ode is one that is yet to be picked up by the algorithm. In every cycle, it picks one node, to which it as found the shortest path, and colours it black.

```
Dijkstra's Algorithm()
 1  Colour all nodes white
 2  cost(start) ← 0
 3  parent(start) ← NIL
 4  for all other nodes n
 5     do cost(n) ← ∞
 6  repeat
 7     Select lowest cost white node n
 8     Colour n black
 9     for all white neighbours m of n
10        do if (cost(n) + k(n, m)) < cost(m)
11           then  cost(m) ← cost(n) + k(n, m)
12                 parent(m) ← n
13  until all nodes are coloured black
```

FIGURE 5.9 Dijkstra single source shortest path algorithm.

We illustrate the algorithm with our tiny search graph shown in Figure 5.10.

- Note that in the last iteration, a better path to G was found from E. The cost was updated from 11 to 10, and the parent pointer reassigned.
- The last node to be coloured is G, and the algorithm terminates.

The shortest route to *any* node can be traced back.

5.5 Algorithm A*

The algorithm A^*, first described by Hart, Nilsson and Raphael, see (Hart et al., 1968: Nilsson, 1980) combines the best features of *B&B*, Dijkstra's algorithm and *Best First Search* described earlier in Chapter 3.

Both *B&B* and Dijkstra's algorithm extend the least cost partial solution. While the latter is designed to solve a general problem, the former uses a similar blind approach, even though it has a specific goal to achieve. *B&B* generates a search tree that may have duplicate copies of the same nodes with different costs; while Dijkstra's algorithm searches over a given graph, keeping exactly one copy of each node and back pointers for the best routes. Neither has a sense of direction.

Best First Search does have a sense of direction. It uses a heuristic function to decide which of the candidate nodes is likely to be closest to the goal, and expands that. However, it does not keep track of the cost incurred to reach that node, as illustrated in Figure 5.11. *Best First Search* only looks ahead from the node n, seeking a quick path to the goal, while *B&B* only looks behind, keeping track of the best paths found so far. Algorithm A^* does both.

A^* uses an evaluation function $f(node)$ to order its search.

$f(n)$ = Estimated cost of a path from Start to Goal *via* node n.

Let $f^*(n)$ be the (actual but unknown) cost of an optimal path $S \rightarrow n \rightarrow G$ as described above, of which $f(n)$ is an estimate. The evaluation function has two components as shown in Figure 5.12 below. One, backward looking, $g(n)$, inherited from *B&B*, the *known* cost of the path found from S to n. The

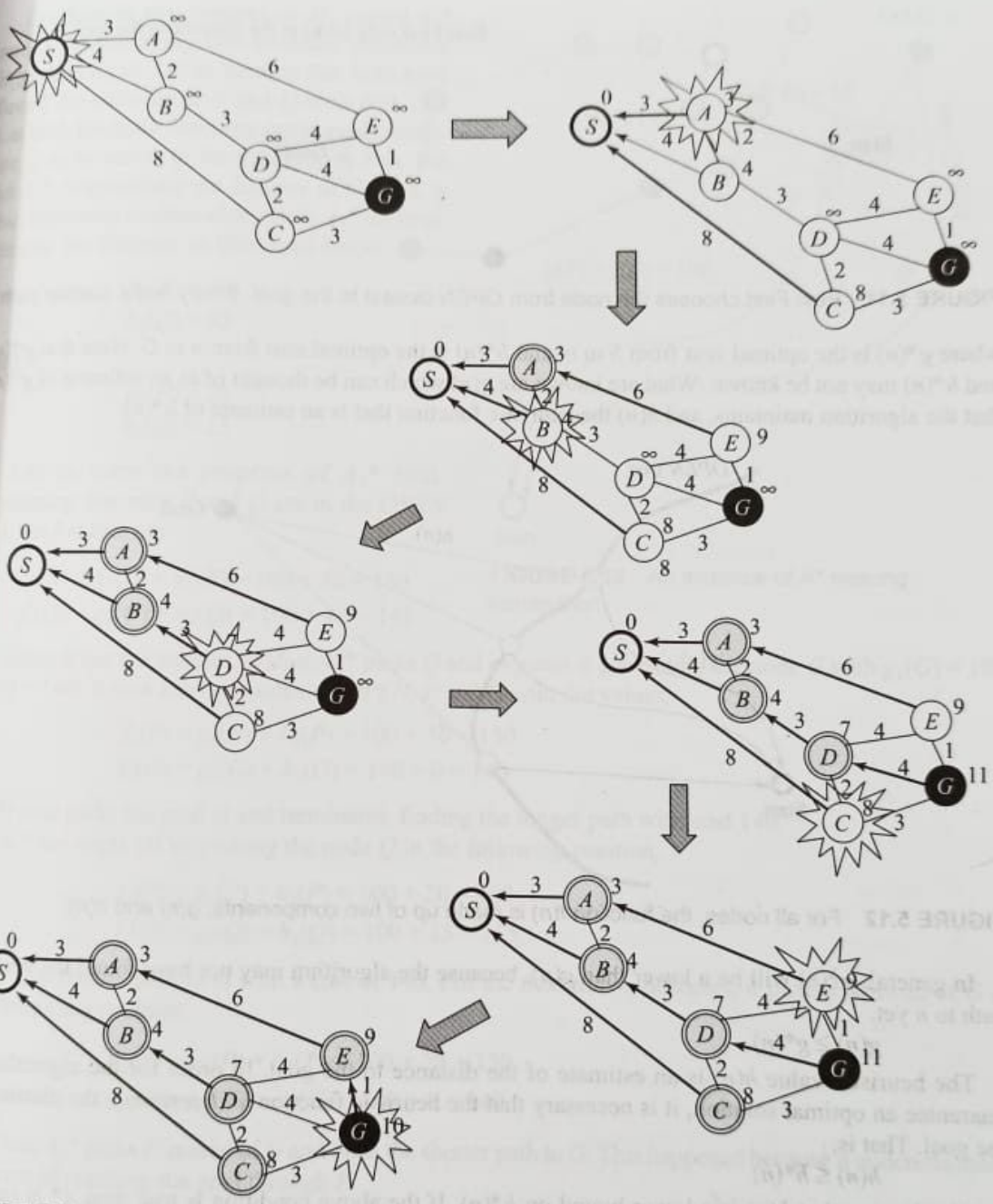

FIGURE 5.10 Dijkstra's algorithm on the tiny search graph.

other, forward looking and goal seeking, $h(n)$, inherited from *Best First Search*, is the *estimated* cost from n to G.

