Problem Solving By Searching

In real world, there are different types of problems. Problem Solving in games such as <Sudoku= can be an example. It can be done by building an artificially intelligent system to solve that particular problem. To do this one needs to define the problem statements first and then generating the solution by keeping the conditions in mind. Some of the most popularly used problem solving with the help of artificial intelligence is, Chess, Travelling, Salesman Problem, Tower of Hanoi Problem.

Problem Searching

In general, searching refers to as finding information one needs. Searching is the most commonly used technique of problem solving in artificial intelligence. The searching algorithm helps us to search for solution of particular problem.

Problem

Problems are the issues which come across any system. A solution is needed to solve that particular problem.

Steps to Solve Problem using Artificial Intelligence

The process of solving a problem consists of five steps. These are:

1. **Defining the Problem:**
   The definition of the problem must be included precisely. It should contain the possible initial as well as final situations which should result in acceptable solution.
2. **Analyzing the Problem:**
   Analyzing the problem and its requirement must be done as few features can have immense impact on the resulting solution.
3. **Identification of Solutions:**
   This phase generates reasonable amount of solutions to the given problem in a particular range.
4. **Choosing a Solution:**
   From all the identified solutions, the best solution is chosen basis on the results produced by respective solutions.
5. **Implementation:**
   After choosing the best solution, its implementation is done.

PROBLEM SOLVING AGENTS

1.Goal formulation

It is the simplest and first step in problem solving ,where it organizes the sequence required to formulate one goal from out of many goals as well as action to the achieve the goal.

2Problem Formulation

Problem formulation involves deciding what actions and states to consider, when the description about the goal is provided. It is composed of:

- Initial State - start state
- Possible actions that can be taken
- Transition model – describes what each action does
- Goal test – checks whether current state is goal state
- Path cost – cost function used to determine the cost of each path.

The initial state, actions and the transition model constitutes state space of the problem - the set of all states reachable by any sequence of actions. A path in the state spaceis a sequence of states connected by a sequence of actions. The solution to the given problem is defined as the sequence of actions from the initial state to the goal states. The quality of the solution is measured by the cost function of the path, and an optimal solution is the one with most feasible path cost among all the solutions.

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.

Search Algorithm Terminologies

- **Search:** Searching is a step by step procedure to solve a search-problem in a given search space.

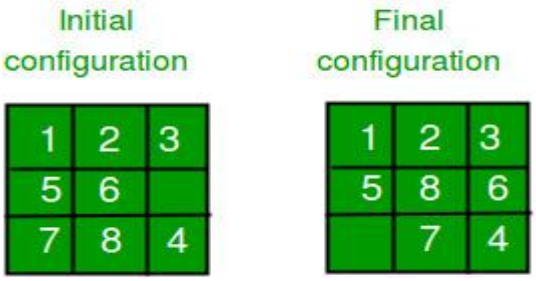
  A search problem can have three main factors:
  - Search Space: Search space represents a set of possible solutions, which a system may have.
  - Start State: It is a state from where agent begins the search.
  - Goal test: It is a function which observe the current state and returns whether the goal 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.

EXAMPLE:

8 puzzle Problem

n this puzzle solution of the 8 puzzle problem is discussed.
Given a 3×3 board with 8 tiles (every tile has one number from 1 to 8) and one empty space. The objective is to place the numbers on tiles to match the final configuration using the empty space. We can slide four adjacent (left, right, above, and below) tiles into the empty space.



Properties of Search Algorithms:

Following are the four essential properties of search algorithms to compare the efficiency of these 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

Based on the search problems we can classify the search algorithms into uninformed (Blind search) search and informed search (Heuristic search) algorithms.

