Search Algorithms in Artificial Intelligence

Search in AI is the process of find the path from a starting state to a goal state by transitioning through intermediate states

Parameters for search Evaluation:-

- **Completeness:** Algorithm find answer in some finite time. it guarantees to return a solution if at least any solution exists for any random input.
- **Optimality:** If algorithm is guaranteed to be the best solution (lowest path cost) among all other solutions, then such a solution for is said to be an optimal solution.
- **Time Complexity:** Time complexity is a measure of **time** for an algorithm to complete its task.
- **Space Complexity:** It is the maximum storage space required at any point during the search, as the complexity of the problem.

Types of search algorithms

1. Uninformed (Blind search) search
2. Informed search (Heuristic search) algorithms.

1. Uninformed search:

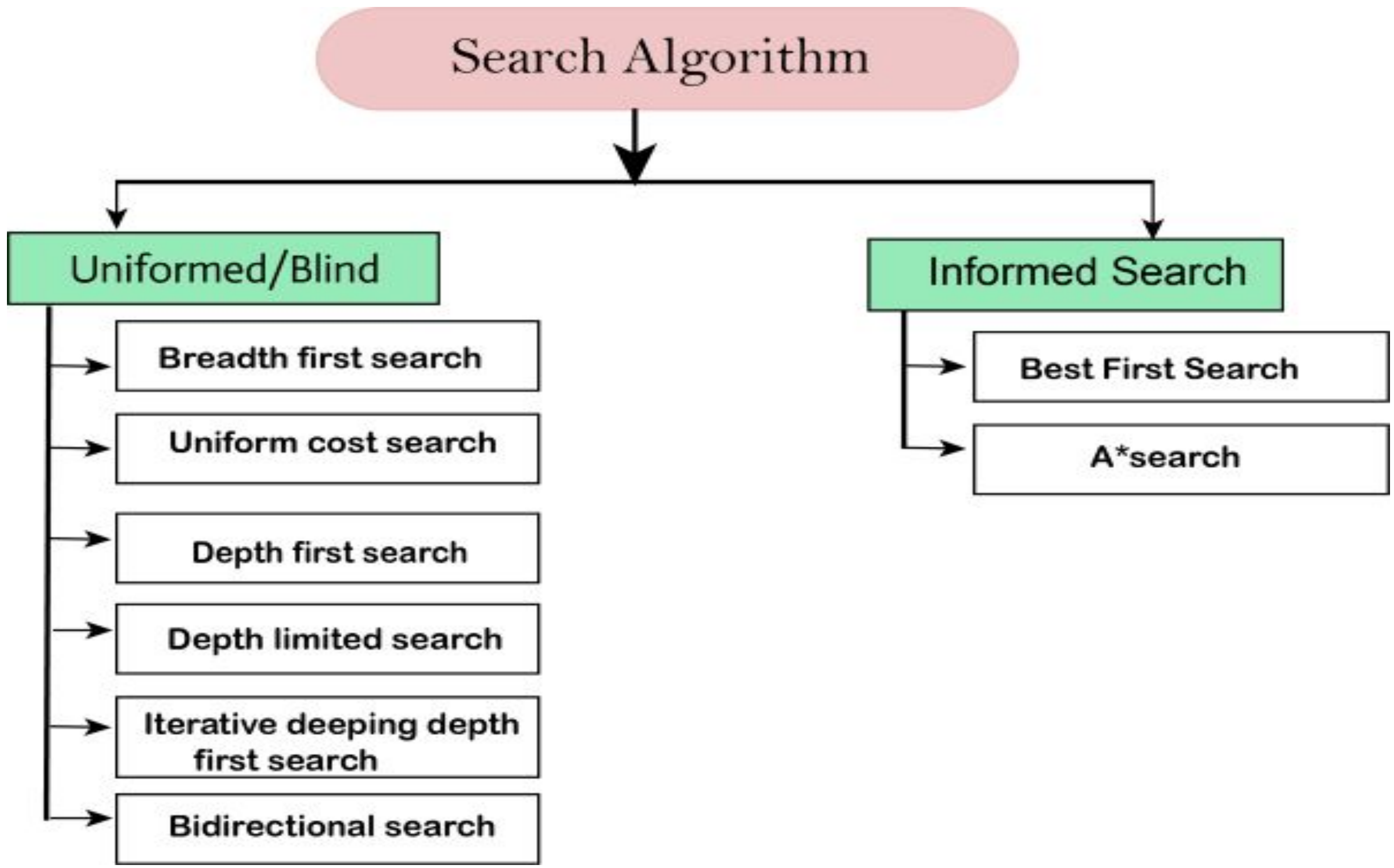
- It does not contain any domain knowledge such as closeness, the location of the goal.
- This search has no additional information about the distance from the current state to the goal so it is also called **blind search.**
- It examines each node of the tree until it achieves the goal node.

Types of search algorithms

2. Informed search:

- It is also called a **Heuristic search** because it search with information
- In an informed search, problem information is available which can guide the search.
- Informed search strategies can find a solution more efficiently than an uninformed search strategy.

Types of search algorithms



Informed search Vs formed search

Basis of comparison	Informed search	Uninformed search
Information	search with information so it called **heuristic search**	search without information so it called **Blind search**
Basic Knowledge	Use knowledge to find step to the solution	No use of Knowledge
Efficiency	Highly Efficient give quick solution	Less efficient Time consuming
Cost	Low	High
Complexity (Time ,space)	Less	High
Algorithms	A*, Heuristic search, , Best first search	depth first search Breadth first search

1.Breadth-first Search:

- It is the most common search strategy for traversing a tree or graph. This algorithm searches breadth wise in a tree or graph, so it is called breadth-first search.
- Breadth-first search implemented using **FIFO queue** data structure.
- This algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level.
- The breadth-first search algorithm is an example of a general-graph search algorithm.

Breadth-first Search

Advantages:

- BFS will provide a solution if any solution exists.
- If there is more than one solution for a given problem, then BFS will provide the minimal solution which requires the least number of steps.

