A* AND AO* ALGORITHM

Dr. Krishnendu Guha

Assistant Professor (On Contract)

National Institute of Technology (NIT), Jamshedpur

Email: krishnendu.ca@nitjsr.ac.in

INTRODUCTION TO A* ALGORITHM

- Some problems can be solved by representing the world in the initial state
- 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 we do this until the world is in the state that we desire as a solution, then the route from the start to this goal state is the solution to our problem

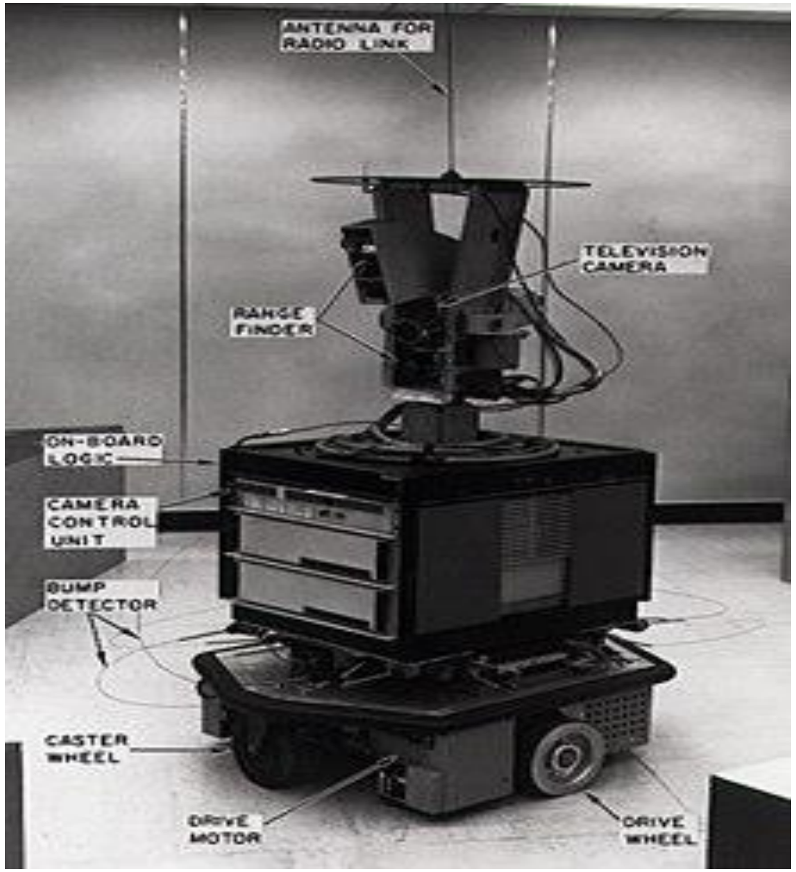
- Why A* is advantageous?
- As the A* algorithm will not only find a path, if there is one, but it will find the shortest path

BRIEF HISTORY

- Peter Hart, Nils Nilsson and Bertram Raphael of Stanford Research Institute (now SRI International) first published the algorithm in 1968

- A* was created as part of the Shakey project, which had the aim of building a mobile robot that could plan its own actions

- Nils Nilsson originally proposed using the Graph Traverser algorithm for Shakey's path planning

