UNIT II: Searching Techniques

Notes,

Syllabus: **Uninformed Search techniques, Informed Heuristic Based Search, Hill-climbing, Best-First Search, AND graph, Problem Reduction, AND-OR graph Algorithm, Constraint Satisfaction, Means-end analysis.**

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:** 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:
  1. **Search Space:** Search space represents a set of possible solutions, which a system may have.
  2. **Start State:** It is a state from where agent begins **the search**.
  3. **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:

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.

Importance of Search Algorithms in Artificial Intelligence

The following points explain how and why the search algorithms in AI are important:

- **Solving problems**: Using logical search mechanisms, including problem description, actions, and search space, search algorithms in artificial intelligence improve problem-solving. Applications for route planning, like Google Maps, are one real-world illustration of how search algorithms in AI are utilized to solve problems. These programs employ search algorithms to determine the quickest or shortest path between two locations.
- **Search programming**: Many AI activities can be coded in terms of searching, which improves the formulation of a given problem's solution.
- **Goal-based agents**: Goal-based agents' efficiency is improved through search algorithms in artificial intelligence. These agents look for the most optimal course of action that can offer the finest resolution to an issue to solve it.
- **Support production systems**: Search algorithms in artificial intelligence help production systems run. These systems help AI applications by using rules and methods for putting them into practice. Production systems use search algorithms in artificial intelligence to find the rules that can lead to the required action.
- **Neural network systems**: The neural network systems also use these algorithms. These computing systems comprise a hidden layer, an input layer, an output layer, and coupled nodes. Neural networks are used to execute many tasks in artificial intelligence. For example, the search for connection weights that will result in the required input-output mapping is improved by search algorithms in AI.

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.

Uninformed/Blind Search:

1. Breadth-first Search
2. Depth-first Search
3. Depth-limited Search
4. Iterative deepening depth-first search
5. Uniform cost search
6. Bidirectional Search

Uninformed search is a class of general-purpose search algorithms which operates in brute force-way. Uninformed search algorithms do not have additional information about state or search space other than how to traverse the tree, so it is also called blind search.

Following are the various types of uninformed search algorithms:

1. **Breadth-first Search**
2. **Depth-first Search**
3. **Depth-limited Search**
4. **Iterative deepening depth-first search**
5. **Uniform cost search**
6. **Bidirectional 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:

- 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.

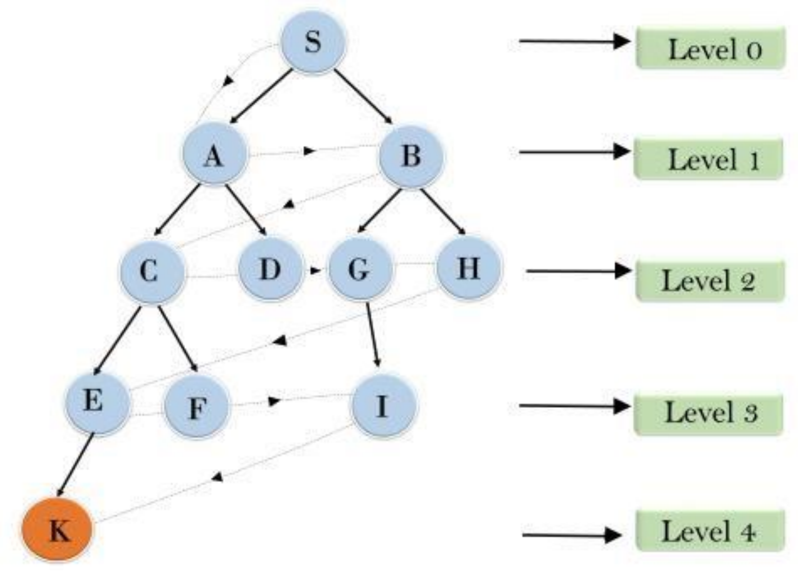
Disadvantages:

- 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:

1. 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. Where the d= depth of shallowest solution and b is a node at every state.

$T(b) = 1+b^2+b^3+. \quad + b^d= O(b^d)$

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.

Note: Backtracking is an algorithm technique for finding all possible solutions using recursion.

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:

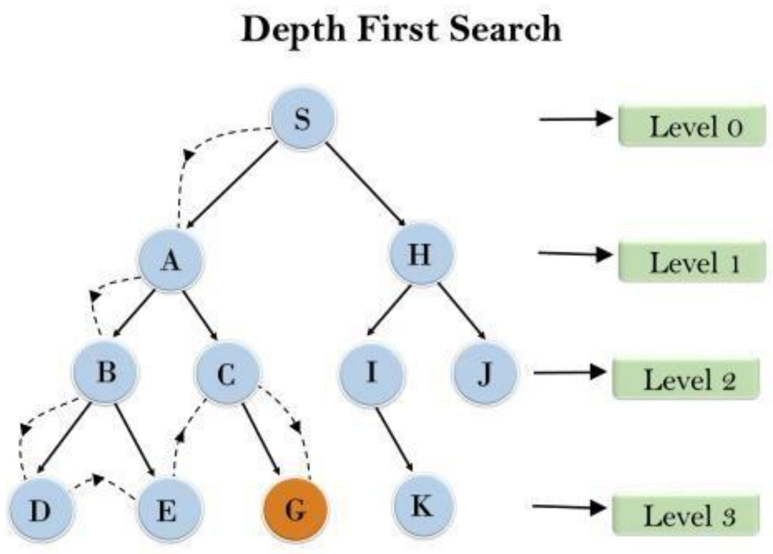
- There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
- DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

Example:

In the 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.

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 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.

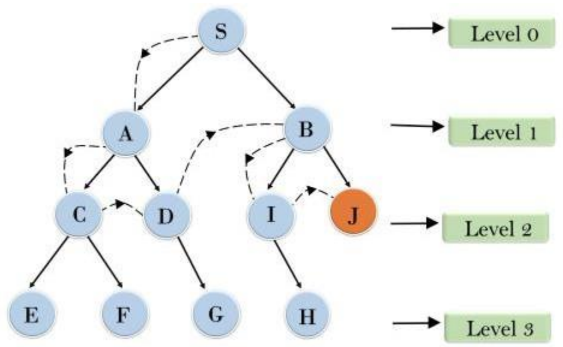
Advantages:

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.

Example:



Completeness: DLS search algorithm is complete if the solution is above the depth-limit.

Time Complexity: Time complexity of DLS algorithm is **O(b^ℓ)**.

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.

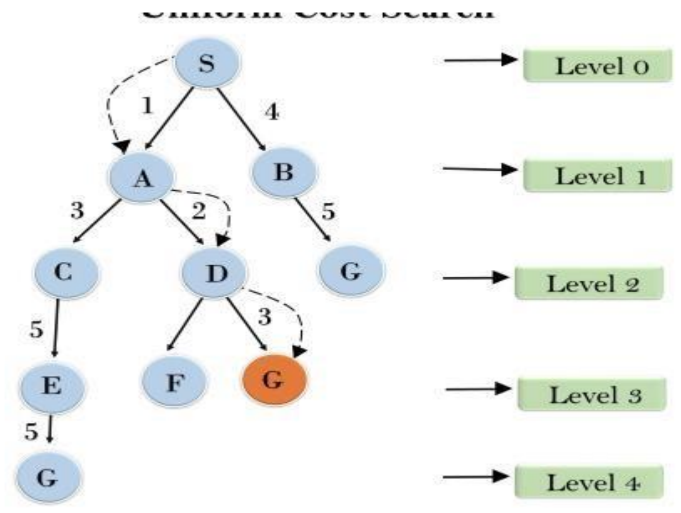
Advantages:

- 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.

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**$O(b^{1 + [C^*/\varepsilon]})$/**.

Space Complexity:

The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is **$O(b^{1 + [C^*/\varepsilon]})$**.

