Beam-Stack Search: Integrating Backtracking with Beam Search

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

Abstract

We describe a method for transforming beam search into a complete search algorithm that is guaranteed to find an optimal solution. Called beam-stack search, the algorithm uses a new data structure, called a beam stack, that makes it possible to integrate systematic backtracking with beam search. The resulting search algorithm is an anytime algorithm that finds a good, sub-optimal solution quickly, like beam search, and then backtracks and continues to find improved solutions until convergence to an optimal solution. We describe a memory-efficient implementation of beam-stack search, called divide-and-conquer beam-stack search, as well as an iterative-deepening version of the algorithm. The approach is applied to domain-independent STRIPS planning, and computational results show its advantages.

Introduction

Beam search is a widely-used approximate search algorithm. By focusing its search effort on the most promising paths through a search space, beam search can find a solution within practical time and memory limits – even for problems with huge search spaces. Although the idea is simple, beam search works very well in a wide variety of domains, including planning (Zhou & Hansen 2004), scheduling (Habenicht & Monch 2002), speech recognition (Huang, Acero, & Hon 2001), and many others.

Despite its wide use, beam search has a serious drawback – it is *incomplete*. Because the technique that beam search uses to prune the search space is inadmissible, it is possible for it to prune paths that lead to an optimal solution, or even to prune all paths that lead to any solution. As a result, there is no guarantee that beam search will find an optimal solution, or any solution at all, even when a solution exists.

In this paper, we describe a way to transform beam search into a complete search algorithm that is guaranteed to find an optimal solution. The key innovation is a new data structure called a *beam stack* that makes it possible to integrate beam search with systematic backtracking. Thus we call the new algorithm *beam-stack search*. We also describe a memory-efficient implementation of beam-stack search, called *divide-and-conquer beam-stack search*. The new search algorithm is an anytime algorithm. Like conventional beam search, it can find a good, sub-optimal solution quickly. Then it backtracks and continues to search for improved solutions until convergence to an optimal solution. We use the new algorithm for domain-independent STRIPS planning and report computational results that demonstrate its advantages.

Background

We begin with a brief review of beam search and some related work.

Beam search

Beam search can be viewed as an adaptation of branch-and-bound search that uses an inadmissible pruning rule in which only the most promising nodes at each level of the search graph are selected for further branching, and the remaining nodes are pruned off permanently. The standard version of beam search expands nodes in breadth-first order. In each layer of a breadth-first search graph, it expands only the w most promising nodes, and discards the rest, where the integer w is called the *beam width*. A heuristic is used to select the most promising nodes. Note that by varying the beam width, it is possible to vary the search between greedy search (with a width of 1) and complete search (with no limit on width). By bounding the width, the complexity of the search becomes linear in the depth of the search instead of exponential; the time and memory complexity of beam search is wd, where d is the depth of the search.

Although beam search is usually associated with a breadth-first search strategy, the name "beam search" is sometimes used in a more general sense. For example, Rich and Knight (1991) suggest applying a beam to best-first search; after a node's children are added to the Open list, the Open list is truncated so that it has at most w nodes. Bisiani (1987) proposes an even more general definition in which any search algorithm that uses heuristic rules to discard non-promising alternatives is an example of beam search. Based on this definition, Zhang (1998) refers to a depth-first branch-and-bound algorithm that uses a non-admissible pruning rule as a beam-search algorithm.

The approach developed in this paper applies to a standard beam-search algorithm that expands nodes in breadth-first order.

Complete beam search with iterative weakening

Zhang (1998) describes how to transform beam search into a complete (and anytime) search algorithm by using *iterative weakening* (Provost 1993).[1] The idea is to perform a sequence of beam searches in which a weaker inadmissible pruning rule is used each iteration, until a solution of desired quality is found, or until the last iteration, in which no inadmissible pruning rule is used. This creates an anytime algorithm that finds a sequence of improved solutions, and eventually converges to an optimal solution.

In this approach to creating a complete beam search algorithm, Zhang uses "beam search" in the general sense in which it refers to any search algorithm that uses an inadmissible pruning rule. He points out that this technique can be used to transform a best-first or depth-first beam-search algorithm into a complete algorithm. In fact, all the experimental results he reports are for depth-first branch-and-bound search applied to tree-search problems. In theory, his approach can be used to make best-first (or breadth-first) beam search complete. In practice, this creates a memory bottleneck that limits the scalability of the complete beam-search algorithm that results.

The reason for this is that iterative weakening only guarantees completeness in the last iteration in which no inadmissible pruning rule is used. At that point, the memory requirement is the same as for the underlying search algorithm. If this approach is used with a depth-first search algorithm, the memory requirement of the complete beam search is linear in the depth of search (the same as for depth-first search). But if it is used with a best-first (or breadth-first) search algorithm, the memory requirement of the complete beam search is the same as that of best-first (or breadth-first) search – typically, *exponential* in the depth of search.

In the rest of this paper, we describe a very different approach to making beam search complete. The approach applies to a standard beam-search algorithm that uses a breadth-first strategy of node expansion. But the memory complexity of the complete algorithm is the same as that of traditional beam search (and depth-first search) – linear in the depth of the search.