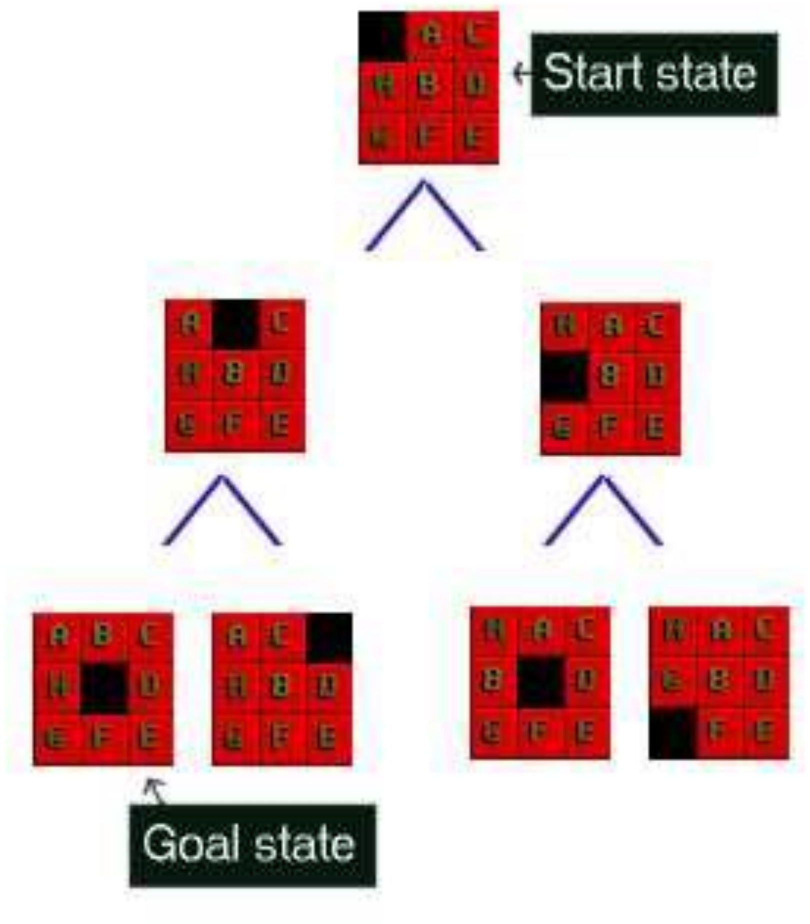


EXPLANATION WITH EXAMPLE

- We consider the 8 puzzle problem
- This is a simple sliding tile puzzle on a 3*3 grid, where one tile is missing and you can move the other tiles into the gap until you get the puzzle into the goal position
- There are 362,880 different states that the puzzle can be in, and to find a solution the search has to find a route through them.

- Here are two states for example;
- The first is a jumbled up example that you may start from.
  - Start state SPACE, A, C, H, B, D, G, F, E
- the last one is the GOAL state, at which point we've found the solution.
  - Goal state A, B, C, H, SPACE, D, G, F, E

- RULE: If there is a blank tile above, below, to the left or to the right of a given tile, then you can move that tile into the space



IMPLEMENTING A*

- What we need to do is start with the start state and then generate the graph downwards from there
- We ask how many moves can we make from the start state?
  - The answer is 2, there are two directions we can move the blank tile, and so our graph expands.
- If we were just to continue blindly generating successors to each node, we could potentially fill the computer's memory before we found the goal node
- Obviously we need to remember the best nodes and search those first
- We also need to remember the nodes that we have expanded already, so that we don't expand the same state repeatedly.

- Let's start with the OPEN list.
- This is where we will remember which nodes we haven't yet expanded.
- When the algorithm begins the start state is placed on the open list, it is the only state we know about and we have not expanded it.
- So we will expand the nodes from the start and put those on the OPEN list too.
- Now we are done with the start node and we will put that on the CLOSED list.
- The CLOSED list is a list of nodes that we have expanded.

- $f = g + h$

- Using the OPEN and CLOSED list lets us be more selective about what we look at next in the search.
- We want to look at the best nodes first.
- We will give each node a score on how good we think it is.
- This score should be thought of as the cost of getting from the node to the goal plus the cost of getting to where we are.
- Traditionally this has been represented by the letters f, g and h.
- 'g' is the sum of all the costs it took to get here,
- 'h' is our heuristic function, the estimate of what it will take to get to the goal.
- 'f' is the sum of these two.
- We will store each of these in our nodes.

