Search Algorithms In Artificial Intelligence

Course Name: Artificial Intelligence

Course code: CSE-403 [SECTION - A]

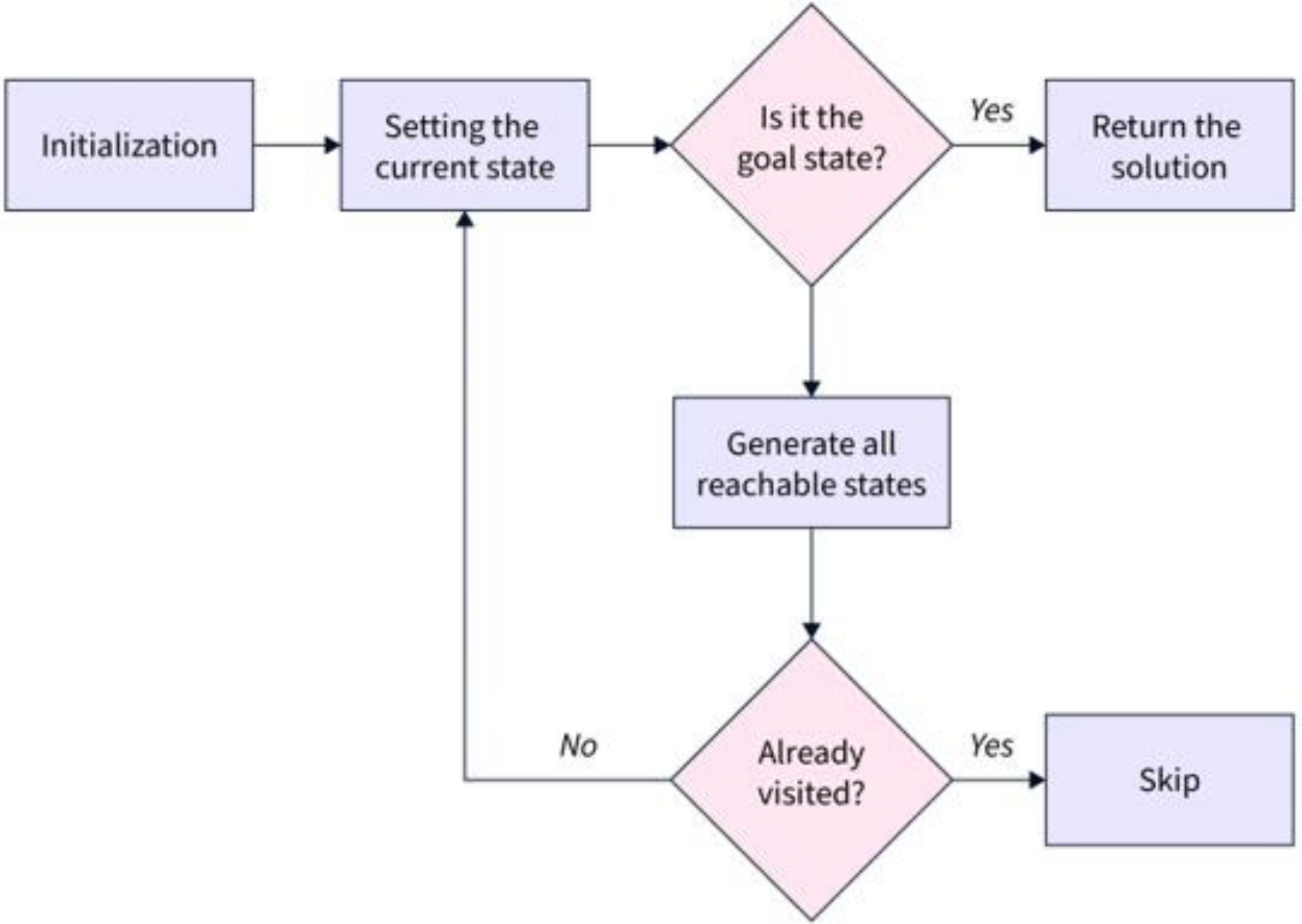
Outline:-

- State Space Search
- 8-puzzle Problem Without Heuristic
- Un-informed Search Algos:
    1. Breadth First Search
    2. Depth First Search
    3. Depth-limited Search Algorithm:
    4. Uniform Cost Search
    5. Iterative Deepening Depth-first Search
    6. Bidirectional Search Algorithm
- Informed Search:
    1. Best-first Search Algorithm
    2. Beam search
    3. A* Search Algorithm
- Hill Climbing
- Simulated Annealing
- Constraint Satisfaction Problem In AI

