Artificial Intelligence 62 (1993) 41–78
Elsevier

ARTINT 965

Linear-space best-first search

Richard E. Korf

Computer Science Department, University of California, Los Angeles, Los Angeles, CA 90024, USA

Received September 1991
Revised July 1992

Abstract

Korf, R.E., Linear-space best-first search, Artificial Intelligence 62 (1993) 41–78.

Best-first search is a general heuristic search algorithm that always expands next a frontier node of lowest cost. It includes as special cases breadth-first search, Dijkstra's single-source shortest-path algorithm, and the A* algorithm. Its applicability, however, is limited by its exponential memory requirement. Previous approaches to this problem, such as iterative deepening, do 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 nondecreasing cost function. On the sliding-tile puzzles, RBFS with a nonmonotonic 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.

1. Introduction and overview

Heuristic search is a general problem-solving mechanism in artificial intelligence. The sequence of steps required for solution of most AI problems is not known a priori, but must be determined by a systematic trial-and-error exploration of alternatives. A typical single-agent search problem is the Traveling Salesman Problem (TSP) of finding a shortest tour that visits each of a set of cities, and returns to the starting city. Classic toy examples include the sliding-tile puzzles, consisting of a frame containing numbered square tiles, and an empty position called the "blank" (see Fig. 1). The

Correspondence to: R.E. Korf, Computer Science Department, University of California, Los Angeles, Los Angeles, CA 90024, USA. Telephone: (310) 206-5383. E-mail: korf@cs.ucla.edu.

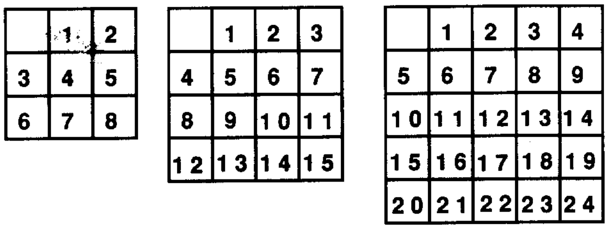

Fig. 1. Eight, Fifteen, and Twenty-Four Puzzles.

legal operators slide any tile horizontally or vertically adjacent to the blank into the blank position. The task is to rearrange the tiles from some random initial configuration into a particular goal configuration, in a minimum number of moves. Finding optimal solutions to the TSP [9] or sliding-tile puzzles [21] is NP-complete.

A search task is formulated as a *problem space graph.* Nodes represent states of the problem, and edges represent legal moves. A *problem instance* consists of a problem space together with an initial state and a set of goal states. A solution to a problem instance is a path from the initial node to a goal node. Although most problem spaces are graphs with cycles, for simplicity we represent them as trees, at the cost of a single state in the graph being represented by multiple nodes in the tree. The *branching factor* (b) of a search tree is the average number of children of a given node, while the maximum *depth* (d) of a problem instance is the length of the longest path generated in solving the problem instance. The term *generating* a node refers to creating the data structure representing the node, while *expanding* a node means to generate all of its children.

1.1. Best-first search

Best-first search is a general heuristic search algorithm. It maintains an *open* list containing the frontier nodes of the tree that have been generated but not yet expanded, and a *closed* list containing the interior or expanded 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 node 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.

Special cases of best-first search include breadth-first search, Dijkstra's single-source shortest-path algorithm, and the A* algorithm, differing only in their cost functions. If the cost of a node is its depth in the tree, then best-first search becomes breadth-first search, expanding all nodes at a given depth before expanding any nodes at a greater depth. If edges have different costs, and the cost of a 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 [5]. 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 [8]. If $h(n)$ never overestimates the actual cost from node n to a goal, then A* will return minimum-cost solutions. For example, a common non-overestimating heuristic function for the sliding-tile puzzles is Manhattan distance: the sum over all tiles of the number of grid units each tile is displaced from its goal position. Similarly, the cost of a minimum spanning tree is a lower bound on the shortest TSP tour of a set of cities. Furthermore, under certain conditions, A* generates the fewest nodes of any algorithm guaranteed to find a lowest-cost solution [4].

1.2. Limitations of best-first search

The most serious limitation, however, of best-first search is its memory requirement, which is proportional to the number of nodes stored. Since best-first search stores all nodes in the open or closed lists, its space complexity is the same as its time complexity, which is often exponential. Given the ratio of memory to processing speed on current computers, best-first search typically exhausts the available memory in a matter of minutes, halting the algorithm. Secondary storage doesn't help, since in order to efficiently detect duplicate nodes, the open and closed lists must be randomly accessed.

A secondary limitation is the time required to generate new nodes and insert them into the open list. To generate each new node, best-first search must make a complete copy of the parent state, modified accordingly. This takes time proportional to the size of the state representation. For example, in the sliding-tile puzzles, this requires time proportional to the number of tiles. In addition, each new state must be inserted into the open list in a manner that facilitates retrieving a lowest-cost open node. If a heap is used, insertion and retrieval require $O(\log N)$ time per node, where N is the total number of open nodes.

1.3. Overview

We will address both these limitations in this paper. First we review a previous approach to these problems, the iterative-deepening algorithm. For some problems, however, iterative deepening generates many more nodes than best-first search, and if the cost function can decrease along a path, then iterative deepening no longer expands nodes in best-first order. We present a new Recursive Best-First Search algorithm (RBFS) that runs in linear space, and always expands new nodes in best-first order, regardless of the cost function. For pedagogical reasons, we first present a simple but inefficient version of the algorithm, Simple Recursive Best-First Search (SRBFS), and then the full RBFS algorithm. We then prove that RBFS is a best-first search. The space complexity of both algorithms is $O(bd)$. For the special case of cost as depth, SRBFS generates asymptotically more nodes than breadth-first search, but RBFS is asymptotically optimal. We also show that with a nondecreasing cost function, RBFS generates fewer nodes than iterative deepening. We present empirical results for RBFS on the TSP using the A* evaluation function, showing that RBFS generates significantly fewer nodes than IDA*. On the sliding-tile puzzles, using a weighted cost function, $f(n) = g(n) + Wh(n)$, RBFS finds nearly optimal solutions with orders of magnitude reductions in nodes generated, and produces consistently shorter solutions than iterative deepening with the same value of W, but generates more nodes. These algorithms first appeared in [13], and the main results were summarized in [14,15].

2. Previous work

The memory limitation of best-first search was first addressed by iterative deepening [10]. Other related algorithms will be discussed in Section 7.

2.1. *Iterative deepening*

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 threshold on the previous iteration. The algorithm terminates when a goal node is chosen for expansion. Since iterative deepening is a depth-first search, it only stores the nodes on the current search path, requiring $O(d)$ space.

If the cost of a node is its depth, iterative deepening becomes depth-first iterative deepening (DFID), corresponding to breadth-first search [10,26]. DFID performs a series of depth-first searches, increasing the depth cutoff by one at each iteration, until a goal is found. The total number of nodes generated by DFID is $b^d + 2b^{d-1} + 3b^{d-2} + \cdots + db = O(b^d)$, and the ratio of the number of nodes generated by DFID to the number generated by breadth-first search is approximately $b/(b-1)$.

If we run iterative deepening with cost equal to $g(n)$, we get an iterative-deepening version of Dijkstra's algorithm. Alternatively, if we use the A* evaluation function, $f(n) = g(n) + h(n)$, we get iterative-deepening-A* (IDA*) [10]. IDA* was the first algorithm to find optimal solutions to the Fifteen Puzzle.

An additional benefit of iterative deepening is that the time per node generation is less than for best-first search. Iterative deepening only stores a single copy of the state, making incremental changes to the state to generate a child node, and undoing those changes to return to the parent. For example, to make a move in the sliding-tile puzzles, only the locations of the tile moved and the blank are modified. Thus, the time to generate a move is less than for best-first search, which must copy the entire state along with any changes. Furthermore, iterative deepening does not maintain any open or closed lists, and hence does not incur the logarithmic insertion and retrieval costs of standard best-first search. For the same reason, the code for IDA* is also shorter and easier to implement than that of A*.

In the case of breadth-first search, Dijkstra's algorithm, and A* with *consistent* heuristic functions [18], the cost function is *monotonic*, in the sense that the cost of a child is always greater than or equal to the cost of its parent. Formally, a cost function $f(n)$ is monotonic if and only if for all nodes n and n', where n' is a child of n, $f(n') \geqslant f(n)$.

With a monotonic cost function, iterative deepening expands nodes in best-first order. To prove this, we first define an open node as one that has been generated at least once, but not expanded yet. In standard best-first search, all open nodes are explicitly stored in the open list, whereas in iterative deepening, at most one open node is ever in memory at any given time. We then define an algorithm to expand nodes in best-first order if whenever a node is expanded, its cost is less than or equal to the costs of all open nodes at the time.

Theorem 2.1. *Given a monotonic cost function, iterative deepening expands nodes in best-first order.*

Proof. The proof is by induction on the number of iterations. For the basis step, consider the first iteration. At the start of the first iteration, the root node r is the only open node. The initial cutoff threshold $C_1 = f(r)$, and the first iteration only expands nodes for which $f(n) \leqslant C_1 = f(r)$. Since the cost function is monotonic, all nodes in the tree rooted at r have $f(n) \geqslant f(r)$. Thus, any open nodes generated during the first iteration have $f(n) \geqslant f(r)$, and the first iteration expands nodes in best-first order.

For the induction step, assume that for all iterations up to and including iteration k, iterative deepening expands nodes in best-first order. Now consider iteration $k+1$. Let C_k and C_{k+1} be the cutoff thresholds for iterations k and $k+1$, respectively. Since the cost function is monotonic, iteration k expands all nodes n for which $f(n) \leqslant C_k$. At the end of iteration k, the open nodes consist of those that were generated but not expanded during iteration k, and thus have costs $f(n) > C_k$. C_{k+1} is the minimum of those values. During iteration $k+1$, only those nodes for which $f(n) \leqslant C_{k+1}$ are expanded. Any new open nodes generated during iteration $k+1$ must have $f(n) \geqslant C_{k+1}$, since if $f(n) < C_{k+1}$, then $f(n) \leqslant C_k$, and

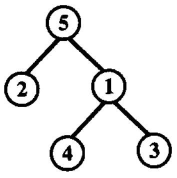

Fig. 2. Tree fragment with nonmonotonic cost function.

node n would have been expanded during iteration k, and would no longer be an open node during iteration $k+1$. Thus, since all nodes expanded during iteration $k+1$ have $f(n) \leqslant C_{k+1}$, and all open nodes n during iteration $k+1$ have $f(n) \geqslant C_{k+1}$, iteration $k+1$ expands nodes in best-first order. Therefore, by induction, iterative deepening always expands nodes in best-first order, given a monotonic cost function. □

2.2. Limitations of iterative deepening

When the cost function is nonmonotonic, however, meaning that the cost of a child can be less than the cost of its parent, then iterative deepening no longer expands nodes in best-first order. For example, this occurs in *pure heuristic search*, which is best-first search with $f(n) = h(n)$ [6], or in the *heuristic path algorithm*, a best-first search with $f(n) = g(n) + Wh(n)$ [19], with $W > 1$. The importance of these algorithms is that even though they find suboptimal solutions, they generate many fewer nodes than are required to find optimal solutions.