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

FIGURE 5.11 Best First chooses the node from *OPEN* closest to the goal. It may find a costlier path.

where $g^*(n)$ is the optimal cost from S to n, and $h^*(n)$ is the optimal cost from n to G. Note that $g^*(n)$ and $h^*(n)$ may not be known. What are known are $g(n)$ which can be thought of as an estimate of $g^*(n)$ that the algorithm maintains, and $h(n)$ the heuristic function that is an estimate of $h^*(n)$.

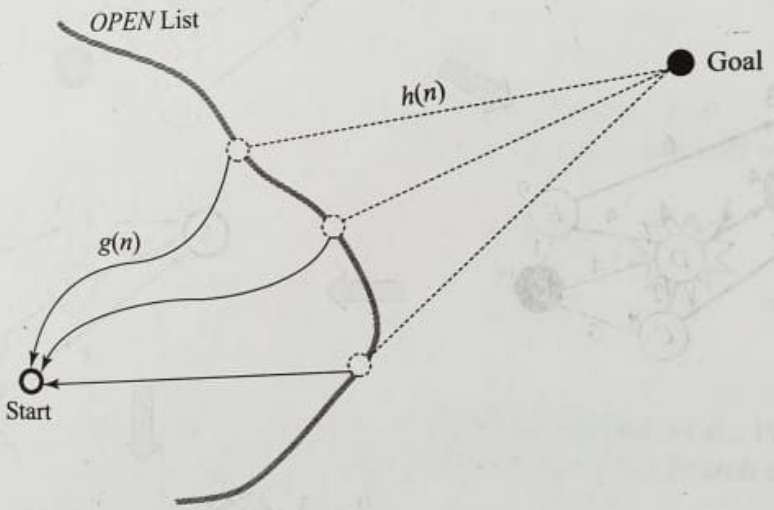

FIGURE 5.12 For all nodes, the function $f(n)$ is made up of two components, $g(n)$ and $h(n)$.

In general, $g^*(n)$ will be a lower than $g(n)$, because the algorithm may not have found the optimal path to n yet.

$$g(n) \geq g^*(n)$$

The heuristic value $h(n)$ is an estimate of the distance to the goal. In order for the algorithm to guarantee an optimal solution, it is necessary that the heuristic function *underestimate* the distance to the goal. That is,

$$h(n) \leq h^*(n)$$

We also say that $h(n)$ is a lower bound on $h^*(n)$. If the above condition is true then A^* is said to be *admissible*; that is, it is guaranteed to find the optimal path. We will look at a formal proof of the admissibility of algorithm A^* later in the chapter. Meanwhile, we illustrate with an example the intuition behind the condition that the heuristic function should underestimate the actual cost. Let an instance of A^* have two nodes, P and Q on the *OPEN* list, such that both are one move away from the goal. Let the cost of reaching both P and Q be the same, say 100. Let the actual cost of the move from P to G be 30, and let the cost of the move from Q to G be 40, as shown in Figure 5.13.

Let there be two versions of A^*, named A_1^* and A_2^*, employing two heuristic functions, $h_1(n)$ and $h_2(n)$. Let us assume that both have found the paths up to P and Q with cost 100. Let both heuristic functions erroneously evaluate Q to be nearer to the goal G than P is. But let A_1^* overestimate the distance to G; thus, in fact becoming inadmissible, while A_2^* underestimates the distance, as illustrated below.

$$h_1(P) = 50$$
$$h_1(Q) = 45$$

and

$$h_2(P) = 20$$
$$h_2(Q) = 15$$

Let us trace the progress of A_1^* first. Assuming that only P and Q are in the *OPEN* list, the f values are

$$f_1(P) = g_1(P) + h_1(P) = 100 + 50 = 150$$
$$f_1(Q) = g_1(Q) + h_1(Q) = 100 + 45 = 145$$

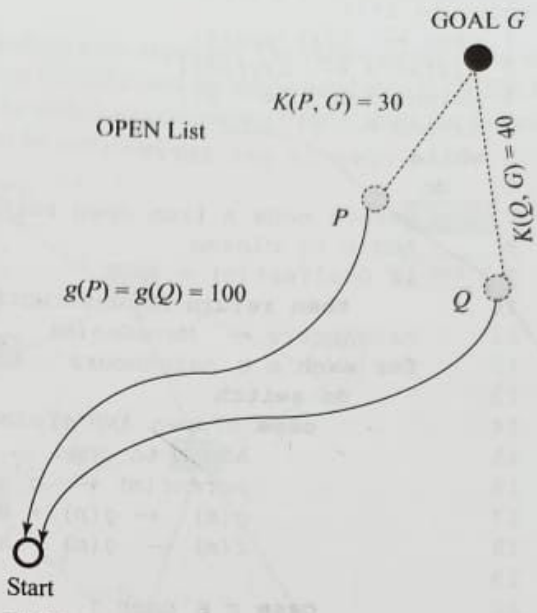

FIGURE 5.13 An instance of A^* nearing termination.

