithm that uses the same divide-and-conquer approach to memory reduction reviewed in the previous section. Its memory requirements depend on the number of nodes needed to store the search frontier and perform duplicate detection, and not the total number of nodes expanded. As we will show, a breadth-first branch-and-bound search algorithm can have a smaller search frontier than a best-first search algorithm like A*, giving it an advantage in terms of memory efficiency.

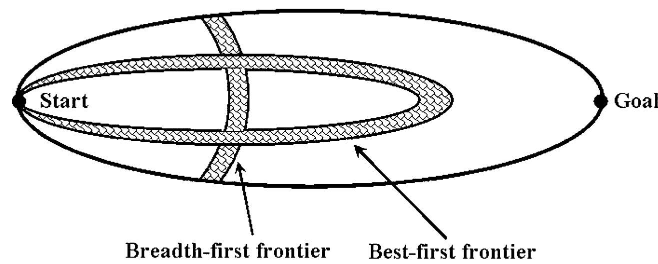

Fig. 1. Comparison of best-first and breadth-first search frontiers. The outer ellipse encloses all nodes with f-cost less than or equal to an (optimal) upper bound.

3.1. Relative size of breadth-first and best-first frontiers

Fig. 1 illustrates, in an intuitive way, how breadth-first branch-and-bound search could result in a smaller search frontier. It shows that best-first node expansion "stretches out" the search frontier, whereas breadth-first search does not, and uses bounds to limit its width. Even if breadth-first branch-and-bound search expands more nodes than best-first search, its smaller search frontier could result in more memory-efficient search and improved scalability. We do not claim this always happens. But results reported in Section 5 provide evidence that it often happens.

The intuition conveyed by this picture can be expressed in a somewhat more formal argument, although not a proof. Just as each layer of a breadth-first search graph is associated with a different g-cost, we can imagine that a best-first search graph is divided into layers, with each layer corresponding to a different f-cost. Note that the size of any layer is a lower bound on the size of the frontier. We can compare the average layer size of breadth-first heuristic search with that of best-first heuristic search. The average layer size is defined as the total number of nodes expanded divided by the number of layers in the search graph. Given an optimal upper bound, the number of nodes expanded by BFBnB and A* is the same, disregarding ties. Therefore, the average layer size is inversely proportional to the number of layers. In other words, the more layers, the smaller the layers, on average. For BFBnB, the number of layers (or equivalently, the depth of the search) is $f^* + 1$, where f^* is the length of the shortest solution path. For A* (using a consistent heuristic function), the number of layers (or equivalently, the number of distinct f-costs) is $f^* - h(\mathit{start}) + 1$, where $h(\mathit{start})$ is the heuristic estimate of the start node. Thus, the average layer size of breadth-first heuristic search is always less than for best-first search. For example, if $h(\mathit{start}) = \frac{1}{2} f^*$, then a layer of breadth-first heuristic search is, on average, half the size of a layer of best-first search. As the heuristic function becomes more accurate, the value of $h(\mathit{start})$ increases, and this increases the relative size of an average best-first layer compared to an average breadth-first layer. For example, suppose the value of $h(\mathit{start})$ is 90% of f^*. Then a layer of breadth-first heuristic search is ten times smaller than a layer of best-first search, on average.

Because this reasoning considers the average size of a layer, it does not prove that the peak memory requirements of breadth-first heuristic search are less than those of a best-first strategy. The layers of a search graph are of different sizes, and peak memory requirements depend on the size of the largest layer (or largest adjacent layers), and not on the average layer size. Nevertheless, it gives us reason to believe that the size of a breadth-first layer/frontier can often be smaller than the size of a best-first layer/frontier.

3.2. Frontier breadth-first branch-and-bound search

Frontier search can be combined with other graph-search algorithms besides A*. It has previously been combined with Dijkstra's single-source shortest-path algorithm [15] and with (uninformed) breadth-first search [16]. We now discuss how to combine frontier search with BFBnB.

In one very important way, frontier search can be more memory-efficient in combination with BFBnB than in combination with A*. Given an upper bound on the cost of an optimal solution, any node with an f-cost greater than this bound cannot be part of an optimal path and will never be expanded. We already pointed out that pruning these nodes from the Open list can save a lot of memory, and this can be done in both A* and sparse-memory A*. But for reasons given in Section 2.2.1, frontier-A* cannot prune nodes from the Open list without allowing duplicate nodes to be generated. When frontier search is combined with BFBnB, however, and the search graph has uniform edge

costs, then using an upper bound to prune the Open list does not affect duplicate detection. (This is because BFBnB generates nodes in order of g-cost, and after a node is generated, a better path to it cannot be found.) Because using bounds to prune the Open list can significantly reduce memory use, this is an important advantage of frontier-BFBnB over frontier-A*.

Use of breadth-first search can improve efficiency in other ways. Instead of storing the g-cost of each node, a single g-cost is stored for each layer of the search graph, since every node in the same layer has the same g-cost. This simplifies the node data structure. Because all nodes in the same layer have the same g-cost, the Open list does not need to be sorted, and this allows a simpler data structure for both the node and the Open list. A first-in-first-out queue, implemented as a circular buffer, can be used, instead of a priority queue, which is commonly used to implement the Open list of A*. Use of a circular buffer not only saves memory, it makes dynamic memory allocation easier. In addition, breadth-first search usually has better cache performance than best-first search, since expanding nodes from the front of a first-in-first-out queue and appending newly-generated nodes to the end usually leads to better memory-access locality than selecting nodes for expansion from a repeatedly-sorted Open list.

3.3. Sparse-memory breadth-first branch-and-bound search

Combining sparse-memory graph search with breadth-first branch-and-bound search is also straightforward. As already mentioned, an advantage of sparse-memory A* over frontier-A* is that the former can use an upper bound to prune the Open list, and the latter cannot. But since frontier-BFBnB can use an upper bound to prune the Open list, sparse-memory BFBnB does not have this advantage over frontier-BFBnB. For high-branching factor problems, however, where storing a predecessor counter requires less memory than storing used-operator bits, or in directed graphs, where frontier search generates dummy nodes, sparse-memory BFBnB may still have an advantage over frontier-BFBnB.

3.4. Layered duplicate detection

Although breadth-first branch-and-bound search can be combined with frontier search or sparse-memory graph search, an important advantage of the breadth-first approach is that it also allows a much simpler method of duplicate detection which we call *layered duplicate detection*. Because it is simpler, this method of duplicate detection can be used in solving search problems for which frontier search and sparse-memory graph search may be too complex to implement.

A breadth-first search graph divides into layers, one for each depth, and all nodes in one layer are expanded before considering nodes in the next layer. In fact, the Open and Closed lists of a breadth-first search algorithm can be considered to have a layered structure, where $Open_\ell$ denotes the set of open nodes in layer ℓ and $Closed_\ell$ denotes the set of closed nodes in layer ℓ. As a result, we sometimes refer to the Open or Closed list of a particular layer, as if each layer has its own Open and Closed lists. At any time, all open nodes are in the current layer or its successor layer, whereas closed nodes can be in the current layer or any previous layer.

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

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

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

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

The locality of a graph is a non-negative integer that determines how many closed layers of a breadth-first search graph need to be stored in order to completely prevent duplicate nodes. Note that $g^*(n)$ can never be less than $g^*(n')$

by more than one. But in general, there is no a priori limit on how much greater $g^*(n)$ can be than $g^*(n')$. In other words, the shortest path to a node n may be arbitrarily longer than the shortest path to its successor node n'.

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

Proof. First assume that k, the number of previous layers saved in memory, is less than the locality of the graph. To see how this allows regeneration of a node, consider nodes n and n' such that $g^*(n) - g^*(n') > k$. When node n is expanded, its successor n' is either in the previous k layers or not. If it is not, it is regenerated. If it is, it has been previously regenerated since it was first generated more than k layers before. In either case, there is a duplicate node.

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

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

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

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

In graphs with a locality of one, such as undirected graphs, the number of layers that a breadth-first algorithm must keep in memory for the purpose of duplicate detection is three; the previous layer, the currently-expanding layer, and the next layer. In general, the number of layers that need to be retained for duplicate detection is equal to the locality of the graph plus two. (To allow divide-and-conquer solution reconstruction, an additional layer of relay nodes may also be retained in memory.)

Frontier search uses used-operator bits to prevent regeneration of closed nodes. It is easy to see that use of used-operator bits (without dummy nodes) has the same effect as storing one previous layer of the breadth-first search graph, since blocking a node from regenerating a predecessor has the same effect as storing the previous layer and checking for duplicates. But in graphs with locality greater than one, used-operator bits alone are not sufficient for duplicate detection. In this case, breadth-first search provides an alternative method of duplicate detection that is simpler than creating dummy nodes: store more than one previous layer of the search graph.

