Heuristic Search

Heuristic Search is a very general problem solving method in artificial intelligence. The problems we will be concerned with, require a sequence of steps to solve but the correct sequence is not known a priori and must be determined by a trial and error exploration of alternatives.

In this chapter, a framework for describing search methods is provided and several general purpose search techniques are discussed. These methods are all varieties of heuristic search. They can be described independently of any particular task or problem domain. But when applied to particular problems, their efficacy is highly dependent on the way they exploit domain specific knowledge since in and of themselves they are unable to overcome the combinatorial explosion to which search processes are so vulnerable. These techniques continue to provide the framework into which domain specific knowledge can be placed, either by hand or as a result of automatic learning. We have already discussed two very basic search strategies: DFS and BFS. Here will see some others.

Heuristic Function

Heuristic is a technique which makes our search algorithm more efficient. Some heuristics help to guide asearch process without sacrificing any claim to completeness and some sacrificing it. Heuristic is aproblem specific knowledge that decreases expected search efforts. It is a technique which sometimesworks but not always. Heuristic search algorithm uses information about the problem to help directing thepath through the search space. These searches uses some functions that estimate the cost from the currentstate to the goal presuming that such function is efficient. A heuristic function is a function that maps from problem state descriptions to measure of desirability usually represented as number. The purpose of heuristic function is to guide the search process in the most profitable directions by suggesting which pathto follow first when more than one is available. Which aspects of the problem state are considered, how those aspects are evaluated, and the weights given to individual aspects are chosen in such a way that the value of the heuristic function at a given node in the search process gives as good an estimate as possible of whether that node is on the desired path to a solution.

Generally heuristic incorporates domain knowledge to improve efficiency over blind search. In AIheuristic has a general meaning and also a more specialized technical meaning.

Generally a term heuristicis used for any advice that is effective but is not guaranteed to work in every case. For example in case of travelling sales man (TSP) problem we are using a heuristic to calculate the nearest neighbour. Heuristicis a method that provides a better guess about the correct choice to make at any junction that would be achieved by random guessing. This technique is useful in solving though problems which could not be solved in any other way.

Well-designed heuristic functions can play an important role in efficiently guiding a search process toward a solution. Sometimes very simple heuristic functions can provide a fairly good estimate of whether a path is any good or not. In other situations, more complex heuristic functions should be employed. Sometimes a high value of the heuristic function indicates a relatively good position (as shown for chess and tic-tac- toe), while at other times a low value indicates an advantageous situation (as shown for the traveling salesman). It does not matter in general which way the function is stated. The program that uses the values of the function can attempt to minimize it or to maximize it as appropriate. Some simple heuristic functions are:

Chess – the material advantage of our side over the opponent.

Traveling Salesman – the sum of the distances so far.

Tic-Tac-Toe - 1 for each row in which we could win and in which we already have one piece plus 2 for each such row in which we have two pieces

The more accurately the heuristic function estimates the true merits of each node in the search tree (or graph), the more direct the solution process. In the extreme, the heuristic function would be so good that essentially no search would he required. The system would move directly to a solution. But for many problems, the cost of computing the value of such a function would outweigh the effort saved in the search process. In general there is a trade-off between the cost of evaluating a heuristic function and the savings in search time that the function provides.

Generate and test

Algorithm:

1. Generate a possible solution. For some problems, this means generating a particular point in the problem space. For others, it means generating a path from a start state.

2. Test to see if this is actually a solution by comparing the chosen point or the endpoint of the chosen path to the set of acceptable goal states.
3. If a solution has been found, quit Otherwise, return to step l.

If the generation of possible solutions is done systematically, then this procedure will find a solution eventually, if one exists. Unfortunately, if the problem space is very large, "eventually" may be a very long time. The generate-and-test algorithm is a depth-first search procedure since complete solutions must be generated before they can be tested. In its most systematic form, it is simply an exhaustive search of the problem space. Generate-and-test can, of course, also operate by generating solutions randomly but then there is no guarantee that a solution will ever be found.
The most straightforward way to implement systematic generate-and-test is as a depth-first search tree with backtracking. If some intermediate states are likely to appear often in the tree, however, it may be better to modify that procedure, to traverse a graph rather than a tree.

