Linear-Space Best-First Search: Summary of Results

Richard E. Korf*
Computer Science Department
University of California, Los Angeles
Los Angeles, Ca. 90024
korf@cs.ucla.edu

Abstract

Best-first search is a general search algorithm that always expands next a frontier node of lowest cost. Its applicability, however, is limited by its exponential memory requirement. Iterative deepening, a previous approach to this problem, does not expand nodes in best-first order if the cost function can decrease along a path. We present a linear-space best-first search algorithm (RBFS) that always explores new nodes in best-first order, regardless of the cost function, and expands fewer nodes than iterative deepening with a non-decreasing cost function. On the sliding-tile puzzles, RBFS with a weighted evaluation function dramatically reduces computation time with only a small penalty in solution cost. In general, RBFS reduces the space complexity of best-first search from exponential to linear, at the cost of only a constant factor in time complexity in our experiments.

Introduction: Best-First Search

Best-first search is a very general heuristic search algorithm. It maintains an Open list of the frontier nodes of a partially expanded search graph, and a Closed list of the interior nodes. Every node has an associated cost value. At each cycle, an Open node of minimum cost is expanded, generating all of its children. The children are evaluated by the cost function, inserted into the Open list, and the parent is placed on the Closed list. Initially, the Open list contains just the initial node, and the algorithm terminates when a goal node is chosen for expansion.

If the cost of a node is its depth in the graph, then best-first search becomes breadth-first search. If the cost of node n is $g(n)$, the sum of the edge costs from the root to node n, then best-first search becomes Dijkstra's single-source shortest-path algorithm[Dijkstra 1959]. If the cost function is $f(n) = g(n) + h(n)$, where $h(n)$ is a heuristic estimate of the cost of reaching a goal from node n, then best-first search becomes the A* algorithm[Hart, Nilsson, & Raphael 1968].

Since best-first search stores all generated nodes in the Open or Closed lists, its space complexity is the same as its time complexity, which is typically exponential. Given the ratio of memory to processing speed on current computers, in practice best-first search exhausts the available memory on most machines in a matter of minutes, halting the algorithm.

Previous Work: Iterative Deepening

The memory limitation of best-first search was first addressed by iterative deepening[Korf 1985]. Iterative deepening performs a series of depth-first searches, pruning branches when their cost exceeds a threshold for that iteration. The initial threshold is the cost of the root node, and the threshold for each succeeding iteration is the minimum node cost that exceeded the previous threshold. Since iterative deepening is a depth-first algorithm, it only stores the nodes on the current search path, requiring space that is only linear in the search depth. As with best-first search, different cost functions produce different iterative deepening algorithms, including depth-first iterative deepening ($f(n) = depth(n)$) and iterative-deepening-A* ($f(n) = g(n) + h(n)$).

If $h(n)$ is consistent[Pearl 1984], all the cost functions described above are *monotonic*, in the sense that the cost of a child is always greater than or equal to the cost of its parent. With a monotonic cost function, the order in which nodes are first expanded by iterative deepening is best first. Many important cost functions are non-monotonic, however, such as $f(n) = g(n) + W \cdot h(n)$[Pohl 1970] with $W > 1$. The advantage of this cost function is that while it returns suboptimal solutions, it generates far fewer nodes than are required to find optimal solutions.

With a non-monotonic cost function, the cost of a

*This research was supported by an NSF Presidential Young Investigator Award, No. IRI-8552925, and a grant from Rockwell International. Thanks to Valerie Aylett for drawing the graphs and figures, to Cindy Mason, Joe Pemberton, and Ed Purcell for their comments on an early draft of this manuscript, and to Stuart Russell for discussions on related work.