State Space Search

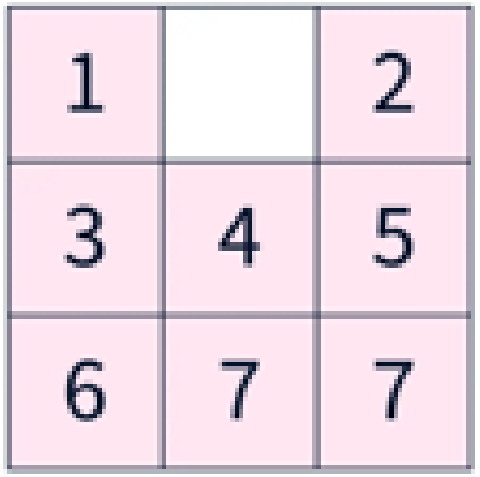
- ***Artificial Intelligence*** is the study of building agents that act rationally. A search problem consists of:
    - **A State Space.** Set of all possible states where you can be.
    - **A Start State.** The state from where the search begins.
    - **A Goal State.** A function that looks at the current state returns whether or not it is the goal state.
- The **Solution** to a search problem is a sequence of actions, called the **plan** that transforms the start state to the goal state. This plan is achieved through search algorithms.
- A **state space** is a way to mathematically represent a problem by defining all the possible states in which the problem can be. This is used in search algorithms to represent the initial state, goal state, and current state of the problem. Each state in the state space is represented using a set of variables.
- The **efficiency** of the search algorithm greatly depends on the size of the state space, and it is important to choose an appropriate representation and search strategy to search the state space efficiently.
- One of the most well-known **state space search algorithms** is the A algorithm. Other commonly used state space search algorithms include **breadth-first search (BFS)**, **depth-first search (DFS)**, **hill climbing**, **simulated annealing**, and **genetic algorithms**.

Steps in State Space Search

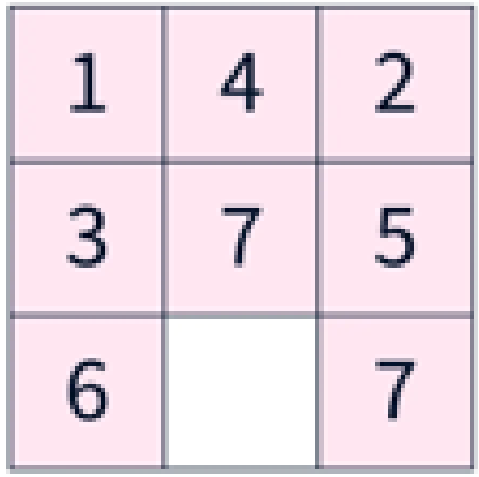


Example Of State Space Search: 8-puzzle Problem Without Heuristic

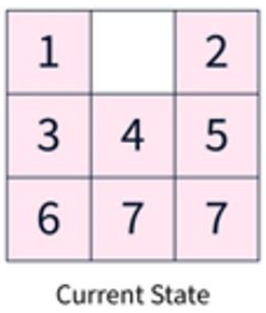
- The **8-puzzle** problem is a commonly used example of a state space search. It is a sliding puzzle game consisting of 8 numbered tiles arranged in a 3x3 grid and one blank space. The game aims to rearrange the tiles from their initial state to a final goal state by sliding them into the blank space.
- To represent the state space in this problem, we use the nine tiles in the puzzle and their respective positions in the grid. Each state in the state space is represented by a 3x3 array with values ranging from 1 to 8, and the blank space is represented as an empty tile.
- The initial state of the puzzle represents the starting configuration of the tiles, while the goal state represents the desired configuration. **Search algorithms** utilize the state space to find a sequence of moves that will transform the initial state into the goal state.