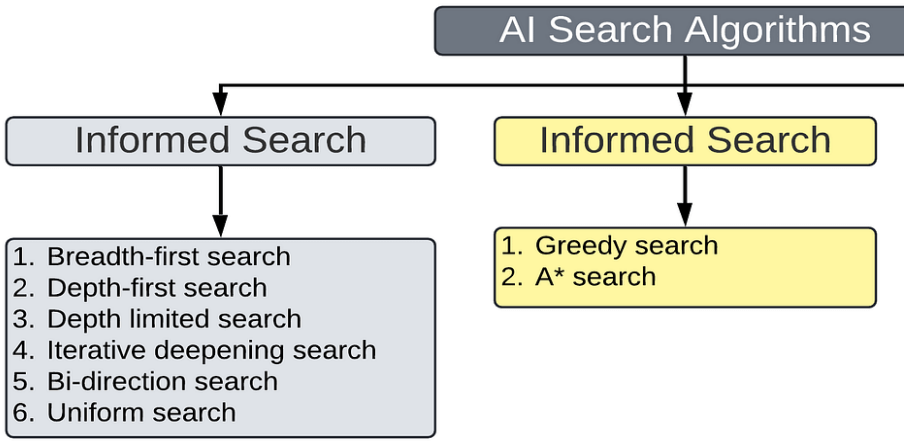


Fig: Types of Search Algorithm

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

- **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 nodes 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:

- 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:

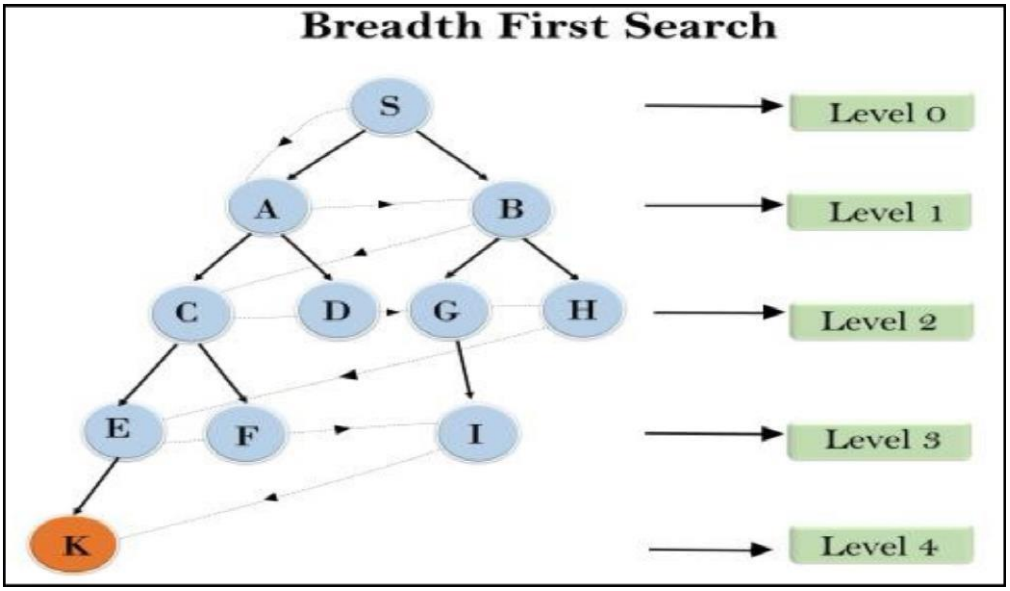
- It requires lots of memory since each level of the tree must be saved into memory to expand the next level.

BFS needs lots of time if the solution is far away from the root node.

- Example:

In the below tree structure, we have shown the traversing of the tree using BFS algorithm from the root node S to goal node K. BFS search algorithm traverse in layers, so it will follow the path which is shown by the dotted arrow, and the traversed path will be:

S→ A→ B→C→ D→G→H→ E→ F→ I → K



- Time Complexity: Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node.

$$\mathbf{T\ (b) = 1+b^2+b^3+.\ldots\ldots+ b^d= O\ (b^d)}$$

Where, d= depth of shallowest solution

b = node at every state.

- Space Complexity: Space complexity of BFS algorithm is given by the Memory size of frontierwhich is $\mathbf{O(b^d)}$.

- Completeness: BFS is **complete**, which means if the shallowest goal node is at some finitedepth, then BFS will find a solution.
- 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 for its implementation.
  - The process of the DFS algorithm is similar to the BFS algorithm.