Beam-stack search

Like beam search, beam-stack search – which we introduce in this paper – can be viewed as a modification of breadth-first branch-and-bound (BFBnB) search in which no layer of the breadth-first search graph is allowed to grow greater than the beam width. It expands nodes in breadth-first order, and, like BFBnB, uses upper and lower bounds to prune the search space. For any node n, a lower-bound estimate of the cost of an optimal path through that node is given by the node evaluation function $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of a best path from the start node to node n, and $h(n)$ is an admissible heuristic that never over-estimates the remaining cost from node n to a goal node. An initial upper bound U can be found by using an approximation algorithm to find a sub-optimal solution. Like BFBnB, each time beam-stack search finds an improved solution, it updates the upper bound. Use of upper and lower bounds to prune the search space is called *admissible pruning*. Pruning nodes based on memory limitations, as in beam search, is *inadmissible pruning*.

[1]A closely-related technique called iterative broadening (Ginsberg & Harvey 1992) could also be used to make beam search complete in the way suggested by Zhang.

To compensate for inadmissible pruning, beam-stack search can backtrack to a previously-explored layer in order to generate nodes that were not generated when the layer was previously visited. To allow systematic backtracking, beam-stack search uses a novel data structure called a *beam stack*, which generalizes the conventional stack used in depth-first search. A beam stack contains one item for each layer of the breadth-first search graph (with the exception of the deepest layer, which does not need to be backtracked to). The item contains a high-level record of search progress in that layer.

The beam stack supplements the search graph in keeping track of the progress of the search. Each layer of a breadth-first search graph consists of a set of stored nodes, and, for any node in one layer, all of its possible successor nodes are in the next layer. But in beam search, if the beam width is not large enough, there is not room for all successor nodes in the next layer. Only some can be stored in the next layer, and the algorithm must backtrack to consider the rest.

To allow backtracking, the beam stack keeps track of which nodes have been considered so far, and which have not. It does so by leveraging the fact that the nodes in a layer can be sorted uniquely. In the simple case in which every node has a unique f-cost, they can be sorted by their f-cost. (We consider this case first, in order to explain the idea, and consider the possibility of ties later.) In this simple case, an item of the beam stack describes a range of f-costs, $[f_{\min}, f_{\max})$, which indicates that only successor nodes with an f-cost in this range are stored in the next layer.

When expanding nodes in a layer associated with an item with range $[f_{\min}, f_{\max})$, the algorithm prunes any successor node (generated for the next layer) with an f-cost less than $f_{\min}$, or greater than or equal to $f_{\max}$. The item associated with layer 0, which contains a single start node, is stored at the bottom of the beam stack, and the item associated with the currently-expanding layer is stored at the top of the beam stack. The first time the algorithm expands (a node in) a layer, it initializes the layer's corresponding beam-stack item to be $[0, U)$, where U is the current upper bound.

When memory is full (or the size of a layer reaches a predetermined bound, i.e., the beam width), then beam-stack search *inadmissibly* prunes nodes with the highest f-cost in order to make room for new nodes. It prunes nodes in the next layer, as our pseudocode will show. (It could be implemented to prune nodes in other layers also.) When beam-stack search prunes nodes in a layer, it changes the $f_{\max}$ of the previous layer's beam-stack item to the lowest f-cost of the nodes that have just been pruned. This ensures that the search algorithm will not generate any successor node with an f-cost greater than or equal to the lowest f-cost of the just-pruned nodes, before backtracking to this layer.

Beam-stack search backtracks once it reaches a layer for which all successor nodes have an f-cost greater than the upper bound U, which we call an *empty layer*. (Note that this generalizes the condition under which depth-first branch-and-bound search backtracks, since it backtracks as soon as all successors of the node on the top of the stack have an f-cost greater than the upper bound.) When beam-stack search backtracks, it removes from the top of the beam stack consecutive items with an $f_{\max}$ greater than or equal to U, the upper bound. Note that the layer associated with the item left on the top of the beam stack after beam-stack search backtracks corresponds to the deepest layer that contains some node(s) with one or more inadmissibly pruned successors.

Every time beam-stack search backtracks to a layer, it forces the search beam to admit a different set of successor nodes by systematically shifting the range of $[f_{\min}, f_{\max})$ stored in the beam-stack item associated with the layer. That is, the algorithm uses the current $f_{\max}$ as the new $f_{\min}$, and the upper bound U as the new $f_{\max}$ when it backtracks to a layer. The search is continued by re-expanding all nodes in this layer. Note that this requires re-expanding the same nodes that were expanded the last time this layer was visited, in order to generate successor nodes that may have been inadmissibly pruned in the previous visit.

A layer is said to be *backtracking-complete* if and only if the $f_{\max}$ of its beam-stack item is greater than or equal to the upper bound U after all nodes in the layer have been expanded. This indicates that no successor node with an f-cost greater than or equal to $f_{\min}$ but less than U has been pruned since the last time the algorithm backtracked to that layer. Therefore, all successor nodes with an f-cost within range $[0, U)$ must have been generated for the layer.