For example, consider the tree fragment in Fig. 2, where the numbers represent node costs. A best-first search would expand these nodes in the order 5, 1, 2. With iterative deepening, however, 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 subtrees below nodes 4 and 3 before expanding node 2. The problem 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 depth-first.

This problem is even more serious on a graph with cycles. For example, with a pure heuristic cost function, $f(n) = h(n)$, there may be cycles in the graph in which the maximum node cost is less than the current cost threshold. In that case, iterative deepening will go around such a cycle forever without terminating. Fortunately, since iterative deepening maintains the current search path on the recursion stack, this problem can be fixed by comparing each newly generated node with those on the current path, and

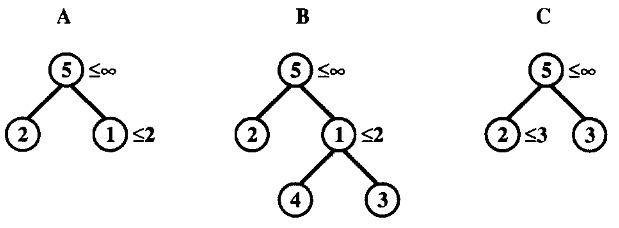

Fig. 3. SRBFS example with nonmonotonic cost function.

pruning any node that already appears on the stack.

An additional limitation is that for some problems, iterative deepening generates many more nodes than best-first search. The worst case occurs when all node values in the tree are unique. Since there is a separate iteration for each distinct cost value, if each node has a unique cost, then each iteration expands only one new node that was not expanded in the previous iteration. In that case, the time complexity of iterative deepening becomes $O(N^2)$, where N is the number of nodes generated by best-first search [17]. For example, iterative deepening performs poorly on the TSP if the edge costs have many significant digits, since in that case many nodes have distinct cost values.

3. Recursive best-first search

Recursive Best-First Search (RBFS) is a linear-space algorithm that expands nodes in best-first order even with a nonmonotonic cost function, and generates fewer nodes than iterative deepening with a 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).

3.1. 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 3 shows how SRBFS searches the tree in Fig. 2 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 Fig. 3A. 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

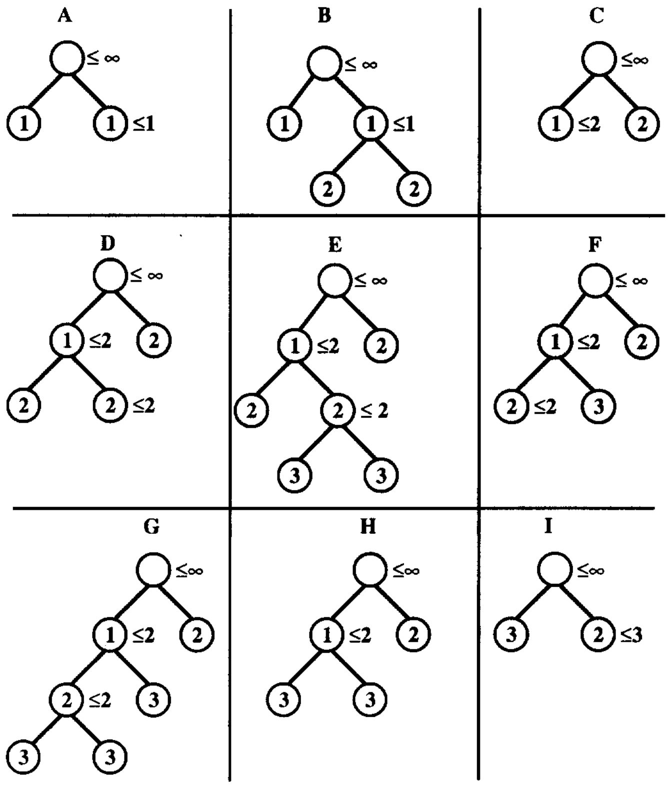

Fig. 4. SRBFS example with cost equal to depth.

the value of the left child. Thus, the recursive call on the right child has an upper bound equal to the value of its lowest-cost brother, 2. SRBFS expands the right child, and evaluates the grandchildren, as shown in Fig. 3B. 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 (Fig. 3C), 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 lowest-cost 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 4 shows a more extensive example of SRBFS. In this case, the cost function is simply the depth of the node in the tree, corresponding to breadth-first search. The children of a node are generated from left to right, and ties are broken in favor of the most recently generated node.

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

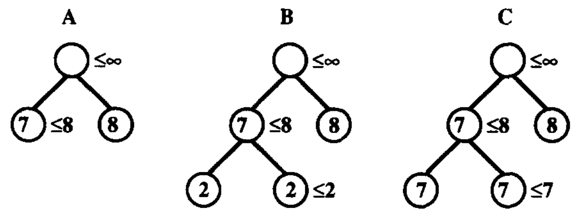

Fig. 5. Inefficiency of SRBFS and its solution.

minimum value of all frontier nodes in the subtree explored during the last call. SRBFS proceeds down a path until the static values of all children of the last node expanded exceed 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 and freeing memory, 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. At any point, the recursion stack contains the path to a lowest-cost frontier node, plus the siblings of all nodes on that path. Thus, its space complexity is $O(bd)$, which is linear in the maximum search depth d. 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 and F[1] < infinity)
  F[1] := SRBFS(N1, MIN(B, F[2]))
  insert N1 and F[1] in sorted order
return F[1]
```

Once a goal node is chosen for expansion, 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 nonmonotonic. Unfortunately, however, SRBFS is inefficient. If we continue the example from Fig. 4, where cost is equal to depth, eventually we would reach the situation shown in Fig. 5A, 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 Fig. 5B. 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, then 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.

3.2. 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 their parent's value of 7 as their stored value, instead of 2, as shown in Fig. 5C. 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 Fig. 2, where the fact that the child's value is smaller than its parent's value is due to nonmonotonicity in 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 has never been expanded before, its stored value equals its static value. In order to be expanded, the upper bound on a node must be at least as large as its stored value. The recursive call on the node will not return until the values of all the frontier nodes in the subtree below it exceed its upper bound, and its new stored value will be set to the minimum of these values. Thus, if a node has been previously expanded, its stored value exceeds its static value.

If the stored value of a node is greater than its static value, its stored value is the minimum of the last stored values of its children. 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. If the stored value of a node is equal to its static value, then the node has never been expanded before, and the values of its children should be set to their static values. In general, a parent's stored value is passed down to its children, which inherit the value only if it exceeds both the parent's static value and the child's static value.

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

```
RBFS (node: N, value: F(N), bound: B)
IF f(N)>B, return f(N)
IF N is a goal, EXIT algorithm
IF N has no children, RETURN infinity
```

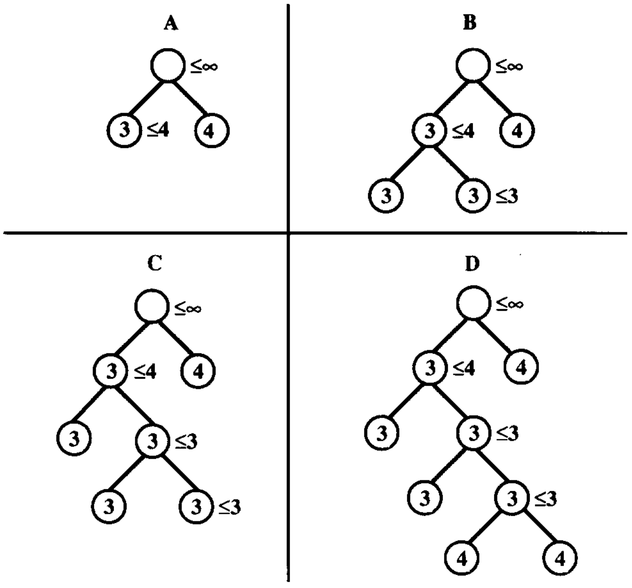

Fig. 6. Recursive best-first search (RBFS) example.

