Module 2

SEARCH STRATEGIES

Search algorithms are one of the most important areas of artificial intelligence. Search is an important component of problem solving in artificial intelligence and, more generally, in computer science, engineering and operations research. Combinatorial optimization, decision analysis, game playing, learning, planning, pattern recognition, robotics and theorem proving are some of the areas in which search algorithms play a key role.

Graphs

A graph G is an ordered pair (V; E) where the elements of V are called *vertices* or *nodes* and the elements of E are called *edges* or *sides*. Each element of E can be thought of an arc joining two vertices in V. Sometimes the arcs may have specific directions in which case the graph is called a *directed graph*. There may be multiple edges joining two given edges. If there is more than one edge joining two vertices, the graph is called a *multigraph.* There may also be loops in graphs, a loop being an edge joining a vertex with itself. In the sequel, unless otherwise explicitly stated, by “graph” we shall always mean an undirected graph without loops and without multiple edges.

Trees

A graph is said to be connected if there is a path along the edges joining any two vertices in the graph. A path which returns to the starting vertex is called a cycle. A tree is a connected graph without cycles.

In search strategies, we generally consider rooted trees where one vertex is designated as the root and all edges are directed away from the root.

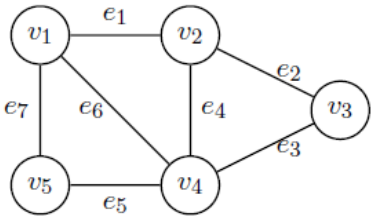
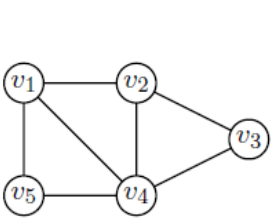
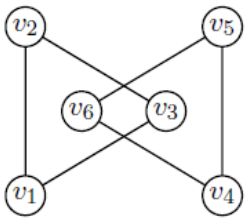


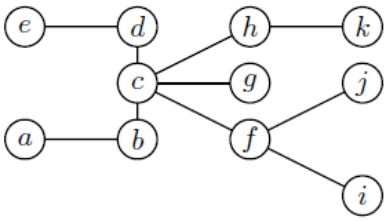
Figure 3.1: A graph without loops and without multiple edges



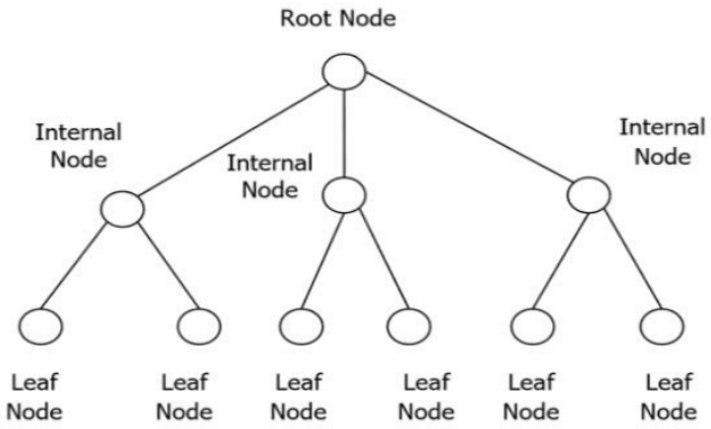
(a) Connected graph



(b) Disconnected graph



(c) Tree



Search algorithm

The process of looking for a sequence of actions that reaches the goal is called *search*. A *search algorithm* takes a problem as input and returns a *solution* in the form of an action sequence. Once a solution is found, the actions it recommends can be carried out. This is called the *execution phase*.

Search tree

A solution is an action sequence, search algorithms work by considering various possible action sequences. The possible action sequences starting at the initial state form a search tree with the initial state at the root; the branches are actions and the nodes correspond to states in the state space of the problem. The root node of the tree corresponds to the initial state.

Search strategy

The first step is to test whether this is a goal state. Then we need to consider taking various actions. We do this by expanding the current state; that is, applying each legal action to the current state, thereby generating a new set of states. Now we must choose which of these possibilities to consider further. A *search strategy* specifies how to choose which state to expand next.

Blind Search (Uninformed Search)

A blind search is a search that has no information about its domain. The only thing that a blind search can do is *distinguish a non-goal state from a goal state*. The blind search algorithms have no domain knowledge of the problem state. *The only information available to blind search algorithms are the state, the successor function, the goal test and the path cost*. Strategies that know whether one non-goal state is “more promising” than another are called informed search or heuristic search strategies.

Blind search is useful in situations where there may not be any information available to us. We might just be looking for an answer and won't know we have found it until we see it.

Breadth-first search

Breadth-first search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.

Algorithm

The algorithm returns the goal state or failure.

1. Create a variable called NODE-LIST and set it to the initial state.

2. Until a goal state is found or NODE-LIST is empty do:

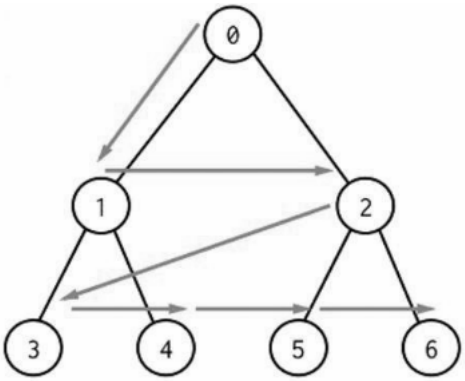
    (a) Remove the first element from NODE-LIST and call it E. If NODE-LIST was empty, quit and return failure.