Current State

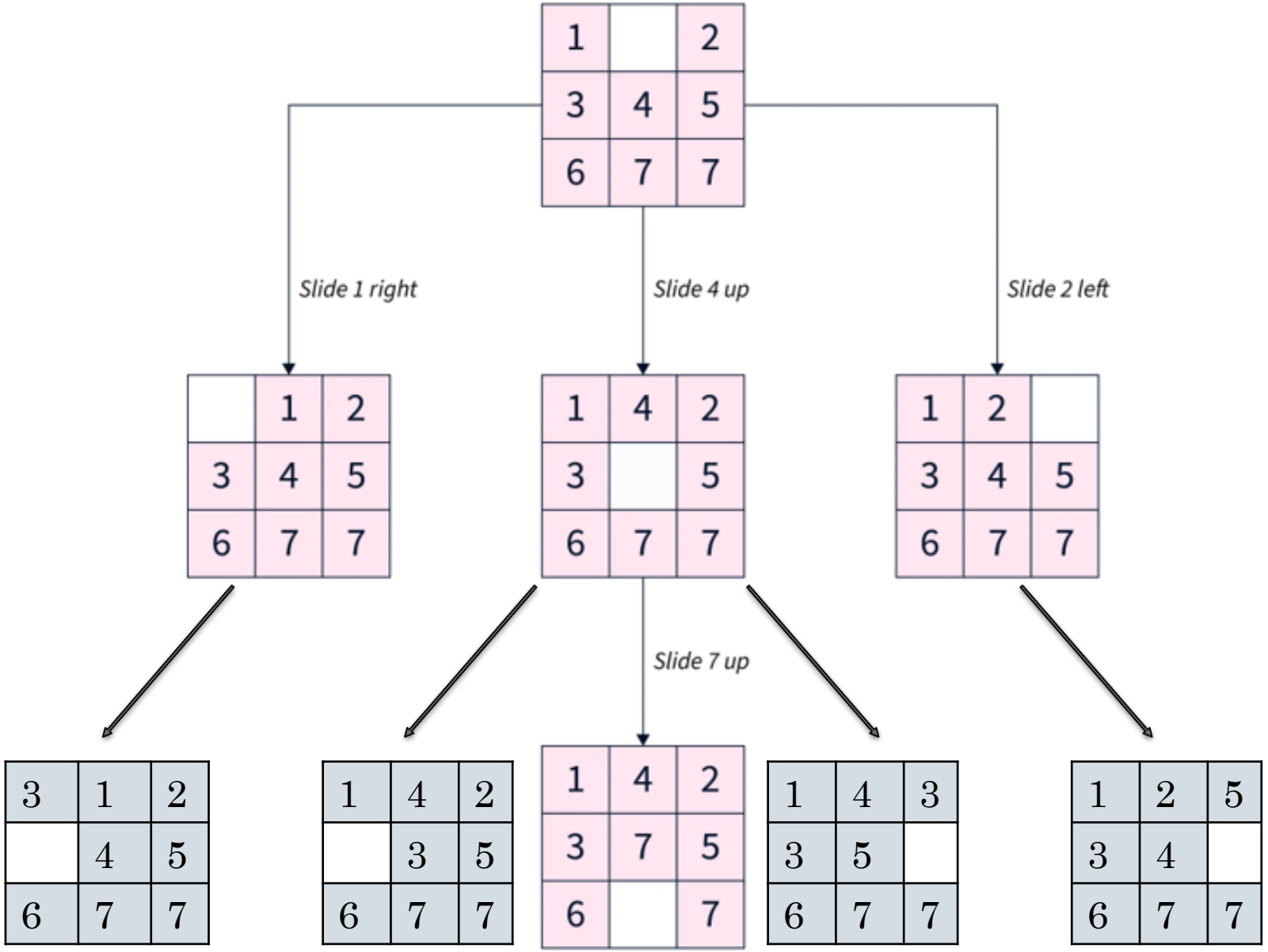


Target State



Current State

Target State



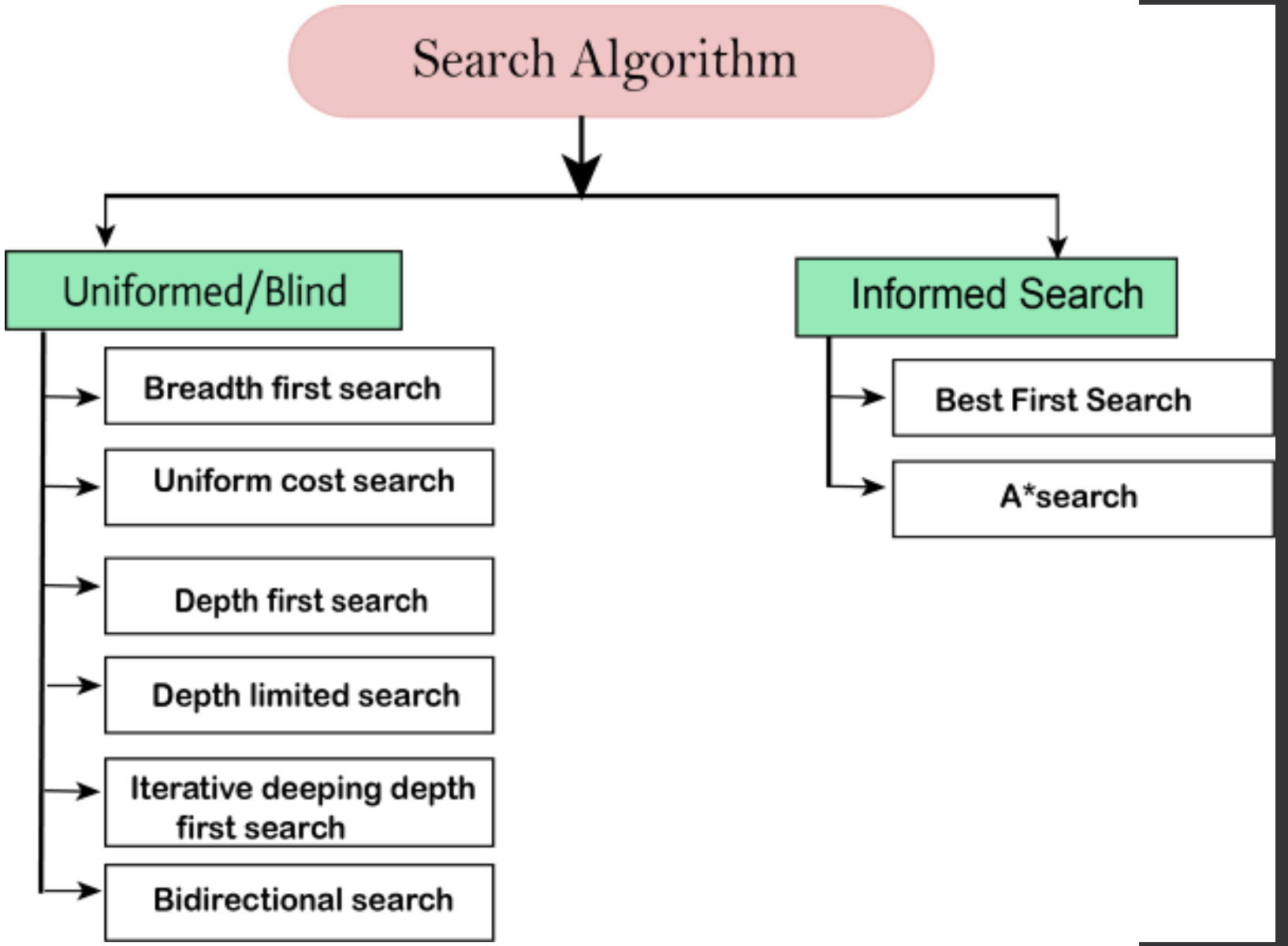
Goal State

Search Algorithms in Artificial Intelligence

Search algorithms are one of the most important areas of Artificial Intelligence. This topic will explain all about the 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 Search Algorithms:

The search algorithms in this section have no additional information on the goal node other than the one provided in the problem definition. The plans to reach the goal state from the start state differ only by the order and/or length of actions. Uninformed search is also called **Blind search**. These algorithms can only generate the successors and differentiate between the goal state and non goal state.