```
FOR each child Ni of N,
  IF f(N)<F(N) THEN F[i] := MAX(F(N),f(Ni))
  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 and F[1] < infinity)
  F[1] := RBFS(N1, F[1], MIN(B, F[2]))
  insert N1 and F[1] in sorted order
return F[1]
```

Figure 6 shows RBFS in action, again for the special case where cost is equal to depth. Once the search has progressed to the situation in Fig. 6A, the left child of the root is expanded, with an upper bound equal to its brother's value of 4, and its stored value of 3 is passed down to its children. Since the stored value of the parent, 3, exceeds its static value, 1, and also the children's static values of 2, the children's stored values are set to their parent's stored value of 3, as shown in Fig. 6B. At this point, the rightmost grandchild is expanded with an upper bound equal to 3, the minimum of its parent's upper bound of 4, and its brother's value of 3, and its stored value of 3 is passed down to its children. Once again, since the stored value of the parent, 3, exceeds the static value of the parent, 2, and is not less than the children's static value of 3, the children inherit their parent's stored value of 3, as shown in Fig. 6C. The rightmost great-grandchild is now expanded with an upper bound of 3, which is the minimum of its parent's upper bound of

3 and its brother's value of 3, and its stored value of 3 is passed down to its children. In this case the parent's stored value of 3 equals its static value of 3, and the children's stored values are set to their static values of 4, as shown in Fig. 6D. At this point, even if the static values of the children were less than that of their parent, the children would adopt their static values, and the subtrees below them would be explored in best-first order, until the values of all the frontier nodes exceeded the upper bound of 3.

Like SRBFS, RBFS explores new nodes in best-first order, even with a nonmonotonic cost function, and also runs in linear space. Its advantage over SRBFS is that it is more efficient. RBFS behaves differently depending on whether it is expanding new nodes, or previously expanded nodes. In new territory, it proceeds strictly best-first, while on old ground it goes depth-first, until it reaches a lowest-cost node on the old frontier, at which point it reverts back to best-first behavior.

If there are cycles in the graph, and cost does not always increase while traversing a cycle, as with a pure heuristic cost function, $f(n) = h(n)$, the same modification we described for iterative deepening in Section 2.2 must be made to RBFS, in order 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.

4. Correctness of RBFS

In this section we prove that RBFS searches a tree in best-first order, finding a goal if one exists. The same result applies to SRBFS as well, but we present the results for RBFS only, since RBFS performs the same task as SRBFS, but more efficiently. The reader who is inclined to skip over the proofs is encouraged to at least read the definitions and the statements of the lemmas and theorems, since they provide insight into the behavior of the algorithm.

We assume that the problem space is a potentially infinite tree, rooted at node r. If the problem space is a graph, then the algorithm searches the tree generated by a depth-first traversal of the graph starting at state r. Any node that has multiple paths to it in the graph will be represented by multiple copies in the tree, one for each distinct path to it.

Associated with every node n in the tree is a static evaluation $f(n)$. Given a tree T containing a node n, we define a *subtree* $T(n)$ as a tree rooted at n, such that for any node m in $T(n)$, all the brother nodes of m are also in $T(n)$. In other words, if one child of a node is in the subtree, then all children of the node are in the subtree as well. A subtree is composed of *interior nodes*, all of whose children are in the subtree, if any, and *frontier nodes*, none of whose children are in the subtree. A subtree is *explored* by generating all of its nodes. Non-goal terminal nodes are regarded as interior nodes once they have been expanded, and RBFS returns ∞ as the value of such nodes.

Given a tree T, a node n, and an upper bound b, we define a *b-bounded subtree* of n, $T(n,b)$, as a subtree $T(n)$ in which all interior nodes m have $f(m) \leqslant b$, and all frontier nodes m have $f(m) > b$. For a given n and b, $T(n,b)$ is unique. It is explored by successively expanding those nodes m with $f(m) \leqslant b$, until all unexpanded nodes are either terminal nodes, or frontier nodes with $f(m) > b$. If $b \leqslant c$, then $T(n,b)$ is a subtree of $T(n,c)$, since increasing the bound can only generate a larger subtree.

To guarantee termination, we constrain the evaluation function $f(n)$ so that all finite-bounded subtrees are finite. In other words, for all nodes n, and finite values b, $T(n,b)$ contains a finite number of nodes and edges. This constraint is satisfied by the A* evaluation function $f(n) = g(n) + h(n)$, and the weighted evaluation function $f(n) = g(n) + Wh(n)$, but not by the pure heuristic function $f(n) = h(n)$ on an infinite tree. Even standard best-first search may not terminate with this evaluation function on an infinite tree.

We define $MF(n,b)$ to be the minimum f value of any frontier node in $T(n,b)$. If there are no frontier nodes in $T(n,b)$, then $MF(n,b)$ is defined as ∞. Note that $MF(n,b) > b$, since for all frontier nodes m in $T(n,b)$, $f(m) > b$. Let $D(n,b)$ be the maximum depth of $T(n,b)$, or the depth of a deepest node of $T(n,b)$ below node n. Finally, let $F(n)$ be the current stored value of node n in RBFS. We use $F[i]$ to denote the current stored F value of the ith child in the sorted order of children.

We first establish a simple property of all calls to RBFS.

Lemma 4.1. *All calls to RBFS are of the form $RBFS(n,F(n),b)$, where $F(n) \leqslant b$.*

Proof. The initial call to RBFS is $RBFS(r,f(r),\infty)$, where r is the root of the tree. Since $f(r) \leqslant \infty$, the initial call is of the form $RBFS(n,F(n),b)$, where $F(n) \leqslant b$. By inspection of the code, all recursive calls are of the form $RBFS(n1,F[1],\min(b,F[2]))$. The condition of the WHILE loop guarantees that $F[1] \leqslant b$. Since the children are sorted by F values, $F[1] \leqslant F[2]$. Thus, $F[1] \leqslant \min(b,F[2])$. Therefore, all recursive calls are of the form $RBFS(n,F(n),b)$, where $F(n) \leqslant b$. $\square$

Next, we show that $RBFS(n,F(n),b)$ explores $T(n,b)$, the b-bounded subtree of n, and returns $MF(n,b)$, the minimum frontier node value.

Lemma 4.2. *If b is finite, and $T(n,b)$ does not contain an interior goal node, then $RBFS(n,F(n),b)$ explores $T(n,b)$ and returns $MF(n,b)$.*

Proof. The proof is by induction on $D(n,b)$, the depth of $T(n,b)$. Since b is finite, and by assumption all finite-bounded subtrees are finite, $D(n,b)$ is finite.

Basis step: $D(n,b) = 0$. In other words, n is the only node in $T(n,b)$, and thus $f(n) > b$. Therefore, $RBFS(n,F(n),b)$ will return $f(n)$ without expanding n, hence exploring $T(n,b)$ and returning $MF(n,b)$.

Induction step: Assume the lemma is true for all calls $RBFS(n,F(n),b)$ for which $D(n,b) \leqslant k$. Now consider a call $RBFS(n,F(n),b)$, with finite b, no interior goal node in $T(n,b)$, and $D(n,b) = k+1$. Let $RBFS(n',F(n'),b')$ be a recursive call within $RBFS(n,F(n),b)$, where n' is a child of n. Since $b' = \min(b,F[2])$, $b' \leqslant b$. Therefore, $T(n',b')$ is a subtree of $T(n',b)$, and $D(n',b') \leqslant D(n',b)$. Furthermore, since n' is a child of n, $D(n',b) < D(n,b)$. Since $D(n,b) = k+1$, and $D(n',b') \leqslant D(n',b) < D(n,b)$, $D(n',b') \leqslant k$. Finally, since $T(n',b')$ is a subtree of $T(n,b)$, if $T(n,b)$ does not contain an interior goal node, then $T(n',b')$ cannot contain an interior goal node either. Thus, since all recursive calls satisfy the conditions of the induction hypothesis, $RBFS(n',F(n'),b')$ explores $T(n',b')$ and returns $MF(n',b')$.

By Lemma 4.1, for all calls $RBFS(n',F(n'),b')$, $F(n') \leqslant b'$. The new value of $F(n')$ is the value returned by $RBFS(n',F(n'),b')$, which by the induction hypothesis is $MF(n',b')$, which is $> b'$. Thus, every recursive call on n' increases its stored value $F(n')$. Since the $F(n')$ value of a node must be less than b to generate a recursive call, and b is finite, the WHILE loop eventually terminates and no further recursive calls are made, assuming that there are no infinitely increasing cost sequences with a finite upper bound, such as $\frac{1}{2}$, $\frac{3}{4}$, $\frac{7}{8}$, etc.

If a child n' is never expanded during the call on its parent, its initial stored value $F(n')$ must be greater than b. Since initially $F(n')$ is either $F(n)$ or $f(n')$, and $F(n) \leqslant b$, if $F(n') > b$, then $F(n') = f(n') > b$. Thus, n' is the only node of $T(n',b)$, and $F(n') = f(n') = MF(n',b)$.

If a child n' was expanded during $RBFS(n,F(n),b)$, its current value $F(n')$ is the one returned by the last recursive call on it, $RBFS(n',F(n'),b')$. Since $b' = \min(b,F[2])$, where $F[2]$ is the lowest F value among the brothers of n', and the successive F values of a given node form a strictly increasing sequence within a parent call, the upper bound b' on the last call to $RBFS(n',F(n'),b')$ must be the largest upper bound for any call to $RBFS(n',F(n'),b')$ made during the parent call to $RBFS(n,F(n),b)$. Thus, the largest subtree explored below n' during $RBFS(n,F(n),b)$ was explored during the last recursive call $RBFS(n',F(n'),b')$, and $T(n',b')$ contains any previous subtrees of n' explored during $RBFS(n,F(n),b)$. Since $b' \leqslant b$, all interior nodes m of $T(n',b')$ have costs $f(m) \leqslant b' \leqslant b$. The value returned from this last call, $MF(n',b') = F(n') > b$. Thus, all frontier nodes m of $T(n',b')$ have $f(m) > b$. Since all interior nodes m

of $T(n',b')$ have $f(m) \leqslant b$, and all frontier nodes m of $T(n',b')$ have $f(m) > b$, $T(n',b') = T(n',b)$, and $MF(n',b') = MF(n',b) = F(n')$.

The union of all the subtrees $T(n',b)$ for all the children n' of n is equal to $T(n,b)$. Furthermore, the minimum of $F(n') = MF(n',b)$ for all the children n' of n, which is the value returned by $RBFS(n,F(n),b)$, is equal to $MF(n,b)$. Thus, $RBFS(n,F(n),b)$ explores $T(n,b)$ and returns $MF(n,b)$. By induction then, the lemma is true for all such calls. □

We now turn to the order of node expansion by RBFS. Recall that we define an open node to be one that has been generated at least once, but not expanded yet. While in standard best-first search all open nodes are explicitly stored, in RBFS very few of these nodes are actually in memory at any point in time. Define $OD(n)$ to be the minimum f value of all open nodes that are descendents of node n, and let $ON(n)$ be the minimum f value of all open nodes that are *not* descendents of node n.

Lemma 4.3. *For all calls* $RBFS(n,F(n),b)$, $F(n) \leqslant OD(n)$ *and* $b \leqslant ON(n)$.

