Chapter 9
Heuristic Search

Abstract This chapter provides in depth study of heuristic search methods—the methods for searching the goal (solution) to problems, that are more like human, and do not follow the exhaustive search approach, making them far more efficient than the uninformed search methods. The introduction starts with formal definition of heuristic search, then follows Hill-climbing searches, their algorithm and analysis, best-first search, its algorithm and analysis, optimization, A-star search, and approaches to better heuristics. Finally, the search methods—simulated annealing (based on treatment of metals), Genetic Algorithm (GA)-based search method, along with their analyses are presented, followed with chapter summary, and at end an exhaustive list of practice exercises, along with multiple-choice questions are provided.

Keywords Heuristic search · Hill-climbing search · Best-First Search · A-star search · Simulated annealing · Genetic algorithm

9.1 Introduction

The subject of combinatorial optimization comprises a set problems that are central to the domain of computer science. The field of combinatorial optimization aims to develop efficient algorithms to find out minimum/maximum values of some function having large number of independent variables. The function is usually called *objective* function or *cost* function, and represents the quantitative measure of the "goodness" of some complex system. The cost function depends on the total configuration of the system which comprises many parts.

The best-quoted example of a combinatorial optimization problem is Traveling Salesman Problem (TSP). It is stated as: given a list of n cities, and distance between every two cities, it is required to plan a salesman's route which guarantees to pass through every city once only, covering all the cities, and finally returns to the starting point. The order of cities to be visited should be so planned, that the total distance, to cover all the cities, is minimum. Instead of distance, it can be cost of travel from city to city, or some other parameter. For the sake of generality, we call this as *cost*. The problems of this type are common in the areas of *scheduling* and *design*. Two subsidiary problems in these areas are of general interest: 1. Predicting the estimated cost of a salesman's optimal route averaged over some arrangement of cities, having given the arrangement of cities and the distance between pairs of cities, 2. Estimating/obtaining the upper bounds of computing efforts necessary to determine the optimal route.

All the exact methods (also called exhaustive) known so far, for determining the optimal route of the salesman problem, requires the computing efforts, that grows exponentially with the total number of cities n. Hence, to carryout the solution in realistic times, the exact solution can be attempted only for a small number of cities n, may be less than a hundred. The TSP belongs to a class of large set of problems, called *NP-Complete* (NP for nondeterministic polynomial in time) problems. So far, no method of exact solution having computing effort bounded by a polynomial power of n (say n^k, $k \geq 1$) has been found for any of these problems.[1]

However, if such a solution was found for any problem, say A, which is NP-complete, then it would be possible to map to A, all the remaining NP problems, as all NP problems are member problems of A (an NP-complete). However, it is yet not known as what are the features of the individual problems that make it NP-complete, and that cause this level of difficulty of solving!

The problems of NP-complete class are at common place in many situations of practical interest, and hence the importance of their solutions. Fortunately, the solution methods, called, *heuristic methods* have been developed, which have computational requirements proportional to only a limited powers of n (the size of the problem). Unfortunately, there is no universal heuristic method which can be applied to all types of problems. In fact, the heuristics are not general, but problem-specific, and there is no guarantee that one heuristic procedure for finding near-optimal solutions for one NP-complete problem will be effective for another problem also.

Fundamentally, there exist two basic approaches to heuristics: 1. "divide-and-conquer" and 2. "iterative improvement". The first approach is based on the concept of dividing the original problem into subproblems of manageable sizes, and then the subproblems are individually solved. At the end, the solutions to the subproblems are patched back together to get the complete solution. For this method to produce a good solution, it is necessary that subproblems are naturally disjoint, and the division of the original into subproblems is proper, in the sense that errors made in patching up the subproblems do not offset the gains obtained in applying more powerful methods to the subproblems.

Other approach to heuristics is based on iterative improvement, where one has to start with a *known configuration*. And, for this configuration, a standard rearrangement operation is applied to all parts of the system in turn, until a rearranged configuration which improves the cost function is discovered. As a next step, this rearranged configuration becomes the new configuration of the system, and the process is repeated until we reach to a configuration such that no further improvements

[1]The NP-Complete problems require exponential power of computing efforts, in terms of n, i.e., k^n.

can be found. The iterative improvement comprises a search space for rearrangement, so that there is a flat region in space, indicating that an optimized solution has reached—called global maxima.

Instead of settling to global maxima, the search quite often gets stuck-up in a local maxima. Due to this, it is usual to perform the search several times, starting from different randomly generated configurations, and save the best result, so as to reach the global maxima.

This chapter presents the heuristic methods for AI search problems. These methods are better informed, hence explore the state space in a more right directions. The analysis and complexities of these methods are also discussed.

Learning Outcomes of this Chapter:

1. Describe the role of heuristics and describe the trade-offs among completeness, optimality, time complexity, and space complexity. [Familiarity]
2. Select and implement an appropriate informed search algorithm for a problem by designing the necessary heuristic evaluation function. [Usage]
3. Evaluate whether a heuristic for a given problem is admissible (i.e., can guarantee optimal solution). [Assessment]
4. Design and implement a genetic algorithm solution to a problem. [Usage]
5. Design and implement a simulated annealing schedule to avoid local minima in a problem. [Usage]
6. Design and implement A^* search to solve a problem. [Usage]
7. Compare and contrast genetic algorithms with classic search techniques. [Assessment]
8. Compare and contrast various heuristic searches vis-a-vis applicability to a given problem. [Assessment]

9.2 Heuristic Approach

The search efficiency can improve tremendously—reducing the search space, if there is a way to order the nodes to be visited in such that most promising nodes are explored first. These approaches are called *informed* methods, in contrast to the uninformed or blind methods discussed in the previous chapter. These methods depend on some heuristics determined by the nature of the problem. The heuristics is defined in the form of a function, say f, which some how represents the mapping to the total distance between start node and the goal node. For any given node n, the total distance between start and goal node is $f(n)$, such that

$$f(n) = g(n) + h(n), \tag{9.1}$$

where $g(n)$ is distance between start node and the node n, and $h(n)$ is the distance between node n and the goal node. We note that $g(n)$ can be easily determined and can be taken as shortest. However, the distance to goal is $f(n)$, which requires computation of $h(n)$, called heuristics, cannot be so easily determined. In deciding the next state every time, which is represented by node n, the state is chosen such that $f(n)$ is minimum.

Considering the case of the traveling salesman problem, which otherwise, is a combinatorially explosive problem, with exponential time complexity of $O(n!)$ for n nodes, reduces to only $O(n^2)$ if every time the next node selected is the nearest neighbor, that is, the one having shortest distance from the current node.

Similarly, in the 8-puzzle problem, the next move is chosen the one having minimum *disagreement* from the goal, i.e., having minimum number of misplaced positions with respect to the goal.

The heuristic methods reduce the state space to be searched, and supposed to give the solution, but may fail also.

9.3 Hill-Climbing Methods

The name hill-climbing comes from the fact that to reach the top of a hill, one selects the steepest path at every node, out of the number of alternatives available. Naturally, one has to sort the slope values available, pick up the direction of move having highest angle, then reach to the next point (node) toward the hill top, then repeat the process. The hill-climbing Algorithm 9.1 is an improved variant of the depth-first search method. A Hill-climbing method is called *greedy local search*. Local, because it considers a node close to the current node, at a time; and greedy because it always selects the nearest neighbor without knowing its future consequences. The inputs to this algorithm are **G**, **S**, and **Goal**, which stand for—graph, start(root) node, and the goal node, respectively.

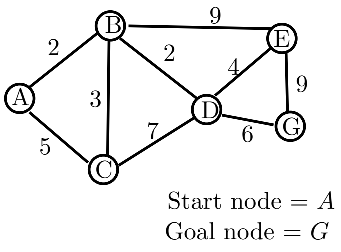
Consider the graph shown in Fig. 9.1a, where start node is A and goal node is G. It is required to find out the shortest path from node A to node G, using the method of hill-climbing. Figure 9.1b shows the search tree for reaching to goal node G from start node A, with shortest path A, B, D, E, G, and path length 14. It can be easily worked out that this approach cannot lead to shortest path always.

Though simple, hill-climbing suffers from various problems. These problems are prevalent when hill-climbing is used to optimize the solution.

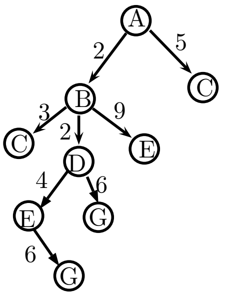
- **Foothill Problem**: This occurs when there are secondary peaks or *local maxima*. These are mistaken for the *global maxima*, as the user is left with false sense of achieving the goal.
- **Plateau Problem**: This occurs when there is a flat region separating the peaks.
- **Ridge Problem**: It is like a knife edge or an edge on top of a hill, both the sides are valleys. It again gives a false sense of top of the hill, as no slope change appears.