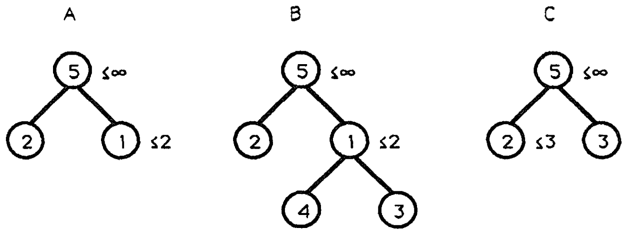

Figure 1: SRBFS with non-monotonic cost function

child can be less than the cost of its parent, and iterative deepening no longer expands nodes in best-first order. For example, consider the tree fragment in Figure 1B, ignoring the caption and inequalities for now, where the numbers in the nodes represent their costs. A best-first search would expand these nodes in the order 5, 1, 2. With iterative deepening, the initial threshold would be the cost of the root node, 5. After generating the left child of the root, node 2, iterative deepening would expand all descendents of node 2 whose costs did not exceed the threshold of 5, in depth-first order, before expanding node 1. Even if all the children of a node were generated at once, and ordered by their cost values, so that node 1 was expanded before node 2, iterative deepening would explore the subtrees below nodes 4 and 3 before expanding node 2. The real problem here is that while searching nodes whose costs are less than the current threshold, iterative deepening ignores the information in the values of those nodes, and proceeds strictly depth first.

Recursive Best-First Search

Recursive best-first search (RBFS) is a linear-space algorithm that always expands nodes in best-first order, even with a non-monotonic cost function. For pedagogical reasons, we first present a simple version of the algorithm (SRBFS), and then consider the more efficient full algorithm (RBFS). These algorithms first appeared in [Korf 1991a], the main results were described in [Korf 1991b], and a full treatment appears in [Korf 1991c], including proofs of all the theorems.

Simple Recursive Best-First Search

While iterative-deepening uses a global cost threshold, Simple Recursive Best-First Search (SRBFS) uses a local cost threshold for each recursive call. It takes two arguments, a node and an upper bound on cost, and explores the subtree below the node as long as it contains frontier nodes whose costs do not exceed the upper bound. It then returns the minimum cost of the frontier nodes of the explored subtree. Figure 1 shows how SRBFS searches the tree in Figure 1B in best-first order. The initial call on the root is made with an upper bound of infinity. We expand the root, and compute the costs of the children as shown in Figure 1A. Since the right child has the lower cost, we recursively call SRBFS on the right child. The best Open nodes in the tree will be descendents of the right child as long as their costs do not exceed the value of the left child. Thus, the recursive call on the right child is made with an upper bound equal to the value of its best (only) brother, 2. SRBFS expands the right child, and evaluates the grandchildren, as shown in Figure 1B. Since the values of both grandchildren, 4 and 3, exceed the upper bound on their parent, 2, the recursive call terminates. It returns as its result the minimum value of its children, 3. This backed-up value of 3 is stored as the new value of the right child (figure 1C), indicating that the lowest-cost Open node below this node has a cost of 3. A recursive call is then made on the new best child, the left one, with an upper bound equal to 3, which is the new value of its best brother, the right child. In general, the upper bound on a child node is equal to the minimum of the upper bound on its parent, and the current value of its lowest-cost brother.

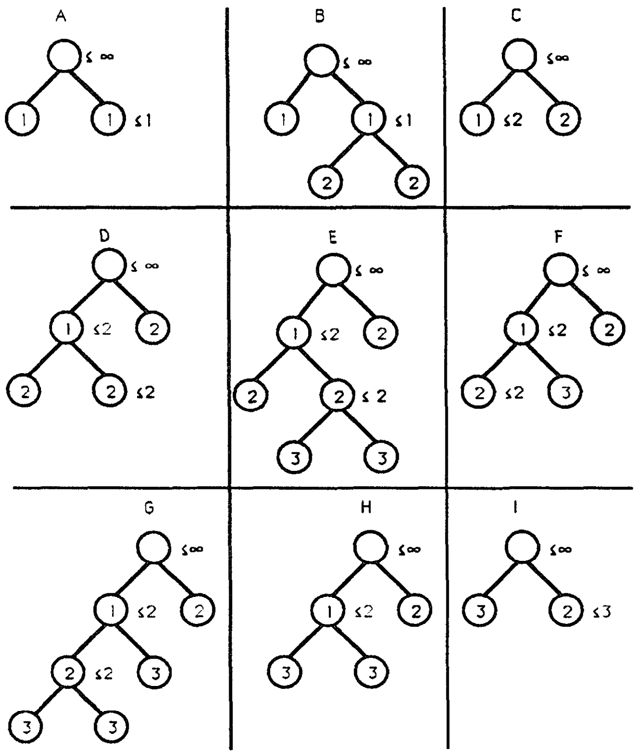

Figure 2: SRBFS example with cost equal to depth

Figure 2 shows a more extensive example of SRBFS. In this case, the cost function is simply the depth of a node in the tree, corresponding to breadth-first search.