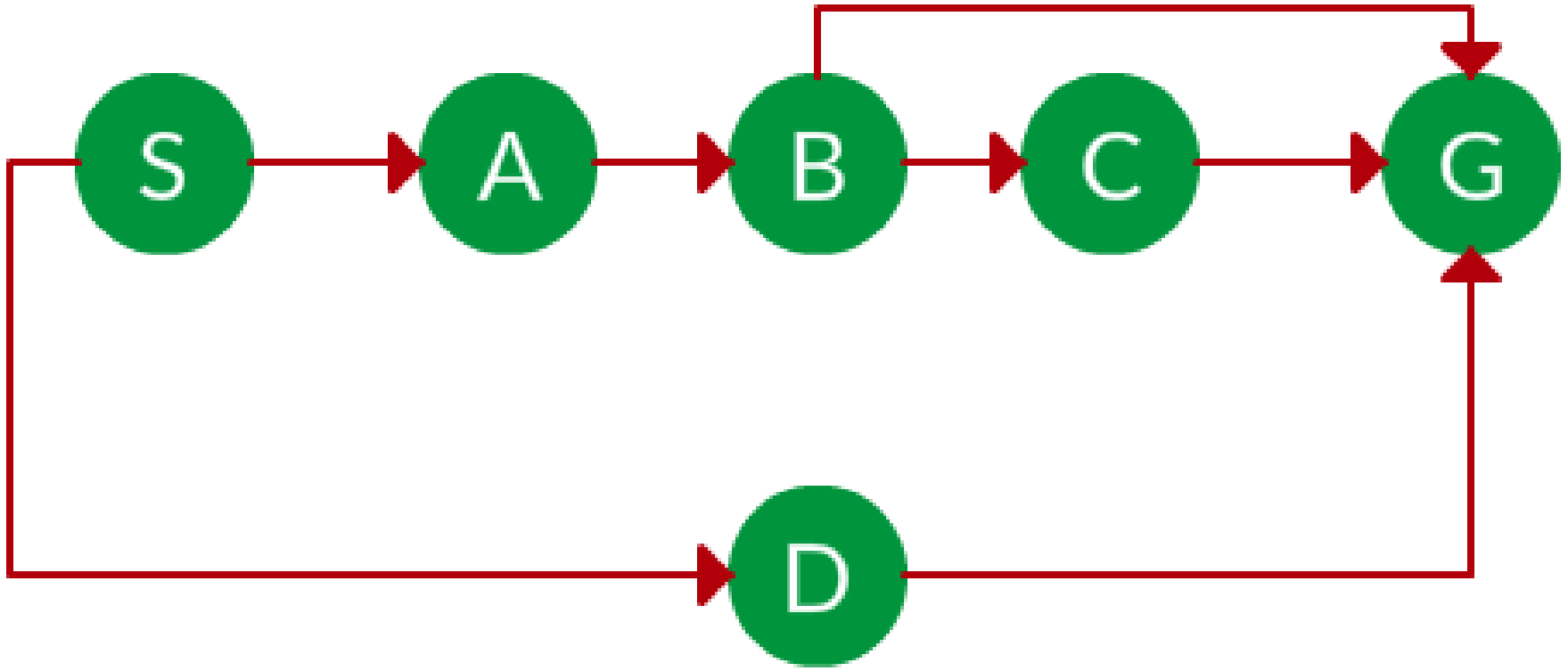
Each of these algorithms will have:

1. A problem **graph,** containing the start node S and the goal node G.
2. A **strategy,** describing the manner in which the graph will be traversed to get to G.
3. A **fringe,** which is a data structure used to store all the possible states (nodes) that you can go from the current states.
4. A **tree,** that results while traversing to the goal node.
5. A solution **plan,** which the sequence of nodes from S to G.

Breadth First Search:

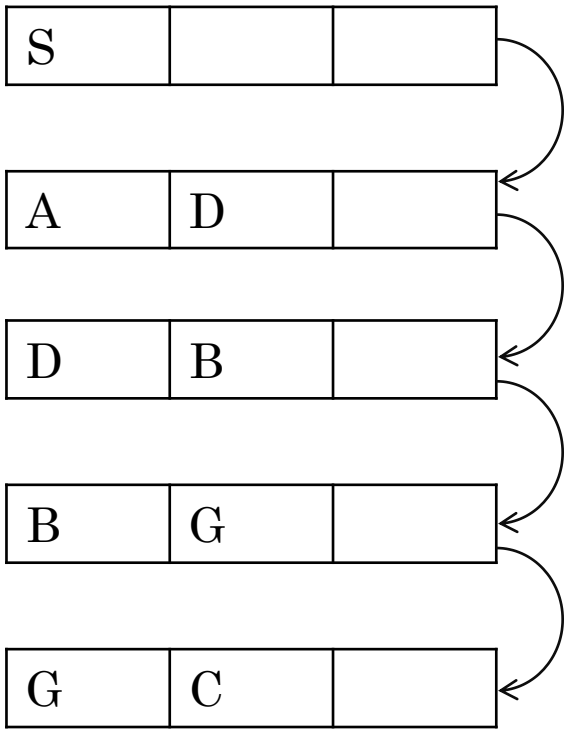
- Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a ‘search key’), and explores all of the neighbor nodes at the present depth prior to moving on to the nodes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