Algorithm 9.1 Hill-Climb(Input: **G**, **S**, **Goal**)

```
1: Open = [$S$]
2: Closed = $nil$
3: if Open = $nil$ then
4:    return fail
5: end if
6: repeat
7:    if Open.$Head$ = Goal then
8:       return success
9:    end if
10:   expand Open.$Head$ and generate children's set, call it C
11:   reject all paths in C having loops
12:   delete Open.$Head$ and insert it into Closed
13:   sort C in order of heuristic, with best heuristic node in the front
14:   insert C at the front of List
15: until Open = $nil$
16: Return fail
```



(a) Graph to be searched for goal 'G'.



(b) Search-Tree.

Fig. 9.1 Graph with Hill-climbing search

Due to above difficulties, a hill-climbing algorithm is not able to find a goal, even if the goal exist. Thus, hill-climbing algorithms are *not complete*.

Consider the following phenomenas:

1. Rotating the brightness knob in control panel of an analog TV does not improve the quality of picture,
2. While testing a program, running it again and again, with different data sets does not indicate new discovery of errors,
3. Participating in a sports again and again (without new ideas and training), does not improve further performance.

In all the above three cases, we strive for optimum performance. In case 1, the adjustable parameter is TV control, in second types of different input data, and in the third, adjustable parameter is more energy and preparedness. But it is appears that optimum has reached (a highest point in performance, from where no improvements take place, an indication of saturation point, the goal, but not the true goal).

The above scenarios are created due to either of the foothill problem, or plateau, or ridge.

Local Versus Global Search

The preference for local search where only one state is considered to further expand at a time, out of the newly explored states, is good for a memory efficiency, in comparison to expanding all the successor nodes, but appear to be a too extreme step. Thus, instead, k best nodes out of successors generated are considered for further expansion. This gives rise to a new method, called *local beam search*.

However, the local beam search too will lead to concentration to a specific direction to those successors which generate more potential nodes. Consequently, this search also ultimately becomes a local search. A better solution is to elect these k nodes such that they are not the best successors, but randomly selected, out of the next generation of successor nodes. This new method is called *stochastic beam search*. Thus, we grow a population (of nodes) and from that we generate further another population by random selection, as well as on some criteria of merit. Thus, we reach closely to the approach used in *genetic algorithms*-based search.

9.4 Best-First Search

If a problem that has a very large search space and can be solved by iteration (unlike theorem proving),[2] there is usually no alternative to using the iterative methods. In such problems, there is a serious issue in bounding the effort to make the search tractable. Due to this, the search should be kept limited in some way, e.g., in terms of total number of nodes to be expanded, or maximum depth to which it may reach. Since there is no guarantee that a goal node will be ultimately reached, an evaluation function is invoked to help us decide the approximate distance to the goal, for any given node at the periphery of the search. This or a similar function can also be used for deciding which tip node to sprout next. Hence, the effort for proper design of valuation functions limits the later stage difficulty in solving the problem. Note that heuristics-based methods are called *informed search* methods, in contrast to the *uninformed* methods discussed in the previous chapter.

Among all the problem-solving strategies based on search the heuristics, one of the most popular methods of exploiting heuristic information to cut down search time is *best-first* search strategy. This strategy possess general philosophy of using heuristic information to assess the "merit" latent in every candidate search avenue exposed during the search, and then continues the exploration along the direction of highest merit. There are no local maxima "traps" in best-first search like in hill-climbing methods. The best-first search strategy is usually seen in context of path-searching problems—a formulation that represents many combinatorial problems with practical applications, such as routing telephone traffic, layout of printed circuit

[2]In problems such as theorem proving, the search must continue until a proof is found.

board, scheduling, speech recognition, scene analysis, mechanical theorem proving, and problem-solving.

The heuristic approach typically uses special knowledge about the domain of the problem being represented by the graph to improve the computational efficiency of solution to particular graph-searching problem. However, the procedures developed via the heuristic approach generally have not been able to guarantee that minimum cost solution paths will always be found.

Given a weighted directional graph $G = (V, E, W)$ with a distinguished start node **S** and a set of goal nodes R, the optimal path problem is to find a least cost path from **S** to any member of R where the cost of the path may, in general, be an arbitrary function of them, weights assigned to the nodes and branches along that path. A Generalized Best-First Search (GBFS) strategy will pursue this problem by constructing a tree T of selected paths of G using the elementary operation of node expansion, that is, generating all successors of a given node, Starting with **S**, the GBFS will select for expansion that leaf node of T that features the highest "merit," and will maintain in T all paths which have been encountered so far. And, that still appear as viable candidates for *sprouting* an optimal solution path. When no such candidate is available for further expansion, the search terminates. In that case the best solution path found so far is issued as a solution; if none has been found, a failure is declared. Due to its nature of search, the best-first search is also called *branch-and-bound method*, i.e., branching a search to other directions to which path cost is minimum, and bounding the cost to that minimum, until a better minimum is found after next expansion.

9.4.1 GBFS Algorithm

A best-first search algorithm maintains two lists of nodes, an "Open-list"and a "Closed-list". The Closed-list contains those nodes that have been expanded, by generating all their children, and the Open-list contains those nodes that have been generated, but not yet expanded. At each iteration of the algorithm, an Open node having smallest total cost from start node, is expanded, moved to the Closed-list and, its children are added into the Open-list [2].

The best-first search method takes the best node to be explored first, among the number of nodes in the open-list. When a node is selected as a candidate for expanding, its all children's distance is computed from the start node, which serve as heuristic value. This approach works because there is always a best node available for expanding, until the goal is reached or the entire search has taken place.

Since, best node is selected every time, it is guaranteed to give the best solution. The value of heuristic function $f(n)$ for a given node n, does not here include the distance from current node to the goal node, as required in Eq. (9.1, page no. 241), however, since the best path is chosen every time, it is likely to provide the optimum solution for the problem.

Algorithm 9.2 shows the steps for best-first search.

Algorithm 9.2 Best-first Search(Input: **S**, **Goal**)

```
1: Open = [$S$]
2: Closed = []
3: repeat
4:   if Open.$Head$ = Goal then
5:     return $success$
6:   end if
7:   generate children's set C of Open.$Head$
8:   if $n \in C$ already exists in OPEN and new $n$ is reachable by shorter path then
9:     remove the old $n$
10:  end if
11:  if $n \in C$ already exists in Closed and reachable by shorter path then
12:    replace $n \in C$ by the same node from Closed, along with shorter distance from root
13:  end if
14:  remove Open.$Head$ and insert into Closed
15:  update distance from root for all C nodes
16:  add all C to either side of Open and record their parents
17:  sort Open by path length so that least cost path node is at front
18: until Open = $nil$
19: return $fail$
```

Example 9.1 Best-first search.

Fig. 9.2a shows the graph, and Fig. 9.2b shows the search tree using GBFS for reaching to goal node G.

Every node in best-first search shows the node identification along with its distance from the root node S. The order in which the nodes are explored is shown with dotted line. Since G is goal, its path from root is S, A, C, G with shortest path length 11. Note that any other path will be of longer or equal length. □

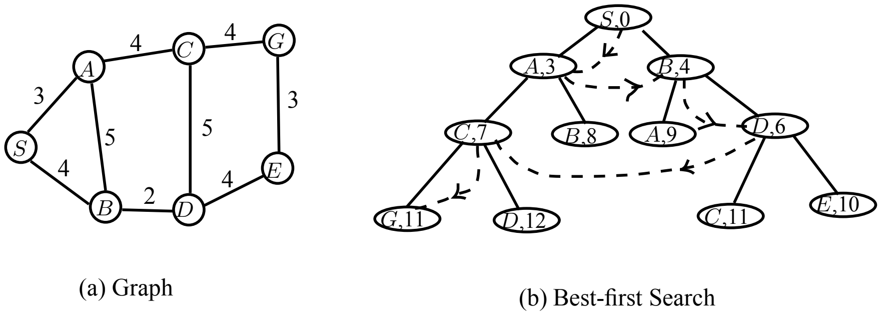


Fig. 9.2 Best-First (Branch-and-Bound) search

9.4.2 Analysis of Best-First Search

As we have noted that, the best-first searches tend to put the searching effort into those subtrees that seem most promising (i.e., they are most likely of providing the best solution). However, the best-first searches require a great deal of bookkeeping for keeping track of all competing nodes, contrary to the great efficiencies possible in depth-first searches.

Depth-first searches, on the other hand, tend to be forced to stop at inappropriate moments thus giving rise to the horizon effect (number of possible states is immense and only a small portion can be searched). They also tend to investigate huge trees, large parts of which have nothing to do with any solution (since every potential arc of the losing side must be refuted). However, these large trees sometimes turn up something that the evaluation functions would not have found were they guiding the search. Sometimes the efficiencies and discovery potential of the depth-first methods appear to out-weight what best-first methods have to offer. In fact, both methods have some glaring deficiencies.