Since it has the smaller *f-value*, A_1^* picks Q and expands it generating the node G with $g_1(G) = 100 + 40 = 140$. It now has two nodes on *OPEN*, P and G with the values,

$$f_1(P) = g_1(P) + h_1(P) = 100 + 50 = 150$$
$$f_1(G) = g_1(G) + h_1(G) = 140 + 0 = 140$$

It now picks the goal G and terminates, finding the longer path with cost 140.
A_2^* too starts off by picking the node Q in the following position.

$$f_2(P) = g_2(P) + h_2(P) = 100 + 20 = 120$$
$$f_2(Q) = g_2(Q) + h_2(Q) = 100 + 15 = 115$$

It also finds a path to G with a cost of 140. For the next move, however, it picks P instead of G in the following position,

$$f_2(P) = g_2(P) + h_2(P) = 100 + 20 = 120$$
$$f_2(G) = g_2(G) + h_2(G) = 140 + 0 = 140$$

Thus, A_2^* picks P instead of G and finds the shorter path to G. This happened because it underestimated the cost of reaching the goal through P.

The algorithm A^* is described below. Like the Dijkstra's Algorithm, it uses a graph structure but one which it generates on a need basis during search. It is also called a *graph search algorithm*. It keeps track of the best route it has found so far to every node on the *OPEN* and *CLOSED*, via the *parent* link. Since it may find cheaper routes to nodes it has already expanded, a provision to pass on any improvements in cost to successors of nodes generated earlier, has to be made.

```
Procedure A*()
 1 open ← List(start)
 2 f(start) ← h(start)
 3 parent(start) ← NIL
 4 closed ← {}
 5 while open is not EMPTY
 6   do
 7     Remove node n from open such that f(n) has the lowest value
 8     Add n to closed
 9     if GoalTest(n) = TRUE
10         then return ReconstructPath(n)
11     neighbours ← MoveGen(n)
12     for each m ∈ neighbours
13         do switch
14           case m∉open AND m∉closed :      /* new node */
15                Add m to open
16                parent(m) ← n
17                g(m) ← g(n) + k(n, m)
18                f(m) ← g(m) + h(m)
19
20           case m ∈ open :
21              if (g(n) + k(n, m)) < g(m)
22                then  parent(m) ← n
23                      g(m) ← g(n) + k(n, m)
24                      f(m) ← g(m) + h(m)
25
26           case m ∈ closed :        /* like above case */
27              if (g(n) + k(n, m)) < g(m)
28                then  parent(m) ← n
29                      g(m) ← g(n) + k(n, m)
30                      f(m) ← g(m) + h(m)
31                      PropagateImprovement(m)
32 return FAILURE

PropagateImprovement(m)
1 neighbours ← MoveGen(m)
2 for each s ∈ neighbours
3   do newGvalue ← g(m) + k(m, s)
4     if newGvalue < g(s)
5       then  parent(s) ← m
6             g(s) ← newGvalue
7             if s ∈ closed
8           then PropagateImprovement(s)
```

FIGURE 5.14 Algorithm *A**.

The representation used here is different from the *nodePair* representation introduced in Chapter 2. Instead, an explicit *parent* pointer is maintained. This has been done because we want to keep only one copy of each node, and reassign parents when the need arises. Consequently, the definition of the *ReconstructPath* function will change. The revised definition is left as an exercise for the user.

In the following example, the node labelled N is about to be expanded. The values shown in the nodes are the g values. The double lined boxes are in the set *CLOSED* and the single lined ones in *OPEN*. Each node, except the start node, has a parent pointer. The dotted arcs emanating from N show the successors of N, including a new node whose g value is yet to be computed.

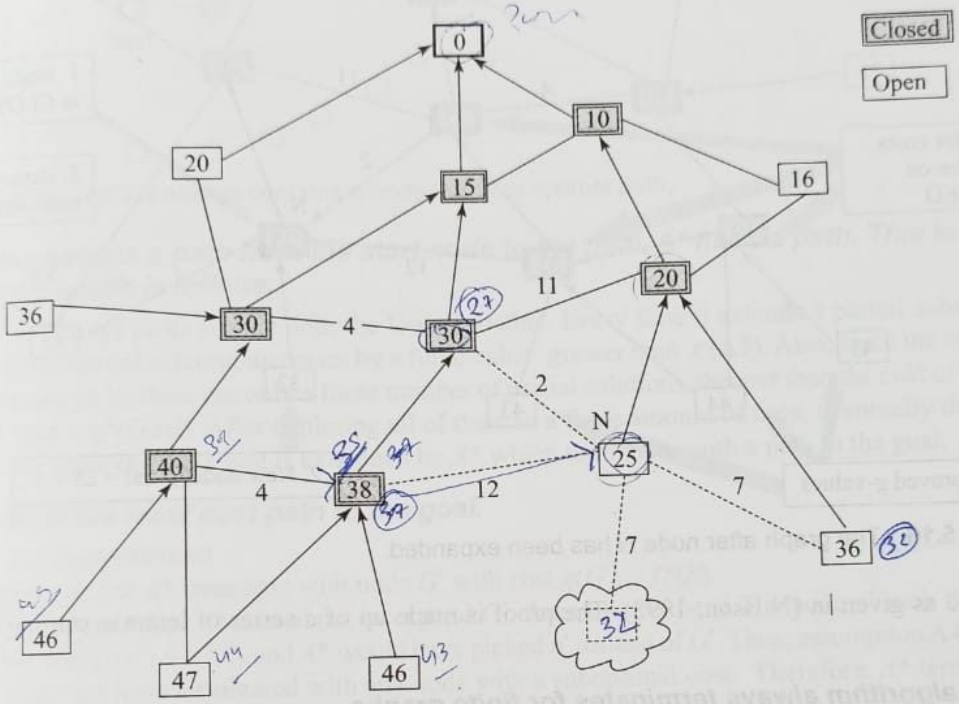

FIGURE 5.15 Node N is about to be expanded.