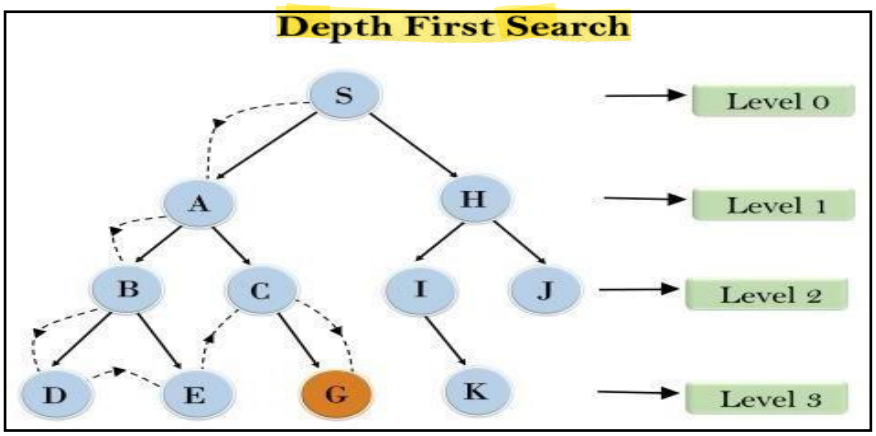
Advantage:

- 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).

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.

Example:



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.

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

  It is given by: $T(n)= 1+ n^2+ n^3 +.....+ n^m=O(n^m)$

Where, m= maximum depth of any node & this can be much larger than d (Shallowest solution depth)

- Space Complexity: DFS algorithm needs to store only single path from the root node, hencespace complexity of DFS is equivalent to the size of the fringe set, which is **O(bm)**.
- Optimality: DFS search algorithm is non-optimal, as it may generate a large number of steps orhigh cost to reach to the goal node.

- **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.

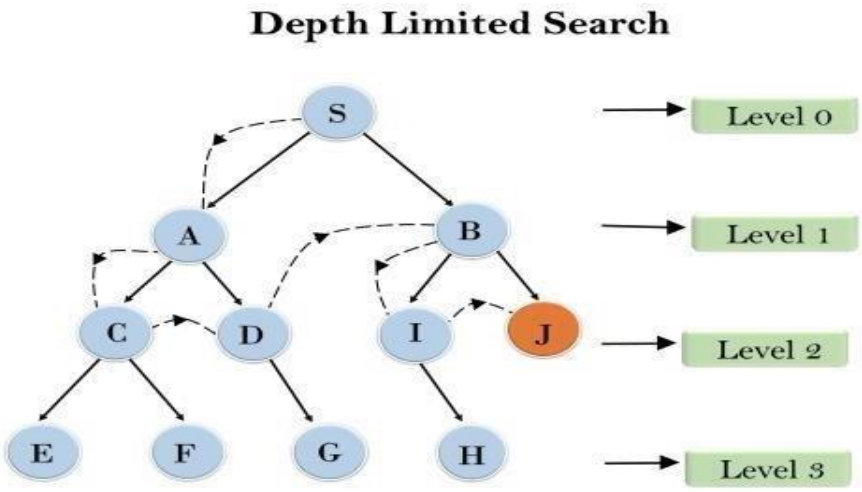
Advantages:

- 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.

Example:



- 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 $\mathbf{O(b \times \ell)}$.
- Optimality: Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if $\mathbf{\ell > d}$.

- **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 cumulativecost.
  - 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.

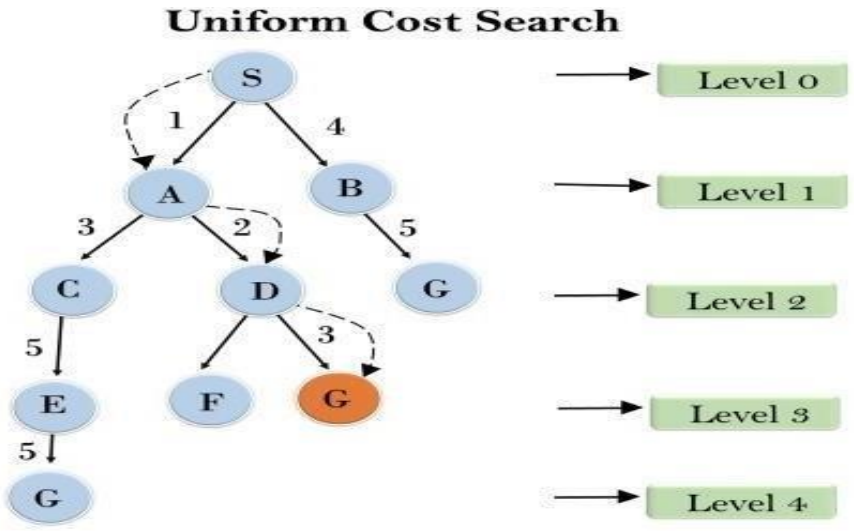
Advantages:

- 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.

Example:



- Completeness: Uniform-cost search is complete, such as if there is a solution, UCS will find it.
- 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-costsearch is $\mathbf{O(b^{1 + |C^*/\varepsilon|})}$.
- Optimality: Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

- **Iterative deepening depth-first Search Algorithm :**
  - 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:

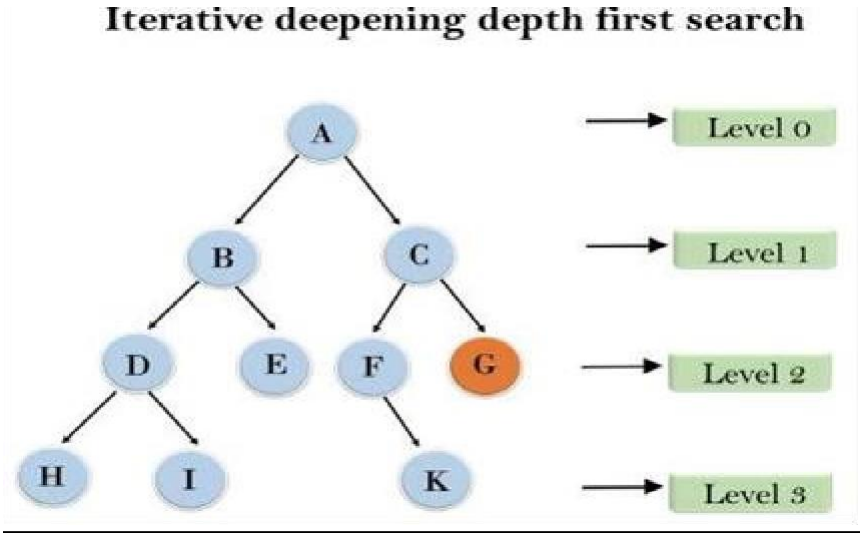
- It combines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.

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:



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)**.
- Optimality**:** 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:

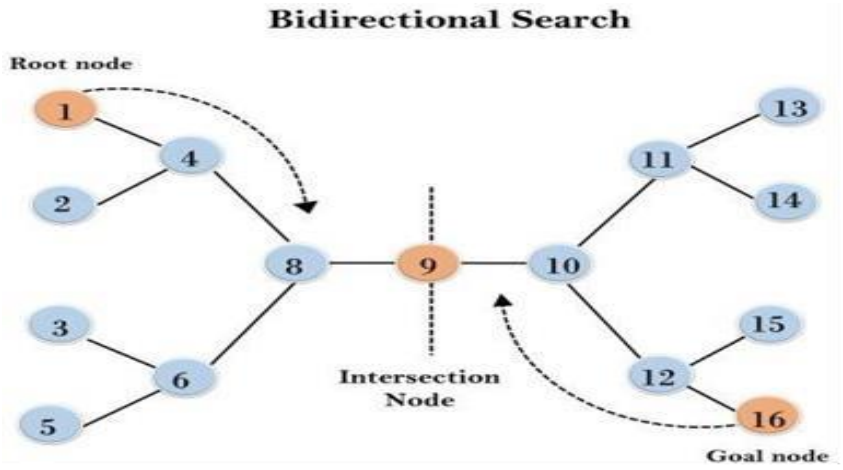
- Bidirectional search is fast.
- Bidirectional search requires less memory

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 **$O(b^d)$**.

- Optimality: Bidirectional search is Optimal.