Beam-stack search does not stop as soon as it finds a solution, but continues to search for improved solutions. It terminates when the beam stack is empty, which means that all layers are backtracking-complete. It is easily proved that the best solution found must be optimal. Thus beam-stack search is an anytime algorithm that finds an initial solution relatively quickly, and continues to search for improved solutions until convergence to an optimal solution. (Note that each time beam-stack search finds an improved solution, it updates its upper bound.)

So far we have assumed that successor nodes all have different f-costs, which allows use of their f-costs to determine the order in which to prune nodes when memory is full. An important advantage of ordering nodes by f-cost is that beam-stack search explores nodes with the least f-cost first, which means it explores the most promising nodes first. But if some nodes have the same f-cost, a tie-breaking rule is needed to impose a total ordering on nodes. There are many possibilities. Beam-stack search can break ties based on the state encoding of a node, which is guaranteed to be unique. Or domain-specific information can be used. For multiple sequence alignment, a total ordering can be based on the coordinate of a node in an n-dimensional hypercube (Hohwald, Thayer, & Korf 2003), where n is the number of sequences being aligned. For STRIPS planning problems, our implementation of beam-stack search uses the admissible *max-pair heuristic* (Haslum & Geffner 2000) as the f-cost, and the *additive heuristic* (Bonet & Geffner 2001) to break ties. Note that beam-stack search does not require all ties to be broken. It allows some unbroken ties, as long as the beam width is greater than the number of ties in a layer.

It is interesting to note that beam-stack search includes both breadth-first branch-and-bound search and depth-first branch-and-bound search as special cases. When the beam width is greater than or equal to the size of the largest layer, beam-stack search is equivalent to breadth-first branch-and-bound search, and no backtracking occurs. When the beam width is one, beam-stack search is equivalent to depth-first search branch-and-bound search. In all other cases, it employs a hybrid search strategy that combines breadth-first and depth-first branch-and-bound search and offers a flexible tradeoff between available memory and the time overhead of backtracking. It is also interesting to note that the first phase of beam-stack search, before backtracking begins, is equivalent to traditional beam search, and often finds an initial solution very quickly. From that point on, beam-stack search is an anytime algorithm that finds a sequence of improved solutions before converging to optimality.

Theorem 1 *Beam-stack search is guaranteed to find an optimal solution if one exists.*

Proof: First we show that beam-stack search always terminates. To show this, we make the following two assumptions; each operator costs at least δ, which is a positive constant, and the number of applicable operators at any state is finite. Let U be the upper bound on the cost of an optimal solution. Because each operator costs at least δ, the maximum length of a path is at most $\lceil \frac{U}{\delta} \rceil$, which also bounds the maximum depth of the beam stack.

Let w be the beam width and let b be the maximum number of applicable operators at any state. Every time beam-stack search backtracks to a layer, it generates a new set of successor nodes that have been inadmissibly pruned and inserts no more than w of them into the Open list for the next layer. Thus, in order to fully generate all successors of a layer, the algorithm backtracks to the layer at most b times. Since the maximum depth of the beam stack is bounded by $\lceil \frac{U}{\delta} \rceil$ and in the worst case all layers (except for the deepest layer) have some inadmissibly pruned successors, it follows that the maximum number of backtrackings in beam-stack search is bounded by $O(b^{\lceil \frac{U}{\delta} \rceil})$, which is finite. The algorithm is guaranteed to terminate after this many backtrackings, because the beam stack must be empty by then.

Next we show that beam-stack search always terminates with an optimal solution. Recall that beam-stack search systematically enumerates all successors of a layer by shifting the range of f-costs of nodes that are admitted in the next layer. Only the deepest, consecutive backtracking-complete layers are removed from the top of the beam stack. This ensures that no path will be ignored forever in beam-stack search, unless it contains

1. a node with an f-cost greater than or equal to U, or
2. a node to which a lower-cost path has already been found.

Because in either case, the path is sub-optimal and can be safely pruned without affecting the optimality of the solution found, it follows that by the time beam-stack search finishes enumerating all (admissible) paths, as indicated by an empty beam stack, it must have found an optimal solution. □

Memory efficiency

The memory complexity of beam-stack search, like that of beam search, is dw, where d is the depth of the search and w is the beam width. For any fixed w, this is linear memory complexity. But the deeper the search, the smaller the beam width w must be, in order for the stored nodes in all layers to fit in a fixed amount of available memory.

Zhou and Hansen (2004) describe an implementation of breadth-first branch-and-bound search that uses divide-and-conquer solution reconstruction to reduce memory requirements, and a related implementation of beam search, called *divide-and-conquer beam search*, that achieves memory reduction in a similar way. The memory complexity of divide-and-conquer beam search is $4w$, which is independent of the depth of the search. This allows much larger beam widths, and, experimental results show, much stronger performance.

We begin this section with a brief review of divide-and-conquer beam search. Then we describe how the same divide-and-conquer technique can be used to improve the memory efficiency of beam-stack search. Because this allows much wider beam widths, it significantly improves the performance of beam-stack search and reduces the amount of expensive backtracking. We also describe how to use external memory to improve the time efficiency of divide-and-conquer beam-stack search.

