Search Algorithms in Artificial Intelligence

- Search algorithms are one of the most important areas of Artificial Intelligence. This topic will explain all about the search algorithms in AI.

Problem-solving agents:

- In Artificial Intelligence, Search techniques are universal problem-solving methods. **Rational agents** or **Problem-solving agents** in AI mostly used these search strategies or algorithms to solve a specific problem and provide the best result. Problem-solving agents are the goal-based agents and use atomic representation. In this topic, we will learn various problem-solving search algorithms.

Search Algorithm Terminologies:

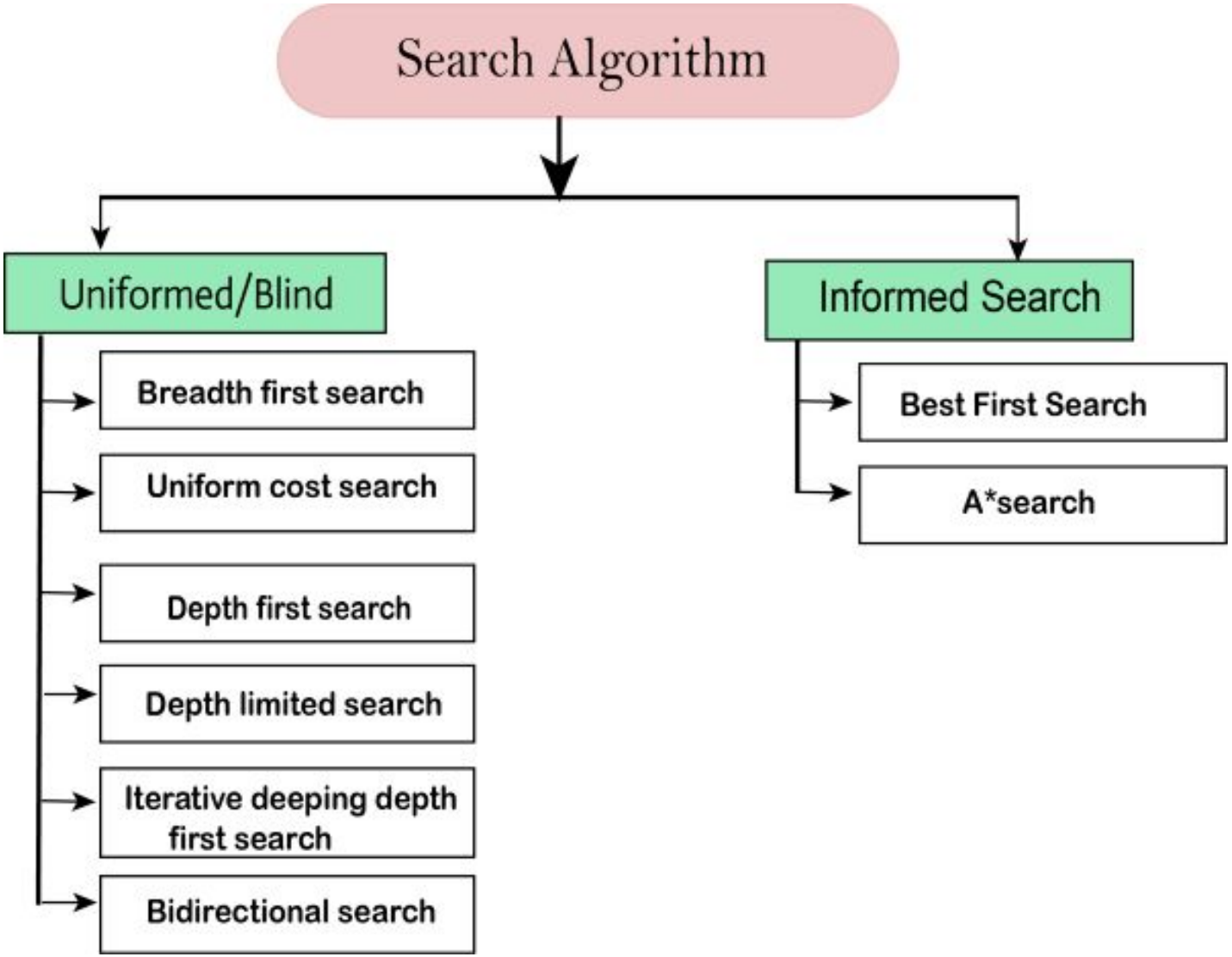
- **Search:** Searchingis a step by step procedure to solve a search-problem in a given search space. A search problem can have three main factors:
  - **Search Space:** Search space represents a set of possible solutions, which a system may have.
  - **Start State:** It is a state from where agent begins **the search**.
  - **Goal test:** It is a function which observe the current state and returns whether the goal state is achieved or not.

- **Search tree:** A tree representation of search problem is called Search tree. The root of the search tree is the root node which is corresponding to the initial state.
- **Actions:** It gives the description of all the available actions to the agent.
- **Transition model:** A description of what each action do, can be represented as a transition model.
- **Path Cost:** It is a function which assigns a numeric cost to each path.
- **Solution:** It is an action sequence which leads from the start node to the goal node.
- **Optimal Solution:** If a solution has the lowest cost among all solutions.