Optimal:

Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

Iterative deepeningdepth-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:

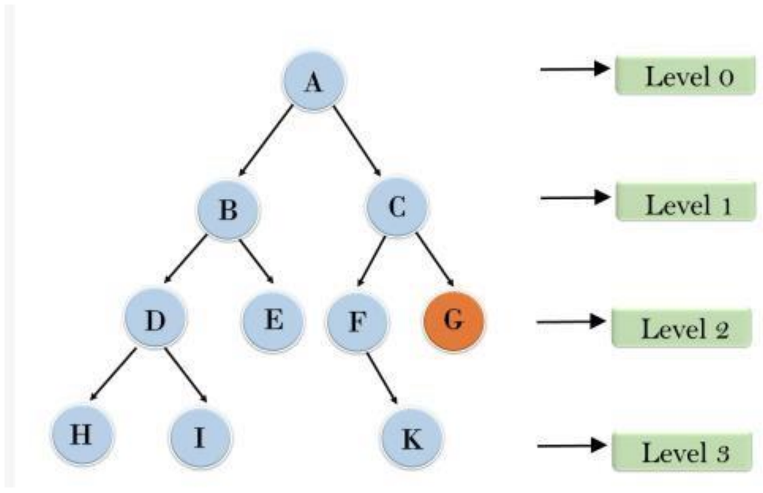
- Itcombines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.

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 ifthe 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 subgraphs 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

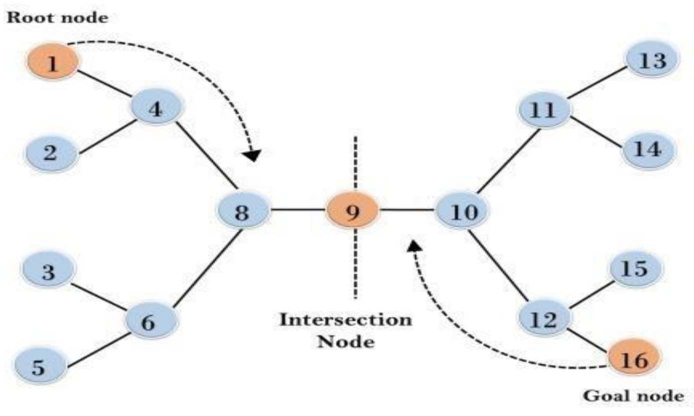
Disadvantages:

- 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.

So far we have talked about the uninformed search algorithms which looked through search space for all possible solutions of the problem without having any additional knowledge about search space. But informed search algorithm contains an array of knowledge such as how far we are from the goal, path cost, how to reach to goal node, etc. This knowledge help agents to explore less to the search space and find more efficiently the goal node.

The informed search algorithm is more useful for large search space. Informed search algorithm uses the idea of heuristic, so it is also called Heuristic search.

Heuristics function: Heuristic is a function which is used in Informed Search, and it finds the most promising path. It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal. The heuristic method, however, might not always give the best solution, but it guaranteed to find a good solution in reasonable time. Heuristic function estimates how close a state is to the goal. It is represented by h(n), and it calculates the cost of an optimal path between the pair of states. The value of the heuristic function is always positive.

Admissibility of the heuristic function is given as:

1. $h(n) <= h^*(n)$

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.

2. Informed Search/Heuristic Search:

Pure heuristic search is the simplest form of heuristic search algorithms. It expands nodes based on their heuristic value h(n). It maintains two lists, OPEN and CLOSED list. In the CLOSED list, it places those nodes which have already expanded and in the OPEN list, it places nodes which have yet not been expanded.

On each iteration, each node n with the lowest heuristic value is expanded and generates all its successors and n is placed to the closed list. The algorithm continues unit a goal state is found.

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

- Best First Search Algorithm(Greedy search)
    - Beam Search
- A* Search Algorithm
- AO * Search Algo
- Hill Climbing

- Simulated Annealing
- Constraint Satisfaction Problem {CSP}

1.) 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.

1. 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.

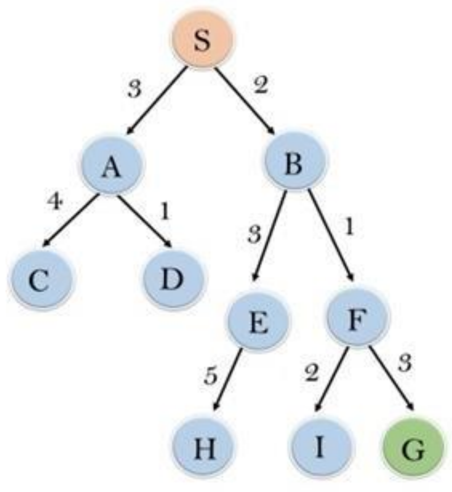
Disadvantages:

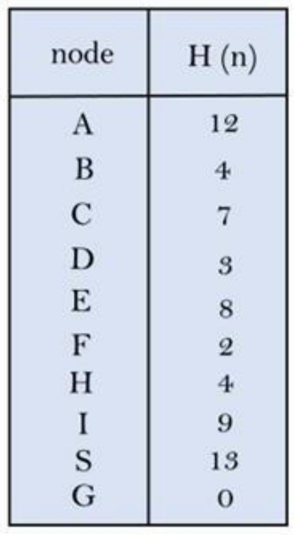
- It can behave as an unguided depth-first search in the worst case scenario.
- It can get stuck in a loop as DFS.

- This algorithm is not optimal.

Example:

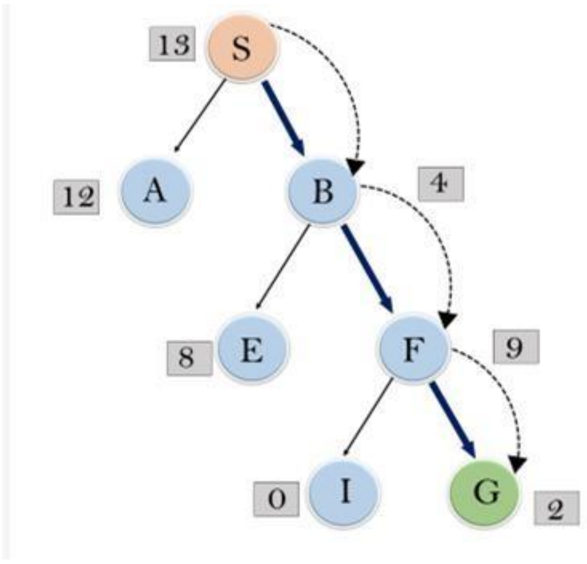
Consider the below search problem, and we will traverse it using greedy best-first search. At each iteration, each node is expanded using evaluation function f(n)=h(n) , which is given in the below table.





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.

Optimal: Greedy best first search algorithm is not optimal.

2)Beam Search

Beam Search is a greedy search algorithm similar to Breadth-First Search (BFS) and Best First Search (BeFS). In fact, we'll see that the two algorithms are special cases of the beam search.

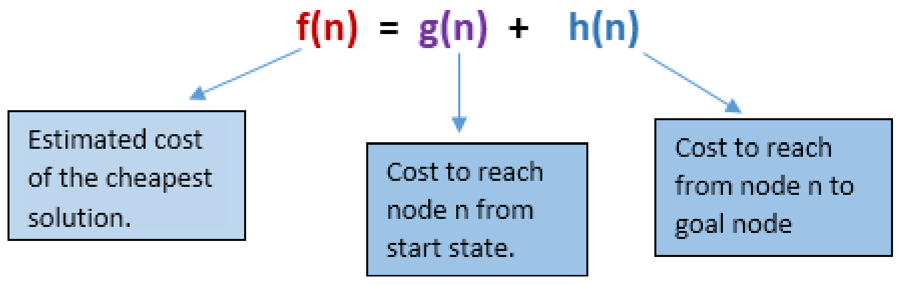
Compared to best-first search, an advantage of the beam search is that it requires less memory. This is because it doesn't have to store all the successive nodes in a queue. Instead, it selects only the best (beam width) ones.

On the other hand, it still has some of the problems of BeFS. The first one is that it is not complete, meaning that it might not find a solution at all. The second problem is that it is not optimal either. Therefore, when it returns a solution, it might not be the best solution.