Proof. The proof is by induction on the depth of node n below the root r of the entire tree.

Basis step: n is at depth zero below the root, meaning $n = r$, the root node. The call on the root is made with $F(n) = f(r)$ and $b = \infty$. At that point, the only open node descendent of r is r itself, and $F(n) = f(r) = OD(r) \leqslant OD(r)$. Since there can't be any open nodes in the tree that are not descendents of the root, any value of b, including ∞, can be considered a lower bound on their f values.

Induction step: Assume that for all calls $RBFS(n,F(n),b)$, where n is at depth k from the root, $F(n) \leqslant OD(n)$ and $b \leqslant ON(n)$. Now consider a call $RBFS(n',F(n'),b')$, where n' is a child of n, and hence n' is at depth $k+1$ from the root.

If n has never been expanded before, it is its only open node descendent, $OD(n) = f(n)$, and since $F(n) \leqslant OD(n)$, $F(n) \leqslant f(n)$. In that case, the initial value of $F(n') = f(n')$. Since none of the children of n have been expanded before either, $F(n') = f(n') = OD(n')$, and hence $F(n') \leqslant OD(n')$.

Alternatively, if n has been expanded before, the initial $F(n')$ values of the children n' of n are set to either $f(n')$ or $F(n)$. If n' has not been expanded before, $f(n') = OD(n')$, and if n' has been expanded before, it would be an interior node of the largest subtree explored below it, and $f(n') < OD(n')$. In either case, $f(n') \leqslant OD(n')$. Since n was expanded before, and $F(n) \leqslant OD(n)$, $F(n) \leqslant OD(n')$ for all children n' of n as well. Since $f(n') \leqslant OD(n')$, and $F(n) \leqslant OD(n')$, and $F(n')$ is either equal to

$f(n')$ or $F(n)$, $F(n') \leqslant OD(n')$. Thus, regardless of whether n has ever been expanded before, the initial value of $F(n') \leqslant OD(n')$.

If a recursive call has been made on a child n' during $RBFS(n,F(n),b)$, then by Lemma 4.2, $F(n') = MF(n',b')$, where $T(n',b')$ was the subtree explored during the last recursive call on n', $RBFS(n',F(n'),b')$. Let $T(n',c')$ be the largest subtree ever explored below node n'. Thus, the frontier nodes of $T(n',c')$ are the open node descendents of n', $MF(n',c')$ is their minimum value, and $MF(n',c') = OD(n')$. $T(n',b')$, the last subtree explored below n', either equals $T(n',c')$, or is a subtree of $T(n',c')$. If $T(n',b') = T(n',c')$, then $F(n') = MF(n',b') = MF(n',c') = OD(n')$. Alternatively, if $T(n',b')$ is a subtree of $T(n',c')$, then $b' < c'$, $F(n') = MF(n',b') \leqslant MF(n',c') = OD(n')$. In either case, $F(n') \leqslant OD(n')$. Thus regardless of how many recursive calls have been made on a child n' of n, $F(n') \leqslant OD(n')$.

When a recursive call $RBFS(n',F(n'),b')$ is made, $F(n') \leqslant F(m')$, for all brothers m' of n', since recursive calls are always made on a brother with the lowest F value. Since $F(m') \leqslant OD(m')$, $F(n') \leqslant OD(m')$ for all brothers m' of n'. Since b' is the minimum of the parent's bound b, and the lowest $F(m')$ value of all the brothers m' of n', $b' \leqslant F(m') \leqslant OD(m')$ for all brothers m' of n'. Furthermore, since $b' \leqslant b$, and since $b \leqslant ON(n)$, $b' \leqslant ON(n)$. The open nodes that are not descendents of n' are the union of the open nodes that are not descendents of its parent n, $ON(n)$, and the open node descendents of its brothers m', $OD(m')$. Thus, since $b' \leqslant ON(n)$, and $b' \leqslant OD(m')$, $b' \leqslant ON(n')$.

Thus, when a recursive call $RBFS(n',F(n'),b')$ is made, $F(n') \leqslant OD(n')$, and $b' \leqslant ON(n')$. By induction, then, for all calls $RBFS(n,F(n),b)$, $F(n) \leqslant OD(n)$, and $b \leqslant ON(n)$. □

Next, we show that RBFS expands nodes in best-first order.

Lemma 4.4. *When a node is expanded by RBFS, its f value is less than or equal to the f values of all open nodes at the time.*

Proof. Node n is expanded by a call of the form $RBFS(n,F(n),b)$. Before expanding node n, RBFS checks to make sure that $f(n) \leqslant b$. By Lemma 4.3, we know that $b \leqslant ON(n)$. Thus, $f(n) \leqslant ON(n)$.

Since the initial $F(n)$ value of a node n is either $f(n)$ or $\max(v,f(n))$, the initial value $F(n) \geqslant f(n)$. Furthermore, we showed in the proof of Lemma 4.2 that the sequence of $F(n)$ values of a node n within a call to its parent is strictly increasing. Thus, for all values of $F(n)$, $F(n) \geqslant f(n)$. Since, by Lemma 4.3, $F(n) \leqslant OD(n)$, $f(n) \leqslant OD(n)$.

Every open node in the tree must either be a descendent of node n, or not a descendent of node n. Since we have shown that $f(n) \leqslant ON(n)$, and $f(n) \leqslant OD(n)$, $f(n)$ is less than or equal to the f values of all open nodes in the tree when n is expanded. □

Finally, we show that RBFS searches the entire tree in best-first order, eventually finding a solution if one exists.

Theorem 4.5. *$RBFS(r, f(r), \infty)$ will perform a complete best-first search of the tree rooted at node r, exiting after finding the first goal node chosen for expansion.*

Proof. Lemma 4.4 shows that RBFS performs a best-first search. All that remains is to show that $RBFS(r, f(r), \infty)$ will perform a complete search of the tree, rooted at node r, until choosing a goal node for expansion. Since $f(r) < \infty$, $RBFS(r, f(r), \infty)$ will expand node r, generating each of its children r', and assigning $F(r') = f(r')$. It will then perform a series of recursive calls $RBFS(r', F(r'), b')$ on the child r' with the lowest current value of $F(r')$. From the proof of Lemma 4.2, we know that the sequence of $F(r')$ values for each child r' is strictly increasing. Since the bound on the parent call is ∞, the WHILE loop for the recursive calls can never terminate, generating an unbounded number of recursive calls. By repeatedly making recursive calls on a child of the root with the lowest F value, and strictly increasing the F value of the child with each call, we guarantee that there will be an unbounded number of recursive calls on each child. Since the upper bound on each of these calls is the next lowest F value, the upper bounds also increase continually, meaning that larger and larger subtrees of each child are explored, guaranteeing a complete search of the tree. Finally, the only exit from the top-level call on the root of the tree occurs when a goal node is chosen for expansion. □

5. Complexity of SRBFS and RBFS

We now turn to the space and time complexities of SRBFS and RBFS.

5.1. Space complexity

The space complexity of SRBFS and RBFS is $O(bd)$, where b is the branching factor and d is the maximum search depth. Each recursive call of either algorithm must store all the children of the argument node, along with their F values. This requires $O(b)$ space, where b is the branching factor of the tree. The maximum number of active calls that must be stored on the recursion stack at any point is equal to the maximum search depth d. Thus, the overall space complexity is $O(bd)$.

In general, the maximum search depth d depends on the problem space, the problem instance, and the cost function. For the traveling salesman problem, for example, the maximum search depth is typically the number of cities. In the case of a monotonic cost function, the maximum search depth is the longest path in which no node costs exceed the minimum solution cost. For example, in the sliding-tile puzzles with a monotonic cost function, the maximum search depth will not exceed the shortest solution length.

5.2. *General time complexity*

The time complexity of standard best-first search is $O(N \log N)$, where N is the number of nodes generated, due to the management of the open list. Since SRBFS and RBFS have no global open lists, however, their time complexity is linear in the number of nodes generated, assuming a constant branching factor. Each call to SRBFS or RBFS generates b new nodes, where b is the branching factor. The most expensive step in a call is selecting a child of minimum F value. Regardless of whether the children are kept in a heap or sorted, since there are only a constant number of children, insertion and retrieval take only constant time. Thus, the overall time complexity is a constant times the number of node generations.

The actual number of nodes generated by RBFS or SRBFS depends on the particular cost function. We consider the best case, worst case, and the special case of cost as depth. In addition, we compare the time complexity of RBFS to iterative deepening in the case of a monotonic cost function. Finally, we refer to an average-case analysis of iterative deepening with a monotonic cost function, which implies that both iterative deepening and RBFS are asymptotically optimal in a particular abstract model.

5.3. *Best-case time complexity*

The best-case time complexity of both RBFS and SRBFS occurs when all nodes have identical cost values. In that case, neither algorithm expands any node more than once. In an infinite tree, they will continue down a single path forever, but on a uniform finite tree, both algorithms generate every node exactly once, and their time complexity is $O(b^d)$, assuming that no goal is found.

5.4. *Worst-case time complexity*

Conversely, the worst-case time complexity occurs when all nodes have unique cost values, and are arranged so that successive nodes in an ordered sequence of cost values are in different subtrees of the root node. For example, Fig. 7 shows a worst-case binary tree. In this case, in order to expand each new open node at depth d, both SRBFS and RBFS must

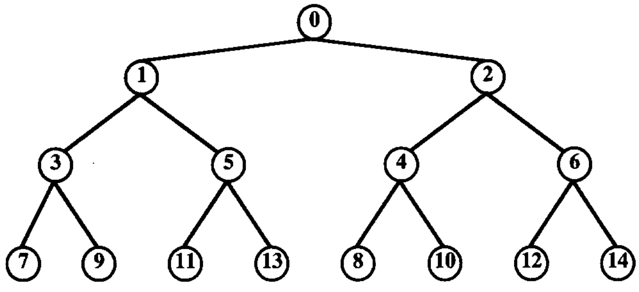

Fig. 7. Worst-case binary tree for SRBFS and RBFS.

abandon their current path all the way back to the root. The stored value of one of the remaining children of the root will equal the cost of the next open node, indicating which child it is to be found under. However, there is no information as to where it is to be found under this child. In the worst case, we may have to generate all nodes down to and including depth d under this child, in order to find one of lowest cost. Since they are all descendents of the same child of the root, there are $O(b^{d-1})$ such nodes. Thus, each new node expansion at depth d may require $O(b^{d-1})$ node generations. Since there are b^d nodes at depth d, to completely explore a tree of depth d may require $O(b^{2d-1})$ time. This is only slightly better than the $O(b^{2d})$ worst-case time complexity of iterative deepening [17].

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. For example, in a binary tree, an additional bit must be added to the representation of the cost values at each depth.

5.5. A special case: cost equal to depth

The above results indicate that the time complexity of SRBFS and RBFS is very sensitive to the number of distinct cost values and their distribution in the search tree. This makes any general average-case analysis difficult. To get some sense of actual performance to be expected in practice, we analyzed both SRBFS and RBFS for the special case where the cost function is simply the depth of a node in the tree, corresponding to breadth-first search.