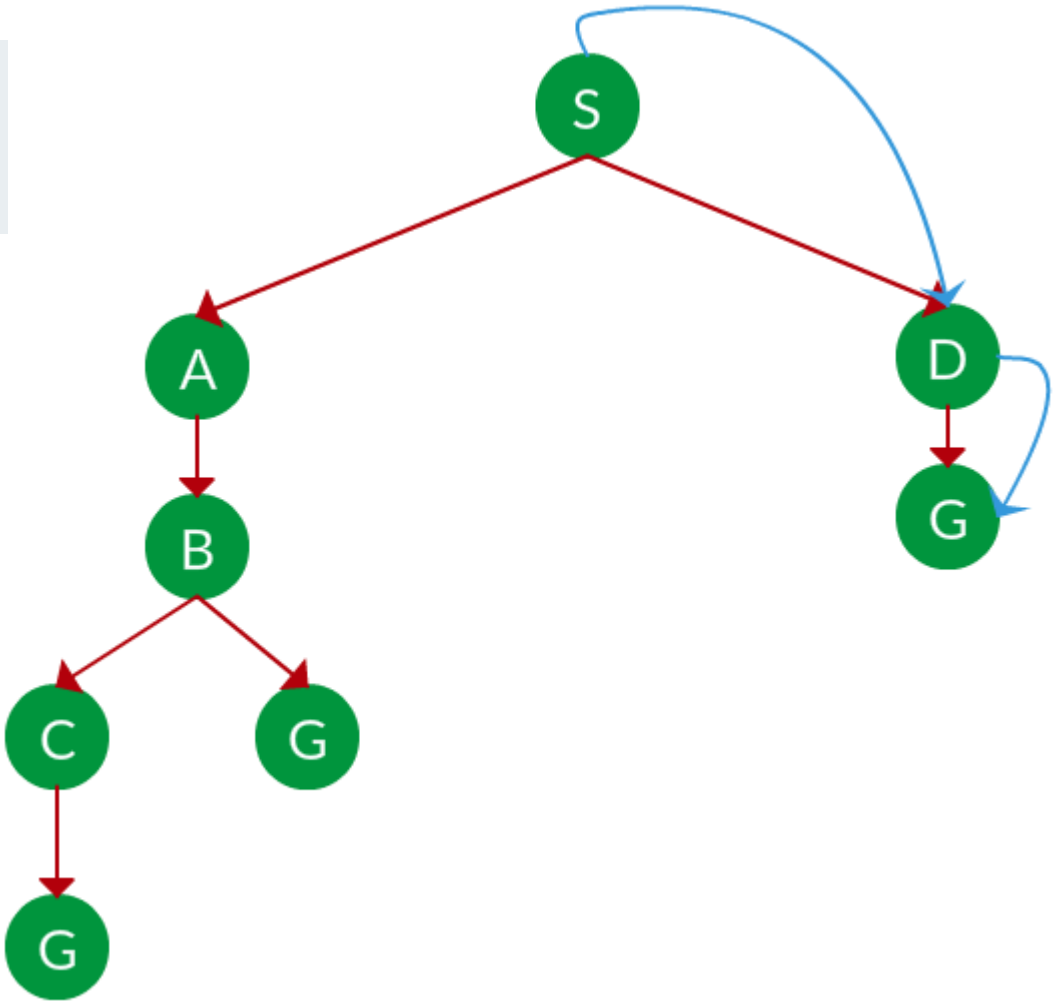
Ex-1. Which solution would BFS find to move from node S to node G if run on the graph below?



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.

Visit S
→Delete S from the Queue
→Insert at the last, Nodes connected with S





Example-2:

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

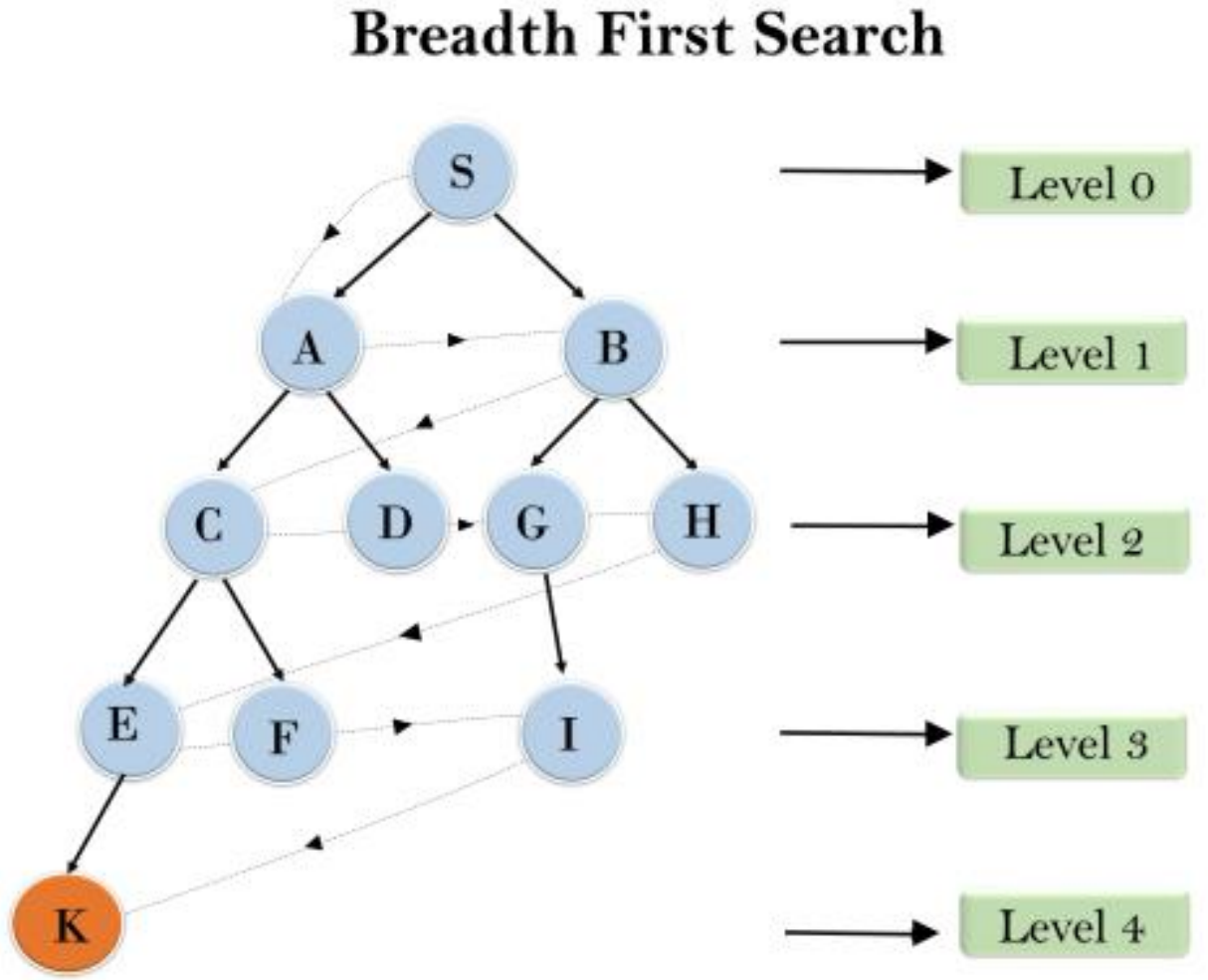
Time Complexity: Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node. Where the d= depth of shallowest solution and b is a node at every state.

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

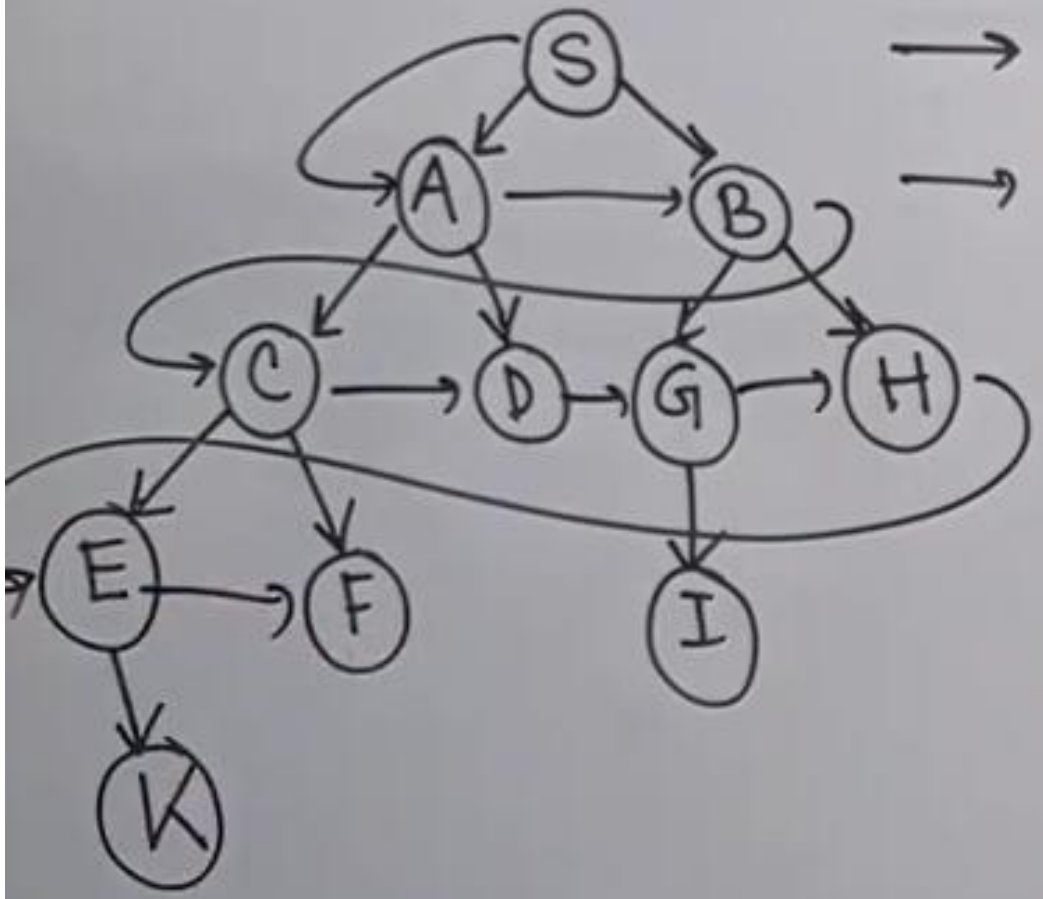
Space Complexity: Space complexity of BFS algorithm is given by the Memory size of frontier which is $O(b^d)$.