Figure 5.16 shows the changes that are made after the node N is expanded. Cheaper paths were found for some of the nodes on *OPEN*. Likewise, for some nodes on *CLOSED* too, and in their case the improved g-values had to be passed on to their descendents as well.

5.6 Admissibility of A*

The algorithm A^* will always find an optimal solution, provided the following assumptions (A1 – A3) are true.

A1. The branching factor is finite. That is, there are only a finite number of choices at every node in the search space.

A2. The cost of each move is greater than some arbitrarily small nonzero positive value ε. That is,

$$\text{for all } m, n\text{: } k(m, n) > \varepsilon \tag{5.1}$$

A3. The heuristic function underestimates the cost to the goal node. That is

$$\text{for all } n\text{: } h(n) \leq h^*(n) \tag{5.2}$$

We prove the admissibility of algorithm A^* via a series of lemmas. We also prove that as the heuristic function becomes a better estimate of the optimal cost, the A^* search examines fewer nodes. The

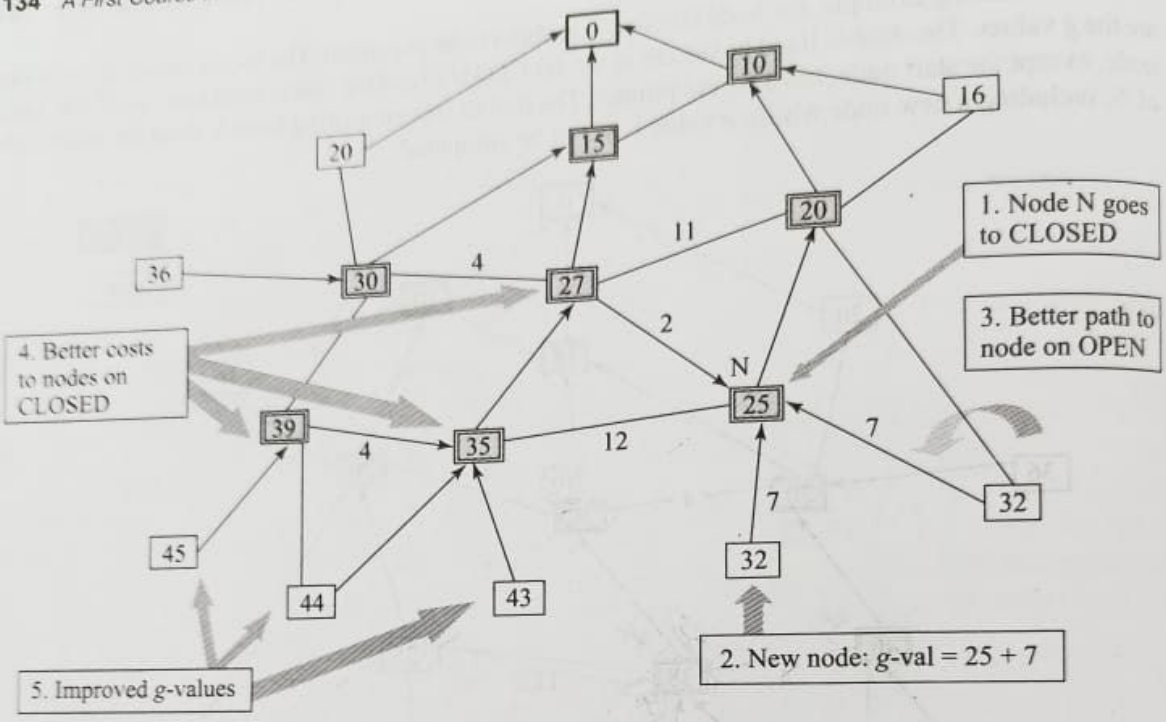

FIGURE 5.16 The graph after node *N* has been expanded.

proofs are as given in (Nilsson, 1998). The proof is made up of a series of lemmas starting with L1 below.

L1: The algorithm always terminates for finite graphs.

Proof In every cycle of the main loop in A^*, the algorithm picks one node from *OPEN* and places it in *CLOSED*. Since there are only a finite number of nodes, the algorithm will terminate in a finite number of cycles, even if it never reaches the goal (that is, the goal is not reachable).

L2: If a path exists to the goal node then the OPEN list always contains a node n′ from an optimal path. Moreover, the f-value of that node is not greater than the optimal cost.

Proof Let $(S, n_1, n_2, \ldots, G)$ be an optimal path as shown in Figure 5.17. To begin with, S is on *OPEN*. Node n_1 is a child of S. When S is removed from *OPEN*, n_1 is placed on *OPEN*. In this manner, whenever a node from the above path is removed from *OPEN*, the next node is placed on *OPEN*. And if G is removed from *OPEN* then A^* has terminated with the optimal path $(S, n_1, n_2, \ldots, G)$ to G.

Furthermore,

$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g^*(n') + h(n') && \text{because } n' \text{ is on the optimal path } g(n') = g^*(n') \\ &\le g^*(n') + h^*(n') && \text{from (A3) } h(n') \le h^*(n') \\ &\le f^*(n') \\ &\le f^*(S) \\ \therefore \quad f(n') &\le f^*(S) && \text{because } n' \text{ is on the optimal path } f^*(n') = f^*(S) \end{aligned} \tag{5.3}$$

Note that $f^*(S)$ is the optimal cost path from S to G.