5.5.1. Time complexity of SRBFS with cost as depth

Theorem 5.1. *The asymptotic time complexity of SRBFS on a uniform tree with branching factor b, depth d, and cost equal to depth, is $O(x^d)$, where*

$$x = \tfrac{1}{2}(b + 1 + \sqrt{b^2 + 2b - 3}).$$

Proof. In order to perform a search to depth d, SRBFS first generates all children of the root, requiring b node generations. All but the last child, $b-1$ in all, will be searched to depth 1 below the child. Then the last child will be searched to depth 2 below the child. Next, all but the last of the remaining children will be searched to depth 2, which again is a total of $b-1$ of them. In other words, after the children of the root are generated, there are a series of iterations in which $b-1$ of the children are searched to successively greater depths from 1 to $d-1$, inclusive. Finally, the last remaining child need only be searched to depth $d-1$ to complete the entire search to depth d. Thus, if $N(d)$ is the total number of nodes generated by SRBFS in a search to depth d, we have the recurrence:

$$N(d) = b + (b-1)\sum_{i=1}^{d-1} N(i) + N(d-1).$$

Hypothesizing that $N(d) = O(x^d)$, and substituting into our recurrence gives

$$x^d = b + (b-1)\sum_{i=1}^{d-1} x^i + x^{d-1},$$

$$\sum_{i=1}^{d-1} x^i \approx \frac{x^d}{(x-1)},$$

$$x^d \approx b + (b-1)\frac{x^d}{(x-1)} + x^{d-1}.$$

Dropping the b term, which is insignificant, and factoring out x^{d-1} gives

$$x \approx \frac{(b-1)x}{(x-1)} + 1.$$

Some simple algebra gives us the quadratic equation

$$x^2 - (b+1)x + 1 = 0,$$

which by the quadratic formula has a single root greater than one,

$$x = \tfrac{1}{2}(b + 1 + \sqrt{b^2 + 2b - 3})$$

for an overall complexity of

$$N(d) = O(x^d), \quad \text{where } x = \tfrac{1}{2}(b + 1 + \sqrt{b^2 + 2b - 3}). \qquad \square$$

The asymptotic branching factor, x, is strictly greater than b, indicating that SRBFS is not asymptotically optimal in this case. In the limit of large b, x approaches $b+1$. Table 1 gives the value of x for b from 2 to 10.

Table 1
Asymptotic branching factor of SRBFS for breadth-first search.

b	2	3	4	5	6	7	8	9	10
x	2.62	3.73	4.79	5.83	6.85	7.87	8.89	9.90	10.91

5.5.2. *Time complexity of RBFS with cost as depth*

RBFS, however, is asymptotically optimal for the special case of cost as depth.

Theorem 5.2. *The asymptotic time complexity of RBFS on a uniform tree with branching factor b, depth d, and cost equal to depth, is* $O(b^d)$.

Proof. RBFS behaves like depth-first iterative deepening (DFID) in this case. In searching to depth d, RBFS first searches every node to depth $d-1$. To transition from searching from one depth to the next, DFID returns to the root and begins a new iteration from scratch. RBFS, however, begins the search to depth d immediately after the last node at depth $d-1$ has been generated. At this point, instead of returning to the root, RBFS has a stack of nodes from the root to this last node at depth $d-1$. The total number of nodes on this stack, not counting the root, is $b(d-1)$, since the siblings of all nodes on the current path are stored as well. These nodes are not regenerated during the search to depth d, resulting in a small savings over DFID. The total number of nodes in a tree of depth d is

$$\sum_{i=1}^{d} b^i \approx \frac{b^{d+1}}{(b-1)}.$$

Thus, to search to depth d, RBFS must first search to depth $d-1$, then generate $b^{d+1}/(b-1) - b(d-1)$ nodes in searching to depth d. This gives the following recurrence:

$$N(d) = N(d-1) + \frac{b^{d+1}}{(b-1)} - b(d-1).$$

This is exactly the recurrence for DFID except for the savings of $b(d-1)$ nodes on each iteration. Thus, the total savings over DFID is

$$\sum_{i=1}^{d} b(i-1) = b\sum_{i=0}^{d-1} i = \tfrac{1}{2}bd(d-1) = O(d^2).$$

Since the asymptotic complexity of DFID is $O(b^d)$ [10], and RBFS generates $O(d^2)$ fewer nodes than DFID, the asymptotic complexity of RBFS is also $O(b^d)$. □

5.5.3. Comparison of "breadth-first" search algorithms

If we compare the four best-first search algorithms with cost equal to depth, we find that breadth-first search (BFS), DFID, and RBFS all have asymptotic complexity $O(b^d)$, while SRBFS is $O(x^d)$ where $x > b$ and approaches $b + 1$ for large b. BFS generates the fewest nodes, and the ratio of the nodes generated by DFID to those generated by BFS is approximately $b/(b-1)$. RBFS generates $O(d^2)$ fewer nodes than DFID.

5.6. RBFS versus iterative deepening with monotonic costs

How do RBFS and iterative deepening compare in general? If the cost function is nonmonotonic, the two algorithms are not directly comparable since they explore different parts of the tree and return different solutions. With a monotonic cost function, however, they both explore the same part of the tree, and produce lowest-cost solutions, but RBFS generates fewer nodes than iterative deepening on average.

In a best-first search with a monotonic cost function, the cost of nodes expanded for the first time is monotonically nondecreasing. Each distinct cost value in the tree generates a different iteration of iterative deepening, with a threshold equal to that cost value. In each iteration, iterative deepening performs a depth-first search of $T(r,c)$, the c-bounded subtree of r, where r is the root of the tree, and c is the cutoff threshold for that iteration. We can similarly define an "iteration" of RBFS, corresponding to a cost threshold c, as that interval during which those nodes being expanded for the first time all have cost c. We begin with a necessary lemma.

Lemma 5.3. *During an iteration with a monotonic cost function, for all calls* $RBFS(n, F(n), b)$, $F(n) = c$, *where* c *is the cost threshold for that iteration.*

Proof. The proof is by induction on the number of calls to RBFS during the iteration. For the basis step, we consider the first call of an iteration. For the very first iteration, the cost threshold c is equal to the static value of the root node r, or $f(r)$. Since the initial call is $RBFS(r, f(r), \infty)$, $F(n) = f(r) = c$, the cost threshold for that iteration.

At the end of each iteration, since all nodes of a given cost c have been expanded, but no nodes of greater cost, RBFS has explored $T(r,c)$, the c-bounded subtree of the root node r. At this point, the $F(n)$ values of all nodes n on the stack exceed c. The minimum of these $F(n)$ values, $MF(r,c) = c'$, is the cost threshold for the next iteration. The first call to RBFS in the new iteration will be on one of these nodes with $F(n) = c'$. Thus, for the first call in any iteration, $RBFS(n, F(n), b)$, $F(n) = c$, the cost threshold for the iteration.

The induction hypothesis is that for the first k calls to RBFS in an iteration with cost threshold c, $RBFS(n,F(n),b)$, $F(n) = c$. Now consider the $(k+1)$st call during the iteration, $RBFS(n',F(n'),b')$. Its parent call, $RBFS(n,F(n),b)$, was one of the first k calls, and hence by the induction hypothesis, $F(n) = c$. Consider the initial F value of n', $F(n')$. If $f(n) < F(n)$ and $f(n') < F(n)$, then $F(n') = F(n)$. Otherwise, if $f(n) \geqslant F(n)$, then $F(n') = f(n')$. With a monotonic cost function, $f(n') \geqslant f(n)$. Thus, $F(n') = f(n') \geqslant f(n) \geqslant F(n)$. Therefore, in either case, the first F value of n', $F(n') \geqslant F(n)$. Since the sequence of F values of a child node is strictly increasing, at any point $F(n') \geqslant F(n) = c$.

Now consider a recursive call $RBFS(n',F(n'),b')$ during the current iteration with cost threshold c. If $F(n') > c$, since $b' \geqslant F(n')$ by Lemma 4.1, $b' > c$. Since, by Lemma 4.3, $F(n') \leqslant OD(n')$, and $b' \leqslant ON(n')$, all open nodes m must have $f(m) > c$. In that case, we are no longer in the iteration with cost threshold c. Thus, during this iteration, $F(n') \leqslant c$. Since $F(n') \geqslant F(n) = c$, and $F(n') \leqslant c$, $F(n') = c$. Therefore, by induction, for all calls $RBFS(n,F(n),b)$ during an iteration with cost threshold c, $F(n) = c$. □

Lemma 5.4. *No node is expanded more than once during an iteration of RBFS.*

Proof. Assume the contrary, that a node is expanded more than once during an iteration. If such a node exists, there must be a shallowest such node n' closest to the root. Let n be its parent node. Since n' is a shallowest node expanded more than once, its parent n must only be expanded once. Thus, the multiple expansions of n' must occur during the same call to RBFS on its parent node n. Consider the first expansion of n', which occurs with a call $RBFS(n',F(n'),b')$. If the cost threshold of the iteration is c, then by Lemma 5.3, $F(n') = c$. By Lemma 4.1, $F(n') \leqslant b'$. This call will explore $T(n',b')$, and return as the new $F(n')$ value $MF(n',b')$ which is $> b'$. Thus, the next, and any subsequent, calls on n' must be $RBFS(n',F(n'),b')$, with $F(n') > c$. But by Lemma 5.3, such a call cannot be in the same iteration with cost threshold c, thus violating our assumption that n' is expanded twice in the same iteration. □

Theorem 5.5. *With a monotonic cost function, RBFS generates fewer nodes than iterative deepening, up to tie-breaking among nodes whose cost equals the solution cost.*

Proof. In each iteration, iterative deepening performs a depth-first search of $T(r,c)$, a c-bounded subtree of r, where r is the root of the tree, and c is the cutoff threshold for that iteration. Since RBFS is a best-first search, at the end of an "iteration" that expands all nodes of cost c, RBFS has also explored $T(r,c)$. Furthermore, by Lemma 5.4, RBFS doesn't expand any node twice during the same iteration. The difference between iterative deepening and RBFS is that iterative deepening performs a depth-first search of $T(r,c)$, starting at the root r of the tree, whereas RBFS performs depth-first searches of $T(n,c)$ for those nodes n on the stack at the end of the last iteration, for which $F(n) = c$. If nodes of a given cost are clustered together in the tree, then significant savings will be realized by RBFS over iterative deepening. Even in the worst case, when no such clustering exists, as for example with cost equal to depth, the nodes on the stack at the end of the last iteration are not regenerated by RBFS, while they are regenerated by iterative deepening. Thus, on all iterations except the last one, RBFS expands strictly fewer nodes than iterative deepening.

On the last iteration, however, the one that finds the goal, both iterative deepening and RBFS terminate as soon as a goal node is chosen for expansion. Since tie-breaking among nodes of equal cost is performed differently by the two algorithms, either may find a goal first, and hence expand fewer nodes on the last iteration. Thus, we can only conclude that with a monotonic cost function, RBFS expands fewer nodes than iterative deepening, up to tie-breaking among nodes whose cost equals the goal cost. □