    (b) For each way that each rule can match the state described in E do:

        i. Apply the rule to generate a new state.

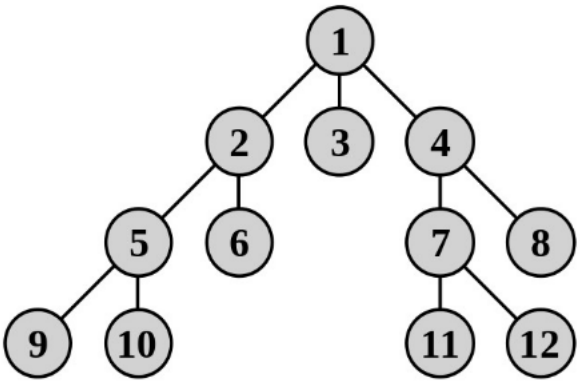
        ii. If the new state is a goal state, quit and return this state.

        iii. Otherwise, add the new state to the end of NODE-LIST.



Example 1

Let the search tree be as in Figure, the initial and goal states being 1 and 8 respectively. Apply the breadth-first search algorithm to find the goal state.



Solution

The details of the various steps in finding a solution to the problem are depicted in Table. It can be seen that the nodes are visited in the following order:

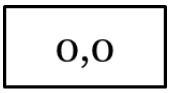
1 → 2 → 3 → 4 → 5 → 6 → 7 → 8

Step no.	*NODE-LIST*	Node E	Node generated from E	Comment
1	1	-	-	Initial state
2	-	1	-	-
3	-	1	2	-
4	2	1	-	-
5	2	1	3	-
6	2, 3	1	-	-
7	2, 3	1	4	-
8	2, 3, 4	1	-	-
9	3, 4,	2	-	-
10	3, 4	2	5	-
11	3, 4, 5	2	-	-
12	3, 4, 5	2	6	-
13	3, 4, 5, 6	2	-	-
13	4, 5, 6	3	-	No node generated from E
14	5, 6	4	-	-
15	5, 6	4	7	-
16	5, 6, 7	4	-	-
17	5, 6, 7	4	8	Goal state. Return 8 and quit.

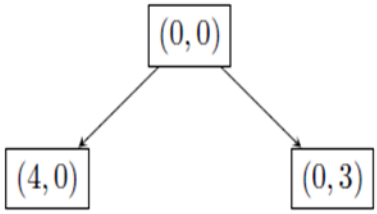
Example 2

Consider the water jug problem and the associated. We now construct the search tree using the breadth-first search algorithm.

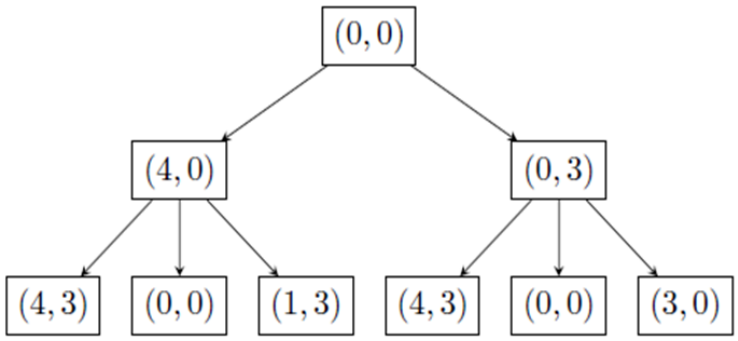
Stage1. Construct a tree with the initial state (0; 0) as its root.



Stage 2. Construct all the children of the root by applying each of the applicable rules to the initial state.



Stage 3. Now for each leaf node in Figure 3.9, generate all its successors by applying all the rules that are appropriate.



Stage 4. The process is continued until we get a node representing the goal state, namely, (2; 0).

Depth-first search

Algorithm

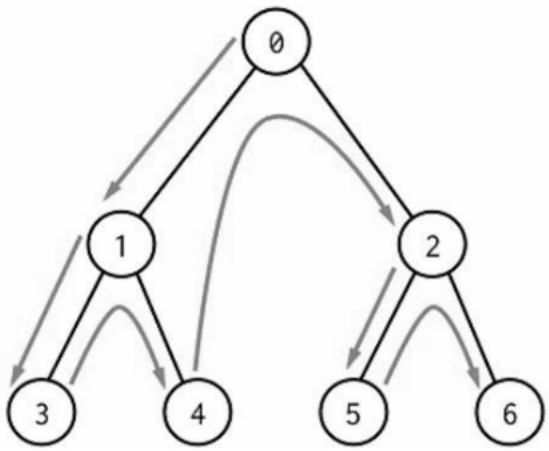
1. If the initial state is a goal state, quit and return success.
2. Otherwise, do the following until success or failure is signaled:

   (a) Generate a successor E of the initial state. If there are no more successors, signal failure.