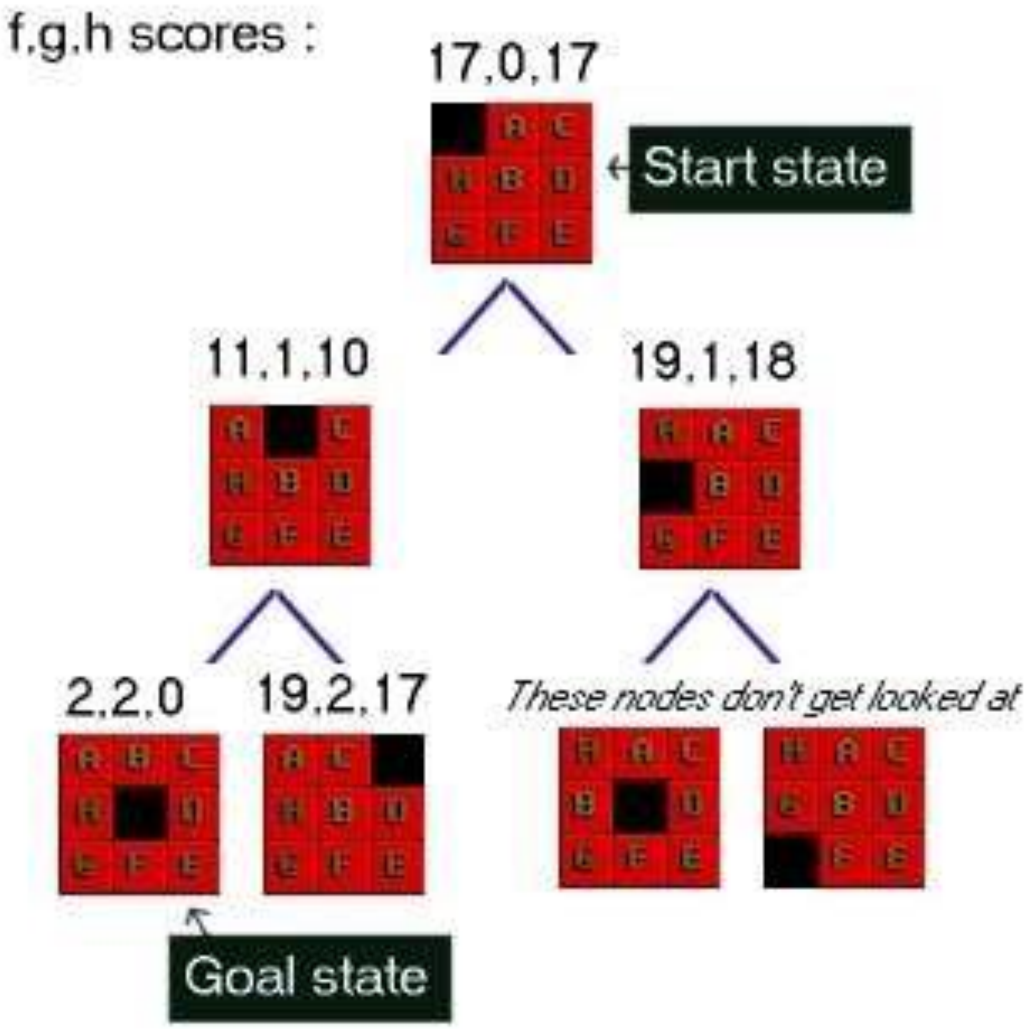
PSEUDOCODE OF A*

```
Create a node containing the goal state node_goal
Create a node containing the start state node_start
Put node_start on the open list
while the OPEN list is not empty
{
Get the node off the open list with the lowest f and call it node_current
if node_current is the same state as node_goal we have found the solution; break from the while loop
    Generate each state node_successor that can come after node_current
    for each node_successor of node_current
    {
        Set the cost of node_successor to be the cost of node_current plus the cost to get to node_successor from node_current
        find node_successor on the OPEN list
        if node_successor is on the OPEN list but the existing one is as good or better then discard this successor and continue
        if node_successor is on the CLOSED list but the existing one is as good or better then discard this successor and continue
        Remove occurences of node_successor from OPEN and CLOSED
        Set the parent of node_successor to node_current
        Set h to be the estimated distance to node_goal (Using the heuristic function)
         Add node_successor to the OPEN list
    }
    Add node_current to the CLOSED list
}
```

- First of all look at the g score for each node.
- This is the cost of what it took to get from the start to that node.
- So in the picture the center number is g, which increases by one at each level.

- Next look at the last number in each triple.
- This is h, the heuristic score, obtained by Nilsson's Sequence, which converges quickly to a correct solution

- **Nilsson's sequence score**
- **A** tile in the center scores 1 (since it should be empty)
- For each tile not in the center, if the tile clockwise to it is not the one that should be clockwise to it then score 2.
- Multiply this sequence by three and finally add the total distance you need to move each tile back to its correct position



ADVANTAGES AND DISADVANTAGES OF A*

- **Advantages:**
- It is complete and optimal.
- It is the best one from other techniques. It is used to solve very complex problems.
- It is optimally efficient, i.e. there is no other optimal algorithm guaranteed to expand fewer nodes than A*.

