Heuristic Search

Chapter 3

In the search algorithms described in Chapter 2, the only role that the goal node plays is in testing whether the candidate presented by the search algorithm is a goal or not. Otherwise, the search algorithms are quite oblivious of the goal node. Any intelligent observer watching the progress of the algorithms would soon get exasperated! They *always* go about exploring the state space in the same order, irrespective of the goal to be achieved. They are, therefore, called *blind* or *uninformed. The Depth First Search* (see Figure 3.1) dives into the search space, backtracking only if it reaches a dead end. If the search space were infinite, it might just keep going along an endless path. *The Breadth First Search,* on the other hand, ventures out cautiously, going further away, only if it has finished inspecting the nodes the same distance away from the start position. Consequently, it always finds the shortest solution, though its space requirements grow exponentially. *DFID* is basically a sequence of 'depth first searches' masquerading as a breadth first search.

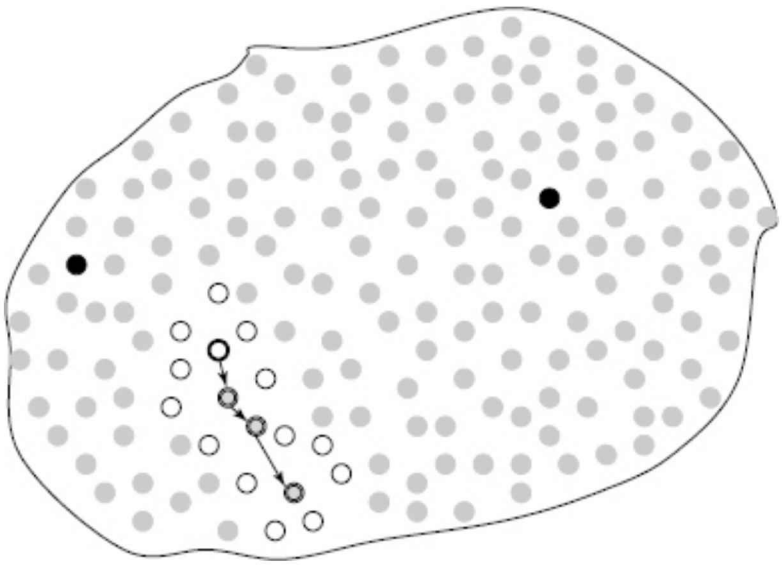

FIGURE 3.1 *DFS* searches blindly, wherever the goal may be.

The search algorithms described so far maintain a list, called *OPEN,* of candidate nodes. Depending upon whether the algorithm operates *OPEN* as a stack or as a queue, the behaviour is either depth first or breadth first. What we would like is the algorithm to have, instead, some sense of direction. If it could make a guess as to which of the candidates is *more likely* to lead to the goal, it would have a chance of finding the goal node faster. We introduce the notion of a *heuristic function* to enable the search algorithm to make an informed guess.

3.1 Heuristic Functions

As seen in Chapter 2, the time required for search could be exponential in the distance to the goal. The idea of using a heuristic function is to guide the search, so that it *has a tendency* to explore the region leading to the goal. A heuristic function is designed to help the algorithm to pick that candidate from the *OPEN* list that is most likely to be on the path to the goal. A heuristic value is computed for each of the candidates. The heuristic value could be *an estimate* of the distance to the goal from that node, as shown in Figure 3.2. The heuristic function could also embody some knowledge gleaned from human experts that would indicate which of the candidate nodes are more promising. The algorithm then simply has to choose the node with the lowest heuristic value to expand next.

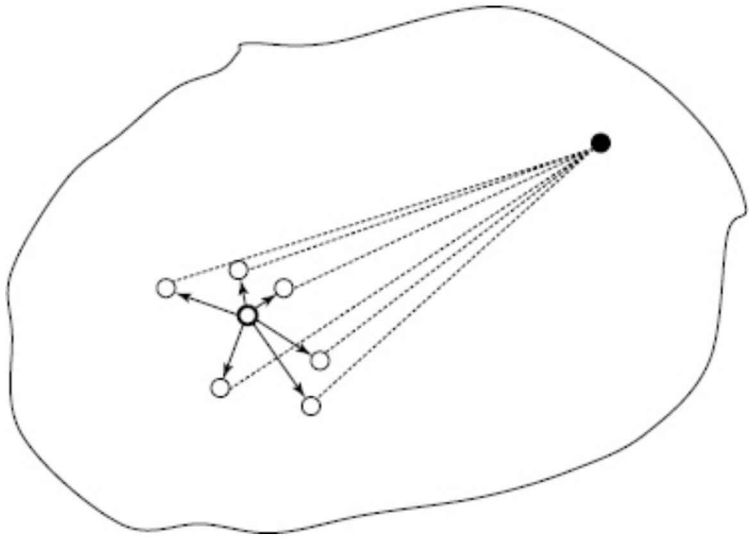

FIGURE 3.2 The heuristic function estimates the distance to the goal.

The word *heuristic* has its roots in the Greek word εὑρίσκω or *euriskō*, which means *"I find, discover"*. It has the same root as the word *eureka* from εὕρηκα or *heúrēka* meaning "I *have found (it)"*—an expression, the reader might remember, that is attributed to Archimedes

when he realized that the volume of water displaced in the bath was equal to the volume of his body. He is said to have been so eager to share his discovery, that he leapt out of his bathtub and ran through the streets of Syracuse naked[1].

The heuristic function must not be computationally expensive, because the idea is to cut down on the computational cost of the search algorithm. The simplest way to do this is to make the heuristic function a *static evaluation function* that looks only at the given state and returns a value. It will also have to look at the goal state, or a goal description. Since we expect the move generation function to transform the given state into the goal state via a sequence of moves, the heuristic function has to basically estimate how much of that required transformation still needs to be done for a given state. In other words, it is some kind of a *match* function that computes some *similarity* measure of the current state with the goal state. Such a heuristic function will be domain dependent, and will have to be included into the domain functions, along with the *moveGen* and *goalTest* functions described in Chapter 1. In later chapters, we will also look at the notion of domain independent heuristic functions. These functions estimate the distance to the goal by solving a *relaxed* version of the original problem. The relaxed problems are such that they are simpler to solve, typically being linear or polynomial in complexity. They typically give us a lower bound on the distance to the goal.

Traditionally, the heuristic function is depicted by $h(n)$, where n is the node in question. The fact that the heuristic value is evaluated with respect to a goal state is implicit, and, therefore, the heuristic value should be seen to be for a given problem in which the goal has been specified. To incorporate heuristic values in search, the node-pair representation used will have to be augmented with the heuristic value, so that a node in the search tree will now look like,

searchNode = (currentState, parentState, heuristicValue)

We illustrate the idea of heuristic functions with a few example problems.

In a route finding application in a city, the heuristic function could be some measure of distance between the given state node and the goal state. Let us assume that the location of each node is known in terms of its coordinates. Then a heuristic estimate of distance could be the Euclidean distance of the node from the goal node. That is, it estimates how close to the goal the current state is

Euclidean distance:

$$h(n) = \sqrt{(x_{Goal} - x_n)^2 + (y_{Goal} - y_n)^2}$$

Note that this function gives an optimistic estimate of distance. The

actual distance is likely to be more than the straight line distance. Thus, the Euclidean distance is lower bound on the actual distance. Another distance measure we could use is the *Manhattan distance* or the *city block distance,* which is given below:

Manhattan distance:

$$h(n) = |x_{\text{Goal}} - x_n| - |y_{\text{Goal}} - y_n|$$

This estimates the distance assuming that the edges form a grid, as the streets do in most of Manhattan. Observe that at this point, we are not really interested in knowing the distance accurately; though later we will encounter algorithms that will benefit from such accuracy. At this moment it suffices if the heuristic function can reliably say as to which of the candidates is likely to be *closer* to the goal.

Next, consider the Eight puzzle. The following diagram shows three choices faced by a search algorithm. The choices in the given state are *R* (move a tile right), *U* (up) and *L* (left). Let us call the corresponding states too R, *U* and L. One simple heuristic function could be simply to count the number of tiles out of place. Let this function be called h_1. The values for the three choices are:

$h_1(R) = 6$	(Only 4, 5 and 7 are in place. The rest are in a wrong place.)
$h_1(U) = 5$	(Again only 4, 5 and 7 are in their final place, but also the blank tile.)
$h_1(L) = 5$	(2,4,5 and 7 are in place.)

