Unit II

Problem Solving Method

The reflex agent of AI directly maps states into action. Whenever these agents fail to operate in an environment where the state of mapping is too large and not easily performed by the agent, then the stated problem dissolves and sent to a problem-solving domain which breaks the large stored problem into the smaller storage area and resolves one by one. The final integrated action will be the desired outcomes.

On the basis of the problem and their working domain, different types of problem-solving agent defined and use at an atomic level without any internal state visible with a problem-solving algorithm. The problem-solving agent performs precisely by defining problems and several solutions. So we can say that problem solving is a part of artificial intelligence that encompasses a number of techniques such as a tree, B-tree, heuristic algorithms to solve a problem.

We can also say that a problem-solving agent is a result-driven agent and always focuses on satisfying the goals.

There are basically three types of problem in artificial intelligence:

1. Ignorable: In which solution steps can be ignored.

2. Recoverable: In which solution steps can be undone.

3. Irrecoverable: Solution steps cannot be undo.

Steps problem-solving in AI: The problem of AI is directly associated with the nature of humans and their activities. So we need a number of finite steps to solve a problem which makes human easy works.

These are the following steps which require to solve a problem :

- **Problem definition:** Detailed specification of inputs and acceptable system solutions.
- **Problem analysis:** Analyse the problem thoroughly.
- **Knowledge Representation:** collect detailed information about the problem and define all possible techniques.
- **Problem-solving:** Selection of best techniques.

Components to formulate the associated problem:

- **Initial State:** This state requires an initial state for the problem which starts the AI agent towards a specified goal. In this state new methods also initialize problem domain solving by a specific class.
- **Action:** This stage of problem formulation works with function with a specific class taken from the initial state and all possible actions done in this stage.
- **Transition:** This stage of problem formulation integrates the actual action done by the previous action stage and collects the final stage to forward it to their next stage.
- **Goal test:** This stage determines that the specified goal achieved by the integrated transition model or not, whenever the goal achieves stop the action and forward into the next stage to determines the cost to achieve the goal.
- **Path costing:** This component of problem-solving numerical assigned what will be the cost to achieve the goal. It requires all hardware software and human working cost.

Problem State Definition

States

A State is a representation of elements in a given moment.

A problem is defined by its elements and their relations.

At each instant of a problem, the elements have specific descriptors and relations; the descriptors

indicate how to select elements?

Among all possible states, there are two special states called:

- Initial state – the start point
- Final state – the goal state

State Change: Successor Function

A ‘successor function’ is needed for state change. The Successor Function moves one state to another state.

Successor Function:

- It is a description of possible actions; a set of operators.
- It is a transformation function on a state representation, which converts that state into another state.
- It defines a relation of accessibility among states.
- It represents the conditions of applicability of a state and corresponding transformation function.

State space

A state space is the set of all states reachable from the initial state.

- A state space forms a graph (or map) in which the nodes are states and the arcs between nodes are actions.
- In a state space, a path is a sequence of states connected by a sequence of actions.
- The solution of a problem is part of the map formed by the state space.

Structure of a state space

The structures of a state space are trees and graphs.

- A tree is a hierarchical structure in a graphical form.
- A graph is a non-hierarchical structure.
- A tree has only one path to a given node;

  i.e., a tree has one and only one path from any point to any other point.
- A graph consists of a set of nodes (vertices) and a set of edges (arcs). Arcs establish relationships (connections) between the nodes; i.e., a graph has several paths to a given node.

- The Operators are directed arcs between nodes.

A search process explores the state space. In the worst case, the search explores all possible paths between the initial state and the goal state.

Problem Tree

The problem tree method is a visual tool that helps you identify the core problem, its effects, and its root causes. You start by writing the problem statement in the center of a paper or a board, and then you brainstorm the negative consequences of the problem and write them above the problem statement as branches

It is a tool used during project planning and involves constructing a problem tree to identify potential causes of the problem and potential solutions. It does this by connecting potential causes of the problem with potential solutions

Ex

Root, Left, Right

Tree structure

Tree is a way of organizing objects, related in a hierarchical fashion.

- Tree is a type of data structure in which each element is attached to one or more elements directly beneath it.
- The connections between elements are called branches.
- Tree is often called inverted trees because it is drawn with the root at the top.
- The elements that have no elements below them are called leaves.
- A binary tree is a special type: each element has only two branches below it.

Properties

- Tree is a special case of a graph.
- The topmost node in a tree is called the root node.
- At root node all operations on the tree begin.
- A node has at most one parent.