- **Disadvantages:**
- This algorithm is complete if the branching factor is finite and every action has fixed cost.
- The speed execution of A* search is highly dependent on the accuracy of the heuristic algorithm that is used to compute h (n).

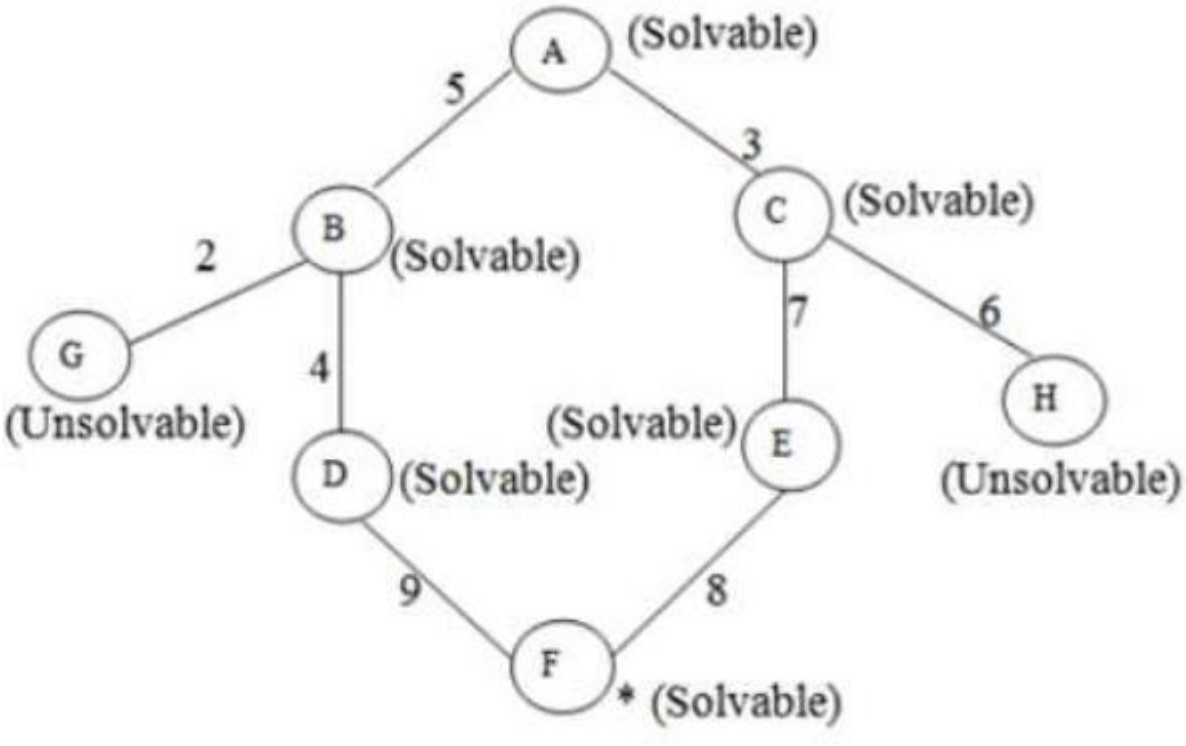
AO* GRAPH
AND-OR GRAPH

- 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; whereas 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.

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 level n as solved and level 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.

EXAMPLE FOR IMPLEMENTATION



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

Step 1:

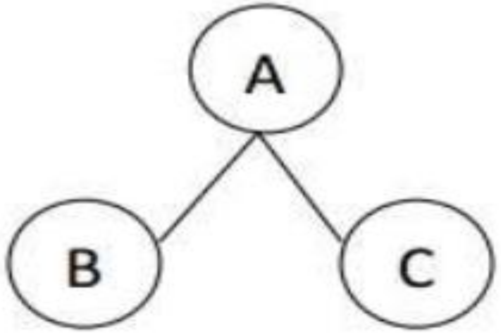
In the above graph, the solvable nodes are A, B, C, D, E, F and the unsolvable nodes are G, H. Take A as the starting node. So place A into OPEN.

i.e. OPEN = [A] CLOSE = (NULL) [ϕ]

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 = [B | C] CLOSE = [A]



Step 3:

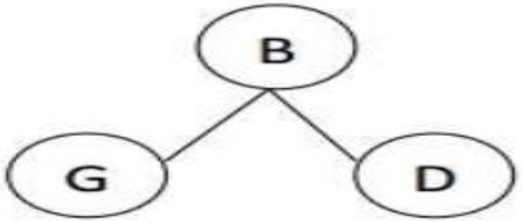
Now process the nodes B and C. The children of B and C are to be placed into OPEN. Also remove B and C from OPEN and place them into CLOSE.