Optimizing Best-First Search

In practice, several shortcuts have been devised to simplify the computation of GBFS. First, if the evaluation function $f(n)$ used for node selection always provides optimistic estimates of the final costs of the candidate paths evaluated, then we can terminate the search as soon as the first goal node is selected for expansion, without compromising the optimality of the solution used. This guarantee is called *admissibility* and is, in fact, the basis of the branch-and-bound method. This we can observe, for example in a chess game, where goal is property of a configuration and not the property of path from start node. Hence, once a winning configuration is reached, there is no need to try it from other paths.

Second, we are often able to purge from tree T, large sets of paths that are recognized at an early stage to be dominated (i.e., superior) by other paths in T. This becomes particularly easy if the evaluation function f is *order-preserving*, that is, if, for any two paths p_1 and p_2, leading from **S** to n, and for any common extension p_3 of those paths, the following holds (Fig. 9.3) [2]:

$$f(p_1) \geq f(p_2) \Rightarrow f(p_1 p_3) \geq f(p_2 p_3). \tag{9.2}$$

The property of *Order-preserving* is a judgmental version of the principle of *optimality* in *Dynamic Programming*. The principle states that, a path p_1 is judged to be more meritorious than another path p_2, such that both are paths from one source state **S** to some future state n, and there is a common extension p_3 of p_1 and p_2. In such a scenario, the common extension cannot later reverse the judgment made earlier. Under such conditions, there is no need to keep multiple copies of nodes in the tree T. Every time, the expansion process generates a node n, which already resides in T, only lower path to node n be maintained, and the link from more expensive parent of n is discarded. This has been illustrated in Fig. 9.3.

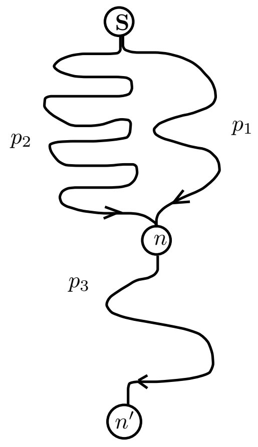


Fig. 9.3 Order-preserving in GBFS

The best-first search allows revisiting the decisions. This is possible when a newly generated state by expansion of one of the state in Open-list is found in the closed-list also. The best-first would retain the shorter path to this node, and purge the other to save space. However, in a variant of best-first, called, *greedy best-first* search once a state is visited the decision is final and the state is not visited again, thus eventually accepting the suboptimal solution. This however, does not require the *Close-list*, thus saving the memory space significantly.

Special Cases of Best-First Search

Individual best-first search algorithms differ primarily in the cost function $f(n)$. If $f(n)$ is the total depth of node n (not the distance from start), best-first search becomes breadth-first search. Note that breadth-first searches all the closer nodes (to start) before farther nodes. If $f(n) = g(n)$, where $g(n)$ is the cost of the current path from the start state to node n, then best-first search becomes Dijkstra's single-source shortest-path algorithm.

If $f(n) = g(n) + h(n)$, where $h(n)$ is a heuristic estimate of the cost of reaching a goal from node n, then best-first search becomes a new algorithm A^* algorithm.

Breadth-first search can terminate as soon as a goal node is generated, while Dijkstra's algorithm and A^* must wait until a goal node is chosen for expansion to guarantee *optimality*. Every node generated by a best-first search is stored in either the Open- or Closed-lists, for the following reasons:

1. To detect when the same state has previously been generated. This is to prevent expanding it more than once.
2. To generate the solution path once a goal is reached. This is done by saving with each node a pointer to its parent node along an optimal path to the node, and then tracing these pointers back from the goal state to the initial state.
3. To choose only the node, which is at shorter distance, when a newly generated node already exists in the closed-list.

The primary drawback of best-first search is its memory requirements. By storing all nodes generated, best-first search typically exhausts the available memory in very short time on most machines.

While breadth-first search manages the Open-list as a first-in first-out queue, generalized best-first searches manage the Open-list as a *priority-queue* in order to facilitate efficiently determining the best node to expand next.

All of these algorithms suffer the same memory limitation as breadth-first search, since they store all nodes generated in their "Open"or "Closed"lists, and will exhaust the available memory in a very short time.

9.5 Heuristic Determination of Minimum Cost Paths

The objective of heuristic determination of minimum cost path is to find an algorithm that searches a graph, $G = (V, E)$ to obtain an optimal path from start node S to its perfect goal node t. In the search process, each time a new node is expanded, two things are stored with each successor node: 1. The cost of reaching to n through a least cost path created so far, and 2. A pointer to the predecessor node of n. Ultimately, the algorithm gets terminated at some goal node t, and no further nodes are expanded. At this state, we can reconstruct the minimum cost path from S to t, simply by chaining back the nodes from t to S through the pointers to predecessor nodes.

In order to expand as few nodes as possible for searching an *optimal* path, the search algorithm must constantly make an informed decision about what node is to be expanded next. The expansion of nodes that are not going to be in the optimal path, will result to wastage of efforts. On the other hand, if the algorithm ignores the nodes that might be in the optimal path, it will fail to find such a path, in that case the algorithm is not *admissible*. Thus, a good algorithm obviously needs some way to evaluate the available nodes to determine which node to expand next.

Consider that an *evaluation function* could be calculated for some node n. Let, $f^*(n)$ is estimated minimum distance from start state to goal state, constrained through node n. Assume that this evaluation function be defined in such a way that the node with smallest value of f^* is expanded next. We will show that for a suitable choice of the evaluation function f^*, the algorithm A^* is guaranteed to provide an optimal path to a preferred goal node from the start node S, which is sufficient condition for the *admissibility* of the algorithm.

9.5.1 Search Algorithm A*

By far, the most studied version of best-first-search is the algorithm A^*, which was developed for additive cost measures, that is, where the cost of a path is defined as the sum of the costs of its arcs. The A^* is in fact a family of algorithms, we will see it shortly. The algorithm makes use of ordered state-space search and the estimated heuristic cost to determine the evaluation function f^*, a function which provides the goal state. This process is similar to the best-first search, but now it is unique in the sense that it defines f^*, which will provide a guarantee of optimal path to the goal state. The A^* algorithm is in the class of *branch-and-bound* algorithms, which are common in use in operations research for finding the solution of a problem, in the form of a shortest path in a graph [1].

The evaluation function $f^*(n)$ estimates the quality of the solution path through node n, and it is based on values returned from two components: 1. $g^*(n)$, and 2. $h^*(n)$. The first component is the *minimal cost* of a path from a start state to n, and the second component, called *heuristic value*, is a *lower bound* on the minimal cost of a solution path from state n to goal state.

In graphs, g^* can have error only in the direction of overestimating the minimal cost. In future steps of the algorithm, if a shorter path is found, the value of g^* can be readjusted to lower side. The function h^* carries the heuristic information, such that it has the capability that ensures that value of $h^*(n)$ is less than $h(n)$. This later condition is essential to the optimality of A^* algorithm. The property, as per which, $h^*(n)$ is always less than $h(n)$, is called *admissibility condition.*

To match this cost measure, A^* employs an additive evaluation function $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of the currently evaluated path from S to n and h is a heuristic estimate of the cost of the path remaining between n and some goal node. Since $g(n)$ is *order-preserving* and $h(n)$ depends only on the description of the node n, therefore $f(n)$ is also order-preserving, and one is justified in discarding all but one parent for each node.

The admissible search algorithm for A^* (A-star) is given as Algorithm 9.3.

Algorithm 9.3 Admissible search A^*(Input: **G**, **S**, **Goal**)

```
1: Open = [$S$]
2: Closed = []
3: compute $f^*(n)$ for all $n \in$ Open
4: repeat
5:    select the open node $n$ whose $f^*(n)$ is smallest
6:    resolve ties arbitrarily, but always in favor of any node $n \in$ Goal
7:    if $n \in$ Goal then
8:       move $n$ to Closed
9:       terminate algorithm
10:   else
11:      move $n$ to Closed
12:      apply successor operator to $n$
13:      calculate $f^*$ for each successor $n_i$ of $n$
14:      move all $n_i \notin$ Closed to Open
15:      move to Open any $n_i \in$ Closed and for which $f^*(n_i)$ is smaller now than it was when $n_i$
         was in Closed
16:   end if
17: until Open = $nil$
18: return $fail$
```

9.5.2 The Evaluation Function

Let, for any graph G and any set of goal nodes $Goal$, let us assume that $f(n)$ is the actual cost of an optimal path that is restricted to go through only the node n, i.e., from source S to a preferred goal node n. Note that at the begin of the A^* search algorithm, the constrained node n is nothing but S. Hence, $g(n) = g(S) = 0$. Therefore, $f(S) = h(S)$ is the cost of unconstrained optimal path from node S to preferred goal node, what so ever it is. Actually, for every node n on optimal path, the condition $f(n) = f(S)$ holds, and for every node n not on an optimal path, $f(n) > f(S)$ holds. Thus, although $f(n)$ may not be known in advance, it seems reasonable to use the evaluation function $f^*(n)$ as an estimate of $f(n)$. This is because determination of the true value of $f(n)$ may be main problem of interest.