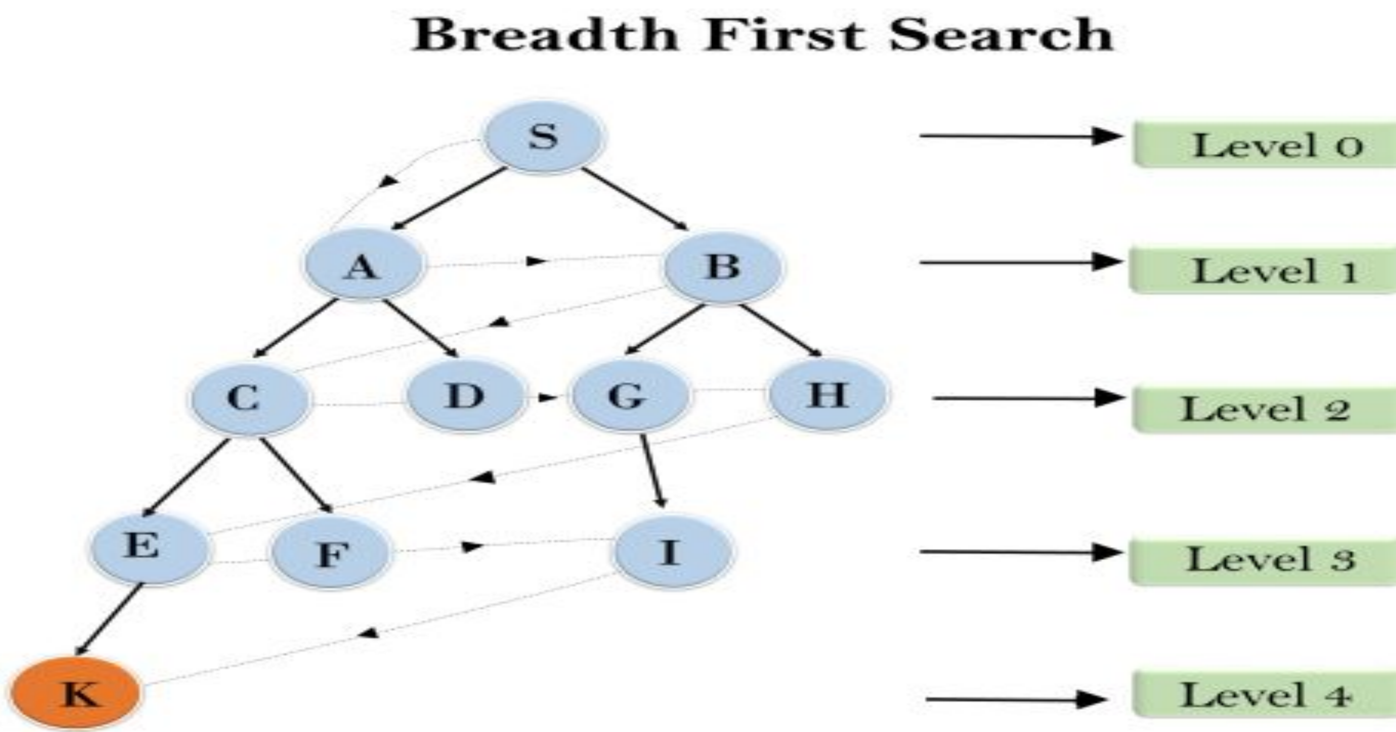
Disadvantages:

- It requires lots of memory since each level of the tree must be saved into memory to expand the next level.
- BFS needs lots of time if the solution is far away from the root node.

Breadth-first Search

Example:

S---> A--->B---->C--->D---->G--->H--->E---->F---->I->K



S-A,B
A-C,D
B-G,H
C-E,F
E-K

Breadth-first Search

Algorithm:-

1.Create a node list (Queue) that initially contains the first node N and mark it as visited.

2. Visit the adjacent unvisited vertex of N and insert it in a queue.

3.If there are no remaining adjacent vertices left, remove the first vertex from the queue mark it as visited, display it,.

4.Repeat step 1 and step 2 until the queue is empty or the desired node is found..

Breadth-first Search

- **Time Complexity:** Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest(nearest) Node.
- Where d= depth of shallowest solution

  b =node at every state.

In DSA: O(V+E)

In AI

- **$T(b) = 1+b^2+b^3+.......+ b^d= O(b^d)$**
- **Space Complexity:** Space complexity is given by the Memory size of frontier which is $O(b^d)$.
- **Completeness:** BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution always.
- **Optimality:** BFS is optimal if path cost is a non-decreasing function of the depth of the node.

DEPTH-FIRST SEARCH

- Depth-first search is a recursive algorithm for traversing a tree or graph data structure.
- It is called the depth-first search because it starts from the root node and follows each path to its greatest depth node before moving to the next path.
- DFS uses a stack data structure implemented in **LIFO** manner.

DEPTH-FIRST SEARCH

Advantage:

- ❖It require very **less memory** as it only needs to store a stack of the nodes on the path from root node to the current node.
- ❖It takes **less time** to reach to the goal node than BFS algorithm (if it traverses in the right path).

Disadvantage:

- There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
- DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

DEPTH-FIRST SEARCH

Algorithm

Step 1: Push the starting node(Root) on the stack .

Step 2: Repeat Steps 3 and 4 until STACK is empty.

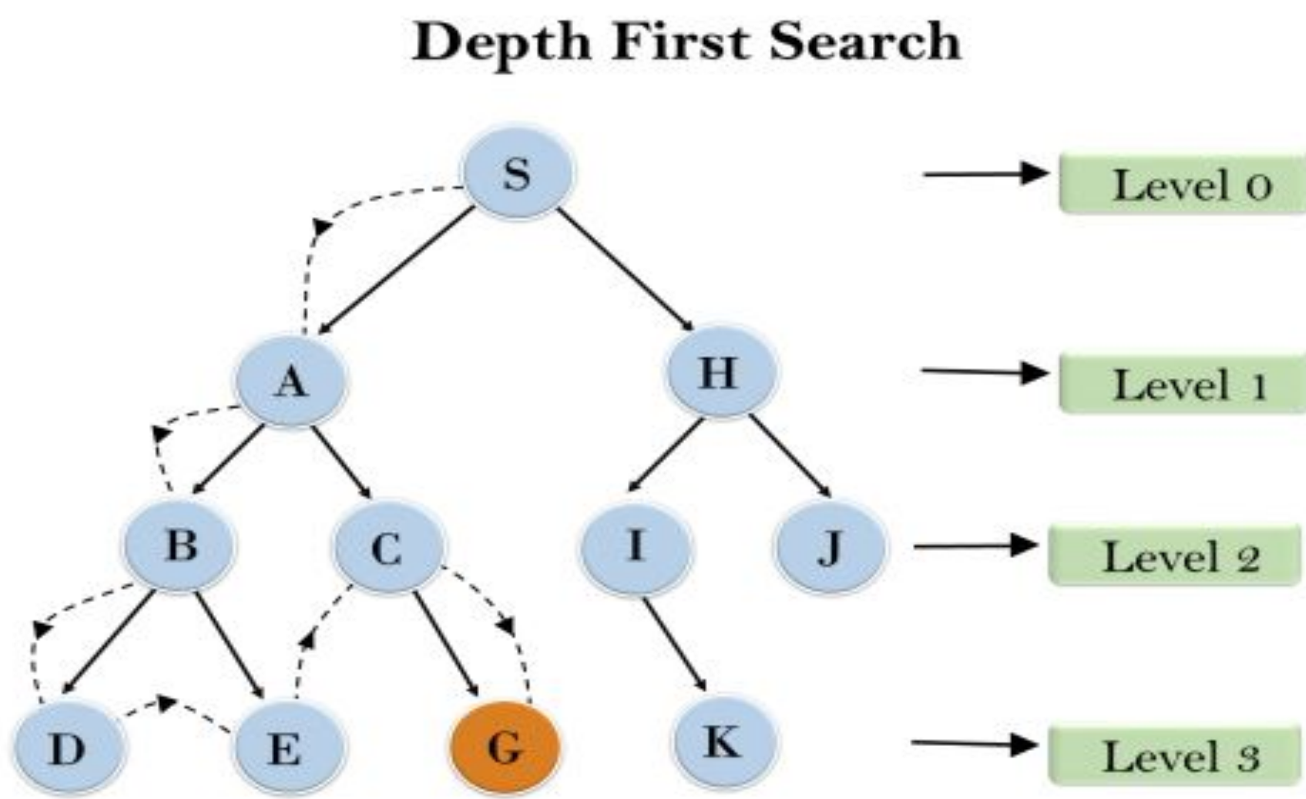
Step 3: Remove(Pop)the top node N.