Best-First search

Best first search is an instance of graph search algorithm in which a node is selected for expansion basedon evaluation function f(n). Traditionally, the node which is the lowest evaluation is selected for the explanation because the evaluation measures distance to the goal. This search algorithm serves as combination of depth first and breadth firstsearch algorithm. Best first search algorithm is often referred greedy algorithm this is because they quickly attack the most desirable path as soon as its heuristic weight becomes the most desirable.

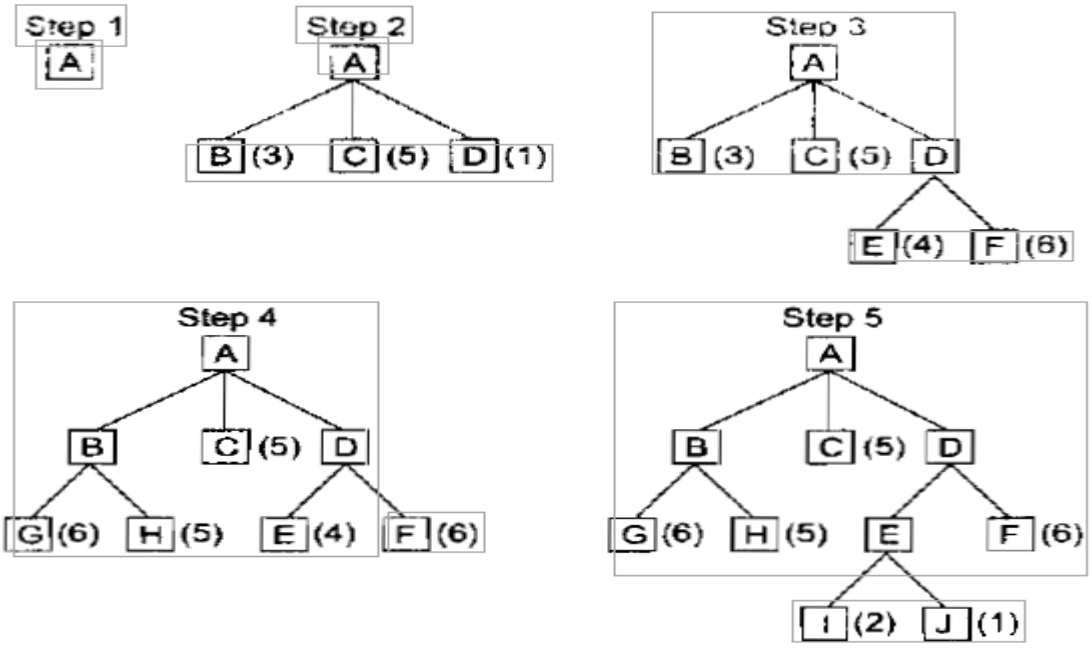
OR Graphs

Depth-first search is good because it allows a solution to be found without all competing branches having to be expanded. Breadth-first search is good because it does not get trapped on dead-end paths. One way of combining the two is to follow a single path at a time, but switch paths whenever some competing path looks more promising than the current one.

At each step of the best-first search process, we select the most promising of the nodes we have generated so far. This is done by applying an appropriate heuristic function to each of them. We then expand the chosen node by using the rules to generate its successors. If one of them is a solution, we can quit. If not, all those new nodes are added to the set of nodes

generated so far. Again the most promising node is selected and the process continues. Usually what happens is that a bit of depth-first searching occurs as the most promising branch is explored. But eventually, if a solution is not found, that branch will start to look less promising than one of the top-level branches that had been ignored. At that point, the now more promising, previously ignored branch will be explored. But the old branch is not forgotten. Its last node remains in the set of generated but unexpanded nodes. The search can return to it whenever all the others get bad enough that it is again the most promising path.

Figure below shows the beginning of a best-first search **procedure**. Initially, there is only one node, so It will be expanded. Doing so generates three new nodes. The heuristic function, which in this example is an estimate of the cost of getting to a solution from a given node, is applied to each of these new nodes. Since node D is the most promising, it is expanded next, producing two successor nodes, E and F. But then the heuristic function is applied to them. Now another path, that going through node B, look, more promising, so it is pursued, generating nodes G and H. But again when these new nodes are evaluated they look less promising than another path, so attention is returned to the path through D to E. E is then expanded, yielding nodes I and J. At the next step, J will be expanded, since it is the most promising. This process can continue until a solution is found.