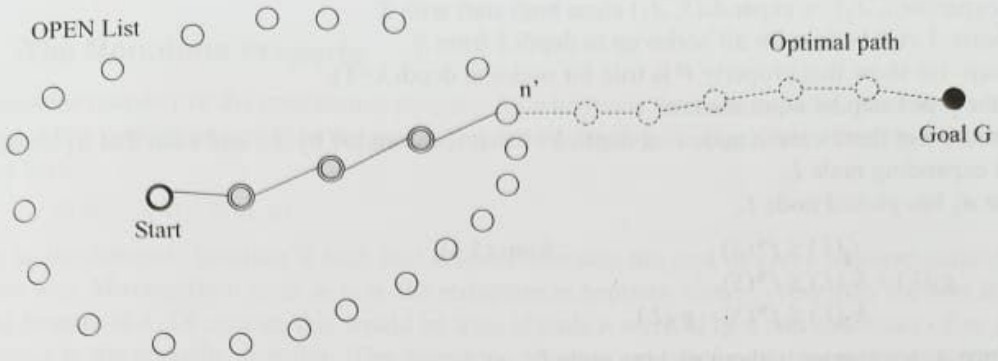

FIGURE 5.17 OPEN always contains a node from the optimal path.

L3: If there exists a path from the start node to the goal, A* finds a path. This is true even if the graph is infinite.

Proof A^* always picks a node with the lowest f-value. Every time it extends a partial solution, the g-value of the partial solution increases by a finite value[2] greater than ε (A2). Also, since the branching factor is finite (A1), there are only a finite number of partial solutions cheaper than the cost of a path to the goal, that is g^*(Goal). After exploring all of them in a finite amount of time, eventually the path to the goal becomes cheapest, and is examined by A^* which terminates with a path to the goal.

L4: A* finds the least cost path to the goal.

Proof (by contradiction)

Assumption A4: Let A^* terminate with node G' with cost $g(G') > f^*(S)$.

At the last step when A^* was about to expand G', there must exist (L2) a node n' such that $f(n') \leq f^*(S)$. Therefore, $f(n') < f(G')$, and A^* would have picked n' instead of G'. Thus, assumption A4 is wrong, and A^* could not have terminated with any node with a suboptimal cost. Therefore, A^* terminates by finding the optimal cost path.

L5: For every node n expanded by A*, f(n) ≤ f*(S)

Proof A^* picked node n in preference to node n'. Therefore,

$$f(n) \leq f(n') \leq f^*(S) \tag{5.4}$$

L6: A more informed heuristic leads to more focussed search.

Let A_1 and A_2 be two admissible versions of A^* using heuristic functions h_1 and h_2 respectively, and let $h_2(n) > h_1(n)$ for all n. We say h_2 is *more informed* than h_1, because it is closer to the h^* value. Since both versions are admissible, both heuristic functions have $h^*(n)$ as the upper bound. Then any node expanded by A_2 is also expanded by A_1. That is, the version with the more informed heuristic function is more focused on the goal, and will never generate more nodes than the less informed one.

Proof (by induction) We show that any node expanded by A_2 is also expanded by A_1. The property P that we need to prove for all nodes n is,

$$\text{expands}(n, A_2) \Rightarrow \text{expands}(n, A_1)$$

which should be read as "if A_2 expands node n then A_1 expands node n".

[2] One might think that it is enough to assume that the cost of each arc is positive. But if arc costs are real, then one could conjure up a problem where there exist paths of infinite steps, but whose total cost is smaller than a given value — first pointed out by Arvind Narayanan, a student, during my AI class at IIT Madras in the mid-nineties. See also Zeno's paradox in (Hofstadter, 1999).

Basis: expands(S, A_2) $\Rightarrow$ expands(S, A_1) since both start with S.
Hypothesis: Let P be true for all nodes up to depth k from S.
Proof step: (to show that property P is true for nodes at depth $k+1$).
We do the proof step by contradiction.
Assumption: Let there exist a node L at depth $k+1$ that is expanded by A_2, and such that A_1 terminates without expanding node L.

Since A_2 has picked node L,

$$f_2(L) \le f^*(S) \qquad \text{from (5.4)}$$

That is $g_2(L) + h_2(L) \le f^*(S)$, (5.5)
or $h_2(L) \le f^*(S) - g_2(L)$

Now, since A_1 terminates without picking node L,

$$f^*(S) \le f_1(L) \qquad \text{because otherwise } A_1 \text{ would have picked } L$$

or $f^*(S) \le g_1(L) + h_1(L)$
or $f^*(S) \le g_2(L) + h_1(L)$

because $g_1(L) \le g_2(L)$ since A_1 has seen all nodes up to depth k seen by A_2, and would have found an equal or better cost path to L.

We can rewrite the last inequality as,

$$f^*(S) - g_2(L) \le h_1(L) \tag{5.6}$$

Combining (5.5) and (5.6), we get,

$$h_2(L) \le h_1(L)$$

which contradicts the given fact that $h_2(n) > h_1(n)$ for all nodes.

The assumption that A_2 terminates without expanding L is false, and therefore A_2 must expand L. Since L was an arbitrary node picked at depth $k+1$, the property P is true for depth $k+1$ as well.

Thus, by induction for all nodes n, the property P is true. That is,

For all nodes n, expands(n, A_2) $\Rightarrow$ expands(n, A_1)

q.e.d.

The search space explored by the two functions is illustrated in Figure 5.18.