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

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

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

Proof. Let $\Delta \geqslant 2$ be the total number of layers saved by the algorithm. Obviously, no duplicate nodes can exist in these Δ layers, because the algorithm always checks for duplicates in all saved layers before inserting any newly-generated node into the Open list for the next layer. Therefore, the earliest time for a node n to be regenerated is

$g^*(n) + \Delta$ and the earliest time for the same node to be regenerated twice is $g^*(n) + 2\Delta$, and so on. Since the total number of layers is bounded by the length of the shortest solution path (f^*), the number of times a node n is regenerated cannot exceed the bound stated in the theorem. □

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

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

If only one previous layer of the search graph is stored, this means that any node in the immediate previous layer whose f-cost is greater than or equal to the upper bound minus one can be removed from memory. (This optimization is not included in the pseudocode of Fig. 2, but is included in our implementation.)

Note that an upper bound may or may not be associated with an actual solution. Theorem 3 does not assume it is. But if a solution has been found with the same cost as the upper bound, breadth-first heuristic search can prune the Closed list even more aggressively than Theorem 3 allows by removing nodes with an f-cost greater than or equal to the upper bound minus $(k + 1)$ in the kth previous layer. This follows because the search algorithm only needs to find a solution if the currently available solution is not already optimal. By similar reasoning, BFBnB can prune the Open list more aggressively by not inserting any node with an f-cost greater than *or equal to* an upper bound, when the upper bound is the cost of a solution that has already been found.

An interesting question is whether layered duplicate detection can also be used in best-first search. As suggested in Section 3.1, we can consider a best-first search graph to be divided into layers, one for each f-cost. However, use of layered duplicate detection in best-first search seems impractical because best-first search makes it difficult to bound the number of previous layers, or the number of successor layers, that need to be retained in memory to perform duplicate detection. It is difficult to bound the number of previous layers because it is difficult to determine the locality of a best-first search graph (i.e., the maximum difference between the f-cost of a node when it is expanded and the f-cost of any of its successor nodes that have been previously expanded), at least without making special assumptions about the heuristic function. It is difficult to bound the number of successor layers because, in general, there is no a priori limit on how much greater the f-cost of a node can be than the f-cost of its predecessor. There may be some special cases in which layered duplicate detection could be useful in best-first search. But in general, it seems impractical.

3.5. Breadth-first branch-and-bound search with layered duplicate detection

Fig. 2 gives the pseudocode of a breadth-first branch-and-bound search algorithm that uses layered duplicate detection and divide-and-conquer solution reconstruction. The main algorithm, *BFBnB*, differs from A* in the following ways; the Open and Closed lists are indexed by layers, previous layers are deleted to recover memory, and the solution is reconstructed by the divide-and-conquer method once a goal node is *generated*. (In breadth-first search, one can be sure a solution is optimal as soon as a goal node is generated, without waiting for the goal node to be selected from the Open list for expansion.)

The function *ExpandNode* works in the usual way, with two differences. First, it uses an upper bound U to prune nodes that cannot be on an optimal path. (Note that for subproblems solved during divide-and-conquer solution reconstruction, the upper bound is optimal since the optimal cost of the overall solution is determined before beginning solution reconstruction.) Second, it sets the ancestor pointer in each node to an intermediate node along an optimal path, called a relay node, in order to allow divide-and-conquer solution reconstruction. For simplicity, the pseudocode stores all relay nodes in a single intermediate layer of the breadth-first search graph (called a *relay layer*), which is approximately in the middle of the search graph. Only nodes that come after the relay layer have a pointer to a relay node. Nodes that come before have a pointer to the start node.

```
Function ExpandNode (Node n, start, goal; Integer ℓ, relay, U)
1  for each n′ ∈ Successors(n) do
2     if g(n) + 1 + h(n′) > U continue     /* prune sub-optimal node */
3     if n′ ∈ Closed_{ℓ−1} ∪ Closed_ℓ ∪ Open_ℓ ∪ Open_{ℓ+1} continue     /* duplicate */
4     g(n′) ← g(n) + 1
5     if ℓ < relay then
6        ancestor(n′) ← start
7     else if ℓ = relay then
8        ancestor(n′) ← n
9     else     /* ℓ > relay */
10       ancestor(n′) ← ancestor(n)
11    if n′ is goal then return n′
12    Open_{ℓ+1} ← Open_{ℓ+1} ∪ {n′}
13 return nil

Algorithm BFBnB (Node start, goal; Integer U)     /* U = upper bound */
14 g(start) ← 0, ancestor(start) ← nil
15 Open_0 ← {start}, Open_1 ← ∅, Closed_0 ← ∅
16 ℓ ← 0 /* ℓ = index of layer */
17 relay ← ⌊U/2⌋ /* relay = index of relay layer */
18 while Open_ℓ ≠ ∅ or Open_{ℓ+1} ≠ ∅ do
19    while Open_ℓ ≠ ∅ do
20       n ← arg min_h {h(n) | n ∈ Open_ℓ}     /* (optional) tie-breaking rule */
21       Open_ℓ ← Open_ℓ \ {n}, Closed_ℓ ← Closed_ℓ ∪ {n}
22       sol ← ExpandNode (n, start, goal, ℓ, relay, U)
23       if sol ≠ nil then     /* solution reconstruction */
24          middle ← ancestor(sol)
25          if g(middle) = 1 then     /* recursion ends */
26             π_0 ← ⟨start, middle⟩
27          else
28             π_0 ← BFBnB(start, middle, g(middle))
29          if g(sol) − g(middle) = 1 then     /* recursion ends */
30             π_1 ← ⟨middle, sol⟩
31          else
32             π_1 ← BFBnB(middle, sol, g(sol) − g(middle))
33          return Concatenate (π_0, π_1)
34    if 1 < ℓ ⩽ relay or ℓ > relay + 1 then
35       for each n ∈ Closed_{ℓ−1} do     /* delete previous layer */
36          Closed_{ℓ−1} ← Closed_{ℓ−1} \ {n}
37          delete n
38    ℓ ← ℓ + 1     /* move on to next layer */
39    Open_{ℓ+1} ← ∅, Closed_ℓ ← ∅
40 return ∅
```

Fig. 2. Pseudocode for breadth-first branch-and-bound search with layered duplicate detection and divide-and-conquer solution reconstruction. For simplicity, this pseudocode does not include several optimizations described in the text.

The divide-and-conquer method can be implemented in a more sophisticated and efficient way than presented in the pseudocode. An important observation about breadth-first branch-and-bound search is that the middle layers of the search graph are typically the largest, and often orders of magnitude larger than the other layers. (For an example, see Fig. 3.) The reason for this is that layers close to the start node are small due to reachability constraints, and layers close to the goal node are small due to tightness of the bounds. Therefore, storing a layer of relay nodes in the middle of the search graph will cause the search algorithm to use more memory than necessary. We have found that it is more memory-efficient to save a relay layer at the $3/4$ point in the graph. This reduces the peak memory requirements of the algorithm, and in practice, increases the time overhead of solution reconstruction by an insignificant amount.

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

In an implementation of BFBnB that uses layered duplicate detection, we can also improve the efficiency of divide-and-conquer solution reconstruction by taking advantage of the fact that the deepest three layers of the breadth-first search graph are guaranteed to be retained in memory. Thus, in each level of divide-and-conquer recursion, instead of solving the subproblem that corresponds to finding an optimal path from the intermediate node to the goal node, we only need to find an optimal path from the intermediate node to the grandparent of the goal node, and the complete optimal path can be obtained by appending the parent of the goal node and the goal node itself to the end of the solution path for this subproblem. This improvement is incorporated in our implementation, but not in the pseudocode in Fig. 2.

4. Breadth-first iterative-deepening A*

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

An upper bound can be obtained by finding an approximate solution to the search problem. Many different search methods can be used to find approximate solutions quickly, including weighted A* [23] and beam search [28]. For search problems with many close-to-optimal solutions, a good upper bound can often be found very quickly. For search problems in which solutions are sparse, a close-to-optimal solution and corresponding upper bound may not be easy to find.

Instead of computing an upper bound using an approximate search method, it is possible to use an iterative-deepening strategy that does not expand nodes with an f-cost greater than a hypothetical upper bound, and gradually increases the hypothetical upper bound until an optimal solution is found. This is the strategy used in depth-first iterative-deepening A* (DFIDA*) [13]. We can also use it to create a breadth-first iterative-deepening A* (BFIDA*) algorithm. The algorithm begins by performing breadth-first branch-and-bound search using the f-cost of the start node as an upper bound. If no solution is found, the upper bound is increased by one (or else set to the least f-cost of any unexpanded node from the previous iteration) and BFBnB is repeated. This continues until a solution is found, which is guaranteed to be optimal. Both DFIDA* and BFIDA* use an identical iterative-deepening strategy. The only difference is that DFIDA* uses depth-first branch-and-bound search (DFBnB), whereas BFIDA* uses a memory-efficient implementation of BFBnB.