Properties of Search Algorithms:

- **Completeness:** A search algorithm is said to be complete if it guarantees to return a solution if at least any solution exists for any random input
- **Optimality:** If a solution found for an 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



Uninformed/Blind Search:

- The uninformed search does not contain any domain knowledge such as closeness, the location of the goal. It operates in a brute-force way as it only includes information about how to traverse the tree and how to identify leaf and goal nodes. Uninformed search applies a way in which search tree is searched without any information about the search space like initial state operators and test for the goal, so it is also called blind search. It examines each node of the tree until it achieves the goal node.
- **It can be divided into five main types:**
- Breadth-first search
- Uniform cost search
- Depth-first search
- Iterative deepening depth-first search
- Bidirectional Search

Informed Search

- Informed search algorithms use domain knowledge. 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. Informed search is also called a Heuristic search.
- A heuristic is a way which might not always be guaranteed for best solutions but guaranteed to find a good solution in reasonable time.
- Informed search can solve much complex problem which could not be solved in another way.
- An example of informed search algorithms is a traveling salesman problem.
- Greedy Search
- A* Search

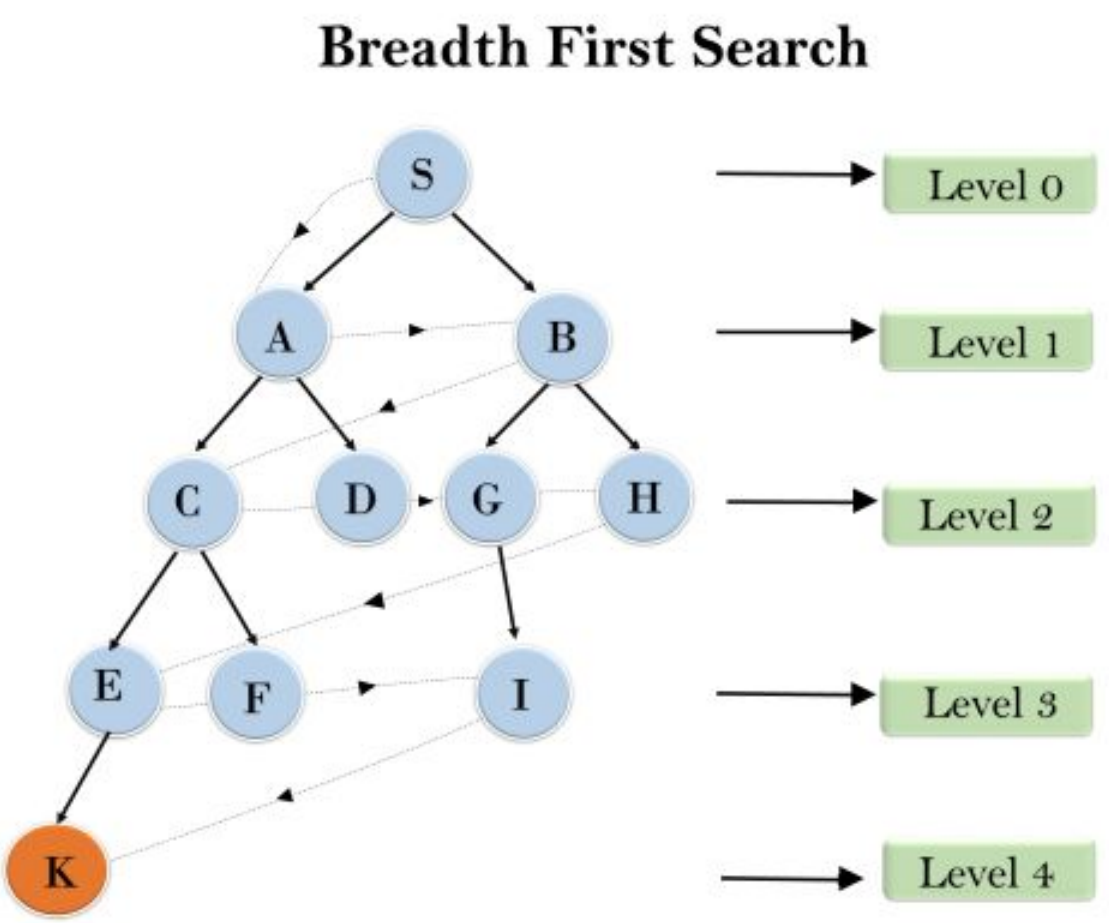
1. Breadth-first Search:

- Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches breadthwise in a tree or graph, so it is called breadth-first search.
- BFS 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 implemented using FIFO queue data structure.

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

- **Disadvantages of BFS:**

- 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.



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

BFS

- **Time Complexity:** Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node. Where the d= depth of shallowest solution and b is a node at every state.
- **$T(b) = 1+b^2+b^3+.....+ b^d= O(b^d)$**
- **Space Complexity:** Space complexity of BFS algorithm 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.
- **Optimality:** BFS is optimal if path cost is a non-decreasing function of the depth of the node.
-

2. Depth-first Search

- Depth-first search isa 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 for its implementation.
- The process of the DFS algorithm is similar to the BFS algorithm.

DFS Advantages

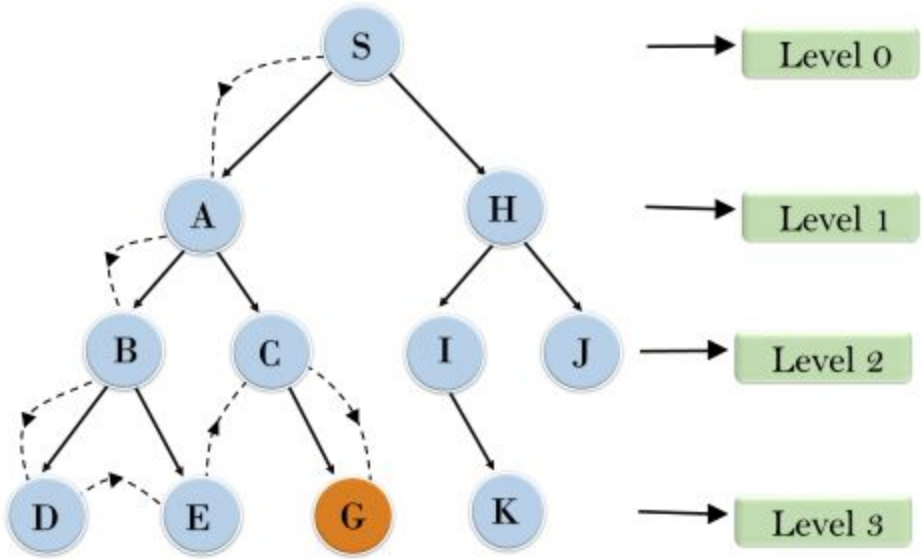
- **Advantage:**
- DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
- It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).