Thus, according to h_1, the best move is either *U* or *L*. Let us look at another heuristic function h_2 that adds up the Manhattan distance of each tile from its destination. The values, counting from the blank tile, and then for tile-1 to tile-8, are:

$h_2(R) = (2 + 1 + 1 + 3 + 0 + 0 + 2 + 0 + 1) = 10$
$h_2(U) = (0 + 1 + 1 + 3 + 0 + 0 + 3 + 0 + 2) = 10$
$h_2(L) = (2 + 1 + 0 + 3 + 0 + 0 + 3 + 0 + 1) = 10$

If one were to think of the heuristic values as obtained from solving a relaxed version of the Eight-puzzle then the first one can be thought of as a problem, where a tile can be moved to its destination in one move, and the second heuristic from a problem where a tile can move even over existing tiles. The values are the total number of moves that need to be made in the relaxed problem(s). Curiously, this more detailed function seems to think that all moves are equally good. We leave it as an exercise for the user to pick the best move in this situation.

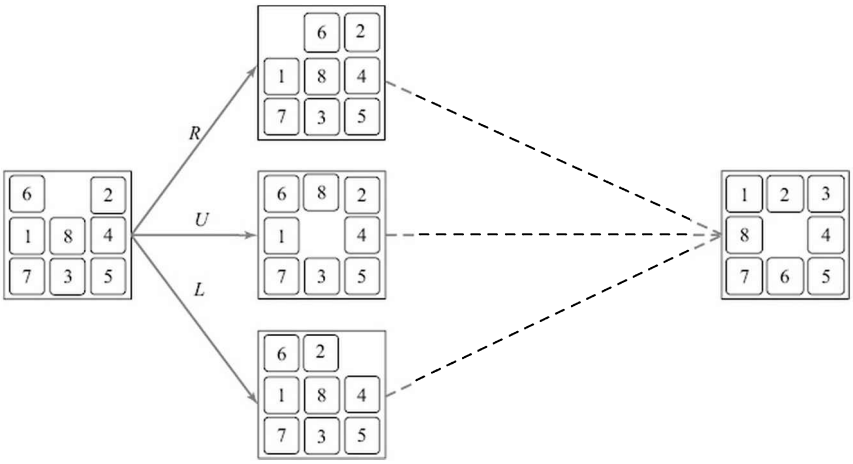

FIGURE 3.3 Which state is closest to the goal?

The Eight-puzzle has its bigger cousins, the 15-puzzle and the 24-puzzle, played on a 4 × 4 and a 5 × 5 grid respectively. The state spaces for these puzzles are much larger. The 15-puzzle has about 10000000000000 (or more succinctly 10^{13}) states, while the 24-puzzle has about 10^{25} states. These are not numbers to trifle with. With an algorithm inspecting a billion states a second, we would still need more than a thousand centuries to exhaustively search the state space. Only recently have machines been able to solve the 24-puzzle, and that requires a better heuristic function than the ones we have.

We now look at our first algorithm to exploit the heuristic function, called *Best First Search,* because it picks the node which is best according to the heuristic function.

3.2 Best First Search

We need to make only a small change in our algorithm *Depth First Search* from Chapter 2. We simply maintain an *OPEN* list sorted on the heuristic value, so that the node with the best heuristic value automatically comes to the head of the list. The algorithm then picks the node from the head of *OPEN* as before. Conceptually, this can be written by replacing the line,

OPEN ← append (NEW, tail(OPEN))

with

OPEN ← sort_h (append (NEW, tail(OPEN)))

In practice, though, it would be more efficient to implement *OPEN* as a priority queue. In addition, one has to make the changes to the search node representation to include the heuristic value, and the consequent

changes in the other functions. These changes are left as an exercise for the reader.

The following figure illustrates the progress of the algorithm on an example problem. The graph in Figure 3.4 represents a city map, where the junctions are located on a two dimensional grid, each being 1 km. A path from the start node *S* to the goal node *G* needs to be found. The search uses the Manhattan distance as the estimate of the distance. The graph depicts the nodes on *CLOSED* with double circles, and the nodes on *OPEN* with single circles, at the instance when the algorithm terminates. Both the sets of nodes are labelled with the heuristic values. The labels near the nodes show the order in which the nodes are expanded. Note that the heuristic function initially takes the search down a path that is not part of the solution. After exploring nodes labelled 2, 3 and 4 in the search order, the search abandons them and goes down a different path. This is characteristic of a heuristic search, and happens because the heuristic function is not "aware" of the entire roadmap. It only has a sense of direction. Perhaps, in this example, there is a river on the way without a bridge at that point. The search then has to explore an alternate route. The back pointers on the edges point to the parent nodes, and the thick edges with back arrows show the path found by the search.

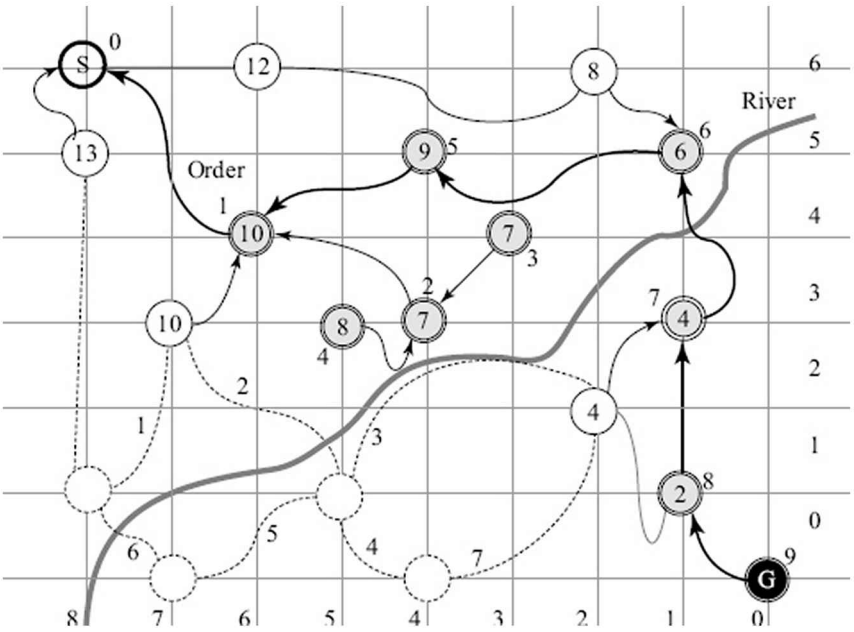

FIGURE 3.4 Best First Searches has a sense of direction, but may hit a dead end too.

What about the performance of the *Best First Search* algorithm? How much does it improve upon the uninformed search methods seen in Chapter 1?

3.2.1 Completeness

Best First Search is obviously complete, at least for finite domains. The reasoning is straightforward. The only change we are making is in the ordering of *OPEN.* It still adds all unseen successors of a node to the list, and like the earlier search algorithms, it will report failure only after *OPEN* becomes empty. That is, only when it has inspected all the candidate nodes. Note that *Best First Search* is systematic too, in the sense that it will inspect all nodes before giving up.

For infinite state spaces, the 'completeness' property will depend upon the quality of the heuristic function. If the function is good enough, the search will still home in on the goal state. If the heuristic function yields no discriminating information (for example if $h(n) = 0$ for all nodes n), the algorithm will behave like its parent algorithm. That is, either *DFS* or *BFS*, depending upon whether it treats *OPEN* like a stack or a queue.

Let us discuss the quality of the solution before looking at complexity.

3.2.2 Quality of Solution

So far we have only talked about the quality of the solution in terms of its length, or the number of moves required to reach the goal. With a simple example shown in Figure 3.5, we can see that it is possible that *Best First Search* can choose a longer solution, in terms of the number of hops. This may happen if the heuristic function measures the difference in terms of some metric which does not relate to the number of steps. Even when we look at other measures for quality, it will be possible to construct examples for which the algorithm picks a sub-optimal solution. This happens mainly because the algorithm only compares two states by estimating the distance to the goal, without taking into account the cost of reaching the two states. Thus, if two states have the same heuristic value, but one of them was more expensive to achieve, the *Best First Search* algorithm has no way of discriminating between the two. As we shall see later (in Chapter 5), incorporating this cost and some conditions on the heuristic will ensure the finding of optimal solutions.

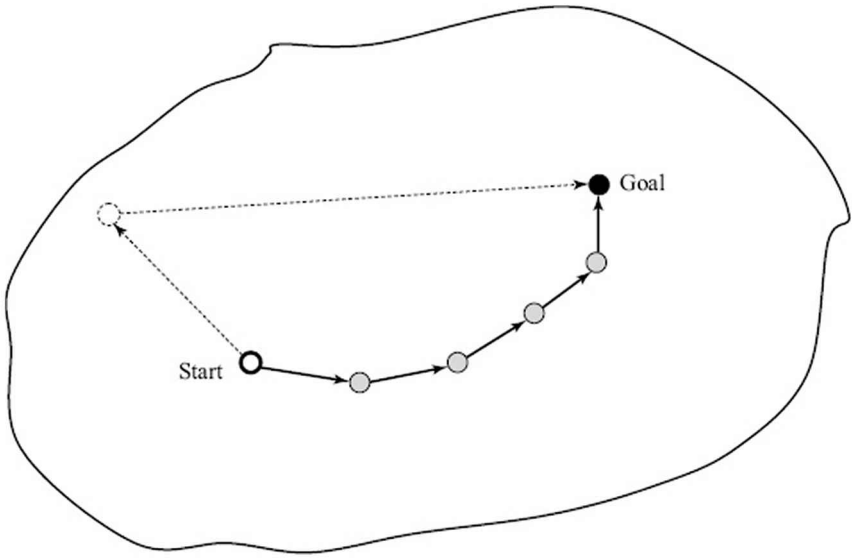

FIGURE 3.5 Best First Search chooses a solution with five moves.