   (b) Call depth-first search with E as the initial state.

   (c) If success is returned, signal success. Otherwise continue in this loop.

Example 1



The order in which the nodes are explored while applying the DFS algorithm:

0 → 1→ 3 → 4 → 2 → 5 → 6

Example 2

We denote by DFS(x) an invocation of the depth-first search algorithm with x as the

initial vertex. Note that the various nodes are visited in the following order:

1 → 2 → 3 → 4 → 5 → 6 → 7

Call DFS(1)

1. Node 1 is not a goal state.

2. Do until successor or failure is signaled.

   (a) Choose 2 as successor of node 1: E = 2.

   (b) **Call DFS(2).**

      1. Node 2 is not a goal state.

      2. Do until success or failure is signaled.

         (a) Choose 3 as successor of 2: E = 3

         (b) **Call DFS(3).**

            1. Node 3 is not a goal state.

            2. Do until successor or failure is signaled.

               (a) Node 3 has no successor. Signal failure.

            DFS(3) ends.

         (a) Choose 4 as successor of 2: E = 4

         (b) **Call DFS(4).**

            1. Node 4 is not a goal state.

            2. Do until successor or failure is signaled.

               (a) Node 4 has no successor. Signal failure.

            DFS(4) ends.

DFS(2) ends.

(a) Choose 5 as successor of node 1: E = 5

(b) Call DFS(5).

1. Node 5 is not a goal state.

2. Do until success or failure is signaled.

(a) Choose 6 as successor of 5: E = 6

(b) Call DFS(6).

1. Node 6 is not a goal state.

2. Do until successor or failure is signaled.

(a) Choose 7 as a successor of 6: E = 7

(b) Call DFS(7).

1. Node 7 is a goal state. Quit and return success.

DFS(7) ends.

DFS(6) ends.

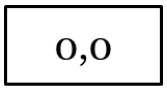
DFS(5) ends.

DFS(1) ends.

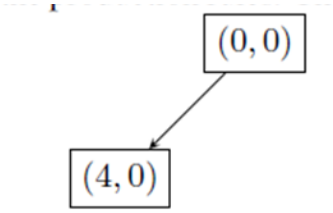
Example 2

Consider the water jug problem and the associated production rules. We now construct the search tree using the depth-first search algorithm.

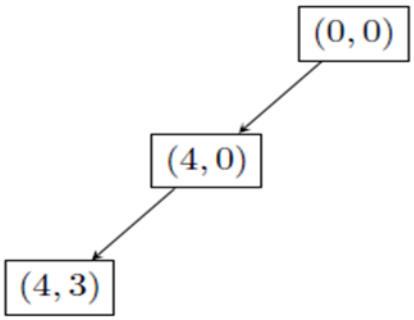
Stage 1. Construct a tree with the initial state (0; 0) as its root. The initial state is not a good state.



Stage 2. We generate a successor E for (0; 0). We choose the successor obtained by the application of Rule 1 in the production rules.



Stage 3. Now for the leaf node in Figure 3.15, generate a successor by applying an applicable rule.



Stage 4. The process is continued until we get a node representing the goal state, namely, (2; 0).

Breadth First Search vs. Depth First Search

Sl No	BFS	DFS
1	Requires more memory because all of the tree that has so far been generated must be stored.	Requires less memory because only the nodes in the current path are stored
2	All parts of the search tree must be examined at level *n* before any node on level *n + 1* can be examined.	May find a solution without examining much of the search space. It stops when one solution is found.
3	Will not get trapped exploring a blind alley.	May follow an unfruitful path for a long time, perhaps forever.
4	If there is a solution, Guaranteed to find a solution if there exists one. If there are multiple solutions then a minimal solution (that is, a solution that takes a minimum number of steps) will be found.	May find a long path to a solution in one part of the tree when a shorter path exists in some other unexplored path of the tree.
5	Breadth-first search uses queue data structure.	Depth-first search uses stack data structure.
6	More suitable for searching vertices which are closer to the given source.	More suitable when there are solutions away from source.

Best First Search

An uninformed search algorithm blindly traverses to the next node in a given manner without considering the cost associated with that step. An informed search, on the other hand, uses an evaluation function *f(n)* to decide which among the various available nodes is the most promising (or "best") before traversing to that node. An algorithm that implements this approach is known as a *best first search algorithm*.

At each step of the best first search process, we select the most promising of the nodes we have generated. 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.

A bit of depth first searching occurs as the most promising branch is explored. 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, 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 node.

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.

Operation of the algorithm

At each step

Select the most promising of the nodes we have generated.

This is done by applying an appropriate heuristic function to each of them.

Expand the chosen node by using the rules to generate its successors.