If node=goal state stop

Step 4: Push all the children of node N in stack.

Step 5 exit

DEPTH-FIRST SEARCH



DEPTH-FIRST SEARCH

- In the above search tree, we have shown the flow of depth-first search, and it will follow the order as:
- Root node--->Left node ----> right node.

- It will start searching from root node S, and traverse A, then B, then D and E, after traversing E, it will backtrack the tree as E has no other successor and still goal node is not found.
- After backtracking it will traverse node C and then G, and here it will terminate as it found goal node.

DEPTH-FIRST SEARCH

- **Completeness:** DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.
- **Time Complexity:** Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by:

$$T(n)= 1+ n^2+ n^3 +.........+ n^m=O(n^m)$$

- Where, m= maximum depth of any node and this can be much larger than d (Shallowest solution depth)
- **Space Complexity:** DFS algorithm needs to store only single path from the root node, hence space complexity of DFS is equivalent to the size of the fringe set, which is O(bm).
- **Optimal:** DFS search algorithm is non-optimal, as it may generate a large number of steps or high cost to reach to the goal node.

Depth-Limited Search Algorithm:

- It is similar to depth-first search with a predetermined limit.
- can solve the drawback of the infinite path in the Depth-first search.
- Depth-limited search can be terminated with two Conditions of failure:

1. **Standard failure value**: It indicates that problem does not have any solution.
2. **Cutoff failure value**: It defines no solution for the problem within a given depth limit.

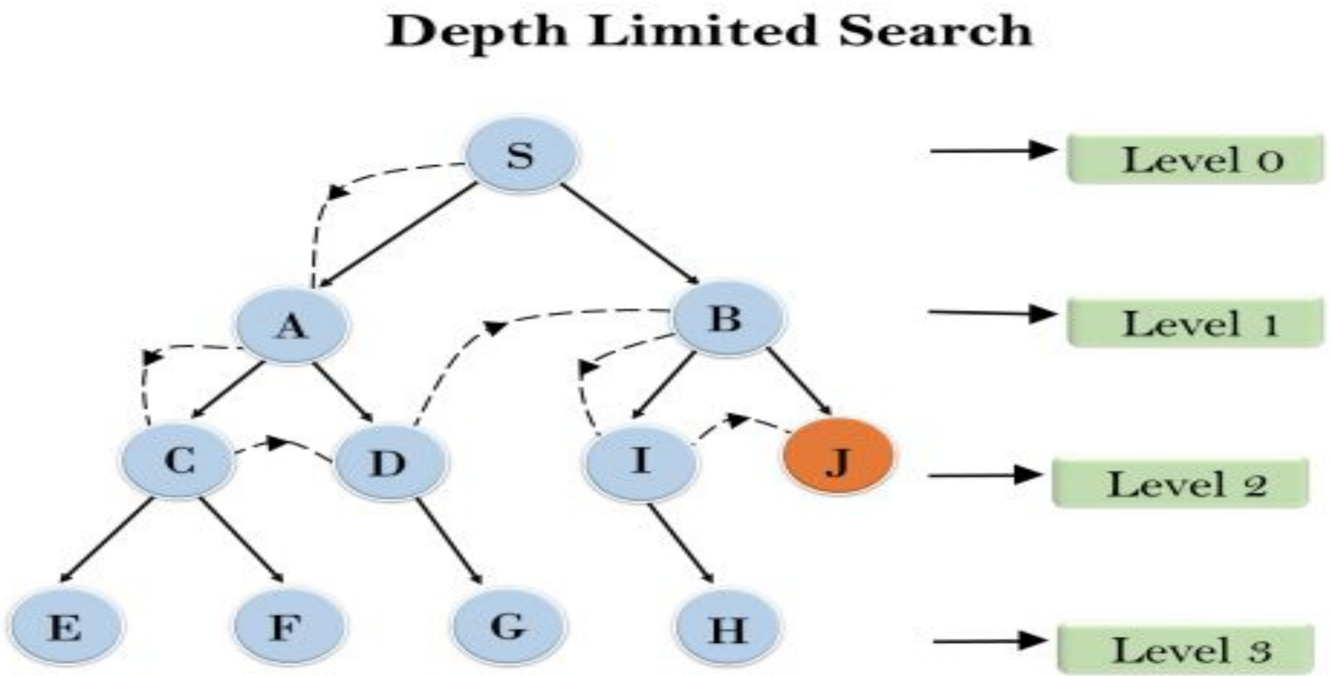
Advantages:

- Depth-limited search is Memory efficient.

Disadvantages:

- It can be terminated without finding solution.
- It may not be optimal if the problem has more than one solution.

Depth-Limited Search Algorithm:



Depth-Limited Search Algorithm:

- **Completeness:** DLS search algorithm is complete if the solution is above the depth-limit.
- **Time Complexity:** Time complexity of DLS algorithm is $\mathbf{O(b^{\ell})}$.
- **Space Complexity:** Space complexity of DLS algorithm is O$\mathbf{(b\times\ell)}$.
- **Optimal:** Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if $\ell>d$.

