What we learn

Branch and Bound,

Brute Force,

Best First Search- OR Graphs,

Algorithm A*,

- Admissibility of A*,
- Iterative Deepening A*,
- Algorithm AO*,
- Pruning the CLOSED List,
- Pruning the OPEN List,

Divide and conquer

Beam stacksearch.

Branch and Bound

- Branch and bound is one of the techniques used for problem solving.
- It is similar to the backtracking since it also uses the state space tree.
- It is used for solving the optimization problems and minimization problems.
- If we have given a **maximization problem** then we can convert it using the Branch and bound technique by simply converting the problem into a **maximization problem.**

Basic Concepts

The basic concept underlying the branch-and-bound technique is to : divide and conquer.

Since the original “large” problem is hard to solve directly,

it is divided into smaller and smaller subproblems

until these subproblems can be conquered.

The dividing (branching) is done by partitioning the entire set of

feasible solutions into smaller and smaller subsets.

Basic Concepts

The conquering (fathoming) is done partially by

(i) giving a bound for the best solution in the subset;

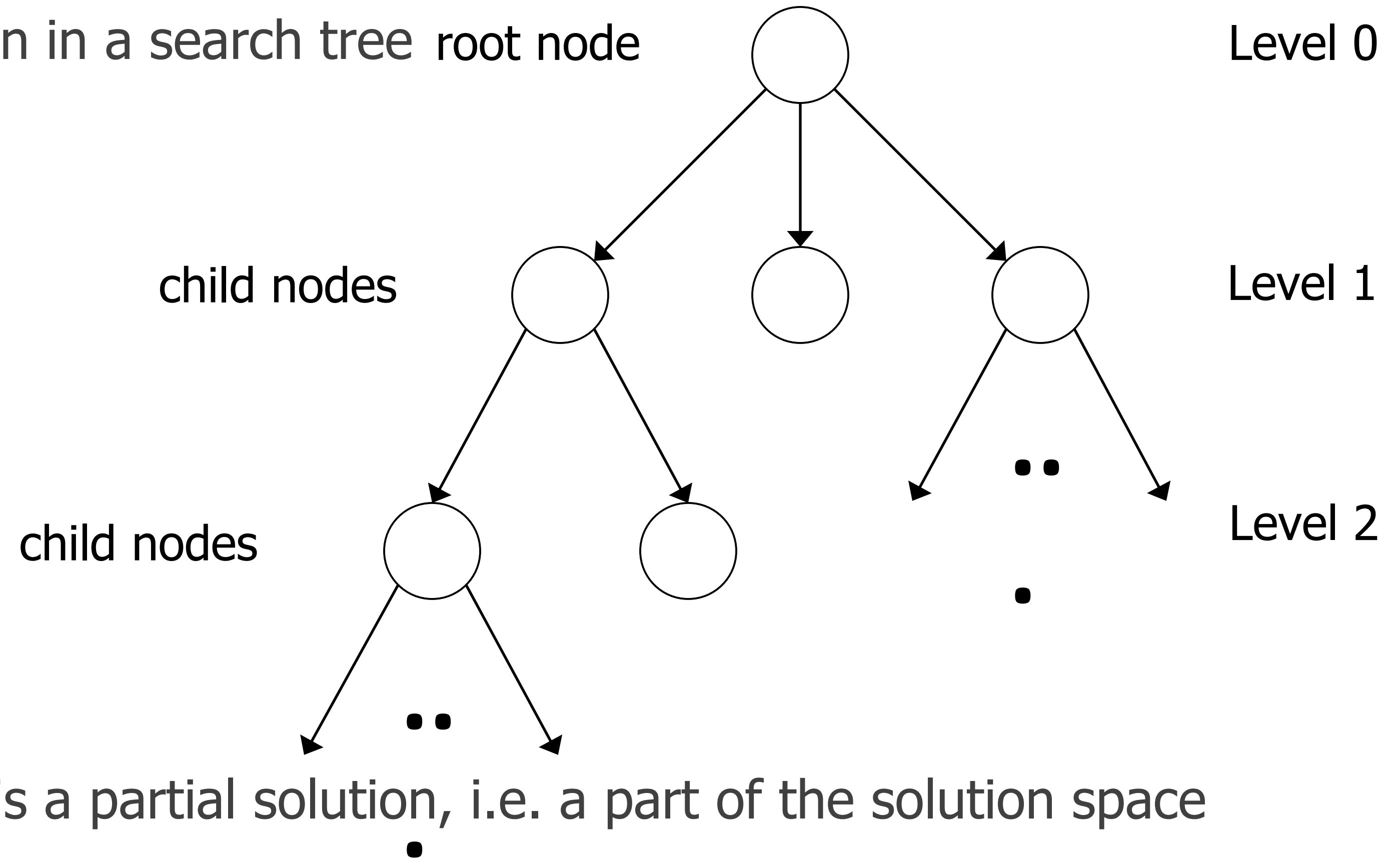
(ii) discarding the subset if the bound indicates that it can’t contain an optimal solution.

These three basic steps – **branching, bounding, and fathoming**

are illustrated on the following example.

Branch and bound

Enumeration in a search tree



root node

child nodes

child nodes

Level 0

Level 1

Level 2

each node is a partial solution, i.e. a part of the solution space

Branch and bound example 1

Disjunctive programming:

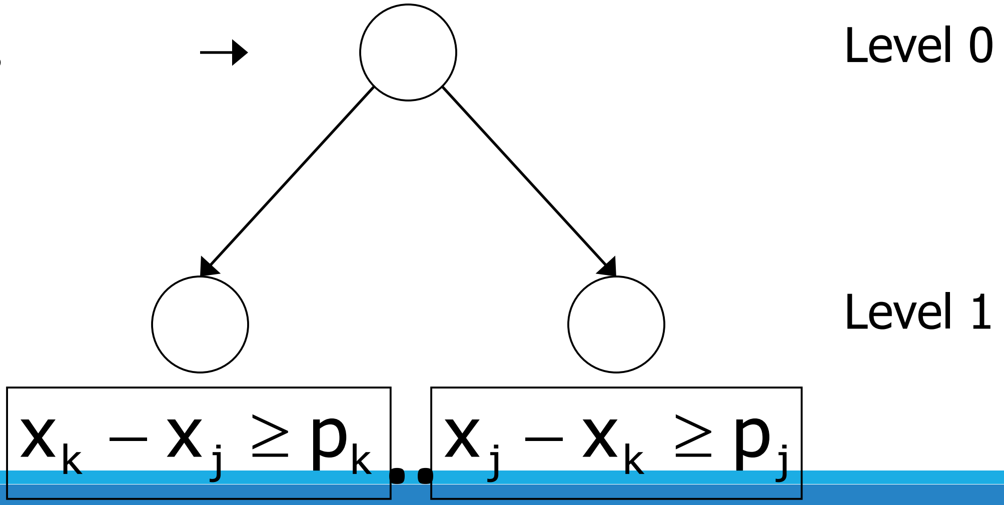
disjunctive set of constraints: <u>at least one</u> must be satisfied

x_j = completion time of job j

restriction: either $x_k - x_j \geq p_k$ or $x_j - x_k \geq p_j$ $(\forall j,k \in I)$

solve LP <u>without</u> disjunctive restrictions (= LP relaxation) →

if disjunct. restr. violated for j & k →



Branch and bound (cont.)

Upper bound: e.g. a feasible solution

Lower bound:

e.g. a solution to an “easier” problem

Node elimination (fathom/discard nodes):

when lower bound >= upper bound

Branch and bound (cont.)

Branching strategy:

how to partition solution space

Node selection strategy:

- sequence of exploring nodes:
  - depth first (tries to obtain a solution fast)
  - breadth/best bound first (tries to find the best solution)
- which nodes to explore (filter and beam width)
  - filter width: #nodes selected for thorough evaluation
  - beam width: #nodes that are branched on ($\leq$ filter width)

$\Rightarrow$ Beam search

Steps

Step 1: Traverse the root node.
Step 2: Traverse any neighbour of the root node that is maintaining least distance from the root node.
Step 3: Traverse any neighbour of the neighbour of the root node that is
maintaining least distance from the root node.
Step 4: This process will continue until we are getting the goal node