In the following, we present some properties of search algorithm A^* where cost $f(n)$ of an optimal path through node n is estimated using an evaluation function $f^*(n)$. The function $f(n)$ can be written as the sum of two parts, $g(n)$ and $f(n)$:

$$f(n) = g(n) + h(n). \tag{9.3}$$

In the above, $g(n)$ is the actual cost of an optimal path from node S to n, and $h(n)$ is the actual cost of an optimal path from node n to a preferred goal node.

If we had the estimates of g and h, we could easily get the estimate of $f(n)$, as the simple addition of the two. An obvious choice for $g^*(n)$ is, so far smallest cost path found by the algorithm, from S to n. This indicates that $g^*(n) \geq g(n)$.

Through a simple example we will explain that the above estimate is easy to compute as the algorithm progresses through its computations [6].

Example 9.2 Consider the subgraph shown in Fig. 9.4, with start node S and three other nodes n_1, n_2, n_3, with cost on edges as the weights.

Having given this, we trace the algorithm A^* as it proceeds. With S as start node, n_1 and n_2 are the successor nodes. The estimates for $g^*(n_1)$ and $g^*(n_2)$ are then 4 and 9, respectively. Let, A^* expands the next node as n_1, and obtains the successors n_2 and n_3. At this stage $g^*(n_3) = 4 + 3 = 7$, and $g^*(n_2)$ is $4 + 4 = 8$. The value of $g^*(n_1)$ ultimately remains equal to 4, irrespective of the goal node. □

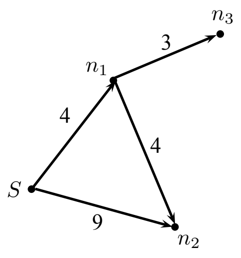


Fig. 9.4 Admissibility test

We have the following arguments for an estimate $h^*(n)$ of $h(n)$, which we are not able to compute for this example, as no criteria for heuristics is specified here. We usually rely on information from the problem domain for heuristics. For example, many problems that can be represented in the form of problem of finding minimum cost path through a graph that contains some "physical" information, and this information is used to form the basis for estimation of h^*. When considering the connection between cities through roads, the value $h^*(n)$ might be air distance from city n to goal city, because this distance is the shortest possible length of any road connecting city n to the goal city. Thus, it is lower bound on $h(n)$. However, the above conclusion is based on the assumption that air connectivity between any two cities follows a straight line rule.

As another example, in an 8-puzzle problem, at node n, the distance $h^*(n)$ might be equal to the number of tiles dislocated with respect to the goal state.

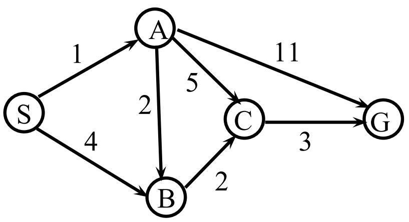
We will discuss later about using information from specific problem domains, to form estimate of f^*. However, first we can prove that if h^* is any lower bound of h, then the algorithm A^* is admissible.

Example 9.3 A^*-Search.

Figure 9.5 shows a graph, heuristic function table for $h(n)$ for every node in the graph, and tree constructed for A^* search for the graph, for given start state S and goal state G. To expand the next node, the one having smallest value of function f is chosen out of the nodes in the frontiers. The function f for a node n is sum of three values: the g value of the parent of n, the distance from parent of n to the node n, and heuristic value (estimated distance from n to goal, given in the table) indicated by h. In the case of a tie, i.e., two states having equal values of f, the one to the left of the tree is chosen.

If, in addition, $h(n)$ is a lower bound to the cost of any continuation path from n to goal, then $f(n)$ is an optimistic estimate of all possible solutions containing the currently evaluated path. Then, terminating A^* upon the selection of the first goal node does not compromise its admissibility. Several other properties of A^* can be established if admissibility holds, such as the conditions for node expansion, node reopening, and the fact that the number of nodes expanded decreases with increasing h.

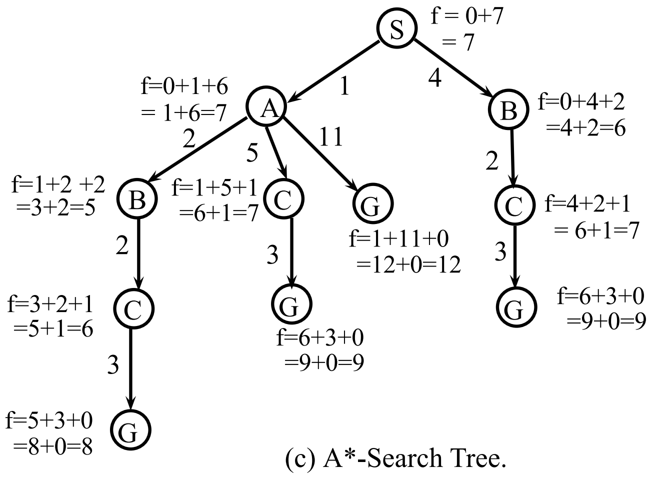
Based on the criteria set for A^* (i.e., to always expand the node having smallest value of f). The distance to goal (f) for each node n is: distance from source S to parent, plus distance from parent to this node n, plus distance h from this node n to goal. For current node $n = A$, these distances are 0, 1, 6, respectively. The order in which nodes have been expanded for Fig. 9.5a, and shown in search tree in Fig. 9.5c with start node S and goal node G are: $(S, 0)$, $(B, 6)$, $(A, 7)$, $(B, 5)$, $(C, 6)$, $(C, 7)$ (with parent A), $(C, 7)$ (with parent B), $(G, 8)$, $(G, 9)$, $(G, 12)$. Finally, we note that the best path is corresponding to goal $(G, 8)$, and it is: S, A, B, C, G. Note that, in the A^*-tree, we followed the sequence $(S, 0)$, $(B, 6)$, with $(A, 7)$ and not $(C, 7)$, which are equally weighted. We chose the node to the left-side subtree. Had we chosen, $(C, 7)$ in place of $(A, 7)$, we would have reached to $(G, 9)$ as next node, which incidentally was not a good choice. □



(a) Graph.

State	h
S	7
A	6
B	2
C	1
G	0

(b) Heuristic Table.



(c) A*-Search Tree.

Fig. 9.5 Graph and A*-Search tree

9.5.3 *Analysis of A* Search*

The A^* is actually a family of search algorithms, as mentioned earlier, and many other search algorithms are special cases of this algorithm.

The estimated value $h^*(n)$ for $h(n)$ can be obtained from the problem domain. For example, in the case of 8-puzzle problem or the 8-queen problem, this value is inverse of the number of tiles out of place with the goal or the inverse of number of queens giving checks to other queens. Even better value is, actual number of moves from n to goal.

Let us consider the specific cases of $f^*(n)$ as follows:

- When $h = 0$, then $g = d$ (the distance to goal in the search tree). This algorithm is called as **A**, and it is identical to Breadth-First Search (BFS).
- We claimed that the *BFS* algorithm is guaranteed to find the minimum path length to the goal node. If h is lower bound on h^*, i.e., $h(n) \leq h^*(n)$, for all n, then the algorithm will find an optimal path to a goal node. This algorithm is called as $\mathbf{A}^*$.

*9.5.4 Optimality of Algorithm A**

A search algorithm B is called *optimal* if there does not exist any other search algorithm performing the searching in less time or space or can do the job by expanding fewer nodes, having a guarantee of solution quality as that of algorithm A. Hence, if $h(n)$ is lower bound on $h^*(n)$ then the solution of $\mathbf{A}^*$ is *admissible*. This estimate also concludes that any open node n may even be arbitrarily close to a preferred goal node.

In one way, we can define an optimal search algorithm as one that picks the correct next node at each choice. However, this specification of an algorithm is not of much use as this much specification is insufficient to design an algorithm. In fact, whether such an algorithm may ever exist is also an open question in itself [6].

9.6 Comparison of Heuristics Approaches

The heuristic search that finds the shortest path to a goal wherever it exists is called *admissible*. We may like to know, in what sense one heuristics is better than other, is called *informedness* of the heuristics.

When search is made, it is expected that same node will not be accessible from a shorter path later on. This property is called *monotonicity*.

The breadth-first search algorithm is admissible algorithm, because it searches a path at level n, before searching the paths at level $n + 1$, hence if the goal exists at level n it will be certainly found out. However the BFS algorithm is too expensive as a general purpose algorithm.

The $\mathbf{A}^*$ algorithm does not require $g(n) = g^*(n)$. This shows that there may be subgoals in the path, which are longer than $g^*(n)$; this is due to monotonicity [8].

Definition 9.1 (*Monotonicity*) A heuristic function h is monotonous if for all the nodes n_i, n_j, where n_j is descendant of n_i, such that

$$h(n_i) - h(n_j) \leq cost(n_i, n_j), \tag{9.4}$$

where $cost(n_i, n_j)$ is actual cost, in number of moves from node n_i to n_j.