If one of them is a solution, 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

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:

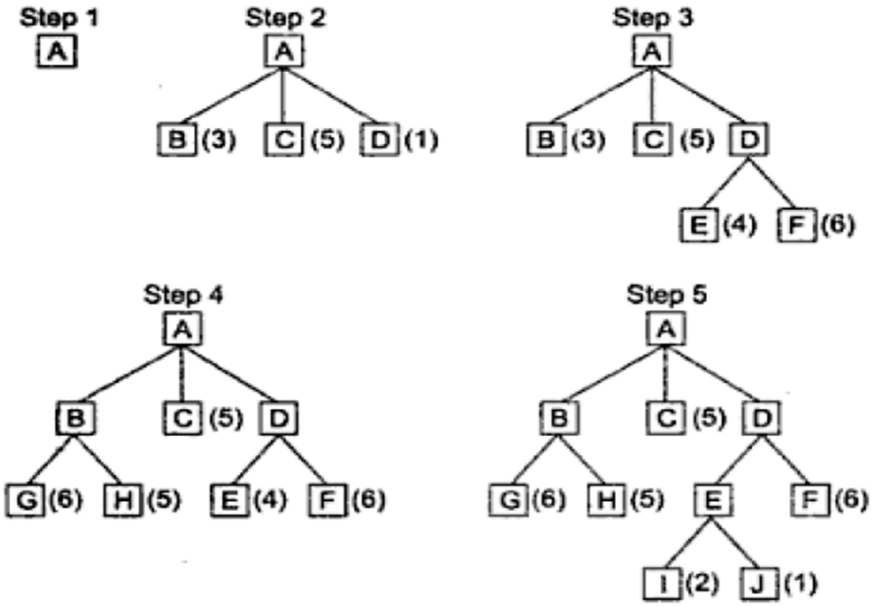
   a) Pick them best node on OPEN

   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



Greedy best first search

The greedy best first search algorithm uses the following notations and conventions:

OPEN: A list which keeps track of the current “immediate” nodes available for traversal.

CLOSED: A list that keeps track of the nodes already traversed.

h(n) : The heuristic function used as the evaluation function f(n), that is, f(n) = h(n).

Algorithm

1. Create two empty lists: OPEN and CLOSED.

2. Start from the initial node (say N) and put it in the ordered OPEN list.

3. Repeat the next steps until goal node is reached.

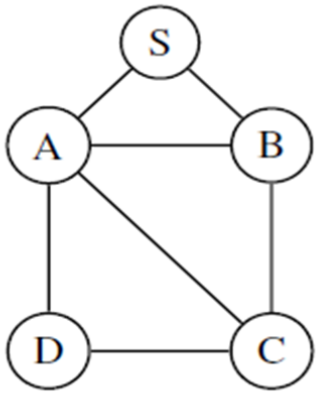
(a) If OPEN list is empty, then exit the loop returning FALSE.

(b) Select the first/top node (say N) in the OPEN list and move it to the CLOSED list. Also record the information of the parent node of N.

(c) If N is a goal node, then move the node to the CLOSED list and exit the loop returning TRUE. The solution can be found by backtracking the path.

(d) If N is not the goal node, generate the immediate successors of N, that is, the immediate next nodes linked to N and add all those to the OPEN list.

(e) Reorder the nodes in the OPEN list in ascending order according to the values of the evaluation function f(n) at the nodes.



Use the following heuristic function:

Node n	S	A	B	C	D
$h(n)$	6	2	3	0	2

From the list of nodes in the CLOSED list at the stage when the goal state has been reached we see that the path from S to C is “S → A → D → C”.

Depth-First Iterative Deepening (DFID) Search

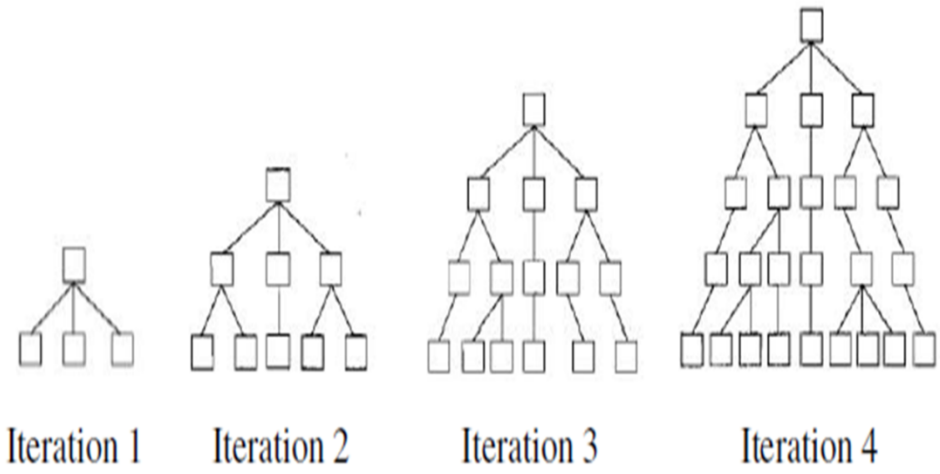
Iterative deepening combines the benefits of depth-first and breadth-first searches. Like breadth-first search, the iterative deepening search is guaranteed to find a solution if one exists. In general, iterative deepening is the preferred uninformed search method when the search space is large and the depth of the solution is not known.