The relative performance of DFIDA* and BFIDA* depends on the structure of the search space. In a search tree, or in a search space that can be closely approximated by a tree, DFIDA* is more effective because duplicate detection is not critical and node-generation overhead is less in DFIDA* than in BFIDA*. But in a search graph with many duplicate paths, BFIDA* can be more effective.

Whether BFIDA* is more effective than BFBnB depends on whether a close-to-optimal solution and corresponding tight upper bound can be found quickly for use by BFBnB. If so, BFBnB may be faster than BFIDA* because BFIDA* performs multiple iterations of BFBnB and this takes extra time. But when a close-to-optimal solution is not easy to compute, BFIDA* can be more effective than BFBnB because it avoids generating and storing many nodes that would be considered by BFBnB if it did not have a tight upper bound. In this respect, there is a symmetry between breadth-first and depth-first search, since the relative effectiveness of DFBnB and DFIDA* is problem-dependent in a similar way.

4.1. Asymptotic optimality

Korf [13] proves that DFIDA* expands the same number of nodes, asymptotically, as A*, assuming the search space is a tree and the number of nodes expanded increases exponentially in each iteration of the algorithm. Given

that DFIDA* expands the same number of nodes, asymptotically, as A*, it follows that DFIDA* is asymptotically optimal with respect to the number of node expansions, since A* is known to be optimal in terms of the number of node expansions, disregarding ties, under reasonable assumptions [3].

We now show that BFIDA* expands the same number of nodes, asymptotically, as A*, under the same assumptions – but *in graphs*, not just trees. From this, it follows that BFIDA* is asymptotically optimal with respect to the number of node expansions *in graphs*. We first prove this for undirected graphs and then for directed graphs. It is possible for us to prove this for graphs and not just trees because BFIDA* eliminates all duplicates in undirected graphs, and bounds the number of duplicates in directed graphs. Like Korf's result for DFIDA*, our theorems rely on the assumption that the number of nodes expanded increases exponentially in each iteration of the algorithm. This is a stronger assumption in our case because we consider graph search, and assume the number of *distinct* nodes expanded increases exponentially each iteration. But the result is also stronger because it compares our iterative-deepening algorithm to A* graph search, and not just A* tree search.

Theorem 4. *BFIDA* is asymptotically optimal with respect to the number of node expansions in undirected graphs, if b^i new nodes are expanded in the ith iteration, where b is a constant greater than one.*

Proof. Let d be the depth of the shallowest goal and let N be the number of nodes expanded by A*. Since b^i new nodes are expanded in the ith iteration, the total number of node expansions in the dth (last) iteration is

$$1 + b + b^2 + \cdots + b^d = \left(b^{d+1} - 1\right)/(b-1) = b^d(1-1/b)^{-1} \quad \left(b^d \gg 1\right).$$

Since BFIDA* only expands nodes whose f-cost is less than or equal to d (the optimal upper bound) in the last iteration and never regenerates (or reexpands) a node in the same iteration for undirected graphs, it follows that the number of nodes expanded by BFIDA* in the last iteration is N, which is the same as the number of nodes expanded by A*, disregarding ties. Thus, we have $b^d(1-1/b)^{-1} = N$.

The total number of nodes expanded by BFIDA* is

$$\begin{aligned}
\sum_{i=1}^{d} 1 + b + \cdots + b^i &= b^d + 2b^{d-1} + 3b^{d-2} + \cdots + db \\
&= b^d\left(1 + 2b^{-1} + 3b^{-2} + \cdots + db^{1-d}\right) \\
&\leqslant b^d\left(1 + 2b^{-1} + 3b^{-2} + \cdots + db^{1-d} + \cdots\right) \\
&= b^d(1-1/b)^{-2} \\
&= N(1-1/b)^{-1} \quad \left(b^d(1-1/b)^{-1} = N\right) \\
&= \mathrm{O}(N).
\end{aligned}$$

According to this analysis, the time complexity of BFIDA* is determined by the last iteration. Thus asymptotically speaking, BFIDA* expands as many nodes as A*. Since A* is optimal with respect to the number of node expansions [3], BFIDA* must be asymptotically optimal in the same respect. □

In directed graphs, we cannot guarantee (in general) that all duplicates will be eliminated, unless all generated nodes are stored in memory (since the graph may not have any locality). But using layered duplicate detection, we can bound the number of duplicates by storing enough layers that the ratio between f^* and the number of stored layers is a constant. This lets us prove that BFIDA* is also asymptotically optimal with respect to the number of nodes expanded in directed graphs.

Theorem 5. *BFIDA* is asymptotically optimal with respect to the number of node expansions in directed graphs, if* (1) *the depth of the shallowest goal is bounded by the number of stored layers times a constant factor and* (2) *b^i new nodes are expanded in the ith iteration, where b is a constant greater than one.*

Proof. If the depth of the shallowest goal is bounded by the number of stored layers times a constant factor, then according to Theorem 2, the maximum number of times a node can be regenerated (or reexpanded) is bounded by the same constant factor. Similarly, the number of nodes expanded by BFIDA* in the last iteration is also bounded by the

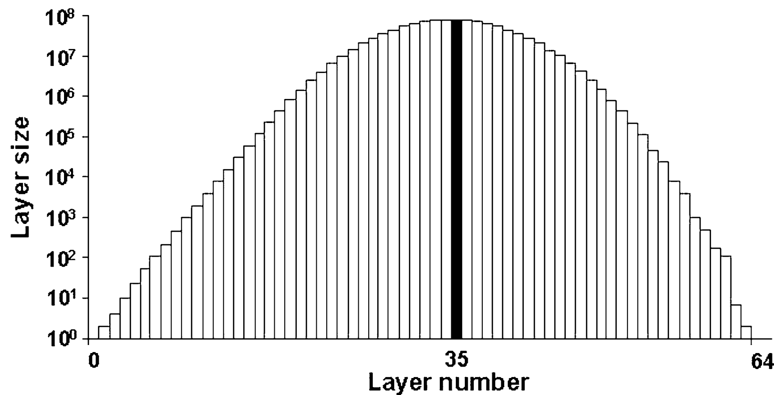

Fig. 3. Size of layer as function of search depth, in search graph generated by frontier-BFBnB for Korf's most difficult 15-puzzle instance (No. 88).

number of nodes expanded by A* times the same constant factor. In other words, BFIDA* expands, asymptotically, as many nodes as A* in the last iteration. Since the last iteration determines the time complexity of BFIDA* (see proof of Theorem 4), and A* is optimal with respect to the number of node expansions, Theorem 5 follows. □

4.2. Tie breaking

The optimality of A* with respect to the number of node expansions, and the asymptotic optimality of DFIDA* and BFIDA*, hold when the effect of tie breaking is not considered. One algorithm may expand more or fewer nodes than another, depending on how it breaks ties.

In graphs with unit edge costs, tie breaking is very important because there are typically many nodes with the same f-cost as the least-cost goal node. Because the search can terminate as soon as an optimal solution is found, the order in which tie nodes are considered can have a significant effect on both running time and memory use. It is well known that A* performs well using a tie-breaking rule that expands nodes with the same f-cost in order of least h-cost, since such nodes are typically closer to the goal. Similarly, DFIDA* performs well using *node ordering* [24], a strategy in which a depth-first traversal branches on least-cost successor nodes first, and, when successor nodes have the same f-cost, expands nodes in order of least h-cost. Node ordering can significantly reduce the number of nodes visited in the last iteration of DFIDA*, which is important since the last iteration dominates the running time of the algorithm.

Unfortunately, BFIDA* has limited ability to use a tie-breaking strategy to improve performance. Commitment to breadth-first traversal means that it must expand all nodes with an f-cost less than or equal to the hypothetical upper bound, in every layer of the breadth-first search graph until the layer that immediately precedes the layer containing the least-cost goal node. Only in this layer can BFIDA* leverage some tie-breaking benefit by node ordering, since breadth-first search can terminate as soon as a goal node is generated. But since layers that are close to the goal tend to be small due to the strength of the heuristic (see Fig. 3), this form of tie breaking has very limited benefit. In fact, BFIDA* suffers from almost worst-case tie-breaking performance, since it expands almost all nodes with an optimal f-cost. Even random tie-breaking in best-first search can perform much better than worst-case tie-breaking.