Disadvantages of DFS

- 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.

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

Depth First Search



DFS

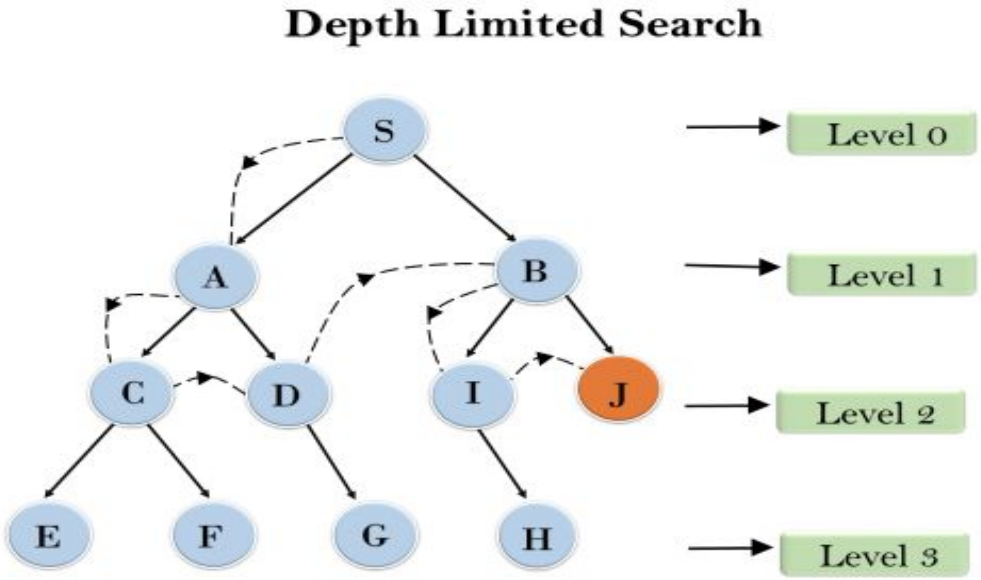
- **Completeness:** DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.
- **Time Complexity:** Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by:

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

3. Depth-Limited Search Algorithm:

- A depth-limited search algorithm is similar to depth-first search with a predetermined limit. Depth-limited search can solve the drawback of the infinite path in the Depth-first search. In this algorithm, the node at the depth limit will treat as it has no successor nodes further.
- Depth-limited search can be terminated with two Conditions of failure:
- Standard failure value: It indicates that problem does not have any solution.
- Cutoff failure value: It defines no solution for the problem within a given depth limit.

- **Advantages:**
- Depth-limited search is Memory efficient.
- **Disadvantages:**
- Depth-limited search also has a disadvantage of incompleteness.
- It may not be optimal if the problem has more than one solution.



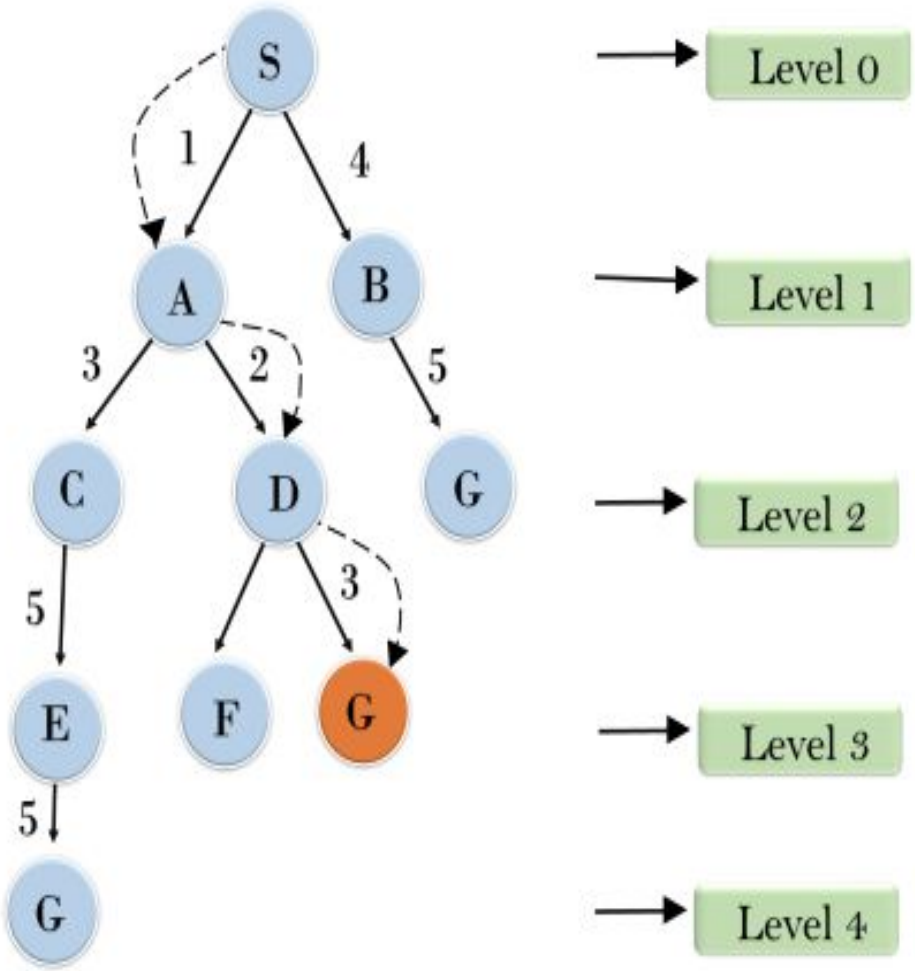
- **Completeness:** DLS search algorithm is complete if the solution is above the depth-limit.
- **Time Complexity:** Time complexity of DLS algorithm is **$O(b^{\ell})$**.
- **Space Complexity:** Space complexity of DLS algorithm is **$O(b \times \ell)$**.
- **Optimal:** Depth-limited search can be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