Initially, the *stored* value of a node, $F(n)$, equals its *static* value, $f(n)$. After a recursive call on the node returns, its stored value is equal to the minimum value of all frontier nodes in the subtree explored during the last call. SRBFS proceeds down a path until the stored values of all children of the last node expanded exceed

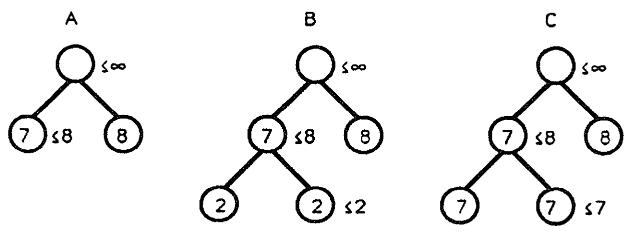

Figure 3: Inefficiency of SRBFS and its solution

the stored value of one of the nodes further up the tree. It then returns back up the tree, replacing parent values with the minimum of their children's values, until it reaches the better node, and then proceeds down that path. The algorithm is purely recursive with no side effects, resulting in very low overhead per node generation. In pseudo-code, the algorithm is as follows:

```
SRBFS (node: N, bound: B)
IF f(N)>B, RETURN f(N)
IF N is a goal, EXIT algorithm
IF N has no children, RETURN infinity
FOR each child Ni of N, F[i] := f(Ni)
sort Ni and F[i] in increasing order of F[i]
IF only one child, F[2] := infinity
WHILE (F[1] <= B)
   F[1] := SRBFS(N1, MIN(B, F[2]))
   insert N1 and F[1] in sorted order
return F[1]
```

Once a goal is reached, the actual solution path is on the recursion stack, and returning it involves simply recording the moves at each level. For simplicity, we omit this from the above description.

SRBFS expands nodes in best-first order, even if the cost function is non-monotonic. Unfortunately, however, SRBFS is inefficient. If we continue the example from figure 2, where cost is equal to depth, eventually we would reach the situation shown in figure 3A, for example, where the left child has been explored to depth 7 and the right child to depth 8. Next, a recursive call will be made on the left child, with an upper bound of 8, the value of its brother. The left child will be expanded, and its two children assigned their static values of 2, as shown in figure 3B. At this point, a recursive call will be made on the right grandchild with an upper bound of 2, the minimum of its parent's bound of 8, and its brother's value of 2. Thus, the right grandchild will be explored to depth 3, the left grandchild to depth 4, the right to depth 5, the left to depth 6, and the right to depth 7, before new ground can finally be broken by exploring the left grandchild to depth 8. Most of this work is redundant, since the left child has already been explored to depth 7.

Full Recursive Best-First Search

The way to avoid this inefficiency is for children to inherit their parent's values as their own, if the parent's values are greater than the children's values. In the above example, the children of the left child should inherit 7 as their stored value instead of 2, as shown in figure 3C. Then, the right grandchild would be explored immediately to depth 8 before exploring the left grandchild. However, we must distinguish this case from that in figure 1, where the fact that the child's value is smaller than its parent's value is due to non-monotonicity of the cost function, rather than previous expansion of the parent node. In that case, the children should not inherit their parent's value, but use their static values instead.

The distinction is made by comparing the stored value of a node, $F(n)$, to its static value, $f(n)$. If a node's stored value equals its static value, then it has never been expanded before, and its children's values should be set to their static values. If a node's stored value exceeds its static value, then it has been expanded before, and its stored value is the minimum of its children's last stored values. The stored value of such a node is thus a lower bound on the values of its children, and the values of the children should be set to the maximum of their parent's stored value and their own static values. A node's stored value cannot be less than its static value.