Step1: Consider node A as root node. Find successor B,C,F

B = 0+5

F= 0+9

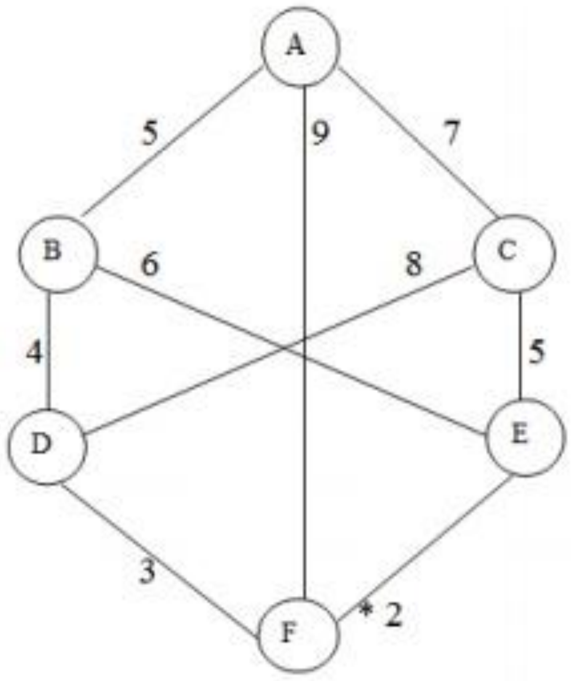
C= 0+7

B= 5 is least

Step 2: Now stack will be

C, F,B,-----

B is on top of the stack



Figure

D = 0+5+4 = 9

E = 0+5+6 = 11

The least distance is D from B

So it will be on top of stack

Step 3

C,F,D

C= 0+5+4+8 = 17

F=0+5+4+3= 12

Step 4

C,F

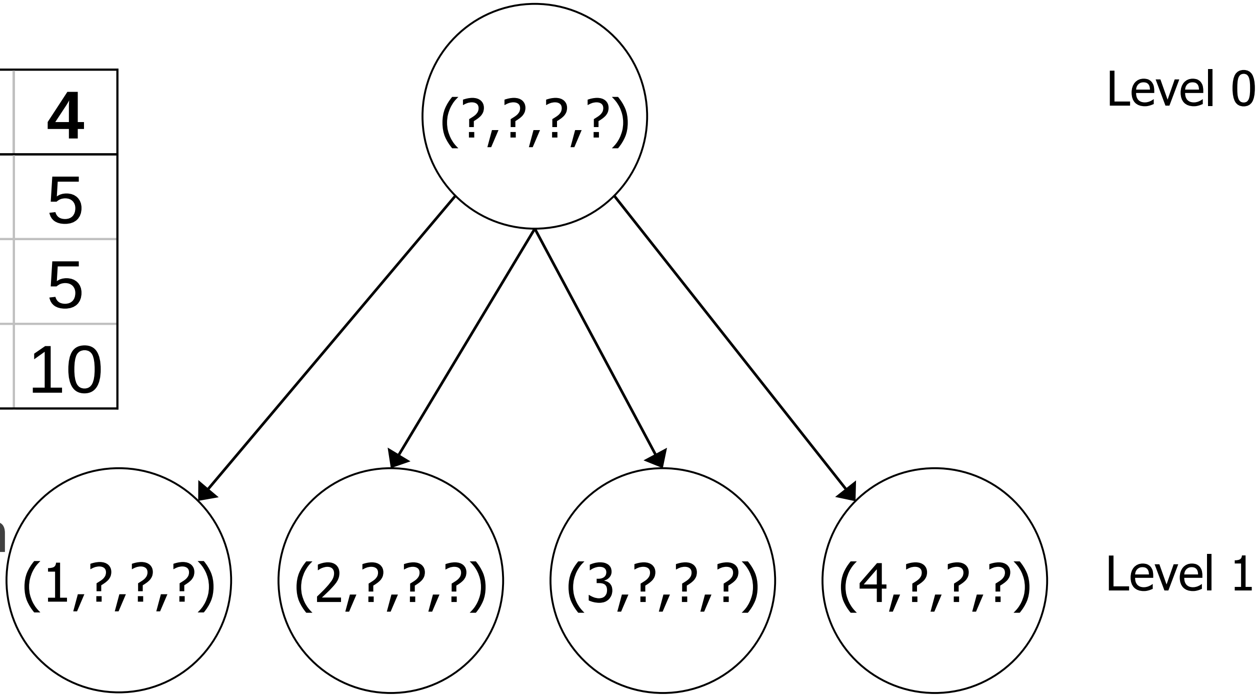
Searching Path : A-B-D-F

Branch and bound example 2

Single machine, maximum lateness, release and due dates

Jobs	**1**	**2**	**3**	**4**
p(j)	4	2	6	5
r(j)	0	1	3	5
d(j)	8	12	11	10

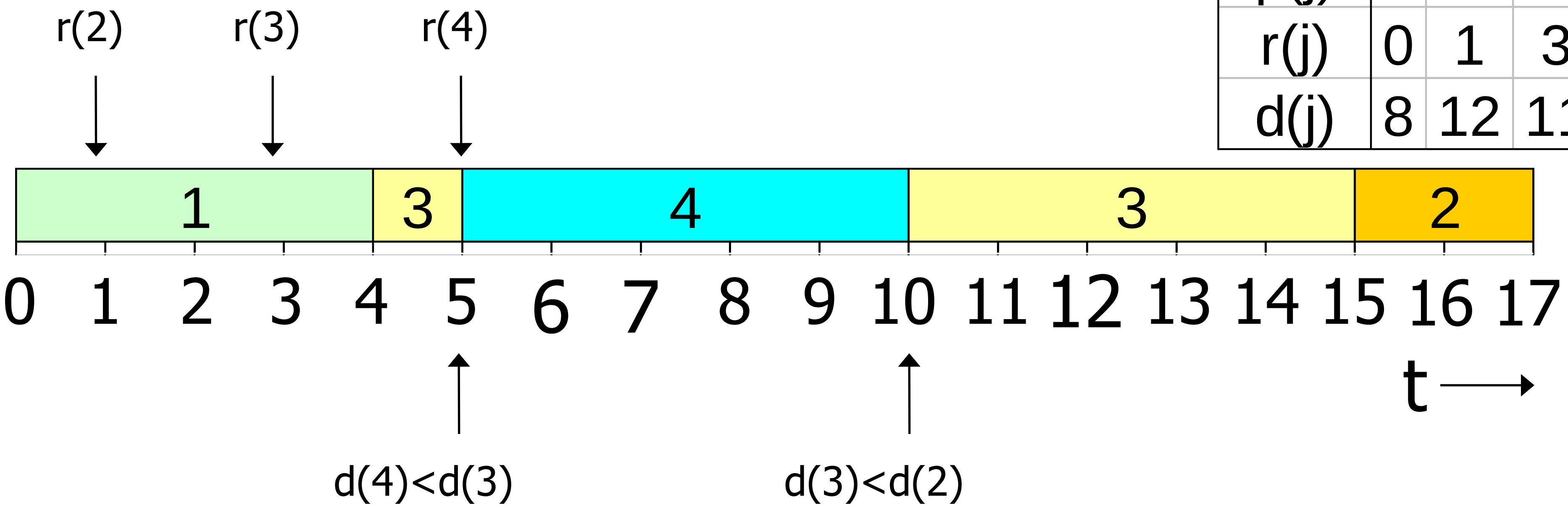
lower bound: EDD + preemption



Branch and bound example 2

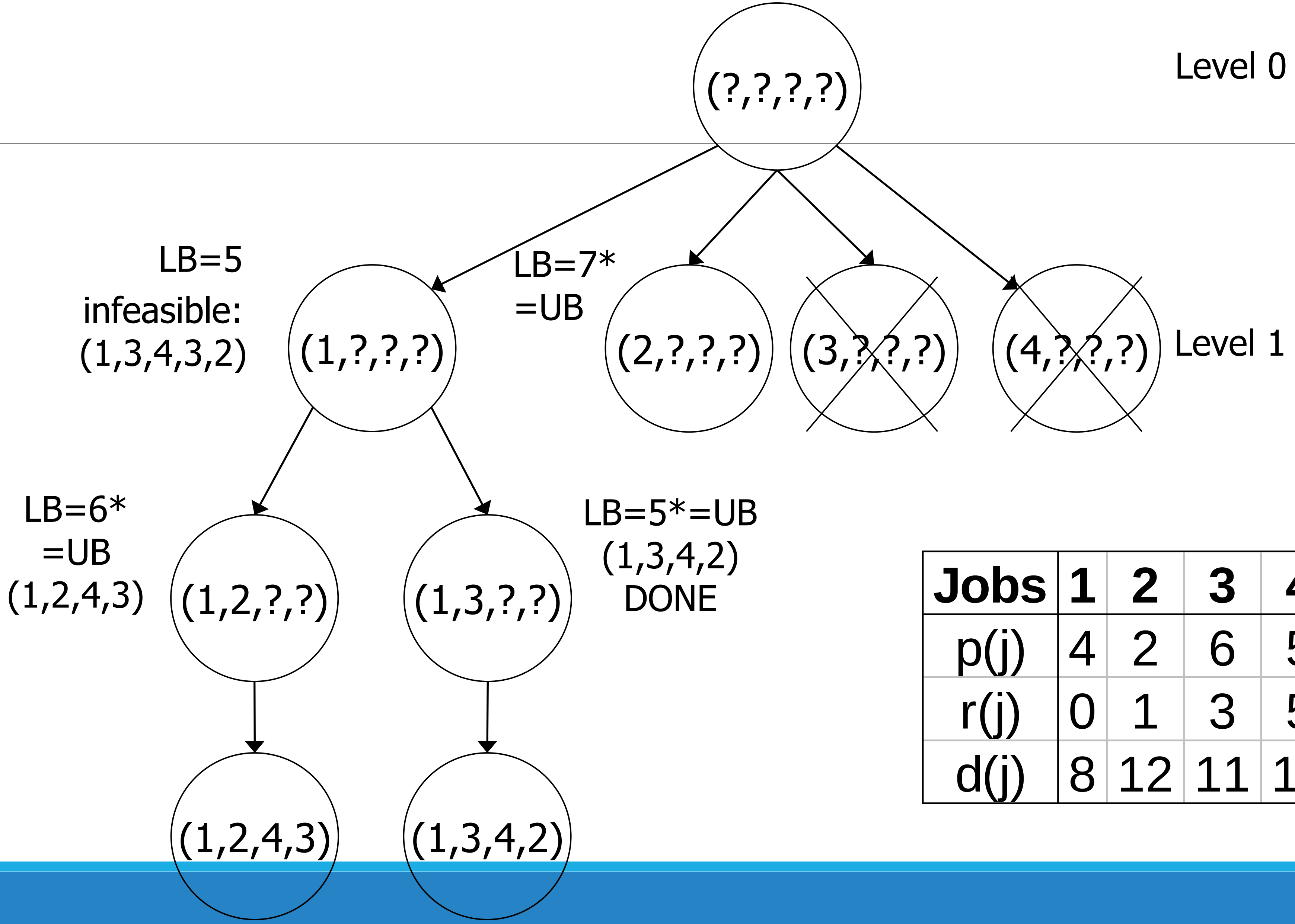
Lower bound for: (1,?,?,?)

Jobs	1	2	3	4
p(j)	4	2	6	5
r(j)	0	1	3	5
d(j)	8	12	11	10



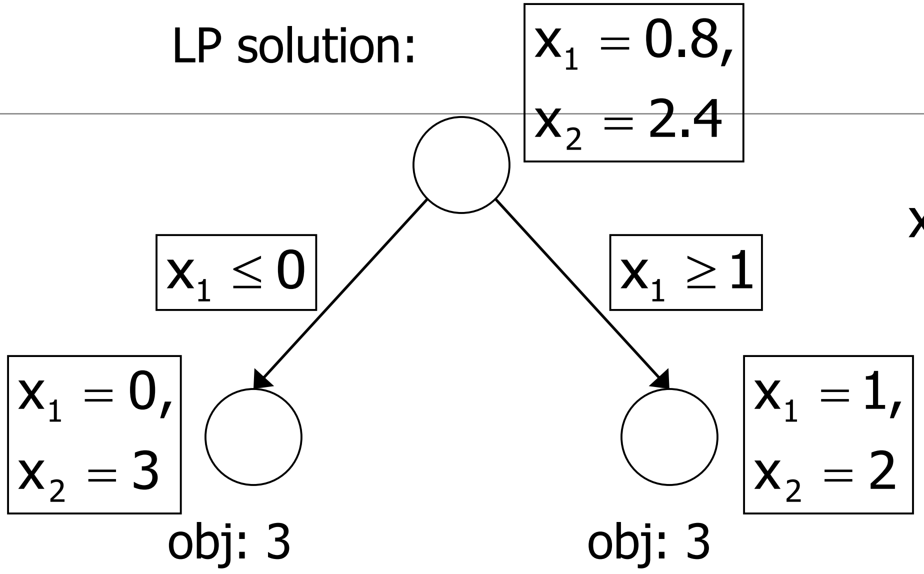
Lower bound: Lmax = max(0,17-12,15-11,0)=5

Branch and bound example 2 (cont.)



Jobs	**1**	**2**	**3**	**4**
p(j)	4	2	6	5
r(j)	0	1	3	5
d(j)	8	12	11	10

Branch and bound example 3



LP solution: $x_1 = 0.8,$ $x_2 = 2.4$

$x_1 \leq 0$

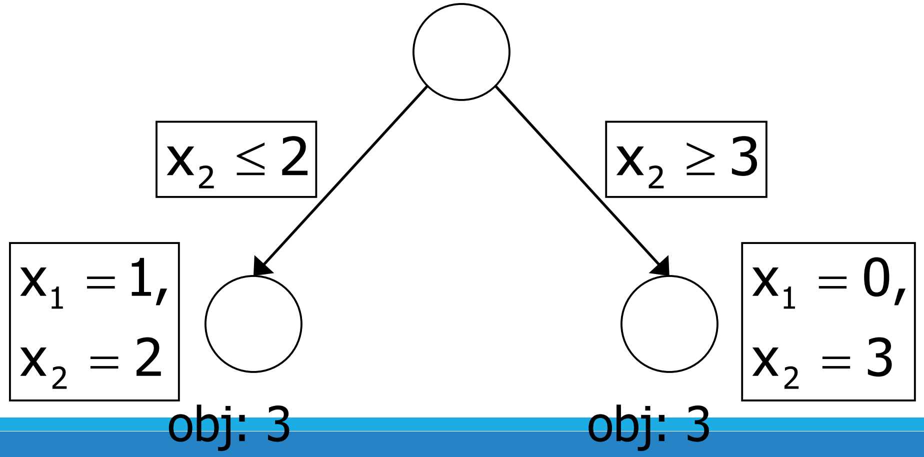
$x_1 = 0,$ $x_2 = 3$

obj: 3

$x_1 \geq 1$

$x_1 = 1,$ $x_2 = 2$

obj: 3



$x_2 \leq 2$

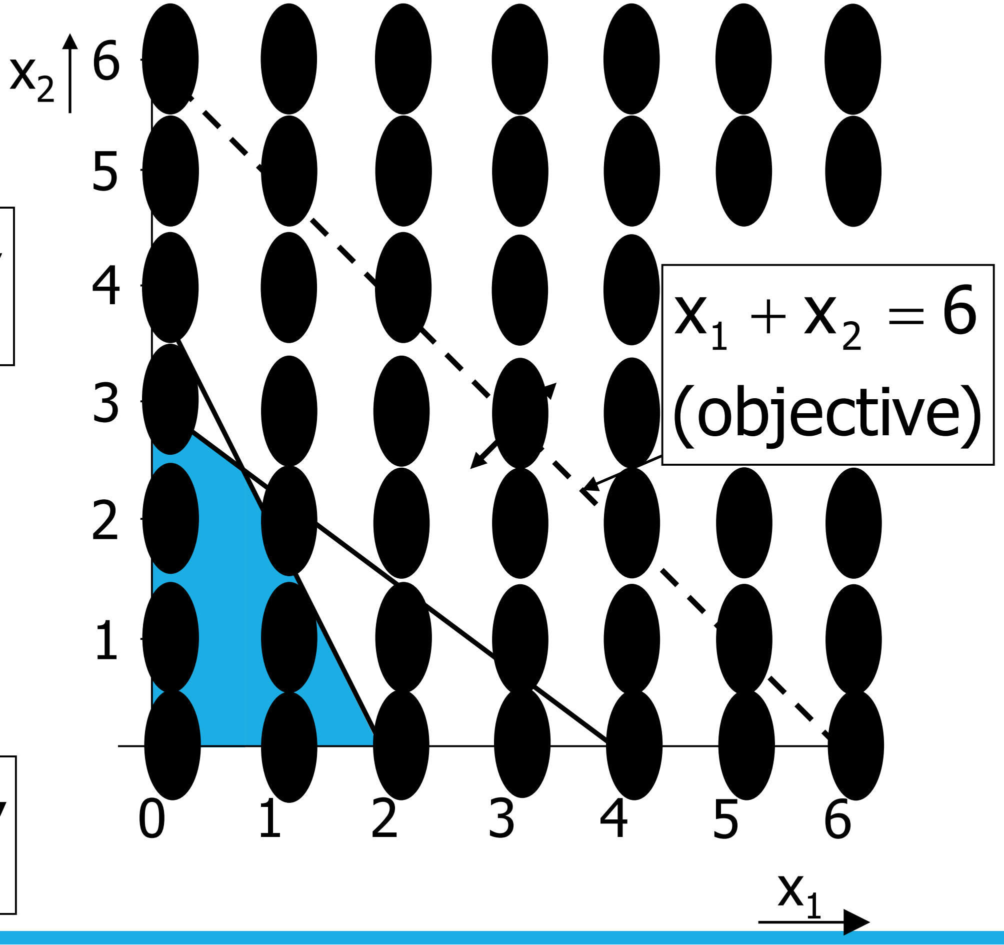
$x_1 = 1,$ $x_2 = 2$

obj: 3

$x_2 \geq 3$

$x_1 = 0,$ $x_2 = 3$

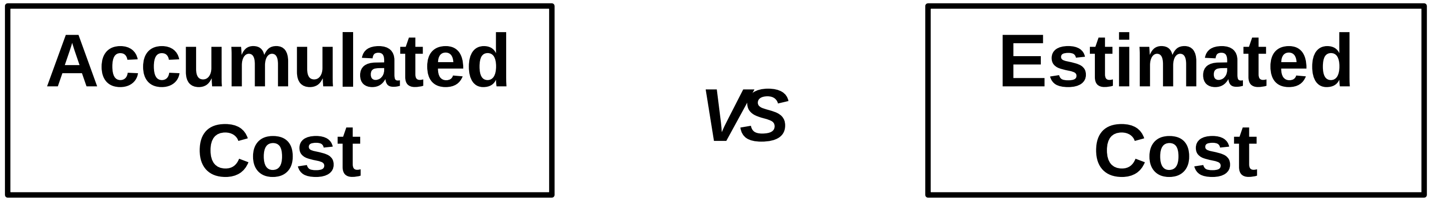
obj: 3



$x_1 + x_2 = 6$ (objective)

Best first search

- The basic idea of best first search is similar to uniform cost search.
- The only difference is that the “cost” (or in general, the evaluation function) is estimated based on some heuristics, rather than the accumulated cost calculated during search.



Accumulated Cost

vs

Estimated Cost

Uniform Cost search

Best first search

Algorithm: Best first search

1. Start with OPEN containing with just the initial state.
2. Until a goal is found or there are no node left on OPEN do:
    1. Pick the best node in open
    2. Generate it’s successors.
    3. For each successor do:
        1. If it has not been generated before, evaluate it, add it to OPEN and record it’s parent.
        2. If it has generated before, change the parent if this new path is better than the previous one. In that case, update the cost of getting to this node and to any successors that this node may already, have.

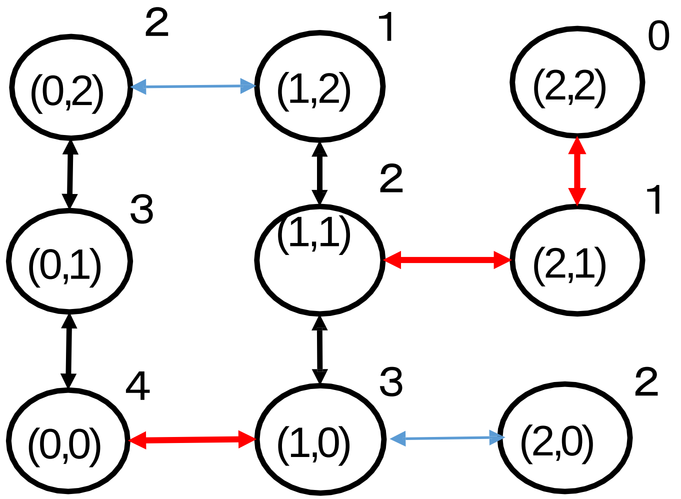
Best first search

Estimation of the cost (e.g. distance to the goal)

- Step 1: Put the initial node x0 and its **heuristic value** H(x0) to the open list.
- Step 2: Take a node x from the top of the open list. If the open list is empty, stop with failure. If x is the target node, stop with success.
- Step 3: Expand x and get a set S of child nodes. Add x to the closed list.
- Step 4: For each x’ in S but not in the closed list, estimate its heuristic value H. If x’ is not in the open list, put x’ along with the edge (x,x’) and H into the open list; otherwise, if H is smaller than the old value H(x’), update x’ with the new edge and the new heuristic value.
- Step 5: Sort the open list according to the heuristic values of the nodes, and return to Step 2.