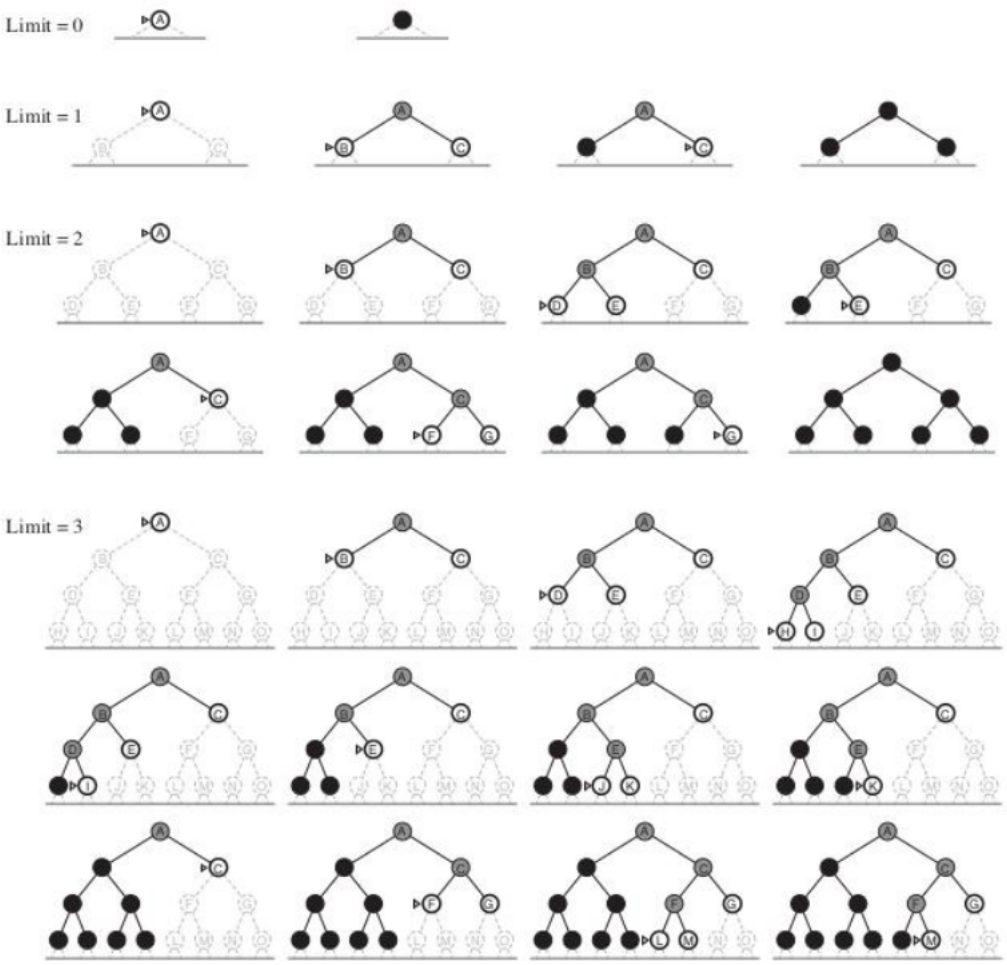
Algorithm

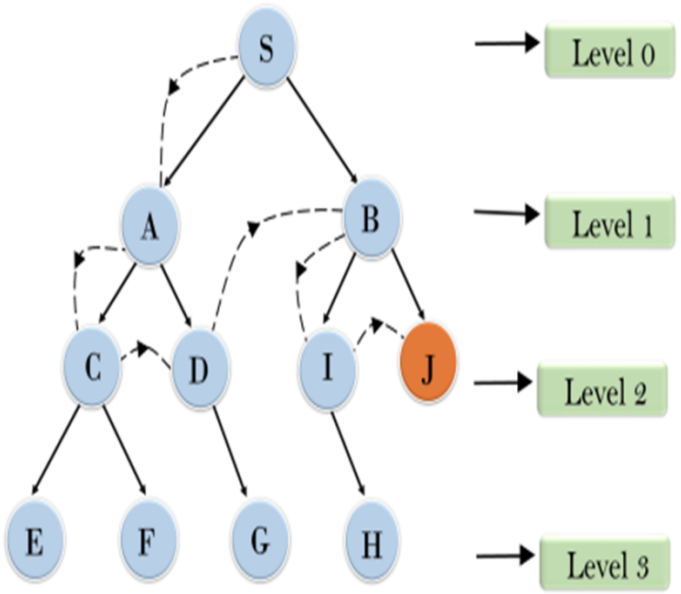
1. Set SEARCH-DEPTH = 1.

2. Conduct a depth-first search to a depth of SEARCH-DEPTH. If a solution is found, then return it.

3. Otherwise, increment SEARCH-DEPTH by 1 and go to step 2.







Levels	Searched Nodes
0	S
1	SAB
2	SACDBIJ
3	SACEFDGBIH

Advantages and disadvantages

1. The DFID will always find the shortest solution path to the goal state if it exists.

2. The maximum amount of memory used is proportional to the number of nodes in the solution path.

3. All iterations but the final one are wasted.

4. The DFID is slower than the depth-first search only by a constant factor.

5. DFID is the optimal algorithm in terms of space and time for uninformed search.

A* Algorithm

Let h(n) be a heuristic function associated with the search problem and g(n) be the cost of traversing the path from the start node to the node n. In the A* best first search, we use the following evaluation function:

f(n) = g(n) + h(n):

Use the following notations

OPEN: Nodes that have been generated, but not yet examined

CLOSED: Nodes that have already been examined

g(n) : The cost of getting from start node to node n

h(n) : The heuristic function

f(n) : g(n) + h(n)

start: The initial node

BESTNODE: Node in OPEN with lowest f value

SUCCESSOR: A successor of BESTNODE

OLD: A node in OPEN/CLOSED

cost(X,Y) : Cost of getting from node X to successor node Y.