- The topmost node (root node) has no parents.
- Each node has zero or more child nodes, which are below it .
- The nodes at the bottommost level of the tree are called leaf nodes. Since leaf nodes are at the bottom most level, they do not have children.
- A node that has a child is called the child's parent node.
- The depth of a node n is the length of the path from the root to the node.
- The root node is at depth zero.

Forward and Backward Reasoning in AI

The objective of search in Artificial Intelligence (AI) is to find the path to solve different problems. The search in AI can be executed in two ways namely, **Forward Reasoning** and **Backward Reasoning**. The most basic difference between the two is that forward reasoning starts with the new data to find conclusions, whereas backward reasoning starts with a conclusion to determining the initial data.

Read this article to learn more about **Forward Reasoning** and **Backward Reasoning** and how they are different from each other.

What is Forward Reasoning?

Forward reasoning is a process in artificial intelligence that finds all the possible solutions of a problem based on the initial data and facts. Thus, the forward reasoning is a data-driven task as it begins with new data. The main objective of the forward reasoning in AI is to find a conclusion that would follow. It uses an opportunistic type of approach.

Forward reasoning flows from incipient to the consequence. The inference engine searches the knowledge base with the given information depending on the constraints. The precedence of these constraints have to match the current state.

In forward reasoning, the first step is that the system is given one or more constraints. The rules are then searched for in the knowledge base for every constraint. The rule that fulfils the condition is selected. Also, every rule can generate a new condition from the conclusion which is obtained from the invoked one. This new conditions can be added and are processed again.

The step ends if no new conditions exist. Hence, we can conclude that forward reasoning follows the top-down approach.

What is Backward Reasoning?

Backward reasoning is the reverse process of the forward reasoning in which a goal or hypothesis is selected and it is analyzed to find the initial data, facts, and rules. Therefore, the backward reasoning is a goal driven task as it begins with conclusions or goals that are uncertain. The main objective of the backward reasoning is to find the facts that support the conclusions.

Backward reasoning uses a conservative type of approach and flows from consequence to the incipient. The system helps to choose a goal state and reasons in a backward direction. The first step in the backward reasoning is that the goal state and rules are selected. Then, sub-goals are made from the selected rule, which need to be satisfied for the goal state to be true.

The initial conditions are set such that they satisfy all the sub-goals. Also, the established states are matched to the initial state provided. If the condition is fulfilled, the goal is the solution, otherwise the goal is rejected. Therefore, backward reasoning follows bottom-up technique.

Backward reasoning is also known as a **decision-driven** or **goal-driven** inference technique because the system selects a goal state and reasons in the backward direction.

Difference between Forward and Backward Reasoning in AI

The following are the important differences between Forward and Backward Reasoning in AI −

S.No.	Forward Reasoning	Backward Reasoning
1.	It is a data-driven task.	It is a goal driven task.
2.	It begins with new data.	It begins with conclusions that are uncertain.
3.	The objective is to find a conclusion that	The objective is to find the facts that

	would follow.	support the conclusions.
4.	It uses an opportunistic type of approach.	It uses a conservative type of approach.
5.	It flows from incipient to the consequence.	It flows from consequence to the incipient.
6.	Forward reasoning begins with the initial facts.	Backward reasoning begins with some goal (hypothesis).
7.	Forward reasoning tests all the rules.	Backward reasons tests some rules.
8.	Forward reasoning is a bottom-up approach.	Backward reasoning is a top-down approach.
9.	Forward reasoning can produce an infinite number of conclusion.	Backward reasoning produces a finite number of conclusions.
10.	In the forward reasoning, all the data is available.	In the backward reasoning, the data is acquired on demand.
11.	Forward reasoning has a small number of initial states but a large number of conclusions.	Backward reasoning has a smaller number of goals and a larger number of rules.
12.	In forward reasoning, the goal formation is difficult.	In backward reasoning, it is easy to form a goal.
13.	Forward reasoning works in forward direction to find all the possible	Backward reasoning work in backward direction to find the facts that justify the

	conclusions from facts.	goal.
14.	Forward reason is suitable to answer the problems such as planning, control, monitoring, etc.	Backward reasoning is suitable for diagnosis like problems.