3.2.3 Space Complexity

We have already seen that the search frontier, represented by the list *OPEN*, grows linearly for *DFS* and exponentially for *Breadth First Search.* The search frontier for *Best First Search* depends upon the accuracy of the heuristic function. If the heuristic function is accurate then the search will home in onto the goal directly, and the frontier will only grow linearly. Otherwise, the search algorithm may go down some path, change its mind, and sprout another branch in the search tree (Winston, 1977). Figure 3.6 depicts the frontiers for the three algorithms being discussed. Empirically, it has been found though that for most interesting problems, it is difficult to devise accurate heuristic functions, and consequently the search frontier also grows exponentially in best first searches.

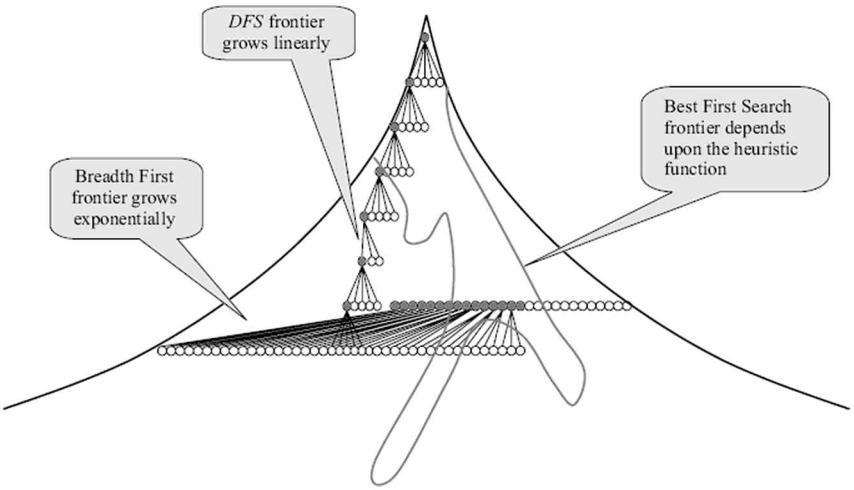

FIGURE 3.6 The Search Frontier is an indication of space requirement.

3.2.4 Time Complexity

Like space complexity, time complexity too is dependent upon the accuracy of the heuristic function. With a perfect heuristic function, the search will find the goal in linear time. Again, since it is very difficult to find very good functions for interesting enough problems, the time required too tends to be exponential in practice.

In both space and time complexity, we have pegged the performance on the accuracy of the heuristic function. In fact, by measuring performance experimentally, we could talk about the accuracy of heuristic functions. A perfect heuristic function would only inspect nodes that lead to the goal. If the function is not perfect then the search would often pick nodes that are not part of the path. We define *effective branching factor* as a measure of how many extra nodes a search algorithm inspects.

> *effective branching factor = total-nodes-expanded / nodes-in-the-solution*

For a perfect search function, the effective branching factor would tend to be 1. For a very poor heuristic function, one expects the effective branching factor to tend to be much greater that the branching factor of the underlying search space. This is because a really poor function would lead the search away from the goal, and explore the entire space. As an exercise, the reader could implement the two heuristic functions discussed earlier for the Eight-puzzle, and compare the performance over a set of problems. The function that performs better on the average is likely to be better.

A related measure for heuristic functions that has been suggested in literature is called *penetrance* (Nilsson 1998). Penetrance can be defined

as the inverse of effective branching factor. That is:

penetrance = *nodes-in-the-solution* / *total-nodes-expanded*

The best value of penetrance is 1, and if the search does not find a solution, which it might in an infinite search space *with a poor heuristic,* the penetrance value could be said to be zero. Note that both the above measures are for specific search instances. To get a measure of the heuristic function that does not depend upon the given problem, one would have to average the values over many problems.

Best First Search is an *informed* search algorithm. Instead of blindly searching in a predefined manner, it inspects all candidates to determine which of them is most likely to lead to the goal. It does this by employing a heuristic function that looks at the state in the context of the goal, and evaluates it to a number that in some way represents the closeness to the goal. The heuristic functions we have seen are domain dependent. They require the user to define them. Typically, the heuristic function measures closeness by measuring similarity from some perspective. For the route map, finding the similarity is in terms of location; while for the Eight-puzzle, it is in terms of similarity of the patterns formed by the tiles.

In Chapter 10 on advanced methods in planning, we will also see how domain independent heuristics functions can be devised. They will essentially solve simpler versions of the problem in such a way that the time required is significantly smaller. By solving the simpler version for all the candidates, they will be able to give an estimate as to which of the candidates is the closest to the goal. Of course, care has to be taken that the work done by the heuristic function is offset by a reduction in the number of nodes explored, so that the overall performance is better than an uninformed search.

Meanwhile, there exist many interesting real problems that need to be solved, and for which heuristic functions can be devised. For many of these problems, the state representations may be quite complex. There is a need to find algorithms that are easier on space requirements. In the next few sections, we begin by looking at heuristic search algorithms that are guaranteed to have low space requirements. This will enable us to work on complex problems with large search spaces. The cost we may have to pay is in terms of completeness. While *Best First Search* is complete for finite spaces, it becomes impractical if the search spaces are too large. There is still a market for heuristic algorithms to search these spaces, even though they may not *always* find the solution.

3.3 Hill Climbing

If our heuristic function is good then perhaps we can rely on it to drive the search to the goal. We modify the way *OPEN* is updated in *Best First Search.* Instead of,

OPEN ← append (sort$_h$ (NEW, tail(OPEN)))

we can try using

OPEN ← sort$_h$ (NEW),

where, NEW is the list of new successors of the nodes just expanded. Observe that in doing so we have jettisoned the nodes that were added to *OPEN* earlier. We have burned our bridges as we move forward. A direct consequence of this will be that the new algorithm may not be complete. But having done this, we might as well not maintain an *OPEN* list, since only the best node from the successors will be visited. The revised algorithm is given in Figure 3.7.

```
HillClimbing()
   1  node ← start
   2  newNode ← Head(Sort_h(MoveGen(node)))
   3  while h(newNode) < h(node)
   4     do node ← newNode
   5        newNode ← Head(Sort_h(MoveGen(node)))
   6  return newNode
```

FIGURE 3.7 Algorithm Hill Climbing.

Observe that this algorithm has modified our problem solving strategy *considerably*.

In the search algorithms we saw till now, the termination criterion was finding the goal node. In *Hill Climbing,* the termination criterion is when no successor of the node has a better heuristic value. Searching for the goal has been replaced with optimizing the heuristic value. The problem has been transformed into an optimization problem, in which we seek the node with the lowest heuristic value, rather than one that satisfies the *goalTest* function. This is consistent with the notion that the goal state has the lowest heuristic value, since the distance to the goal is zero. We will also intermittently use the term *objective function* used by the optimization community to refer to the *heuristic function*.

Let us consider the negation of the heuristic function –*h*(*n*). Instead of looking for lower values of *h(n),* we can equivalently say that we are looking for higher values of –*h*(*n*). That is, instead of a minimization problem, we have a maximization problem.

As long as the *moveGen* function keeps generating nodes with better (higher) values, the search keeps stepping forward. Of the choices available, the algorithm picks the best one. In other words, the algorithm is performing *steepest gradient ascent.* Imagine you were blindfolded and left on a hillside, and asked to go to the top. What would your strategy be? Presumably, you might test the ground on all sides and take a step in the direction of the steepest gradient. And you would stop when the

terrain in no direction is going upwards. That is precisely what the algorithm *Hill Climbing* is doing; moving along the steepest gradient of a terrain defined by the heuristic function. The idea is illustrated in Figure 3.8 below.