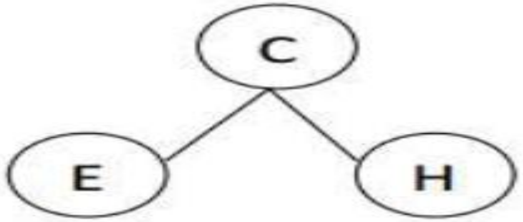
So OPEN =

G	D	E	

C

A	B	C





Step 4:

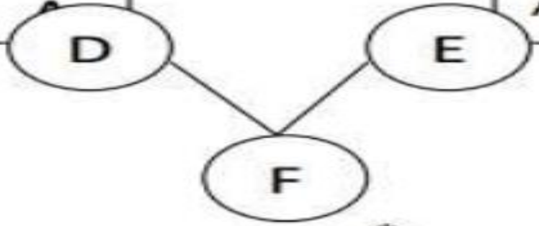
As the nodes G and H are unsolvable, so place them into CLOSE directly and process the nodes D and E.

i.e. OPEN =

A

CLOSE =

A	B	C		G (O)	D	E	H (O)



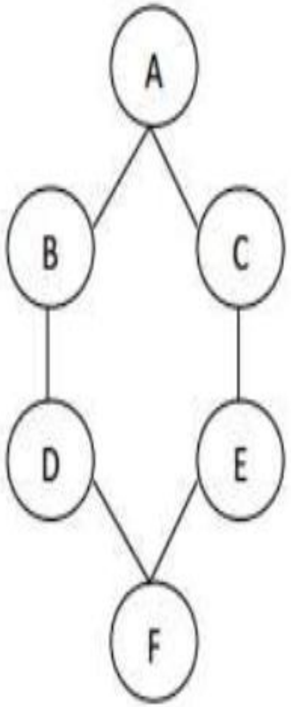
*

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

AO* Graph:



Step 6:

Success and Exit

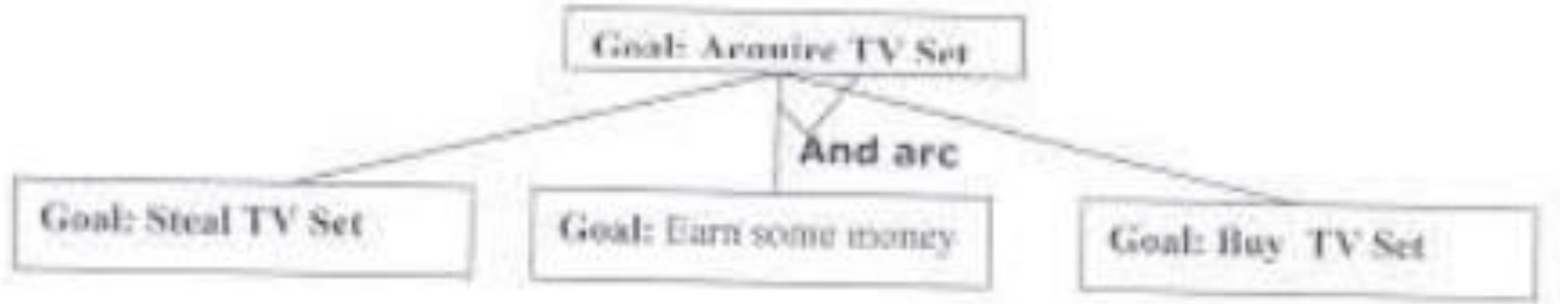
ADVANTAGES AND DISADVANTAGES OF AO*

- **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.