4. Uniform-cost Search Algorithm:

- Uniform-cost search is a searching algorithm used for traversing a weighted tree or graph. This algorithm comes into play when a different cost is available for each edge. The primary goal of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost. Uniform-cost search expands nodes according to their path costs form the root node. It can be used to solve any graph/tree where the optimal cost is in demand. A uniform-cost search algorithm is implemented by the priority queue. It gives maximum priority to the lowest cumulative cost. Uniform cost search is equivalent to BFS algorithm if the path cost of all edges is the same.

- **Advantages:**
- Uniform cost search is optimal because at every state the path with the least cost is chosen.
- **Disadvantages:**
- It does not care about the number of steps involve in searching and only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.

Uniform Cost Search



- **Time Complexity:**
- Let C* **is Cost of the optimal solution**, and **ε** is each step to get closer to the goal node. Then the number of steps is = C*/ε+1. Here we have taken +1, as we start from state 0 and end to C*/ε.
- Hence, the worst-case time complexity of Uniform-cost search is$\mathbf{O(b^{1+[C^*/\varepsilon]})/}$.
- **Space Complexity:**
- The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is $\mathbf{O(b^{1+[C^*/\varepsilon]})}$.
- **Optimal:**
- Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

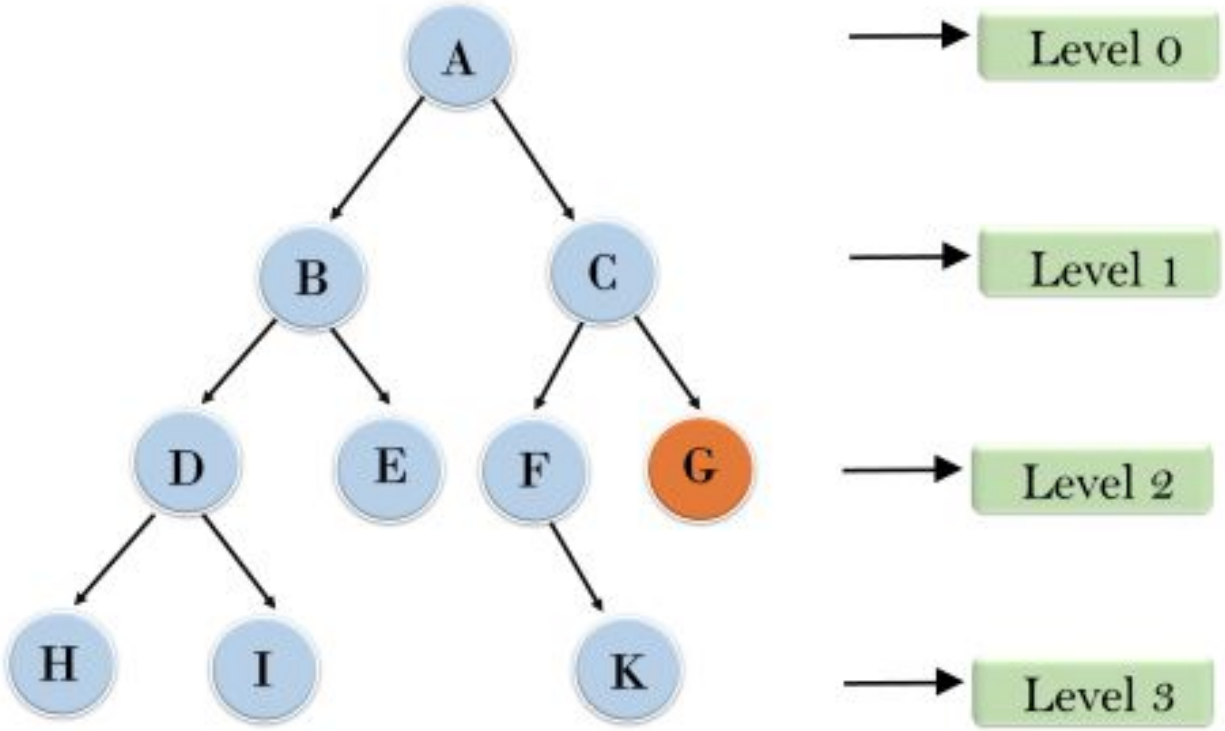
Iterative deepening depth-first Search:

- The iterative deepening algorithm is a combination of DFS and BFS algorithms. This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found.
- This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found.
- This Search algorithm combines the benefits of Breadth-first search's fast search and depth-first search's memory efficiency.
- The iterative search algorithm is useful uninformed search when search space is large, and depth of goal node is unknown.

- **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.