In best-first search, one move is selected, but the others are kept around so that they can be revisited later if the selected path becomes less promising. Further, the best available state is selected in best-first search even if that state has a value that is lower than the value of the

state that was just explored. The example shown above illustrates a best-first search of a tree. But it is sometimes important to search a graph instead so that duplicate paths will not be pursued. An algorithm to do this will operate by searching a directed graph in which each node represents a point in the problem space. Each node will contain, in addition to a description of the problem state it represents, an indication of how promising it is, a parent link that points back to the best node from which it came, and a list of the node, that were generated from it. The parent link will make it possible to recover the path to the goal once the goal is found. The list of successors will make it possible, if a better path is found to an already existing node to propagate the improvement down to its successors. We call a graph of this sort an **OR graph**, since each of its branches represents an alternative problem-solving path.

To implement such a graph-search procedure, we will need to use two lists of nodes:

OPEN: nodes that have been generated and have had the heuristic function applied to them but which have not yet been examined.

CLOSED: nodes that have already been examined. We need to keep these nodes in memory if we want to search a graph rather than a tree, since whenever a new node is generated, we need to check whether it has been generated before.

We also need a heuristic function that estimates the merits of each node generated. This will enable the algorithm to search more promising paths first. We call this function f ' to indicate that it is an approximation to a function that gives the true evaluation of the node. For many applications, it is convenient to define this function as the sum of two components that we call g and h'. The function g is a measure of the cost of getting from the initial state to the current node. The function h' is an estimate of the additional cost of getting from the current node to a goal state. The combined function f ' then represents an estimate of the cost of getting from the initial state to a goal state along the path that generated the current node. If more than one path generated the node, then the algorithm will record the best one.

Algorithm

1. Start with OPEN containing just the initial state.
2. Until a goal is found or there are no nodes left on OPEN do;

(a) Pick the best node on OPEN.

(b) Generate its successors.

(c) For each successor do:

(i) If it has not been generated before, evaluate it, add it to OPEN, and record its parent.

(ii) If it has been generated before, change the parent if this new path is better than the previous one. In that case, update the cost of getting to this node and to any successors that this node may already have.

Advantage:

- It is more efficient than that of BFS and DFS.
- Time complexity of Best first search is much less than Breadth first search.
- The Best first search allows us to switch between paths by gaining the benefits of both breadth first and depth first search.

Disadvantages:

- Sometimes, it covers more distance than our consideration.
- It is rarely the case that graph traversal algorithms are simple to write correctly.
- It is difficult to guarantee the correctness of such algorithms.

Hill climbing

Hill climbing search algorithm is simply a loop that continuously moves in the direction of increasing value. It stops when it reaches a "peak" where no neighbour has higher value. This algorithm is considered to be one of the simplest procedures for implementing heuristic search. This heuristic combines the advantages of both depth first and breadth first searches into a single method.

The name hill climbing is derived from simulating the situation of a person climbing the hill. The person will try to move forward in the direction of at the top of the hill. His movement stops when it reaches at the peak of hill and no peak has higher value of heuristic function than this. Hill climbing uses knowledge about the local terrain, providing a very useful and effective heuristic for eliminating much of the unproductive search space. It is a branch by a local evaluation function.

Hill climbing is a variant of generate-and-test in which feedback from the test procedure is used to help the generator decide which direction to move in the search space. In a pure generate-and-test procedure, the test function responds

with only a yes or no. But if the test function is augmented with a heuristic function that provides an estimate of how close a given state is to a goal state, the generate procedure can exploit it. This is particularly nice because often the computation of the heuristic function can be done at almost no cost at the same time that the test for a solution is being performed. Hill climbing is often used when a good heuristic function is available for evaluating states but when no other useful knowledge is available. For example, suppose you are in an unfamiliar city without a map and you want to get downtown. You simply aim for the tall buildings. The heuristic function is just distance between the current location and the location of the tall buildings and the desirable states are those in which this distance is minimized.

Algorithm:

Step 1: Evaluate the starting state. If it is a goal state then stop and return success.

Step 2: Else, continue with the starting state as considering it as a current state.

Step 3: Continue step-4 until a solution is found i.e. until there are no new states left to be applied in thecurrent state.

