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

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

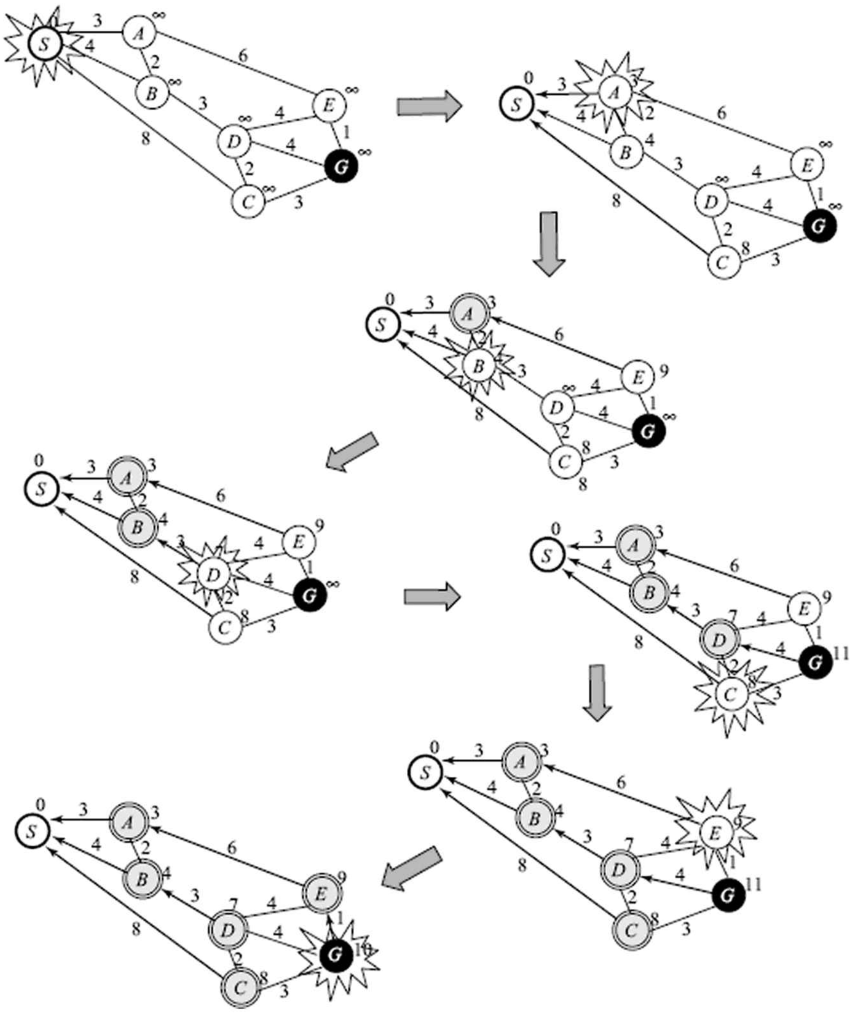

FIGURE 5.10 Dijkstra's algorithm on the tiny search graph.

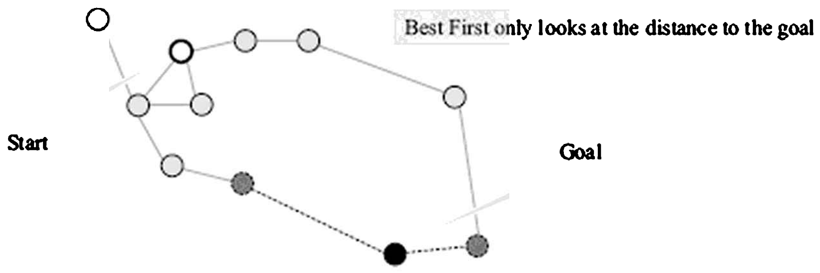

Figure 5.11 Best First chooses the node from *OPEN* closest to the goal. It may find a costlier path.

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 $h^*(n)$.