3.) A* Search Algorithm:

A* search is the most commonly known form of best-first search. It uses 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. 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 to UCS 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**.


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

At each point in the search space, only those node is expanded which have the lowest value of f(n), and the algorithm terminates when the goal node is found.

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:

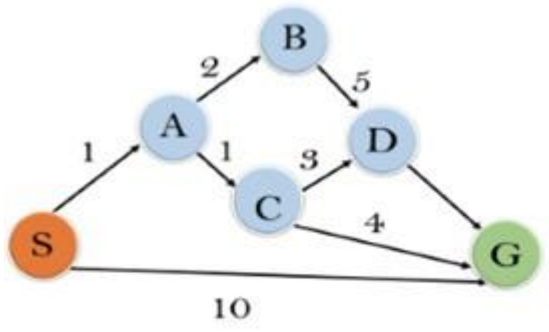
- A* search algorithm is the best algorithm than other search algorithms.
- A* search algorithm is optimal and complete.
- This algorithm can solve very complex problems.

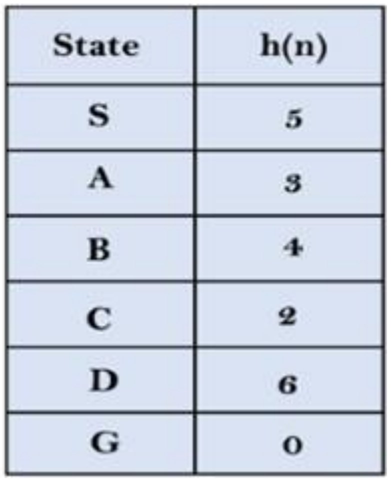
Disadvantages:

- 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:

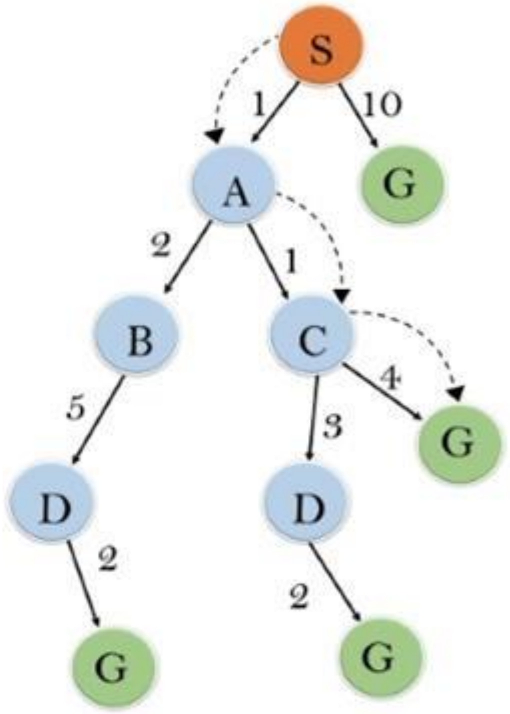
In this example, we will traverse the given graph using the A* algorithm. The heuristic value of all states is given in the below table so we will calculate the f(n) of each state using the formula f(n)= g(n) + h(n), where g(n) is the cost to reach any node from start state.
Here we will use OPEN and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





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

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.

Points to remember:

- A* algorithm returns the path which occurred first, and it does not search for all remaining paths.
- The efficiency of A* algorithm depends on the quality of heuristic.
- A* algorithm expands all nodes which satisfy the condition f(n)<="" li="">

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:

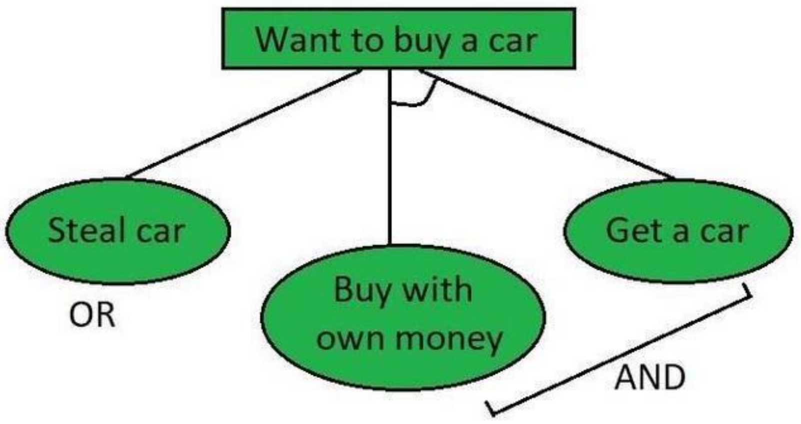
- **Admissible:** the first condition requires for optimality is that h(n) should be an admissible heuristic for A* tree search. An admissible heuristic is optimistic in nature.
- **Consistency:** Second required condition is consistency for only A* graph-search.

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 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 is what the AO* algorithm does. 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 canbe 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.

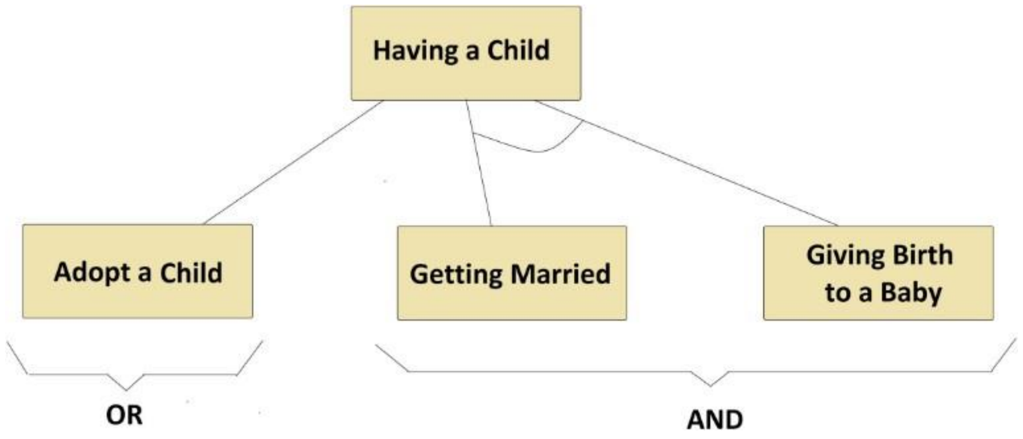


AND-OR Graph

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 subproblems 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.

Example:



In the above example, we can see both tasks of the AND side, which are connected by an AND-ARCS, need to be done to have a child, while the OR side lets having a child just by a single task of adoption.

In general, in a graph for each node, there could be multiple OR and AND sides, where each AND side itself may have multiple successor nodes connected to each other by an AND-ARCS.

4) Working of AO* algorithm:

The evaluation function in AO* looks like this:

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

f(n) = Actual cost + Estimated cost

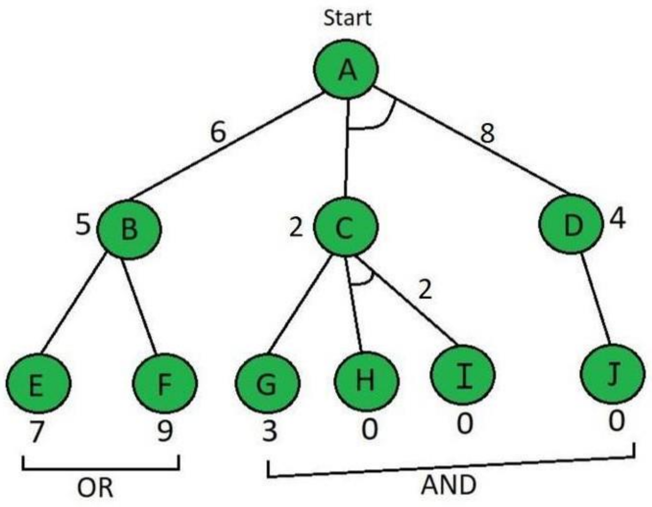
here,

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.**

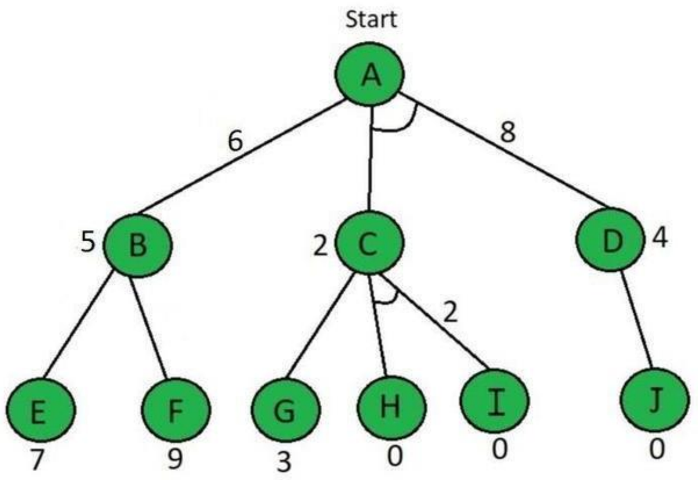
Example:



AO Algorithm – Question tree*

Here in the above example below the Node which is given is the heuristic value i.e **h(n)**. Edge length is considered as **1**.

Step 1



AO Algorithm (Step-1)*

With help of **f(n) = g(n) + h(n)** evaluation function,
Start from node A,

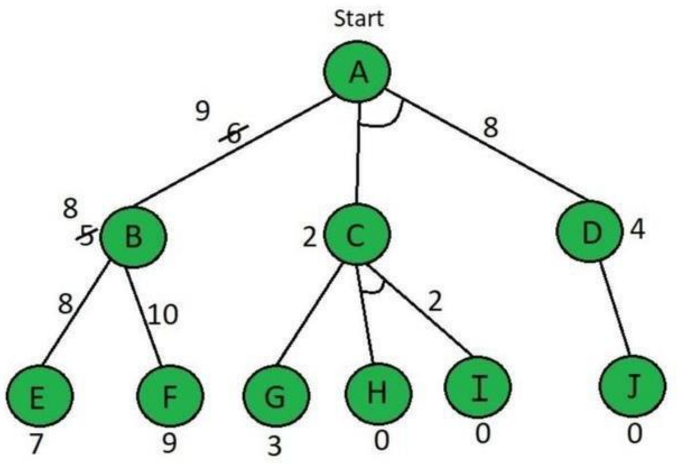
f(A⇢B) = g(B) + h(B)

= 1 + 5 ……here **g(n)=1** is taken by default for path cost
= 6

f(A⇢C+D) = g(c) + h(c) + g(d) + h(d)
= 1 + 2 + 1 + 4 ……here we have added **C & D** because they are in **AND**
= 8

So, by calculation **A⇢B** path is chosen which is the minimum path, i.e **f(A⇢B)**

Step 2



AO Algorithm (Step-2)*

According to the answer of step 1, explore node B

Here the value of E & F are calculated as follows, f(B⇢E) = g(e) + h(e)

f(B⇢E) = 1 + 7

= 8

f(B⇢f) = g(f) + h(f)

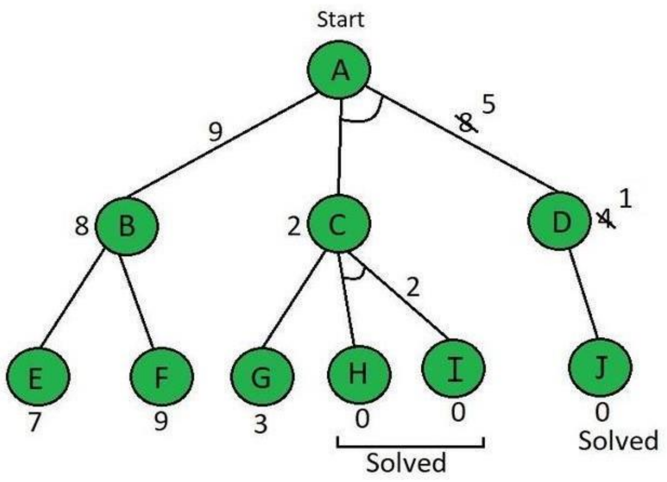
f(B⇢f) = 1 + 9

= 10

So, by above calculation B⇢E path is chosen which is minimum path, i.e **f(B⇢E)** because **B's** heuristic value is different from its actual value The heuristic is updated and the minimum cost path is selected. The minimum value in our situation is **8**. Therefore, the heuristic for **A** must be updated due to the change in **B's** heuristic. So we need to calculate it again.

f(A⇢B) = g(B) + updated h(B)
= 1 + 8
= 9
We have Updated all values in the above tree.

Step 3



AO Algorithm (Step-3) -Geeksforgeeks*

By comparing **f(A⇢B)** & **f(A⇢C+D)**
f(A⇢C+D) is shown to be **smaller**. i.e 8 < 9
Now explore f(A⇢C+D)
So, the current node is **C**

f(C⇢G) = g(g) + h(g)
f(C⇢G) = 1 + 3
= 4

f(C⇢H+I) = g(h) + h(h) + g(i) + h(i)

f(C⇢H+I) = 1 + 0 + 1 + 0 ……here we have added **H & I** because they are in **AND**
= 2

f(C⇢H+I) is selected as the path with the lowest cost and the heuristic is also left unchanged because it matches the actual cost. Paths H & I are solved because the heuristic for those paths is **0,** but Path **A⇢D** needs to be calculated because it has an **AND**.

f(D⇢J) = g(j) + h(j)
f(D⇢J) = 1 + 0
= 1
the heuristic of node D needs to be updated to 1.

f(A⇢C+D) = g(c) + h(c) + g(d) + h(d)
= 1 + 2 + 1 + 1
= 5

as we can see that path **f(A⇢C+D)** is get solved and this **tree has become a solved tree** now.
In simple words, the main flow of this algorithm is that we have to find **firstly level 1st** heuristic value and **then level 2nd** and after that **update the values** with going **upward** means towards the root node.
In the above tree diagram, we have updated all the values.

5) Hill Climbing Algorithm in Artificial Intelligence

- Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value 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.
- Hill climbing algorithm is a technique which is used for optimizing the mathematical problems. One of the widely discussed examples of Hill climbing algorithm is Traveling-salesman Problem in which we need to minimize the distance traveled by the salesman.
- It is also called greedy local search as it only looks to its good immediate neighbor state and not beyond that.
- A node of hill climbing algorithm has two components which are state and value.
- Hill Climbing is mostly used when a good heuristic is available.
- In this algorithm, we don't need to maintain and handle the search tree or graph as it only keeps a single current state.

Features of Hill Climbing:

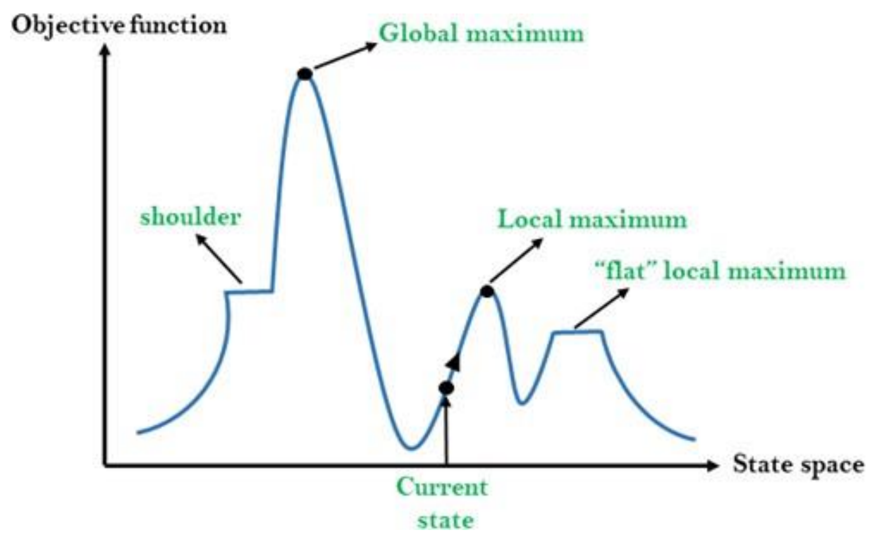
Following are some main features of Hill Climbing Algorithm:

- **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 landscape 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, and state-space on the x-axis. If the function on Y-axis is cost then, the goal of search is to find the global minimum and local minimum. If the function of Y-axis is Objective function, then the goal of the search is to find the global maximum and local maximum.