Divide-and-conquer beam search

Divide-and-conquer solution reconstruction is a technique for reducing the memory requirements of best-first (or breadth-first) graph search without incurring (significant) node re-expansion overhead. Introduced to the heuristic search community by Korf (1999), several variations have since been developed (Korf & Zhang 2000; Zhou & Hansen 2003a; 2003b; 2004). The strategy is based on the recognition that it is not necessary to store all explored nodes in memory in order to perform duplicate detection, that is, in order to detect when any newly-generated node is a duplicate of an already-explored node. Often, it is only necessary to store nodes that are on or near the search frontier. This allows nodes in the search interior to be removed from memory. But since removing interior search nodes from memory prevents recovery of the solution path by the traditional *traceback method* of tracing pointers backwards from the goal node to the start node, search algorithms that use this memory-saving technique rely on a divide-and-conquer technique of solution recovery. Each node stores information about an intermediate node along the best path from the start node. 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 original 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.

Algorithms that use this memory-saving technique must address two key issues; how to perform duplicate detection, and how to recover a solution path. Different algorithms address these issues differently. Zhou and Hansen (2004) describe a simple method of duplicate detection for algorithms that use a breadth-first search strategy, called *layered duplicate detection*. Because a breadth-first search graph divides into layers, one for each depth, and all the nodes in one layer are expanded before considering nodes in the next layer, they show that storing only the previous layer, the currently-expanding layer, and the next layer is sufficient to prevent re-generation of closed nodes in undirected graphs. In directed graphs, the same technique guarantees that the number of times a node can be re-generated is at most linear in the depth of an optimal path.

To allow divide-and-conquer solution reconstruction, Zhou and Hansen (2004) describe a technique in which each node (past the midpoint) stores a pointer to an intermediate node, called a *relay node*, that is retained in memory. (Nodes that come before the midpoint, store a pointer to the start node.) For simplicity, all relay nodes are stored in the same layer, which is called the *relay layer* of the search graph. It could be the middle layer of the search graph, although making it the $3/4$ layer is usually more efficient, since that layer is usually smaller. As a result, divide-and-conquer beam search stores four layers; the currently-expanding layer, its successor layer, its previous layer, and the relay layer.

Although BFBnB can use the technique of divide-and-conquer solution reconstruction to significantly reduce its memory requirements, it can still run out of memory if the number of nodes in any layer becomes too large. If the largest layer (or adjacent layers) in a breadth-first search graph does not fit in memory, one way to handle this is to use beam search. Instead of considering all nodes in a layer, a beam-search variant of BFBnB search considers the most promising nodes until memory is full (or reaches a predetermined bound). At that point, the algorithm recovers memory by pruning the least-promising nodes from memory. 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 brings some significant advantages. First, it allows a beam-search algorithm to use a much larger beam width in order to improve performance. The memory complexity of divide-and-conquer beam search is $4w$ instead of dw, that is, it is constant instead of linear. Whereas conventional beam search must use a smaller beam width for deeper searches, in order to ensure that it does not run out of memory, divide-and-conquer beam search can use the same beam width no matter how deep it searches. Second, once the beam-search algorithm finds an initial, sub-optimal goal node, it performs divide-and-conquer solution reconstruction. In recursively solving subproblems of the original problem, it often improves the quality of the solution, as follows: given two nodes along a solution path that is sub-optimal, it often finds a shorter path between the two nodes, improving the overall solution.

Zhou and Hansen (2004) show that divide-and-conquer beam search (DCBS) outperforms weighted A* in solving STRIPS planning problems. But like beam search, DCBS is incomplete – not only is it not guaranteed to find an optimal solution, it is not guaranteed to find any solution at all.

Divide-and-conquer beam-stack search

In most respects, the divide-and-conquer technique can be combined with beam-stack search in the same way as with beam search, creating an algorithm that we call *divide-and-conquer beam-stack search*. But there are two complications to consider. The first results from the combination of the divide-and-conquer technique with backtracking. Since divide-and-conquer beam-stack search (DCBSS) only keeps four layers of the search graph in memory, the layer to which it backtracks may not be in memory. In this case, the algorithm must recover the missing layer. It is possible to recover any previously-explored layer by using the information stored in the beam stack. To recover a missing layer, the algorithm goes back to the start node and generates successor nodes at each layer according to the layer's beam-stack item, until all nodes in the layer preceding the missing layer have been expanded – which recreates the missing layer. Then the algorithm follows the backtracking procedure described previously.

Another closely-related question is when to perform divide-and-conquer solution reconstruction, and how to continue the search afterwards. If the algorithm wants to use all available memory for solution reconstruction, all search information accumulated before solution reconstruction may be lost. In this case, DCBSS can use a technique we call *delayed solution reconstruction*, in which the algorithm keeps track of the best goal node expanded so far, but does not reconstruct a solution until it begins to backtrack. Divide-and-conquer solution reconstruction is delayed until then since backtracking will delete search layers anyway. In the meantime, the algorithm uses the improved upper bound to reduce the number of node expansions before beginning to backtrack. Because nodes at the same layer often have similar g-costs, the improved upper bound often makes it possible to prune many or most nodes in the current or subsequent layer. Thus the time between expanding a goal node and beginning to backtrack is often short.

