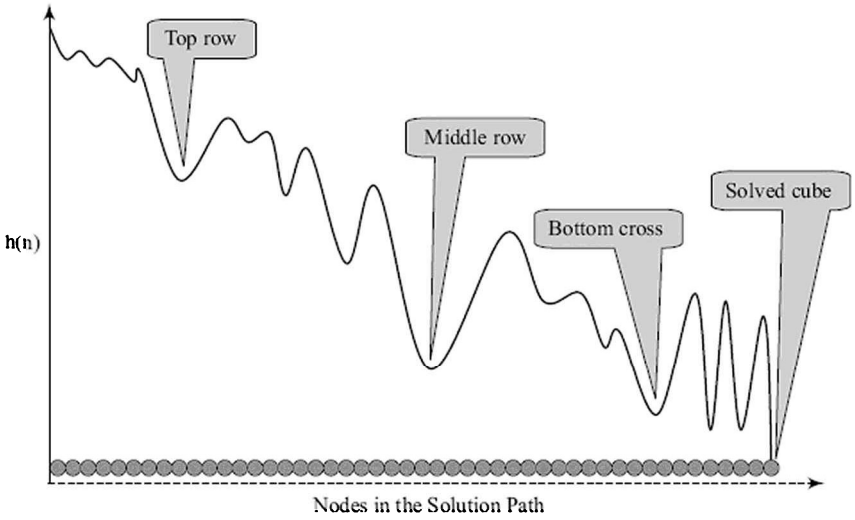

FIGURE 3.15 For human generated solutions of the Rubik's cube, the heuristic value passed through many local minima.

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.

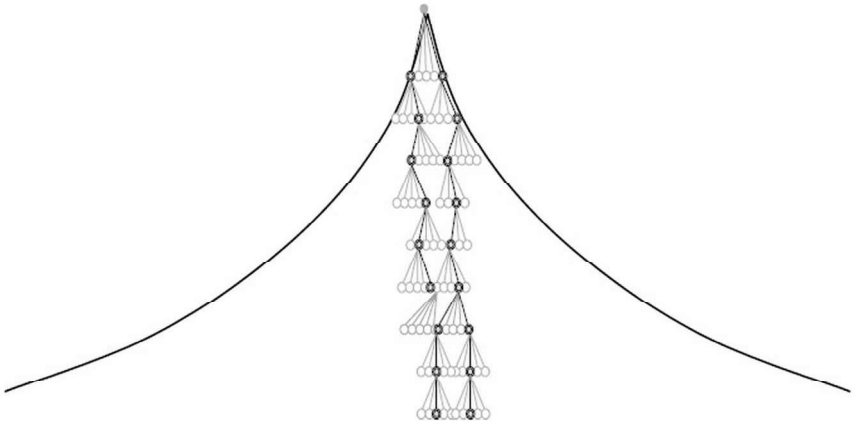

FIGURE 3.17 Beam Search with beam width = 2.

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.

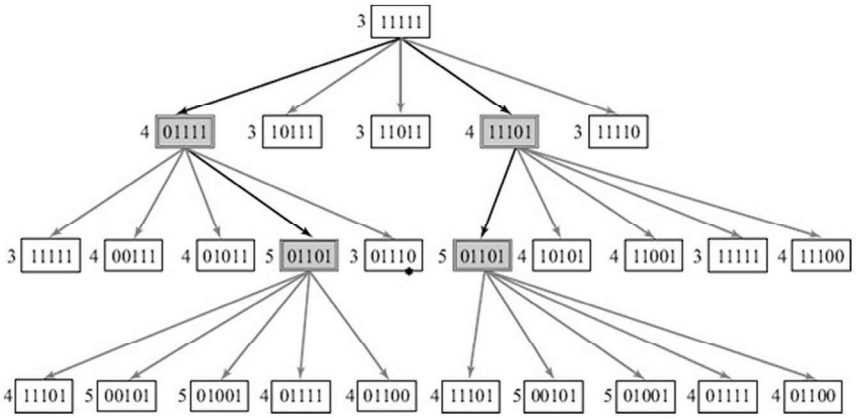

FIGURE 3.18 Beam Search with width 2 fails to solve the SAT problem. Starting with a value 3, the solution should be reached in 3 steps. In fact the node marked * leads to a solution in three steps.

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.

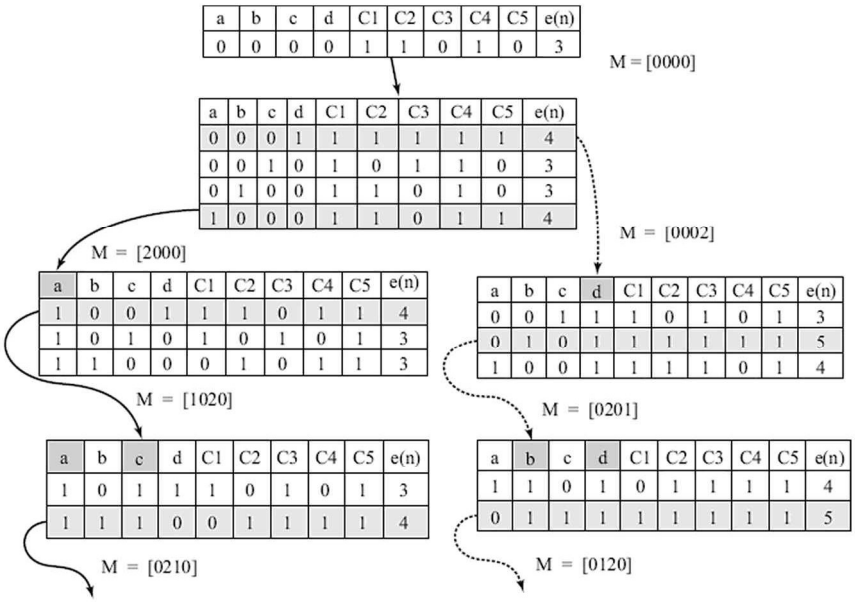

FIGURE 3.19 Two possible paths chosen by the *Tabu* search.

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