Iterative deepening depth-first Search:

- It is a combination of DFS and BFS algorithms.
- It performs depth-first search up to a certain "**depth limit**", and it increasing the depth limit after each iteration until the goal node is found.

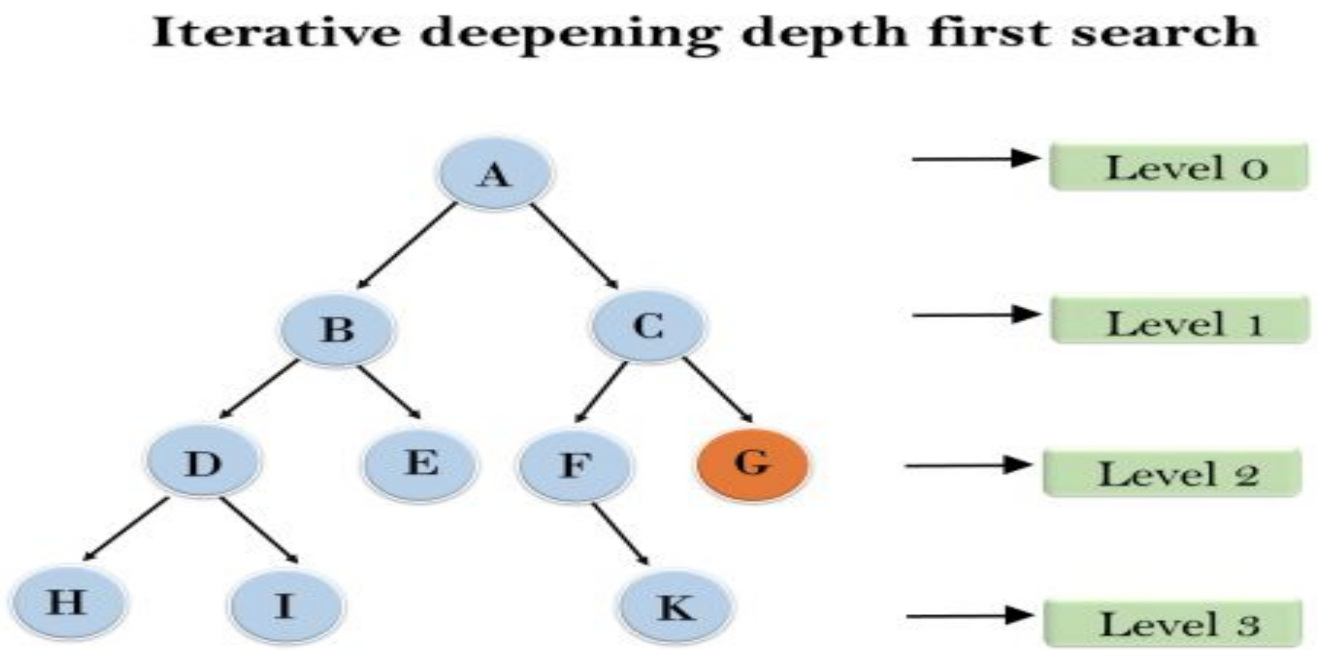
Advantages:

- It combines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.

Disadvantages:

- The main drawback of IDDFS is that it repeats all the work of the previous phase.

Iterative deepening depth-first Search:



Initially depth d=0
1st Iteration : d=0= A
2nd Iteration : d=1=A-B-C
3rd iteration : d=2= A-B-C-D-E-C-F-G

Iterative deepening depth-first Search

- **Completeness:**

This algorithm is complete is if the branching factor is finite.

- **Time Complexity: O(b^d)**.

Where b : is the branching factor

d : is depth

- **Space Complexity: O(bd)**.
- **Optimal:** It will be optimal if path cost is a non- decreasing function of the depth of the node.

Comparison Among DFS, BFS and IDDFS

A comparison table between DFS, BFS and IDDFS

	Time Complexity	Space Complexity	When to Use ?
DFS	$O(b^d)$	$O(d)$	=> Don't care if the answer is closest to the starting vertex/root. => When graph/tree is not very big/infinite.
BFS	$O(b^d)$	$O(b^d)$	=> When space is not an issue => When we do care/want the closest answer to the root.
IDDFS	$O(b^d)$	$O(bd)$	=> You want a BFS, you don't have enough memory, and somewhat slower performance is accepted. In short, you want a BFS + DFS.

Bidirectional Search Algorithm:

- This algorithm runs two simultaneous searches, one form initial state called as **forward-search** and other from goal node called as **backward-search**, to find the goal node.
- This replaces one single search graph with two small sub graphs in which one starts the search from an initial vertex and other starts from goal vertex.
- The search stops when these two graphs **intersect** each other.

Bidirectional Search Algorithm:

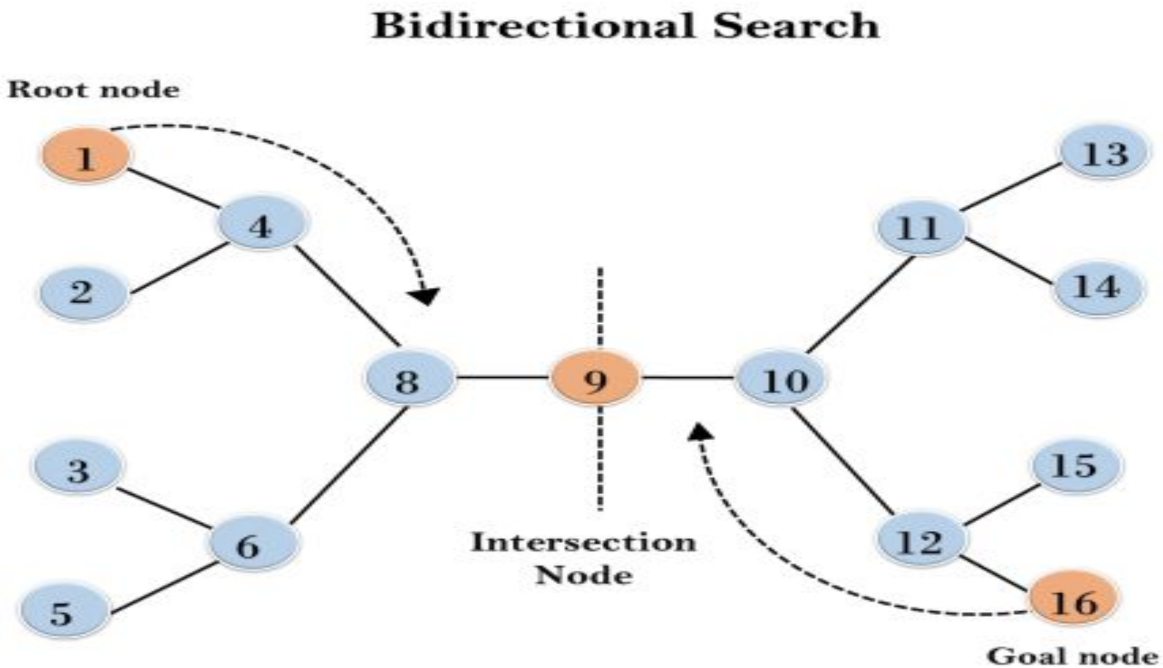
Advantages:

- Bidirectional search is fast.
- Bidirectional search requires less memory

Disadvantages:

- Implementation of the bidirectional search tree is difficult.
- **In bidirectional search, one should know the goal state in advance.**

Bidirectional Search Algorithm:



In given search tree, bidirectional search algorithm is applied. This algorithm divides one graph/tree into two sub-graphs. It starts traversing from **node 1** in the forward direction and starts from goal **node 16** in the backward direction.

The algorithm terminates at node 9 where two searches meet.

Bidirectional Search Algorithm:

- **Completeness:** It is complete if we use BFS in both searches.
- **Time Complexity:** Time complexity of bidirectional search using BFS is *$O2(b^{d/2})$ and* total complexity would be $O(b^{d/2} + b^{d/2})$
- **Space Complexity:** Space complexity of bidirectional search is $\mathbf{O(b^{d})}$.
- **Optimal:** Bidirectional search is Optimal.

Informed Search Algorithms

- This algorithm use domain knowledge. It is also called a **Heuristic search**.
- Information is available which can guide the search to achieve the goal.
- It can find a solution more efficiently than an uninformed search strategy.
- A heuristic is a way which might not always be guaranteed for best solutions but guaranteed to find a good solution in reasonable time.
- An example of informed search algorithms is a traveling salesman problem.

Informed Search Algorithms

Heuristic search.

- It is use try to optimize a problem using heuristic function.
- Optimization means try to solve problem in minimum cost

Heuristic function.:-

- **It** is a function that gives an estimation cost from source to Goal node.
- It helps for selecting optimal node for expansion.
- The value of the heuristic function is always positive.

Heuristic function.

- It is a search which tries to reduce amount of search that must be done by making intelligent choice for the nodes that are selected for the expansion .
- It is represented by **h(n),**
- it calculates the cost of an optimal path between the pair of states.
- The value of the heuristic function is always positive.

Heuristic function.

- **The heuristic function is given as:**
- $h(n) <= h^*(n)$

Here h(n) is heuristic cost

h*(n) is the estimated cost.

Hence heuristic cost should be less than or equal to the estimated cost.

F(n)=G(n)+H(n)

F(n): Overall Cost or Estimated cost

G(n): Path cost

H(n): Heuristic Value

Types of Informed search Algorithm

1.) Best-first Search Algorithm (Greedy Search)

2.) A* Search Algorithm

3.)AO* Search Algorithm

Best-first Search Algorithm (Greedy Search)

- It always selects the path which appears best at that moment.
- It is the combination of depth-first search and breadth-first search algorithms.

Best-first Search Algorithm

Step 1: Place the starting node into the OPEN list.

Step 2: If the OPEN list is empty, Stop and return failure.

Step 3: Remove the node **n**, from the **OPEN list** which has the lowest value of h(n), and places it in the **CLOSED list.**

If node n is goal then return

else

Step 4: Expand the node n, and generate and check the successors of node n. and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 5.

Step 5: For each successor node, algorithm checks for evaluation function f(n), and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both list, then add it to the OPEN list.

Step 6: Return to Step 2.

Best-first Search Algorithm

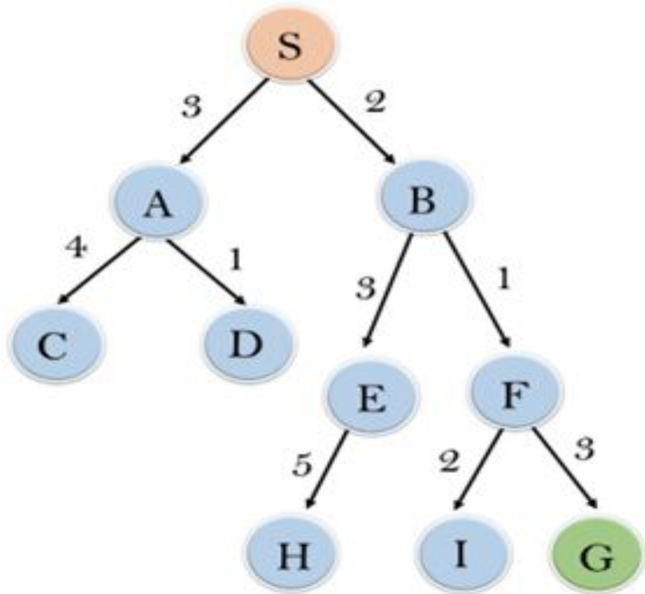
Advantages:

- Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
- This algorithm is more efficient than BFS and DFS algorithms.

Disadvantages:

- It can behave as an unguided depth-first search in the worst case scenario.
- It can get stuck in a loop as DFS.
- This algorithm is not optimal.

Best-first Search Algorithm



node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0

In this search example, we are using two lists which are **OPEN** and **CLOSED** Lists

Best-first Search Algorithm

OPEN List:-

- This list contain nodes that have been generated but not yet expanded.

CLOSED Lists

- This list contain the nodes that have been expanded and whose children's are available

to search program.

Best-first Search Algorithm

Expand the nodes of S and put in the CLOSED list

- **Open list** **closed list**

Initialization: Open [A, B], Closed [S]

Iteration 1: Open [A], Closed [S, B]

Iteration 2: Open [E, F, A], Closed [S, B]

Open [E, A], Closed [S, B, F]

Iteration 3: Open [I, G, E, A], Closed [S, B, F]

Open [I, E, A], Closed [S, B, F, G]

Hence the final solution path will be:

S----> B----->F----> G

Best-first Search Algorithm

Time Complexity: The worst case time complexity of Greedy best first search is $O(b^m)$

Space Complexity: The worst case space complexity of Greedy best first search is $O(b^m)$. Where, m is the maximum depth of the search space.

Complete: Greedy best-first search is also incomplete, even if the given state space is finite.

Optimal: Greedy best first search algorithm is not optimal.