Step 4:

a) Select a state that has not been yet applied to the current state and apply it to produce a new state.

b) Procedure to evaluate a new state.

   i. If the current state is a goal state, then stop and return success.

   ii. If it is better than the current state, then make it current state and proceed further.

   iii. If it is not better than the current state, then continue in the loop until a solution is found.

Step 5: Exit.

Advantages:

- It is helpful to solve pure optimization problems where the objective is to find the best state according to the objective function.
- It requires much less conditions than other search techniques.

Disadvantages:

The question that remains on hill climbing search is whether this hill is the highest hill

possible.Unfortunately without further extensive exploration, this question cannot be answered. This techniqueworks but as it uses local information that's why it can be fooled. The algorithm doesn't maintain a searchtree, so the current node data structure need only record the state and its objective function value. Itassumes that local improvement will lead to global improvement.

There are some reasons by which hill climbing often gets stuck. They are:

Local Maxima: A local maxima is a state that is better than each of its neighbouring states, but not better than some otherstates further away. Generally this state is lower than the global maximum. At this point, one cannotdecide easily to move in which direction! This difficulties can be overcome by the process of backtracking ie backtrack to any of one earlier node position and try to go on a different event direction. To implement this strategy, maintain a list of path almost taken and go back to one of them. If the pathwas taken that leads to a dead end, then go back to one of them.

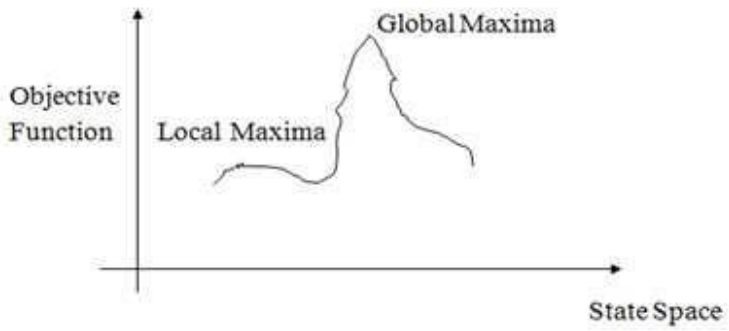
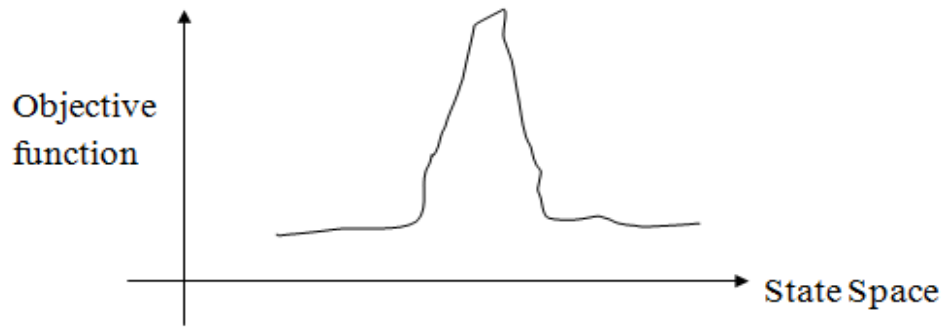


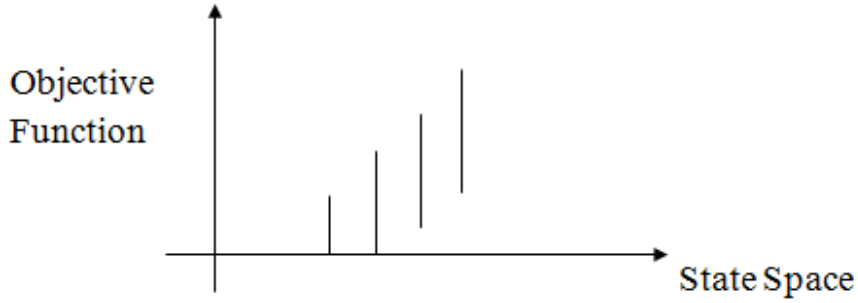
Figure: Local Maxima

Ridges:

It is a special type of local maxima. It is a simply an area of search space. Ridges result in a sequence oflocal maxima that is very difficult to implement. Ridge itself has a slope which is difficult to traverse. Inthis type of situation apply two or more rules before doing the test. This will correspond to move inseveral directions at once.