Informed (Heuristic) Search Strategies

To solve large problems with large number of possible states, problem-specific knowledge needs to be added to increase the efficiency of search algorithms.

Pure Heuristic Search

- It expands nodes in the order of their heuristic values.
- It creates two lists, a closed list for the already expanded nodes and an open list for the created but unexpanded nodes.
- In each iteration, a node with a minimum heuristic value is expanded; all its child nodes are created and placed in the closed list.
- Then, the heuristic function is applied to the child nodes and they are placed in the open list according to their heuristic value.
- The shorter paths are saved and the longer ones are disposed.

Admissibility of the heuristic function is given as:

h(n) <= h*(n)

Here h(n) is heuristic cost, and h*(n) is the estimated cost.
Hence heuristic cost should be less than or equal to the estimated cost.

In the informed search we will discuss two main algorithms which are given below:

I. Best First Search Algorithm(Greedy search)
II. A* Search Algorithm

I. 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 andthe closest cost is estimated by heuristic function, i.e. **f(n)= h(n).**
  Where, 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:

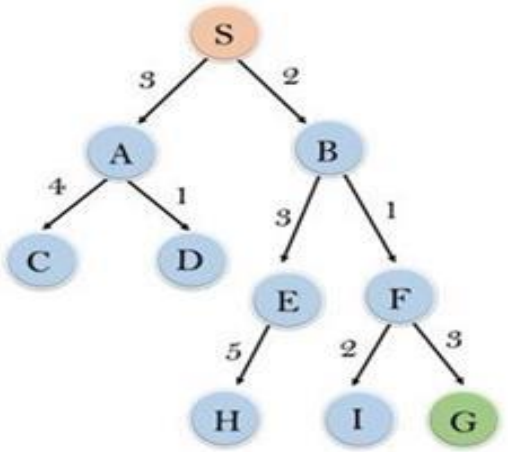
- Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
- This algorithm is more efficient than BFS and DFS algorithms.

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.

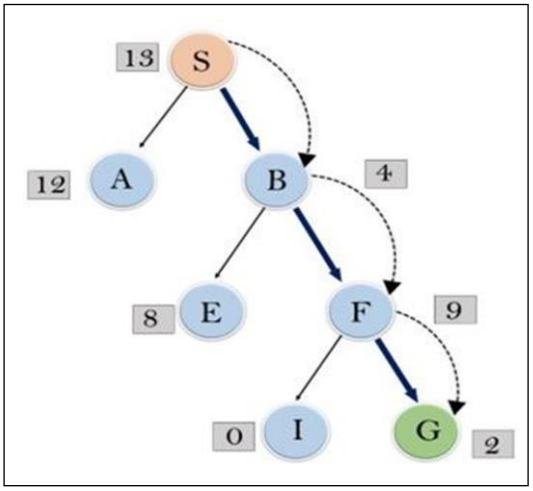


node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0

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

Following are the iteration for traversing the above example.

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

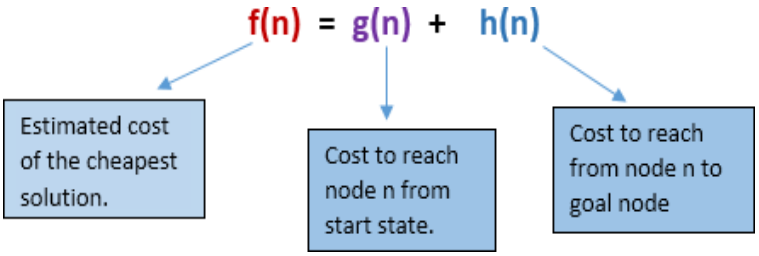


- **Time Complexity:** The worst case time complexity of Greedy best first search is **$O(b^m)$.**
- **Space Complexity:** The worst case space complexity of Greedy best first search is $O(b^m)$.Where, m is the maximum depth of the search space.
- **Complete:** Greedy best-first search is also incomplete, even if the given state space is finite.
- **Optimality:** Greedy best first search algorithm is not optimal.