Example 2.6 p. 25

Steps	Open List	Closed List
0	{(0,0),4}	--
1	{(1,0),3} {(0,1),3}	(0,0)
2	{(2,0),2} {(1,1),2} {(0,1),3}	(0,0) (1,0)
3	{(1,1),2} {(0,1),3}	(0,0) (1,0) (2,0)
4	{(2,1),1} {(1,2),1} {(0,1),3}	(0,0) (1,0) (2,0) (1,1)
5	{(2,2),0} {(1,2),1} {(0,1),3}	(0,0) (1,0) (2,0) (1,1) (2,1)
6	(2,2)=target node.	(0,0) (1,0) (2,0) (1,1) (2,1)



Advantages and disadvantages

Advantages:

1. Can switch between BFS and DFS, thus gaining the advantages of both.
2. More efficient when compared to DFS.

Disadvantages:

1. Chances of getting stuck in a loop are higher.

$g(n)$	Path Distance
$h(n)$	Estimate to Goal
$f(n)$	Combined Hueristics i.e. $g(n) + h(n)$

The A* Algorithm

- We cannot guarantee the optimal solution using the “best-first search”. This is because the true cost from the initial node to the current node is ignored.
- The A* algorithm can solve this problem (A=admissible).
- In the A* algorithm, the cost of a node is evaluated using both the estimated cost and the true cost as follows:

$$\mathbf{f}(x) = \mathbf{g}(x) + \mathbf{h}(x)$$

It has been proved the A* algorithm can obtain the best solution provided that the estimated cost h(x) is always smaller (conservative) than the best possible value h*(x)

A* algorithm has 3 parameters:

g : the cost of moving from the initial cell to the current cell. Basically, it is the sum of all the cells that have been visited since leaving the first cell.

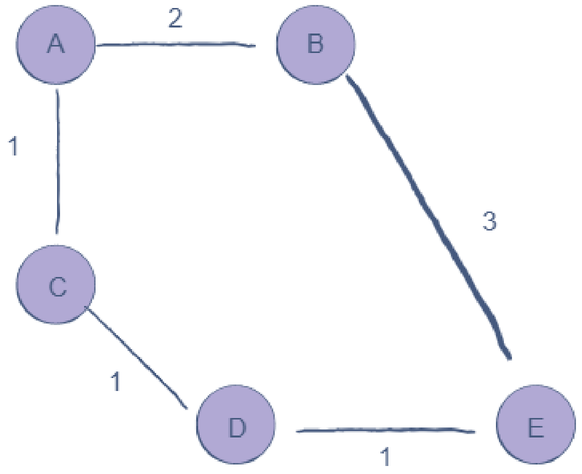
h : also known as the *heuristic value,* it is the **estimated** cost of moving from the current cell to the final cell. The actual cost cannot be calculated until the final cell is reached. Hence, h is the estimated cost. We **must** make sure that there is **never** an over estimation of the cost.

f : it is the sum of g and h. So, **f = g + h**

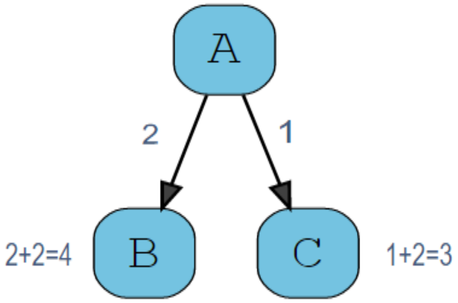
The algorithm makes its decisions is by taking the f-value into account. The algorithm selects the *smallest f-valued cell* and moves to that cell. This process continues until the algorithm reaches its goal cell.

Example

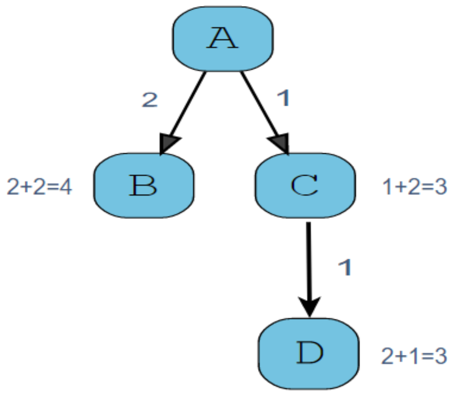
The initial node is **A** and the goal node is **E**.



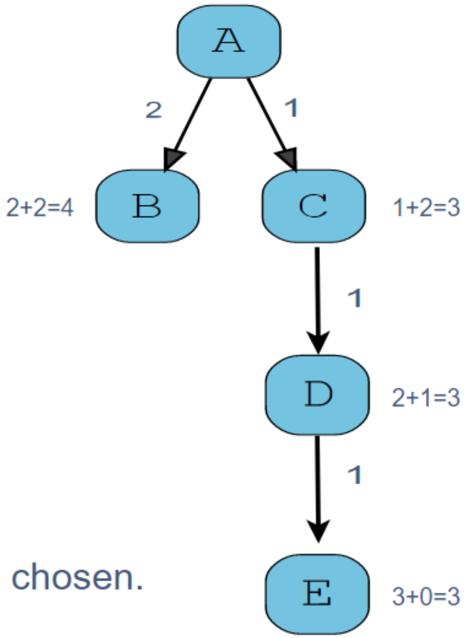
Root node A.



Node C is chosen.



Node D is chosen.



Node E is chosen.

The A* Algorithm

- Step 1: Put the initial node x0 and its cost F(x0)=H(x0) to the open list.
- Step 2: Get a node x from the top of the open list. If the open list is empty, stop with failure. If x is the target node, stop with success.
- Step 3: Expand x to get a set S of child nodes. Put x to the closed list.

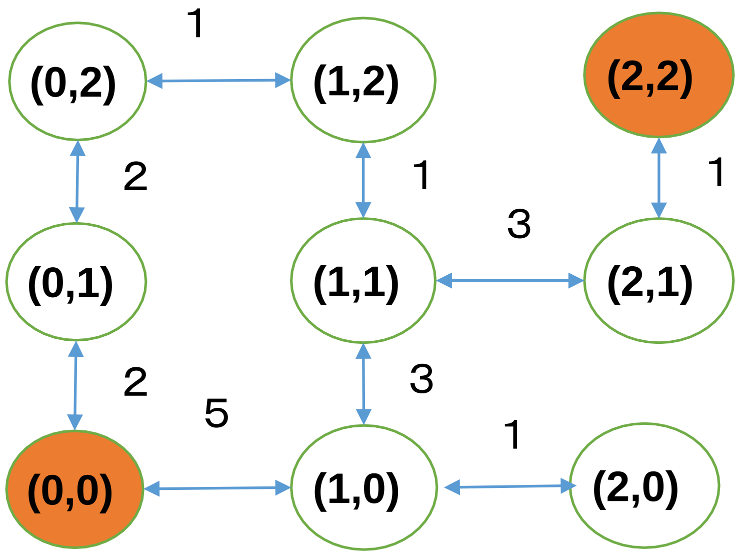
The A* Algorithm

- Step 4: For each x’ in S, find its cost

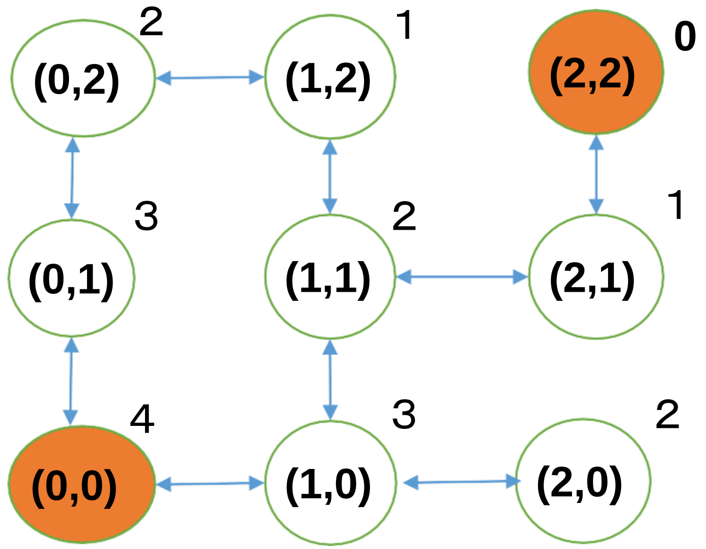
$$F = F(x) + d(x, x') + [H(x') - H(x)]$$

  - If x’ is in the closed list but the new cost is smaller than the old one, move x’ to the open list and update the edge (x,x’) and the cost.
  - Else, if x’ is in the open list, but the new cost is smaller than the old one, update the edge (x,x’) and the cost.
  - Else (if x’ is not in the open list nor in the closed list), put x’ along with the edge (x,x’) and the cost F to the open list.
- Step 5: Sort the open list according to the costs of the nodes, and return to Step 2.

Example 2.7 p. 27



True cost



Estimated cost

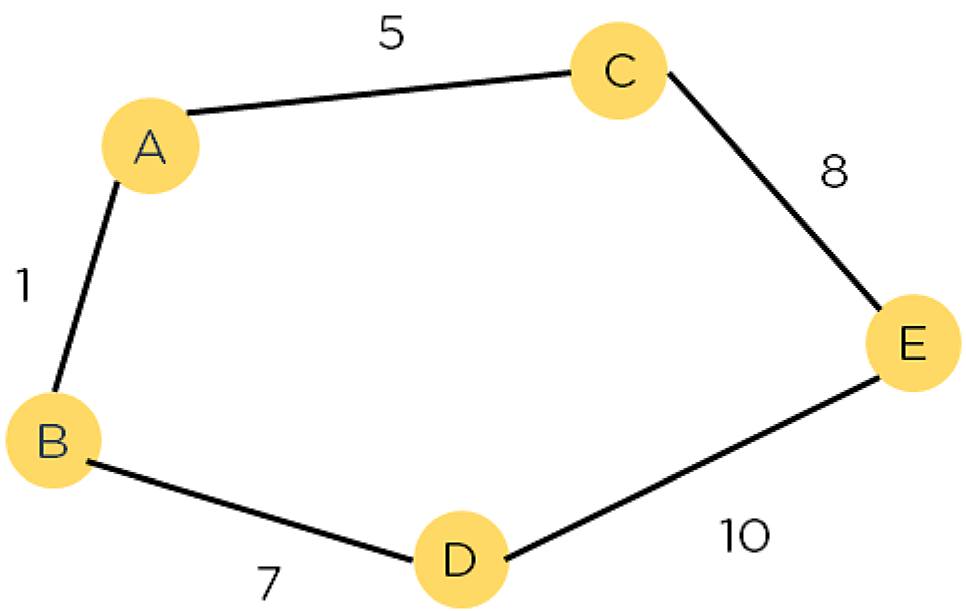
Example 2.7 p. 27

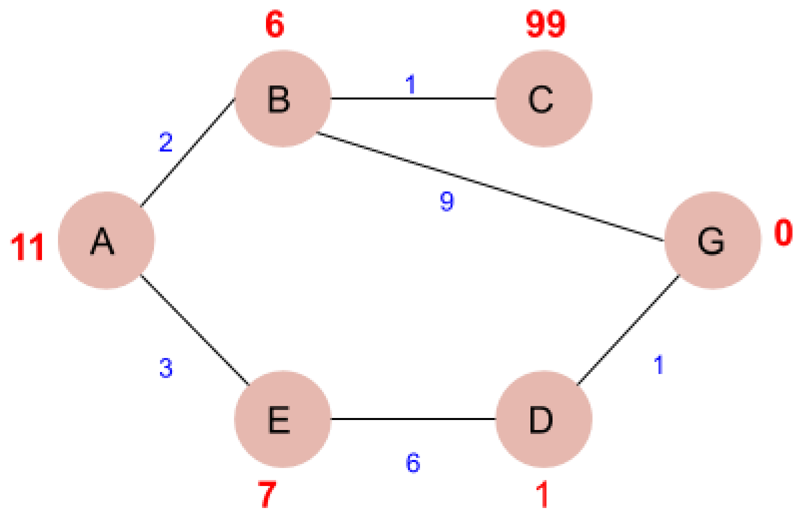
Steps	Open List	Closed List
0	{(0,0), 4}	--
1	{(0,1),5} {(1,0),8}	{(0,0),4}
2	{(0,2),6} {(1,0),8}	{(0,0),4} {(0,1),5}
3	{(1,2),6} {(1,0),8}	{(0,0),4} {(0,1),5} {(0,2),6}
4	{(1,0),8} {(1,1),8}	{(0,0),4} {(0,1),5} {(0,2),6} {(1,2),6}}
5	{(1,1),8} {(2,0),8}	{(0,0),4} {(0,1),5} {(0,2),6} {(1,2),6}} {(1,0),8}
6	{(2,0),8} {(2,1),10}	{(0,0),4} {(0,1),5} {(0,2),6} {(1,2),6}} {(1,0),8} {(1,1),8}
7	{(2,1),10}	{(0,0),4} {(0,1),5} {(0,2),6} {(1,2),6}} {(1,0),8} {(1,1),8} {(2,0),8}
8	{(2,2),10}	{(0,0),4} {(0,1),5} {(0,2),6} {(1,2),6}} {(1,0),8} {(1,1),8} {(2,0),8} {(2,1),10}
9	(2,2)=target node	{(0,0),4} {(0,1),5} {(0,2),6} {(1,2),6}} {(1,0),8} {(1,1),8} {(2,0),8} {(2,1),10}

What is an A* Algorithm?

- It is a searching algorithm that is used to find the shortest path between an initial and a final point.
- It searches for shorter paths first, thus making it an optimal and complete algorithm. An optimal algorithm will find the least cost outcome for a problem, while a complete algorithm finds all the possible outcomes of a problem.