Definition 9.2 (*Informedness*) For a problem, suppose there are two A^* heuristic functions h_1 and h_2. Then, if $h_1(n) \leq h_2(n)$, for all states n in the search space, the heuristic h_1 is called *more informed* that h_2.

For example, the criteria of number of tiles out of place, in the 8-puzzle is better informed than the breadth-first or depth-first search methods. Similarly, the heuristics which calculates the number of transitions to reach the goal is better informed than the one considering the heuristics based on the number of tiles out of place. In general, a more informed is an algorithm, there is less expansion of space for searching [6].

Approaches to Better Heuristics

From the above two examples, we note that in some cases of search, the path matters (TSP), while in other the path does not matter, and only the final configuration matters (8-puzzle).

A simple algorithm for heuristic search could be considering only single state at a time rather than many states corresponding to many paths sprouting at the same time. Such algorithms are called *local search* in contrast to the *global search*, which maintains many active paths at the same time. The local search algorithms consume much less memory, usually a constant amount.

The *Branch-and-bound* algorithms are implicitly enumeration algorithms. These are the principal general methods for finding out optimal solutions for discrete optimization problems. The branch-and-bound algorithms are based on the following parameters:

$$(D, E, L, N, P, U), \tag{9.5}$$

where

D: Node dominance function,
E: Set of node elimination rules,
L: Node lower bound solution cost function,
N: Next node selection rule,
P: Partitioning or branching rule, and
U: Upper bound solution cost function.

A branch-and-bound algorithm is a two-step algorithm: first step is a *splitting or branching* step which returns two or more smaller sets $S_1, S_2, \ldots$, whose union covers S, where S is set of candidates. Minimum of $f(x)$ over S is $min\{v_1, v_2, \ldots\}$, where each v_i is the minimum of $f(x)$ within S_i. The recursive application of this step defines a tree structure (a search tree) whose nodes are the subsets of S. The second step, called *bounding*, computes upper and lower bounds of the minimum value of $f(x)$, within a given subset of S.

Application of the branch-and-bound technique has grown rapidly. Representative examples of this include: *flow-shop* and *job-shop* sequencing problem, traveling salesman problem, *integer programming* problem, and *general quadratic assignment* problem. Though, the branch-and-bound algorithms are usually more efficient than complete enumeration, however, these algorithms have computational requirements that usually grow exponentially or high degree polynomial of the problem size n. In these cases, their usefulness is limited to small size problems.

These are other search techniques based on "natural phenomena". Under this we are going to discuss two techniques: (1) *Simulated annealing*, which is based on changes in the properties of metals and alloys due to heating them to higher temperature and then slowly decreasing their temperature; and (2) Something based on the Darwin's theory of evolution, called *genetic algorithms*.

9.7 Simulated Annealing

Annealing is process of treatment of metal or alloy by heating to a predetermined temperature, holding for a certain time, and then cooling to room temperature to increase ductility and reduce brittleness. The process of annealing is carried out intermittently during the working of a piece of a metal to restore ductility lost through repeated hammering or other working. Annealing is also done for relief of internal stresses. The annealing temperature varies with metals and alloys, and with properties desired, but must be done within a range, that prevents the growth of crystals. It is an optimization algorithm, its strength is that it avoids getting caught at local maxima—the solutions that are better than nearby, but not best.

Simulated Annealing (SA) is a probabilistic search for the global optimization of a problem for locating a good approximation to the global optimum of a given function, in a large search space. The name of the process and its inspiration come from *annealing* in metallurgical processes, where a function $E(S)$ needs to be minimized. This function is analogous to the internal energy of the system in that state. The goal of SA is to bring the system from some arbitrary initial state, to a state having minimum possible energy [7].

Process

The SA makes use of heuristics to reach to the goal state, such that at each step, the heuristic considers some neighboring state s' of the current state s, and probabilistically decides of moving the system to move to s' state or staying in s. When the above sequence of steps are repeated, the probabilities ultimately move the system to more stable states at lower energy. Typically, the iterations continue until the system reaches to a state that is good enough for the application, or until a given number of iterations are exhausted.

In SA we make one change from the normal heuristic search; we attempt to minimize the function's value instead of maximizing. So, instead of heuristic function it is called *object function*. This is like a *valley descending* rather than *hill-climbing*. Note that in 8-puzzle, for hill-climbing, we compute inverse of number of tiles out of place with respect to goal to obtain heuristic value. So, if zero tiles are out of place (i.e., at goal), the heuristic function is infinite. This would correspond to object function as zero (or minimum).

The physical substances usually move from higher energy configuration to lower levels, so that the valley descending occurs naturally. But, there is some probability that a transition to higher energy will occur, given by

$$p = e^{\frac{\Delta E}{kT}}, \tag{9.6}$$

where ΔE is positive change in energy level (difference, i.e., current cost–new cost, so ΔE is negative in valley descending), T is temperature in Kelvin absolute temperature scale, and k is *Boltzmann's* constant. As per this property, the probability to a large uphill move will be lower than probability of small move. Also, the probability that a large uphill move will take place, decreases as the temperature (T)

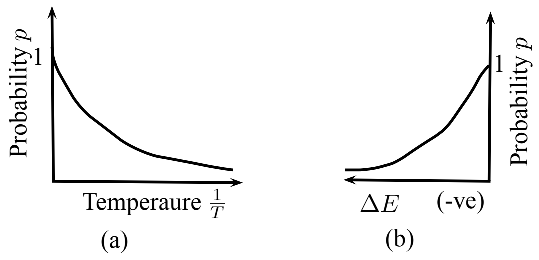


Fig. 9.6 Probability of uphill move in simulated annealing, as a function of temperature and change in energy

decreases. Figure 9.6 shows the effect of increase of temperature and decrease of ΔE, on probability.

In other words, the uphill moves are more possible when temperature is high, but as the temperature decreases, relatively small uphill moves are made until finally process converge to a local minimum configuration.

The rate at which system cools is called *annealing schedule*. If cooling occurs too fast, the physical system will form stable regions high energy. That is, local but not global minima is reached. If a slower schedule is used, a uniform crystalline structure, which corresponds to a global minimum, will develop.

In search techniques, change in E (ΔE) is equal to change in object function. The constant k represents the correspondence between the unit of temperature and unit of energy. Since it is constant, we take probability p in Eq. (9.6) as

$$p' = e^{\frac{\Delta E}{T}}. \tag{9.7}$$

SA mimics annealing process in metallurgy by combination of random search and hill-climbing. During metallurgical annealing, alloys are cooled at a controlled rate to allow for the formation of larger crystals. Larger crystals are chemically at a lower energy state than smaller ones; alloys made of crystals in the lowest energy state are comparably stronger and more rigid. At a high temperature, the search is a random walk, and as the temperature lowers the search gradually transits to a local search. Capturing this idea in an algorithm yields a random process over a space of configurations where the probability of actually moving to a new configuration is determined by the difference in energy and the current temperature. Algorithm 9.4 shows the steps for this process.

We note that SA uses

1. Iterative improvement,
2. Local random search,
3. Exploration, and
4. Greedy search.

When the temperature is high, atoms can move anywhere freely, and have equal probability. When temperature does down, this freedom is reduced.

Algorithm 9.4 Simulated Annealing

```
1: $T$ = high
2: generate random solution
3: calculate energy ($E$) of the solution
4: set initial temperature $T$ (sufficiently high)
5: (Gradually decrease the temperature)
6: while $T >$ cut-off temperature do
7:    test solution $\leftarrow$ solution
8:    for $n$ iterations do
9:       adjust test solution
10:      calculate energy $E$ of test solution
11:      $\Delta E = E_1 - E_2$
12:      if $\Delta E < 0.1$ then
13:         update solution and energy $E$
14:      else if $e^{\frac{\Delta E}{T}} > random(0\ to\ 1)$ then
15:         update solution and $E$
16:      end if
17:      decrease $T$
18:   end for
19: end while
20: end
```

Formal Approach

The Boltzmann probability function tends to return True at higher temperature and False at lower temperature; thus in essence the search gradually shifts from random walk to local hill-climb.

A simulated annealing algorithm is suitable for minimization of an objective function f, having the mapping, $f : \mathbf{S} \rightarrow \mathbb{R}$, where $\mathbf{S}$ is some finite *search space*, and $\mathbb{R}$ is real number. Typically, the search spaces, designated as $\mathbf{S}_n$, comprise sets of bit strings $\{0, 1\}^n$ of fixed length or the set of all possible the permutations over the set $\{1, 2, ..., n\}$.

When considering the search space $\mathbf{S}$, it is necessary to define some notion of *neighborhood* N, which is a relation $N \subseteq \mathbf{S} \times \mathbf{S}$. A function

$$N : \mathbf{S} \rightarrow \mathscr{P}(\mathbf{S}) \tag{9.8}$$

refers to the neighborhood of a search point $s \in \mathbf{S}$, expressed as

$$N(s) = \{s' \in \mathbf{S} \mid (s, s') \in N\}. \tag{9.9}$$

Simulated annealing is considered efficient if it can locate a global maximum of f at sufficiently high probability, and at the same time use fewer number of steps.

