

TABLE 2.3. Iterative deepening algorithm.

<pre> ID(<i>root</i>) <i>threshold</i> ← <i>cost</i>(<i>root</i>); <i>next_threshold</i> ← ∞; REPEAT DFS(<i>root</i>); <i>threshold</i> ← <i>next_threshold</i>; <i>next_threshold</i> ← ∞. DFS(<i>n</i>) FOR (each child <i>n_i</i> of <i>n</i>) IF (<i>n_i</i> is a goal node and <i>cost</i>(<i>n_i</i>) ≤ <i>threshold</i>) EXIT with optimal goal node <i>n_i</i>; IF (<i>cost</i>(<i>n_i</i>) ≤ <i>threshold</i>) DFS(<i>n_i</i>); ELSE IF (<i>cost</i>(<i>n_i</i>) < <i>next_threshold</i>) <i>next_threshold</i> ← <i>cost</i>(<i>n_i</i>); RETURN. </pre>

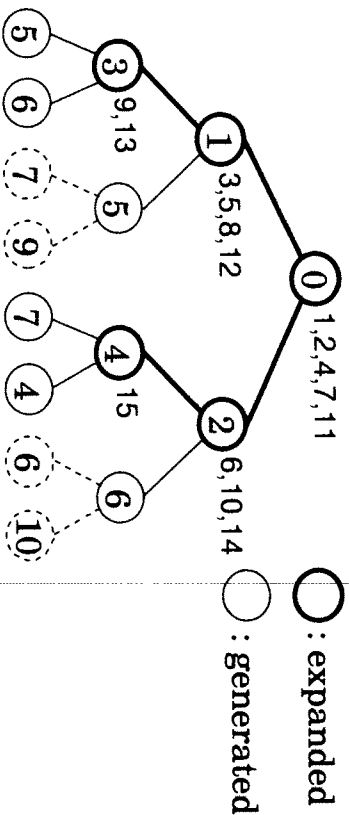


FIGURE 2.5. The order of node expansions of iterative deepening.

path algorithm [28], uses the sum of the costs on the path from the root to a node as the node cost. Iterative deepening A* (IDDA*) employs the A* cost function $f(n) = g(n) + h(n)$.

Although iterative deepening will not expand a node whose cost is greater than the cost of the optimal goal node, it may expand some nodes more than once, as shown in Figure 2.5. Furthermore, iterative deepening does not expand new, unexplored nodes in best-first order.

2.1.6 Recursive best-first search

Recursive best-first search (RBFS) [76] always expands unexplored nodes in best-first order, but still runs in a space that is linear in the search

depth. It is also more efficient than iterative deepening with a monotonic cost function [76].

The key difference between recursive best-first search and iterative deepening is that while iterative deepening maintains a single global cutoff threshold for the whole tree, recursive best-first search computes a separate local cutoff threshold for each subtree of the current search path. Instead of using a global bound, recursive best-first search uses two *local* bounds, upper bound $\alpha(n)$ and lower bound $\beta(n)$, to a node n currently being explored. The lower bound $\beta(n)$ is the minimum cost of all frontier nodes in the subtree $T(n)$ rooted at n . If a node has not been visited before, the initial lower bound to the node is its cost. The upper bound $\alpha(n)$ is the minimum cost of all frontier nodes in the whole currently expanded tree except those in $T(n)$. To expand new, unexplored nodes in best-first order, recursive best-first search examines a subtree $T'(n)$ if it has such a frontier node that has the minimum cost among all the frontier nodes of the search space. In other words, recursive best-first search explores $T'(n)$ when $\beta(n) \leq \alpha(n)$. The upper bound $\alpha(n)$ is passed along to n from its parent, and is the minimum of the upper bound on n 's parent and the all lower bounds of n 's siblings. The lower bound $\beta(n)$ is percolated upward from the children of n , and is the minimum of the lower bounds of n 's children.