This difference in node generations may or may not be significant, depending on the problem. For example, in the above example of cost equal to depth, the difference between RBFS and DFID is not significant, whereas on the traveling salesman problem, discussed below, the difference is significant.

5.7. *Average case analysis: monotonic costs*

Finally, we refer the reader to an average-case analysis of IDA* with a monotonic cost function, presented in [29]. The analytic model consists of a tree with uniform branching factor and independent and identically distributed nonnegative edge costs, chosen from an arbitrary discrete probability distribution. The cost of a node is the sum of the edge costs from the root to the given node, guaranteeing monotonicity. In this model, iterative deepening is asymptotically optimal, meaning that it generates asymptotically the same number of nodes as classical best-first search. Since RBFS generates fewer nodes than iterative deepening on average, it follows that RBFS is asymptotically optimal in this model as well.

6. Experimental results

We implemented RBFS, along with iterative deepening and A* for comparison, on the traveling salesman problem and the sliding-tile puzzles.

6.1. Traveling salesman problem

To demonstrate the superiority of RBFS over iterative deepening with a monotonic cost function, we compared RBFS to IDA* on the Euclidean travelling salesman problem, using the A* evaluation function, $f(n) = g(n) + h(n)$. The heuristic function used was the minimum spanning tree (MST) of the cities not yet visited. Since the MST is a lower bound on the cost of a TSP tour, both IDA* and RBFS find optimal solutions. With 10, 11, and 12 cities, RBFS generated an average of 16%, 16%, and 18% of the nodes generated by IDA*, respectively. The time per node generation was roughly the same for the two algorithms, since the cost of computing the heuristic function dominates the running time. Additional experimental results, on the asymmetric TSP and on random trees, can be found in [29].

Unfortunately, however, both these algorithms generate significantly more nodes than depth-first branch-and-bound, with nodes ordered by the nearest neighbor heuristic. The reason is that with 16 bits to represent city positions, and for the size of problems that can be practically run, many nodes have unique cost values and there are relatively few ties. Thus, the overhead of node regenerations becomes prohibitive for both RBFS and IDA*, since a new iteration is needed for each distinct cost value. The potential advantage of IDA* and RBFS over depth-first branch-and-bound is that since the former are best-first searches, they never generate any nodes with cost greater than the optimal solution cost. However, depth-first branch-and-bound is very effective on the TSP, generating only about 10% more than the minimum number of nodes, i.e. those with cost less than the optimal solution cost.

On the sliding-tile puzzles, however, depth-first branch-and-bound is not effective, since it is difficult to find any solution at all. Thus, best-first searches are the algorithms of choice.

6.2. Optimal solutions to Eight and Fifteen Puzzles

We ran RBFS and IDA* on both the Eight and Fifteen Puzzles, using the A* evaluation function with the Manhattan distance heuristic. Both algorithms find optimal solutions. When averaged over 1000 different Eight Puzzle problem instances, RBFS generated 1.4% fewer nodes than IDA*. On the Fifteen Puzzle, however, when averaged over only 100 different problem instances, RBFS generated more nodes than IDA*. This effect is due to noise in the tie-breaking on the last iteration, however, and would disappear if a larger number of problem instances could be run. In fact, RBFS generated fewer nodes than IDA* on 51 of the 100 problems. Finding optimal solutions for the Fifteen Puzzle is expensive, however, requiring hundreds of millions of node generations per problem for either algorithm.

6.3. *Weighted evaluation function on Fifteen Puzzle*

In order to find solutions more quickly, Ira Pohl [19] proposed the *heuristic path algorithm*, a best-first search with $f(n) = (1-w)g(n) + (w)h(n)$, which we simplify to $f(n) = g(n) + Wh(n)$, where $W = w/(1-w)$. These forms are equivalent, since only the relative costs of different nodes can affect the algorithm, and not their absolute values. As W increases beyond one, Pohl conjectured that solution lengths would increase, while node generations would decrease. John Gaschnig [7] empirically studied this algorithm on the Eight Puzzle, and confirmed that increasing W resulted in finding solutions faster at the expense of increased solution length. More recently, Davis et al. [3] confirmed these results on both the Eight Puzzle and the TSP. They also proved that for W greater than one, the resulting solutions cannot exceed the optimal solutions by more than a factor of W. This gives us a guaranteed upper bound on the resulting solution quality.

On the other hand, Judea Pearl [18] showed that on an abstract tree with uniform branching factor and only a single goal node, equal weighting of g and h results in the fastest algorithm, with optimal solutions being an additional bonus. The discrepancy between these experimental and analytical results is probably due to the fact that the assumption of a single goal node is not valid in the sliding-tile puzzles or the TSP, since there is a path to a goal node below every branch of the tree.

The cost function $f(n) = g(n) + Wh(n)$ is nonmonotonic if $W > 1$, even if $h(n)$ is non-overestimating. For example, if n' is a child of n, $h(n') = h(n) - 1$, and $g(n') = g(n) + 1$, then

$$\begin{aligned} f(n') &= g(n') + Wh(n') \\ &= g(n) + 1 + W(h(n) - 1) \\ &= g(n) + Wh(n) + 1 - W \\ &= f(n) + 1 - W \\ &< f(n) \quad \text{if } W > 1. \end{aligned}$$

We compared weighted-A* (WA*), weighted-IDA* (WIDA*), and RBFS using this evaluation function on the 100 randomly generated Fifteen Puzzle problem instances from [10], varying W from 1 to 99. With $W < 3$, WA* tends to exhaust the available memory of 100,000 nodes. The raw data from these experiments are shown in Table 2. Column A gives W, expressed as the integer ratio (W_h/W_g) between the weight on h (W_h) and the weight on g (W_g). The remaining columns will be described below, with the data in each representing the average over all 100 problem instances.

Table 2
Data for Fifteen Puzzle experiments.

A	B	C	D	E	F	G	H
Weight	Solution Lengths			Nodes Generated			
Wh/Wg	WA*	WIDA*	RBFS	WA*	WIDA*	RBFS Total	RBFS New
1 / 1		53.05	53.05		363,028,090	545,370,378	453,404,715
5 / 4		53.91	53.91		93,293,373	102,864,933	20,697,285
4 / 3		54.49	54.41		32,024,611	36,091,447	9,004,518
3 / 2		56.31	55.83		7,790,462	11,026,977	3,575,447
61/39		57.35	56.61		36,216,006	43,284,480	2,521,071
31/19		58.37	57.47		8,791,249	15,836,880	1,427,179
63/37		60.03	58.63		8,235,290	15,086,563	960,442
16/9		61.53	59.35		2,690,727	6,994,463	637,965
13/7		63.71	60.77		862,412	2,415,856	443,640
33/17		66.15	61.89		874,547	4,324,086	385,924
2 / 1		68.13	62.89		229,407	600,844	263,453
67/33		68.51	63.45		826,154	3,832,495	274,698
17/8		71.23	65.07		389,118	2,292,478	219,421
69/31		74.13	66.35		408,828	2,898,425	211,229
7 / 3		77.83	68.39		173,472	613,581	179,591
71/29		80.85	69.47		144,536	1,622,023	127,011
18/7		84.19	71.05		126,360	1,219,000	117,229
73/27		88.75	72.49		78,099	1,548,946	119,954
37/13		92.81	74.35		75,578	902,793	89,253
3 / 1	78.41	98.23	77.45	22,841	59,477	172,454	93,032
19/6	79.63	102.91	78.59	21,250	54,263	606,893	65,016
77/23	81.25	107.69	80.95	19,145	58,794	767,035	75,176
39/11	83.27	114.43	82.67	18,286	71,105	694,364	72,223
79/21	85.81	121.39	84.61	17,212	78,416	685,572	61,819
4 / 1	88.15	128.89	87.43	15,819	60,450	269,289	77,429
81/19	89.29	136.21	89.33	14,254	54,034	798,793	72,999
41/9	91.25	145.13	91.17	12,406	98,344	706,827	73,310
83/17	93.11	154.59	92.81	12,975	91,683	789,684	73,568
5 / 1	97.31	160.15	95.83	12,689	67,989	206,435	84,060
21/4	97.80	166.85	97.41	13,163	99,234	735,584	76,744
17/3	101.63	179.21	99.47	11,445	55,102	335,885	63,822
6 / 1	103.29	189.41	101.93	10,460	79,259	306,088	72,973
43/7	103.31	193.07	102.27	10,523	80,397	543,133	59,802
87/13	106.09	210.11	105.15	10,050	65,058	985,480	92,381
7 / 1	109.23	219.91	107.31	10,787	57,218	270,845	94,803
22/3	109.83	230.67	109.33	11,070	63,768	844,607	95,074
8 / 1	112.55	251.07	111.91	10,772	103,136	617,906	125,637
89/11	112.85	254.35	112.33	10,957	141,098	1,280,954	126,251
9 / 1	116.49	281.63	114.93	9,528	101,383	472,465	144,092
91/9	119.21	314.31	118.19	7,809	190,633	3,694,920	352,669
23/2	120.09	356.35	121.83	7,474	273,991	3,919,395	465,035
93/7	121.99	407.79	122.79	7,569	324,811	6,666,693	692,032
47/3	124.15	481.65	126.45	7,743	521,441	6,304,235	824,769
19/1	127.65	580.93	130.59	7,925	1,224,161	7,275,106	1,670,287
24/1	132.11	736.45	132.89	8,250	7,207,605	20,577,463	2,880,395
97/3	135.33	996.23	137.71	7,076	7,269,718	55,927,351	6,077,104
49/1	144.29			7,002			
99/1	145.27			6,972			
	145.27			6,972			

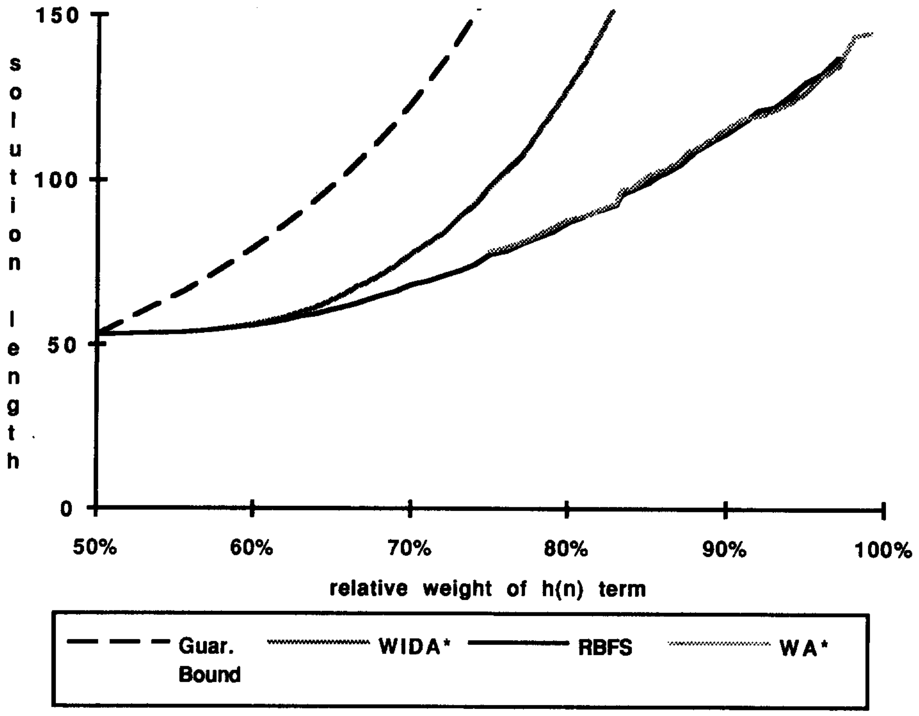

Fig. 8. Solutions lengths for WIDA*, RBFS, and WA* on the Fifteen Puzzle.

6.3.1. Solution lengths

Figure 8 shows the solution lengths produced by the different algorithms. The horizontal axis shows the percentage of total weight on the $h(n)$ term, which is $100 \times W_h/(W_h + W_g)$. The top line shows the guaranteed bound, which is W times the average optimal solution length (53.05), the middle line shows the solution lengths returned by WIDA* (column C), and the bottom two lines show the solution quality of WA* and RBFS (columns B and D, respectively).

All three algorithms produce better solutions than the guaranteed bound, and for small values of W, they produce nearly optimal solutions whose lengths grow very slowly with W. As expected, RBFS produces solutions of the same average quality as WA* throughout. The reason they are not exactly the same is that the two algorithms break ties differently, and with a nonmonotonic cost function, tie-breaking can affect solution quality on individual problems. WIDA*, however, produces longer solutions than RBFS and WA*, and the difference between them increases with increasing W. The reason is that with a nonmonotonic cost function, in those regions of the search tree where the cost of a node is less than the maximum cost of any of its ancestors, WIDA* searches depth-first, ignoring cost as long as it is less than the current cutoff threshold, whereas RBFS and WA* search

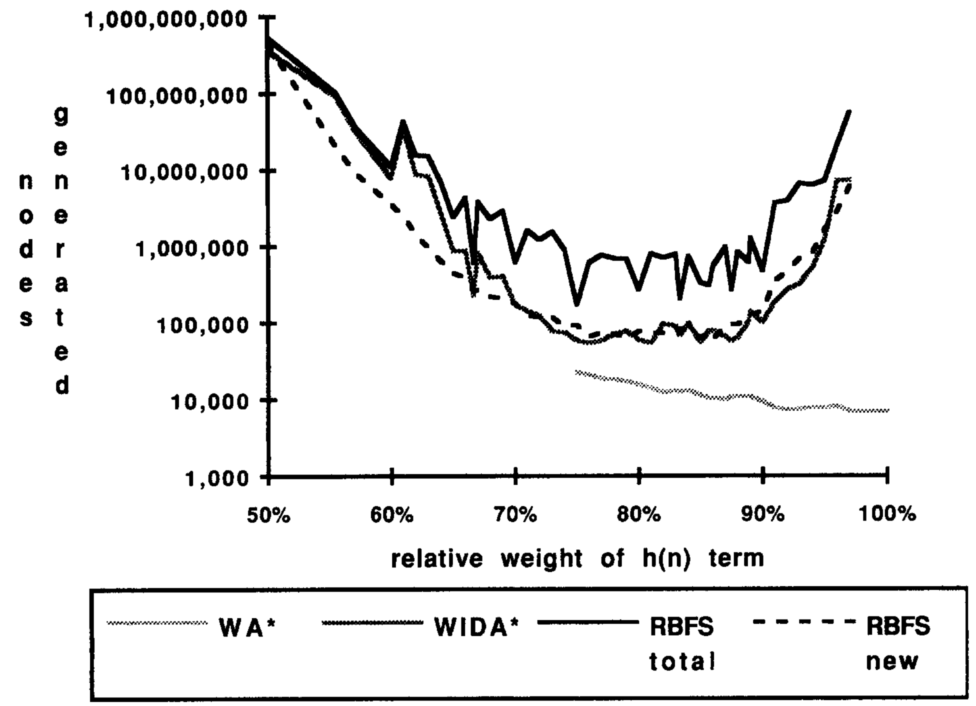

Fig. 9. Nodes generated by WIDA*, RBFS, and WA* on the Fifteen Puzzle.

best-first, always expanding next a lowest-cost open node. The slope of the WIDA* solution line continues to increase with W, and approaches vertical for large values of W (see column C). With 97% of the weight on h, for example, the average solution found by WIDA* is 996 moves, while WA* and RBFS find solutions of about 136 moves.

6.3.2. Nodes generated

Figure 9 shows the number of nodes generated by each of the algorithms as a function of the relative weight on h, on a logarithmic scale.