- Example:
- Following tree structure is showing the iterative deepening depth-first search. IDDFS algorithm performs various iterations until it does not find the goal node. The iteration performed by the algorithm is given as:

Iterative deepening depth first search



1'st Iteration-----> A
2'nd Iteration----> A, B, C
3'rd Iteration------>A, B, D, E, C, F, G
4'th Iteration------>A, B, D, H, I, E, C, F, K, G
In the fourth iteration, the algorithm will find the goal node.

- **Completeness:**

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

- **Time Complexity:**

Let's suppose b is the branching factor and depth is d then the worst-case time complexity is **$O(b^d)$**.

- **Space Complexity:**

The space complexity of IDDFS will be **O(bd)**.

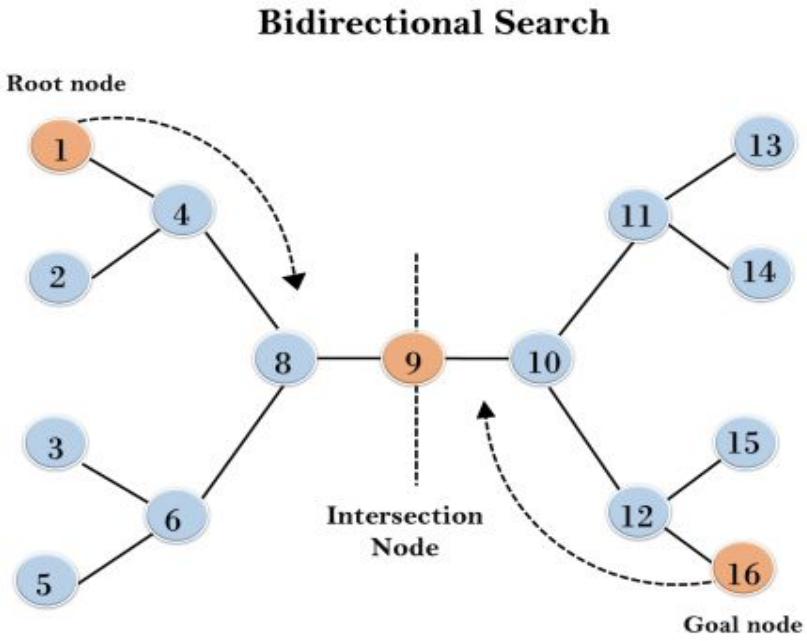
- **Optimal:**

IDDFS algorithm is optimal if path cost is a non-decreasing function of the depth of the node.

Bidirectional Search Algorithm

- Bidirectional search 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. Bidirectional search **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 can use search techniques such as **BFS, DFS, DLS,** etc.

- **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.**
- Example:
- In the below 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.**



- **Completeness:** Bidirectional Search is complete if we use BFS in both searches.
- **Time Complexity:** Time complexity of bidirectional search using BFS is **$O(b^d)$**.
- **Space Complexity:** Space complexity of bidirectional search is **$O(b^d)$**.
- **Optimal:** Bidirectional search is Optimal.

Heuristic techniques

- A heuristic is a technique that is **used to solve a problem faster than the classic methods**. These techniques are used **to find the approximate solution** of a problem when classical methods do not.
- Heuristics are said to be the problem-solving techniques that result in **practical and quick solutions.**

Why do we need heuristics?

- Heuristics are used in situations in which there is the requirement of a **short-term solution**. On facing complex situations with limited resources and time, Heuristics can help the companies to make **quick decisions** by **shortcuts** and **approximated calculations**. Most of the heuristic methods **involve mental shortcuts** to make **decisions on past experiences**.
- The heuristic method might **not** always provide us the **finest solution**, but it is assured that it helps us find a **good solution in a reasonable time.**

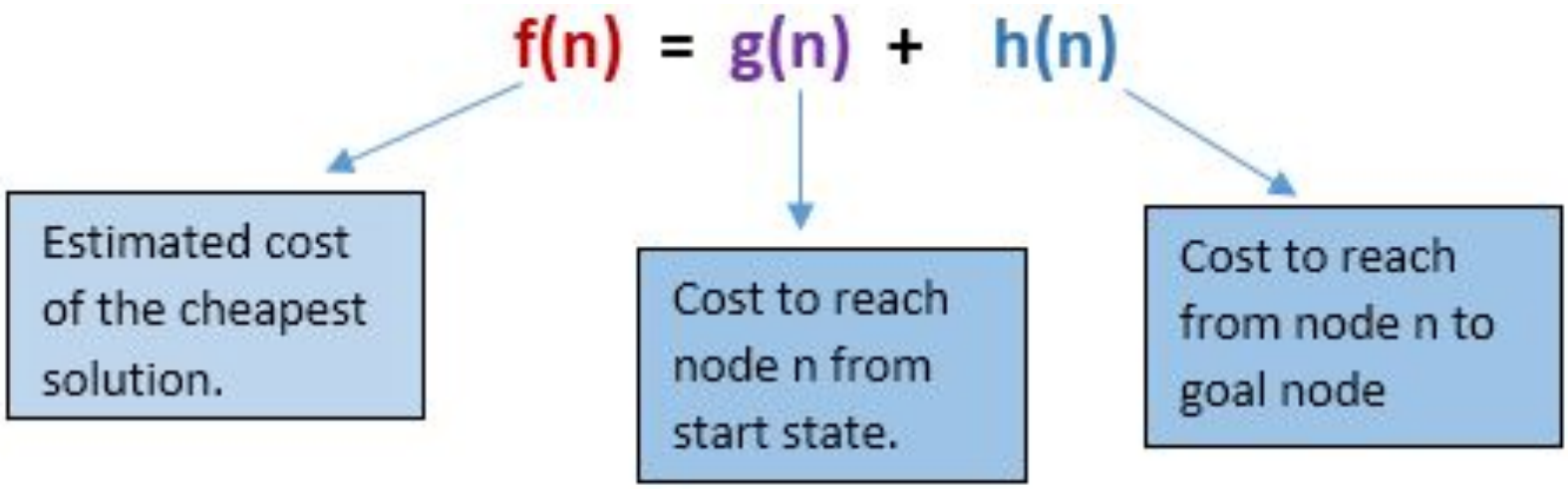

Heuristic Search in Artificial Intelligence
Hill Climbing
Constraint Satisfaction Problems
Simulated Annealing
Best-First Search (BFS)