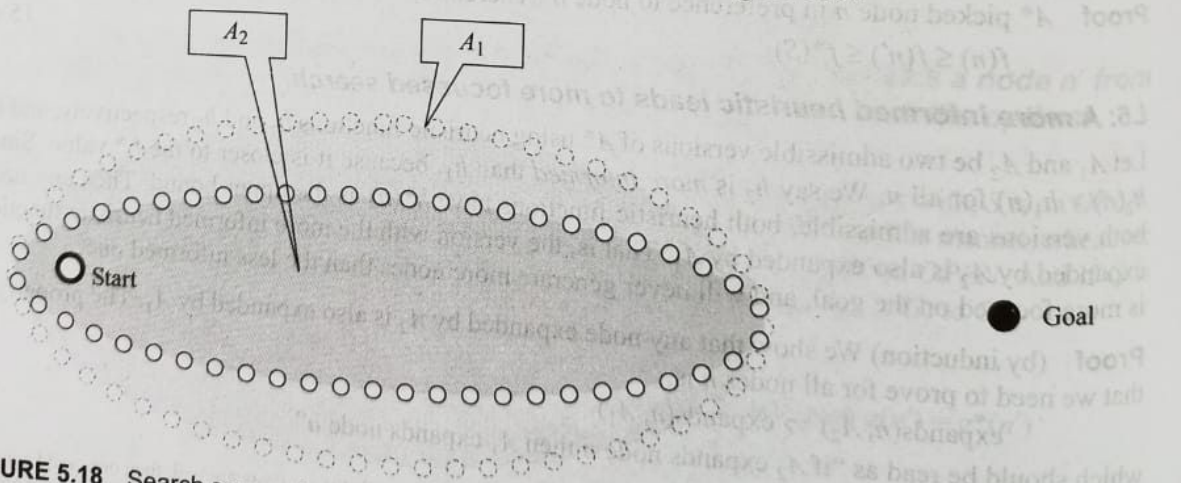

FIGURE 5.18 Search spaces for A_1 and A_2 when $h_2(n) > h_1(n)$.

5.6.1 The Monotone Property

The monotone property or the consistency property for a heuristic function says that for a node n that is a successor to a node m on a path to the goal being constructed by the algorithm A^* using the heuristic function $h(x)$,

$$h(m) - h(n) \leq k(m, n) \tag{5.7}$$

That is, the heuristic function is such that it underestimates the cost of *every segment* individually along the way. Moving from node m to n, the reduction in heuristic value is less than the cost $k(m, n)$ incurred from m to n. Of course, this would be true, if node n were to be a bad successor of m, going for example in the opposite direction. The monotone property says that *even if* m and n were to be on an optimal path, this is true.

Given that we started from node S, we can rewrite the above inequality and add the term $g(m)$ to both sides to get,

$$h(m) + g(m) \leq k(m, n) + h(n) + g(m)$$

Since n is the successor of m, we have

$$g(m) + k(m, n) = g(n)$$

Therefore,

$$h(m) + g(m) \leq h(n) + g(n), \tag{5.8}$$

or
$$f(m) \leq f(n) \tag{5.9}$$

That is, on a path being constructed from to S to G by A^*, the f-values *increase* as we move towards the goal G. This is true even for the optimal path. The closer you get to the goal, the better you estimate the actual cost, since the contribution of the heuristic component decreases. Remember that it is the heuristic function that underestimates the cost. Observe that the hill climbing algorithm using f-values would simply not work, because these values are increasing.

For A^*, the interesting consequence of searching with a heuristic function satisfying the monotone property is that every time it picks a node for expansion, it does so by finding an optimal path to that node. As a result, there is no necessity of improved cost propagation through nodes in *CLOSED* (lines 26–31 in Figure 5.14), because A^* would have already found the best path to them in the first place when it picked them from *OPEN* and put them in *CLOSED*.

L7: If the monotone condition holds for the heuristic function then at the time when A^* picks a node n for expansion $g(n) = g^*(n)$.

Figure 5.19 illustrates the situation. Let A^* be about to pick up node n with a value $g(n)$. Let there be a (known) optimal path from S to n via n_L and n_{L+1}.

Proof Let A^* expand node n with cost $g(n)$.

Let n_L be the last node on the optimal path from S to n that has been expanded. Let n_{L+1} be the successor of n_L that must be on *OPEN*. The following property holds,

$$h(n_L) + g(n_L) \leq h(n_{L+1}) + g(n_{L+1}) \qquad \text{from (5.8)}$$

or
$$h(n_L) + g^*(n_L) \leq h(n_{L+1}) + g^*(n_{L+1})$$

because both are on the optimal path

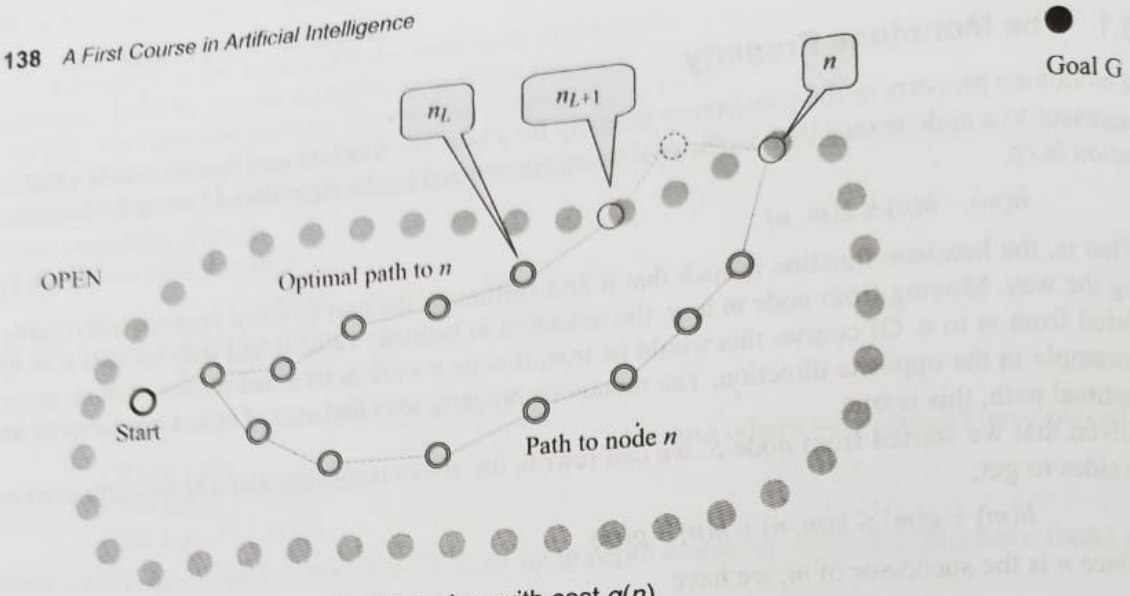

FIGURE 5.19 A^* is just about to pick node n with cost $g(n)$.