A* uses weighted graphs in its implementation. A weighted graph uses numbers to represent the cost of taking each path or course of action





F (x) = g(x) + h(x)

A-B = 2+6 = 8

A-E = 3+7 = 10

Since A-B is less move to B

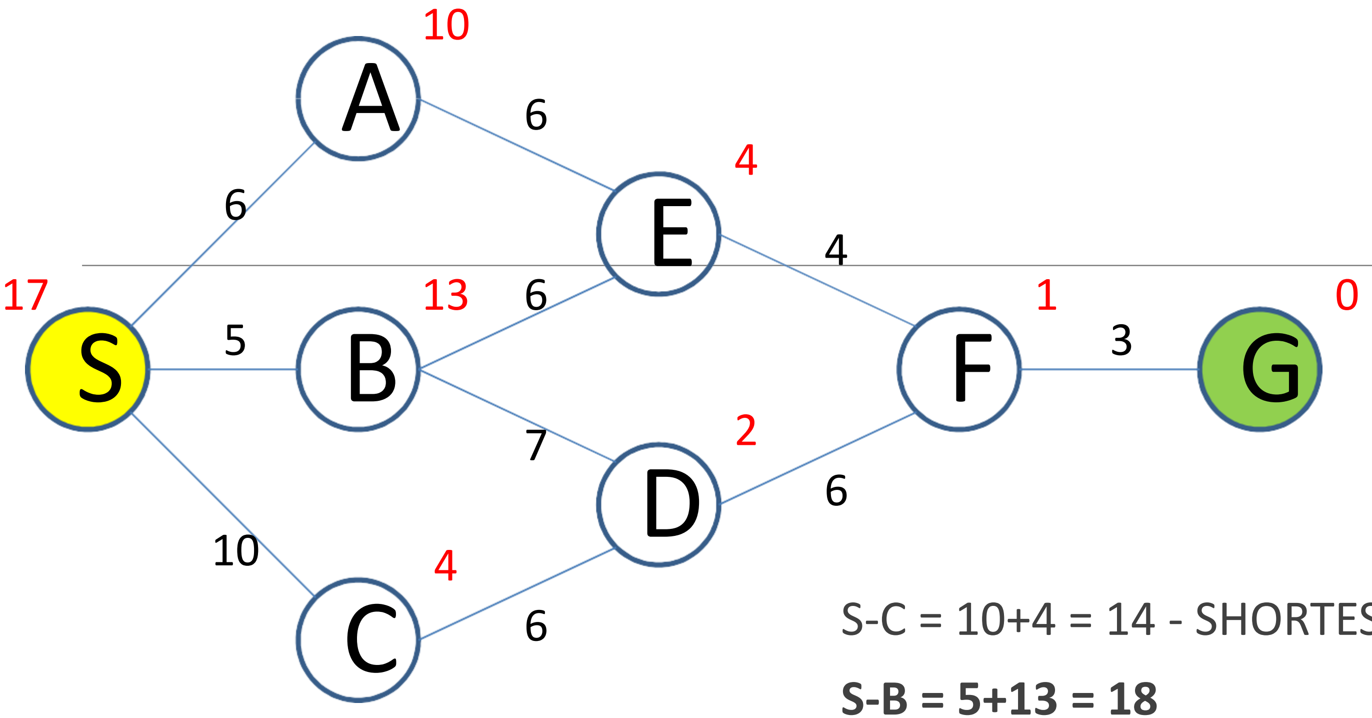
A-B-C = (2+1)+99 = 102

A-B-G = (2+9+0) = 11

A-B-G is least cost but still more than A-E. So explore that path

A-E-D = (3+6)+1 = 10

A-E-D-G = (3+6+1)+0= 10



OPEN : CLOSE

S-C = 10+4 = 14 - SHORTEST

S-B = 5+13 = 18

S-A = 6+10 = 16

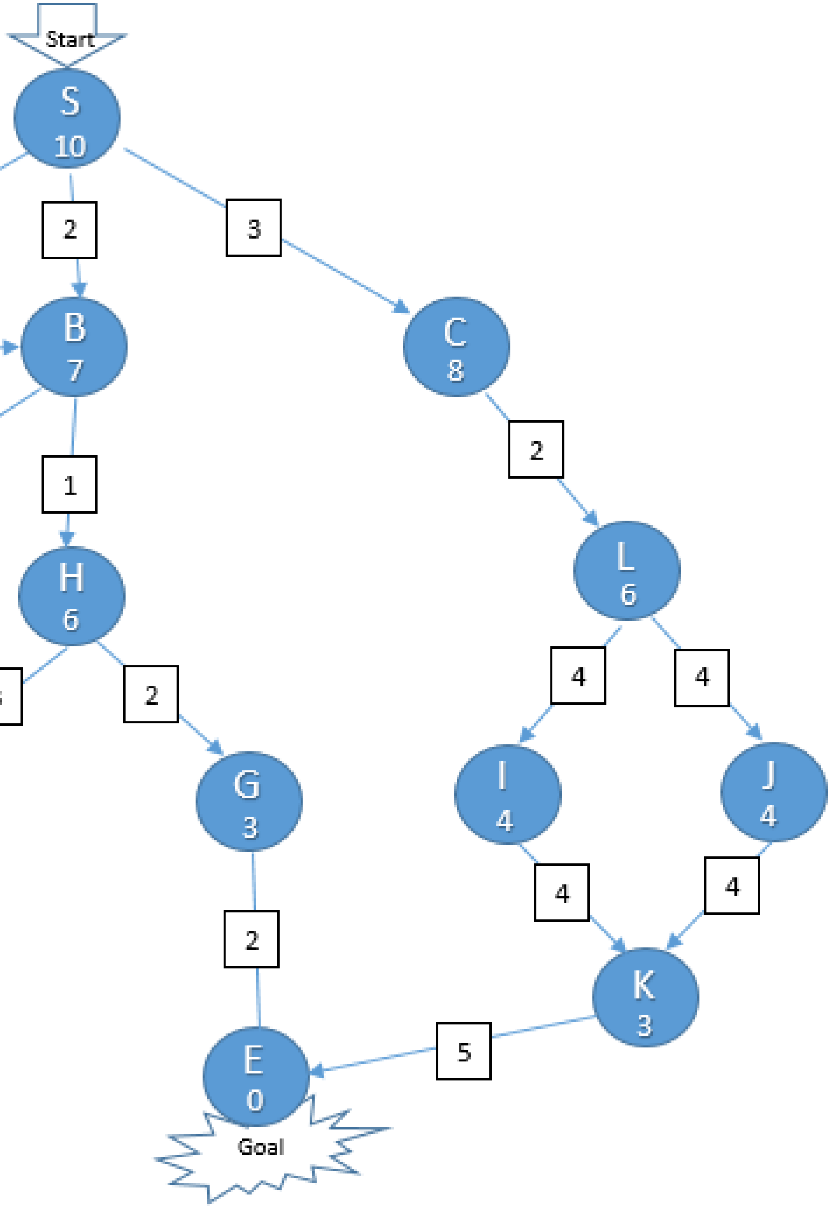
S-C-D = (10+6)+2 = 18 {S-A SHORTER}

S-A-E = (6+6)+4 = 16

S-A-E-B = (6+6+4+6) +13 = 35 (DO NOT VISIT)

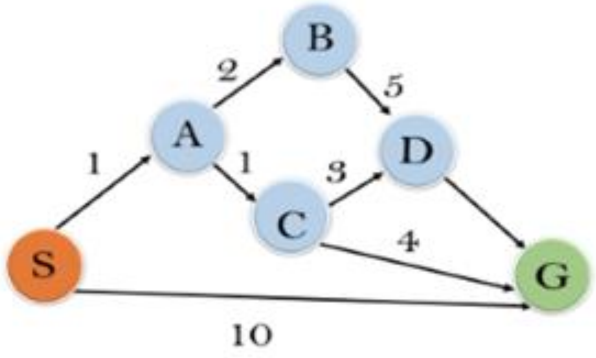
S-A-E-F = (6+6+4) +1 = 17 ---- S-A-E-F-G = 19

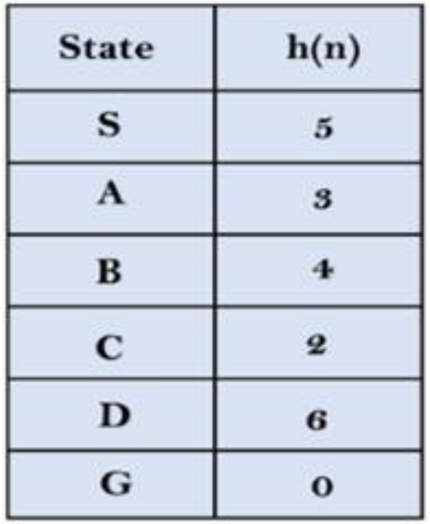
S-B-E-F-G = (5+6+4+3)+0 = 18


Start
S
10
7
2
3
A
9
3
B
7
C
8
4
4
1
2
D
8
H
6
L
6
5
3
2
4
4
F
6
G
3
I
4
J
4
2
4
4
K
3
5
E
0
Goal

A* Algorithm

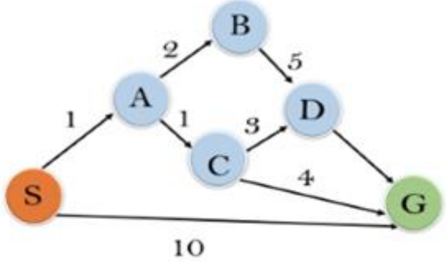
- In this example, we will traverse the given graph using the A* algorithm. The heuristic value of all states is given in the below table so we will calculate the f(n) of each state using the formula f(n)= g(n) + h(n), where g(n) is the cost to reach any node from start state.

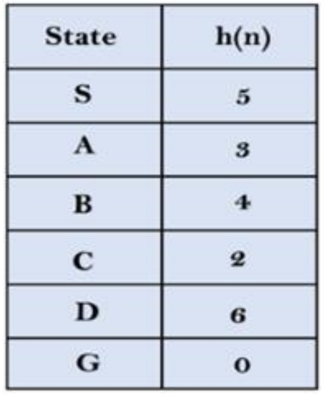




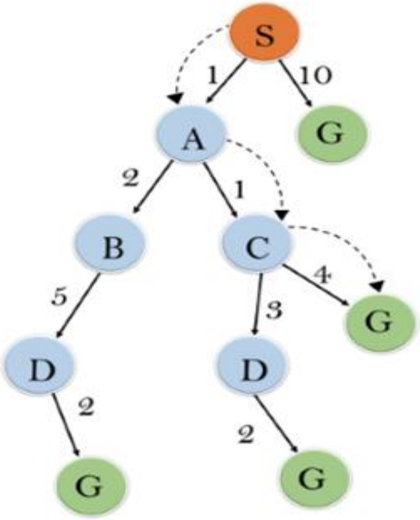
State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0

A* Algorithm





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



Initialization: {(S, 5)}
Iteration1: {(S--> A, 4), (S-->G, 10)}
Iteration2: {(S--> A-->C, 4), (S--> A-->B, 7), (S-->G, 10)}
Iteration3: {(S--> A-->C--->G, 6), (S--> A-->C--->D, 11), (S--> A-->B, 7), (S-->G, 10)}
Iteration 4: will give the final result, as **S--->A--->C--->G** it provides the optimal path with cost 6.

A* Algorithm

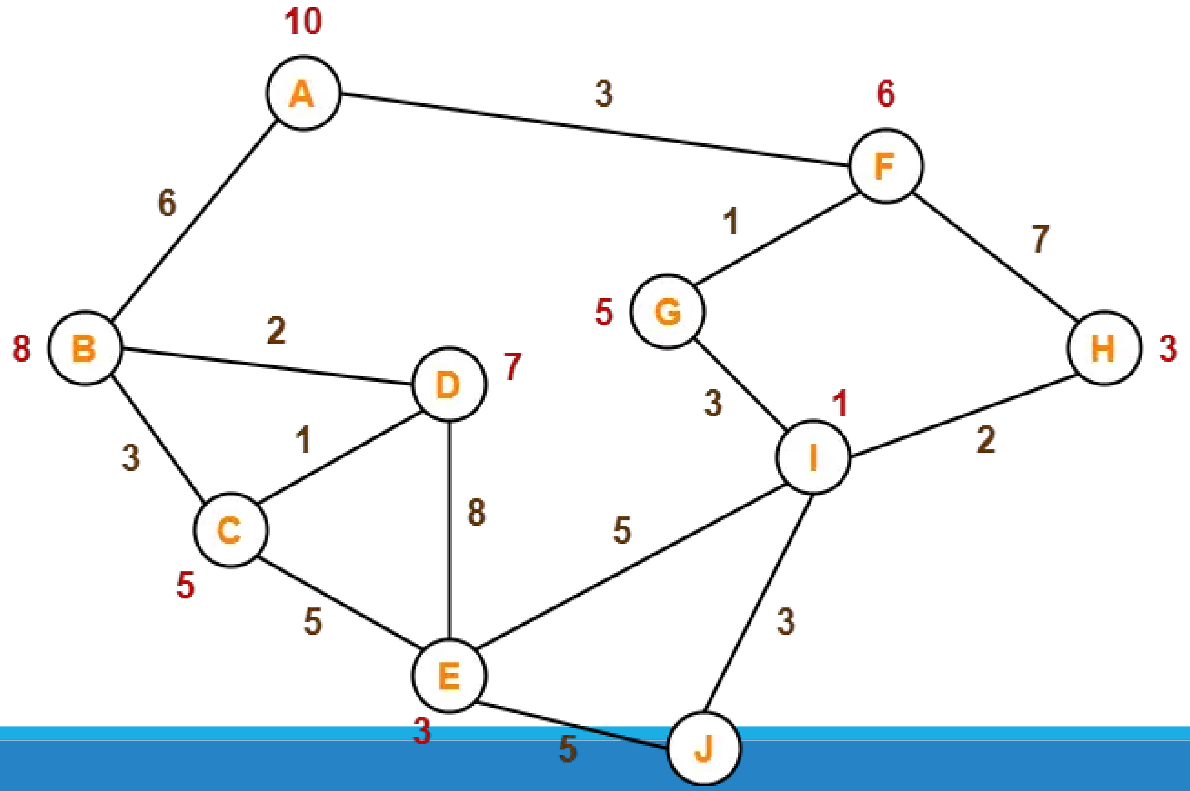
- **Points to remember:**
- A* algorithm returns the path which occurred first, and it does not search for all remaining paths.
- The efficiency of A* algorithm depends on the quality of heuristic.
- A* algorithm expands all nodes which satisfy the condition of f(n).