Plateau: It is a flat area of search space in which the neighbours have same value. So it is very difficult tocalculate the best direction. So to get out of this situation, make a big jump in any direction, which willhelp to move in a new direction. This is the best way to handle the problem like plateau.



i

Steepest-Ascent Hill climbing

A useful variation on simple hill climbing considers all the moves from the current state and selects the best one as the next state. This method is called steepest-ascent hill climbing or gradient search. This contrasts with the basic method in which the first state that is better than the current state is selected. The difference between best-first search and steepest-ascent hill climbing is that it considers all paths from the start node to the end node, whereas steepest ascent hill climbing only remembers one path during the search. The algorithm works as follows.

Algorithm

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the initial state as the current state.
2. Loop until a solution is found or until a complete iteration produces no change to current state:

(a) Let SUCC be a state such that any possible successor of the current state will be better than SUCC.

(b) For each operator that applies to the current state do:

(i) Apply the operator and generate a new state.

(ii) Evaluate the new state. If it is a goal state, then return it and quit. If not, compare it to SUCC. If it is better, then set SUCC to this state. If it is not better, leave SUCC alone.

(c) If the SUCC is better than current state, then set current state to SUCC.

There is a trade-off between the time required to select a move (usually longer for steepest-ascent hill climbing) and the number of moves required to get to a solution (usually longer for basic hill climbing) that must be considered when deciding which method will work better for a particular problem. Both basic and steepest-ascent hill climbing may fail to find a solution. Either algorithm may terminate not by finding a goal state but by getting to a state from which no better states can be generated. This will happen if the program has reached either a local maximum, a plateau, or a ridge as discussed above.

It is not always possible to construct a perfect heuristic function. For example, consider again the problem of driving downtown. The perfect heuristic function would need to have knowledge about one-way and dead-end Streets, which, in the case of a strange city, is not always available. And even if perfect knowledge is available, it may not be computationally tractable to use. As an extreme example, imagine a heuristic function that computes a value for a state by invoking its own problem-solving procedure to look ahead from the state it is given to find a solution. It then knows the exact cost of finding that solution and can return that cost as its value. A heuristic function that does this converts the local hill-climbing procedure into a global method by embedding a global method within it. But the computational advantages of a local method have been lost. Thus it is still true that hill climbing can be very inefficient in a large, rough problem space. But it is often useful when combined with other methods that get it started in the right general neighborhood.

Variable Neighbourhood Descent

Variable Neighborhood Search (VNS) is a recent metaheuristic, or framework for building heuristics, which exploits systematically the idea of neighborhood change, both in the descent to local minima and in the escape from the valleys which contain them. Local search heuristic that explores several neighborhood structures in a deterministic way is called variable neighborhood descent (VND). Its success is based on the simple fact that different neighborhood structures do not usually have the same local minimum. Thus, the local optima trap problem may be resolved by deterministic change of neighborhoods. VND may be seen as a local search routine and therefore could be used within other metaheuristics.

Based on the following observations:

– A local minimum w.r.t. one neighborhood structure is not necessarily locally minimal w.r.t. another neighborhood structure

– A global minimum is locally optimal w.r.t. all neighborhood structures

– For many problems, local minima with respect to one or several neighborhoods are relatively close to each other

Basic principle: change the neighborhood during the search

The first task is to generate the different neighborhood structures: N_1, N_2, ..., N_k

• Find neighborhoods that depend on some parameter – k-Opt (k=2,3,…)

• Some neighborhoods are associated with distance measures so increase the distance

Variable Neighborhood Descent

```
1: input: starting solution, s_0
2: input: neighborhood operators, {N_k}, k = 1, ..., k_max
3: input: evaluation function, f
4: current ⇐ s_0
5: k ⇐ 1;
6: while k ≤ k_max do
7:    s ⇐ the best neighbor in N_k(current)
8:    if f(s) < f(current) then
9:       current ⇐ s
10:      k ⇐ 1
11:   else
12:      k ⇐ k + 1
13:   end if
14: end while
```

The final solution is locally optimal with respect to all neighborhoods: $N_1, N_2, \ldots, N_{k_{max}}$