By transitivity of ≤, the above property holds true for any two nodes on the optimal path. In particular, it holds for n_{L+1} and node n. That is,

$$h(n_{L+1}) + g^*(n_{L+1}) \leq h(n) + g^*(n) \tag{5.10}$$

That is,

$$f(n_{L+1}) \leq h(n) + g^*(n) \qquad \text{because } n_{L+1} \text{ is on the optimal path to } n$$

But since A^* is about to pick node n instead,

$$f(n) \leq f(n_{L+1})$$

That is,

$$h(n) + g(n) \leq f(n_{L+1})$$

or

$$h(n) + g(n) \leq h(n_{L+1}) + g^*(n_{L+1}) \tag{5.11}$$

Combining (5.10) and (5.11) we get,

$$h(n) + g(n) \leq h(n) + g^*(n)$$

$$\therefore \quad g(n) \leq g^*(n)$$

$$\therefore \quad g(n) = g^*(n) \text{ because } g(n) \text{ cannot be less than } g^*(n) \text{ the optimal cost.}$$

Therefore, whenever A^* expands any node, it does so after finding an optimal path to it.

That means that nodes that are in *CLOSED* cannot have better paths to them, and consequently need not be updated. q.e.d.

5.6.2 Performance of Algorithm A*

The algorithm A^* is complete and admissible. The results have been proven in the previous section. Both space complexity and time complexity of A^* are directly dependent on the heuristic function. With a perfect heuristic function, the algorithm will home in onto the goal in linear time with linear space requirement. In practice, heuristic functions are less than perfect, and the space and time requirements

can be exponential, though with a lower branching factor than the actual one, as was discussed in the case of *Best First Search* earlier. Figure 5.18 above also illustrates the point that more accurate heuristic functions require lesser space, and, therefore, lesser time (since the size of *CLOSED* can be seen to be a measure of time). Incidentally, the figure can be visually misleading since we tend to see the search space as area inside the curve that grows only as quadratic. The reader must keep in mind that in many search spaces, the sizes of successive layers are multiples of preceding layers, leading to exponential growth. However, figures like Figure 5.18 above are useful for visualizing algorithms and we will continue to use them. We will also see problems where the growth in search space is quadratic, which are better illustrated by these figures, and in which there is a combinatorial growth of choices.

Time complexity by itself can only improve with a better heuristic function, or at the expense of admissibility. For example, many researchers experiment with a variation known as *Weighted A** which uses the following function to order the search nodes,

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

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.

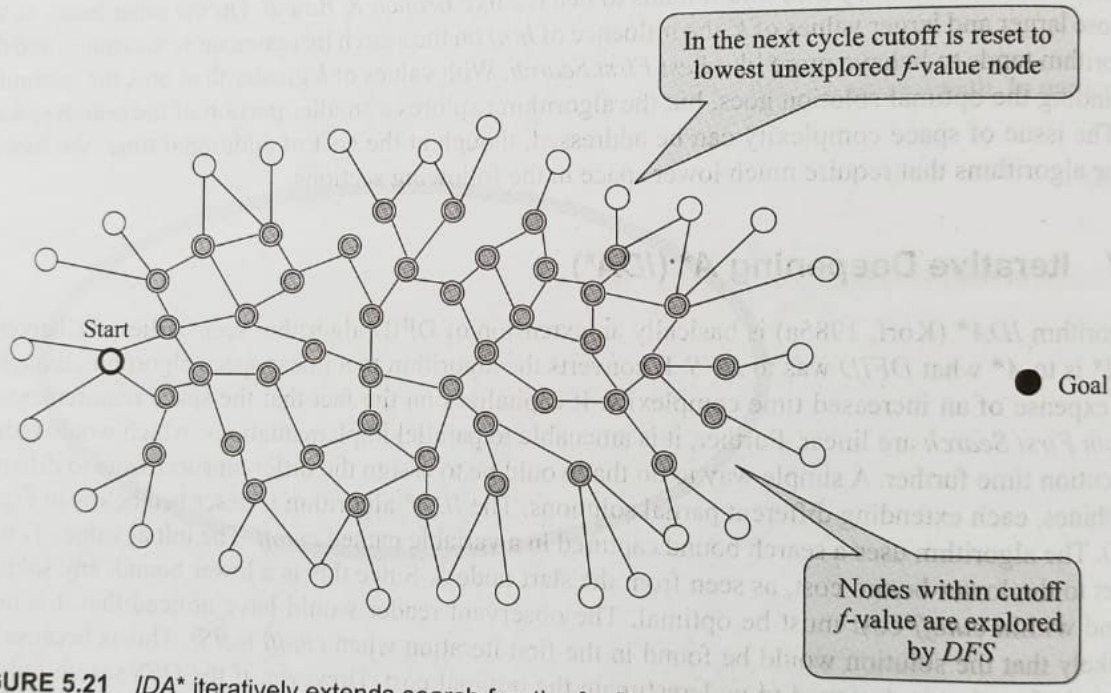

FIGURE 5.21 *IDA** iteratively extends search frontier for Depth First Search.

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.