The algorithm starts at the root node as the current node n . The initial upper bound is set to infinity, meaning that n does not have siblings, and the initial lower bound is set to the cost of n , meaning that n has not been visited before. The algorithm then generates all the children of the current node n , computes their upper and lower bounds, and updates the lower bound of n to be the minimum of the lower bounds of the children. If the node-exploration condition, $\beta(n) \leq \alpha(n)$, still holds for node n , the algorithm then selects a child node of n which has the minimum lower bound among all children as the next current node, and the procedure of expanding a node and updating bounds repeats. Whenever the node-exploration condition is violated, the lower bound $\beta(n)$ is percolated upward to the parent of n , and the algorithm subsequently backtracks to the parent node of n , and updates the lower bound of the parent of n accordingly. The algorithm terminates when a goal node is chosen for expansion. Pseudocode of the recursive best-first search algorithm is given in Table 2.4. The initial call is $\text{RBFS}(\text{root}, \text{cost}(\text{root}), \infty)$.

We now describe the behavior of recursive best-first search on the tree in Figure 2.2 (see Figure 2.6). After the root is expanded (see Figure 2.6), recursive best-first search is called recursively on the left child with an upper bound of 2, the value of the right child. This value is chosen because the best frontier node will be in the left subtree as long as its value is less than or equal to the upper bound 2, the value of the best frontier node in the right subtree. After the left child is expanded, the values of its two children (3 and 5) exceed the upper bound of 2. Recursive best-first search

TABLE 2.4. Recursive best-first search algorithm.

```

RBFS( $n_i, \beta(n_i), \alpha(n_i)$ )
  IF ( $cost(n_i) > \alpha(n_i)$ )
    RETURN  $cost(n_i)$ ;
  IF ( $n_i$  is a goal)
    EXIT with optimal goal node  $n_i$ ;
  IF ( $n_i$  has no children)
    RETURN  $\infty$ ;
  FOR (each child  $n_i$  of  $n$ )
    IF ( $cost(n_i) < \beta(n_i)$ )
       $\beta(n_i) \leftarrow \text{MAX}(\beta(n_i), cost(n_i))$ ;
    ELSE  $\beta(n_i) \leftarrow cost(n_i)$ ;
  SORT  $n_i$  and  $\beta(n_i)$  in increasing order of  $\beta(n_i)$ ;
  IF (only one child)
     $\beta(n_2) \leftarrow \infty$ ;
  WHILE ( $\beta(n_1) \leq \alpha(n)$  and  $\beta(n_1) < \infty$ )
     $\beta(n_1) \leftarrow \text{RBFS}(n_1, \beta(n_1), \text{MIN}(\alpha(n), \beta(n_2)))$ ;
    INSERT  $n_1$  and  $\beta(n_1)$  in sorted order;
  RETURN  $\beta(n_1)$ .

```

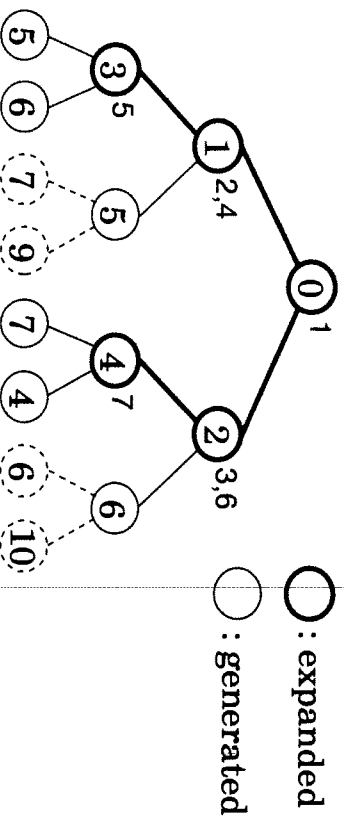


FIGURE 2.6. The order of node expansions of recursive best-first search.