Note that delayed solution reconstruction is only necessary for planning problems with non-unit action costs, in which the g-cost of a node is not the same as its layer index (i.e., depth). For problems with unit (or uniform) edge costs, the algorithm can reconstruct a solution each time a goal node is generated. For these problems, all nodes at the same layer have the same g-cost. So when a goal node is generated, it must be the node with the minimum g-cost in this layer or any successor layer. Thus, the algorithm can reconstruct a solution immediately after generating a goal node, because it is impossible to find any better solution without

```
**Procedure** pruneLayer (Integer $\ell$)
Keep ← the best $w$ nodes $\in$ Open[$\ell$]
Prune ← $\{n \mid n \in Open[\ell] \wedge n \notin Keep\}$
beam-stack.top().$f_{\max} \leftarrow \min \{f(n) \mid n \in Prune\}$
for each $n \in$ Prune do /* inadmissible pruning */
    Open[$\ell$] ← Open[$\ell$] $\setminus \{n\}$
    delete $n$
end for

Function search (Node start, Node goal, Real U, Integer relay)
Open[0] ← {start}
Open[1] ← $\emptyset$
Closed[0] ← $\emptyset$
best-goal ← nil
$\ell \leftarrow 0$ /* $\ell$ = index of layer */
while Open[$\ell$] $\neq \emptyset$ or Open[$\ell + 1$] $\neq \emptyset$ do
    while Open[$\ell$] $\neq \emptyset$ do
        node ← $\arg\min_n \{ f(n) \mid n \in Open[\ell]\}$
        Open[$\ell$] ← Open[$\ell$] $\setminus$ {node}
        Closed[$\ell$] ← Closed[$\ell$] $\cup$ {node}
        if node is goal then
            U ← g(node)
            best-goal ← node
        node.generateAdmittedSuccessors(beam-stack.top())
        if layerSize($\ell + 1$) > w then pruneLayer($\ell + 1$)
    end while
    if $1 < \ell \leq relay$ or $\ell > relay + 1$ then
        for each $n \in$ Closed[$\ell - 1$] do /* delete previous layer */
            Closed[$\ell - 1$] ← Closed[$\ell - 1$] $\setminus \{n\}$
            delete $n$
        end for
    $\ell \leftarrow \ell + 1$ /* move on to next layer */
    Open[$\ell + 1$] ← $\emptyset$
    Closed[$\ell$] ← $\emptyset$
    beam-stack.push($[0, U)$) /* new beam-stack item */
end while
if best-goal $\neq$ nil then /* delayed solution reconstruction */
    return solutionReconstruction(best-goal)
else
    return nil

Algorithm DCBSS(Node start, Node goal, Real U, Integer relay)
beam-stack ← $\emptyset$
beam-stack.push($[0, U)$) /* initialize beam stack */
$\pi^* \leftarrow$ nil /* initialize optimal solution path */
while beam-stack.top() $\neq$ nil do
    $\pi \leftarrow$ search(start, goal, U, relay) /* $\pi$ = solution path */
    if $\pi \neq$ nil then
        $\pi^* \leftarrow \pi$
        $U \leftarrow \pi$.getCost()
    while beam-stack.top().$f_{max} \geq U$ do
        beam-stack.pop()
    end while
    if beam-stack.isEmpty() then return $\pi^*$
    beam-stack.top().$f_{\min}$ ← beam-stack.top().$f_{\max}$
    beam-stack.top().$f_{\max} \leftarrow U$
end while
```

Figure 1: Pseudocode for divide-and-conquer beam-stack search. The Open and Closed lists are indexed by the layer of the breadth-first search graph, and sorted by f-cost within each layer.

backtracking. Since we do not assume that problems have unit edge costs in this paper, we use delayed solution reconstruction in the pseudocode in Figure 1. As mentioned previously, a benefit of performing divide-and-conquer solution reconstruction each time a new goal node is found is that divide-and-conquer solution reconstruction often improves the solution by finding better solutions to subproblems, further improving the upper bound. Details of divide-and-conquer solution reconstruction are not shown in the pseudocode. For these details, see (Zhou & Hansen 2004).

Using external memory

Instead of discarding previous layers and regenerating them as needed, we can create an external-memory version of DCBSS that copies previously-explored layers to disk and copies them back to internal memory as needed, when DCBSS backtracks to a missing layer. This can reduce the time overhead of the algorithm significantly. Note that if DCBSS backtracks to a layer that is missing from internal memory, both this layer and its immediate previous layer must be copied from disk, since the previous layer is needed for duplicate detection.

One issue to be careful about when using pointers and relay nodes is that once a relay node is saved to disk, all pointers to that relay node are no longer valid. The way we implement this is that when we copy nodes to disk, we write all state information about the relay node, instead of its pointer. Later, when we copy layers from disk back into internal memory, we copy the relay layer first. Once all relay nodes are inserted in the hash table that contains nodes in memory, we use the state information about the relay node to index the hash table and extract the current pointer, when copying nodes in other layers into internal memory. (Note that only nodes that come after the relay layer need to worry about this. Nodes that come before have a *static* pointer to the start node.)