SA is a widely used heuristic to NP-complete problems that appear in real life from job-shop scheduling to groundwater remediation design.

Most analysis of search algorithms usually focuses on the worst-case situation. There are relatively few discussions of the average performance of heuristic algorithms, because the analysis is usually more difficult and the nature of the appropriate average to study is not always clear. However, as the size of optimization problems increases, the worst-case analysis of a problem will become increasingly irrelevant, and the average performance of algorithms will dominate the analysis of practical applications. This large number domain of statistical mechanics, and hence of simulated annealing.

9.8 Genetic Algorithms

The Genetic Algorithms (GAs) are search procedures based on the process of *natural selection* and *genetics*. These are increasingly used in applications in difficult search problems, optimization, and machine-learning, across a wide spectrum of human endeavor. A GA processes a finite population of fixed-length binary strings. In practice, the strings are: bit codes, k-ary codes, real (floating-point) codes, permutation (order) codes, etc. Each of these has their place, but here we examine a simple GA to better understand basic mechanics and principles [3].

A simple GA consists of three operators: *selection, crossover, and mutation*. Selection is the survival of the fittest within the GA. Figure 9.7 shows the sequence of operations on a population P_n and producing next population P_{n+1}. To understand the operation of the genetic algorithm, a population of three individuals is taken. Each is assigned a fitness value by the function F, called *fitness function*. On the basis of these fitnesses, the selection phase assigns the first individual (00111) zero copies, the second (111000) two, and the third (01010) one copy. After selection, the genetic operators are applied probabilistically; the first individual has its first bit mutated from a 1 to a 0, and crossover combines the next two individuals into two new ones. The resulting population is shown in the box labeled T_{n+1}. Algorithm 9.5 shows the steps for search using GA.

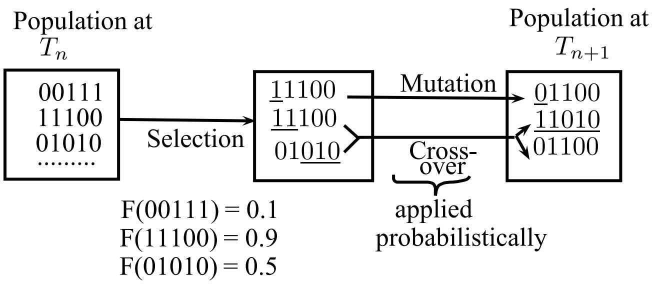


Fig. 9.7 Sequence of operations in GA

Algorithm 9.5 Genetic Algorithm(Input: Initial Population, fitness function, percent for mutation, selection threshhold)

```
1: Initialize the population with random candidate solutions
2: Apply fitness function to Evaluate each candidate's fitness value
3: repeat
4:     Select parents based on fitness value
5:     Recombine pairs of parents (crossover)
6:     Mutate resulting offspring
7:     Apply fitness function to Evaluate new candidates' fitness value
8: until termination condition/goal is reached
```

There are many ways to achieve effective selection, including ranking, tournament, and proportionate schemes, but the key notion is to give preference to better individuals. Consider that individuals are strings of fixed length. In a selection game of two-party of such individuals, pairs of strings are drawn randomly from the parental (original) population, and the better/fitting individuals places an identical copy into the mating pool. When the whole population is selected in this manner, every individual will participate in two tournaments; the best individuals in the population will win both trials. The median individual will typically win one trial and those worst, do not win at all.

For the selection to function, there must be some way of determining who is fitter individual. This evaluation can come directly from the formal objective function, or from the subjective judgment by a human observer. The ordering used is usually partial ordering.

The population holds representations of possible solutions. It usually has a fixed size and is a multi-set. Selection operators usually take whole population into account, i.e., reproductive probabilities are relative to current generation and diversity of a population refers to the number of different fitnesses.

If we were to do nothing but selection, GAs would not be very interesting because the trajectory of populations could contain nothing but changing proportions of strings contained in the original population. In fact, if run repeatedly, selection alone is a fairly expensive way of—with high probability-filling a population with the best structure of the initial population [4].

9.8.1 Exploring Different Structures

To do something more sensible, the GA needs to explore different structures. The main operator used in GAs is *crossover*, which can be one-point or multi-point crossover. A simple one-point crossover is performed using these three steps:

1. two individuals structures are chosen from the population using selection operator, and considered for mating,

2. a crossover site along the string length is chosen uniformly at random, and,
3. the values following the crossover site are exchanged between the two strings.

Let the two strings be, $A = 00000$ and $B = 11111$. If the random choice of a cross site turns up at 2, the two new strings following the crossover will be $A' = 00111$ and $B' = 11000$. These resulting strings are placed in the new population pool, and the process continues pair-by-pair from original population, until the new population is completely filled with "off-springs" constructed from the bits and pieces of good (selected) parents [5].

9.8.2 Process of Innovation in Human

Note that, the *selection* and *crossover* are simple operators, which do the job of: generating random number, copying string, and exchange of partial strings. However, their combined action makes much of the genetic algorithm's search ability. To understand this, we need to think the processing required to be done by human (us) when we innovate. Often we combine the notions that worked well in one context, with those that worked well in another context, to generate possibly better ideas (new notions) of how to attack the problem at hand. In similar way, the GAs juxtapose many different, highly fit substrings (called as notions) through the combined actions of selection and crossover to form new strings (can be called as ideas).

9.8.3 Mutation Operator

If selection and crossover provide much of the innovative capability of a GA, what is the role of the mutation operator? In a binary-coded GA, mutation is the occasional (low-probability) alteration of a bit position, and with other codes a variety of diversity generating operators may be used. By itself, mutation induces a simple random walk through string space. When used with selection alone, the combination form a parallel, noise-tolerant hill-climbing algorithm. When used together with selection and crossover, mutation acts as both insurance policy and as a hill-climber.

9.8.4 GA Applications

The simplest GAs are discrete, nonlinear, stochastic, highly dimensional algorithms operating on problems of infinite varieties. Due to this, GAs are hard to design and analyze [5].

The nature of problems GAs can solve are:

- GAs can solve hard problems quickly and reliably,
- GAs are easy to interface to existing simulations and models,
- GAs are extensible, and
- GAs are easy to hybridize.

Because GAs use very little problem-specific information, they are remarkably easy to connect to extant application code. Many algorithms require high degree of interconnection between the solver and the objective function. For example, dynamic programming requires a stagewise decomposition of the problem that not only limit its applicability, but also can require massive rearrangement of system models and objective functions. GAs on the other hand have clean interface, requiring no more than the ability to propose a solution and receive its evaluation.

Although there are many problems for which the genetic algorithm can evolve a good solution in reasonable time. There are also problems for which they are not suitable, such as problems in which it is important to find the exact global optimum. The domains for which one is likely to choose an adaptive method such as the genetic algorithm are precisely those about which we typically have little analytical knowledge, they are complex, noisy, or dynamic (changing over time). These characteristics make it virtually impossible to predict with certainty how well a particular algorithm will perform on a particular problem, especially if the algorithm is nondeterministic, as is the case with the genetic algorithm. In spite of this difficulty, there are fairly extensive theories about how and why genetic algorithms work in idealized settings.

Example 9.4 4-Queen Puzzle.

Suppose we choose to solve the problem for $N = 4$. This means that the board size is $4^2 = 16$, and the number of queens we can fit inside the board without crossing each other is 4. A configuration of 4 queens can be represented as shown in Fig. 9.8, using 4-digit string made of decimal numbers in the range 1–4. Each digit in a string represents the position of queen in that column. Thus, all queens in the left-to-right diagonal will be represented by a string 1234.

To solve this problem we take initial populations as [1234, 2342, 4312, 3431]. Let us recombine by randomly choosing the crossover after digit position 2. We recombine 1, 2 and 3, 4 members of population, producing [1242, 2334, 4331, 3412]. When this is combined with the original population, we get [1234, 2342, 4312, 3431, 1242, 2334, 4331, 3412]. Next, a random mutation is applied on members 3431 and 2334, changing the third digit gives 3421, 2324. Thus new population is, [1234, 2342, 4312, 3421, 1242, 2324, 4331, 3412].

The fitness of a string is proportional to the inverse of number of queens giving check in each string. for example, in the configuration in Fig. 9.8, total number of checks of $Q_1 \dots Q_4$ are: $2 + 3 + 2 + 1 = 8$. Similarly in configuration 1234, total number of checks are 12.