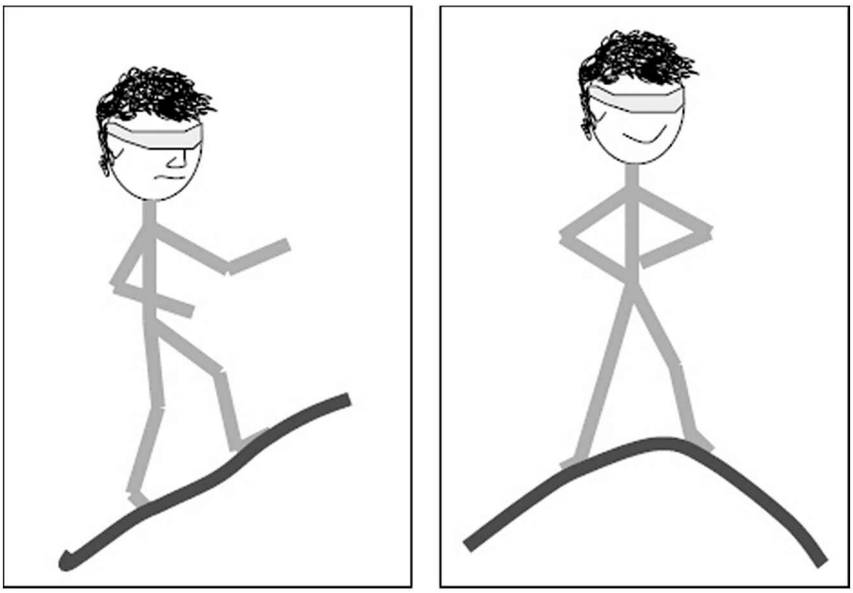

FIGURE 3.8 Hill Climbing.

Observe that while the "terrain" is defined by the heuristic function, and different heuristic functions define different "landscapes", it is the *moveGen* function that independently determines the neighbourhood available to the search function. The *moveGen* function determines the set of points, both in number and location, in the landscape that become the neighbours of a given node. The heuristic function assigns a heuristic value to each point supplied by the *moveGen* function.

The problem with climbing hills blindfolded is that one does not have a global view. We can only do local moves, and the best we can do is to choose the locally good options. If the hill we are asked to climb had a "smooth" surface then the algorithm will work well. However, mountainous terrain is seldom smooth, and one is more likely than not to end up in the situation depicted in Figure 3.9. Instead of reaching the highest peak, we might end up in a lower peak somewhere on the way. In other words, one has reached a maximum, but only one that is local. If the heuristic function used by *Hill Climbing* is not perfect then the algorithm might well find a local maximum as opposed to a global maximum.

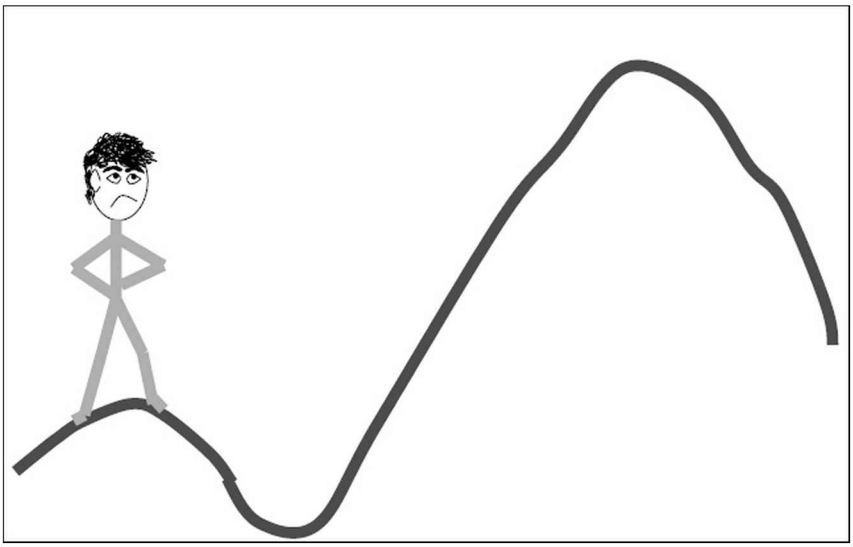

FIGURE 3.9 Stopping at a Local Maximum.

3.4 Local Maxima

We look at an example problem where the choice of a heuristic function determines whether there are local maxima in the state space or not. The blocks world domain consists of a set of blocks on an infinitely large table. There can be only one block on top of each block. The problem is to find a sequence of moves to rearrange a set of blocks, assuming that one can only lift the topmost block from a pile, or keep a block only on the top of a pile. That is, each pile of blocks behaves like a stack data structure. The figure below illustrates a sample problem.

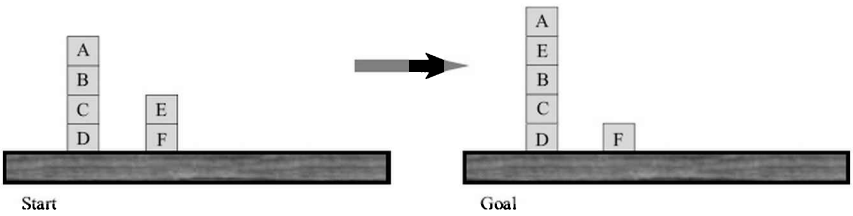

FIGURE 3.10 A blocks world problem.

In the initial state, four moves are possible as shown in Figure 3.11 below. Either block *A* can be moved, or block *E* can be moved. The destination is either the other pile or the table. Let us see how two different heuristic functions guide our *Hill Climbing* search algorithm. The two functions (also described in (Rich & Knight, 1990)) differ in the level of detail they look at in a given state.

The first function $h_1(n)$ simply checks whether each block is on the correct block, with respect to the final configuration. We add one for every block that is on the block it is supposed to be on, and subtract one for every one that is on a wrong one. The value of the goal node will be 6. Observe that the heuristic function is not an estimate of the distance to the goal, but a measure of how much of the goal has been achieved. With such a function we are looking for higher values of the heuristic function. That is, the algorithm is performing *steepest gradient ascent*. For the five states, *S* (start) and its successors *P, Q, R,* and *T*, the values are

$$h_1(S) = (-1) + 1 + 1 + 1 + (-1) + 1 = 2$$
$$h_1(P) = (-1) + 1 + 1 + 1 + (-1) + 1 = 2$$
$$h_1(Q) = 1 + 1 + 1 + 1 + (-1) + 1 = 4$$
$$h_1(R) = (-1) + 1 + 1 + 1 + (-1) + 1 = 2$$
$$h_1(T) = (-1) + 1 + 1 + 1 + (-1) + 1 = 2$$

where,

$$h_1(n) = val_A + val_B + val_C + val_D + val_E + val_F$$

Clearly h_1 thinks that moving to state *Q* is the best idea, because in that state, block *A* is on block *E*.

The second heuristic function looks at the entire pile that the block is resting on. If the configuration of the pile is correct, with respect to the goal, it adds one for every block in the pile, or else it subtracts one for every block in that pile. The values for the six nodes we have seen are

$$h_2(S) = (-3) + 2 + 1 + 0 + (-1) + 0 = -1$$
$$h_2(G) = 4 + 2 + 1 + 0 + 3 + 0 = 10$$
$$h_2(P) = 0 + 2 + 1 + 0 + (-1) + 0 = 2$$
$$h_2(Q) = (-2) + 2 + 1 + 0 + (-1) + 0 = 0\ h_2(R) = (-3) + 2 + 1 + 0 + (-4) + 0 = -4$$
$$h_2(T) = (-3) + 2 + 1 + 0 + 0 + 0 = 0$$

The first thing to notice is that the heuristic function $h_2(n)$ is much more discriminating. Its evaluation for almost all nodes is different. The second is that its choice of the move is different. It thinks that moving to state (node) *P* is the best. It is not tempted into putting block *A* onto block *E*. The reader will agree that the second function's choice is better.

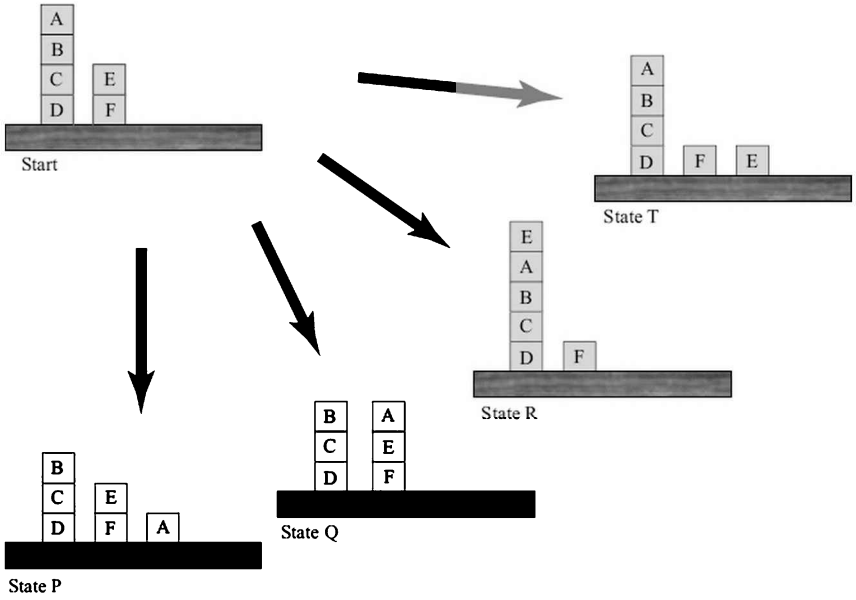

FIGURE 3.11 The first moves possible. A move consists of moving one topmost block to another place.