Beam-stack iterative-deepening A*

Recall that beam-stack search uses an upper bound on the cost of an optimal solution to prune the search space. So far, we have assumed the upper bound corresponds to the cost of an actual solution. But if it is difficult to find a good initial upper bound, it is possible to define a version of beam-stack search that does not need a previously-computed upper bound. Instead, it uses an iterative-deepening strategy to avoid expanding nodes that have an f-cost greater than a hypothetical upper bound. The algorithm first runs beam-stack search using the f-cost of the start node as an upper bound. If no solution is found, it increases the upper bound by one (or to the lowest f-cost of any unexpanded nodes from the last iteration) and repeats the search. This continues until a solution is found. Because of the similarity of this algorithm to depth-first iterative-deepening A* (DFIDA*) (Korf 1985), as well as to the more recent breadth-first iterative-deepening A* (BFIDA*) (Zhou & Hansen 2004), we call it *beam-stack iterative-deepening A** (BSIDA*). In fact, it can be viewed as a generalization of DFIDA* and BFIDA* that includes each as a special case. BSIDA* with a beam width of one corresponds to DFIDA*, and BSIDA* with an unlimited beam width corresponds to BFIDA*. Intermediate beam widths create a spectrum of algorithms that use a hybrid strategy that combines elements of breadth-first search (expanding nodes on a layer-by-layer basis) and depth-first search (stack-based backtracking).

Computational results

We tested the performance of divide-and-conquer beam-stack search in solving problems from eight unit-cost planning domains from the Planning Competition that is hosted by the ICAPS conference series (Long & Fox 2003). 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. As an admissible heuristic function, we used the *max-pair heuristic* (Haslum & Geffner 2000), and to break ties, we used the *additive heuristic* (Bonet & Geffner 2001). All experiments were performed on a Pentium IV 3.2 GHz processer with 1 GB of RAM and 512 KB of L2 cache.

Divide-and-conquer beam-stack search is able to find provably optimal solutions for problem instances that cannot be solved optimally by the best previous algorithm, divide-and-conquer BFBnB (Zhou & Hansen 2004). For example, an external-memory version of DCBSS proves that a 30-step solution to an instance of *logistics-9* is optimal after expanding some 261 million nodes in 6739 CPU seconds, and it proves that a 45-step solution to an instance of *Elevator-14* is optimal after expanding some 984 million nodes in 55,243 CPU seconds. However, in the experiments reported in the rest of this section, we consider the performance of DCBSS in solving the most difficult problem instances that divide-and-conquer BFBnB can also solve. This makes it easier to analyze the tradeoff between memory and time that backtracking allows.

Memory-time tradeoff When available memory is exhausted, beam-stack search relies on inadmissible pruning and backtracking to continue the search without exceeding the memory limit. But inadmissible pruning and backtracking lead to node re-expansions, for two reasons. When the beam width is not large enough to store all nodes in a layer, not all duplicate nodes are detected. And when the search backtracks to a layer, nodes in that layer are re-expanded in order to generate successors that were inadmissibly pruned in the previous visit. The smaller the beam width, and the more extensive backtracking, the more node re-expansions there will be. Thus there is a tradeoff between available memory and time overhead for node re-expansions.

Table 1 illustrates the memory-time tradeoff in eight different planning domains from the Planning Competition. For each domain, the problem instance is among the largest that can be solved optimally using one gigabyte of RAM by breadth-first heuristic search (Zhou & Hansen 2004) (i.e., BFBnB with divide-and-conquer solution reconstruction). Table 1 was created by running BFBnB to see how much memory it uses, and then running divide-and-conquer beam-

		No backtracking		75% Memory		50% Memory		25% Memory	
Problem	Len	Stored	Exp	Stored	Exp	Stored	Exp	Stored	Exp
logistics-6	25	96,505	301,173	72,379	734,170	48,253	1,608,139	24,127	12,789,413
blocks-14	38	79,649	258,098	59,737	478,797	39,825	593,377	19,913	991,460
gripper-6	41	580,009	2,124,172	435,007	-	290,005	-	145,003	-
satellite-6	20	1,592,097	2,644,616	1,194,073	3,928,235	796,049	10,179,757	398,025	43,281,817
elevator-11	37	977,554	3,975,419	733,166	223,579,495	488,777	-	244,389	-
depots-3	27	3,185,570	5,233,623	2,389,178	9,255,319	1,592,785	12,343,501	796,393	29,922,080
driverlog-10	17	2,255,515	4,071,392	1,691,636	6,704,007	1,127,758	8,995,166	563,879	22,005,534
freecell-4	27	3,130,328	9,641,354	2,347,746	27,530,551	1,565,164	62,985,231	782,582	261,093,196

Table 1: Memory-time tradeoff for divide-and-conquer beam-stack search (using an upper bound computed by divide-and-conquer beam search) on STRIPS planning problems. Columns show optimal solution length (Len); peak number of nodes stored (Stored); and number of node expansions (Exp). The $-$ symbol indicates that divide-and-conquer beam-stack search could not solve the problem after 24 hours of CPU time under the given memory constraint.