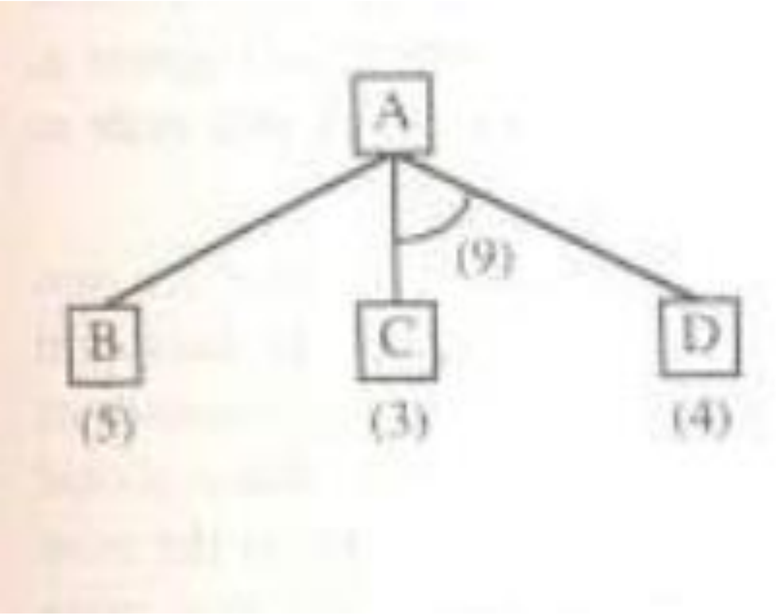
PROBLEM REDUCTION WITH AO*

- When a problem can be divided into a set of sub problems, where each sub problem can be solved separately and a combination of these will be a solution, AND-OR graphs or AND – OR trees are used for representing the solution
- The decomposition of the problem or problem reduction generates AND arcs.
- One AND arc may point to any number of successor nodes
- these must be solved so that the arc will rise to many arcs, indicating several possible solutions.
- Hence the graph is known as AND - OR instead of AND



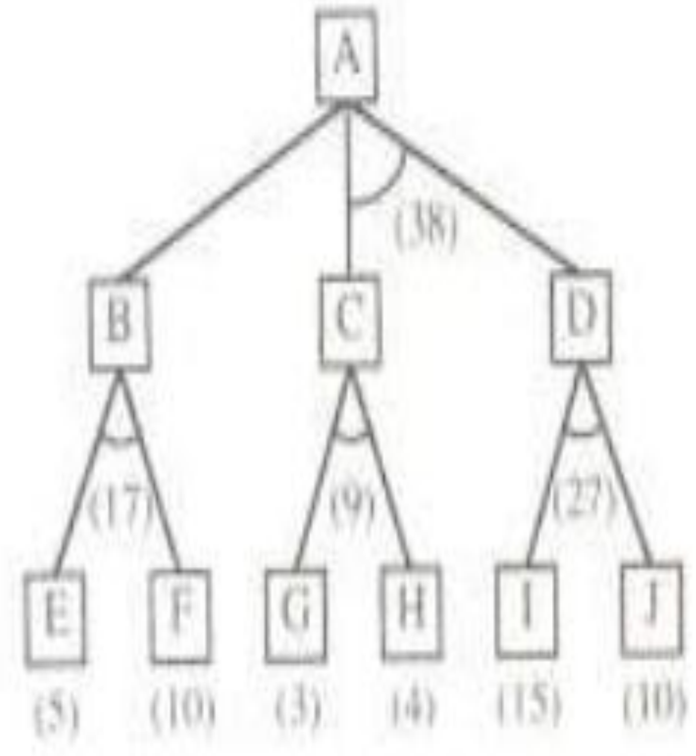
- An algorithm to find a solution in an AND - OR graph must handle AND area appropriately
- A* algorithm can not search AND - OR graphs efficiently

- In figure (a) the top node A has been expanded producing two area one leading to B and leading to C-D .
- the numbers at each node represent the value of f ' at that node (cost of getting to the goal state from current state).
- For simplicity, it is assumed that every operation(i.e. applying a rule) has unit cost, i.e., each are with single successor will have a cost of 1 and each of its components.
- With the available information till now , it appears that C is the most promising node to expand since its f ' = 3 , the lowest but going through B would be better since to use C we must also use D' and the cost would be 9(3+4+1+1). Through B it would be 6(5+1).

