$$f(n) = g(n) + k * h(n)$$

The factor k is used to control the pull of the heuristic function. Observe that as k tends to zero, the algorithm is controlled by $g(n)$ and it tends to behave like *Branch & Bound*. On the other hand, as we choose larger and larger values of k, the influence of $h(n)$ on the search increases more and more, and the algorithm tends to behave more like *Best First Search*. With values of k greater than one, the guarantee of finding the optimal solution goes, but the algorithm explores a smaller portion of the search space.

The issue of space complexity can be addressed, though at the cost of additional time. We look at some algorithms that require much lower space in the following sections.

5.7 Iterative Deepening *A** (*IDA**)

Algorithm *IDA** (Korf, 1985a) is basically an extension of DFID algorithm seen earlier in Chapter 2. *IDA** is to *A** what *DFID* was to *DFS*. It converts the algorithm to a linear space algorithm, though at the expense of an increased time complexity. It capitalizes on the fact that the space requirements of *Depth First Search* are linear. Further, it is amenable to parallel implementations, which would reduce execution time further. A simple way to do that would be to assign the different successors to different machines, each extending different partial solutions. The *IDA** algorithm is described below in Figure 5.20. The algorithm uses a search bound captured in a variable named *cutoff.* The initial value of *cutoff* is set to the lower bound cost, as seen from the start node *S*. Since this is a lower bound, any solution found within *cutoff* cost must be optimal. The observant reader would have noticed that it is quite unlikely that the solution would be found in the first iteration when *cutoff* = *f(S)*. This is because the heuristic function is designed to underestimate the optimal cost. However, if the *DFS* search fails, in the next iteration the cutoff value is incremented to the next lowest *f*-value from the list *OPEN*. In this way, the value of *cutoff* is increased incrementally to ensure that in any iteration, only an optimal cost solution can be found.

```
IDA*()
  1  cutoff  ← f(S) = h(S)
  2  while goal node is not found or no new nodes exist
  3    do use DFS search to explore nodes with f-values within cutoff
  4      if goal not found
  5         then extend cutoff to next unexpanded value if there exists one
```

FIGURE 5.20 Algorithm Iterative Deepening *A**.

While the algorithm IDA* essentially does Depth First Search in each iteration, the space that it explores is biased towards the goal. This is because the *f*-value used for each node is the sum of the *g*-value and the *h*-value. As for paths that are leading away from the goal, the *h*-values

will increase and such paths would be cut off early. Thus, while DFS itself is without a sense of direction, the fact that *f*-values are used to prune the search pulls the overall envelope that it searches within towards the goal node as depicted in Figure 5.21.

Like *DFS*, the space required of *IDA** grows linearly with depth. We do not need to maintain a *CLOSED* list if we keep track of the solution path explicitly. There is, however, a drawback that in problems like city map route, finding the algorithm may expand internal nodes many times. This happens because there are many routes to a node, and each time the node is expanded all over again. For large problem sizes, this can become a problem. Figure 5.21 below illustrates the search space explored by *IDA** with some value of *cutoff*. One can see that there are combinatorially many paths to any node within the cutoff range. In the absence of a *CLOSED* list, *IDA** will visit all nodes through all possible paths. Nevertheless, for many problems, *IDA** can be a good option, specially if the number of combinations are small.

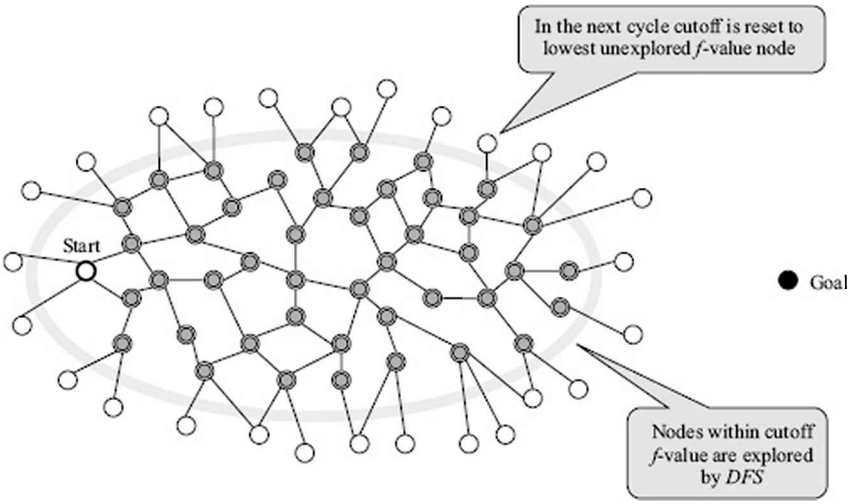

FIGURE 5.21 *IDA** iteratively extends search frontier for Depth First Search.