Although BFBnB can also suffer from worst-case tie breaking, it does not necessarily do so. When BFBnB uses an upper bound that corresponds to an actual solution found by an approximate search algorithm, and if the cost of the solution path found by the approximate search algorithm happens to be optimal, then BFBnB only expands nodes with an f-cost that is *strictly less than* the optimal solution cost. In this case, BFBnB achieves perfect tie-breaking and does not expand any node with an optimal f-cost. But if the upper bound corresponds to a sub-optimal solution, or if it is a hypothetical upper bound generated by iterative deepening, then BFBnB has worst-case tie-breaking performance.

5. Computational results

We now present test results that show the advantages of breadth-first heuristic search. The test results are for the Fifteen Puzzle, the 4-pegs Towers of Hanoi problem, and STRIPS planning in several different domains. For each problem, we begin by discussing relevant implementation details, and then present computational results.

5.1. Fifteen Puzzle

The Fifteen Puzzle is a standard benchmark for evaluating search algorithms, and was used by Korf to evaluate the performance of DFIDA* [13]. We use it to compare the performance of BFIDA* to the best-first alternatives of frontier-A* and sparse-memory A*.

Implementation For the Fifteen Puzzle, our implementation of BFIDA* uses frontier-BFBnB, as described in Section 3.2. For this problem (and for other problems with an undirected search graph and low branching factor), frontier search is easy to implement and provides the most memory-efficient implementation of BFBnB. Instead of storing the previous layer of nodes in memory in order to check for duplicates, as in layered duplicate detection, frontier search stores used-operator bits in each open node in order to block regeneration of nodes in the previous layer. This saves space since a node can be removed from memory as soon as it is expanded. Unlike frontier-A*, frontier-BFBnB can also prune any node from the Open list that has an f-cost greater than an upper bound. In BFIDA*, the upper bound is created by iterative deepening. For divide-and-conquer solution reconstruction, our implementation of frontier-BFBnB uses relay nodes and saves a relay layer at about the $3/4$ depth of the search graph.

To ensure fair comparison, we used the same relay node technique in implementing frontier-A* and sparse-memory A* as in frontier-BFBnB. In our implementation of sparse-memory A*, the Open list is pruned using an upper bound that corresponds to an approximate solution found by weighted A*, with the heuristic weighted by 1.2.

Results We tested BFIDA* on the same 100 instances of the Fifteen Puzzle used as a test set by Korf [13]. We used Manhattan distance as the heuristic. Table 1 shows results for the ten most difficult instances. BFIDA* solves all 100 instances using no more than 1.3 gigabytes of memory. Given 4 gigabytes of memory, neither frontier-A* nor sparse-memory A* can solve more than 96 instances; the instances they cannot solve are numbers 17, 60, 82, and 88. For the 96 solvable instances, frontier-A* stores an average of 4.15 times more nodes than BFIDA*, and sparse-memory A* stores an average of 2.74 times more nodes. But since a breadth-first approach allows a smaller node data structure,[1] the memory savings are actually greater. The memory used by frontier-A* and sparse-memory A* is 6.2 and 4.1 times greater, respectively, than that used by BFIDA*, averaged over these 96 solvable instances.

Based on the number of nodes that need to be expanded to solve these Fifteen Puzzle instances, A* would need 16 times more memory than BFIDA* just to store the Closed list, and it typically needs even more memory to store the Open list. Of course, BFIDA* must reexpand some nodes due to iterative deepening and divide-and-conquer solution reconstruction. But as can be seen from Table 1, this increase in the number of node expansions is dominated by the

Table 1
Performance of BFIDA* (using the Manhattan distance heuristic) on the 10 most difficult instances of Korf's 100 random instances of the Fifteen Puzzle [13]

#	Len	Stored	Last iter	Prev iters	Sol Reconst
17	66	16,584,444	218,977,081	60,096,047	94,283
49	59	21,177,925	243,790,912	101,531,263	377,910
53	64	12,753,096	177,244,033	47,192,566	109,254
56	55	13,066,308	141,157,391	67,716,057	27,529
59	57	13,974,753	158,913,130	69,947,124	40,469
60	66	56,422,199	767,584,679	211,192,399	27,807
66	61	21,435,302	275,076,045	93,017,396	44,823
82	62	46,132,337	549,557,759	215,694,244	356,986
88	65	77,547,650	999,442,569	360,978,147	161,730
92	57	12,591,419	151,699,572	61,733,319	438,877

Columns show the instance number (#); solution length (Len); peak number of nodes stored (Stored); number of node expansions in the last iteration (Last iter); total number of node expansions in all previous iterations (Prev iters); and number of node expansions during solution reconstruction (Sol Reconst).

[1] A node in frontier-A* or sparse-memory A* is about 50% larger than a node in breadth-first search because breadth-first search does not need to store the g-cost for each node, and a circular buffer implementation of the Open list simplifies the node data structure by exploiting the fact that once a node is inserted into the Open list, its priority never changes.

number of nodes expanded by BFIDA* in its last iteration, which is the same as the number of nodes that would be expanded by A* (if it expands all ties). The additional node expansions due to iterative deepening and solution reconstruction range from only 27% to 48% of this total, in solving these Fifteen Puzzle instances.

Fig. 3 shows the size of each layer of the search graph in breadth-first heuristic search for Korf's most difficult instance of the Fifteen Puzzle. Note that the largest layers are in the middle of the search graph, and middle layers are orders of magnitude larger than other layers, especially layers near the start or goal node. (The y-axis is logarithmically scaled.) This illustrates why we use a layer at about the $3/4$ depth of the search space as the relay layer, instead of a middle layer. Because a relay layer is not saved until well past the midpoint of the search, and the frontier is much smaller at the $3/4$ depth, storing a relay layer there has no effect on the peak memory requirements of the algorithm. This is an advantage of breadth-first heuristic search. By contrast, in best-first search (and uninformed breadth-first search), the size of the frontier continues to increase with the depth of the search, and storing intermediate nodes (e.g., a relay layer) inevitably increases peak memory requirements.

Effect of heuristic We have conjectured that the primary reason breadth-first heuristic search is more memory-efficient than best-first heuristic search is that its frontier is smaller. The intuitive explanation we offered is that a best-first frontier is more "stretched-out" toward the goal by the heuristic, as illustrated in Fig. 1. This conjecture suggests that the advantage of breadth-first over best-first heuristic search could increase with a more informative heuristic.

To test this, we compared the performance of breadth-first and best-first heuristic search in solving the Fifteen Puzzle, using three increasingly informative heuristics. In addition to the Manhattan distance heuristic, we used the linear-conflicts heuristic [8] and a disjoint pattern database heuristic [17].[2] Because sparse-memory A* outperforms frontier-A* in solving the Fifteen Puzzle, we used sparse-memory A* as the best-first search algorithm in our comparison.

Table 2 shows the number of nodes stored by sparse-memory A* when it prunes the Open list using an optimal upper bound, compared to the number of nodes stored by BFIDA*, for each of the three heuristics.[3] Of the two columns shown for sparse-memory A*, the column labeled *Tie-breaking* shows the number of nodes stored by sparse-memory A* when the algorithm stops as soon as an optimal solution is found. (Note that our implementation of sparse-memory A* employs the tie-breaking rule of "most recently generated". This tie-breaking rule performs well

Table 2
Comparison of the peak number of nodes stored by sparse-memory A* and BFIDA* using three different heuristics, averaged over the 91 easiest instances of Korf's 100 random instances of the Fifteen Puzzle. (These are the instances that can be solved by sparse-memory A* using 4 GB of RAM, when all ties are expanded.)

Heuristic function	Sparse-memory A*				BFIDA*
	Tie-breaking		All ties		
Manhattan distance	11,296,799	(6.0×)	17,473,487	(9.2×)	1,891,547
Linear conflicts	2,407,332	(5.7×)	4,052,643	(9.5×)	424,739
Disjoint database	31,602	(5.8×)	64,151	(11.9×)	5,414

For sparse-memory A*, the number of nodes stored when the goal node is expanded for the first time (which usually terminates the search) and when all nodes with f-cost less than or equal to the optimal solution length have been expanded, are shown in the columns labeled Tie-breaking and All ties, respectively.

[2] In the disjoint pattern database, one pattern includes tiles 1, 4, 5, 8, 9, 12, 13, and the other pattern includes tiles 2, 3, 6, 7, 10, 11, 14, 15. Both ignore the blank.

[3] It may seem surprising that the relative number of nodes stored by sparse-memory A* compared to BFIDA* is greater in Table 2 than reported in an earlier paragraph, even though sparse-memory A* uses an optimal upper bound for the results reported in Table 2, and a possibly sub-optimal upper bound computed by weighted A* for the previous results. The reason for this is that the optimal upper bound is hypothetical, for comparison to BFIDA*; therefore, sparse-memory A* can only prune open nodes with an f-cost strictly greater than this upper bound. Because the upper bound computed by weighted A* corresponds to an actual solution, sparse-memory A* can prune any open node with an f-cost equal to or greater than the upper bound. In practice, weighted A* often finds an optimal solution for the Fifteen Puzzle (although there is no guarantee that it is optimal). When it does, sparse-memory A* enjoys perfect tie-breaking. For the results reported in Table 2, sparse-memory A* does not have this advantage, even though its upper bound is optimal.

due to low overhead.) The column labeled *All ties* shows the number of nodes stored when every node with an f-cost less than or equal to the optimal upper bound is expanded. Also shown, in parentheses, is the factor by which sparse-memory A* stores more nodes than BFIDA*, using the same heuristic.