The bottom line shows the nodes generated by WA* (column E). It starts at 75% ($W = 3$), since it tends to exhaust the available memory with smaller weights on h. At $W = 3$, WA* generates more than four orders of magnitude fewer nodes than IDA* (23 thousand versus 363 million), while producing solutions that are only 48% longer than optimal (78.41 versus 53.05 moves). As the weight on h increases, the number of nodes generated continues to decrease. With all the weight on h, meaning that g is used only for tie-breaking among nodes with equal h values, the number of nodes generated drops to about 7,000, while the solution lengths average 145 moves.

Figure 9 also shows the number of nodes generated by WIDA* (column F) and RBFS (column G). As with WA*, increasing the weight on h decreases the number of nodes generated, at least up to 75%. Both algorithms find nearly optimal solutions with dramatic reductions in nodes generated. For example, With $W = 2$ (66.67%), RBFS returns solutions that are only 19% longer than optimal (62.89 versus 53.05 moves), with almost three orders of magnitude reduction in nodes generated (600 thousand versus 363 million).

Both WIDA* and RBFS generate significantly more nodes than WA*, however. There are two reasons for this. The first is that unlike WA*, WIDA* and RBFS regenerate many of the same nodes on successive iterations.

The second and more significant reason is that on a graph with cycles, such as the sliding-tile puzzle problem space, exponential-space algorithms such as WA* detect and prune multiple copies of the same node. The linear-space algorithms, however, WA* and RBFS, cannot detect duplicate nodes, but view the same node generated via two different paths as being two separate nodes. This can increase the asymptotic branching factor of a linear-space search. For example, a depth-first search of a square grid has time complexity $O(3^r)$, where r is the search radius, since each node has three new neighbors. Conversely, a breadth-first search of the same graph has complexity $O(r^2)$, since the number of nodes grows only quadratically with the radius. See [27] for a solution to this problem.

The nodes generated by WIDA* and RBFS decrease with increasing weight on h up to about 75%, but beyond about 85%, the number of nodes increases with increasing W. Two opposing effects cause this drop and subsequent rise. Increasing W reduces the number of nodes generated by pruning large parts of the search tree. At the same time, increasing W causes the remaining paths to be searched more deeply. As the depth of the search increases, the difference between the number of nodes in the graph and those in the tree increases, due to undetected duplicate nodes. While the pruning effect predominates for small values of W, this effect diminishes with increasing W, as shown by the WA* line, and eventually the duplicate effect takes over and increases the total number of nodes. This "duplicate bloom" can be largely eliminated, however [27].

RBFS, however, generates more nodes than WIDA* for the same value of W in this domain, on average. In the case of $W = 1$, where WIDA* becomes IDA*, the difference is due to tie-breaking noise as explained previously. With a nonmonotonic cost function, however, either algorithm could generate more nodes, since they search different parts of the tree. In regions of the tree where the cost of a node is less than the maximum cost of all its ancestors, WIDA* ignores costs and proceeds depth-first until a goal is found, expanding significantly more nodes than RBFS in some problem instances. On the other hand, by carefully searching best-first, RBFS searches more nodes than WIDA* in some instances, due to node reexpansions. The actual performance depends on the problem domain and problem instance.

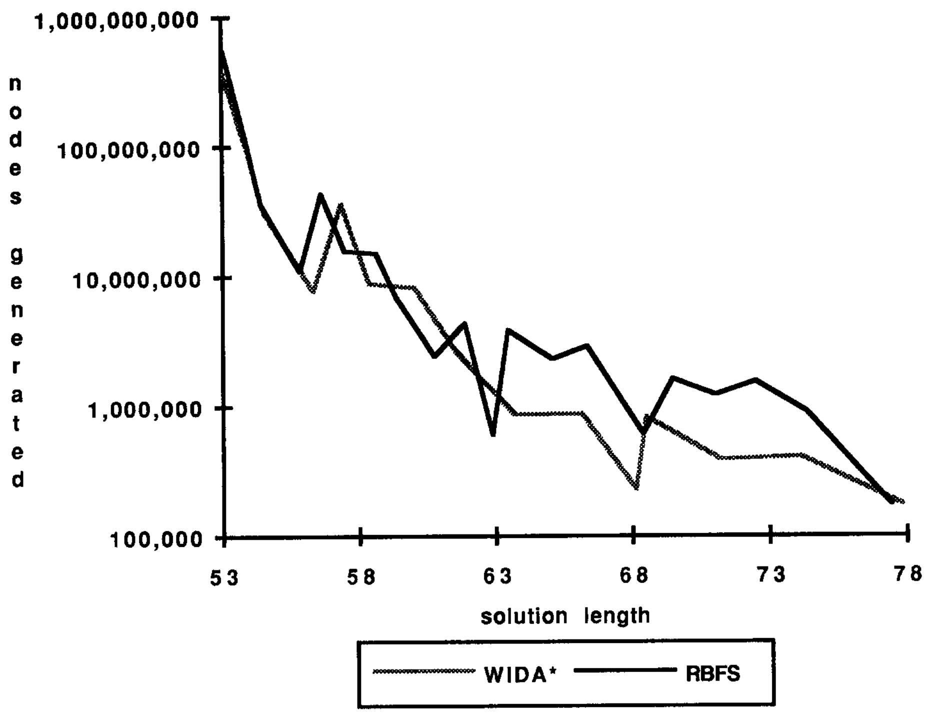

Fig. 10. Nodes generated by WIDA* and RBFS versus solution length.

On the Fifteen Puzzle, with small values of W, both algorithms produce nearly optimal solutions, and their node generations are fairly close as well, while for larger values of W, both solution lengths and node generations diverge. Since WIDA* produces longer solutions than RBFS, Fig. 10 presents a fairer comparison of the two algorithms, by plotting node generations as a function of solution length, for W up to 3. For $W > 3$, WA* would be the algorithm of choice. Figure 10 shows that the two algorithms are roughly comparable in this domain, with neither completely dominating the other.

6.3.3. Node regeneration overhead

Recall that RBFS can tell the difference between the first time a node is generated, and when it is being regenerated. If the static value of a node, $f(n)$, is equal to its stored value, $F(n)$, then the node has not been expanded before, but if $f(n) < F(n)$, then n has previously been expanded. If a node's parent has not been expanded before, the child node is being generated for the first time, and otherwise it is being regenerated. The dashed line in Fig. 9 (column H) records the number of nodes generated for the first time by RBFS, as opposed to node regenerations.

On a tree, the average number of new nodes generated by RBFS would be the same as the number of nodes generated by WA*. On a graph with cycles, however, they can be significantly different, as shown by Fig. 9. The reason is that the same node regenerated via a different path is viewed as a new node by RBFS, while WA* detects and prunes such duplicate nodes.

While the number of new nodes varies smoothly with W, the total number of nodes generated by RBFS is quite jagged. The reason is that the node regeneration overhead, defined as the number of node regenerations divided by new node generations, varies from a low of 0.2 with 50% of the weight on h, to a high of 16.17 with 61% of the weight on h. The reason is as follows.

The actual evaluation function used by all three algorithms is $f(n) = W_g g(n) + W_h h(n)$, where W_g and W_h are the smallest integers that produce a given value of W, which is W_h/W_g. Column A of Table 2 shows these values. For example, with 50% relative weight on h, $W_h = W_g = 1$, while for 61%, $W_h = 61$ and $W_g = 39$. The reason for the large difference in node regeneration overhead between the 50% case and the 61% case is that the 1:1 weight ratio produces many more ties between nodes with different g and h values than the 61:39 ratio. As a general rule, the smaller the values of W_g and W_h when expressed in lowest terms, the greater the number of ties, and the lower the node regeneration overhead. The best values for W_h and W_g are $W_h = k$ and $W_g = k - 1$, for small relative weights on h, and $W_h = k$ and $W_g = 1$ for larger relative weights on h. The downward spikes in the RBFS node generation curves in Figs. 9 and 10 correspond to weights chosen from this set.

For a given value of W, the number of nodes generated in individual problem instances varies over three orders of magnitude. In spite of this, however, the node regeneration overhead remains remarkably consistent. This suggests that while the node regeneration overhead varies considerably for different values of W, for a given value of W, this overhead is a constant, independent of problem size. This is further supported by the data on the Twenty-Four Puzzle, presented below.

6.4. *Twenty-Four Puzzle*

While computationally expensive, finding optimal solutions to the Fifteen Puzzle is feasible with current machines. Therefore, as a more difficult test of RBFS, WIDA*, and WA*, we ran them on the larger 5×5 Twenty-Four Puzzle. With the ability to store up to 100,000 states, WA* was feasible only with all the weight on the h term, using g to break ties. On 100 randomly generated problem instances, this pure heuristic search algorithm found solutions that averaged 339 moves, while generating an average of 96,268 nodes per problem instance.

In order to find shorter solutions, we ran RBFS and WIDA* on 1000 randomly generated Twenty-Four Puzzle problem instances, with $W = 3$, the weight that minimized node generations for the Fifteen Puzzle. 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*. RBFS found a shorter solution than WIDA* on every problem instance, and generated fewer nodes in 43% of the problem instances.

The node regeneration overhead on individual problems was a relatively constant 85%, in spite of over six orders of magnitude variation in nodes generated. This is the same node regeneration overhead for $W = 3$ as in the Fifteen Puzzle, further supporting the claim that the node regeneration overhead of RBFS is a constant, independent of problem size.

As far as we know, there are only three other algorithms that can solve the Twenty-Four Puzzle within practical resource limits. The first is Real-Time-A* (RTA*) [11]. RTA* commits itself to each move in constant time, after looking ahead in the problem space to a fixed search horizon. The greater the search horizon, the shorter the solution lengths, and the greater the node generations. To provide a fair comparison to RBFS, we modified RTA* to report solution lengths with all loops and backtracking removed, and to break ties in favor of nodes closest to the initial state. With a search horizon of 40 moves, RTA* generated more nodes than RBFS with $W = 3$ (242 versus 93 million), and produced longer solutions (236 versus 169 moves).

The other algorithms, Heuristic Subgoal Search [12] and Stepping Stone [22], use more knowledge in the form of a sequence of subgoals, and scale up to much larger problems. These algorithms correctly position the tiles one at a time, and resolve impasses with additional search. While they are very efficient, the solutions they generate average 270 moves for the Twenty-Four Puzzle, compared to 169 moves for RBFS, and these solutions cannot be improved with more computation. Thus, RBFS finds the shortest solutions to the Twenty-Four Puzzle. By decreasing the value of W to 2 or less, even shorter solutions could be found, but at significant computational cost.

6.5. Constant factors

While the asymptotic complexity of RBFS and WIDA* is linear in the number of node generations, constant factors are often important as well. One measure of the speed of a search program is the number of node generations per minute. We measured the speed of IDA*, WIDA*, WA*, and RBFS on the Fifteen Puzzle, in the latter three cases with the weighted evaluation function with $W = 3$. The fastest program is IDA*, generating about 3.5 million nodes per minute on a Hewlett-Packard 9000, model 350 workstation. Next was WIDA*, generating about 2.1 million nodes per minute. The main reason for this difference is that for IDA* on the Fifteen Puzzle, we can predict a priori what the successive threshold values will be, whereas WIDA* must calculate the next threshold during the current iteration. RBFS generates about 1.5 million nodes per minute, making it about 29% slower than WIDA*. Finally, WA* only generates about 470 thousand nodes per minute, making it more than a factor of three slower than RBFS. The magnitude of these constant factor differences is due to the fact that node generation and evaluation is very efficient for the sliding-tile puzzles. In a problem where node generation and/or evaluation were more expensive, the differences in the constant factors would be smaller.