A* Search Algorithm

- A* search is the most commonly known form of **best-first search.** It uses the **heuristic** function **h(n)** and cost to reach the node n from the start state **g(n)**. It has combined features of **UCS** and greedy **best-first search**, by which it solve the problem efficiently.
- It **finds the shortest path** through the search space using the **heuristic function**. This search algorithm **expands fewer search tree** and **gives optimal results faster**.

- **Algorithm of A* search:**
- **Step 1:** 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 the evaluation function (g+h). If node n is the 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 the evaluation function for n' and place it into the 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.

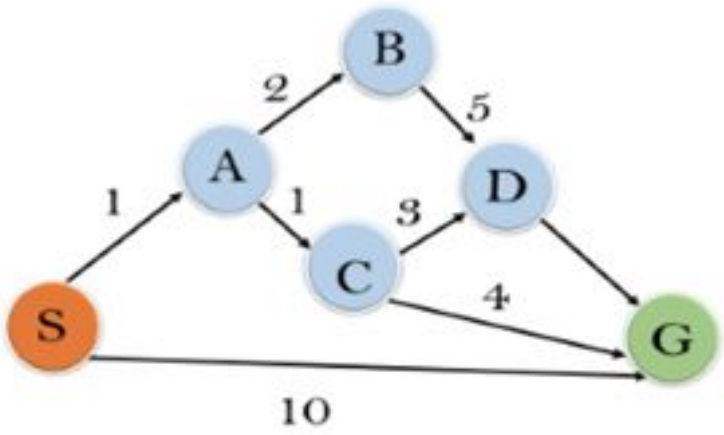
- https://youtu.be/vP5TkF0xJgI


f(n) = g(n) + h(n)
Estimated cost of the cheapest solution.
Cost to reach node n from start state.
Cost to reach from node n to goal node

- **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.**

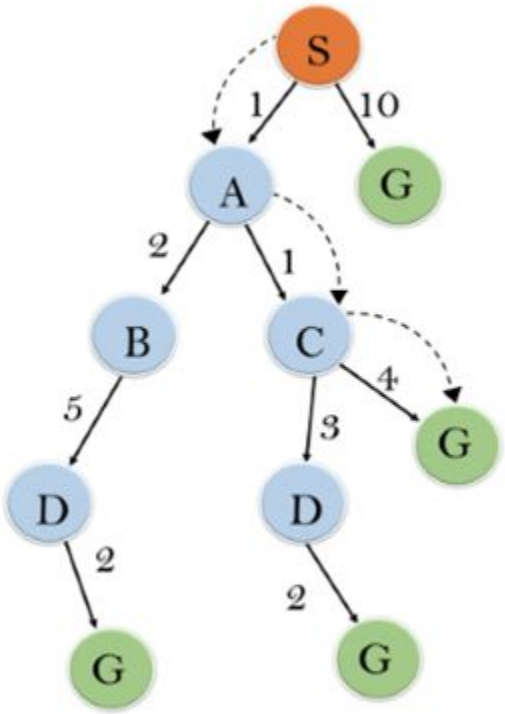
- **Disadvantages:**
- It **does not always produce the shortest path** as it mostly **based on heuristics and approximation**.
- A* search algorithm has **some complexity issues**.
- The main drawback of A* is **memory requirement** as it keeps all generated nodes in the memory, so it is **not practical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

Example



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

Solution



- **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)}
- **Iteration 4** will give the final result, as **S--->A--->C--->G** it provides the optimal path with cost 6.

- **Complete:** A* algorithm is complete as long as:
- Branching factor is finite.
- Cost at every action is fixed.
- **Optimal:** A* search algorithm is optimal if it follows below two conditions:
- **Admissible:** the first condition requires for optimality is that h(n) should be an admissible heuristic for A* tree search. An admissible heuristic is optimistic in nature.
- **Consistency:** Second required condition is consistency for only A* graph-search.

- **Time Complexity:** The time complexity of A* search algorithm 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)**

Best-first Search Algorithm (Greedy Search):

- Greedy best-first search algorithm always selects the path which appears **best** at that moment. It is the **combination of depth-first search and breadth-first** search algorithms. It uses the heuristic function and search. Best-first search allows us to take the advantages of both algorithms. With the help of best-first search, at each step, we can choose the most promising node. In the best first search algorithm, we expand the node which **is closest to the goal node and the closest cost is estimated by heuristic function**, i.e.
- f(n)= g(n).
- Were, h(n)= estimated cost from node n to the goal.
- The greedy best first algorithm is implemented by the priority queue.

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.
- **Step 4:** Expand the node n, and generate the successors of node n.
- **Step 5:** Check each successor 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 6.
- **Step 6:** 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 7:** Return to Step 2.

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.