Although the numbers under the column labeled *Tie-breaking* do not suggest that a more informative heuristic has an effect on the relative size of the frontiers, this result is affected by the tie-breaking strategy of sparse-memory A*, which reduces the number of nodes expanded by sparse-memory A* relative to the number of nodes expanded by BFIDA*. Sparse-memory A* may expand many fewer nodes than BFIDA* due to better tie-breaking. The numbers under the column labeled *All ties* are not affected by tie-breaking because when sparse-memory A* expands "all ties", it expands the same nodes as BFIDA*. As a result, the difference in the number of stored nodes in the column *All ties* reflects the relative size of the frontiers, independent of the effect of tie-breaking. The numbers under this column do suggest that with a more informative heuristic, the size of the best-first frontier relative to the size of the breadth-first frontier increases. Although the effect is minor and far from conclusive, it lends some additional support to our conjecture that a search heuristic can affect not only the shape, but the relative size of the frontiers.

No matter which heuristic is used, the results clearly show that breadth-first heuristic search has a significant advantage over best-first heuristic search. Nevertheless, BFIDA* is still not as effective a search algorithm as DFIDA* in solving the Fifteen Puzzle. DFIDA* expands many more nodes than BFIDA*, of course, since it cannot eliminate duplicates. (The number of nodes expanded by DFIDA* in solving the 100 random instances of the Fifteen Puzzle is given by Korf [13].) Even though DFIDA* expands more nodes, it still runs much faster than BFIDA* in solving the Fifteen Puzzle, due to lower node-generation overhead and the fact that the number of duplicate paths grows rather slowly with the depth of the search, for this problem. For the other test problems we consider in this paper, and for many other graph-search problems, DFIDA* loses its advantage.

5.2. 4-peg Towers of Hanoi

The 4-peg Towers of Hanoi problem has been used as a benchmark in recent research on heuristic search [6,16,31]. The Towers of Hanoi problem consists of at least three pegs and a set of disks of different sizes. The task is to move all disks to a goal peg, subject to the constraints that only the top disk on any peg can be moved at a time, and a larger disk can never be placed on top of a smaller disk. For the well-known 3-peg problem, there is a simple deterministic algorithm that generates a provably optimal solution for moving n disks initially stacked on one peg to a goal peg within 2^{n-1} steps. For the 4-peg Towers of Hanoi problem [11], there is a deterministic algorithm that moves all disks from an initial peg to a goal peg, and a conjecture that it finds the fewest number of moves [7,25]. However, the conjecture remains unproven. Thus, systematic search is currently the only way to find a provably optimal solution, or to verify the conjecture for a given number of disks.

The state-space graph of the (4-peg) Towers of Hanoi problem contains many small cycles, which cause the number of paths from the start state to any other state in the graph to increase exponentially with the depth of the search. Because a depth-first search algorithm, such as DFIDA*, does not detect multiple paths to a state, it will generate many nodes representing the same state and perform very poorly. An efficient algorithm for this problem must prevent duplicate nodes. Previously, the best heuristic-search algorithm for solving this problem was frontier-A* [6].

The state-of-the-art heuristic for the 4-peg Towers of Hanoi problem uses disjoint pattern databases in which all disks are divided into two nonoverlapping groups and a pattern database is built for each group. Since each pattern database only counts moves of disks within its own group, the number of moves stored in two separate pattern databases can be added to get an admissible heuristic. For the Towers of Hanoi problem, the most accurate heuristic is based on dividing disks into one large group that yields the largest pattern database that fits in memory, and one small group that contains the remaining disks [6].

Results For the 4-peg Towers of Hanoi problem, an exact upper bound can be found by the deterministic algorithm that is conjectured (though not known) to be optimal [7,25]. Given this upper bound, there is no need to perform iterative deepening, and we used frontier-BFBnB to solve the 4-peg Towers of Hanoi problem for 17, 18, and 19 disks, assuming the standard start state of all disks stacked on an initial peg. To guide the search, we used a disjoint pattern database heuristic that included a 16-disk pattern database that was reduced in size by a factor of 16 using a compression technique described in [6]. Table 3 compares the performance of frontier-A* and frontier-BFBnB. The results clearly show that a breadth-first frontier is smaller than a best-first frontier. To the best of our knowledge, frontier-

BFBnB is the most memory-efficient algorithm for this problem and no other algorithm can solve the 19-disk problem within 2 gigabytes of memory, using the same heuristic.

Frontier-BFBnB has another important advantage over frontier-A*. The disjoint pattern database heuristic used by both search algorithms is compressed using a lossy compression method that makes the heuristic inconsistent, although it is admissible [6]. When the heuristic is not consistent, frontier-A* is not guaranteed to find an optimal solution [19]. One of the reasons frontier-A* expands fewer nodes than frontier-BFBnB for this problem is that frontier-A* sometimes expands a node when its g-value is sub-optimal, due to the inconsistent heuristic, and once frontier search closes a node, it cannot reopen it. Because sub-optimal g-values are propagated through the search space, some nodes on the frontier have higher f-values than they should, and the search algorithm stops before they are expanded, which can lead to a sub-optimal solution. Unlike frontier-A*, frontier-BFBnB is guaranteed to find an optimal solution when the heuristic is admissible but not consistent, as in this case.

Effect of heuristic To further investigate the effect of a heuristic on the relative size of best-first and breadth-first frontiers for the 4-peg Towers of Hanoi problem, we compared the performance of frontier-A* and frontier-BFBnB using seven increasingly informative disjoint pattern-database heuristics. All of the pattern-database heuristics are uncompressed, and thus, consistent. Table 4 compares the peak number of nodes stored by frontier-A* and frontier-BFBnB. Our implementation of frontier-A* employs the tie-breaking rule of "most recently generated". Although the effect of tie-breaking on the performance of frontier-A* is insignificant in this domain, we still rely on the column labeled *All ties* when comparing the two algorithms, because it guarantees that both algorithms expand the same set of nodes.

As before, the results clearly show that breadth-first heuristic search requires significantly less memory than best-first search, using the same heuristic. In comparing different heuristics, Table 4 shows that the size of the best-first frontier relative to the size of the breadth-first frontier does not increase monotonically with a more informative heuristic. For example, it decreases from 3.6 times to 3.3 times when the disjoint pattern-database heuristic changes

Table 3
Comparison of frontier-A* and frontier-BFBnB in solving the standard start state of the 4-peg Towers of Hanoi problem with 17, 18, and 19 disks, using a disjoint pattern-database heuristic that includes a compressed 16-disk pattern database

Disks	Len	Frontier-A*		Frontier-BFBnB	
		Stored	Exp	Stored	Exp
17	193	2,126,885	10,398,240	390,844	11,628,818
18	225	25,987,984	202,577,805	6,987,695	211,993,782
19	257	> 128,000,000	> 1,193,543,025	55,241,327	1,824,533,083

Columns show the number of disks (Disks); solution length (Len); peak number of nodes stored (Stored); and number of node expansions (Exp). The > symbol indicates that frontier-A* ran out of memory before solving the problem.

Table 4
Comparison of the number of nodes stored by frontier-A* and frontier-BFBnB in solving the standard start state of the 4-peg Towers of Hanoi problem with 16 disks, using 7 different disjoint pattern-database heuristics

Heuristic	Frontier-A*				Frontier-BFBnB
	Tie-breaking		All ties		
9-7	99,837,997	(2.3×)	105,343,542	(2.5×)	42,727,371
10-6	87,903,443	(2.9×)	91,878,927	(3.0×)	30,139,563
11-5	53,038,629	(2.8×)	56,690,926	(3.0×)	19,004,260
12-4	27,029,117	(3.3×)	29,713,012	(3.6×)	8,252,322
13-3	10,495,998	(3.0×)	11,485,231	(3.3×)	3,471,915
14-2	3,109,621	(3.0×)	3,205,198	(3.1×)	1,027,344
15-1	679,287	(5.0×)	729,739	(5.4×)	135,155

The factor by which frontier-A* stores more nodes than frontier-BFBnB using the same heuristic function is shown in parentheses. For frontier-A*, the number of nodes stored when the goal node is expanded for the first time (which usually terminates the search) and when all nodes with f-cost less than or equal to the optimal solution length have been expanded, are shown in the columns labeled Tie-breaking and All ties, respectively.

from a 12-4 partition to a more informative 13-3 partition. Possibly, this is due to random variation in the state-space graph of the 4-peg Towers of Hanoi problem, since only a single problem instance is solved for each heuristic. If we focus on heuristics that differ significantly in terms of accuracy, we observe the expected trend. For example, if we only consider the least informative heuristic (9-7 partition), the most informative heuristic (15-1 partition), and the heuristic that lies in the middle (12-4 partition), we observe that a more informative heuristic increases the size of the best-first frontier relative to the breadth-first frontier. This provides additional evidence that a more informative heuristic gives breadth-first heuristic search a greater relative advantage over best-first search. But it also shows that there is no guarantee this will happen. Even if it happens for a set of problem instances on average, it may not happen for a particular problem instance.