Heuristic Function In AI

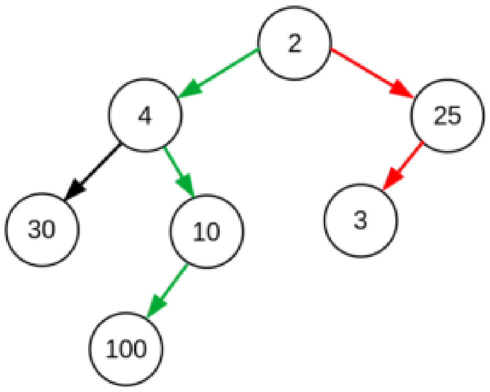
2.1. Definition

A heuristic function (algorithm) or simply a heuristic is a shortcut to solving a problem when there are no exact solutions for it or the time to obtain the solution is too long.

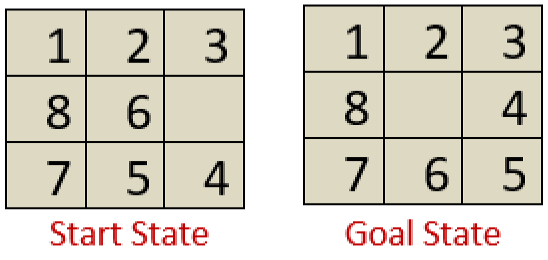
2.2. Speed vs. Accuracy

From the definition, we can conclude that the goal is to find a faster solution or an approximate one, even if it is not optimal. In other words, when using a heuristic, we trade accuracy for the speed of the solution.

For example, greedy algorithms usually produce quick but sub-optimal solutions. A greedy algorithm to find the largest sum in the following tree would go for the red path while the optimal path is the green one:



Consider the following 8-puzzle problem where we have a start state and a goal state. Our task is to slide the tiles of the current/start state and place it in an order followed in the goal state. There can be four moves either **left, right, up, or down**. There can be several ways to convert the current/start state to the goal state, but, we can use a heuristic function h(n) to solve the problem more efficiently.

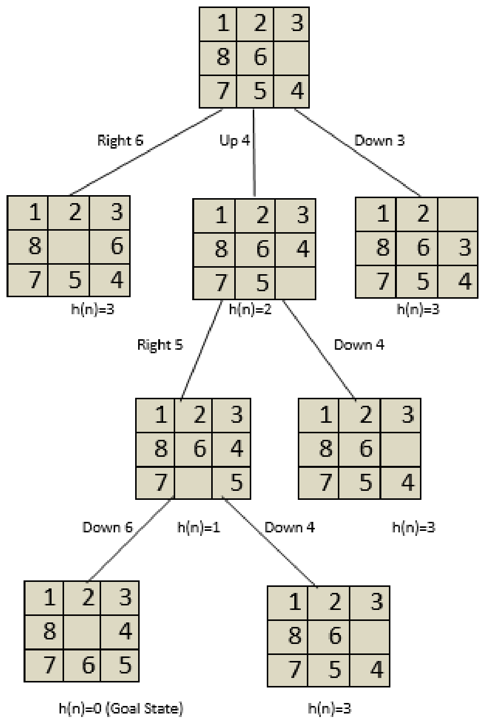


A heuristic function for the 8-puzzle problem is defined below:

h(n)=Number of tiles out of position.

So, there is total of three tiles out of position i.e., 6,5 and 4. Do not count the empty tile present in the goal state). i.e. h(n)=3. Now, we require to minimize the value of **h(n) =0.**

We can construct a state-space tree to minimize the h(n) value to 0, as shown below:



It is seen from the above state space tree that the goal state is minimized from h(n)=3 to h(n)=0. However, we can create and use several heuristic functions as per the reqirement. It is also clear from the above example that a heuristic function h(n) can be defined as the information required to solve a given problem more efficiently. The information can be related to the **nature of the state, cost of transforming from one state to another, goal node characteristics,** etc., which is expressed as a heuristic function.

Depth-first search

A search strategy that extends the current path as far as possible before backtracking to the last choice point and trying the next alternative path is called Depth-first search (DFS).

• This strategy does not guarantee that the optimal solution has been found.

• In this strategy, search reaches a satisfactory solution more rapidly than breadth first, an advantage when the search space is large.

Algorithm

Depth-first search applies operators to each newly generated state, trying to drive directly toward the goal.

1. If the starting state is a goal state, quit and return success.

2. Otherwise, do the following until success or failure is signalled:

a. Generate a successor E to the starting state. If there are no more successors, then signal failure.

b. Call Depth-first Search with E as the starting state.

c. If success is returned signal success; otherwise, continue in the loop.