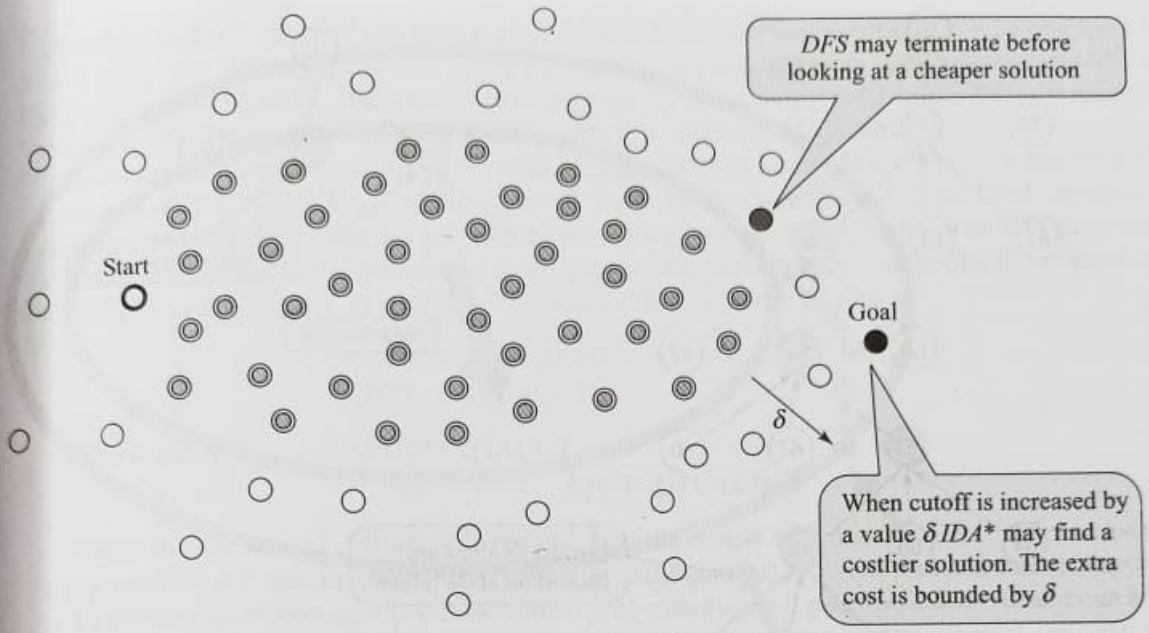

FIGURE 5.22 Loss in optimality is bounded by δ.

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 however difficult to characterize. One can imagine that in a large search space, it will often switch attention, sometimes even between the same two paths. This has been corroborated by experimental results, where it was found to take considerably longer solving problems, specially those where even though the problem only grows polynomially, the number of different paths grow combinatorially. Like *IDA**, *RBFS* ends up repeatedly visiting the

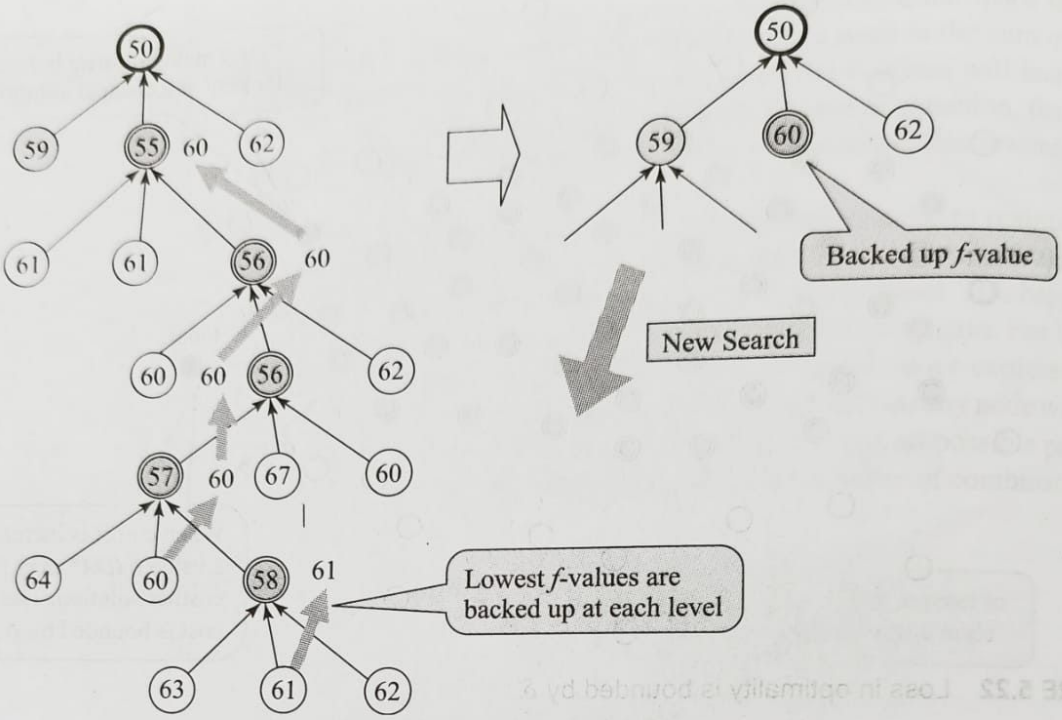

FIGURE 5.23 Recursive Best First Search rolls back a path when it is not looking the best.

same nodes again and again, leading to an increase in the computation time. This is the price both the algorithms have to pay for saving on space.

We now explore some other approaches to reducing memory, starting with the list *CLOSED*.

Maintaining the *CLOSED* list has two benefits. One, that it keeps a check on the nodes already visited, and prevents the search from expanding them again and again. And two, it is the means for reconstructing the path after the solution is found. In the next section, we look at an algorithm that prunes the *CLOSED* list, and still does both but, again, at the expense of more time complexity.