5.3. Domain-independent STRIPS planning

Over the past several years, the effectiveness of heuristic search for domain-independent STRIPS planning has become widely-recognized. Both A* and DFIDA* are used to find optimal plans, guided by an admissible heuristic [10], and weighted A* is used to find approximate plans for difficult planning problems, guided by an informative, though usually non-admissible, heuristic [2]. Heuristic-search planners have performed very well in the biennial International Planning Competition that is hosted at the Artificial Intelligence Planning and Scheduling Conference series [20]. The problems used in the competition provide a good test set for comparing graph-search algorithms since they give rise to a variety of search graphs with different kinds of structure, and memory is a limiting factor in solving many of the problems.

We tested the performance of breadth-first heuristic search in solving problem instances from eight unit-cost planning domains used in previous competitions. One of the domains (*gripper*) is from the 1998 competition, three (*blocks*, *elevator*, and *logistics*) are from the 2000 competition, and the other four (*satellite*, *driverlog*, *depots*, and *freecell*) are from the 2002 competition. For a description of the planning domains, see [20].

Implementation For domain-independent STRIPS planning, breadth-first heuristic search has an important advantage over best-first heuristic search when divide-and-conquer solution reconstruction is used to reduce memory requirements: it is *much* easier to implement. Implementing frontier-A* or sparse-memory A* for domain-independent STRIPS planning poses serious difficulties. First, implementing used-operator bits in a domain-independent way can increase node size substantially, since every possible operator instantiation must be considered. In addition, when STRIPS operators are only conditionally reversible, used-operator bits are difficult to implement because it is impossible to determine reversibility in advance. (Recall that when frontier-A* expands a node, it sets a used-operator bit in each successor node that prevents it from regenerating the just-expanded node, where the operator for which the bit is set is the *reverse* of the operator that generated the node.) Finally, for STRIPS planning problems that correspond to directed graphs, implementing dummy nodes (as in frontier-A*) or predecessor counting (as in sparse-memory A*) is challenging because of the difficulty of identifying all predecessors of a node, especially since the number of potential predecessors is exponential in the size of the Add list of an operator, and all operators must be considered.

By contrast, implementing breadth-first branch-and-bound search using layered duplicate detection is straightforward. With layered duplicated detection, there is no need for used-operator bits, dummy nodes or predecessor counters. The algorithm simply stores one or more previous layers and checks for duplicates. In the experiments reported here, we stored a single previous layer to check for duplicates. Four of the eight domains (*logistics*, *blocks*, *gripper*, and *elevator*) have undirected search graphs[4] for which a single previous layer is guaranteed to detect all duplicates. The other four domains have directed search graphs for which graph locality is not easy to determine. We found empirically that storing a single previous layer is sufficient to eliminate almost all duplicates in the *satellite*, *driverlog*, and *depots* domains. In the *freecell* domain, storing a single previous layer eliminates most duplicates but still allows many to be generated.

Given the difficulty of implementing frontier-A* or sparse-memory A* for domain-independent planning, we compared breadth-first heuristic search to A* and DFIDA*. In this comparison, all of the search algorithms perform regression planning, which involves searching backward from the goal to the start state. We used the HSPr planning

[4] Whether the search graph is directed or undirected sometimes depends on whether forward or regression planning is performed. We assume regression planning.

system of Bonet and Geffner [1] as a foundation for implementing our algorithms. (We made some changes to improve the performance of its implementation of A*.) Since we give timing results for these planning problems, we note that the experiments were conducted on a Pentium IV 2.4 GHz processor with 2 gigabytes of RAM.

Results Table 5 compares the performance of A* to BFBnB, where BFBnB uses layered duplicate detection and an upper bound computed by an approximate search algorithm called divide-and-conquer beam search [28]. The problem instances in Table 5 are the largest that A* can solve in each domain. Both algorithms use the *max-pair* heuristic [10]. The results show that BFBnB expands more nodes than A*, but uses significantly less memory.

For all of the planning problems except *gripper-7*, the divide-and-conquer beam search algorithm used to compute an upper bound finds a solution that turns out to be optimal. This is why there are relatively more node expansions for BFBnB, compared to A*, for *gripper-7*; its upper bound is not optimal. (For *freecell-3*, there are relatively more node expansions despite an optimal upper bound because storing a single previous layer of the graph does not eliminate all duplicates.) Note that the number of node expansions and the running time reported for BFBnB in the table includes the node expansions and running time of the approximate search algorithm.

Table 6 compares the performance of Haslum's implementation of DFIDA* to BFIDA*. Haslum's implementation uses a transposition table to eliminate some duplicates, as described in his paper [10]. The problem instances in Table 6 are easier than those in Table 5 because DFIDA* cannot solve many instances that even A* can solve. The reason we

Table 5
Comparison of A* and BFBnB with layered duplicate detection on STRIPS planning problems

Instance	Len	A*			BFBnB		
		Stored	Exp	Secs	Stored	Exp	Secs
logistics-6	25	364,758	254,072	5	115,141	301,173	5
blocks-14	38	735,905	252,161	13	79,649	258,098	12
gripper-7	47	10,092,451	10,088,369	149	3,417,064	20,908,118	333
satellite-6	20	3,269,703	2,423,288	178	1,663,286	2,644,616	185
elevator-11	37	3,893,277	3,884,960	181	982,838	3,972,828	188
driverlog-10	17	17,901,481	3,874,617	255	2,413,461	4,071,392	210
depots-7	21	21,026,728	7,761,605	372	5,266,323	8,084,389	326
freecell-3	18	5,992,428	2,693,167	212	3,015,483	5,997,287	425

Both algorithms use the *max-pair* heuristic function. Columns show optimal solution length (Len); peak number of nodes stored (Stored); number of node expansions (Exp); and running time in CPU seconds (Secs).

Table 6
Comparison of DFIDA* (with transposition table) and BFIDA* in solving STRIPS planning problems

Instance	Len	DFIDA*			BFIDA*		
		Stored	Exp	Secs	Stored	Exp	Secs
logistics-4	20	2,289	45,194,644	622	1,730	16,077	1
logistics-5	27	–	–	–	37,548	504,712	6
blocks-12	34	5,015	180,305	50	6,354	34,687	2
blocks-14	38	94,011	51,577,732	19,901	224,058	1,324,320	44
gripper-2	17	1,380	6,368,202	121	741	10,375	< 1
gripper-6	41	–	–	–	742,988	35,705,971	389
satellite-3	11	682	22,033	1	1,855	4,001	< 1
satellite-4	17	–	–	–	70,298	303,608	8
driverlog-7	13	42,148	4,304,450	450	193,979	422,475	13
driverlog-10	17	–	–	–	4,960,572	16,849,713	620
depots-2	15	2,073	227,289	31	1,923	8,139	1
depots-3	27	–	–	–	4,841,706	21,233,622	504
freecell-2	14	68,415	42,500,257	34,674	162,215	545,716	35
freecell-3	18	–	–	–	3,015,483	10,154,249	720

Columns show optimal solution length (Len); peak number of nodes stored (Stored); number of node expansions (Exp); and running time in CPU seconds (Secs). No results are shown for DFIDA* when it could not solve the problem after 12 hours of CPU time.

compared A* to BFBnB in Table 5, and DFIDA* to BFIDA* in Table 6, is that the former pair of algorithms do not use iterative deepening and the latter do.

As Table 6 shows, DFIDA* performs much worse than BFIDA* due to excessive node regenerations. Because the successor generation and heuristic evaluation functions for STRIPS planning are expensive to compute, node regeneration is particularly expensive for these search problems. The results clearly illustrate that for Planning Competition problems, effective duplicate detection is essential for good performance. We mentioned before that saving one previous layer does not eliminate all duplicates in the *freecell* domain. Nevertheless, Table 6 shows that it eliminates enough duplicates so that the problem can be solved efficiently; by contrast, DFIDA* with transposition tables is unable to solve *freecell-3* because too many duplicates are generated.