Advantages

1. Low storage requirement: linear with tree depth.

2. Easily programmed: function call stack does most of the work of maintaining state of the search.

Disadvantages

1. May find a sub-optimal solution (one that is deeper or more costly than the best solution).

2. Incomplete: without a depth bound, may not find a solution even if one exists.

Bounded depth-first search

Depth-first search can spend much time (perhaps infinite time) exploring a very deep path that does not contain a solution, when a shallow solution exists. An easy way to solve this problem is to put a maximum depth bound on the search. Beyond the depth bound, a failure is generated automatically without exploring any deeper.

Problems:

1. It’s hard to guess how deep the solution lies.

2. If the estimated depth is too deep (even by 1) the computer time used is dramatically increased, by a factor of bextra.

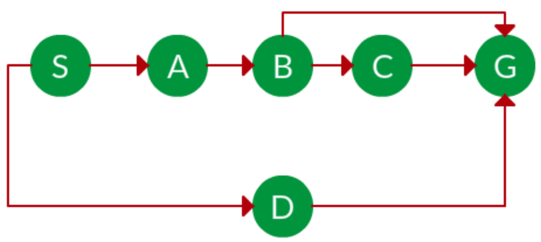
3. If the estimated depth is too shallow, the search fails to find a solution; all that computer time is wasted.

Depth First Search:

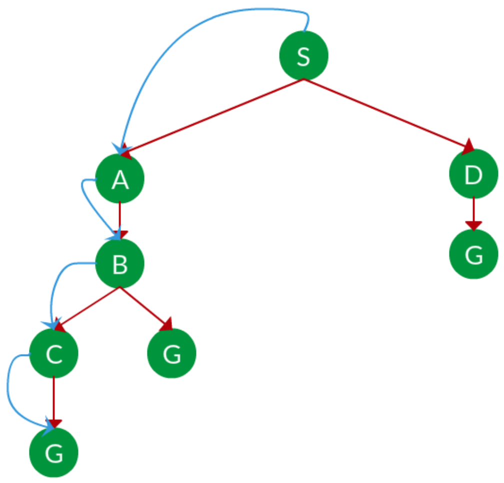
Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. It uses last in- first-out strategy and hence it is implemented using a stack.

Example:

Question. *Which solution would DFS find to move from node S to node G if run on the graph below?*



Solution. The equivalent search tree for the above graph is as follows. As DFS traverses the tree "deepest node first", it would always pick the deeper branch until it reaches the solution (or it runs out of nodes, and goes to the next branch). The traversal is shown in blue arrows.



Path: S -> A -> B -> C -> G

= *the depth of the search tree = the number of levels of the search tree.*

= *number of nodes in level .*

Time complexity: *Equivalent to the number of nodes traversed in DFS.*

Space complexity: *Equivalent to how large can the fringe get.*

Completeness: *DFS is complete if the search tree is finite, meaning for a given finite search tree, DFS will come up with a solution if it exists.*

Optimality: *DFS is not optimal, meaning the number of steps in reaching the solution, or the cost spent in reaching it is high.*

Breadth-first search

A Search strategy, in which the highest layer of a decision tree is searched completely before proceeding to the next layer is called Breadth-first search (BFS).

• In this strategy, no viable solutions are omitted and therefore it is guaranteed that an optimal solution is found.

• This strategy is often not feasible when the search space is large.

Algorithm

1. Create a variable called LIST and set it to be the starting state.

2. Loop until a goal state is found or LIST is empty, Do

a. Remove the first element from the LIST and call it E. If the LIST is empty, quit.

b. For every path each rule can match the state E, Do

(i) Apply the rule to generate a new state.

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

(iii) Otherwise, add the new state to the end of LIST.

Advantages

1. Guaranteed to find an optimal solution (in terms of shortest number of steps to reach the goal).

2. Can always find a goal node if one exists (complete).

Disadvantages

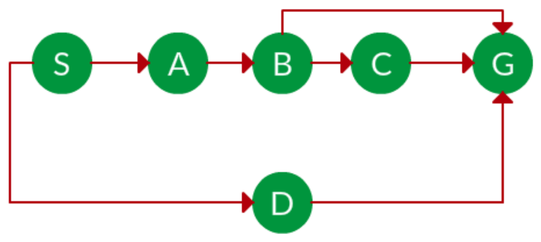
1. High storage requirement: exponential with tree depth.