7. Related work

A number of other memory-limited search algorithms, discussed below, have been designed to reduce the node regeneration overhead of IDA*. The most important difference between these algorithms and RBFS is that none of the other memory-limited 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 to reduce its node regeneration overhead as well.

7.1. MREC

The MREC algorithm [25] executes A* until memory is almost full, then performs IDA* below the stored frontier nodes. Duplicate nodes are detected by comparing them against the stored nodes. Unfortunately, unless a large fraction of the search space can be stored in memory, which is impossible for large problems, the savings from detecting duplicate nodes may not outweigh the additional overhead incurred in checking for them. In Sen and Bagchi's experiments on the Fifteen Puzzle [25], they did not check for duplicate nodes, and achieved speeds comparable to IDA*, but only a 1% reduction in node generations. We implemented MREC on the Fifteen Puzzle, but checked for duplicates. By storing 100,000 nodes, the number of node generations was reduced by 41% compared to IDA*, but MREC ran 64% slower per node than IDA*. MREC could be combined with RBFS by maintaining a table of previously generated nodes, and checking for duplicates among newly generated nodes.

7.2. MA, SMA*, and ITS*

While MREC statically allocates its memory to the first nodes generated, the MA* algorithm [2] dynamically stores the best nodes generated so far, according to the cost function. It behaves like A* until memory is almost full. Then, in order to free space to continue, it prunes a node of highest cost on the open list, updates the value of its parent to the minimum of its children's values, and places the parent back on the open list. Unfortunately, the constant overhead of this algorithm is prohibitively expensive. While the authors report no actual running times, their data on the Fifteen Puzzle represent easy problem instances. Our implementation of MA* on the Fifteen Puzzle runs 20 times slower than IDA*, making it impractical for solving randomly generated problem instances. Both SMA* [23] and ITS [16] represent attempts to significantly reduce the constant factor overhead of MA*. If it could be made practical, MA* could be combined with RBFS by storing more of the tree than the current path, and checking newly generated nodes for duplicates.

7.3. DFS, IDA*-CR, and MIDA**

DFS* [20], IDA*-CR [24], and MIDA* [28] are very similar algorithms, independently developed. All three attempt to reduce the node regeneration overhead of IDA* by setting successive thresholds to values larger than the minimum value that exceeded the previous threshold. This reduces the number of different iterations by combining several together. In order to guarantee optimal solutions, once a goal is found, the algorithms revert to depth-first branch-and-bound, pruning any nodes whose cost equals or exceeds that of the best solution found so far. Setting the best threshold values requires some finesse, since values that are too large will result in excessive node generations on the final iteration. This idea can be applied to RBFS by setting the upper bound on a recursive call to the minimum of the parent's upper bound and a value slightly larger than the value of the best remaining brother. Once a solution is found, the algorithm continues to run with the upper bounds on all remaining recursive calls set to the cost of the best solution found so far.

7.4. Iterative expansion

The algorithm that comes closest to RBFS is Iterative Expansion (IE) [23], which was developed independently. IE is very similar 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 nonmonotonic cost function, but behaves similar to RBFS for the special case of a monotonic cost function. The performance of IE on our sliding-tile puzzle experiments should approximate that of WIDA*.

7.5. Bratko's best-first search

Ivan Bratko [1] anticipated many of the ideas in RBFS, IE, MA*, and MREC, in his alternative formulation of best-first search. In particular, he first introduced the ideas of recursively generating the best frontier node instead of using an open list, using the value of the next best brother as an upper bound, and backing up the minimum values of children to their parents. The main difference between Bratko's algorithm and these others is his use of exponential space. Apparently, most of these authors, including this one, were unaware of Bratko's earlier work.

8. Conclusions

Recursive best-first search is a general heuristic search algorithm that runs in linear space, and expands nodes in best-first order, even when the cost function is nonmonotonic. It achieves this by expanding some nodes more than once. Since it is more efficient per node generation than standard best-first search, however, it may run faster overall on problems where node generation and evaluation are very efficient. For the special case where cost is equal to depth, RBFS is asymptotically optimal in space and time, generating $O(b^d)$ nodes. In general, with a monotonic cost function, it finds optimal solutions, while expanding fewer nodes than iterative deepening on average, although the difference is not significant in all cases. Furthermore, on a particular abstract model, both iterative deepening and RBFS are asymptotically optimal in time.

We performed extensive experiments on sliding-tile puzzles using the nonmonotonic weighted cost function, $f(n) = g(n) + Wh(n)$. We showed that small values of $W > 1$ produce dramatic reductions in node generations with only small increases in solution lengths, extending this result to significantly larger problems. For the same value of W, RBFS always produced shorter solutions than iterative deepening, but generated more nodes on average. The number of nodes generated by RBFS was only a constant multiple of those 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, while increasing the time complexity by only a constant factor in our experiments.

Acknowledgement

This research was supported by an NSF Presidential Young Investigator Award, NSF Grant IRI-9119825, and a grant from Rockwell International. Thanks to Valerie Aylett for drawing the graphs and figures, and to Cindy Mason, Joe Pemberton, Ed Purcell, and Larry Taylor for their comments on earlier drafts of this manuscript.

References

[1] I. Bratko, *PROLOG: Programming for Artificial Intelligence* (Addison-Wesley, Reading, MA, 1986) 265–273.

[2] P.P. Chakrabarti, S. Ghose, A. Acharya and S.C. de Sarkar, Heuristic search in restricted memory, *Artif. Intell.* **41** (2) (1989) 197–221.

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

[4] R. Dechter and J. Pearl, Generalized best-first search strategies and the optimality of A*, *J. ACM* **32** (3) (1985) 505–536.

[5] E.W. Dijkstra, A note on two problems in connexion with graphs, *Numer. Math.* **1** (1959) 269–271.

[6] J.E. Doran and D. Michie, Experiments with the Graph Traverser program, *Proc. Roy. Soc. A* **294** (1966) 235–259.

[7] J. Gaschnig, Performance measurement and analysis of certain search algorithms, Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA (1979).

[8] P.E. Hart, N.J. Nilsson and B. Raphael, A formal basis for the heuristic determination of minimum cost paths, *IEEE Trans. Syst. Sci. Cybern.* **4** (2) (1968) 100–107.

[9] R.M. Karp, Reducibility among combinatorial problems, in: R.E. Miller and J.W. Thatcher, eds., *Complexity of Computer Computations* (Plenum, New York, 1972) 85–103.

[10] R.E. Korf, Depth-first iterative-deepening: an optimal admissible tree search, *Artif. Intell.* **27** (1) (1985) 97–109.

[11] R.E. Korf, Real-time heuristic search, *Artif. Intell.* **42** (2–3) (1990) 189–211.

[12] R.E. Korf, Real-time search for dynamic planning, in: *Proceedings AAAI Spring Symposium on Planning in Uncertain, Unpredictable, or Changing Environments*, Stanford, CA (1990) 72–76.

[13] R.E. Korf, Best-first search in limited memory, in: *UCLA Computer Science Annual* (University of California, Los Angeles, CA, 1991) 5–22.

[14] R.E. Korf, Linear-space best-first search: extended abstract, in: *Proceedings Sixth International Symposium on Computer and Information Sciences*, Antalya, Turkey (1991) 581–584.

[15] R.E. Korf, Linear-space best-first search: summary of results, in: *Proceedings AAAI-92*, San Jose, CA (1992) 533–538.

[16] A. Mahanti, D.S. Nau, S. Ghosh and L.N. Kanal, An efficient iterative threshold heuristic tree search algorithm, Tech. Report UMIACS TR 92-29, CS TR 2853, Computer Science Department, University of Maryland, College Park, MD (1992).

[17] B.G. Patrick, M. Almulla and M.M. Newborn, An upper bound on the complexity of iterative-deepening-A*, in: *Proceedings Symposium on Artificial Intelligence and Mathematics*, Ft. Lauderdale, FL (1989).

[18] J. Pearl, *Heuristics* (Addison-Wesley, Reading, MA, 1984).

[19] I. Pohl, Heuristic search viewed as path finding in a graph, *Artif. Intell.* **1** (1970) 193–204.

[20] V.N. Rao, V. Kumar and R.E. Korf, Depth-first vs. best-first search, in: *Proceedings AAAI-91*, Anaheim, CA (1991) 434–440.

[21] D. Ratner and M. Warmuth, Finding a shortest solution for the $N \times N$ extension of the 15-Puzzle is intractable, in: *Proceedings AAAI-86*, Philadelphia, PA (1986) 168–172.

[22] D. Ruby and D. Kibler, Learning subgoal sequences for planning, in: *Proceedings IJCAI-89*, Detroit, MI (1989) 609–614.

[23] S. Russell, Efficient memory-bounded search methods, in: *Proceedings ECAI-92*, Vienna, Austria (1992).

[24] U.K. Sarkar, P.P. Chakrabarti, S. Ghose and S.C. DeSarkar, Reducing reexpansions in iterative-deepening search by controlling cutoff bounds, *Artif. Intell.* **50** (2) (1991) 207–221.

[25] A.K. Sen and A. Bagchi, Fast recursive formulations for best-first search that allow controlled use of memory, in: *Proceedings IJCAI-89*, Detroit, MI (1989) 297–302.

[26] M.E. Stickel and W.M. Tyson, An analysis of consecutively bounded depth-first search with applications in automated deduction, in: *Proceedings IJCAI-85*, Los Angeles, CA (1985) 1073–1075.

[27] L. Taylor and R.E. Korf, Pruning duplicate nodes in depth-first search, in: *Proceedings AAAI-93*, Washington, DC (1993).

[28] B.W. Wah, MIDA*, an IDA* search with dynamic control, Tech. Report UILU-ENG-91-2216, CRHC-91-9, Center for Reliable and High-Performance Computing, Coordinated Science Laboratory, College of Engineering, University of Illinois at Urbana, Champaign-Urbana, IL (1991).

[29] W. Zhang and R.E. Korf, Depth-first vs. best-first search: new results, in: *Proceedings AAAI-93*, Washington, DC (1993).