This is because each queen is crossing the remaining three. These numbers in the remaining seven configurations are: 8, 4, 4, 4, 6, 8, 8. Thus fitness functions of

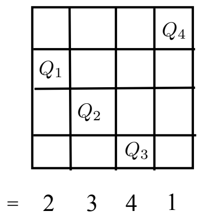


Fig. 9.8 4-Queens' board configuration

above population of elements are $[\frac{1}{12}, \frac{1}{8}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{6}, \frac{1}{8}, \frac{1}{8}]$. Thus, if we need to keep a population of size 4, of more fitter members their fitness functions are $[\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{6}]$. These further combine next time, with population size of 4, having population of [4312, 3421, 1242, 2324]. This sequence can go on until goal is found in one of the configuration. □

9.9 Summary

The domain of combinatorial optimization consists of a set of problems, which requires development of efficient techniques for finding the minimum or maximum of a function having many independent variables. This function is called *cost function* or *objective function*, and represents a quantitative measure of the "goodness" of some complex system. Because these combinatorial class of problems contain many situations of practical importance, *heuristic* methods have been developed, which require computations proportional to only a small polynomial of n, where n is size of the problem. These heuristics cannot be generalized, and are unfortunately problem-specific.

Two basic strategies are common for heuristics: "divide-and-conquer" and "iterative improvement". The first approach divides/splits the problem into subproblems of manageable sizes, then solves each subproblem, and finally the sub-solutions are patched back together, to get the desired solution.

In iterative improvement-based approach, the heuristics starts with the system in a known configuration, and then, a standard rearrangement operation is applied to all parts of the system in turn, until a rearranged configuration that improves the cost function is discovered. The rearranged configuration then becomes the new configuration of the system, and the process is repeated until no further improvements are found.

These methods depend on some heuristics determined by the nature of the problem. The heuristics is defined in the form of a function, say f, which somehow represents the mapping to the total distance between start node and the goal node. For any given node n, the total distance between start and goal node is $f(n)$, such that

$$f(n) = g(n) + h(n), \tag{9.10}$$

where $g(n)$ is distance between start node and the node n, and $h(n)$ is the distance between node n and the goal node.

One of the heuristic methods is *hill-climbing*. The name comes from the fact that to reach the top of a hill, one selects the steepest path at every node, out of the number of alternatives available. A Hill-climbing method is called *greedy local search*.

Among all the heuristic-based problem-solving strategies, informed *best-first* search strategy is one of the most popular methods to exploit heuristic information to cut down the search time. This method assesses exploration along the direction of highest merit, using heuristic information.

Several shortcuts help to simplify the computation of best-first search. First, if the evaluation function $f(n)$ used for node selection always provides optimistic estimates of the final costs of the candidate paths evaluated, then we can terminate the search as soon as the first goal node is selected for expansion, without compromising the optimality of the solution used. This guarantee is called *admissibility*.

We are often able to purge from search tree T, large sets of paths that are recognized at an early stage to be dominated by other paths in T. This becomes possible if the evaluation function f is *order-preserving*, that is, if for any two paths p_1 and p_2, leading from start node S to n, and for any common extension p_3 of those paths, the following holds:

$$f(p_1) \geq f(p_2) \Rightarrow f(p_1 p_3) \geq f(p_2 p_3) \tag{9.11}$$

The most studied version of best-first-search is the algorithm A^*, which provides *additive cost measures*, that is, where the cost of a path is defined as the sum of the costs of its arcs. This algorithm uses ordered state-space search and estimated heuristic to a goal state f^*, like the best-first search.[3] But, f^* is unique in the sense that it can guarantee an optimal path to goal.

Based on values returned from two components: $g^*(n)$ and $h^*(n)$, the evaluation function $f^*(n)$ estimates the quality of a solution path through node n. The one component, i.e., $g^*(n)$, is the minimal cost of a path from start node to node n, and $h^*(n)$ is a lower bound on the minimal cost of a solution path from node n to a goal node.

There are other search techniques based on "natural phenomena". Under this there are two techniques: (1) *Simulated annealing*, which is based on changes in the properties of metals and alloys due to heating them to higher temperature and then slowly decreasing their temperature; and (2) Something based on the Darwin's theory of evolution, called *genetic algorithms*. Annealing is process of treatment of metal or alloy by heating to a predetermined temperature, holding for a certain time, and then cooling to room temperature to improve ductility and reduce brittleness. The process annealing is carried out intermittently during the working of a piece of a metal to restore ductility lost through repeated hammering or other working.

[3] f^* is also known as evaluation function.

Table 9.1 Heuristic search methods

S.No.	Search algorithm	Properties
1.	Best-First Search	It depends on definition of $f(n)$. If $f(n) = h(n)$, then it is likely not optimal, and potentially incomplete. But, A^* is a type of best-first search, which is complete and optimal. It is due to its choice of $f(n)$ that combines $g(n)$ and $h(n)$
2.	Hill-Climbing	It is Non-optimal, Incomplete like DFS, follows heuristics
3.	Beam Search	It is like BFS, expand nodes in $f(n)$ order, and Incomplete for small k (k best nodes from successors are considered for further expansion). But, Complete, and like BFS for k = infinity. It is Non-optimal. When $k = 1$, Beam search is analogous to Hill-Climbing method without backtracking
4.	Branch and Bound	It is Optimal, and $g(n)$ is the cost of path from s to node n and $f(n) = g(n) + 0$
5.	Simulated annealing	Escapes local optima, and is complete and optimal given a long enough cooling schedule

The Genetic Algorithms (GAs) are search procedures based on natural selection and genetics. A GA processes a finite population of fixed-length binary strings. In practice, all these are bit codes, *kry*-codes real (floating-point) codes, permutation (order) codes, etc.

A simple GA consists of three operators: *selection, crossover, and mutation*. Selection is the survival of the fittest within the GA. Each member of population is assigned a fitness value. On the basis of these fitnesses, the selection phase assigned zero or more copies of each individual. After selection, the genetic operators of crossover and mutation are applied probabilistically to the current population to produce new population.

The special features of GAs are as follows:

- GAs can solve hard problems quickly and reliably,
- GAs are easy to interface to existing simulations and models,
- GAs are extensible, and
- GAs are easy to hybridize.

Table 9.1 gives comparison of various heuristic methods:

Exercises

1. Answer the following short review questions.

   a. In what condition the best-first search becomes the breadth-first?
   b. What can you infer from the condition: $f(n) = g(n)$?
   c. What can you infer from the condition: $f(n) = h(n)$?

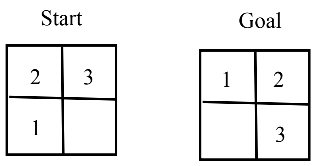


Fig. 9.9 State-space search

d. In what situation the A^* search become best-first search?
e. What is the primary drawback of best-first search?
f. Which search method(s) use the priority-queue data structure?

2. Consider the 3-puzzle problem, where the board is 2×2 and there are three tiles, numbered 1, 2, and 3, and *blank* tile. There are four operators, which move the *blank* tile up, down, left, and right. The start and goal states are given in Fig. 9.9. Show, how the path to the goal can be found using

   a. Breath-first search.
   b. Depth-first search.
   c. A^* search having $g(n)$ equal to number of moves from start state, and $h(n)$ is number of misplaced tiles.

   Assume that there is no possibility to remember states that have been visited earlier. Also, use the given operators in the given order unless the search method defines otherwise. Label each visited node with a number indicating the order in which they are visited. If a search method does not find a solution, explain why this happened.
3. Explain what algorithms or heuristics are suitable for solving constraint satisfaction problems under the following situations. Justify your answers.

   a. The problem is so tightly constrained that it is highly unlikely that solutions exist.
   b. The domain sizes vary significantly: some variables have very large domains (over 1,000 values) and some have very small domains (with fewer than 10 values).
   c. *Eight-Queens Problem*: Arrange eight queens on a chess board in such a manner that none of them can attack any of the others. (Note: A queen will attack another queen if it is crossing other queen while moving horizontally, vertically, or diagonally).
   d. The set of variables and set of domains are handled by a computer say, M. Each constraint is handled by a networked computer, say N. Traffic in the networks is slow. To check a particular constraint, computer M sends a message to computer N through the network, which in turn will send a message back to indicate whether the constraint is satisfied or violated.

4. Suggest a heuristic function for the 8-puzzle that sometimes overestimates, and show how it can lead to a suboptimal solution on a particular case.