In both cases, since there is a successor with a better heuristic value, the *Hill Climbing* algorithm will make a move. Let us look at the second move. In the first case, for h_1, the algorithm is faced with the following choices for the second move, shown in Figure 3.12. We have *named* the states *P* and *Start* the same, though the search will see them as new choices.

The heuristic values of the possible moves are given below. Search is at node *Q*, and all choices have a lower heuristic value. Thus, *Q* is a local maximum, and *Hill Climbing* terminates.

$$h_1(Q) = 1 + 1 + 1 + 1 + (-1) + 1 = 4$$
$$h_1(P) = (-1) + 1 + 1 + 1 + (-1) + 1 = 2$$
$$h_1(S) = (-1) + 1 + 1 + 1 + (-1) + 1 = 2$$
$$h_1(U) = 1 + (-1) + 1 + 1 + (-1) + 1 = 2$$
$$h_1(V) = 1 + (-1) + 1 + 1 + (-1) + 1 = 2$$

Meanwhile, the choices faced by the search using the second heuristic function h_2, from state *P*, are shown below in Figure 3.13. Like in the previous case, two of the four choices result in states seen earlier.

The heuristic values are

$$h_2(P) = 0 + 2 + 1 + 0 + (-1) + 0 = 2$$
$$h_2(S) = (-3) + 2 + 1 + 0 + (-1) + 0 = -1$$

$h_2(Q) = (-2) + 2 + 1 + 0 + (-1) + 0 = 0$
$h_2(W) = 0 + 2 + 1 + 0 + (-1) + 0 = 2$
$h_2(X) = 0 + 2 + 1 + 0 + 3 + 0 = 6$

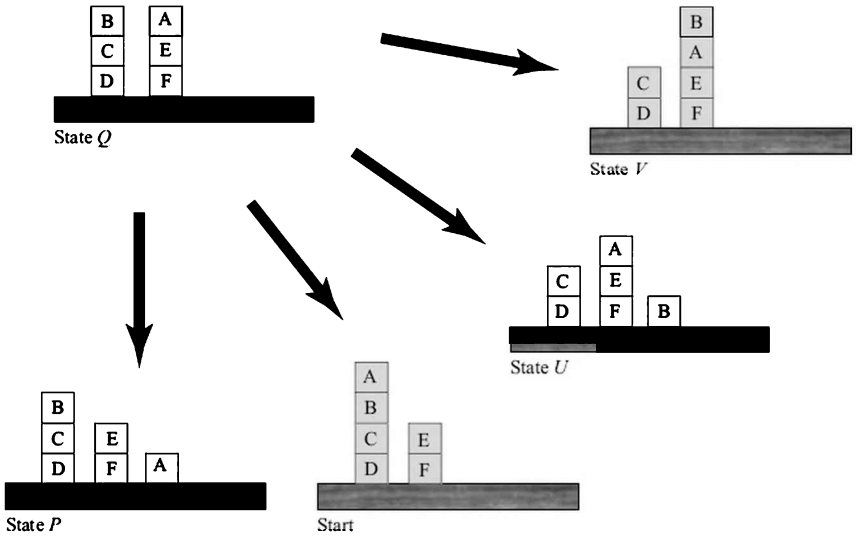

FIGURE 3.12 The choices from state *Q*.

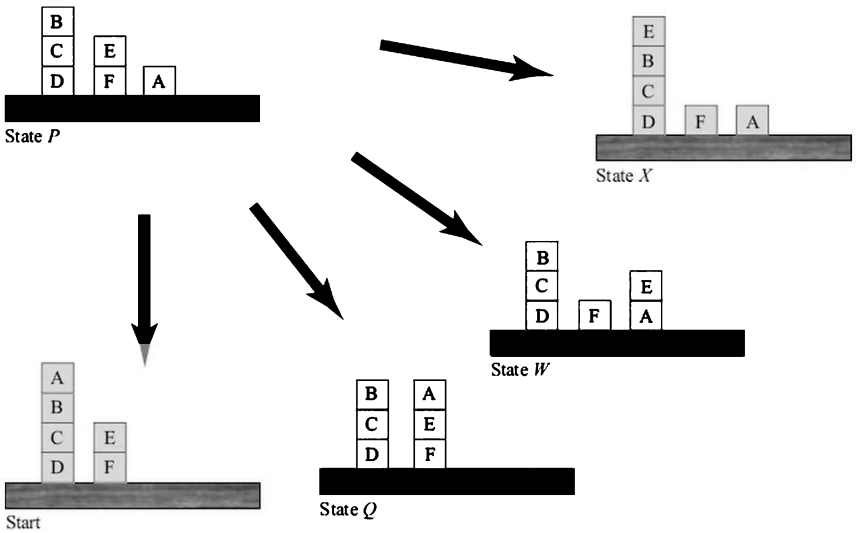

FIGURE 3.13 The choices from state *P*.

The search has a better option available in state *X* and moves to it. The reader can verify that in the next move it will reach the goal state.

Thus, we see that the performance of the *Hill Climbing* algorithm

depends upon the heuristic function chosen. One can think of the heuristic function defining a terrain over the search space, with each state having a heuristic value. While the strategy remains the same, that is the *steepest gradient ascent* (or descent, if the heuristic function is such that lower values are better), the performance depends upon the nature of the terrain being defined, as illustrated in Figure 3.14. If the heuristic function defines a smooth terrain, the search will proceed unhindered. On the other hand, if the terrain is undulating then the search could get stuck on a local optimum.

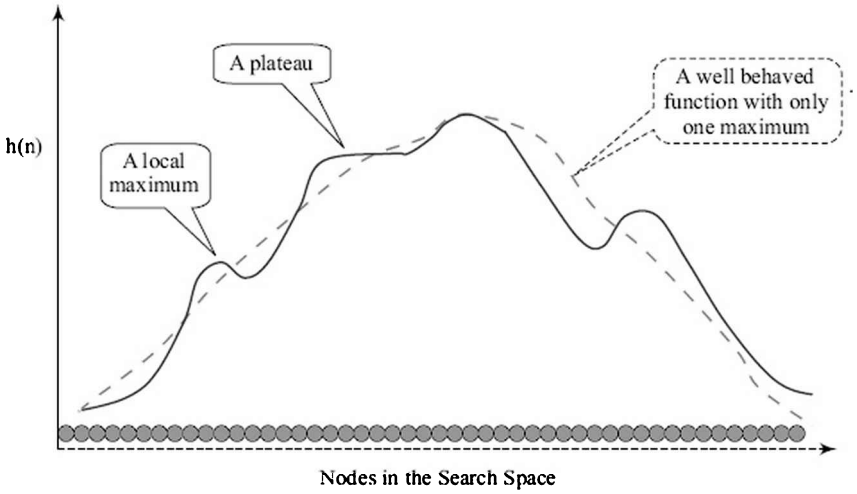

FIGURE 3.14 A well behaved function monotonically improves towards one maximum.

Let us evaluate the *Hill Climbing* algorithm on the four criteria we have been using.

3.4.1 Completeness

Algorithm *Hill Climbing* is not complete. Whether it will find the goal state or not depends upon the quality of the heuristic function. If the function is good enough, the search will still home in on the goal state. If the heuristic function yields no discriminating information (for example, if $h(n) = 0$ for all nodes n), the algorithm will terminate prematurely when it gets stuck.

3.4.2 Quality of Solution

Like *Best First Search,* no guarantee on the quality of the solution can be given.

3.4.3 Space Complexity

This is the *Hill Climbing* algorithm's strongest feature. It requires a constant amount of space. This is because it only keeps a copy of the current state. In addition, it may need some constant amount of memory for storing the previous state and each candidate successor. But overall, the space requirements are constant.

3.4.4 Time Complexity

The time complexity of *Hill Climbing* will be proportional to the length of the *steepest gradient ascent* route from the *Start* position. In a finite domain, the search will proceed along this path and terminate. Thus, one can say that the complexity of *Hill Climbing* is linear.