The full recursive best-first search algorithm (RBFS) takes three arguments: a node N, its stored value V, and an upper bound B. The top-level call to RBFS is made on the start node s, with a value equal to the static value of s, $f(s)$, and an upper bound of ∞. In pseudo-code, the algorithm is as follows:

```
RBFS (node: N, value: V, bound: B)
IF f(N)>B, return f(N)
IF N is a goal, EXIT algorithm
IF N has no children, RETURN infinity
FOR each child Ni of N,
   IF f(N)<V AND f(Ni)<V THEN F[i] := V
   ELSE F[i] := f(Ni)
sort Ni and F[i] in increasing order of F[i]
IF only one child, F[2] := infinity
WHILE (F[1] <= B)
   F[1] := RBFS(N1, F[1], MIN(B, F[2]))
   insert N1 and F[1] in sorted order
return F[1]
```

Like SRBFS, RBFS explores new nodes in best-first order, even with a non-monotonic cost function. Its advantage over SRBFS is that it is more efficient. While SRBFS expands all nodes in best-first order, RBFS only expands new nodes in best-first order, and expands previously expanded nodes in depth-first order.

If there are cycles in the graph, and cost does not always increase while traversing a cycle, as with a cost function such as $f(n) = h(n)$, RBFS must be modified to avoid infinite loops. In particular, each new node must be compared against the stack of nodes on the current path, and pruned if it is already on the stack.

Theoretical Results

We briefly present the main theoretical properties of SRBFS and RBFS. Space limitations preclude the formal statements and proofs of the theorems found in [Korf 1991c]. The most important result, and the most difficult to establish, is that both SRBFS and RBFS are best-first searches. In other words, the first time a node is expanded, its cost is less than or equal to that of all other nodes that have been generated, but not yet expanded so far. What distinguishes these algorithms from classical best-first search is that the space complexity of both SRBFS and RBFS is $O(bd)$, where b is the branching factor, and d is the maximum search depth. The reason is that at any given point, the recursion stack only contains the path to the best frontier node, plus the brothers of all nodes on that path. If we assume a constant branching factor, the space complexity is linear in the search depth.

While the time complexities of both algorithms are linear in the number of nodes generated, this number is heavily dependent on the particular cost function. In the worst case, all nodes have unique cost values, and the asymptotic time complexity of both algorithms is $O(b^{2d})$. This is the same as the worst-case time complexity of IDA* on a tree[Patrick, Almulla, & Newborn]. This scenario is somewhat unrealistic, however, since in order to maintain unique cost values in the face of an exponentially growing number of nodes, the number of bits used to represent the values must increase with each level of the tree.

As a more realistic example of time complexity, we examined the special case of a uniform tree where the cost of a node is its depth in the tree, corresponding to breadth-first search. In this case, the asymptotic time complexity of SRBFS is $O(x^d)$, where $x = (b+1+\sqrt{b^2+2b-3})/2$. For large values of b, this approaches $O((b+1)^d)$. The asymptotic time complexity of RBFS, however, is $O(b^d)$, showing that RBFS is asymptotically optimal in this case, but SRBFS is not.

Finally, for the important special case of a monotonic cost function, RBFS generates fewer nodes than iterative deepening, up to tie-breaking among nodes of equal cost. With a monotonic cost function, both algorithms expand all nodes of a given cost before expanding any nodes of greater cost. In iterative deepening, each new iteration expands nodes of greater cost. Between iterations, the search path collapses to just the root node, and the entire tree must be regenerated to find the nodes of next greater cost. For RBFS, we can similarly define an "iteration" as the interval of time when those nodes being expanded for the first time are all of the same cost. When the last node of a given cost is expanded, ending the current iteration, the recursion stack contains the path to that node, plus the brothers of all nodes on that path. Many of these brother nodes will have stored values equal to the next greater cost, and the subtrees below these nodes will be explored in the next iteration. Other nodes attached to this path may have greater costs associated with them, and will not be searched in the next iteration. Thus, while iterative deepening must regenerate the entire previous tree during each new iteration, RBFS will only explore the subtrees of brother nodes on the last path of the previous iteration whose stored values equal the upper bound for the next iteration. If nodes of similar cost are highly clustered in the tree, this will result in significant savings. Even in those situations where the entire tree must be explored in each iteration, as in the case of breadth-first search, RBFS avoids regenerating the last path of the previous iteration, although the savings is not significant in that case.