II. A* Search Algorithm :

- A* search is the most commonly known form of best-first search.
- It uses heuristicfunction 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.
- A* search algorithm finds the shortest path through the search space using the heuristic function.
- This search algorithm expands less search tree and provides optimal result faster.
- A* algorithm is similar toUCS except that it uses g(n)+h(n) instead of g(n).

In A* search algorithm, we use search heuristic as well as the cost to reach the node. Hence we can combine both costs as following, and this sum is called as a **fitness number**.



Algorithm of A* Search:

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**.

Advantages:

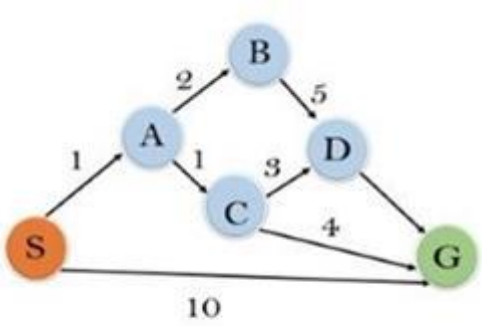
- A* search algorithm is the best algorithm than other search algorithms.
- A* search algorithm is optimal and complete.
- This algorithm can solve very complex problems.

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:

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.



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

Initialization: {(S, 5)}

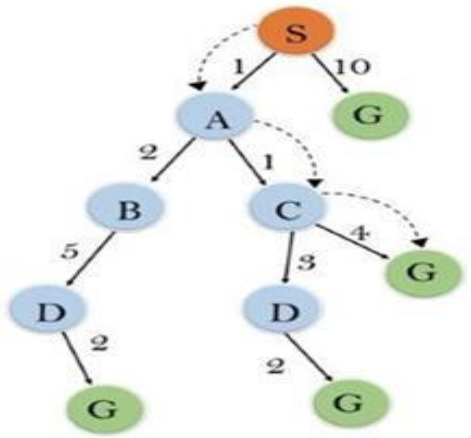
Iteration1: {(S--> A, 4), (S-->G, 10)}

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.



Points to remember:

- A* algorithm returns the path which occurred first, and it does not search for allremaining paths.
- The efficiency of A* algorithm depends on the quality of heuristic.

- **Completeness:** A* algorithm is complete as long as:
  - Branching factor is finite.
  - Cost at every action is fixed.
- **Optimality:** 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.

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

- AO* algorithm is a best first search algorithm.
- The AO* method divides any given difficult problem into a smaller group of problems that are then resolved using the AND-OR graph concept.
- AND OR graphs are specialized graphs that are used in problems that can be divided into smaller problems.
- The AND side of the graph represents a set of tasks that must be completed to achieve the main goal, while the OR side of the graph represents different methods for accomplishing the same main goal.

  - In the above figure, the buying of a car may be broken down into smaller problems or tasks that can be accomplished to achieve the main goal in the above figure, which is an example of a simple AND-OR graph.

- The other task is to either steal a car that will help us accomplish the main goal or use your own money to purchase a car that will accomplish the main goal.
- The AND symbol is used to indicate the AND part of the graphs, which refers to the need that all sub problems containing the AND to be resolved before the preceding node or issue may be finished.

- The start state and the target state are already known in the knowledge-based search strategy known as the AO* algorithm and the best path is identified by heuristics.
- The informed search technique considerably reduces the algorithm's time complexity.
- The AO* algorithm is far more effective in searching AND-OR trees than the A* algorithm.

Working of AO* Algorithm:

The evaluation function in AO* looks like this: f(n) = g(n) + h(n)

f(n) = Actual cost + Estimated cost

here,

f(n) = The actual cost of traversal.

g(n) = The cost from the initial node to the current node.

h(n) = Estimated cost from the current node to the goal state

Difference between the A* Algorithm and AO* Algorithm :

- A* algorithm and AO* algorithm both works on the best first search.
- They are both informed search and works on given heuristics values.
- A* always gives the optimal solution but AO* doesn't guarantee to give the optimal solution.
- Once AO* got a solution doesn't explore all possible paths but A* explores all paths.
- When compared to the A* algorithm, the AO* algorithm uses less memory.
- Opposite to the A* algorithm, the AO* algorithm cannot go into an endless loop.