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

A* Algorithm

Solve the following problem using A* Algorithm:

- The numbers written on edges represent the distance between the nodes [g(n)].
- The numbers written on nodes represent the heuristic value [red colored, h(n)].
- Find the most cost-effective path to reach from start state A to final state J using A* Algorithm.



Solution

Property of the A* Algorithm

- The A* algorithm can obtain the best solution because it considers both the cost calculated up to now and the estimated future cost.
- **A sufficient condition for obtaining the best solution is that the estimated future cost is smaller than the best possible value.**
- If this condition is not satisfied, we may not be able to get the best solution. This kind of algorithms are called A algorithms.

Comparison between A* algorithms

- Suppose that A1 and A2 are two A* algorithms. If for any node x we have H1(x)>H2(x), we say A1 is more efficient.
- That is, if H(x) is closer to the best value H*(x), the algorithm is better.
- If H(x)=H*(x), we can get the solution immediately (because we now which way to go from any “current” node).
- On the other hand, if H(x)=0 for all x (no estimation, most conservative), the A* algorithm becomes the uniform cost search algorithm.

Some heuristics for finding H(x)

- For any given node x, we need an **estimation function** to find H(x).
- For maze problem, for example, H(x) can be estimated by using the **Manhattan distance** between the current node the target node. This distance is usually smaller than the true value (and this is good) because some edges may not exist in practice (See questions given below).
- For more complex problems, we may need a method (e.g. neural network) to learn the estimation function from experiences or observed data.

- **If there is no edge between two nodes x and x', what is the cost d(x,x')?**
- **What will happen is the weights are smaller than 1?**

Generalization of Search

- Generally speaking, search is a problem for finding the best solution x* from a given ***domain*** D.
- Here, D is a set containing all “states” or candidate solutions.
- Each state is often represented as a n-dimensional state vector or ***feature vector***.
- To see if a state x is the best (or the desired) one or not, we need to define an ***objective function*** (e.g. cost function) f(x). The problem can be formulated by

min f(x), for x in D

IDA-Star(IDA*) Algorithm

- Iterative deepening A* (IDA*) is a graph traversal and path search algorithm that can find the shortest path between a designated start node and any member of a set of goal nodes in a weighted graph.
- It is a variant of iterative deepening depth-first search.
- Since it is a depth-first search algorithm, its memory usage is lower than in A*.
- IDA* is a memory constrained version of A*
- it has the optimal characteristics of A* to find the shortest path but it uses less memory than A*.

Pros and Cons

Pros:

- It will always find the optimal solution provided that it exist's and that if a heuristic is supplied it must be admissible.
- Heuristic is not necessary, it is used to speed up the process.
- Various heuristics can be integrated to the algorithm without changing the basic code.
- The cost of each move can be tweaked into the algorithms as easily as the heuristic
- Uses a lot less memory which increases linearly as it doesn't store and forgets after it reaches a certain depth and start over again.

Cons:

- Doesn't keep track of visited nodes and thus explores already explored nodes again.
- Slower due to repeating the exploring of explored nodes.
- Requires more processing power and time than A*.

How IDA* algorithm work

Step 1: Initialization: Set the root node as the current node, and find the f-score.

Sep 2: Set threshold: Set the cost limit as a threshold for a node i.e the maximum f-score allowed for that node for further explorations.

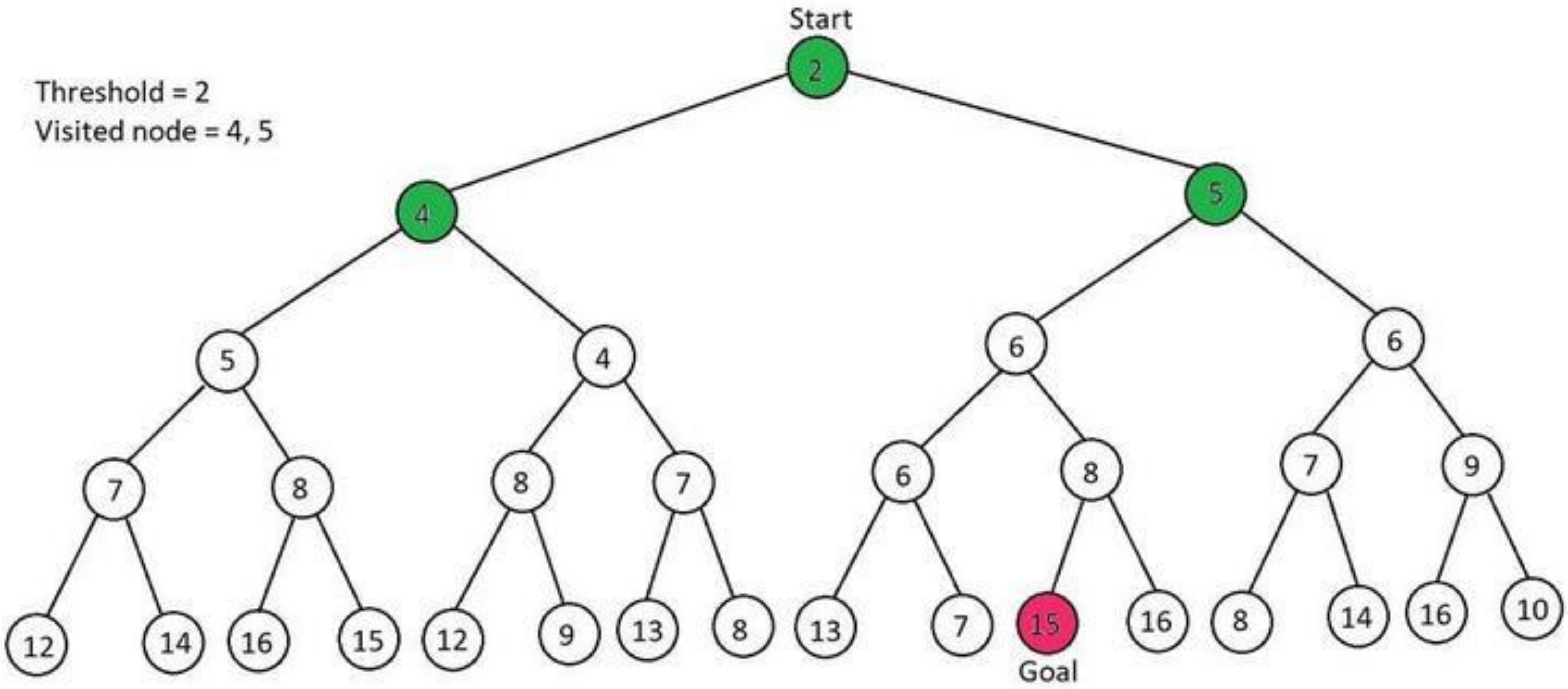
Step 3: Node Expansion: Expand the current node to its children and find f-scores.

Step 4: Pruning: If for any node the f-score > threshold, prune that node because it's considered too expensive for that node. and store it in the visited node list.

Step 5: Return Path: If the Goal node is found then return the path from the start node Goal node.

Step 6: Update the Threshold: If the Goal node is not found then repeat from step 2 by changing the threshold with the minimum pruned value from the visited node list. And Continue it until you reach the goal node.