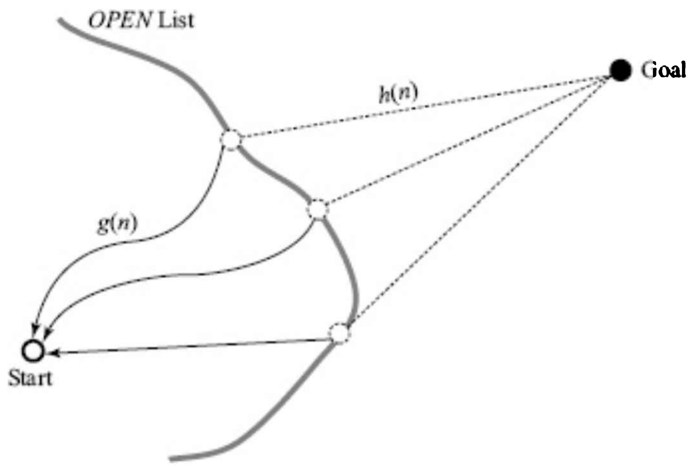

FIGURE 5.12 For all nodes, the function $f(n)$ is made up of two components, $g(n)$ and $h(n)$.

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

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

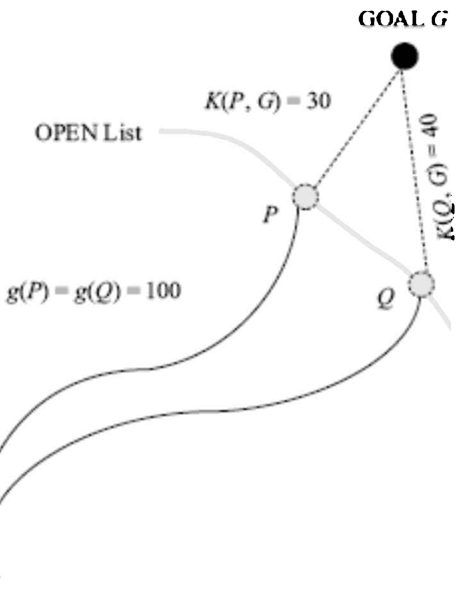

FIGURE 5.13 An instance of *A** nearing termination.

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 mÏopen AND mÏclosed :      /* 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.

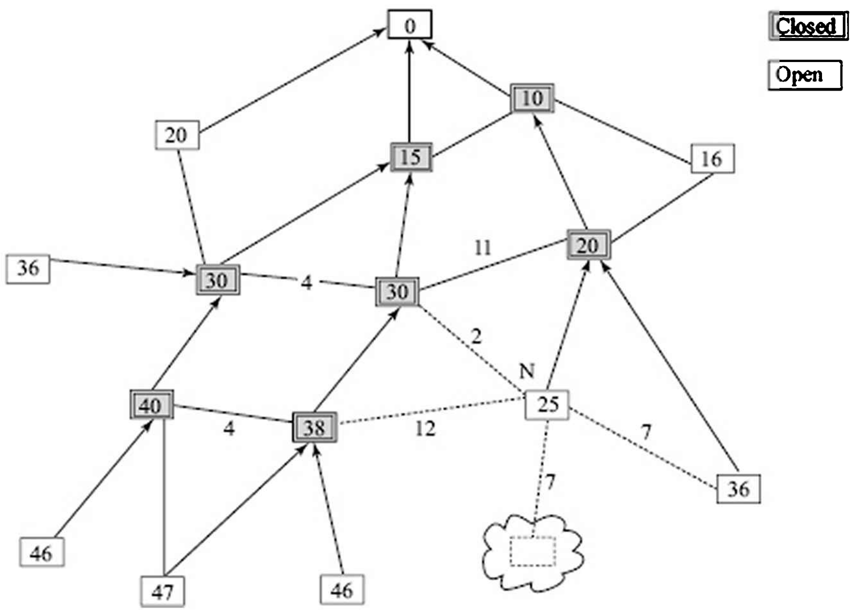

FIGURE 5.15 Node *N* is about to be expanded.

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,

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

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

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