Effect of heuristic To help evaluate the effect of the heuristic on the relative size of breadth-first and best-first frontiers in STRIPS planning, we implemented a *max-triple* heuristic. This is similar to the max-pair heuristic used for the previous results except that it considers the interaction of three atoms instead of two, which makes the heuristic more informative [10]. Table 7 compares the peak number of nodes stored by A* and BFIDA* using the two different heuristics. The implementation of A* employs a tie-breaking rule that favors nodes with the least *additive heuristic* [2]. As before, we rely on the column labeled *All ties* in comparing the algorithms, since it removes the effect of tie-breaking by guaranteeing that both algorithms expand the same set of nodes. Averaged over all problem instances, A* stores 4.8 times more nodes than BFIDA* using the max-pair heuristic, and 6.4 times more nodes than BFIDA* using the max-triple heuristic. This provides additional evidence that a more informative heuristic increases the relative size of the best-first frontier compared to a breadth-first frontier. As the table shows, this effect is problem-dependent. For some problem instances, the more informative heuristic makes little or no difference, or even slightly decreases the advantage of breadth-first search relative to best-first search.

Table 8 compares the peak number of *frontier* (i.e., open) nodes stored by A* and BFIDA* using the same two heuristics. By isolating the effect of the heuristic on the size of the frontiers, these results more unambiguously support the conclusion that a more informative heuristic increases the relative size of a best-first frontier compared to a breadth-first frontier. Interestingly, the peak size of a best-first frontier can sometimes *increase* with a more informative heuristic, although the total number of node expansions is smaller. For the elevator-11 problem, Table 8 shows that the best-first frontier using the max-triple heuristic is bigger than the best-first frontier using the max-pair heuristic, even though the former heuristic expands fewer nodes (as shown in Table 7).

Table 7
Comparison of number of nodes stored by A* and BFIDA* on STRIPS planning problems using the *max-pair* and the *max-triple* heuristic functions

Instance	Heuristic	A*				BFIDA*
		Tie-breaking		All ties		
logistics-6	Max-Pair	364,758	(2.9×)	448,233	(3.5×)	126,484
	Max-Triple	148,552	(4.0×)	207,011	(5.6×)	36,695
blocks-14	Max-Pair	735,905	(3.2×)	2,407,851	(10.6×)	228,020
	Max-Triple	81,192	(3.3×)	249,499	(10.1×)	24,758
gripper-7	Max-Pair	10,092,451	(3.6×)	10,092,513	(3.6×)	2,792,791
	Max-Triple	10,080,141	(3.6×)	10,085,793	(3.6×)	2,792,791
satellite-6	Max-Pair	3,269,703	(1.7×)	4,008,807	(2.1×)	1,953,396
	Max-Triple	1,456,457	(1.8×)	2,152,917	(2.6×)	832,114
elevator-11	Max-Pair	3,893,277	(4.0×)	3,896,505	(4.0×)	977,554
	Max-Triple	3,855,073	(3.9×)	3,883,736	(4.0×)	977,554
driverlog-10	Max-Pair	17,901,481	(2.9×)	45,011,158	(7.3×)	6,161,424
	Max-Triple	1,472,868	(2.9×)	4,160,620	(8.2×)	509,685
depots-7	Max-Pair	21,026,728	(2.0×)	39,961,845	(3.7×)	10,719,221
	Max-Triple	828,537	(4.3×)	1,830,282	(9.6×)	190,896
freecell-3	Max-Pair	5,992,428	(2.0×)	11,121,762	(3.7×)	3,015,483
	Max-Triple	463,697	(2.5×)	1,411,058	(7.6×)	184,797

The factor by which A* stores more nodes than BFIDA* using the same heuristic function is shown in parentheses. For A*, the number of nodes stored when the goal node is expanded for the first time (which usually terminates the search) and when all nodes with f-cost less than or equal to the optimal solution length have been expanded, are shown in the columns labeled Tie-breaking and All ties, respectively.

Table 8
Comparison of peak number of frontier nodes stored by A* and BFIDA* on STRIPS planning problems using the *max-pair* and the *max-triple* heuristic functions

Instance	Heuristic	A* frontier				BFIDA* frontier
		Tie-breaking		All ties		
logistics-6	Max-Pair	111,476	(1.8×)	121,478	(2.0×)	60,421
	Max-Triple	65,489	(3.1×)	82,595	(4.0×)	20,878
blocks-14	Max-Pair	483,798	(2.8×)	1,602,887	(9.2×)	174,179
	Max-Triple	53,504	(2.8×)	167,696	(8.9×)	18,828
gripper-7	Max-Pair	1,750,319	(1.0×)	1,750,319	(1.0×)	1,750,319
	Max-Triple	1,750,319	(1.0×)	1,750,319	(1.0×)	1,750,319
satellite-6	Max-Pair	1,015,478	(1.1×)	1,038,757	(1.1×)	941,496
	Max-Triple	633,730	(1.3×)	785,213	(1.6×)	502,811
elevator-11	Max-Pair	471,962	(1.0×)	471,962	(1.0×)	463,566
	Max-Triple	526,720	(1.1×)	526,720	(1.1×)	463,566
driverlog-10	Max-Pair	14,059,434	(2.9×)	34,685,980	(7.1×)	4,854,223
	Max-Triple	1,191,440	(2.8×)	3,322,892	(7.7×)	429,216
depots-7	Max-Pair	13,366,827	(1.9×)	24,328,038	(3.4×)	7,128,263
	Max-Triple	607,928	(4.0×)	1,291,554	(8.6×)	150,619
freecell-3	Max-Pair	3,303,587	(1.7×)	5,259,939	(2.7×)	1,935,546
	Max-Triple	344,689	(2.2×)	959,777	(6.2×)	155,157

The factor by which A* stores more frontier nodes than BFIDA* using the same heuristic function is shown in parentheses. For A*, the number of frontier nodes stored when the goal node is expanded for the first time (which usually terminates the search) and when all nodes with f-cost less than or equal to the optimal solution length have been expanded, are shown in the columns labeled Tie-breaking and All ties, respectively.

Altogether, these results provide compelling evidence of the effect of search strategy and heuristic on frontier size, and thus, on memory requirements. Frontier size may be influenced by other factors, especially by graph structure, and it is difficult to make completely general statements about the efficiency of graph search when graphs can have so many different kinds of structure. Nevertheless, our results show that breadth-first heuristic search can have a significant advantage over best-first heuristic search.

6. Extensions

We briefly consider two important extensions of breadth-first heuristic search. The details of these extensions are left for future work.

6.1. Non-uniform edge costs

The approach developed in this paper is restricted to search problems with uniform edge costs. Breadth-first search is well-suited for problems with uniform edge costs because it ensures that the first time a node is *generated*, an optimal path to it has been found. For problems with non-uniform edge costs, Dijkstra's single-source shortest-path algorithm can ensure that the first time a node is *expanded*, an optimal path to it has been found. But Dijkstra's algorithm expands nodes in order of least g-cost, instead of in breadth-first order. The breadth-first algorithms developed in this paper can be generalized in two different ways to solve problems with non-uniform edge costs. One keeps the breadth-first search strategy, and the other adopts the search strategy of Dijkstra's algorithm. An advantage of keeping the breadth-first strategy is that the same layered approach to duplicate detection can be used. A disadvantage is that the same node will need to be reexpanded every time a lower-cost path to it is found.

An alternative to breadth-first branch-and-bound search is a version of Dijkstra's algorithm in which nodes are expanded in order of least g-cost, but an admissible heuristic is used to prune the search space. Such an algorithm could be called Dijkstra's (or uniform-cost) branch-and-bound search, and it could be combined with frontier search or sparse-memory graph search. An advantage of this approach is that it makes it possible to guarantee that each node is expanded at most once. A disadvantage is that it is not obvious how to divide the search graph into layers for the purpose of layered duplicate detection. One possibility is to group all nodes with the same g-cost in the same layer. This will work well if there are not too many distinct edge costs in the graph, e.g., all edge costs are from a small set

of integers. For graphs with many distinct edge costs, such as graphs with real-valued edge costs, this approach could result in layers that are too small. An alternative is to group nodes with similar g-costs in the same layer. For example, a layer could contain all nodes whose g-costs are within a certain range. The details of this extension are beyond the scope of this paper and will be addressed in future work. (For an example of a class of graph-search problems with non-uniform edge costs for which a search strategy similar to breadth-first heuristic search is very effective, see [27].)

6.2. Memory limitations

The approach developed in this paper can significantly reduce the memory requirements of search, while ensuring duplicate detection. But its memory requirements are not bounded, and it is still possible to run out of memory if the number of nodes in any layer becomes too large.