1st iteration

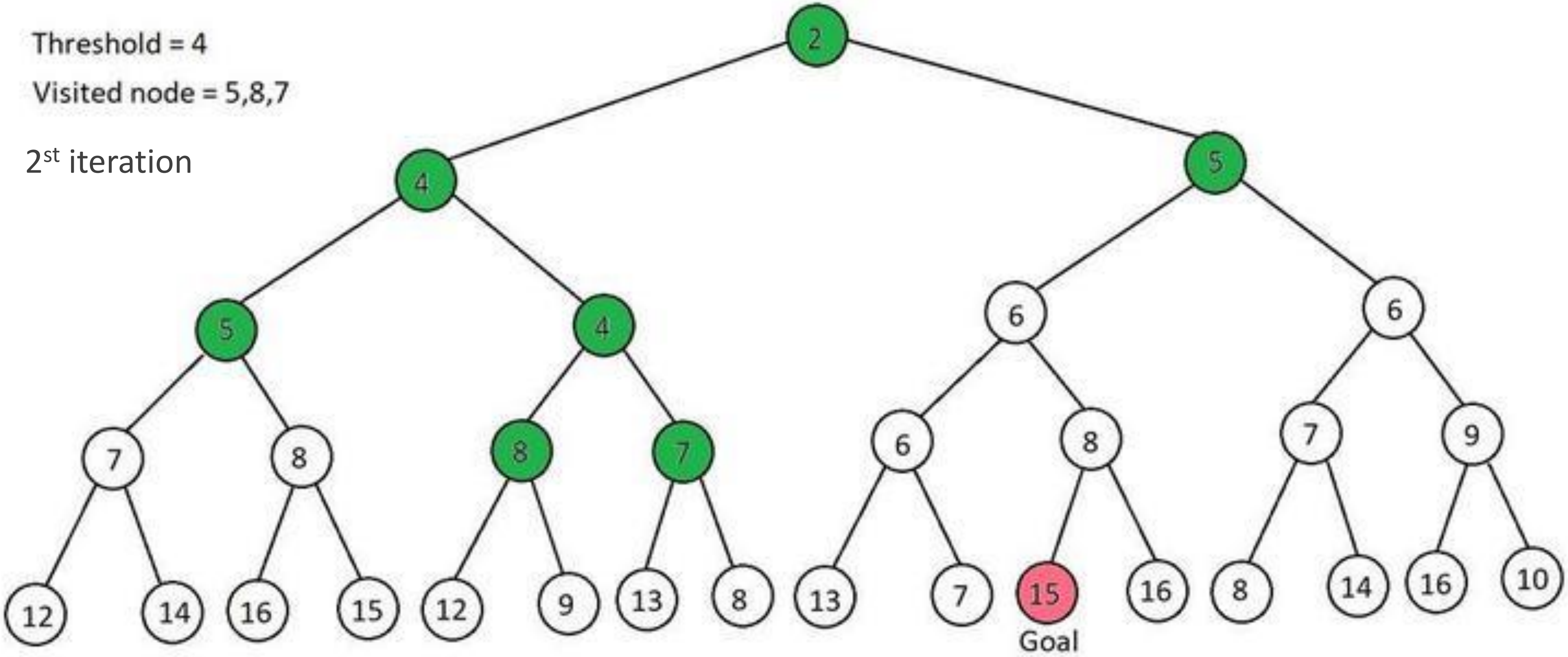


Threshold = 2
Visited node = 4, 5

Threshold = 4

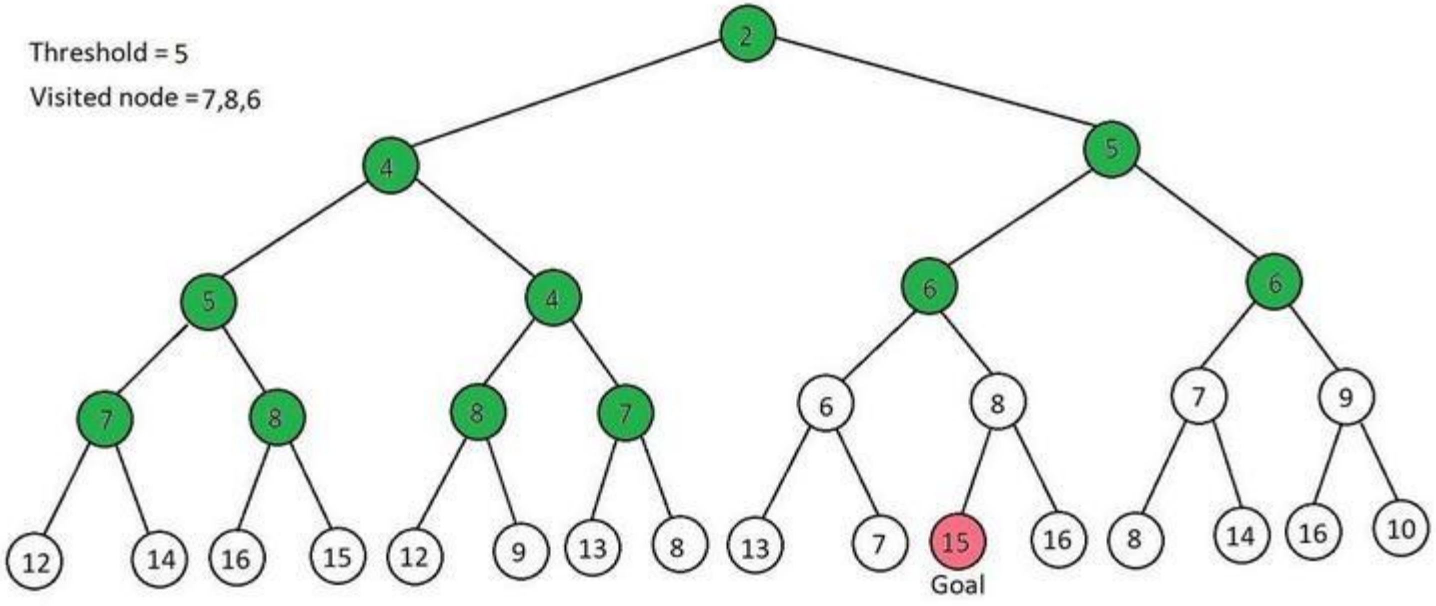
Visited node = 5,8,7

2st iteration



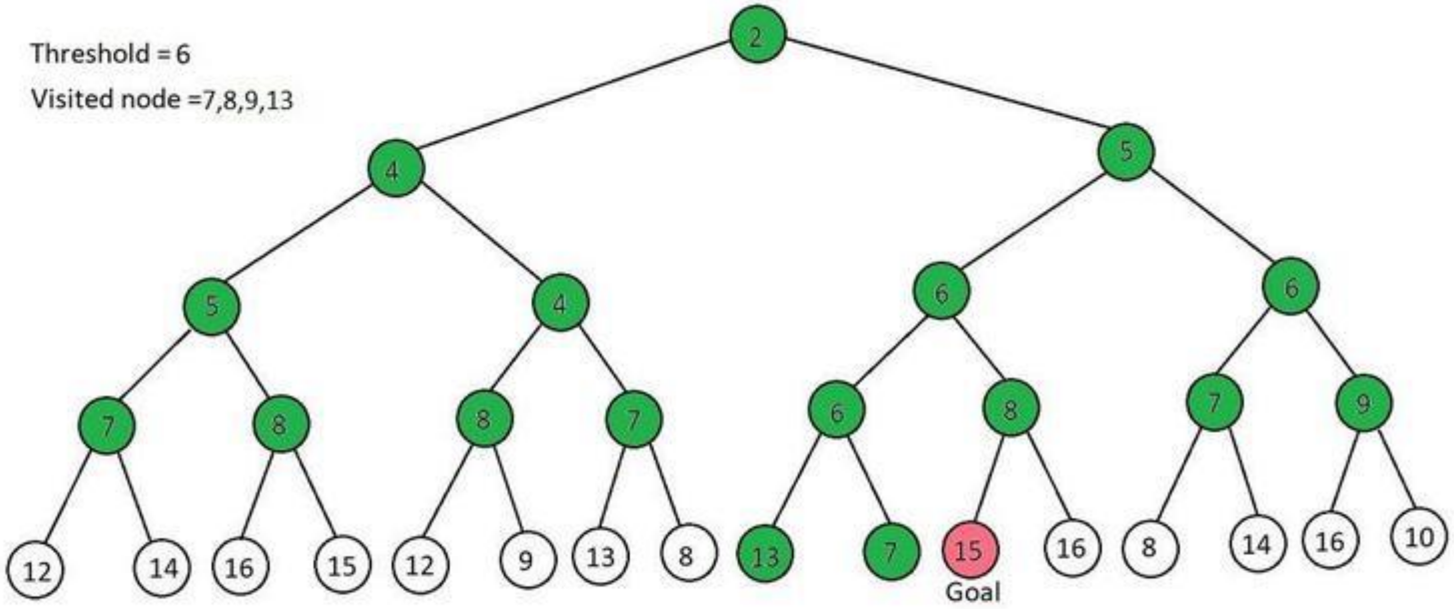
Threshold = 5

Visited node =7,8,6



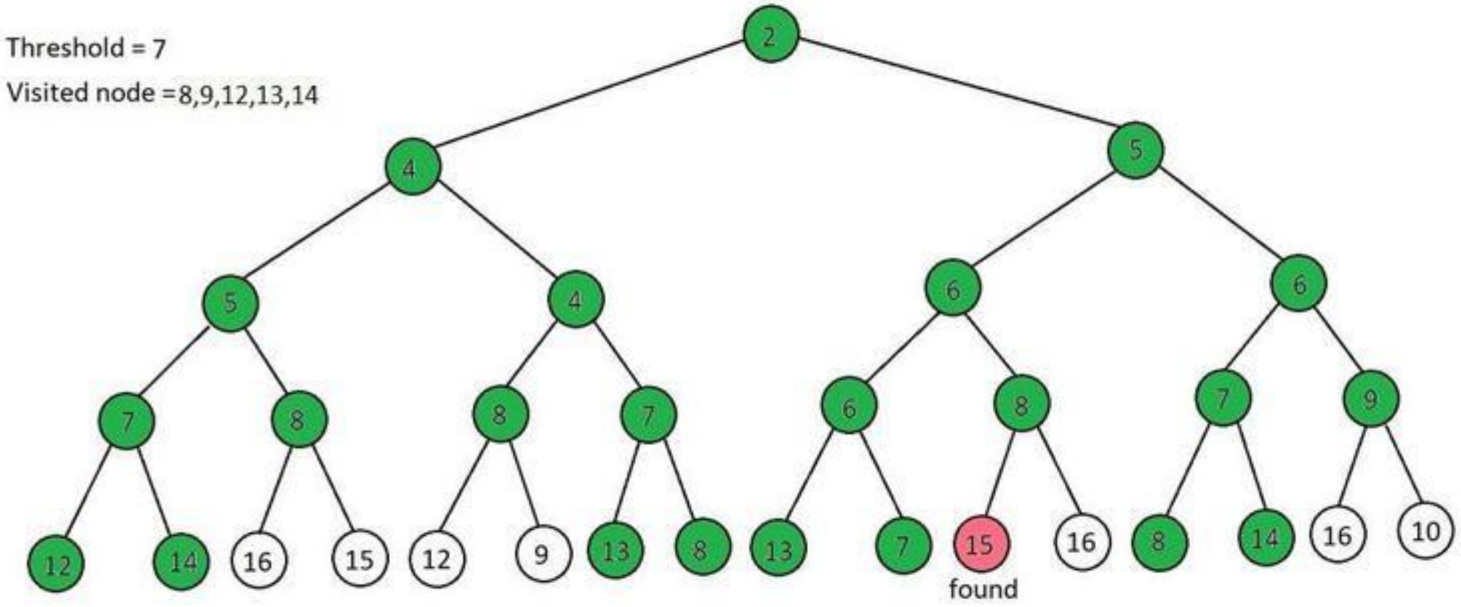
Threshold = 6

Visited node =7,8,9,13



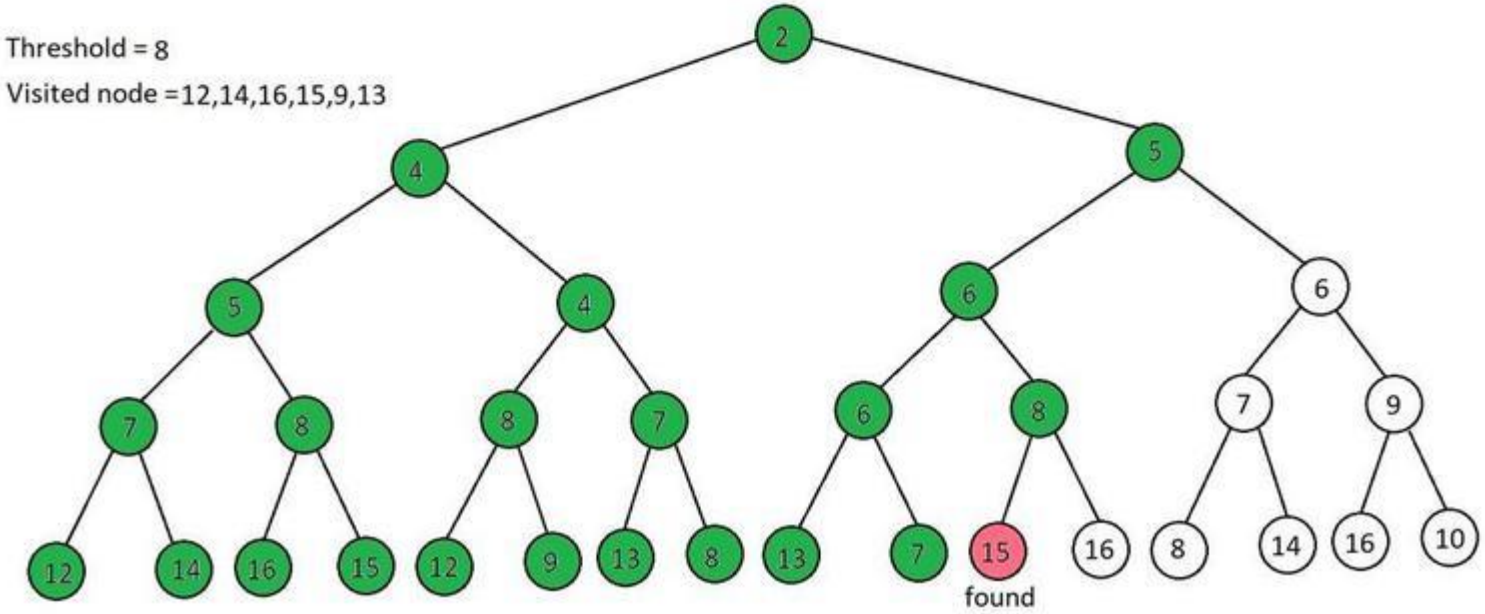
Threshold = 7

Visited node =8,9,12,13,14



Threshold = 8

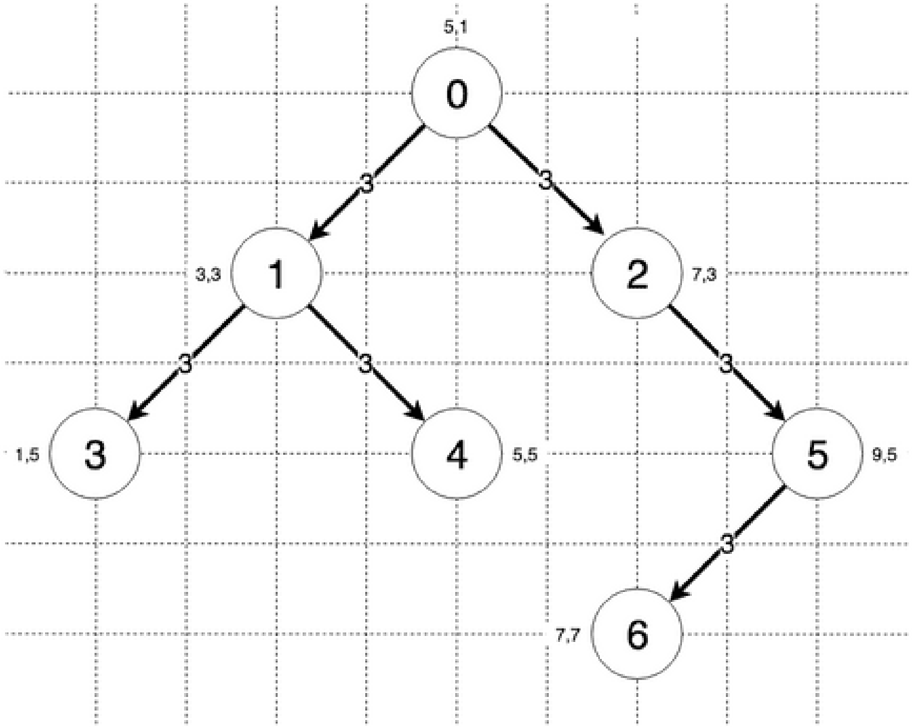
Visited node =12,14,16,15,9,13



Steps of Algorithm

1. For each child of the current node
2. If it is the target node, return
3. If the distance plus the heuristic exceeds the current threshold, return this exceeding threshold
4. Set the current node to this node and go back to 1.
5. After having gone through all children, go to the next child of the parent (the next sibling)
6. After having gone through all children of the start node, increase the threshold to the smallest of the exceeding thresholds.
7. If we have reached all leaf (bottom) nodes, the goal node doesn’t exist.

Example



- The steps the algorithm performs on this graph if given node 0 as a starting point and node 6 as the goal, in order, are:
- Iteration with threshold: 6.32
- Visiting Node 0
- Visiting Node 1
- Breached threshold with heuristic: 8.66
- Visiting Node 2
- Breached threshold with heuristic: 7.00
- Iteration with threshold: 7.00
- Visiting Node 0
- Visiting Node 1
- Breached threshold with heuristic: 8.66
- Visiting Node 2
- Visiting Node 5

- Breached threshold with heuristic: 8.83
- Iteration with threshold: 8.66
- Visiting Node 0
- Visiting Node 1
- Visiting Node 3
- Breached threshold with heuristic: 12.32
- Visiting Node 4
- Breached threshold with heuristic: 8.83
- Visiting Node 2
- Visiting Node 5
- Breached threshold with heuristic: 8.83
- Iteration with threshold: 8.83
- Visiting Node 0
- Visiting Node 1
- Visiting Node 3
- Breached threshold with heuristic: 12.32
- Visiting Node 4
- Visiting Node 2
- Visiting Node 5
- Visiting Node 6
- Found the node we’re looking for!

Final lowest distance from node 0 to node 6: 9

Runtime of the Algorithm

The runtime complexity of Iterative Deepening A Star is in principle the same as Iterative Deepening DFS. In practice, though, if we choose a good heuristic, many of the paths can be eliminated before they are explored making for a significant time improvement

Space of the Algorithm

The space complexity of Iterative Deepening A Star is the amount of storage needed for the tree or graph. *O(|N|), |N|* = number of Nodes in the tree or graph, which can be replaced with *b^d* for trees, where *b* is the branching factor and *d* is the depth. Additionally, whatever space the heuristic requires.

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.

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.

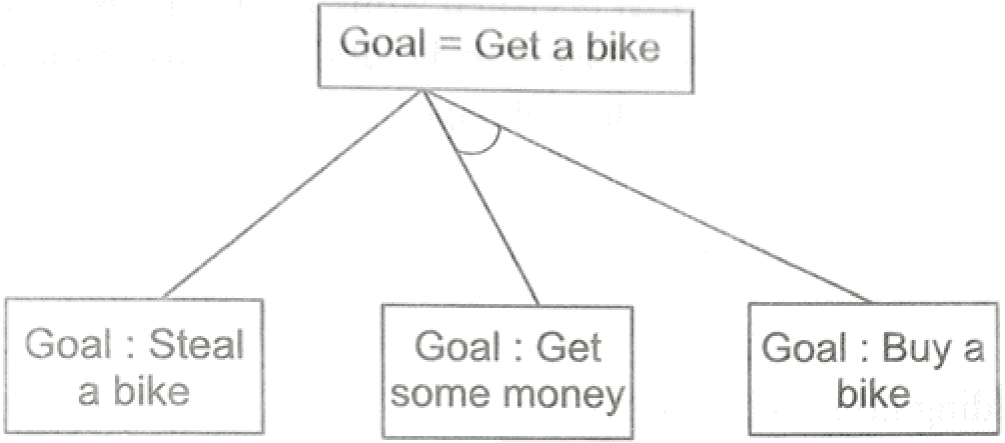


Fig: AND / OR Graph

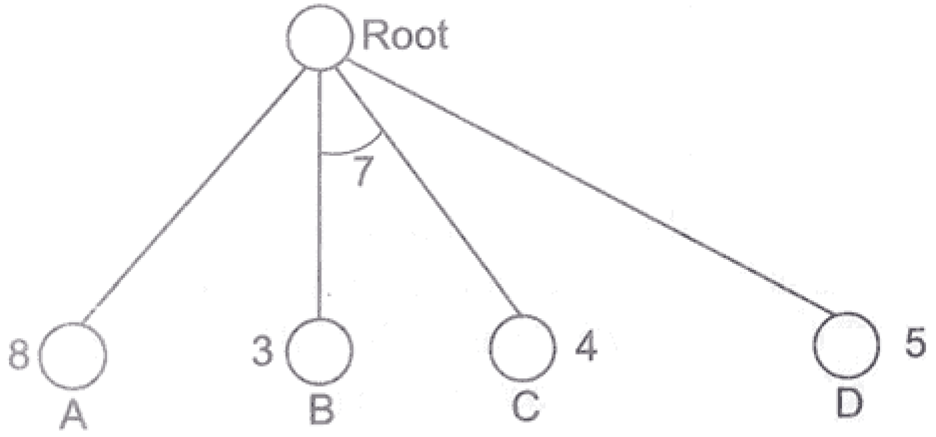


Fig: AND / OR Tree

1. Initialize the graph to the starting node.

2. Loop until the starting node is labelled **SOLVED** or until its cost goes above **FUTILITY**:

(i) Traverse the graph, starting at the initial node and following the current best path and accumulate the set of nodes that are on that path and have not yet been expanded.