Breadth First Search:

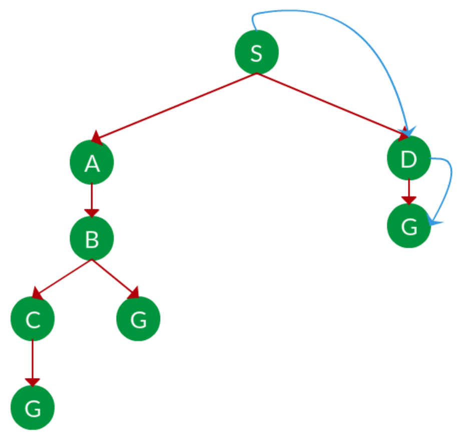
Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a ‘search key’), and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level. It is implemented using a queue.

Example:

Question. Which solution would BFS find to move from node S to node G if run on the graph below?



Solution. The equivalent search tree for the above graph is as follows. As BFS traverses the tree “shallowest node first”, it would always pick the shallower branch until it reaches the solution (or it runs out of nodes, and goes to the next branch). The traversal is shown in blue arrows.



Path: S -> D -> G

= *the depth of the shallowest solution.*

= *number of nodes in level* .

Time complexity: *Equivalent to the number of nodes traversed in BFS until the shallowest solution.*

Space complexity: *Equivalent to how large can the fringe get.*
Completeness: *BFS is complete, meaning for a given search tree, BFS will come up with a solution if it exists.*
Optimality: *BFS is optimal as long as the costs of all edges are equal.*

A* Search Algorithm

A* is a type of search algorithm. Some problems can be solved by representing the world in the initial state, and then for each action we can perform on the world we generate states for what the world would be like if we did so. If you do this until the world is in the state that we specified as a solution, then the route from the start to this goal state is the solution to your problem.

In this tutorial I will look at the use of state space search to find the shortest path between two points (pathfinding), and also to solve a simple sliding tile puzzle (the 8-puzzle). Let's look at some of the terms used in Artificial Intelligence when describing this state space search.

Some terminology

A node is a state that the problem's world can be in. In pathfinding a node would be just a 2d coordinate of where we are at the present time. In the 8-puzzle it is the positions of all the tiles. Next all the nodes are arranged in a graph where links between nodes represent valid steps in solving the problem. These links are known as edges. In the 8-puzzle diagram the edges are shown as blue lines. See figure 1 below.

State space search, then, is solving a problem by beginning with the start state, and then for each node we expand all the nodes beneath it in the graph by applying all the possible moves that can be made at each point.

AO* Search: (And-Or) Graph

The Depth first search and Breadth first search given earlier for OR trees or graphs can be easily adopted by AND-OR graph. The main difference lies in the way termination conditions are determined, since all goals following an AND nodes must be realized; where as a single goal node following an OR node will do. So for this purpose we are using AO* algorithm.

Like A* algorithm here we will use two arrays and one heuristic function.

OPEN:

It contains the nodes that has been traversed but yet not been marked solvable or unsolvable.

CLOSE:

It contains the nodes that have already been processed.

6 7:The distance from current node to goal node.

Algorithm:

Step 1: Place the starting node into OPEN.

Step 2: Compute the most promising solution tree say T0.

Step 3: Select a node n that is both on OPEN and a member of T0. Remove it from OPEN and place it in

CLOSE

Step 4: If n is the terminal goal node then leveled n as solved and leveled all the ancestors of n as solved. If the starting node is marked as solved then success and exit.

Step 5: If n is not a solvable node, then mark n as unsolvable. If starting node is marked as unsolvable, then return failure and exit.

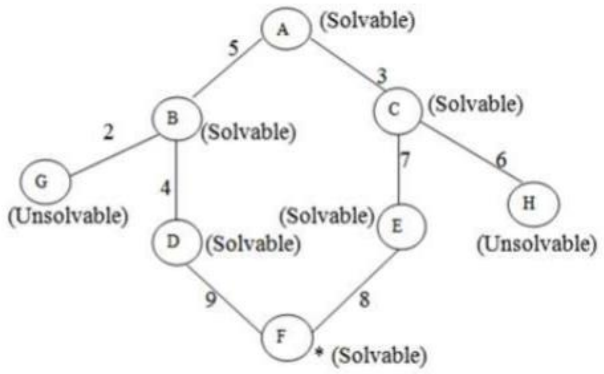
Step 6: Expand n. Find all its successors and find their h (n) value, push them into OPEN.

Step 7: Return to Step 2.

Step 8: Exit.