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.

Types of Hill Climbing Algorithm:

- Simple hill Climbing:
- Steepest-Ascent hill-climbing:
- Stochastic hill Climbing:

1. Simple Hill Climbing:

Simple hill climbing is the simplest way to implement a hill climbing algorithm. **It only evaluates the neighbor node state at a time and selects the first one which optimizes current cost and set it as a current state**. It only checks it's one successor state, and if it finds better than the current state, then move else be in the same state. This algorithm has the following features:

- Less time consuming
- Less optimal solution and the solution is not guaranteed

Algorithm for Simple Hill Climbing:

- **Step 1:** Evaluate the initial state, if it is 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:
    1. If it is goal state, then return success and quit.
    2. Else if it is better than the current state then assign new state as a current state.
    3. Else if not better than the current state, then return to step2.
- **Step 5:** Exit.

2. Steepest-Ascent hill climbing:

The steepest-Ascent algorithm is a variation of simple hill climbing algorithm. This algorithm examines all the neighboring nodes of the current state and selects one neighbor node which is closest to the goal state. This algorithm consumes more time as it searches for multiple neighbors

Algorithm for Steepest-Ascent hill climbing:

- **Step 1:** Evaluate the initial state, if it is goal state then return success and stop, else make current state as initial state.
- **Step 2:** Loop until a solution is found or the current state does not change.
    1. Let SUCC be a state such that any successor of the current state will be better than it.
    2. For each operator that applies to the current state:
        I. Apply the new operator and generate a new state.

II. Evaluate the new state.

III. If it is goal state, then return it and quit, else compare it to the SUCC.

IV. If it is better than SUCC, then set new state as SUCC.

V. If the SUCC is better than the current state, then set current state to SUCC.

Step 5: Exit.

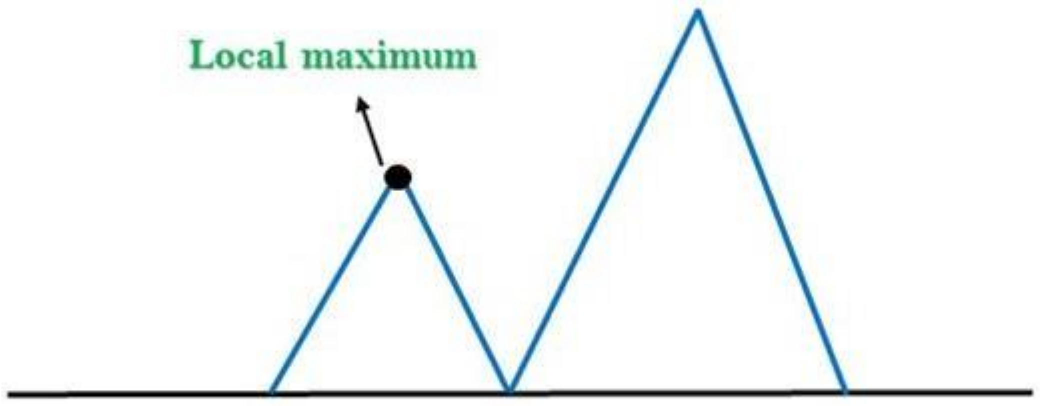
3. Stochastic hill climbing:

Stochastic hill climbing does not examine for all its neighbor before moving. Rather, this search algorithm selects one neighbor node at random and decides whether to choose it as a current state or examine another state.

Problems in Hill Climbing Algorithm:

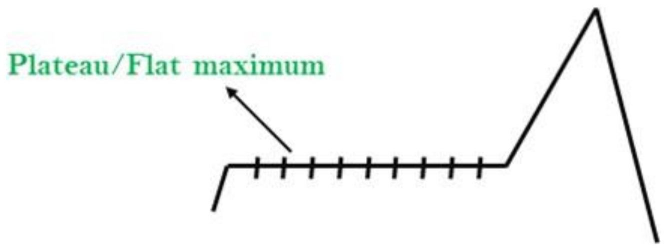
1. Local Maximum: A local maximum is a peak state in the landscape which is better than each of its neighboring states, but there is another state also present which is higher than the local maximum.

Solution: Backtracking technique can be a solution of the local maximum in state space landscape. Create a list of the promising path so that the algorithm can backtrack the search space and explore other paths as well.



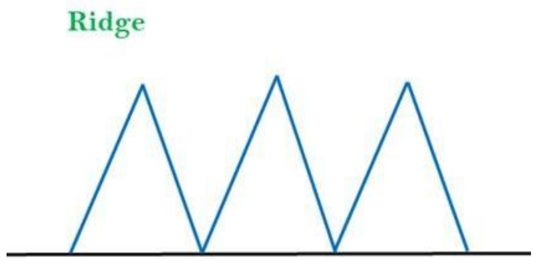
2. Plateau: A plateau is the flat area of the search space in which all the neighbor states of the current state contains the same value, because of this algorithm does not find any best direction to move. A hill-climbing search might be lost in the plateau area.

Solution: The solution for the plateau is to take big steps or very little steps while searching, to solve the problem. Randomly select a state which is far away from the current state so it is possible that the algorithm could find non-plateau region.



3. Ridges: A ridge is a special form of the local maximum. It has an area which is higher than its surrounding areas, but itself has a slope, and cannot be reached in a single move.

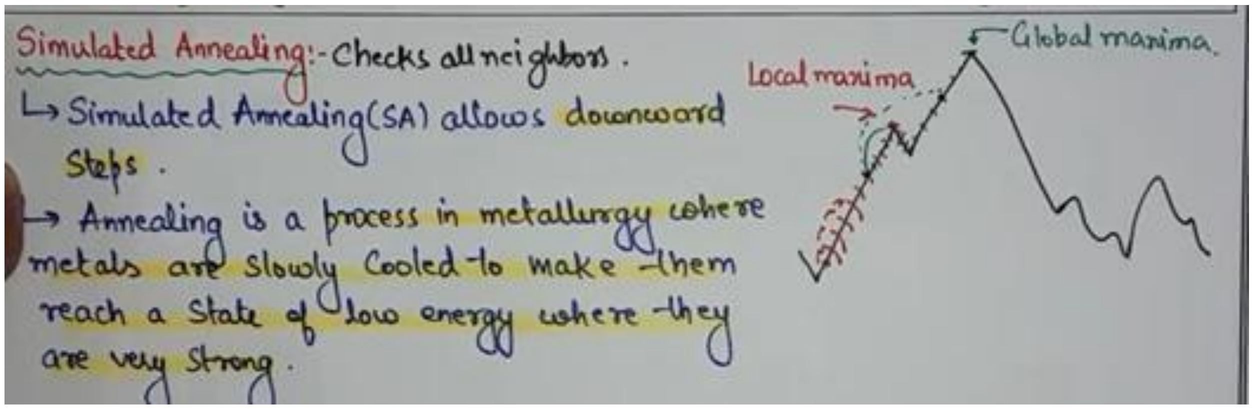
Solution: With the use of bidirectional search, or by moving in different directions, we can improve this problem.



Simulated Annealing:

A hill-climbing algorithm which never makes a move towards a lower value guaranteed to be incomplete because it can get stuck on a local maximum. And if algorithm applies a random walk, by moving a successor, then it may complete but not efficient. **Simulated Annealing** is an algorithm which yields both efficiency and completeness.