Overall, one can observe that the performance of *Hill Climbing* is critically dependent upon the heuristic function. The algorithm is an example of a greedy algorithm that makes locally best choices and halts when no locally better option exists. While this may work for some problem domains, in many domains, finding a well behaved heuristic function is not easy. In particular, this is so when a problem can be seen as decomposable into a set of smaller problems. For example, consider the Eight-puzzle or the Rubik's cube kind of problems. Most human solvers tend to decompose the goals into subgoals. For example, while solving the Rubik's cube, one may first do the top row, then the middle row, and finally the bottom row. Each of these subgoals is itself achieved by further decomposition. It is instructive to take a given solution for the Rubik's cube and plot the values of a heuristic function *along the solution path.* Assuming a Manhattan distance like heuristic function, the plot is likely to look somewhat like the one shown in Figure 3.15, but with more local minima. As each subgoal is achieved, the heuristic value reaches a local minimum. But further progress temporarily disrupts the heuristic value, before showing improvement again. Such problems are called problems with *nonserializable subgoals* (Korf 1985) because the subgoals cannot be independently achieved in any serial order. While solving any subgoal, extra care has to be taken to eventually restore any disrupted subgoals.

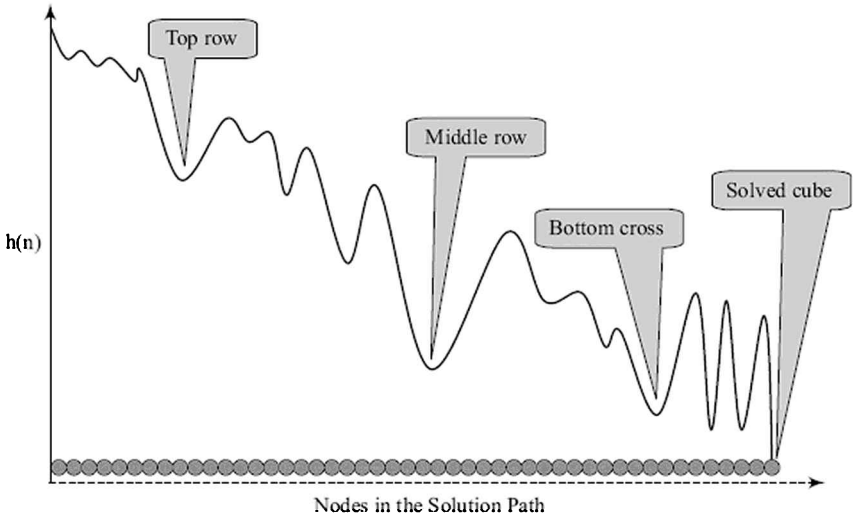

FIGURE 3.15 For human generated solutions of the Rubik's cube, the heuristic value passed through many local minima.

Given that there are going to be problems where the heuristic function will not be well behaved over the domain, there is a need to look for alternate methods to solve such problems. The simplest of them is to increase the memory available by a constant amount, so that more than one option can be kept open. This method is known as the *Beam Search*, described in a later section. But before that, we review the transformation our original problem statement has undergone, and look at an alternate search space formulation.

3.5 Solution Space Search

The problems we have seen so far have been formulated as *constructive* search problems. In a constructive search, we incrementally build the solution. A move consists of extending a given partial solution, and the search terminates when the goal state is found. For example, in the *n-queens* problem, one could start with an empty board, and place one queen at a time. Constructive searches can be both global as well as local. The term *global* and *local* refer to the regions of the search space accessed by the algorithm. The algorithm *Best First Search* is global, in the sense that it keeps the entire search space in its scope. The *Hill Climbing* algorithm, on the other hand, is local, because it is confined only to extending one given path.

Another way to formulate a search problem is with *perturbation* search. In perturbation search, each node in the search space is a *candidate solution.* A move involves perturbation of a candidate solution to produce a new candidate solution. For example, in the *n-queens* problem, the search might start with a random placement of queens, and

then each move may change the position of one or more than one queen. Perturbation searches may be local or global, depending upon whether the algorithm explores the entire search space or only a part of it. Most implementations work with searches looking for local improvements. *Hill Climbing,* the simplest of perturbation search algorithms, does precisely this, and stops when it cannot find a better candidate. Local perturbation search methods are also known as *neighbourhood* search algorithms, because they only search in the neighbourhood of the current node.

Hill Climbing has a termination criterion in which the algorithm terminates when a better neighbour cannot be found. We have already observed that in doing so, it has converted a searching for goal problem into optimization of the heuristic function problem. The *optimization community* refers to the function being optimized as the *objective function.* While focusing on the optimization problem, we will adopt this term.

Box 3.1: The SAT Problem

Consider a Boolean formula made up of a set of propositional variables $V = \{a, b, c, d, e, \ldots\}$ (see Chapter 12, propositional logic). For example,

$$F = ((a \vee \neg e) \wedge (e \vee \neg c)) \supset (\neg c \vee \neg d)$$

Each of the propositional variables can take up one of two values: *true* or *false,* known as truth values, and also referred to by 1 and 0, or *T* and F. Given an assignment *true* or*false* to each variable in the formula, the formula *F* acquires a truth value that is dictated by the structure of the formula and the logic connectives used. The problem of satisfiability, referred to as the *SAT* problem, is to determine whether there exists an assignment of truth values to the constituent variables that make the formula *true.* The formula F given above can be made true by the assignment {*a* = *true, c* = *true, d* = *false, e* = *false}* amongst others.

Very often *SAT* problems are studied in the *Conjunctive Normal Form* in which the formula is expressed as a conjunction of clauses, where each clause is a disjunction of literals, and each literal is a proposition or its negation. The reader should verify that the *CNF* version of the above formula has only one clause with 4 literals.

$$F = (\neg a \vee \neg c \vee \neg d \vee \neg e)$$

SAT is one of the earliest problems to be proven NP-complete (Cook, 1971). Solving the *SAT* problem by brute force can be unviable when the number of variables is large. A formula with 100 variables will have 2^{100} or about 10^{30} candidate assignments. Even if we could inspect a million candidates per second, we would need

3×10^{14} centuries or so. Clearly, that is in the realm of the impossible. Further, it is believed that NP-complete problems do not have algorithms whose worst- case running time is better than exponential in the input size.

One often looks at specialised classes of *SAT* formulas labelled as *k-SAT*, in which each clause as *k* literals. It has been shown that *3-SAT* is NP-complete. On the other hand, *2-SAT* is solvable in polynomial time. For *k-SAT,* complexity is measured in terms of the size of the formula, which in turn is at most polynomial in the number of variables.

Consider the *SAT* problem. It involves finding assignments to the set of variables to satisfy a given formula. For example, the following formula has five variables (*a*, *b*, *c*, *d*, *e*) and six clauses.

$$(b \vee \neg c) \wedge (c \vee \neg d) \wedge (\neg b) \wedge (\neg a \vee \neg e) \wedge (e \vee \neg c) \wedge (\neg c \vee \neg d)$$

The candidate solutions can be represented as five-bit strings, one bit for the truth value of each variable. For example, 01010 represents the candidate solution {a = 0, *b* = 1, *c* = 0, *d* = 1, *e* = 0}. In *solution space search,* we define moves as making some perturbation in a given candidate. For the *SAT* problem, the perturbation could mean changing some *k* bits. For the above example, choosing *k* = 1 will yield five new candidates. They are 11010, 00010, 01110, 01000 and 01011. These five then become the neighbours of 01010 in the search space. If we had chosen *k* = 2 then each candidate would have ten new neighbours (we can choose two bits in $^5C_2$ ways). For 01010, they are: 10010, 11110, 11000, 11011, 00110, 00000, 00011, 01100, 01110, and 01001.

The above example gives us an interesting insight into designing search spaces. For the same search space, or the set of all possible candidate solutions, different neighbourhood functions can be defined by choosing different operators. This would obviously affect the performance of the search algorithm, because all algorithms consider the set of neighbours to select a move. A *sparse* neighbourhood function would imply fewer choices at each point, while a dense function would mean more choices. The more dense the neighbourhood, the more expensive it is to inspect the neighbours of a given node. As an extreme in the *SAT* problem, one could choose an operator that changes all subsets of bits. This would mean that *all* nodes in the search space would become neighbours of the given node, and the search would then reduce to an inspection of all the candidates. Notice that with this all-subsets exchange, there is no notion of a local optimum. When *all* the candidates are neighbours, the best amongst them is the optimum, and that is the global optimum. Conversely, the more sparse the neighbourhood function, the more likelihood of there being a local optima in the search space. The local optima arise because the node (the local optimum) does

not have a better neighbour. That is, better nodes exist in the search space; but the local optimum is not connected to any of them.

The above realization leads to a simple extension of the *Hill Climbing* algorithm, known as the *Variable Neighbourhood Descent*.