(ii) Pick one of these unexpanded nodes and expand it. If there are no successors, assign FUTILITY as the value of this node. Otherwise, add its successors to the graph and for each of them compute f'(n). If f'(n) of any node is O, mark that node as SOLVED.

(iii) Change the f'(n) estimate of the newly expanded node to reflect the new information provided by its successors. Propagate this change backwards through the graph. If any node contains a successor arc whose descendants are all solved, label the node itself as SOLVED.

AO* Algorithm

- Working of AO* algorithm:
- The evaluation function in AO* looks like this:

  f(n) = g(n) + h(n)
- f(n) = Actual cost + Estimated cost

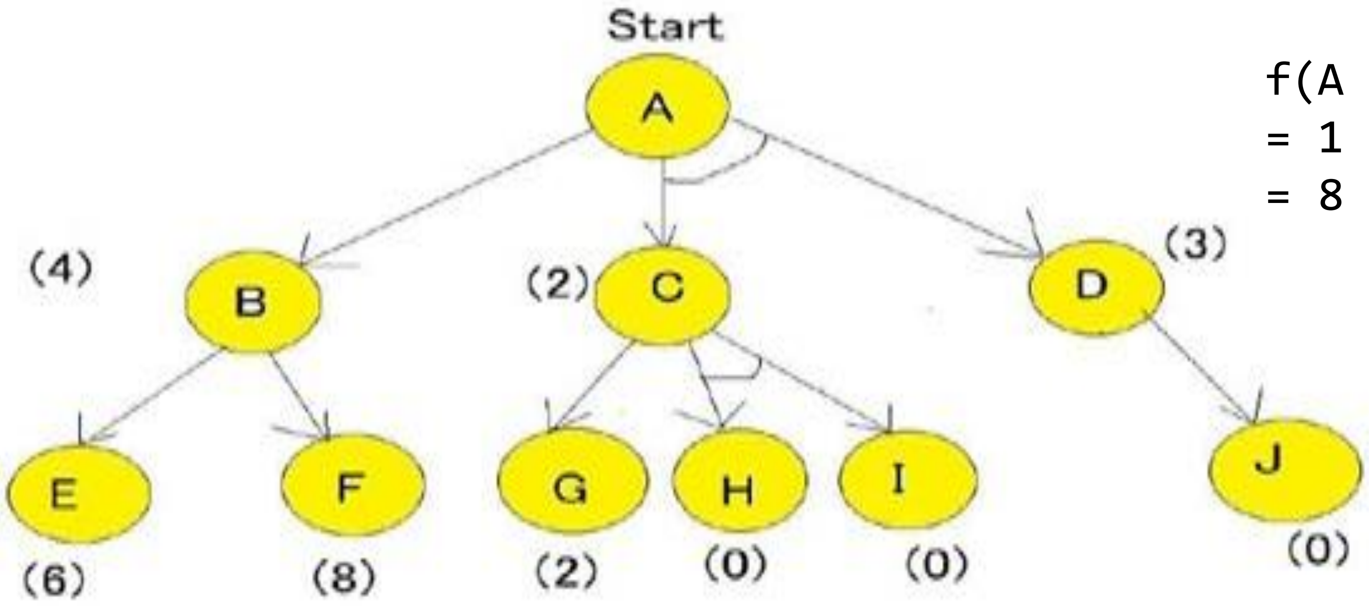
  where,

  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.

f(A -> B) = g(B) + h(B)
= 1 + 5 (g(n) = 1 is the default path cost)
= 6

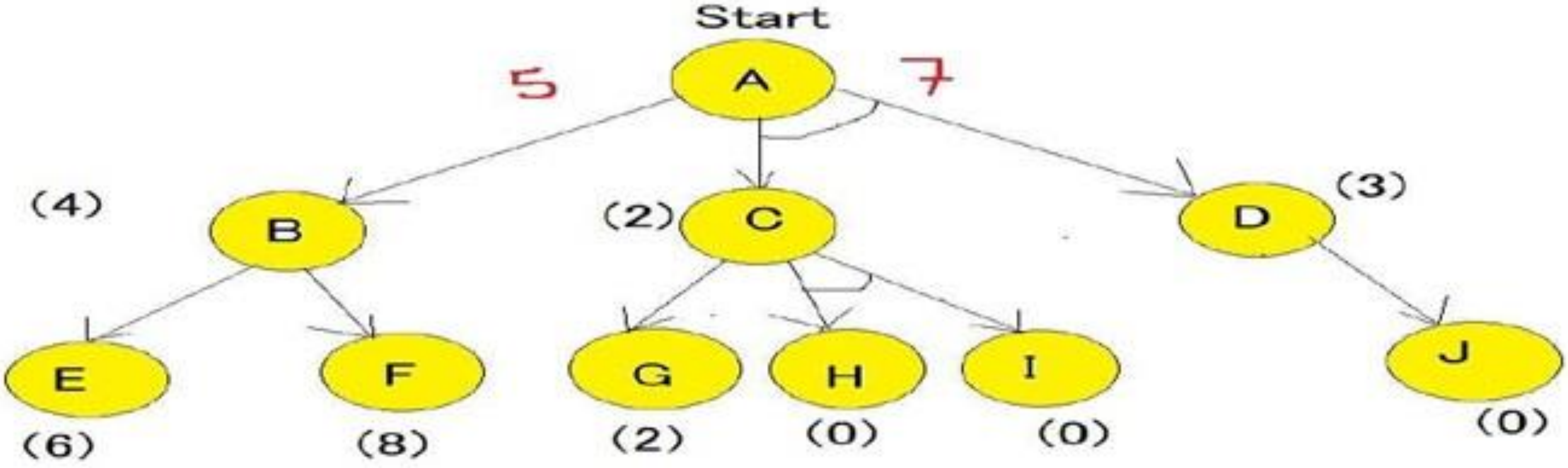


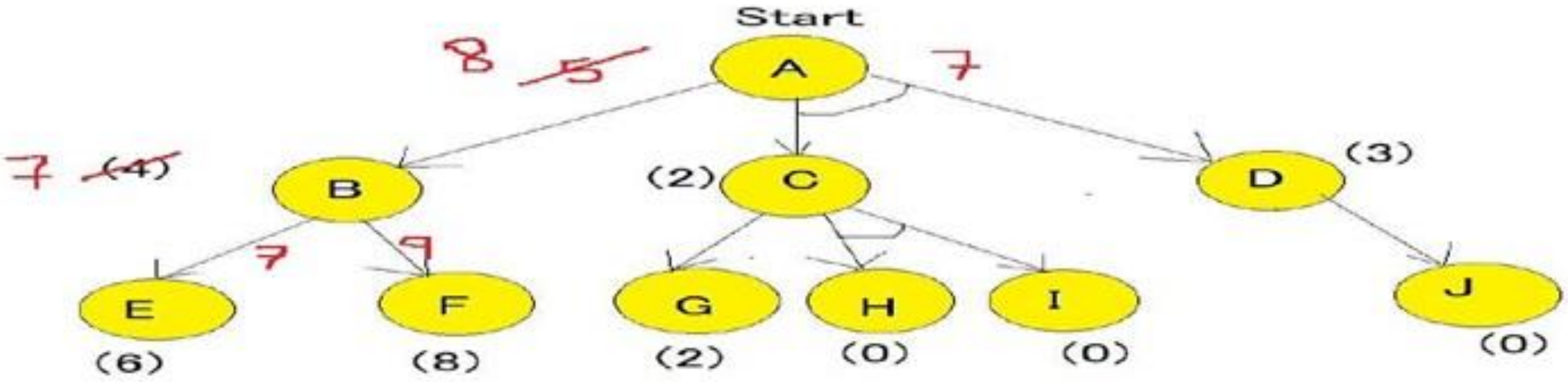
f(A -> C + D) = g(C) + h(C) + g(D) + h(D)
= 1 + 2 + 1 + 4 (C & D are AND nodes)
= 8

f(B -> E) = g(E) + h(E)
= 1 + 6 = 7

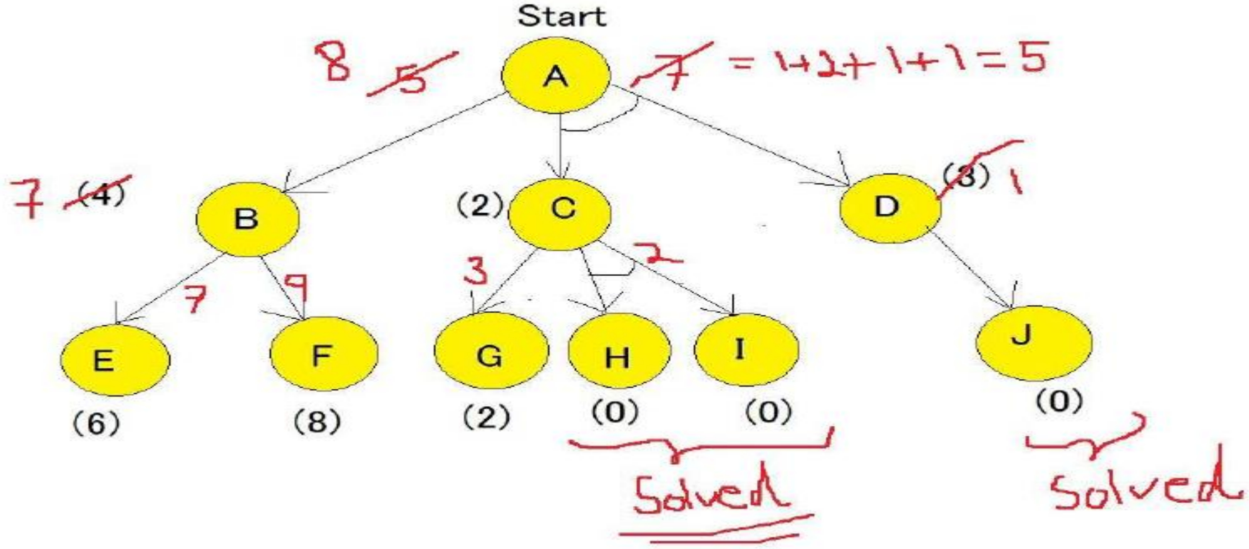
f(B -> F) = g(F) + h(F)
= 1 + 8
= 9

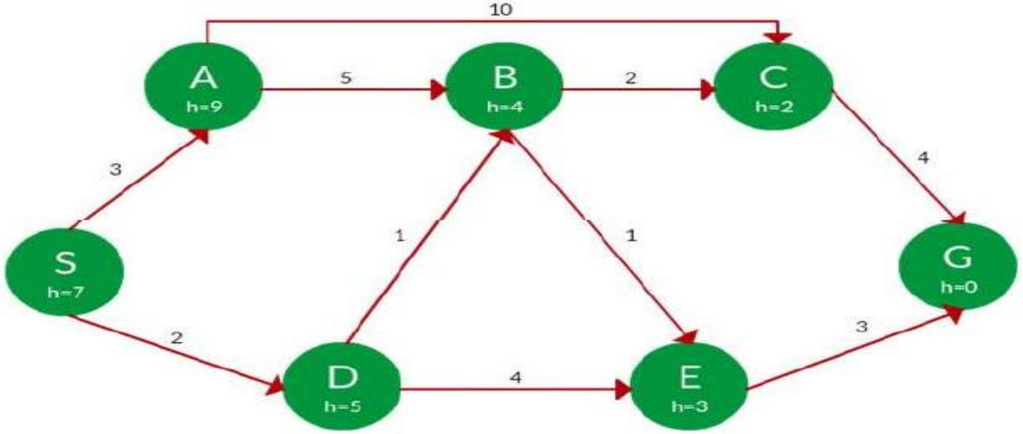
f(A -> B) = g(B) + updated h(B)
= 1 + 7
= 8


Start
A
5
7
B
(4)
C
(2)
D
(3)
E
(6)
F
(8)
G
(2)
H
(0)
I
(0)
J
(0)



Start
A
8
~~5~~
7
7
~~(4)~~
B
(2)
C
D
(3)
7
9
E
F
G
H
I
J
(6)
(8)
(2)
(0)
(0)
(0)


Start
A
8
~~5~~
~~7~~ = 1+2+1+1=5
B
7
~~(4)~~
C
(2)
D
~~(3)~~ 1
7
9
3
2
E
F
G
H
I
J
(6)
(8)
(2)
(0)
(0)
(0)
Solved
Solved

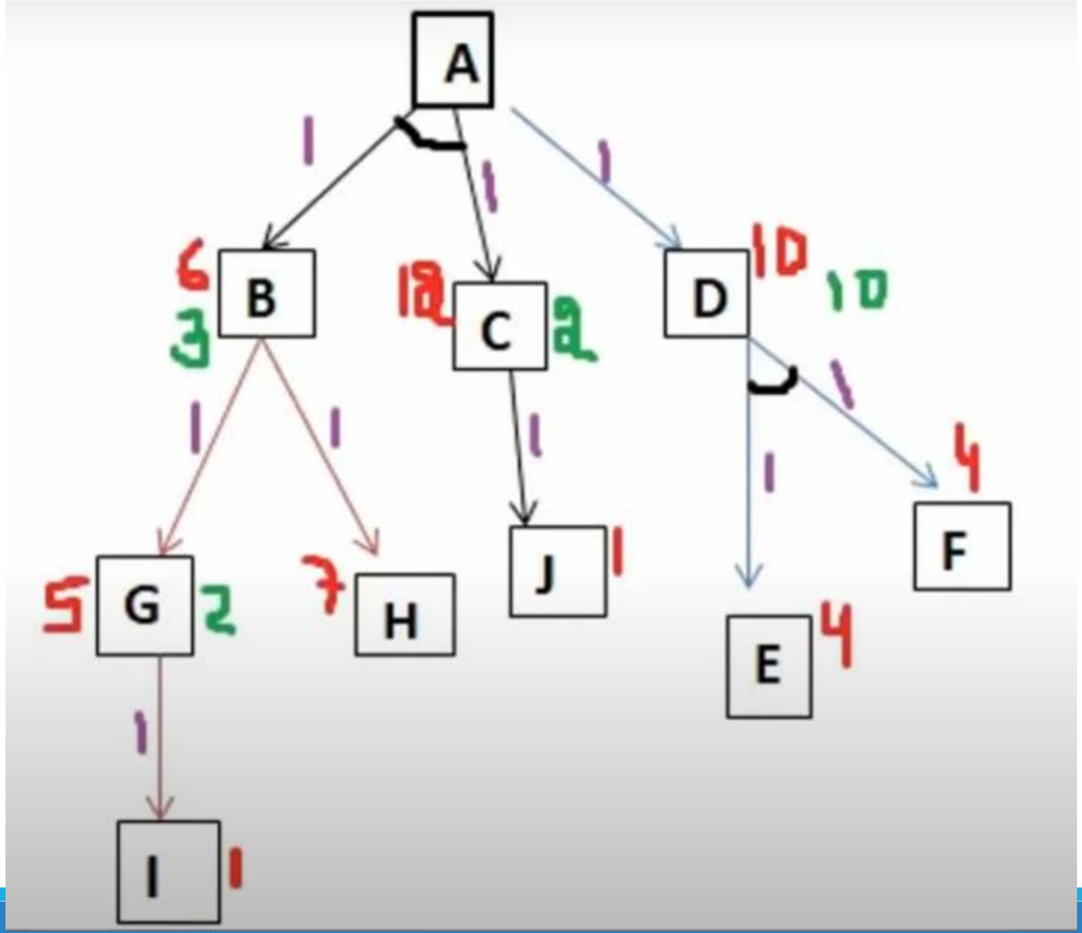

10
A
h=9
5
B
h=4
2
C
h=2
3
4
S
h=7
1
1
G
h=0
2
D
h=5
4
E
h=3
3

AO* Example

Solve the following problem using AO* Algorithm:

All the numbers written in brackets represent the heuristic value [h(n)].