In mechanical term **Annealing** is a process of hardening a metal or glass to a high temperature then cooling gradually, so this allows the metal to reach a low-energy crystalline state. The same process is used in simulated annealing in which the algorithm picks a random move, instead of picking the best move. If the random move improves the state, then it follows the same path. Otherwise, the algorithm follows the path which has a probability of less than 1 or it moves downhill and chooses another path.


Simulated Annealing:- Checks all neighbors.

↳ Simulated Annealing (SA) allows downward Steps.

→ Annealing is a process in metallurgy where metals are slowly cooled to make them reach a State of low energy where they are very Strong.

Global maxima
Local maxima

Parameters	**Hill Climbing**	**Simulated Annealing**
Introduction	Hill Climbing is a heuristic optimization process that iteratively advances towards a better solution at each step in order to find the best solution in a given search space.	Simulated Annealing is a probabilistic optimization algorithm that simulates the metallurgical annealing process in order to discover the best solution in a given search area by accepting less-than-idealsolutions with a predeterminedprobability.
Objective	By iteratively progressing towards a better solution at each stage, Hill Climbing seeks to locate the ideal solution within a predetermined search space.	Simulated annealing seeks the global optimum in a given search space by accepting poorer answers with a predetermined probability. This allows it to bypass local optimum conditions.
Strategy	In order to iteratively move towards the best answer at each stage, Hill Climbing employs a greedy method. It only accepts solutions that are superior to the ones already in place.	Simulated annealing explores the search space and avoids local optimum by employing a probabilistic method to accept a worse solution with a given probability. As the algorithm advances,the likelihood of accepting an inferior answer diminishes.
Local vs. Global Optima	Hill Climbing may not locate the global optimum because it is susceptible to becoming caught in local optima.	Simulated annealing has a chance of escaping the local optimum and locating the global optimum.
Stopping Criteria	Hill Climbing comes to an end after a certain number of iterations or when it achieves a local optimum.	When the temperature hits a predetermined level or the maximum number of repetitions, simulated annealing comes to an end.
Performance	Hill climbing is quick and easy, but it has the potential to become locked in local optima and miss the overall best solution.	Simulated annealing is more efficient at locating the global optimum than Hill Climbing, particularly for complicated situations with numerous local optima.

Parameters	Hill Climbing	Simulated Annealing
		Simulated annealing is slower than Hill Climbing.
Tuning Parameters	Hill Climbing has no tuning parameters.	The beginning temperature, cooling schedule, and acceptance probability function are only a few of the tuning factors for Simulated Annealing.
Applications	Many different applications, including image processing, machine learning, and gaming, use hill climbing.	Several fields, including logistics, scheduling, and circuit design, use simulated annealing

Search vs. Uninformed Search,

Parameters	Informed Search	Uninformed Search
Known as	It is also known as Heuristic Search.	It is also known as Blind Search.
Using Knowledge	It uses knowledge for the searching process.	It doesn’t use knowledge for the searching process.
Performance	It finds a solution more quickly.	It finds solution slow as compared to an informed search.
Completion	It may or may not be complete.	It is always complete.
Cost Factor	Cost is low.	Cost is high.

Parameters	Informed Search	Uninformed Search
Time	It consumes less time because of quick searching.	It consumes moderate time because of slow searching.
Direction	There is a direction given about the solution.	No suggestion is given regarding the solution in it.
Implementation	It is less lengthy while implemented.	It is more lengthy while implemented.
Efficiency	It is more efficient as efficiency takes into account cost and performance. The incurred cost is less and speed of finding solutions is quick.	It is comparatively less efficient as incurred cost is more and the speed of finding the Breadth-Firstsolution is slow.
Computational requirements	Computational requirements are lessened.	Comparatively higher computational requirements.
Size of search problems	Having a wide scope in terms of handling large search problems.	Solving a massive search task is challenging.
Examples of Algorithms	• Greedy Search • A* Search • AO* Search • Hill Climbing Algorithm	• Depth First Search (DFS) • Breadth First Search (BFS) • Branch and Bound

Constraint Satisfaction Problem

Overview

Constraint Satisfaction Problem (CSP) is a fundamental topic in artificial intelligence (AI) that deals with solving problems by identifying constraints and finding solutions that satisfy those constraints.

CSP has a wide range of applications, including scheduling, resource allocation, and automated reasoning.

Introduction

The goal of AI is to create intelligent machines that can perform tasks that usually require human intelligence, such as reasoning, learning, and problem-solving. One of the key approaches in AI is the use of constraint satisfaction techniques to solve complex problems.

CSP is a specific type of problem-solving approach that involves identifying constraints that must be satisfied and finding a solution that satisfies all the constraints. CSP has been used in a variety of applications, including scheduling, planning, resource allocation, and automated reasoning.

Finding a solution that meets a set of constraints is the goal of constraint satisfaction problems (CSPs), a type of AI issue. Finding values for a group of variables that fulfill a set of restrictions or rules is the aim of constraint satisfaction problems. For tasks including resource allocation, planning, scheduling, and decision-making, CSPs are frequently employed in AI.

There are mainly three basic components in the constraint satisfaction problem:

Variables: The things that need to be determined are variables. Variables in a CSP are the objects that must have values assigned to them in order to satisfy a particular set of constraints. Boolean, integer, and categorical variables are just a few examples of the various types of variables Variables, for instance, could stand in for the many puzzle cells that need to be filled with numbers in a sudoku puzzle.

Domains: The range of potential values that a variable can have is represented by domains. Depending on the issue, a domain may be finite or limitless. For instance, in Sudoku, the set of numbers from 1 to 9 can serve as the domain of a variable representing a problem cell.

Constraints: The guidelines that control how variables relate to one another are known as constraints. Constraints in a CSP define the ranges of possible values for variables. Unary constraints, binary constraints, and higher-order constraints are only a few examples of the various sorts of constraints. For instance, in a sudoku problem, the restrictions might be that each row, column, and 3×3 box can only have one instance of each number from 1 to 9.

Constraint Satisfaction Problems (CSP) representation:

- The finite set of variables V_1, V_2, V_3 ………………… V_n.
- Non-empty domain for every single variable D_1, D_2, D_3 ………………… D_n.
- The finite set of constraints C_1, C_2 …………, Cm.
    - where each constraint C_i restricts the possible values for variables,
        - e.g., $V_1 \neq V_2$
    - Each constraint C_i is a pair <scope, relation>
        - Example: <(V_1, V_2), V_1 not equal to V_2>
    - Scope = set of variables that participate in constraint.
    - Relation = list of valid variable value combinations.
        - There might be a clear list of permitted combinations. Perhaps a relation that is abstract and that allows for membership testing and listing.

A State-space

Solving a CSP typically involves searching for a solution in the state space of possible assignments to the variables. The **state-space** is a set of all possible configurations of variable assignments, each of which is a potential solution to the problem. The state space can be searched using various algorithms, including backtracking, forward checking, and local search.

The Notion of the Solution

The **notion of a solution** in CSP depends on the specific problem being solved. In general, a solution is a complete assignment of values to all the variables in a way that satisfies all the constraints. For example, in a scheduling problem, a solution would be a valid schedule that satisfies all the constraints on task scheduling and resource allocation.

Domain Categories within CSP

The domain of a variable in a Constraint satisfaction problem in artificial intelligence can be categorized into three types: finite, infinite, and continuous. **Finite domains** have a finite number of possible values, such as colors or integers. **Infinite domains** have an infinite number of possible values, such as real numbers. **Continuous domains** have an infinite number of possible values, but they can be represented by a finite set of parameters, such as the coefficients of a polynomial function.

In mathematics, a continuous domain is a set of values that can be described as a continuous range of real numbers. This means that there are no gaps or interruptions in the values between any two points in the set.

On the other hand, an infinite domain refers to a set of values that extends indefinitely in one or more directions. It may or may not be continuous, depending on the specific context.