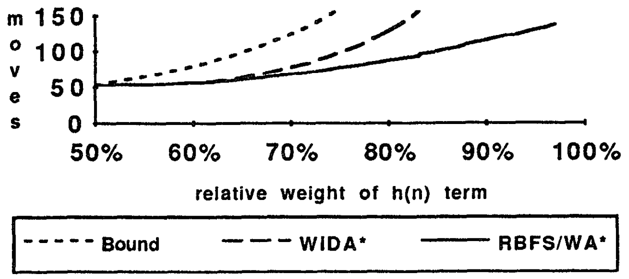

Figure 4: Solutions Lengths vs. Weight on $h(n)$

Experimental Results

We implemented RBFS on the Travelling Salesman Problem (TSP) and the sliding-tile puzzles. For TSP, using the monotonic A* cost function $f(n) = g(n) + h(n)$, with the minimum spanning tree heuristic, RBFS generates only one-sixth as many nodes as IDA*, while finding optimal solutions. Both algorithms, however, generate more nodes than depth-first branch-and-bound, another linear-space algorithm.

On the Eight Puzzle, with the A* cost function and the Manhattan Distance heuristic, RBFS finds optimal solutions while generating slightly fewer nodes than IDA*. Depth-first branch-and-bound doesn't work on this problem, since finding any solution is difficult.

In order to find sub-optimal solutions more quickly, we used the weighted non-monotonic cost function, $f(n) = g(n) + W \cdot h(n)$, with $W > 1$[Pohl 1970]. We ran three different algorithms on the Fifteen Puzzle with this function: weighted-A* (WA*), weighted-IDA* (WIDA*), and RBFS. The solutions returned are guaranteed to be within a factor of W of optimal[Davis, Bramanti-Gregor, & Wang 1989], but in practice all three algorithms produce significantly better solutions, as shown in figure 4. The horizontal axis is the relative weight on $h(n)$, or $100W/(W+1)$, while the vertical axis shows the solution lengths in moves. All data points are averages of the 100 problem instances in [Korf 1985]. The bottom line shows the solution lengths returned by WA* and RBFS. Since both algorithms are best-first searches, they produce solutions

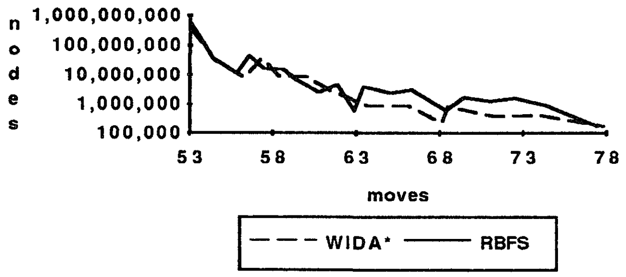

Figure 5: Nodes Generated vs. Solution Length

of the same average quality, with individual differences due to tie-breaking among nodes of equal cost. We were not able to run WA* with less than 75% of the weight on $h(n)$, since it exhausted the available memory of 100,000 nodes. The middle line shows that WIDA* produces significantly longer solutions as the the weight on $h(n)$ increases, since it explores nodes in depth-first order rather than best-first order. As the weight approaches 100%, the solution lengths returned by WIDA* grow to infinity. The top line shows the guaranteed bound on solution quality, which is W times the optimal solution length of 53 moves.

Figure 5 shows the average number of nodes generated as a function of solution length by RBFS and WIDA*, with relative weights on $h(n)$ of less than 75%. Small weights on $h(n)$ reduce the node generations by orders of magnitude, with only small increases in the resulting solution lengths. Neither algorithm dominates the other in this particular problem domain. While the utility of $f(n) = g(n) + W \cdot h(n)$ has previously been demonstrated on problems small enough to fit the search space in memory, such as the Eight Puzzle[Gaschnig 1979; Davis, Bramanti-Gregor, & Wang 1989], these experiments show that the benefit is even greater on larger problems.

Finally, we ran RBFS and WIDA* on 1000 different 5×5 Twenty-Four Puzzle problem instances with $W = 3$. RBFS returned solutions that averaged 169 moves, while generating an average of 93,891,942 nodes, compared to an average of 216 moves and 44,324,205 nodes for WIDA*. In both cases, however, the variation in nodes generated over individual problem instances was over six orders of magnitude. With $W = 3$, RBFS outperforms RTA*[Korf 1990a], heuristic subgoal search[Korf 1990b], and Stepping Stone[Ruby & Kibler 1989]. With $W < 3$, the running times of both RBFS and WIDA* precluded solving sufficient numbers of problems to draw meaningful conclusions.