Algorithm

1. (a) Set start as the only member of OPEN.

   (b) Set g(start) = 0.

   (c) Set f(start) = h(start).

2. Until the goal node is found, repeat the following procedure:

   (a) If there are no nodes in OPEN report failure.

(b) Otherwise do the following:

i. Pick the node in OPEN with the lowest f value. Call it BESTNODE.

ii. Remove BESTNODE from OPEN.

iii. If BESTNODE is a goal, exit and report a solution.

iv. If BESTNODE is not a goal, generate the successors of BESTNODE. Let SUCCESSOR denote a successor of BESTNODE. For each SUCCESSOR do the following:

A. Set SUCCESSOR to point back to BESTNODE.

B. Set g(SUCCESSOR) equal to g(BESTNODE) + cost(BESTNODE; SUCCESSOR):

C. Examine whether SUCCESSOR is in the list OPEN.

Let SUCCESSOR be in OPEN.

Call that node *OLD*. Delete *SUCCESSOR* from the list of *BESTNODE's* successors and add *OLD*. If *g(OLD)* via its current parent is greater than or equal to *g(SUCCESSOR)* via *BESTNODE* then reset OLD's parent link to point to *BESTNODE*, record new cheaper path in g(OLD) and update *f(OLD).*

Let *SUCCESSOR* be not in OPEN and let *SUCCESSOR* be in *CLOSED*. If so, call the node in *CLOSED* as *OLD* and adds *OLD* to the list of *BESTNODE's* successors. Check to see if the new path or the old path is better (as in) and reset the parent link and g and f. If we have found a better path to old, the improvements should be propagated to OLD's successors.

Let *SUCCESSOR* be not in *OPEN* and not in *CLOSED*. Then put it in *OPEN* and add it the list of *BESTNODE*'s successors. Compute *f(SUCCESSOR) = g(SUCCESSOR) + h(SUCCESSOR):*

Differences between Greedy best first and A*

Sl. No.	Greedy best first	A* best first
1	The greedy best first is not complete, that is, the algorithm may not find the goal always.	If the heuristic function is admissible, the A* best first is complete.
2	In general, the greedy best first may not find the optimal path from the initial to the goal state.	If the heuristic function is admissible, the A* best first is always yields the optimal path from the initial state to goal state.
3	In general, greedy best first uses less memory than A* best first.	In general, A* best first uses more memory than greedy best first.

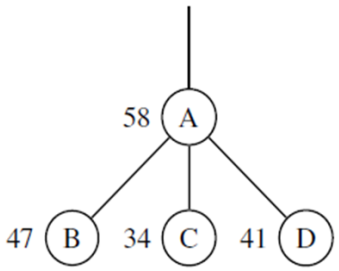
Heuristic Search Strategies

A heuristic1 is a technique designed for solving a problem more quickly when classic (that is, well known) methods are too slow, or for finding an approximate solution when classic methods fail to find any exact solution

The objective of a heuristic is to produce a solution in a reasonable time that is good enough for solving the problem at hand. This solution may not be the best of all the actual solutions to this problem, or it may simply approximate the exact solution. But it is still valuable because finding it does not require a prohibitively long time.

Heuristic function

A heuristic function, also called simply a heuristic, is a function that ranks alternatives in search algorithms at each branching step based on available information to decide which branch to follow.



Definition

The heuristic function for a search problem is a function h(n) defined as follows h(n) = estimated cost of the cheapest path from the state at node n to a goal state.

Note

Unless otherwise specified, it will be assumed that a heuristic function h(n) has the following properties:

1. h(n) ≥ 0 for all nodes n.

2. If n is the goal node, then h(n) = 0.

Admissible heuristic functions

A heuristic function is said to be admissible if it never overestimates the cost of reaching the goal, that is, the cost it estimates to reach the goal is not higher than the lowest possible cost from the current point in the path.

Let *h(n)* be a heuristic function for a search problem. *h(n)* is the estimated cost of reaching the goal from the state *n*. Let h*(n) be the optimal cost to reach a goal from *n*. We say that *h(n)* is admissible if ***h(n) ≤ h*(n)*** **for all *n***

Note

1. There are no limitations on the function h(n). Any function of our choice is acceptable. As an extreme case, the function h(n) defined by h(n) = c for all n where c is some constant can also be taken as a heuristic function. The heuristic function defined by h(n) = 0 for all n is called the ***trivial heuristic function***.