5. Prove that, if the heuristic function h never overestimates by more than a constant cost c, then algorithm A^* making use of h returns a solution whose cost exceeds that of the optimal solution by no more than c.
6. Give the name of the algorithms that results from each of the following special cases:
   a. Local beam search with $k = 1$.
   b. Local beam search with $k = \infty$.
   c. Simulated annealing with $T = 0$ at all times.
   d. Genetic algorithm with population size $N = 1$.
7. Explain, how will you use best-first search in each of the following cases? Give the data structure and explain logic.
   a. Speech recognition
   b. PCB design
   c. Routing telephone traffic
   d. Routing Internet traffic
   e. Scene analysis
   f. Mechanical theorem proving
8. What type of data structure is suitable for implementing best-first search, such that each node in the frontier is directly accessible, and all the vertices behind it remain in the order they have been visited.
9. Answer the following in one sentence/one word.
   a. How will you detect during the search of a graph that a particular node has been already visited?
   b. Is the best-first search optimal?
   c. Is the best-first search order-preserving?
   d. Is the best-first search admissible?
10. In the Traveling Salesperson Problem (TSP) one is given a fully connected, weighted, undirected graph and is required to find the Hamiltonian cycle (a cycle visiting all of the nodes in the graph exactly once) that has the least total weight.
    a. Outline how hill-climbing search could be used to solve TSP.
    b. How good results would you expect hill-climbing to attain?
    c. Can other local search algorithms be used to solve TSP?
11. Show that if a heuristic is consistent, then it can never overestimate the cost to reach the goal state. In other words, if a heuristic is monotonic, then it is admissible.
12. Suggest an admissible heuristic that is not consistent.
13. Can GAs have Local maximas? If it is not, how does the GAs tries to avoid it? If yes, justify it.

14. Explain different data structures that can be used to implement the *open* list in BFS, DFS, Best-first search?
15. Find out the worst-case memory requirements for best-first search.
16. If there is no solution, will A^* explore the whole graph? Justify.
17. Define and describe the following terms related to heuristics: Admissibility, monotonicity, informedness.
18. Show that:
    a. The A^* will terminate ultimately.
    b. During the execution of the A^* algorithm, there is always a node in the open-list, that lies on the path to the goal.
    c. If there exits a path to goal, the algorithm A^* will terminate by finding the path of to goal.
    d. If there is no solution in A^*, the algorithm will explore the whole graph.
19. Discuss the ways using which h function in $f(n) = g(n) + h(n)$ can be improved during the search.
20. Why must the A^* algorithm work properly on a graph search, with graph having cycles?
21. For the graph shown in Fig. 9.5, find out whether the A^* search for this graph is
    a. Optimal?
    b. Oder-preserving?
    c. Complete?
    d. Sound?
22. Apply the BFS algorithm for robot path planing in the presence of obstacles for a square matrix of 8×8 given in Fig. 9.10. Write an algorithm to generate the frontier paths. Assume that each move of robot in horizontal (H) and vertical (V) covers a unit distance, and the robot can take only the H and V moves. The start and goal nodes are marked as S and G. Shaded tiles indicate obstacles, i.e., robot cannot pass through these.
23. Redesign the problem of robot shown in Fig. 9.10 for A^* search. Assume that value of h is number of squares equal to $V - i + H - j$, where V and H are both 8.

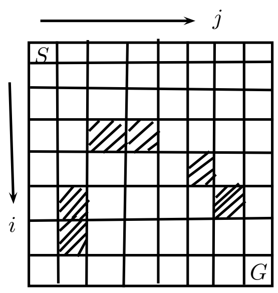


Fig. 9.10 8×8 tiles, with obstacles in shades

24. Solve the 8-puzzle manually for 20-steps, where heuristic is number of tiles out of place with respect to goal state. Assume that $f^*(n) = g^*(n) + h^*(n) = 0 + h^*(n) = h^*(n)$, so that only the heuristics is deciding factor for next node. Note that algorithm shall be DFS. In case of ties, give preference to those nodes which are to the left of the search tree.
25. Find an appropriate state-space representation for the following problems. Also, suggest suitable heuristics for each.
    a. Cryptarithmetic problems (e.g., TWO + TWO = FOUR)
    b. Towers of Hanoi
    c. Farmer, Fox, Goose, and Grain.
26. Suggest appropriate heuristics for each of the following problems:
    a. Theorem proving using resolution method
    b. Blocks world
27. If $P =$ "heuristic is consistent", and $Q =$ "heuristic is admissible". Then show that $P \Rightarrow Q$. Demonstrate by counter example that $Q \not\Rightarrow Q$.
28. Consider the magic-puzzle shown in Fig. 9.11. Suggest the formalism for searching the goal state when started from the start state. (Note that in the goal state all the rows, columns, and diagonals have equal sums equal to 15).
29. Make use of GA to solve 4-puzzle (Fig. 9.12). A move consists, sliding of either of tiles 1 or 2, or 3 into the blank tile. Such a movement creates blank tile at a different position, and the process is repeated until goal state is reached. The solution requires not only reaching to the goal state, but also finds the trace path to reach the goal. Construct a suitable fitness function to implement search by GA, the search should consider only those members of the population which correspond to valid moves.
30. For the graph shown in Fig. 9.13, make use of DFS and certain depth cut-off to backtrack the search from that cut-off.
31. Use best-first search for Fig. 9.14 to find out if the search from start node A to goal node G is

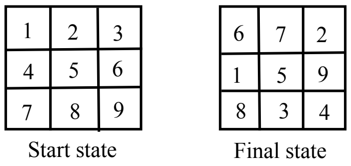


Fig. 9.11 Magic-puzzle

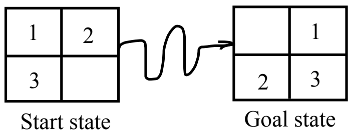


Fig. 9.12 4-puzzle

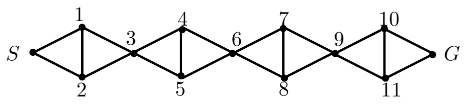


Fig. 9.13 A graph with start node S goal G

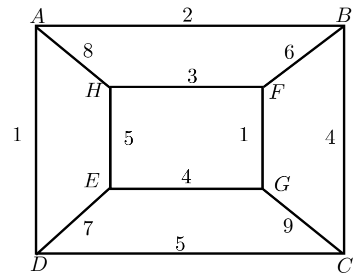


Fig. 9.14 Graph with start node A and goal node G

a. Oder-preserving
b. Admissible
c. Optimal

32. How you will apply the simulated annealing in the following scenarios? For each case, give the problem formation so as to compute the ΔE, temperature T, the states s, s'; the state space $\mathbf{S}$, and object function $f : \mathbf{S} \rightarrow \mathbb{R}$, and perform 5–10 iterations steps manually.

    a. 8-puzzle
    b. 8-Queen problem
    c. Tic-tac-toe problem

33. One fine morning you find that your laptop is not booting. There can be enumerable reasons for this. Assume that you are expert in installation and maintenance of laptops. Represent the search process for trouble-shooting the laptop by constructing a search tree.

    a. Suggest, what search method you consider as most appropriate for this? Also, explain the justification of the particular method you have chosen?
    b. What heuristics you would like to suggest for making the search efficient?
    c. What are the characteristics of this search? Comment for admissibility, monotonocity, and completeness of this solution.

34. Assume a population of 10 members, and apply GA to find out the solution by performing five cycles of iterations, each having selection, mutation, and crossover. Verify that we are far closer to the solution than we were in the begin after performing these iterations. Represent the members as bit strings $\{0, 1\}^n$ for some integer n. Also, fix up some criteria for fitness function, as well as the probability of mutation, and point of crossover.

a. 8-puzzle
b. square-root of a number
c. Factors of an integer

35. What are the consequences of the following special cases of GA-based search?

a. Only the selection operation is performed in each iteration, based on the fitness value.
b. Only the crossover operation is performed in each iteration at a random position.
c. Only the mutation operation is performed in each iteration at a random bit position.

36. Simulated annealing is guided by a changing "temperature" value that determines the likelihood of visiting nodes that appear to be worse than the current node. How does the search behave for very low and very high temperature values, and why it behaves so?
37. Select the best alternatives in each of the following questions.

i. The mutation operation is good for the following:
(a) noise tolerance (b) hill-climbing
(c) random walk (d) all above
ii. The following operation of GA has maximum contribution to search:
(a) mutation (b) selection
(c) crossover (d) fitness function
iii. What operation of GA is responsible for random walk?
(a) mutation (b) crossover
(c) none above (d) both a and b
iv. GAs are not good for the following purpose:
(a) finding exact global optimum (b) local search
(c) approximate solution (d) global search
v. GAs are good in environments which are :
(a) complex (b) noisy
(c) dynamic (d) all above
vi. GA is always:
(a) **P** (b) nondeterministic
(c) **NP** (d) deterministic

References

1. Bagchi A, Mahanti A (1985) Three approaches to heuristic search in networks. J ACM 32: I:1–27
2. Dechter R, Pearl J (1985) Generalized Best-First Search strategies and the optimality of A^*. J ACM 32(3):505–536
3. Forrest S (1993) Genetic algorithms: principles of natural selection applied to computation. Science 261:872–878

4. Goldberg DE (1989) Genetic algorithms in search, optimization and machine learning. Addison-Wesley, Reading
5. Goldberg DE (1994) Genetic and evolutionary algorithms come of age. Commun ACM 37(3):113–119
6. Hart PE et al (1968) A formal basis for the heuristic determination of minimum cost paths. IEEE Trans Syst Sci Cybern 100–107
7. Kirkpatrick S et al (1983) Optimization by simulated annealing. Science 220(4598):671–680
8. Korf RE et al (2005) Frontier search. J ACM 52(5):715–748