3.6 Variable Neighbourhood Descent

In the previous section, we saw that one can define different neighbourhood functions for a given problem. Neighbourhood functions that are sparse lead to quicker movement during search, because the algorithm has to inspect fewer neighbours. But there is a greater probability of getting stuck on a local optimum. This probability of getting stuck becomes lower as neighbourhood functions become denser; but then search progress also slows down because the algorithm has to inspect more neighbours before each move. *Variable Neighbourhood Descent* (VDN) tries to get the best of both worlds (Hansen and Mladenovic, 2002; Hoos and Stutzle, 2005). It starts searching with a sparse neighbourhood function. When it reaches an optimum, it switches to a denser function. The hope is that most of the movement would be done in the earlier rounds, and that the time performance will be better. Otherwise, it is basically a *Hill Climbing* search. In the algorithm in Figure 3.16, we assume that there exists a sequence of *moveGen* functions ordered on increasing density, and that one can pass these functions as parameters to the *Hill Climbing* procedure.

```
VariableNeighbourhoodDescent()
  1 node ← start
  2 for i ← 1 to n
  3   do moveGen ← MoveGen(i)
  4      node ← HillClimbing(node, moveGen)
  5 return node
```

FIGURE 3.16 Algorithm Variable Neighbourhood Descent. The algorithm assumes that the function *moveGen* can be passed as a parameter. It assumes that there are *N moveGen* functions sorted according to the density of the neighbourhoods produced.

3.7 Beam Search

In many problem domains, fairly good heuristic functions can be devised; but they may not be foolproof. Typically, at various levels a few choices may look almost equal, and the function may not be able to discriminate between them. In such a situation, it may help to keep more than one node in the search tree at each level. The number of nodes kept is known as the beam width b. At each stage of expansion, all *b* nodes are expanded; and from the successors, the best *b* are retained. The memory requirement thus increases by a constant amount. The search tree

explored by *Beam Search* of width = 2 is illustrated in Figure 3.17.

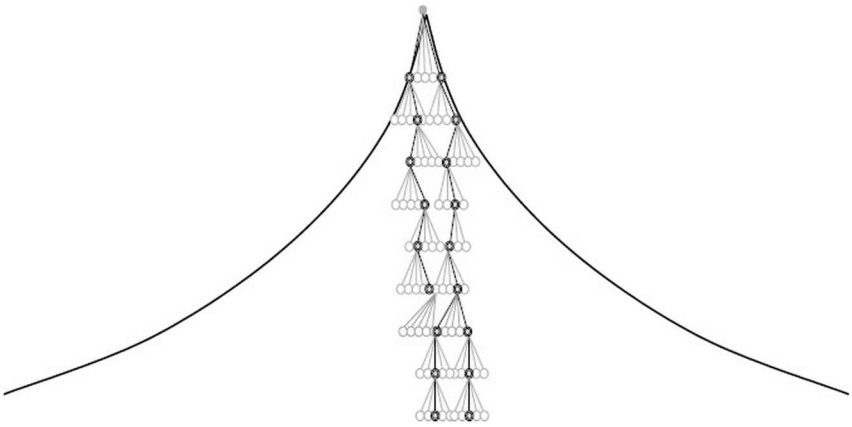

FIGURE 3.17 Beam Search with beam width = 2.

Beam Search is often used in situations where backtracking is not feasible because of other reasons. One of the places where it has been used often is in speech processing. The idea is to combine simpler units of sounds, syllables or phonemes, into bigger units. There are various rules for combining sounds to get meaningful words, and the process has to be done in a continuous (online) mode, producing candidate words as the smaller sound units come in. Since most languages have words that sound similar, where similar symbols that can combine into different word units, a speech processing system benefits by keeping more than one option open. A complete description of various techniques used in speech processing is given in (Huang et al., 2001).

We look at an illustration of *Beam Search* on the instance of *SAT* discussed earlier in the chapter, reproduced below.

$$(b \vee \neg c) \wedge (c \vee \neg d) \wedge (\neg b) \wedge (\neg a \vee \neg e) \wedge (e \vee \neg c) \wedge (\neg c \vee \neg d)$$

The candidate solutions can be represented as five-bit strings, one bit for the truth value of each variable. Let us choose the starting candidate string as 11111. For the objective function, we choose the number of clauses satisfied by the string. For the instance of the *SAT* problem given above, this value can range from 0 to 6. Note that this is an example of a maximization problem, because the value of the objective function is maximum for the goal node. We observe that e(11111) = 3. Figure 3.18 below shows the progress of *Beam Search* of width 2. For each node in the search space, the candidate string and the heuristic (objective) value are depicted. At each level, the best two nodes are chosen for expansion. Since in our problem the value of the objective function increases by a unit amount, and since the maximum value is 6, the search can move forward only three steps. This is because the *Beam Search,* being an

extension of *Hill Climbing,* is constrained to only move forward to better nodes.

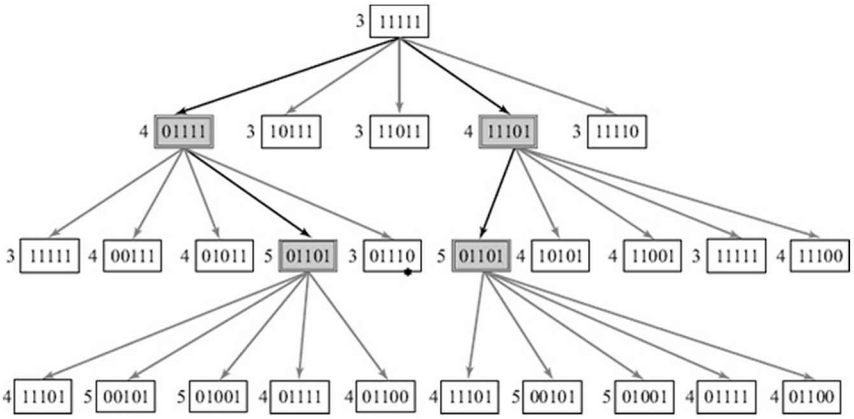

FIGURE 3.18 Beam Search with width 2 fails to solve the SAT problem. Starting with a value 3, the solution should be reached in 3 steps. In fact the node marked * leads to a solution in three steps.

If instead of 11111 we had chosen 01110, the node marked with a star in Figure 3.18, then even the *Beam Search* would have found a path of monotonic increasing values to reach a goal state (there are three different satisfying valuations for the above instance). This means that with a suitable starting point, a solution can be reached. Even with the *Hill Climbing* algorithm, one can reach the solution. An extreme case of this is if one chose a goal as the starting point itself. We explore variations of search algorithms that deploy random choices in the next chapter, and one of them is to randomly choose different starting nodes.

Another way to reach a goal node is to not terminate the search at an optimum point, but to continue looking further. This could be done while keeping track of the best solution found. But if one is to search further beyond an optimum, what should be the terminating criteria? One would then have to devise algorithms that have a criterion decided in advance. Moreover, if one is to get off an (local) optimum, what should be the condition for doing so? Do we simply ignore the idea of gradient ascent? Some possible choices that deal with random moves will be explored in the next chapter. In the following section, we explore a deterministic algorithm that exploits the gradient during search, but is also able to move off optima in search of other solutions.

3.8 Tabu Search

The main idea in *Tabu* search is to augment the *exploitative* strategy of heuristic search with an *explorative* tendency that looks for new areas in the search space (Michalewicz and Fogel, 2004). That is, the search

follows the diktat of the heuristic function as long as better choices are presented. But when there are no better choices, instead of terminating the local search as seen so far, it gives in to its explorative tendency to continue searching. Having got off an optimum, the algorithm should not return to it, because that is what the heuristic function would suggest. *Tabu* search modifies the termination criteria. The algorithm does not terminate on reaching a maximum, but continues searching beyond until some other criterion is met. One way to getting the most out of the search would be to keep track of the best solution found. This would be fairly straightforward while searching the solution space.

Tabu search is basically guided by the heuristic function. As a consequence, even if it were to go beyond a local maximum, the heuristic function would tend to pull it back to the maxima. One way to drive the search away from the maxima is to keep a finite *CLOSED* list in which the most recent nodes are stored. Such a *CLOSED* list could be implemented as a circular queue of k elements, in which only the last k nodes are stored.