			No external memory		External memory		
Problem	Len	Stored in RAM	Exp	Secs	Stored on disk	Exp	Secs
logistics-6	25	48,253	1,608,139	16.4	127,451	770,187	13.8
blocks-14	38	39,825	593,377	17.3	100,528	363,199	17.4
gripper-6	41	290,005	-	-	1,365,890	775,639,416	6,907.7
satellite-6	20	796,049	10,179,757	448.9	1,333,076	5,210,030	225.9
elevator-11	37	488,777	-	-	2,213,275	129,645,271	3,873.0
depots-3	27	1,592,785	12,343,501	232.7	2,256,128	7,460,470	150.4
driverlog-10	17	1,127,758	8,995,166	306.9	1,922,041	6,635,998	233.3
freecell-4	27	1,565,164	62,985,231	4,767.1	3,640,367	30,426,421	2,312.9

Table 2: Comparison of divide-and-conquer beam-stack search with and without using external memory, and under the same internal-memory constraints, on STRIPS planning problems. Columns show optimal solution length (Len); peak number of nodes stored in internal memory (Stored in RAM); number of node expansions (Exp); and running time in CPU seconds (Secs). For the external-memory algorithm, the peak number of nodes stored on disk is shown in the column labelled "Stored on disk". The $-$ symbol indicates that the internal-memory algorithm could not solve the problem after 24 hours of CPU time under a given memory constraint.

stack search with artificial limits on available memory equal to $3/4$ of the memory used by BFBnB, half of the memory used by BFBnB, and $1/4$ of the memory used by BFBnB. As smaller limits on available memory force the beam width to be smaller, Table 1 shows that the number of node re-expansions (and therefore the time overhead) increases. Although the tradeoff can be seen in every domain, the sharpness of the tradeoff is domain-dependent, since it depends on the number of duplicate paths in the search space of the domain. The Gripper domain has the most duplicate paths, and cannot be solved optimally in 24 hours by divide-and-conquer beam-stack search within these artificial memory constraints – unless external memory is used to reduce the time overhead of backtracking, as described next.

External-memory algorithm Although most of the time overhead of divide-and-conquer beam-stack search is due to node re-expansions, some of it results from combining backtracking with the divide-and-conquer memory-saving technique. When layers are removed from memory as part of the divide-and-conquer algorithm, they must be re-generated when the algorithm backtracks to them. To reduce this overhead, we suggested that layers can be copied to disk when they are removed from internal memory, and then copied back from disk to internal memory when they are needed again. Table 2 shows how much this can reduce the time overhead of divide-and-conquer beam-stack search. For each of the problem instances in the table, we first show results for divide-and-conquer beam-stack search using half the memory it would need to converge to optimality without backtracking. (These results are the same as those reported in Table 1.) Restricting memory in this way forces the algorithm to backtrack. Then we show results for the external-memory version of divide-and-conquer beam-stack search, given the same limit on internal memory. (The column labelled "Stored in RAM" shows the limit on the number of nodes stored in internal memory.) The results show that using disk can significantly reduce the time overhead of the algorithm. For example, whereas *gripper-6* and *elevator-11* could previously not be solved optimally in 24 hours of CPU time, now they are solved optimally in less than 2 hours and less than 65 minutes, respectively.

Iterative-deepening algorithm It is well-known that the performance of depth-first iterative-deepening A* (DFIDA*) can be improved by using extra memory for a transposition table (Sen & Bagchi 1989; Reinefeld & Marsland 1994). The transposition table is used to detect and eliminate duplicate nodes, which helps reduce the number of node re-expansions. We compared beam-stack iterative-deepening A* to Haslum's (2000) implementation of DFIDA* using a transposition table, to see which uses the same amount of extra memory more effectively. (The transposition table implemented by Haslum is similar to the

Problem	Len	Stored		DFIDA*	BSIDA*
logistics-4	20	1,662	Exp	55,112,968	16,808
			Secs	568.0	0.2
blocks-12	34	3,683	Exp	207,684	33,256
			Secs	46.0	1.0
gripper-2	17	985	Exp	7,992,970	10,329
			Secs	119.4	0.1
satellite-3	11	520	Exp	24,026	6,144
			Secs	0.5	0.2
driverlog-7	13	31,332	Exp	4,945,377	454,559
			Secs	417.3	11.8
depots-2	15	1,532	Exp	332,336	8,060
			Secs	35.4	0.2

Table 3: Comparison of DFIDA* (using transposition table) and BSIDA* under the same memory constraints on STRIPS planning problems. The table shows optimal solution length (Len); peak number of nodes stored by DFIDA*, which is the memory limit for BSIDA* (Stored); number of node expansions (Exp); and running time in CPU seconds (Secs).