An important feature of RBFS is that it can distinguish a node being expanded for the first time, from one being reexpanded, by comparing its stored value to its static value. If they are equal, it is a new node, and otherwise it has previously been expanded. This allows us to determine the overhead due to node regeneration. In our sliding-tile experiments, for a given value of W, the total nodes generated by RBFS were a constant times the number that would be generated by standard best-first search on a tree, in spite of enormous variation in the number of nodes generated in individual problem instances. For example, with $W = 3$ RBFS incurred an 85% node regeneration overhead, which remained constant over different problem instances of both the Fifteen and Twenty-Four Puzzles. The node regeneration overhead varied from a low of 20% with $W = 1$, to a high of 1671% with 61% of the weight on $h(n)$. The variation is due to the number of ties among nodes with the same $f(n)$ value. For efficiency, the weighted cost function is actually implemented by multiplying both $g(n)$ and $h(n)$ by relatively prime integers W_g and W_h, respectively, where $W = W_h/W_g$. With $W = W_g = W_h = 1$, many nodes with different $g(n)$ and $h(n)$ values have the same $f(n)$ value, whereas with $W_g = 39$ and $W_h = 61$, most nodes with the same $f(n)$ value have the same $g(n)$ and $h(n)$ values as well, resulting in far fewer ties among $f(n)$ values.

In terms of time per node generation, RBFS is about 29% slower than WIDA*, and about a factor of three faster than WA*. The reason that RBFS and WIDA* are faster than WA* is that they are purely recursive algorithms with no Open or Closed lists to maintain. The magnitude of these differences is due to the fact that node generation and evaluation is very efficient for the sliding-tile puzzles, and these differences would be smaller on more complex problems.

Related Work

In comparing RBFS to other memory-limited algorithms such as MREC[Sen & Bagchi 1989], MA*[Chakrabarti et al. 1989], DFS*[Rao, Kumar, & Korf 1991], IDA*-CR[Sarkar et al. 1991], MIDA*[Wah 1991], ITS[Mahanti et al. 1992], IE[Russell 1992], and SMA*[Russell 1992], the most important difference is that none of these other algorithms expand nodes in best-first order when the cost function is nonmonotonic. However, many of the techniques in these algorithms can be applied to RBFS as well, in order to reduce the node regeneration overhead.

Recently, we became aware of the Iterative Expansion (IE) algorithm[Russell 1992], which was developed independently. IE is virtually identical to SRBFS, except that a node always inherits its parent's cost if the parent's cost is greater than the child's cost. As a result, IE is not a best-first search with a non-monotonic cost function, but behaves the same as RBFS for the special case of a monotonic cost function. The performance of IE on our sliding-tile puzzle experiments should be very similar to that of WIDA*.

Some of the ideas in RBFS, IE, MA*, and MREC, in particular using the value of the next best brother as an upper bound, and backing up the minimum values of children to their parents, can be found in Bratko's formulation of best-first search[Bratko 1986], which uses exponential space, however.

Conclusions

RBFS is a linear-space algorithm that always expands new nodes in best-first order, even with a non-monotonic cost function. While its time complexity depends on the cost function, for the special case where cost is equal to depth, corresponding to breadth-first search, RBFS is asymptotically optimal, generating $O(b^d)$ nodes. With a monotonic cost function, it finds optimal solutions, while expanding fewer nodes than iterative-deepening. With a non-monotonic cost function on the sliding-tile puzzles, both RBFS and iterative deepening generate orders of magnitude fewer nodes than required to find optimal solutions, with only small increases in solution lengths. RBFS consistently finds shorter solutions than iterative deepening with the same cost function, but also generates more nodes in this domain. The number of nodes generated by RBFS was a constant multiple of the nodes that would be generated by standard best-first search on a tree, if sufficient memory were available to execute it. Thus, RBFS reduces the space complexity of best-first search from exponential to linear in general, while increasing the time complexity by only a constant factor in our experiments and analyses.