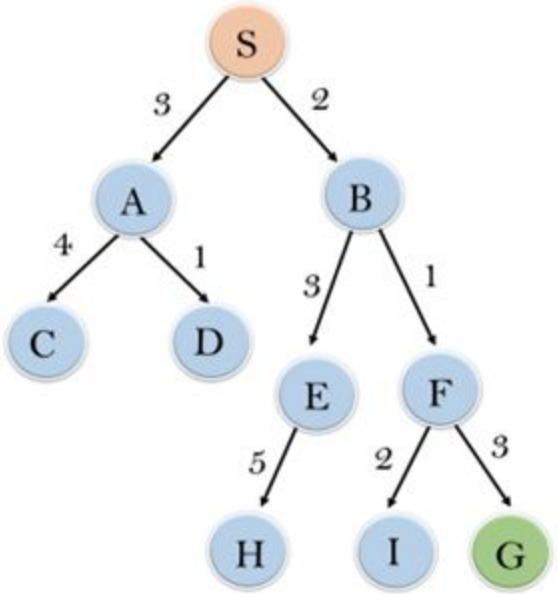
Disadvantages

- 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**.

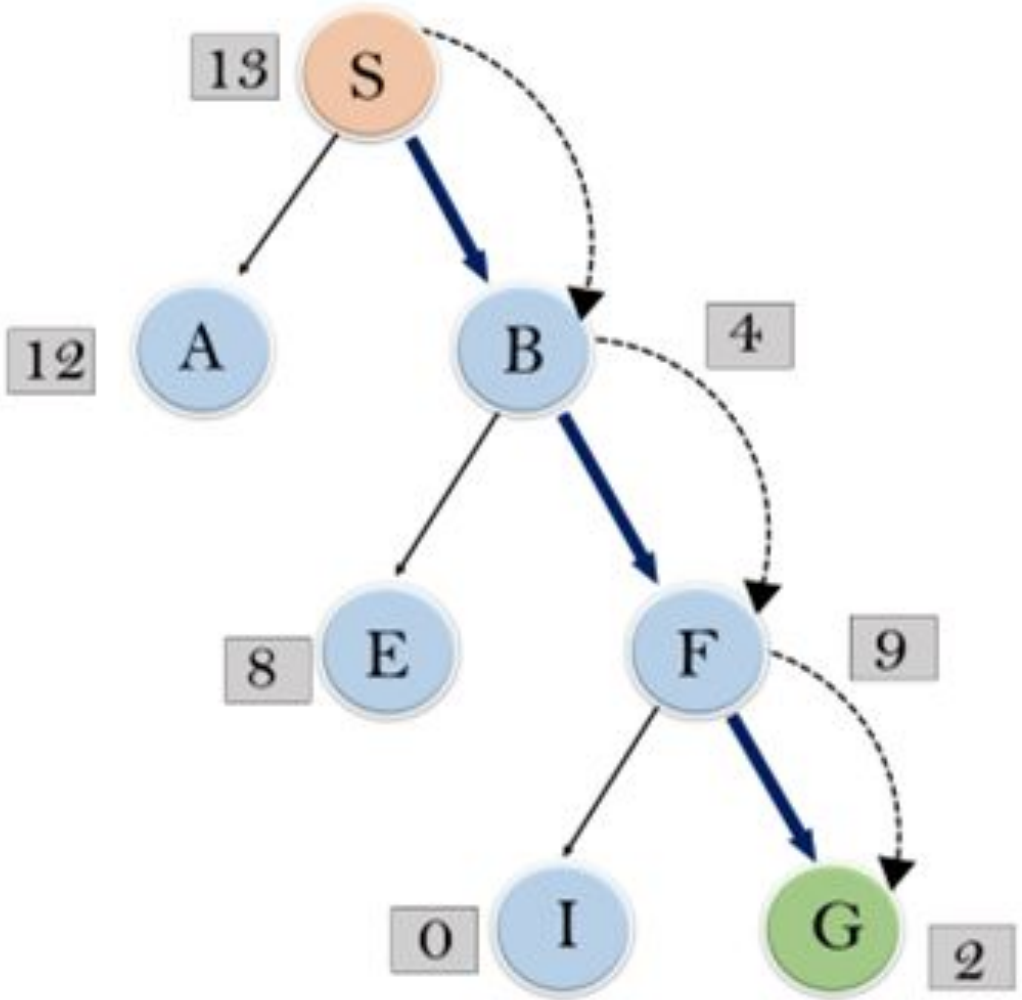
Example

- Consider the below search problem, and we will traverse it using greedy best-first search. At each iteration, each node is expanded using evaluation function f(n)=h(n) , which is given in the below table.

In this search example, we are using two lists which
are **OPEN** and **CLOSED** Lists. Following are the iteration for traversing the above e



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


13
S
12
A
B
4
8
E
F
9
0
I
G
2

- **Expand the nodes of S and put in the 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**

- **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.