In a solution space search where the moves alter components of a solution, one could also keep track of which moves were used in the recent past. That is, the solution component that was perturbed recently cannot be changed. One way to implement this would be to maintain a memory vector M with an entry for each component counting down the waiting period for changing the component. In the *SAT* problem, each bit is seen as a component, and flipping a bit as a move. Consider a four-variable *SAT* problem with 5 clauses: $(\neg a \vee \neg b) \wedge (\neg c \vee b) \wedge (c \vee d) \wedge (\neg d \vee b) \wedge (a \vee d)$. Let us say that the period before a bit can be flipped again is 2 time units. This is known as the *Tabu* tenure tt. This means that if one has flipped some bit then it can be flipped back only after two other moves. Assume the evaluation/heuristic function is the number of clauses satisfied. Let the solution vector be in the order (a b c d), and the corresponding memory vector in the same order. Let the starting candidate be (0 0 0 0). The memory vector M is also initialized to (0 0 0 0). This is interpreted that the waiting time for all moves is zero. As soon as a bit is flipped, the corresponding element in M is set to 2, and decremented in each subsequent cycle. At any point, only bits with a zero in the M vector can be considered for a move. The following figure shows the progress of *Tabu* search. After the first expansion, there are two candidates with the same value $e(n) = 4$. The two alternate expansions are shown on the left and right side with different arrows. Note that *Tabu* search would choose randomly between the two. Cells in the top row coloured grey with a thick border show a *tabu* value 2, and grey cells without a thick border depict a *tabu* value 1. These cannot be flipped in that expansion. The *tabu* bits and their values are also shown alongside in the array M. The shaded rows are the candidates that the *Tabu search* moves to.

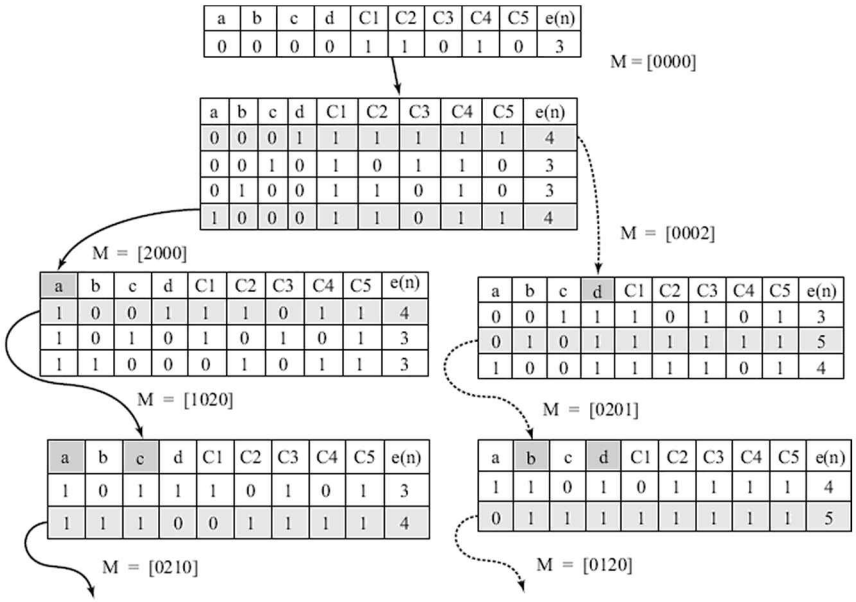

FIGURE 3.19 Two possible paths chosen by the *Tabu* search.

Consider the left branch. Starting with 0000 it goes to 1000. Only the last three bits can be changed and the best choice is 1001. Now in the next move, only the second or third bits can be changed. That is, the state 0001 is also excluded in this path.

The above example illustrates that barring certain moves in a *Tabu* search may also exclude some previously unexplored state. What if one such excluded state is a good one? One could set an aspiration criterion under which moves could overrule the *tabu* placed on them. The criteria could be that all *non-tabu* moves lead to worse nodes, and a *tabu* move yields a value better than all values found so far. Thus, while the *tabu* principle would try and distribute the search amongst different components equitably, the aspiration criteria would still allow potentially best moves to stay in the competition.

Yet another way to diversify a search could be to keep track of the overall frequency of the different moves. Note that this can also be done with a finite memory. Moves that have been less frequently used could be given preference. Imposing a penalty proportional to the frequency on the evaluation value could do this. Thus, nodes generated by frequent moves would get evaluated lower and lower, and the other moves would get a chance to be chosen.

The algorithm *TabuSearch* described below assumes that the candidate solutions have *N* components, and changing them gives *N* neighbours, which can be generated by some move generator function called *Change(node, i)* that changes the i^{th} component. The algorithm works with two arrays of *N* elements. The first called *M,* keeps track of

the *tabu* list, and is a kind of short term memory. The second called *F,* keeps track of the frequency of changing each component, and serves as a long term memory. The algorithm also assumes the existence of an *Eval*(*node*) function that evaluates a given node, and the resulting values for the neighbours are stored in an array *Value.* The algorithm written in Figure 3.20 highlights the special aspects of *TabuSearch,* and hence has explicit array computations. The search features are implicit in the calls to the functions *Change* and *Eval,* and the use of the procedure *moveTo(Index)* that finally makes the move and does the bookkeeping is shown in Figure 3.21.

```
TabuSearch(tt)
1  for i ← 1 to n
2     do M[i] ← 0; F[i] ← 0              /* initialize memory */
3  Choose the current node c randomly    /* or as given */
4  best ← c
5  while some termination criterion
6     do for i ←  1 to n
7          do              /* generate the neighbourhood */
8          tabu[i] ← YES
9          neighbour[i]  ← Change(c, i)     /*change the i'th component */
10         value[i]  ← Eval(neighbour[i])
11         if M[i]  =  0
12              then          /* if not on tabu list */
13                tabu[i] ← NO
14                bestAllowedValue ← value[i]    /*some initial value*/
15                bestAllowedIndex ←  i
16         /* BestAllowedValue is best value neighbour that is not on tabu list */
17         /* BestValue is best value amongst all neighbours */
18         bestValue ←  value[1]
19         bestIndex ←  1
20         for i  ← 1 to n
21           do            /* explore the neighbourhood */
22             if value[i] is better than bestValue
23               then   bestValue ← value[i]
24                    bestIndex ←  i
25             if value[i] is better than bestAllowedValue AND tabu[i] =  NO
26               then   bestAllowedValue ← value[i]
27                    bestAllowedIndex ← i
28         if bestAllowedValue is worse than Eval(c)
29           then if bestValue is better than Eval(best)
30                then MoveTo(bestIndex)     /*the aspiration criterion */
31                else     /* use frequency memory to diversify search */
32                  for i ← 1 to n
33                    do if tabu[i]  =  NO
34                         then   value[i] ← value[i] - Penalty(F[i])
35                              /* some initial value */
36                              bestAllowedValue ← value[i]
37                              bestAllowedIndex ← i
38                  for i  ← 1 to n
39                    do if value[i] is better than bestAllowedValue
40                               AND tabu[i]  =  NO
41                         then   bestAllowedValue ← value[i]
42                                bestAllowedIndex ← i
43                  MoveTo(bestAllowedIndex)
44           else     /* the best allowed node is an improvement */
45             MoveTo(bestAllowedIndex)
```

FIG 3.20 Algorithm *TabuSearch.*

```
MoveTo(index)
  1 c ←  neighbour[index]
  2 if  Value(index) is better than Eval(best)
  3   then best ← c
  4 F[index] ←  F [index] + 1
  5 M[index] ←  tt + 1
  6 for i ← 1 to n
  7   do if M[i] > 0
  8      then M[i] ← M[i] - 1
```

FIGURE 3.21 Procedure *MoveTo* makes the move to the new node, and does all the associated bookkeeping operations.

Tabu search is a deterministic approach to moving away from maxima. In the following chapter, we will also explore stochastic search algorithms that have explorative tendencies built into the move-generation process itself. Before doing that, we take a small detour into a knowledge based approach to navigating a difficult heuristic terrain.

3.9 Peak to Peak Methods

The problem solving algorithms seen so far operate at the operator level. Human beings, the best known problem solvers so far, rarely do so. Instead, we often break the problem into sub-problems and solve them. We also remember our problem solving experiences and learn from them. We will look at both these approaches in more detail in later chapters. Here, we look at the idea of macro operators to solve problems in a given domain.

One of the first uses of macro operators was in the *Means Ends Analysis* (MEA) problem solving strategy proposed by Herbert Simon and Alan Newell in their pioneering study on human problemsolving (Newell and Simon, 1972). The MEA strategy operates in a top down manner. Consider the problem of transporting yourself from IIT Madras to the IIT Bombay guesthouse. The *MEA* strategy attempts to identify the largest difference between the current state and the desired state, and looks for a suitable operator to reduce that difference. Say the operator is *flyToMumbai.* The problem solver now has to solve two new problems: One, to reach the Chennai airport, and the other to reach the guesthouse from the Mumbai[2] airport. In this way, the problem solver works into the details of the solution. We discuss the *MEA* strategy again in Chapter 7.

The idea of macro operators was made more explicit by Richard Korf (1985) and can be illustrated by the way we typically solve the Rubik's cube. Remember that the problem with the Rubik's cube is that it is very difficult to devise a heuristic function that will monotonically drive the search to the solution. Instead, if we were to plot the heuristic function for an expert human solver (see Figure 3.15), we find that there are stages in