References

Bratko, I., *PROLOG: Programming for Artificial Intelligence*, Addison-Wesley, 1986, pp. 265-273.

Chakrabarti, P.P., S. Ghose, A. Acharya, and S.C. de Sarkar, Heuristic search in restricted memory, *Artificial Intelligence*, Vol. 41, No. 2, Dec. 1989, pp. 197-221.

Davis, H.W., A. Bramanti-Gregor, and J. Wang, The advantages of using depth and breadth components in heuristic search, in *Methodologies for Intelligent Systems 3*, Z.W. Ras and L. Saitta (Eds.), North-Holland, Amsterdam, 1989, pp. 19-28.

Dijkstra, E.W., A note on two problems in connexion with graphs, *Numerische Mathematik*, Vol. 1, 1959, pp. 269-71.

Gaschnig, J. *Performance measurement and analysis of certain search algorithms*, Ph.D. thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa, 1979.

Hart, P.E., N.J. Nilsson, and B. Raphael, A formal basis for the heuristic determination of minimum cost paths, *IEEE Transactions on Systems Science and Cybernetics*, Vol. 4, No. 2, 1968, pp. 100-107.

Korf, R.E., Depth-first iterative-deepening: An optimal admissible tree search, *Artificial Intelligence*, Vol. 27, No. 1, 1985, pp. 97-109.

Korf, R.E., Real-time heuristic search, *Artificial Intelligence*, Vol. 42, No. 2-3, March 1990, pp. 189-211.

Korf, R.E., Real-Time search for dynamic planning, *Proceedings of the AAAI Symposium on Planning in Uncertain, Unpredictable, or Changing Environments*, Stanford, Ca., March 1990, pp. 72-76.

Korf, R.E., Best-first search in limited memory, in *UCLA Computer Science Annual*, University of California, Los Angeles, Ca., April, 1991, pp. 5-22.

Korf, R.S., Linear-Space Best-First Search: Extended Abstract, *Proceedings of the Sixth International Symposium on Computer and Information Sciences*, Antalya, Turkey, October 30, 1991, pp. 581-584.

Korf, R.E., Linear-space best-first search, submitted for publication, August 1991.

Mahanti, A., D.S. Nau, S. Ghosh, and L.N. Kanal, An Efficient Iterative Threshold Heuristic Tree Search Algorithm, Technical Report UMIACS TR 92-29, CS TR 2853, Computer Science Department, University of Maryland, College Park, Md, March 1992.

Patrick, B.G., M. Almulla, and M.M. Newborn, An upper bound on the complexity of iterative-deepening-A*, in *Proceedings of the Symposium on Artificial Intelligence and Mathematics*, Ft. Lauderdale, Fla., Dec. 1989.

Pearl, J. *Heuristics*, Addison-Wesley, Reading, Mass, 1984.

Pohl, I., Heuristic search viewed as path finding in a graph, *Artificial Intelligence*, Vol. 1, 1970, pp. 193-204.

Rao, V.N., V. Kumar, and R.E. Korf, Depth-first vs. best-first search, *Proceedings of the National Conference on Artificial Intelligence, (AAAI-91)*, Anaheim, Ca., July, 1991, pp. 434-440.

Ruby, D., and D. Kibler, Learning subgoal sequences for planning, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, Detroit, Mich, Aug. 1989, pp. 609-614.

Russell, S., Efficient memory-bounded search methods, *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI-92)*, Vienna, Austria, Aug., 1992.

Sarkar, U.K., P.P. Chakrabarti, S. Ghose, and S.C. DeSarkar, Reducing reexpansions in iterative-deepening search by controlling cutoff bounds, *Artificial Intelligence*, Vol. 50, No. 2, July, 1991, pp. 207-221.

Sen, A.K., and A. Bagchi, Fast recursive formulations for best-first search that allow controlled use of memory, *Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI-89)*, Detroit, Michigan, Aug., 1989, pp. 297-302.

Wah, B.W., MIDA*, An IDA* search with dynamic control, Technical Report UILU-ENG-91-2216, CRHC-91-9, Center for Reliable and High-Performance Computing, Coordinated Science Laboratory, University of Illinois, Urbana, Ill., April, 1991.