A factor that may adversely affect running time is when the increment in the cutoff value is such that only a few more nodes are included in each cycle. In the worst case, only one node may be added in each iteration. While this is necessary to guarantee admissibility, one could trade off execution time with some controlled loss in the solution cost. For example, one could decide in advance that the cutoff bounds will be increased by a value δ that is predetermined. The loss then will be bounded by δ, and an appropriate choice may be made for a given application. Figure 5.22 illustrates this situation. In the illustration, two goal nodes come within the ambit of *cutoff* when it is increased by δ. However, since the underlying search is *DFS*, it may terminate with the

more expensive solution.

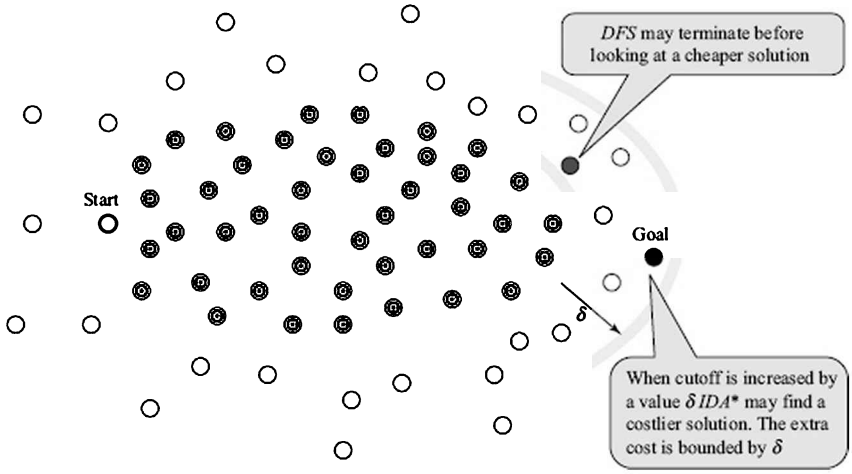

FIGURE 5.22 Loss in optimality is bounded by δ.

5.8 Recursive Best First Search (RBFS)

One thing that *IDA** suffers from is a lack of sense of direction, since the algorithm searches in a depth-first manner. Another algorithm developed by Richard Korf called *Recursive Best First Search* (*RBFS*) also requires linear space, but uses backtracking (Korf, 1993). One can think of the algorithm as heuristic depth-first search in which backtracking occurs when a given node does not have the best *OPEN* node amongst its successors. The interesting feature is that having explored a path once, if it backtracks from a node, it remembers the *f*-values it has found. It uses backed up *f*-values to update the values for nodes it has explored and backtracked on. The backed up value of a node is given by,

$$f_{\text{backed-up}}(\text{node}) = f(\text{node}) \qquad \textit{if node is a leaf node}$$
$$= \min\{f_{\text{backed-up}}(\text{child}) \mid \text{child is a successor of node}\} \qquad \textit{if node is not a leaf node}$$

Figure 5.23 depicts the behaviour of *RBFS*. As shown in the figure on the left, it pursues the middle path from the root node with heuristic value 55, till it reaches a point when all successors are worse than 59, its left sibling. It now rolls back the partial path, and reverts to the left sibling with value 59. It also revises the estimate of the middle node from 55 to 60, the best backed-up values as shown by the upward arrows.

From the point of implementation, *RBFS* keeps the next best value (in Figure 5.23, this is 59) as an upper bound for search to progress further. Backtracking is initiated when all the children of the current node become costlier than the upper bound. Observe that like *DFS*, it maintains only one path in its memory, thus requiring linear space. Its time complexity is