must then return to the root and release the memory for the children of the left child of the root node. First, however, it backs up the minimum child value of 3 and stores that as the new lower bound of the left child of the root. After returning to the root, recursive best-first search is called recursively on the right child with an upper bound of 3, the value of the best frontier node in the left subtree. After the right child is expanded, the values of both its children (4 and 6) exceed the upper bound of 3. Recursive best-first search backs up the minimum of these values, 4, stores it as the new lower bound of the right child, and returns to the root. It then calls itself recursively on the left child with an upper bound of 4, the value of the best frontier node in the right child. After the left child and its left

child are expanded, all frontier nodes in the left child are greater than or equal to 5; therefore, 5 is stored as the new lower bound of the left child of the root and recursive best-first search is called on the right child with an upper bound of 5. It then proceeds down the right subtree until it chooses to expand the goal node of cost 4, and terminates.

At any point, recursive best-first search maintains in memory the current search path, all immediate siblings of nodes on the current path, and the cost of the best frontier nodes below each of those siblings. Thus its space complexity is linear in the search depth. An important feature of recursive best-first search is that with monotonic node costs, it generates fewer nodes than iterative deepening, up to tie-breaking. For example, recursive best-first search expands only seven nodes on the tree in Figure 2.2 (see Figure 2.6), but iterative deepening expands fifteen nodes on the same tree (see Figure 2.5). Like iterative deepening, however, recursive best-first search suffers from node reexpansion overhead.

2.1.7 Space-bounded best-first search

Depth-first branch-and-bound, iterative deepening, and recursive best-first search are linear-space search algorithms [156], since they run in space that is linear in search depth. A linear-space search algorithm needs to maintain information on the nodes on the path from the root to the node that is currently under consideration and that of their siblings. Therefore, a linear-space algorithm can be implemented using a stack, with node-expansion and node-generation operators being executed on the top of the stack. Best-first search, however, is at the opposite extreme of the space-requirement spectrum; it typically needs space that is exponential in search

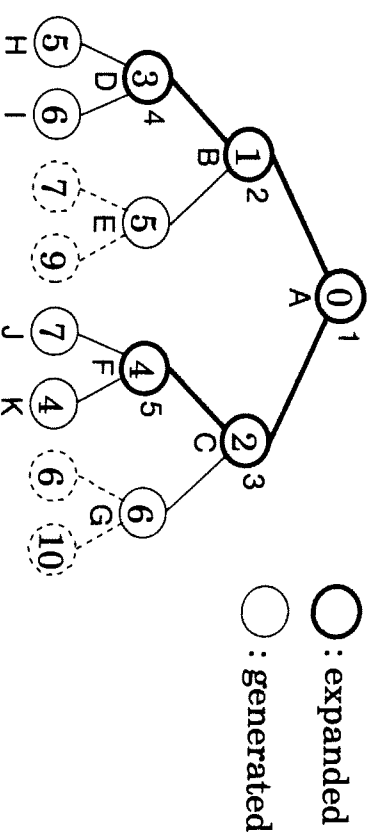


FIGURE 2.7. The order of node expansions in space-bounded best-first search with 7 memory units.

depth, since it has to maintain all the nodes that have been generated but not expanded. The main drawback of a linear-space search, on the other hand, is that it usually generates more nodes than best-first search, visiting nodes that are not explored by best-first search or expanding some nodes more than once.

A linear-space search algorithm does not fully utilize the total available space during its execution. In practice, although available space is a precious resource, it is usually more than linear in search depth. What is needed is an algorithm that can use all of the available space in order to reduce the number of node generations in linear-space search, and still explore new nodes in best-first order. Such an algorithm is MA* [16], which is refined into SMA* [127, 64]. For naming consistency and to reflect its nature, we call this algorithm space-bounded best-first search (SBFS).

TABLE 2.5. Space-bounded best-first search algorithm.