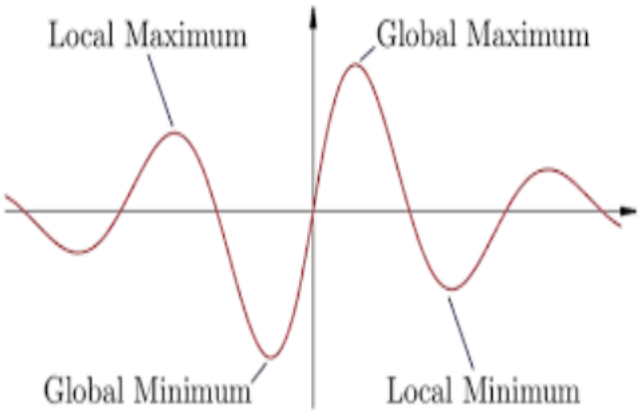
2. The heuristic function for a search problem depends on the problem in the sense that the definition of the function would make use of information that are specific to the circumstances of the problem. Because of this, there is no one method for constructing heuristic functions that are applicable to all problems.

3. However, there is a standard way to construct a heuristic function: It is to find a solution to a simpler problem, which is one with fewer constraints. A problem with fewer constraints is often easier to solve (and sometimes trivial to solve). An optimal solution to the simpler problem cannot have a higher cost than an optimal solution to the full problem because any solution to the full problem is a solution to the simpler problem.

Hill Climbing

Hill climbing is an iterative algorithm that starts with an arbitrary solution to a problem,

then attempts to find a better solution by making an incremental change to the solution. If the change produces a better solution, another incremental change is made to the new solution, and so on until no further improvements can be found. The solution obtained by this method may not be the global optimum; it will only be a local optimum. In this sense, it is a local search method. In the figure, the x-axis denotes the nodes in the state space and the y-axis denotes the values of the objective function corresponding to particular states.



Simple Hill Climbing Algorithm

The word “operator” refers to some applicable action. It usually includes information about the preconditions for applying the action and also about the effect of the action.

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 there are no new operators left to be applied in the current state:

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

b. Evaluate the new state.

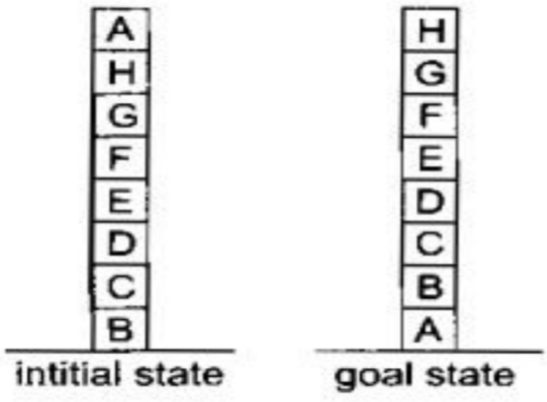
i. If it is a goal state then return it and quit.

ii. If it is not a goal state but if it is better than the current state then make it the current state.

iii. If it is not better than the current state then continue in the loop.

Example

Consider the blocks world problem



Local - Add one point for every block that is resting on the thing it is supposed to be resting on. Subtract one point for every block that is sitting on the wrong thing.

Using the heuristic function, the goal state has a score of 8 and the initial state has a score of 4

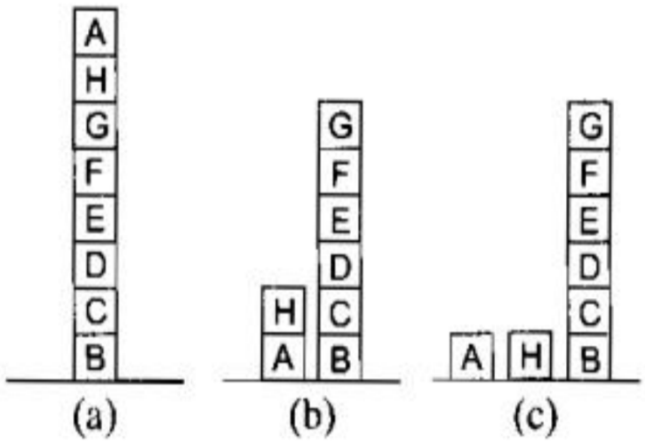
h(initial State) = -1 + 1 + 1 + 1 + 1 + 1 + 1 – 1 = 4

h(Goal state) = +1 + 1 + 1 + 1 + 1 + 1 + 1 + 1= 8

Move block A to the table, then the score becomes 6.

h(State 1) = (-1 + 1 + 1 + 1 + 1 + 1 + 1) + 1 = 6

The hill climbing procedure will accept this move. From the new state, there are three possible moves



h(State 2(a)) = -1 + 1 + 1 + 1 + 1 + 1 + 1 - 1 = 4

h(State 2(b)) = +1 -1 - 1 + 1 + 1 + 1 + 1 + 1 = 4

h(State 2(c)) = +1 - 1 - 1 + 1 + 1 + 1 + 1 + 1 = 4

Hill climbing will halt because all these states have lower scores than the current state. The process has reached a local maximum that is not a global maximum. We have reached such a situation because of the particular choice of the heuristic function. A different choice of heuristic function may not produce such a situation.

Steepest-Ascent Hill Climbing

In the basic hill climbing, the first state that is better than the current state is selected. In steepest-ascent hill climbing, we consider all the moves from the current state and select the best as the next state.