Types of Constraints in CSP

Several types of constraints can be used in a Constraint satisfaction problem in artificial intelligence, including:

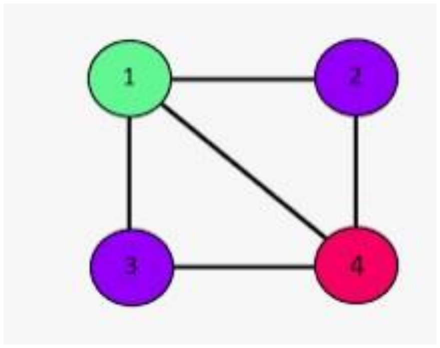
- **Unary Constraints:**
  A unary constraint is a constraint on a single variable. For example, Variable A not equal to “Red”.
- **Binary Constraints:**
  A binary constraint involves two variables and specifies a constraint on their values. For example, a constraint that two tasks cannot be scheduled at the same time would be a binary constraint.
- **Global Constraints:**
  Global constraints involve more than two variables and specify complex relationships between them. For example, a constraint that no two tasks can be scheduled at the same time if they require the same resource would be a global constraint.

Constraint Satisfaction Problems (CSP) algorithms:

- The **backtracking algorithm** is a depth-first search algorithm that methodically investigates the search space of potential solutions up until a solution is discovered that satisfies all the restrictions. The method begins by choosing a variable and giving it a value before repeatedly attempting to give values to the other variables. The method returns to the prior variable and tries a different value if at any time a variable cannot be given a value that fulfills the requirements. Once all assignments have been tried or a solution that satisfies all constraints has been discovered, the algorithm ends.
- The **forward-checking algorithm** is a variation of the backtracking algorithm that condenses the search space using a type of local consistency. For each unassigned variable, the method keeps a list of remaining values and applies local constraints to eliminate inconsistent values from these sets. The algorithm examines a variable's neighbors after itis given a value to see whether any of its remaining values become inconsistent and removesthem from the sets if they do. The algorithm goes backward if, after forward checking, a variable has no more values.
- Algorithms for **propagating constraints** are a class that uses local consistency and inference to condense the search space. These algorithms operate by propagating restrictionsbetween variables and removing inconsistent values from the variable domains using the information obtained.

Example NODE Coloring;

Variable= {1,2,3,4}
Domain= {RED, GREEN, BLUE}
Constraints={Adjacent node should not have same color}



Solution:

Node	1	2	3	4
Initial state	RGB	RGB	RGB	RGB
1=R	R	GB	GB	GB

2=G	R	G	GB	B
3=B	R	G	B	Conflict/error so intelligent backtrack to follow an and change the values
3=G	R	G	G	B

Crypt-arithmetic problem

In the crypt-arithmetic problem, some letters are used to assign digits to it. Like ten different letters are holding digit values from 0 to 9 to perform arithmetic operations correctly. There are two words are given and another word is given an answer of addition for those two words.

As an example, we can say that two words 'BASE' and 'BALL', and the result is 'GAMES'. Now if we try to add BASE and BALL by their symbolic digits, we will get the answer GAMES.

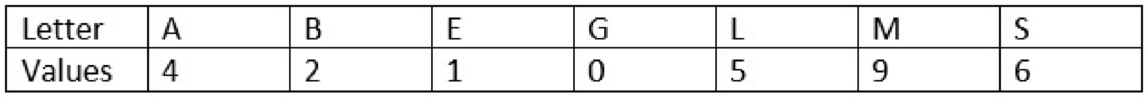
NOTE & minus; There must be ten letters maximum, otherwise it cannot be solved.

Input and Output

Input:
This algorithm will take three words.

```
  B A S E
  B A L L
-------
G A M E S
```

Output:
It will show which letter holds which number from 0 – 9.
For this case it is like this.



Letter	A	B	E	G	L	M	S
Values	4	2	1	0	5	9	6

```
  B A S E          2 4 6 1
  B A L L          2 4 5 5
-----------------------------------------
 G A M E S         0 4 9 1 6
```

Cryptarithmetic Problem
Cryptarithmetic Problem is a type of constraint satisfaction problem where the game is about digits and its unique replacement either with alphabets or other symbols. In **cryptarithmetic problem,** the digits (0-9) get substituted by some possible alphabets or symbols. The task in cryptarithmetic problem is to substitute each digit with an alphabet to get the result arithmetically correct.
We can perform all the arithmetic operations on a given cryptarithmetic problem.

The rules or constraints on a cryptarithmetic problem are as follows:

- There should be a unique digit to be replaced with a unique alphabet.
- The result should satisfy the predefined arithmetic rules, i.e., 2+2 =4, nothing else.
- Digits should be from **0-9** only.
- There should be only one carry forward, while performing the addition operation on a problem.
- The problem can be solved from, i.e., **left-hand side (L.H.S)**
- *Note: it can also solve from right hand side but require more combinations.*

Let's understand the cryptarithmetic problem as well its constraints better with the help of an example:

- Given a cryptarithmetic problem, i.e., **S E N D + M O R E = M O N E Y**

$$\begin{array}{r} S\,E\,N\,D \\ +M\,O\,R\,E \\ \hline M\,O\,N\,E\,Y \\ \hline \end{array}$$

In this example, add both terms **S E N D** and **M O R E** to bring **M O N E Y** as a result.
Follow the below steps to understand the given problem by breaking it into its subparts:

- Starting from the left hand side (L.H.S) , the terms are **S** and **M**. Assign a digit which could give a satisfactory result. Let's assign **S->9** and **M->1**.

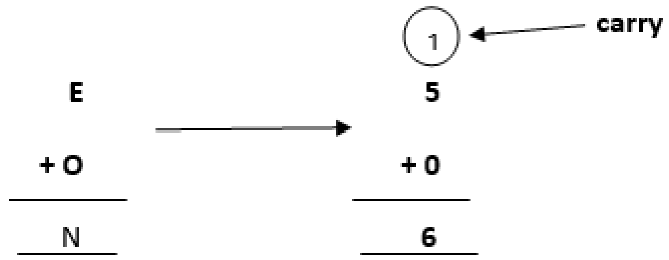
$$\begin{array}{r} S \\ +M \\ \hline M\,O \\ \hline \end{array} \quad \longrightarrow \quad \begin{array}{r} 9 \\ +1 \\ \hline 1\,0 \\ \hline \end{array}$$

Hence, we get a satisfactory result by adding up the terms and got an assignment for **O** as **O->0** as well.

- Now, move ahead to the next terms **E** and **O** to get **N** as its output.

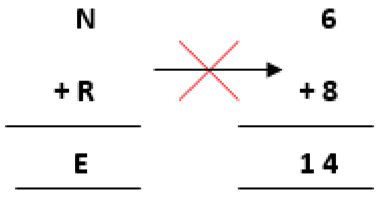
$$\begin{array}{r} E \\ +O \\ \hline N \\ \hline \end{array} \quad \cancel{\longrightarrow} \quad \begin{array}{r} 5 \\ +0 \\ \hline 5 \\ \hline \end{array}$$

Adding E and O, which means 5+0=0, which is not possible because according to cryptarithmetic constraints, we cannot assign the same digit to two letters. So, we need to think more and assign some other value.

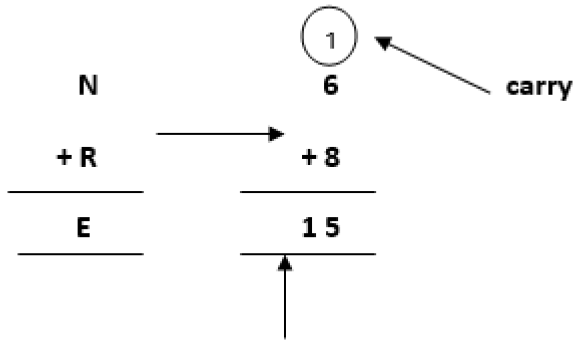


Note: When we will solve further, we will get one carry, so after applying it, the answer will be satisfied.