A* Search Algorithm:

- It is the most commonly variant of best-first search.
- This algorithm finds the shortest path through the search space using the **heuristic function i.e h(n)**
- This search algorithm expands less search tree and provides optimal result faster.
- It use h(n) and cost to reach node n from starting node i.e **g(n)**
- This algorithm is similar to UCS uses g(n)+h(n) instead of g(n).

A* Search Algorithm:

- In A* search algorithm, we use search heuristic h(n) as well as the cost to reach the node g(n).
- Hence we can combine both costs as following, and this sum is called as a fitness number.
- f(n)=g(n)+h(n) ------ fitness number.

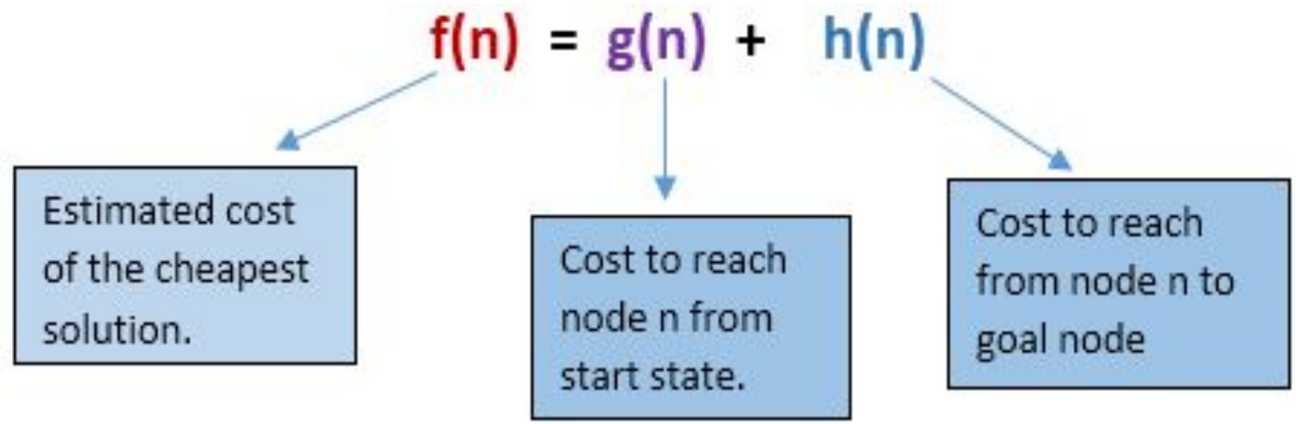
Where:

f(n):-estimated cost of the cheapest solution.

g(n):-cost to reach node ‘n’ from start state.

h(n):- cost to reach from node ‘n’ to goal node.

A* Search Algorithm:



Advantages:

- A* search algorithm is the best algorithm than other search algorithms.
- A* search algorithm is optimal and complete.
- This algorithm can solve very complex problems.

Disadvantages:

- It does not always produce the shortest path as it mostly based on heuristics and approximation.
- A* search algorithm has some complexity issues.
- The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

A* Search Algorithm:

Step1: Place the starting node in the OPEN list.

Step 2: Check if the OPEN list is empty or not, if the list is empty then return failure and stops.

Step 3: Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise

Step 4: Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.

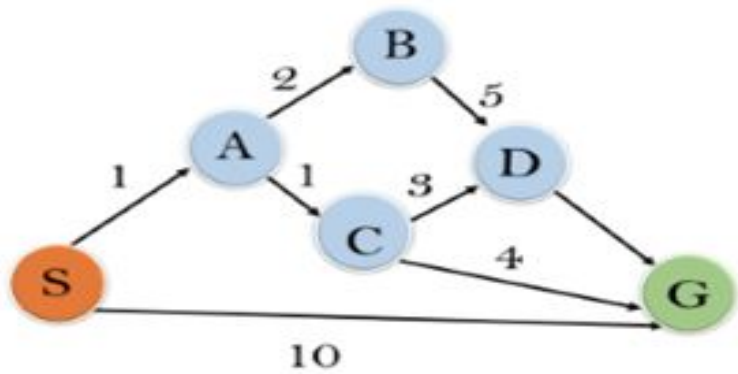
Step 5: Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.

Step 6: Return to **Step 2**.

A* Search Algorithm:

- **Example:**
- In this example, we will traverse the given graph using the A* algorithm. The heuristic value of all states is given in the below table so we will calculate the f(n) of each state using the formula f(n)= g(n) + h(n), where g(n) is the cost to reach any node from start state.
- Here we will use OPEN and CLOSED list.

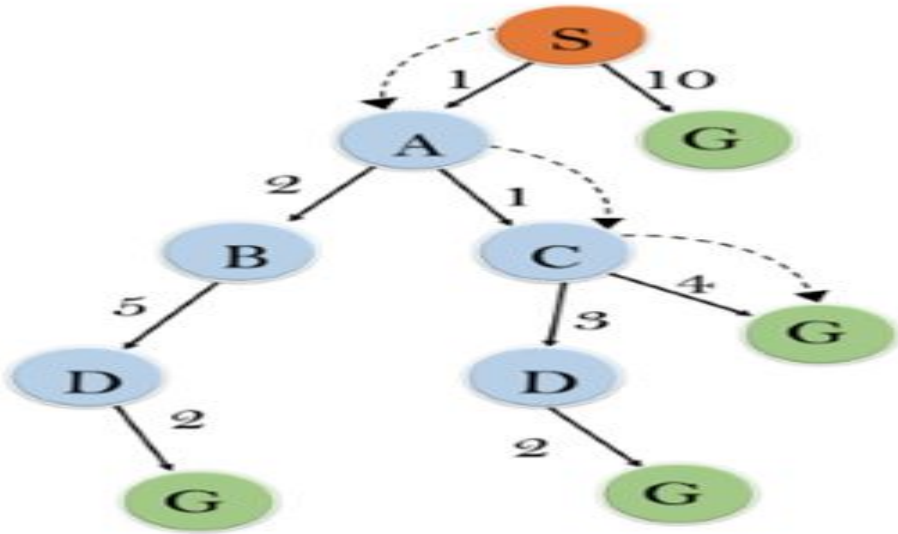
A* Search Algorithm:



State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0

Iteration1 : S-A= f(n)=g(n)+h(n)= 1+3=4
S-G=f(n)=g(n)+h(n)=10+0=10
Iteration2:S-A-B=f(n)=3+4=7
S-A-C=f(n)=2+2=4
Iteration 3: S-A-C-D=f(n)=5+6=11
S-A-C-G=f(n)=6+0=6
OPTIMAPL PATH=S-A-C-G WITH COST 6

A* Search Algorithm:



Initialization: {(S, 5)}
Iteration1: {(S--> A, 4), (S-->G, 10)}
Iteration2: {(S--> A-->C, 4), (S--> A-->B, 7), (S-->G, 10)}
Iteration3: {(S--> A-->C--->G, 6), (S--> A-->C--->D, 11), (S--> A-->B, 7), (S-->G, 10)}

A* Search Algorithm:

Complete: A* algorithm is complete if :

- Branching factor is finite.
- Cost at every action is fixed.