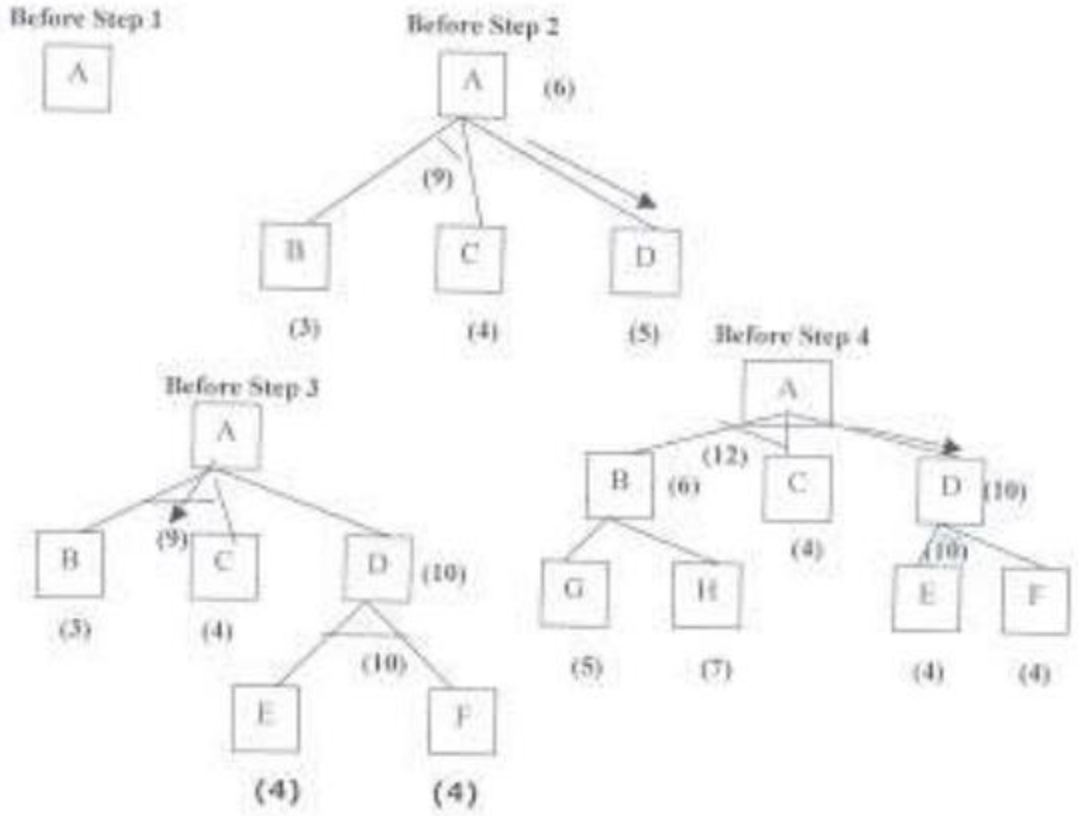


- Thus the choice of the next node to expand depends not only n a value but also on whether that node is part of the current best path form the initial mode

- the node G appears to be the most promising node, with the least f ' value. But G is not on the current best path, since to use G we must use GH with a cost of 9 and again this demands that arcs be used (with a cost of 27).
- The path from A through B, E-F is better with a total cost of (17+1=18).
- Thus we can see that to search an AND-OR graph, the following three things must be done
  - 1. traverse the graph starting at the initial node and following the current best path, accumulate the set of nodes that are on the path and have not yet been expanded.
  - 2. Pick one of these unexpanded nodes and expand it. Add its successors to the graph and compute f ' (cost of the remaining distance) for each of them
  - 3. Change the f ' estimate of the newly expanded node to reflect the new information produced by its successors. Propagate this change backward through the graph. Decide which of the current best path.



- The initial node is expanded and D is Marked initially as promising node.
- D is expanded producing an AND arc E-F.
- f ' value of D is updated to 10.
- Going backwards we can see that the AND arc B-C is better, it is now marked as current best path.
- B and C have to be expanded next. This process continues until a solution is found or all paths have led to dead ends, indicating that there is no solution.
- An A* algorithm the path from one node to the other is always that of the lowest cost and it is independent of the paths through other nodes




Thank
You
Obrigado
Merci
Multumesc
Asante
Spasibo
Maake
Grazie
Kiitos
Kia Ora
Mochchakkeram
Obrigado
Welalin
Matondo
Mamana
Dank Je
Nirringrazzjak
Vinaka
Cam on ban
Arigato
Juspaxar
Chokrane
Ua Tsaug Rau Koj
Salamat
Dankon
Terma Kasih