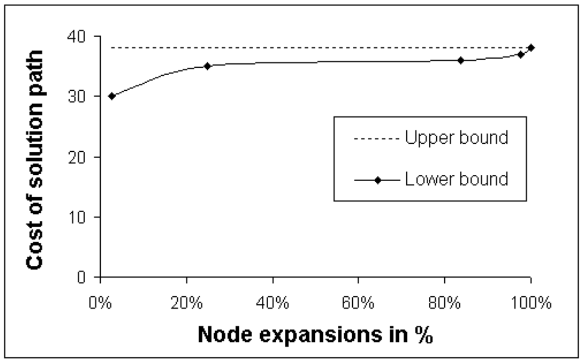

Figure 2: Convergence of bounds for divide-and-conquer beam-stack search in solving *blocks-14*. The upper bound is the cost of the best solution found so far; an optimal solution is found quickly for this problem. The lower bound is the least f-cost of any generated but unexpanded node; it increases in a few discrete steps because this is a unit edge cost problem with many ties.

one used by MREC (Sen & Bagchi 1989).) To obtain the results in Table 3, we first ran DFIDA* with a transposition table, and recorded the number of nodes stored in the transposition table. Then we ran BSIDA* with an artificial memory constraint equal to the number of nodes stored in the transposition table of DFIDA*. (This is actually a little less memory than used by DFIDA*, since it does not include the stack memory used by DFIDA*.)

Table 3 shows that BSIDA* consistently outperforms Haslum's implementation of DFIDA* that uses a transposition table, and often dramatically outperforms it – without using more memory. Recall that BSIDA* keeps only four layers of the search graph in memory. Disregarding the relay layer, the other three layers "move" together with the search frontier, and keep the most recent part of the explored state space in memory for duplicate elimination. This seems responsible for the improved performance since it uses available memory more effectively for duplicate elimination. The problems shown in Table 3 are the largest that DFIDA* can solve in each domain. (They are smaller than problems that can be solved by beam-stack search.)

Anytime performance Beam-stack search and divide-and-conquer beam-stack search are anytime algorithms that find a solution relatively quickly (since their initial phase is beam search), and then continue to improve the solution until convergence to optimality. A solution can be improved in two ways. One is to find a solution that has lower cost. The other is to improve the error bound, which is the difference between the cost of the best solution found so far (an upper bound) and the least f-cost of any generated but unexpanded node (which is a lower bound on optimal solution cost).

Figure 2 shows how these upper and lower bounds converge in solving the *blocks-14* problem. For this problem, an optimal solution is usually found quickly, but it takes a great deal of additional search effort to prove the solution is optimal. As reported in the paper that introduced divide-and-conquer beam search (Zhou & Hansen 2004), it is very effective for domain-independent STRIPS planning. In particular, even if the first goal node found by beam search is not optimal, the process of divide-and-conquer solution reconstruction usually improves it by solving subproblems optimally, and often results in an optimal solution. For the problem in Figure 2, divide-and-conquer beam search finds a solution after expanding less than 3% of the nodes expanded by beam-stack search. Figure 2 shows how the lower bound (and thus the error bound) continues to improve after the first solution is found, until convergence.

Conclusion

We have described an approach to integrating systematic backtracking with beam search, and showed how to combine this approach with a memory-saving technique that uses divide-and-conquer solution reconstruction. Although beam-stack search itself is not difficult to implement, divide-and-conquer beam-stack search presents more of a challenge. But when implemented, it provides a very effective search algorithm with many attractive properties.

We demonstrated the advantages of this approach in solving domain-independent STRIPS planning problems. It finds optimal solutions for planning problems that cannot be solved optimally by any other method (within reasonable time limits). It also offers a flexible tradeoff between memory and time, and creates an anytime algorithm that finds a good solution relatively quickly, like beam search, and then continues to search for improved solutions until convergence to an optimal solution.

Acknowledgements

We thank the anonymous reviewers for helpful comments. This work was supported in part by NSF grant IIS-9984952 and NASA grant NAG-2-1463.

References

Bisiani, R. 1987. Beam search. In Shapiro, S., ed., *Encyclopedia of Articial Intelligence*. John Wiley and Sons. 56–58.

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

Ginsberg, M., and Harvey, W. 1992. Iterative broadening. *Artificial Intelligence* 55:367–383.

Habenicht, I., and Monch, L. 2002. A finite-capacity beam-search-algorithm for production scheduling in semiconductor manufacturing. In *Proceedings of the 2002 Winter Simulation Conference*, 1406–1413.

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

Hohwald, H.; Thayer, I.; and Korf, R. 2003. Comparing best-first search and dynamic programming for optimal multiple sequence alignment. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-2003)*, 1239–1245.

Huang, X.; Acero, A.; and Hon, H. 2001. *Spoken language processing: A guide to theory, algorithm, and system development*. Prentice Hall.

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

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

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

Long, D., and Fox, M. 2003. The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research* 20:1–59.

Provost, F. 1993. Iterative weakening: Optimal and near-optimal policies for the selection of search bias. In *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI-93)*, 769–775.

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

Rich, E., and Knight, K. 1991. *Artificial Intelligence*. McGraw-Hill.

Sen, A., and Bagchi, A. 1989. Fast recursive formulations for BFS that allow controlled use of memory. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI-89)*, 297–302.

Zhang, W. 1998. Complete anytime beam search. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, 425–430.

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

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

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