Each edge is considered to have value of 1 by default [i.e g(n) = 1].



Solution

Comparison

A* Algorithm	AO* Algorithm
• Works on Best-First Search	• Works on Best-First Search
• Informed Search technique	• Informed Search technique
• Works on given Heuristic values	• Works on given Heuristic values
• Always gives Optimal Solution	• Does not guarantee Optimal Solution
• Even after reaching goal state further nodes are explored	• After reaching goal state further nodes are not explored
• Moderate Memory is used	• Comparatively less memory is used
• Endless loop possible	• Endless loop is not possible

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.

Constraint Satisfaction Problem

- 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 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.
- More formally, a CSP is defined as a triple (X,D,C), where:
  - X is a set of variables { x1, x2, ..., xn}.
  - D is a set of domains {D1 , D2 , ..., Dn}, where each Di is the set of possible values for xi.
  - C is a set of constraints {C1, C2, ..., Cm}, where each Ci is a constraint that restricts the values that can be assigned to a subset of the variables.

Constraint Satisfaction Problem

- **Crypt Arithmetic Problem**

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

Solution

Constraint Satisfaction Problem

- **Crypt Arithmetic Problem**

```
  CROSS
+ ROADS
-------
 DANGER
```

```
  DONALD
+ GERALD
--------
  ROBERT
```

Constraint Satisfaction Problem

- **Types of Constraints in CSP**

- 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.

Mean End Analysis

- Means end analysis (MEA) is an important concept in artificial intelligence (AI) because it enhances problem resolution. MEA solves problems by defining the goal and establishing the right action plan.

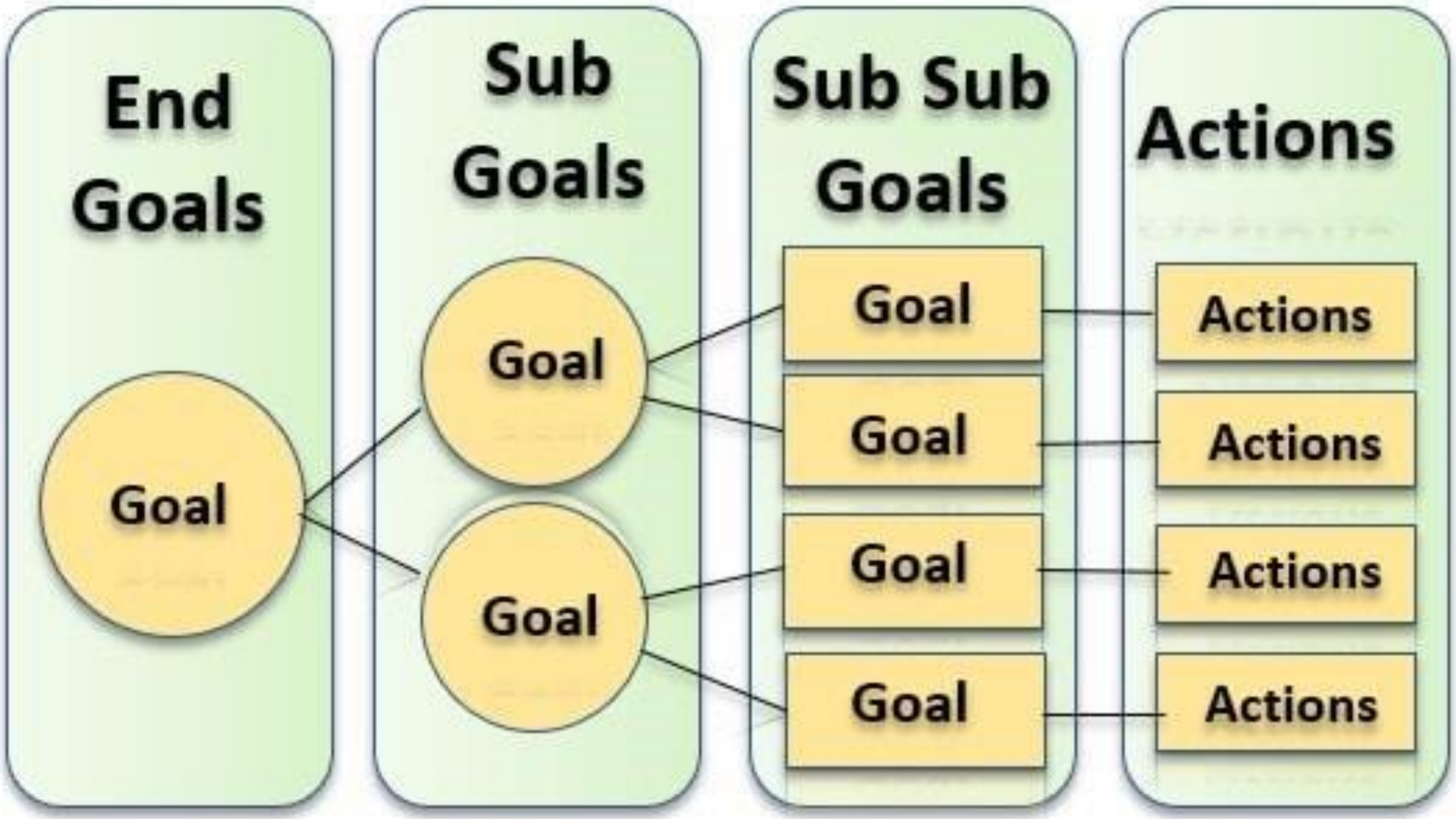
- This technique combines forward and backward strategies to solve complex problems. With these mixed strategies, complex problems can be tackled first, followed by smaller ones.

- In this technique, the system evaluates the differences between the current state or position and the target or goal state. It then decides the best action to be undertaken to reach the end goal.

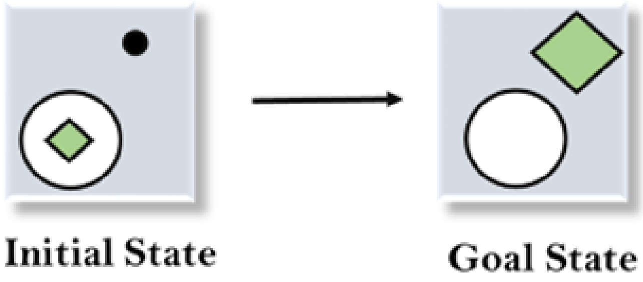
Mean End Analysis

1. First, the system evaluates the current state to establish whether there is a problem. If a problem is identified, then it means that an action should be taken to correct it.
2. The second step involves defining the target or desired goal that needs to be achieved.
3. The target goal is split into sub-goals, that are further split into other smaller goals.
4. This step involves establishing the actions or operations that will be carried out to achieve the end state.
5. In this step, all the sub-goals are linked with corresponding executable actions (operations).
6. After that is done, intermediate steps are undertaken to solve the problems in the current state. The chosen operators will be applied to reduce the differences between the current state and the end state.
7. This step involves tracking all the changes made to the actual state. Changes are made until the target state is achieved.

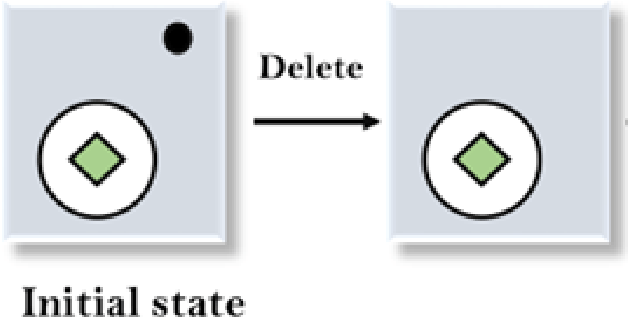
Mean End Analysis



Mean End Analysis

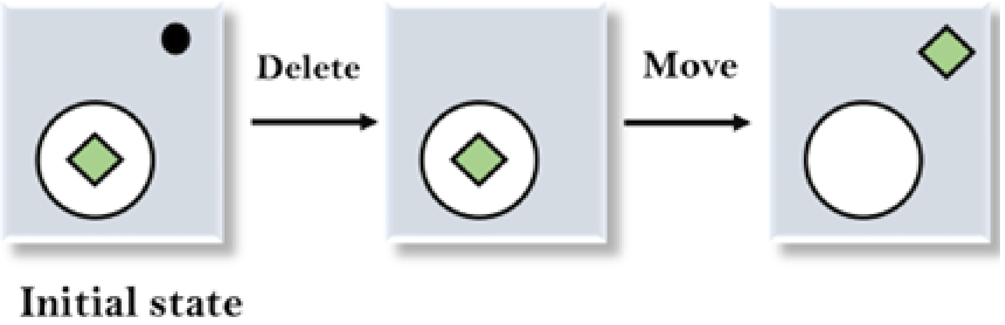


1. Delete Operator: The dot symbol at the top right corner in the initial state does not exist in the goal state. The dot symbol can be removed by applying the delete operator.

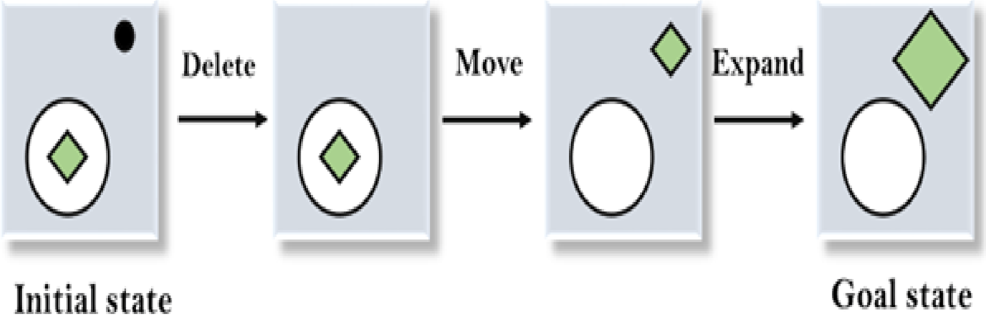


Mean End Analysis

2. Move operator: We will then compare the new state with the end state. The green diamond in the new state is inside the circle while the green diamond in the end state is at the top right corner. We will move this diamond symbol to the right position by applying the move operator.



3. Expand operator: After evaluating the new state generated in step 2, we find that the diamond symbol is smaller than the one in the end state. We can increase the size of this symbol by applying the expand operator.



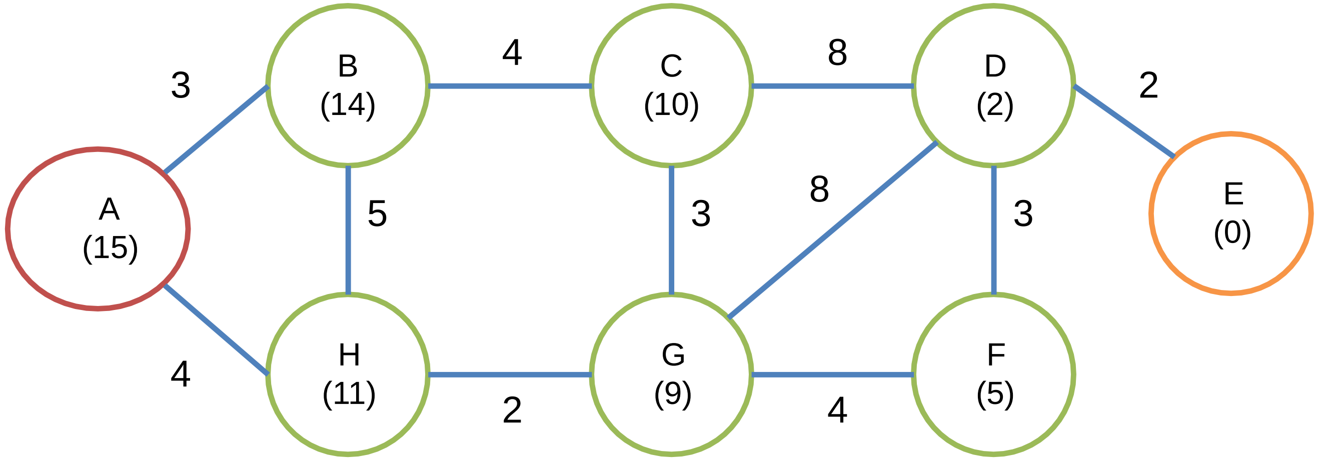
Unit-2 Mean End Analysis

Application of MEA:

1. Organizational planning
2. Business transformation
3. Gap analysis

programming assignment

- Based on the skeleton problem, complete a program containing the best-first search, A* algorithm, AO* and IDA*.
- See the figure below. We suppose that A is the initial node, and E is the target node. The cost of each edge and the heuristic value of each node are also given.


A
(15)
B
(14)
C
(10)
D
(2)
E
(0)
H
(11)
G
(9)
F
(5)
3
4
4
8
2
5
3
8
3
2
4