Optimal: A* search algorithm is optimal if it follows below two conditions:

1. **Admissible:** h(n) should be an admissible heuristic for A* tree search. An admissible heuristic is optimistic in nature.
2. **Consistency:** Second required condition is consistency for only A* graph-search.

   If the heuristic function is admissible, then A* tree search will always find the least cost path.

Time Complexity: It is depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution d. So the time complexity is O(b^d), where b is the branching factor.

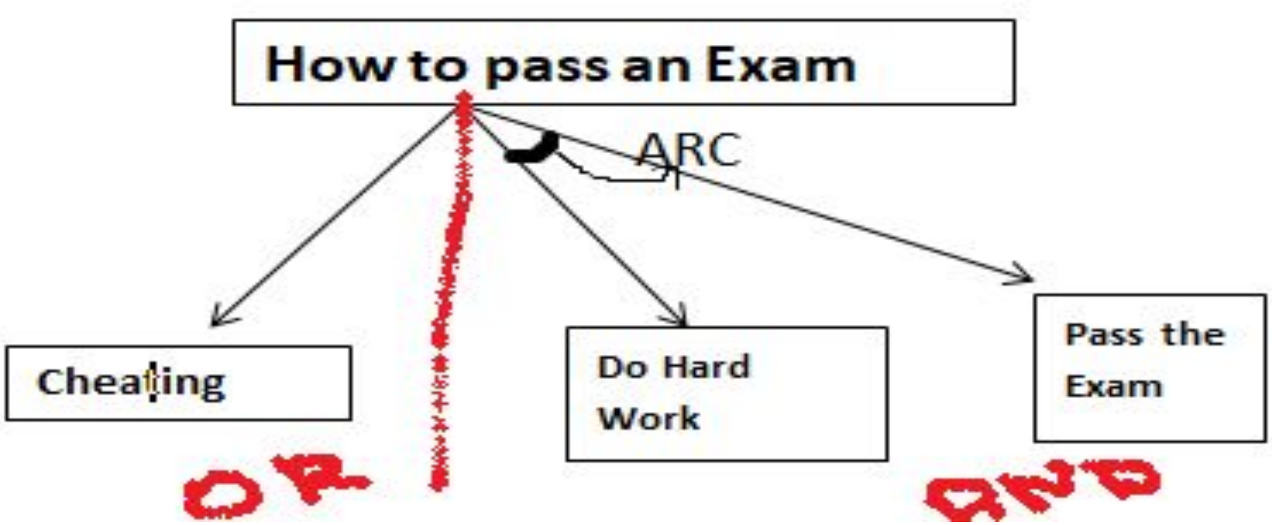
Space Complexity: The space complexity of A* search algorithm is **O(b^d)**

AO* Algorithm(AND –OR -Graph

- When a problem can be divided into a set of sub problems, where each sub problem can be solved separately and a combination of these will be a solution.
- AND-OR graphs or AND - OR trees are used for representing the solution.
- AND-OR graphs are useful for certain problems where the solution involved decomposing the problem into smaller problem and solve these sub problem.

AO* Algorithm

The decomposition of the problem or problem reduction generates AND arcs.



AO* Algorithm

1. To pass any exam, we have two options, either cheating or hard work.
2. In this graph we are given two choices, first do cheating **or** work hard and **(The arc)** pass.
3. When we have more than one choice and we have to pick one, we apply **OR condition** to choose one.
4. Basically the **ARC** here denote **AND condition**.
5. Here we have replicated the arc between the work hard and the pass because by doing the hard work possibility of passing an exam is more than cheating.

AO* Algorithm

Step1:initialize the graph to start node.

Step2:Traversed the graph following the current path accumulating nodes that have not been expanded.

Step3:Pick any of these nodes and expand it if it has no successors call this value FUTILITY otherwise calculate only **f** for each of the successors .

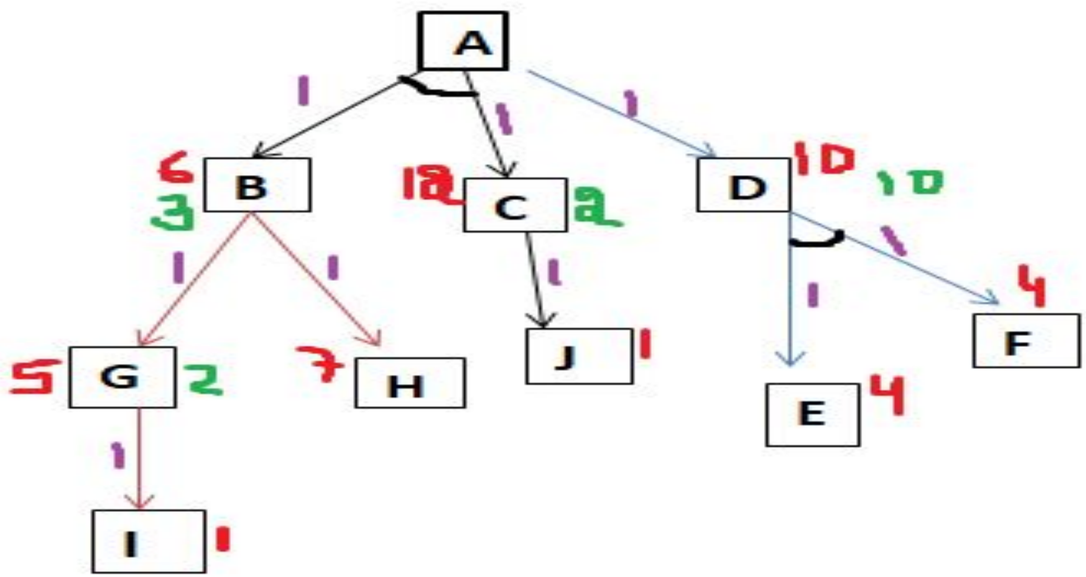
Step4:If f is 0 then mark the node as solved.

Step5:change the value of f for newly created node to reflect its successors by back propagation.

Step6:whenever possible use the most promising routes and if a node is marked as solved then mark the parent node as solved .