- Further, adding the next two terms **N** and **R** we get,

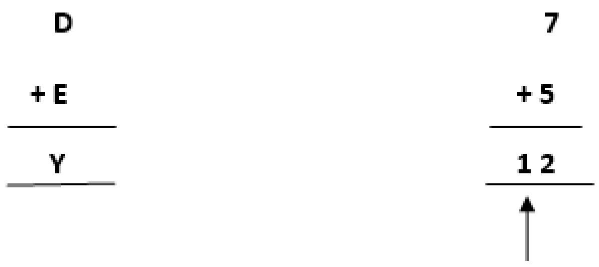


But, we have already assigned **E->5**. Thus, the above result does not satisfy the values because we are getting a different value for **E.** So, we need to think more.
Again, after solving the whole problem, we will get a carryover on this term, so our answer will be satisfied.



where 1 will be carry forward to the above term
Let's move ahead.

- Again, on adding the last two terms, i.e., the rightmost terms **D** and **E**, we get **Y** as its result.



where 1 will be carry forward to the above term

- Keeping all the constraints in mind, the final resultant is as follows:

```
  S E N D
+ M O R E
---------
M O N E Y
---------
```

- Below is the representation of the assignment of the digits to the alphabets.

S	9
E	5
N	6
D	7
M	1
O	0
R	8
y	2

More examples of cryptarithmetic problems can be:

1.

```
  B A S E
+ B A L L
---------
G A M E S
---------
```

$\longrightarrow$

B	7
A	4
S	8
E	3
L	5
G	1
M	9

2.

```
  Y O U R
 + Y O U
---------
H E A R T
```

$\longrightarrow$

Y	9
O	4
U	2
R	6
H	1
E	0
A	3
T	8

Means-Ends Analysis in Artificial Intelligence

- We have studied the strategies which can reason either in forward or backward, but a mixture of the two directions is appropriate for solving a complex and large problem. Such a mixed strategy, make it possible that first to solve the major part of a problem and then go back and solve the small problems arise during combining the big parts of the problem. Such a technique is called **Means-Ends Analysis**.
- Means-Ends Analysis is problem-solving techniques used in Artificial intelligence for limiting search in AI programs.
- It is a mixture of Backward and forward search technique.

- The MEA technique was first introduced in 1961 by Allen Newell, and Herbert A. Simon in their problem-solving computer program, which was named as General Problem Solver (GPS).
- The MEA analysis process centered on the evaluation of the difference between the current state and goal state.

How means-ends analysis Works:

The means-ends analysis process can be applied recursively for a problem. It is a strategy to control search in problem-solving. Following are the main Steps which describes the working of MEA technique for solving a problem.

a) First, evaluate the difference between Initial State and final State.

b) Select the various operators which can be applied for each difference.

c) Apply the operator at each difference, which reduces the difference between the current state and goal state.

Operator Subgoaling

In the MEA process, we detect the differences between the current state and goal state. Once these differences occur, then we can apply an operator to reduce the differences. But sometimes it is possible that an operator cannot be applied to the current state. So we create the subproblem of the current state, in which operator can be applied, such type of backward chaining in which operators are selected, and then sub goals are set up to establish the preconditions of the operator is called **Operator Subgoaling**.

Algorithm for Means-Ends Analysis:

Let's we take Current state as CURRENT and Goal State as GOAL, then following are the steps for the MEA algorithm.

- **Step 1:** Compare CURRENT to GOAL, if there are no differences between both then return Success and Exit.
- **Step 2:** Else, select the most significant difference and reduce it by doing the following steps until the success or failure occurs.

1. Select a new operator O which is applicable for the current difference, and if there is no such operator, then signal failure.

    2. Attempt to apply operator O to CURRENT. Make a description of two states.

       i) O-Start, a state in which O?s preconditions are satisfied.

       ii) O-Result, the state that would result if O were applied In O-start.

3. If

   (First-Part <------ MEA (CURRENT, O-START)

   And

   (LAST-Part <----- MEA (O-Result, GOAL), are successful, then signal Success and return the result of combining FIRST-PART, O, and LAST-PART.

The above-discussed algorithm is more suitable for a simple problem and not adequate for solving complex problems.

Example of Mean-Ends Analysis:

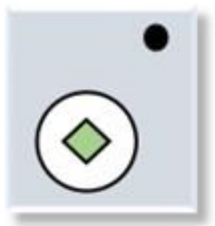
Let's take an example where we know the initial state and goal state as given below. In this problem, we need to get the goal state by finding differences between the initial state and goal state and applying operators.

Solution:

To solve the above problem, we will first find the differences between initial states and goal states, and for each difference, we will generate a new state and will apply the operators. The operators we have for this problem are:

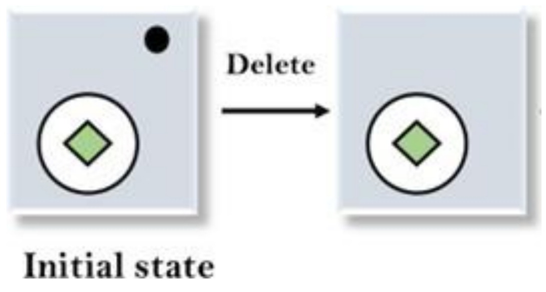
- **Move**
- **Delete**
- **Expand**

1. Evaluating the initial state: In the first step, we will evaluate the initial state and will compare the initial and Goal state to find the differences between both states.

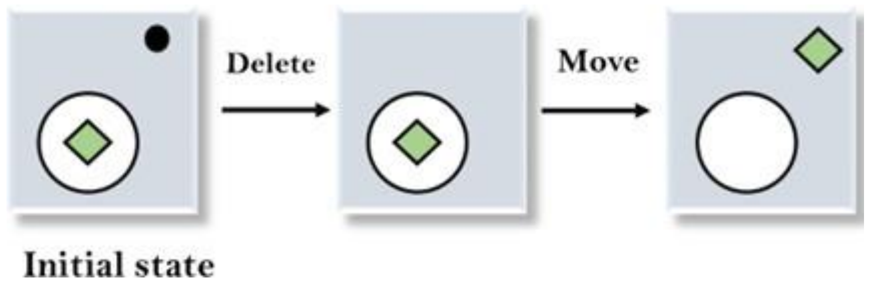


Initial state

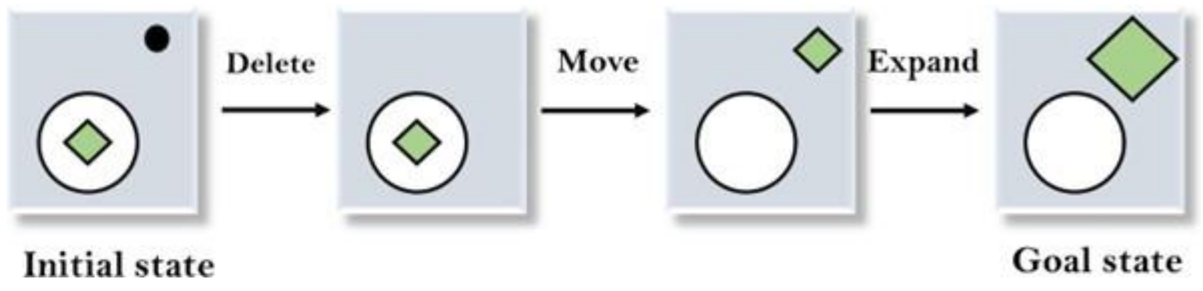
2. Applying Delete operator: As we can check the first difference is that in goal state there is no dot symbol which is present in the initial state, so, first we will apply the **Delete operator** to remove this dot.



3. Applying Move Operator: After applying the Delete operator, the new state occurs which we will again compare with goal state. After comparing these states, there is another difference that is the square is outside the circle, so, we will apply the **Move Operator**.



4. Applying Expand Operator: Now a new state is generated in the third step, and we will compare this state with the goal state. After comparing the states there is still one difference which is the size of the square, so, we will apply **Expand operator**, and finally, it will generate the goal state.



Example: -Want to become AI/web developer then we have to go step by steps. Like learning html, python, CSS, JavaScript, ml, libraries etc…