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 SUCC is better than current state, then set current state to SUCC

Advantages and disadvantages of hill climbing

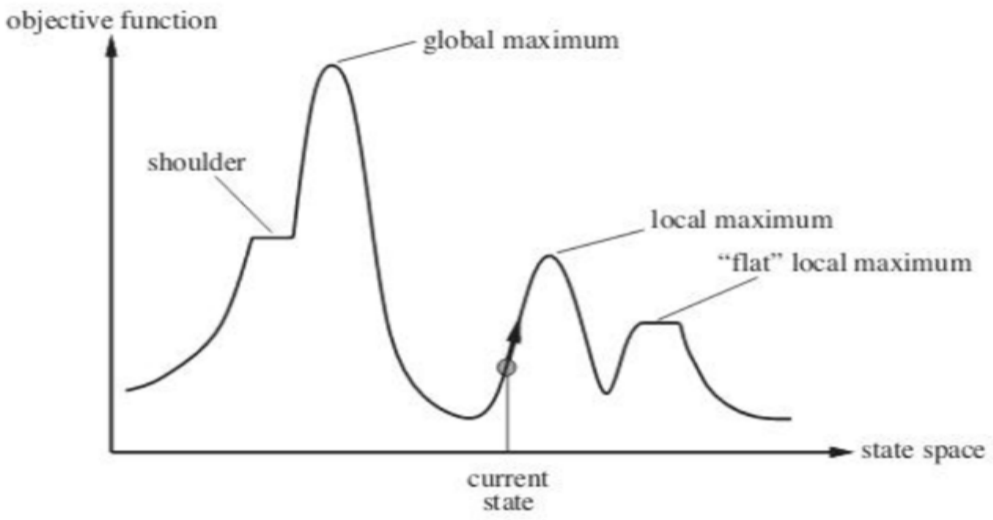
Advantages

- Hill climbing is very useful in routing-related problems like travelling salesmen problem, job scheduling, chip designing, and portfolio management.
- It is good in solving optimization problems while using only limited computation power.
- It is sometimes more efficient than other search algorithms.
- Even though it may not give the optimal solution, it gives decent solutions to computationally challenging problems.

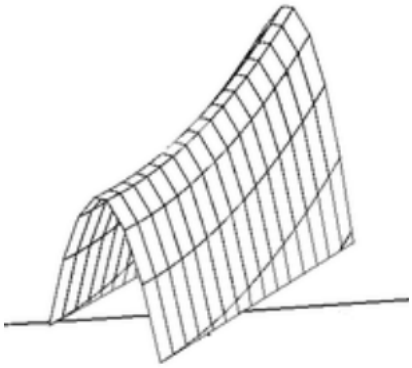
Disadvantages

Both the basic hill climbing and the steepest-ascent hill climbing may fail to produce a solution. Either the algorithm terminates without finding a goal state or getting into a state from which no better state can be generated. This will happen if the program has reached a local maximum, a ridge or a plateau.

1. *Local optima (maxima or minima)*: A local maximum is a peak that is higher than each of its neighbouring states, but lower than the global maximum. A solution to this problem is to backtrack to an earlier solution and try going in a different direction.



2. *Ridges:* A ridge is a sequence of local maxima. Ridges are very difficult to navigate for a hill-climbing algorithm (see Figure 4.17). This problem may be overcome by moving in several directions at once.



3. *Plateaus*: A plateau is an area of the state space where the evaluation function is flat. It can be a flat local maximum, from which no uphill path exists, or a shoulder, from which progress is possible. One solution to handle this situation is to make a big jump in some direction to get to a new part of the search space.

Simulated Annealing

Simulated annealing is a variation of hill climbing. In simulated annealing, some down-hill moves may be made at the beginning of the process. This to explore the whole space at the beginning so that we do not land in a local maximum, or plateau or ridge instead of the global maximum. The algorithm is somewhat analogous to a process known as “annealing” in metallurgy.

Annealing

Annealing is a heat treatment that alters the physical and sometimes chemical properties of a material to increase its ductility and reduce its hardness, making it more workable. It involves heating a material above a certain temperature, maintaining a suitable temperature for an appropriate amount of time and then cooling.

Notations and conventions

- A variable, “BEST-SO-FAR” which takes a problem state as a value.
- A certain function of time denoted by T(t) or simply T. This function is called the annealing schedule.
- AS per standard usage, in the context of the simulated annealing algorithm, the heuristic function will be called an objective function.
- It will be assumed that the problem is a minimization problem

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. Initialize BEST-SO-FAR to the current state.
3. Initialize T according to the annealing schedule.
4. Loop until a solution is found or until there are no new operators left to be applied in the current state.
    a. Select an operator that has not yet been applied to the current state and apply it to produce a new state.
    b. Evaluate the new state. Compute

        ΔE = (value of current state) − (value of new state):

        - If the new state is a goal state, then return it and quit.
        - If it is not a goal state but is better than the current state, then make it the current state. Also set BEST-SO-FAR to this new state.
        - If it is not better than the current state then make it the current state with probability $e^{-\Delta E/T}$.

5. Return BEST-SO-FAR as the answer