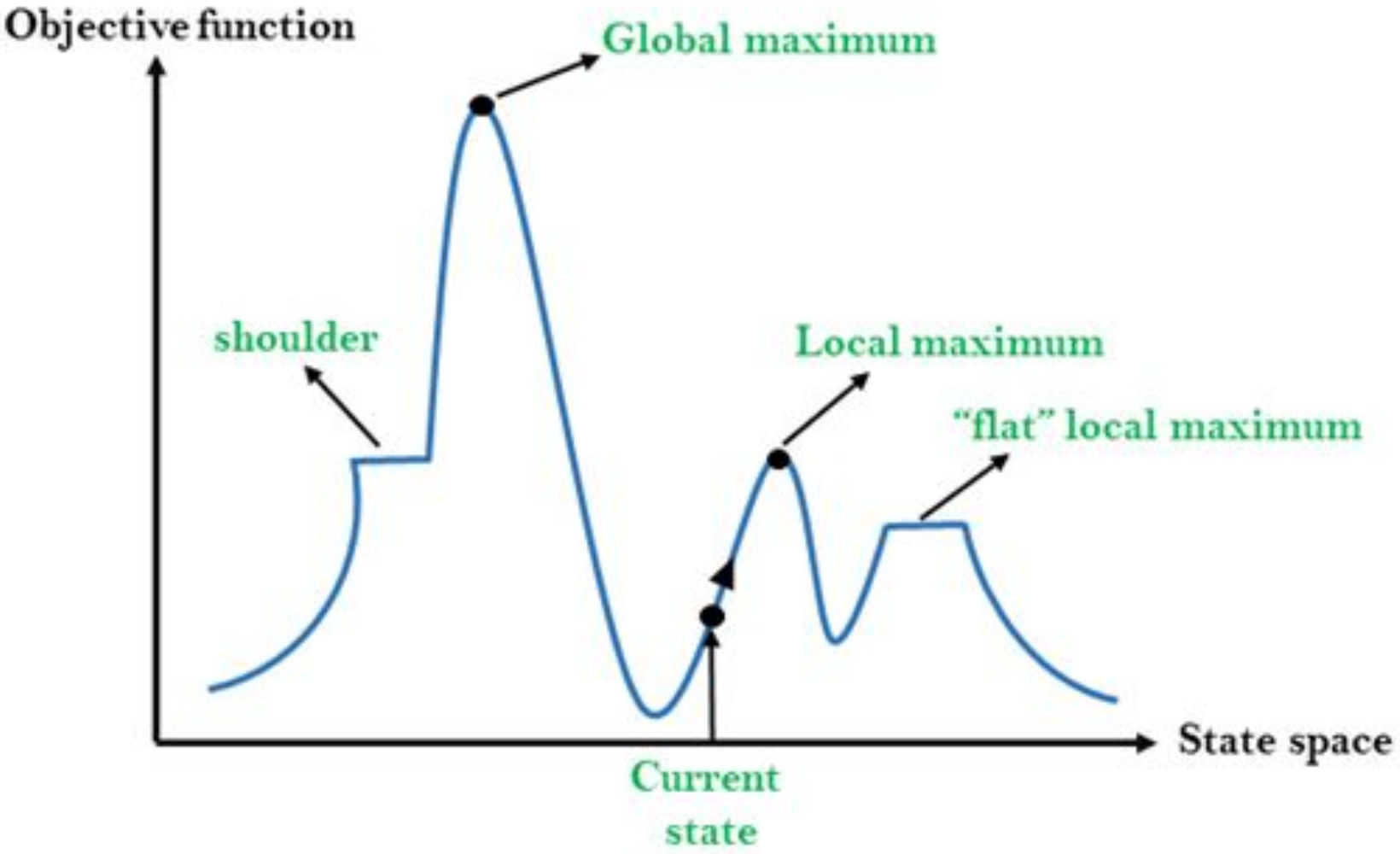
Hill Climbing Algorithm

- It is a technique for optimizing the **mathematical problems**. Hill Climbing is **widely used when a good heuristic is available.**
- It is a local search algorithm that continuously moves in the **direction of increasing elevation/value** to find the **mountain's peak** or the **best solution to the problem**.
- It **terminates when it reaches a peak value** where no neighbour has a higher value.
- Travelling-salesman Problem is one of the widely discussed examples of the Hill climbing algorithm, in which we need to minimize the distance travelled by the salesman.

- **Step 1:** Evaluate the initial state. If it is the goal state, then return success and Stop.
- **Step 2:** Loop Until a solution is found or there is no new operator left to apply.
- **Step 3:** Select and apply an operator to the current state.
- **Step 4:** Check new state:
- If it is a goal state, then return to success and quit.
- Else if it is better than the current state, then assign a new state as a current state.
- Else if not better than the current state, then return to step2.
- **Step 5:** Exit.

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.


Objective function
Global maximum
shoulder
Local maximum
"flat" local maximum
State space
Current
state

Different regions in the state space landscape:

- **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:** 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.

Constraint Satisfaction Problems in Artificial Intelligence

- We have encountered a wide variety of methods, including **adversarial search and instant search,** to address various issues.
- Every method for issue has a single purpose in mind: to **locate a remedy that will enable that achievement** of the objective. However there were no restrictions just on bots' capability **to resolve issues as well as arrive at responses** in **adversarial search and local search**, respectively.
- These section examines the constraint optimization methodology, another form or real concern method. By its name, constraints fulfilment implies that such an issue must be solved while adhering to a set of restrictions or guidelines.

- Three factors affect restriction compliance, particularly regarding:

- It refers to a group of parameters, or X.
- D: The variables are contained within a collection several domain. Every variables has a distinct scope.
- C: It is a set of restrictions that the collection of parameters must abide by.

Types of Constraints in CSP

- Basically, there are three different categories of limitations in regard towards the parameters:
- Unary restrictions are the easiest kind of restrictions because they only limit the value of one variable.
- Binary resource limits: These restrictions connect two parameters. A value between x1 and x3 can be found in a variable named x2.
- Global Resource limits: This kind of restriction includes a unrestricted amount of variables.

- Think of a Sudoku puzzle where some of the squares have initial fills of certain integers.
- You must complete the empty squares with numbers between 1 and 9, making sure that no rows, columns, or blocks contains a recurring integer of any kind. This solving multi - objective issue is pretty elementary. A problem must be solved while taking certain limitations into consideration.
- The integer range (1-9) that really can occupy the other spaces is referred to as a domain, while the empty spaces themselves were referred as variables. The values of the variables are drawn first from realm. Constraints are the rules that determine how a variable will select the scope.

Thank You