Implementation:

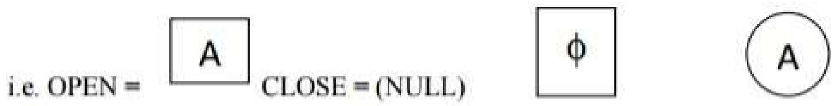
Let us take the following example to implement the AO* algorithm.



Step 1:

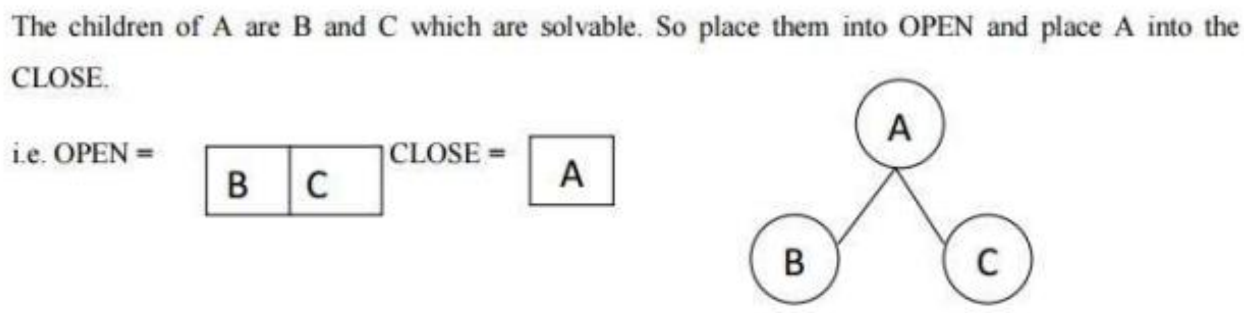
In the above graph, the solvable nodes are A, B, C, D, E, F and the unsolvable nodes are G, H.

Take A as the starting node. So place A into OPEN.



i.e. OPEN = $\boxed{A}$ CLOSE = (NULL) $\boxed{\phi}$

Step 2:



The children of A are B and C which are solvable. So place them into OPEN and place A into the CLOSE.

i.e. OPEN = $\boxed{B \mid C}$ CLOSE = $\boxed{A}$

Step 3

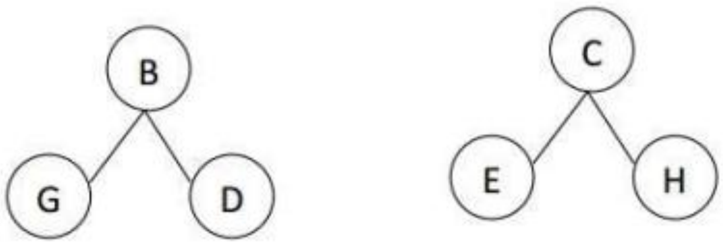
Now process the nodes B and C. The children of B and C are to be placed into OPEN. Also remove B and C from OPEN and place them into CLOSE.

So OPEN =

G	D	E	

C

A	B	C



(O)

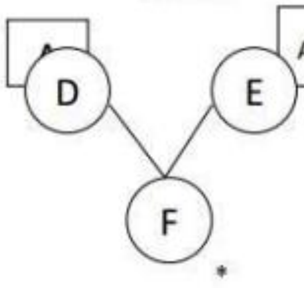
'O' it indicated that the nodes G and H are unsolvable.

Step 4:

As the nodes G and H are unsolvable, so place them into CLOSE directly and process the nodes D and E.

i.e. OPEN = CLOSE =

A	B C		$G^{(O)}$	D E		$H^{(O)}$



Step 5:

Now we have been reached at our goal state. So place F into CLOSE.

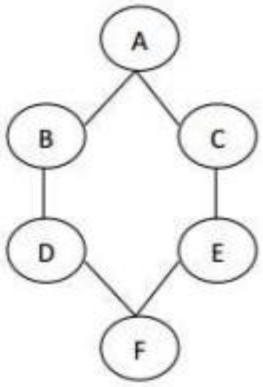
A	B C		$G^{(O)}$	D E		$H^{(O)}$	F

i.e. CLOSE =

Step 6:

Success and Exit

AO* Graph:



Figure

Advantages:

It is an optimal algorithm.

If traverse according to the ordering of nodes. It can be used for both OR and AND graph.

Disadvantages:

Sometimes for unsolvable nodes, it can’t find the optimal path. Its complexity is than other algorithms.