Step7: If starting node is solved or value greater than FUTILITY stop ,else repeat step2

AO* Algorithm



How AO* works

The algorithm always moves towards a **lower cost value.**

We will calculate the **cost function** here **(F(n)= G (n) + H (n))**

H: heuristic/ estimated value of the nodes.

G: actual cost or edge value .

The **Purple color** values are **edge values (here all are same that is one).**

The **Red color** values are **Heuristic values for nodes.**

The Green color values are **New Heuristic values for nodes.**

AO* Algorithm

- **Procedure:**
- In the above diagram we have two ways from **A to D** or **A to B-C** (because of and condition). calculate cost to select a path
- **F(A-D)= 1+10 = 11** and **F(A-BC) = 1 + 1 + 6 +12 = 20**
- As we can see **F(A-D)** is less than **F(A-BC)** then the algorithm choose the path **F(A-D).**
- Form D we have one choice that is **F-E.**
- **F(A-D-FE) = 1+1+ 4 +4 =10**
- Basically **10** is the cost of reaching **FE from D.** And **Heuristic value of node D** also denote the cost of reaching **FE from D**. So, the new Heuristic value of D is 10.
- And the Cost from A-D remain same that is **11**.
- Suppose we have searched this path and we have got the **Goal State**, then we will never explore the other path. (this is what AO* says but here we are going to explore other path as well to see what happen)

Difference between A* and AO*

- An **A* algorithm** represents an OR graph algorithm that is used to find a single solution (either this or that).
- An **AO* algorithm** represents an AND-OR graph algorithm that is used to find more than one solution by ANDing more than one branch.
- **Note: AO* will always find minimum cost solution.**

Local Search Algorithm

- To provide the best possible result, local search algorithms move iteratively from one possible solution to a neighbor solution and so on until the best possible set of results is achieved.

components of local search algorithm.

1. Search /State space
2. Neighbourhood Relation
3. Cost function

Hill Climbing Algorithm

- It is a local search algorithm which continuously moves in the direction to find the peak of the mountain or best solution to the problem.
- It terminates when it reaches a peak value where no neighbor has a higher value.
- It is a technique which is used for optimizing the mathematical problems.
- examples of Hill climbing algorithm is Traveling-salesman Problem in which we need to minimize the distance traveled by the salesman.
- It algorithm mostly used when a good heuristic is available.

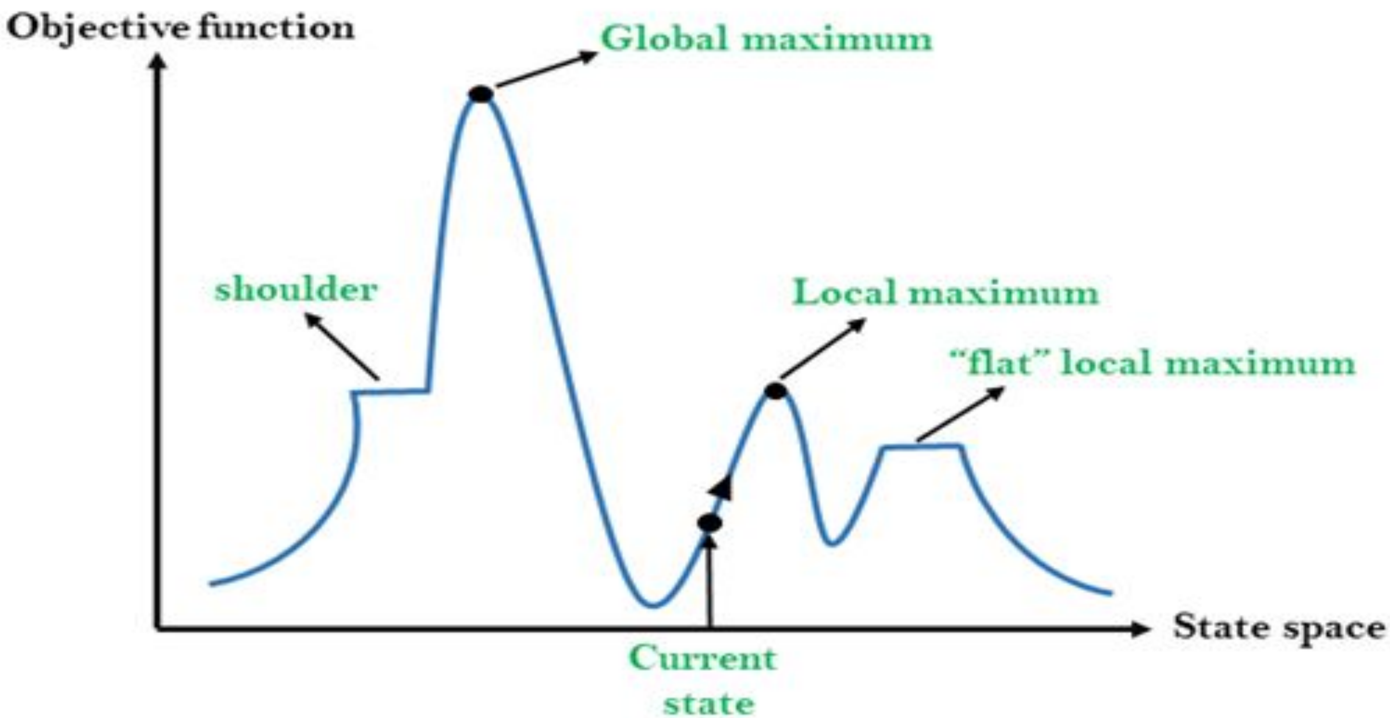
Features of Hill Climbing:

- **Generate and Test variant:** Hill Climbing is the variant of Generate and Test method. The Generate and Test method produce feedback which helps to decide which direction to move in the search space.
- **Greedy approach:** Hill climbing algorithm search moves in the direction which optimizes the cost.
- **No backtracking:** It does not backtrack the search space, as it does not remember the previous states.

State-space Diagram for Hill Climbing:

- The state-space is a graphical representation of the hill-climbing algorithm which is showing a graph between various states of algorithm and Objective function/Cost.
- On Y-axis we have taken the function which can be an objective function or cost function
- on the x-axis we have taken state-space

State-space Diagram for Hill Climbing



Different regions in the state space

- **Local Maximum:** Local maximum is a state which is better than its neighbor states, but there is also another state which is higher than it.
- **Global Maximum:** Global maximum is the best possible state of state space landscape. It has the highest value of objective function.
- **Current state:** It is a state in a landscape diagram where an agent is currently present.
- **Flat local maximum/ plateau :** It is a flat space in the landscape where all the neighbor states of current states have the same value.
- **Shoulder:** It is a plateau region which has an uphill edge.