If the largest layer (or adjacent layers) in the breadth-first search graph does not fit in memory, one way to handle this follows from the recognition that breadth-first search is very closely-related to beam search. Instead of considering all nodes in a layer, a beam-search variant of breadth-first branch-and-bound search considers the most promising nodes in a layer until memory is full (or reaches a predetermined bound). At that point, the algorithm recovers memory by pruning the least-promising nodes (i.e., the nodes with the highest f-cost) from the Open list. Then it continues the search. Aside from pruning the least-promising open nodes when memory is full, this algorithm is identical to BFBnB with layered duplicate detection. The difference from traditional beam search is that divide-and-conquer solution reconstruction is used to reduce memory requirements. This difference allows a beam search algorithm to use a much larger beam width in order to improve performance. It also allows an initial, sub-optimal solution to be improved when the search algorithm recursively solves subproblems during divide-and-conquer solution reconstruction. Further details of this algorithm and computational results are described in a separate paper [28]. It can be considered a form of breadth-first heuristic search that finds approximate solutions. An implementation of this approximate search algorithm was used to compute upper bounds for STRIPS planning problems in the results reported in Section 5.3.

A drawback of beam search is that it is not guaranteed to find an optimal solution; in some cases, it may not find any solution. This drawback may be overcome by allowing the beam search algorithm to backtrack to the points at which it could not generate all nodes in a layer, and continue the search from there. This idea can be used to create a complete beam search algorithm [30].

Another way to handle memory limitations is to use external memory such as disk storage. When a layer of the search graph becomes too large to fit in internal memory, nodes can be stored on disk. There has been some recent work on how to use disk storage efficiently in heuristic search. Zhou and Hansen [29] describe an approach called structured duplicate detection and successfully combine it with BFIDA*. Korf [16] and Edelkamp et al. [5] describe another approach called delayed duplicate detection.

7. Conclusion

Best-first search is traditionally considered more efficient than breadth-first search because it minimizes the number of node expansions. The contribution of this paper is to show that when a reduced-memory approach to graph search is adopted that stores only (or primarily) the search frontier, and relies on a divide-and-conquer method of solution reconstruction, a breadth-first approach has several advantages over best-first search.

The primary advantage is that a breadth-first frontier can be smaller than a best-first frontier, requiring less memory to prevent regeneration of closed nodes. This allows a more memory-efficient search that still guarantees duplicate detection. But this is not the only advantage. In breadth-first search of a graph with uniform edge costs, the f-cost of a node is guaranteed to be optimal as soon as the node is generated; in frontier-BFBnB search, this allows the Open list to be pruned using an upper bound, and it means that frontier-BFBnB search only requires an admissible heuristic, not a consistent heuristic, in order to find an optimal solution. This gives frontier-BFBnB an advantage over frontier-A*. The breadth-first approach also allows a different strategy of duplicate detection which we call layered duplicate detection. Although this method of duplicate detection may not be as memory-efficient as frontier search, it is easier to implement and can be applied to problems for which frontier search is impractical. A drawback of breadth-first heuristic search is that it can result in worst-case tie-breaking. However, computational results show that its advantages outweigh this disadvantage.

The breadth-first heuristic search algorithms described in this paper include a memory-efficient implementation of breadth-first branch-and-bound search and a breadth-first iterative-deepening A* algorithm that is based on it. Computational results show that these algorithms outperform frontier-A* and sparse-memory A* in solving the Fifteen Puzzle; they outperform frontier-A* and all other search algorithms in solving the 4-peg Towers of Hanoi problem; and they outperform A* and DFIDA* in solving STRIPS planning problems.

The breadth-first heuristic search algorithms developed in this paper make the limiting assumption that all operators have uniform cost. In future work, we will show how to relax this assumption and extend this approach to search problems in which operators have non-uniform and possibly real-valued costs.

Acknowledgements

Thanks to the anonymous reviewers for helpful comments and suggestions. Thanks to Blai Bonet and Patrick Haslum for making the code for their heuristic-search planners publicly available. This work was supported in part by NSF grant IIS-9984952 and NASA grant NAG-2-1463.

A preliminary version of this paper appears in the Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS-04) [28], and received the Best Student Paper Award at the conference.

References

[1] B. Bonet, H. Geffner, Heuristic search planner 2.0, AI Magazine 22 (3) (2001) 77–80.
[2] B. Bonet, H. Geffner, Planning as heuristic search, Artificial Intelligence 129 (1) (2001) 5–33.
[3] R. Dechter, J. Pearl, Generalized best-first search strategies and the optimality of A*, J. ACM 32 (3) (1985) 505–536.
[4] E. Dijkstra, A note on two problems in connexion with graphs, Numer. Math. 1 (1959) 269–271.
[5] S. Edelkamp, S. Jabbar, S. Schrodl, External A*, in: Proceedings of the 27th German Conference on Artificial Intelligence, Ulm, Germany, 2004, pp. 226–240.
[6] A. Felner, R. Meshulam, R. Holte, R. Korf, Compressing pattern databases, in: Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-04), San Jose, CA, 2004, pp. 638–643.
[7] J. Frame, Solution to advanced problem 3918, Amer. Math. Monthly 48 (1941) 216–217.
[8] O. Hansson, A. Mayer, M. Yung, Criticizing solutions to relaxed models yields powerful admissible heuristics, Information Sci. 63 (3) (1992) 207–227.
[9] P. Hart, N. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Trans. Syst. Sci. Cybernet. (SSC) 4 (2) (1968) 100–107.
[10] P. Haslum, H. Geffner, Admissible heuristics for optimal planning, in: Proceedings of the 5th International Conference on AI Planning and Scheduling, Breckenridge, CO, 2000, pp. 140–149.
[11] A.M. Hinz, The Tower of Hanoi, in: Algebras and Combinatorics: Proceedings of ICAC'97, Hong Kong, 1997, pp. 277–289.
[12] D. Hirschberg, A linear space algorithm for computing maximal common subsequences, Comm. ACM 18 (6) (1975) 341–343.
[13] R. Korf, Depth-first iterative deepening: An optimal admissible tree search, Artificial Intelligence 27 (1) (1985) 97–109.
[14] R. Korf, Linear-space best-first search, Artificial Intelligence 62 (1) (1993) 41–78.
[15] R. Korf, Divide-and-conquer bidirectional search: First results, in: Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99), Stockholm, Sweden, 1999, pp. 1184–1189.
[16] R. Korf, Best-first frontier search with delayed duplicate detection, in: Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-04), San Jose, CA, 2004, pp. 650–657.
[17] R. Korf, A. Felner, Disjoint pattern database heuristics, Artificial Intelligence 134 (1–2) (2002) 9–22.
[18] R. Korf, W. Zhang, Divide-and-conquer frontier search applied to optimal sequence alignment, in: Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-00), Austin, TX, 2000, pp. 910–916.
[19] R. Korf, W. Zhang, I. Thayer, H. Hohwald, Frontier search, J. ACM 52 (5) (2005) 715–748.
[20] D. Long, M. Fox, The 3rd international planning competition: Results and analysis, J. Artificial Intelligence Res. 20 (2003) 1–59.
[21] T. Miura, T. Ishida, Stochastic node caching for memory-bounded search, in: Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98), Madison, WI, 1998, pp. 450–456.
[22] E. Myers, W. Miller, Optimal alignments in linear space, Comput. Appl. Biosci. 4 (1) (1988) 11–17.
[23] I. Pohl, First results on the effect of error in heuristic search, Machine Intelligence 5 (1970) 219–236.
[24] A. Reinefeld, T. Marsland, Enhanced iterative-deepening search, IEEE Trans. Pattern Anal. Machine Intelligence 16 (7) (1994) 701–710.
[25] B. Stewart, Solution to advanced problem 3918, Amer. Math. Monthly 48 (1941) 217–219.
[26] R. Zhou, E. Hansen, Sparse-memory graph search, in: Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03), Acapulco, Mexico, 2003, pp. 1259–1266.
[27] R. Zhou, E. Hansen, Sweep A*: Space-efficient heuristic search in partially ordered graphs, in: Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence (ICTAI-03), Sacramento, CA, 2003, pp. 427–434.
[28] R. Zhou, E. Hansen, Breadth-first heuristic search, in: Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS-04), Whistler, British Columbia, 2004, pp. 92–100.

[29] R. Zhou, E. Hansen, Structured duplicate detection in external-memory graph search, in: Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-04), San Jose, CA, 2004, pp. 683–688.
[30] R. Zhou, E. Hansen, Beam-stack search: Integrating backtracking with beam search, in: Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS-05), Monterey, CA, 2005, pp. 90–98.
[31] R. Zhou, E. Hansen, External-memory pattern databases using structured duplicate detection, in: Proceedings of the 20th National Conference on Artificial Intelligence (AAAI-05), Pittsburgh, PA, 2005, pp. 1398–1404.