<pre> SBFS(<i>root</i>) <i>F</i>(<i>root</i>) ← <i>f</i>(<i>root</i>); <i>open</i> ← <i>root</i>; <i>used</i> ← 1; WHILE (<i>open</i> is not empty) <i>best</i> ← a deepest node of minimum <i>F</i>-cost in <i>open</i> IF (<i>best</i> is a goal node) RETURN with success; <i>child</i> ← next element of <i>ungenerated.children</i>(<i>best</i>) REMOVE <i>child</i> from <i>ungenerated.children</i>(<i>best</i>) <i>F</i>(<i>child</i>) ← <i>max</i>{<i>f</i>(<i>child</i>), <i>F</i>(<i>best</i>)} IF (all children of <i>best</i> have been generated) BACKUP(<i>best</i>); IF (all children of <i>best</i> are in memory) DELETE <i>best</i> from <i>open</i>; <i>used</i> ← <i>used</i> + 1; IF (<i>used</i> > <i>memory-size</i>) <i>worst</i> ← a shallowest, highest <i>F</i>-cost leaf in <i>open</i>; DELETE <i>worst</i>; ADD <i>worst</i> to <i>ungenerated.children</i>(<i>parent</i>(<i>worst</i>)) IF (<i>parent</i>(<i>worst</i>) ∉ <i>open</i>) INSERT <i>parent</i>(<i>worst</i>) into <i>open</i>; <i>used</i> ← <i>used</i> - 1; INSERT <i>child</i> into <i>open</i>; RETURN with failure. </pre>	<pre> BACKUP(<i>n</i>) IF (<i>completed</i>(<i>n</i>) and <i>n</i> has a parent) <i>F</i>(<i>n</i>) ← least <i>F</i>-cost of all <i>n</i>'s children; IF (<i>F</i>(<i>n</i>) changed) BACKUP(<i>n</i>). </pre>
---	--

Given a limited, constant amount of memory, space-bounded best-first search first runs best-first search until the memory is full. It then prunes the least promising part of the generated search space and reallocates memory to push forward toward the goal. More specifically, space-bounded best-first search generates one node at a time. If there is no memory available for the new node, it removes the worst leaf node in the current saved search tree. Similar to recursive best-first search, space-bounded best-first search maintains two costs on a node: static and dynamic node costs. The static cost of a node is the same as the regular node cost. The initial dynamic cost of a node is set to its static cost. After all of its children have been generated or explored, the dynamic cost of the node is revised to the minimum static cost of all frontier nodes under it, although not all of these frontier nodes must be stored in memory. This node pruning plus dynamic cost propagation is called *node retraction*. The best node is defined as the node with the minimum dynamic cost among all nodes in memory that have not been fully expanded. The worst node is defined as the leaf node with the maximum dynamic cost among all leaf nodes in memory.

The algorithm is given in Table 2.5, where $F(n)$ is the dynamic cost of node n . The initial call is SBFS(*root*). Figure 2.7 shows how space-bounded best-first search works on the tree of Figure 2.2 with total memory size of 7, which is the memory required to run recursive best-first search. In Figure 2.7, the numbers beside the nodes are the order in which they are fully expanded; in this particular example, the second child of an expanded node is generated immediately after its first child if a unit of memory is available. After three full node expansions, nodes A to G , a total of 7 nodes, are generated and saved in memory. The next node to be expanded is node D . In order to accommodate its first child node H , node G (the current worst leaf node in memory) is removed. To provide room for the second child I of D , node E is removed next. The next node selected for expansion is node F . Similarly, to accommodate its children (nodes J and K), nodes H and I must be removed. In this example, space-bounded best-first search expands and generates the same number of nodes as best-first search, but with less memory. Compared to recursive best-first search, space-bounded best-first search uses the same amount of memory and avoids the expansion of some nodes by carefully managing available memory.

2.2 Algorithms for Approximate Solutions

2.2.1 Approximation based on branch-and-bound

It is known in the Operations Research area that approximate solutions can be obtained from branch-and-bound. These approximation algorithms generally use depth-first branch-and-bound since it can repeatedly find better solutions during execution. Three basic approximation schemes have