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.

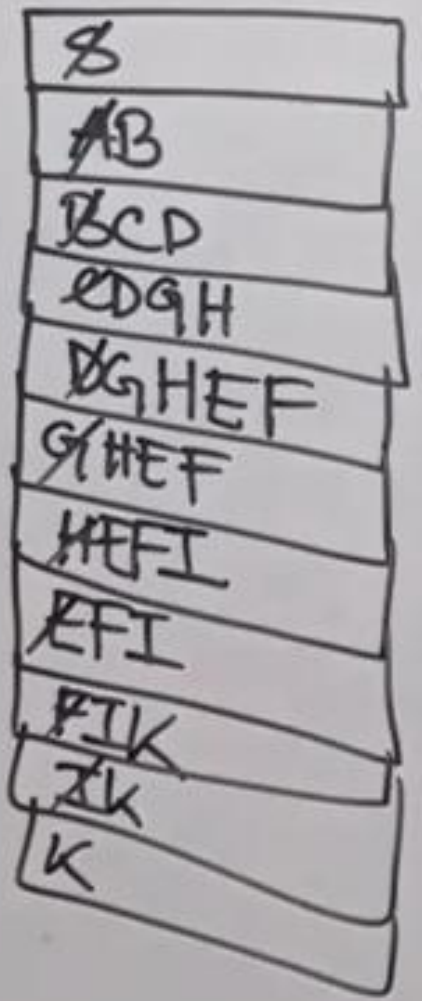


Example:- Root node (S) and Goal node (K).



→ Level 0

→ Level 1

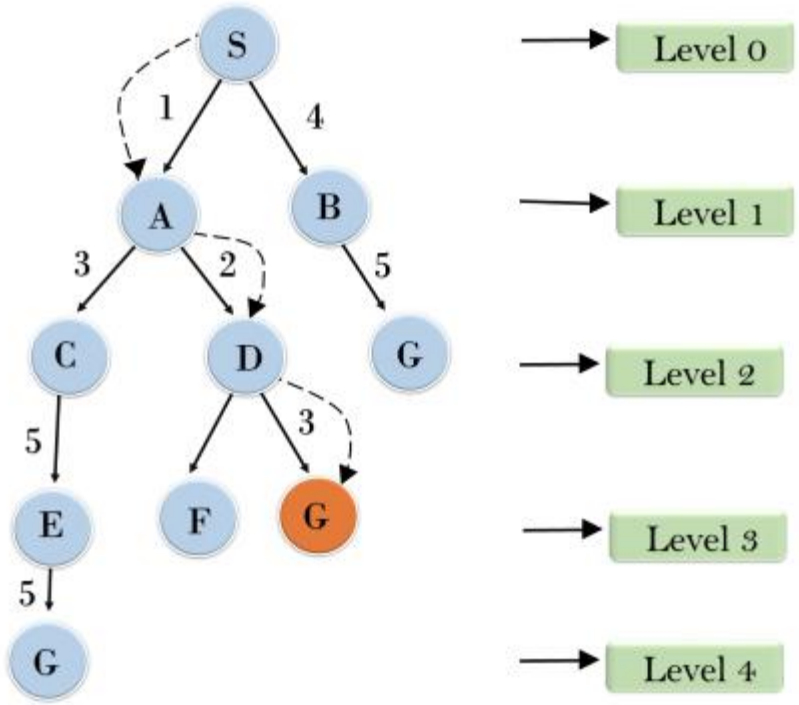


Uniform-cost Search Algorithm

- 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.

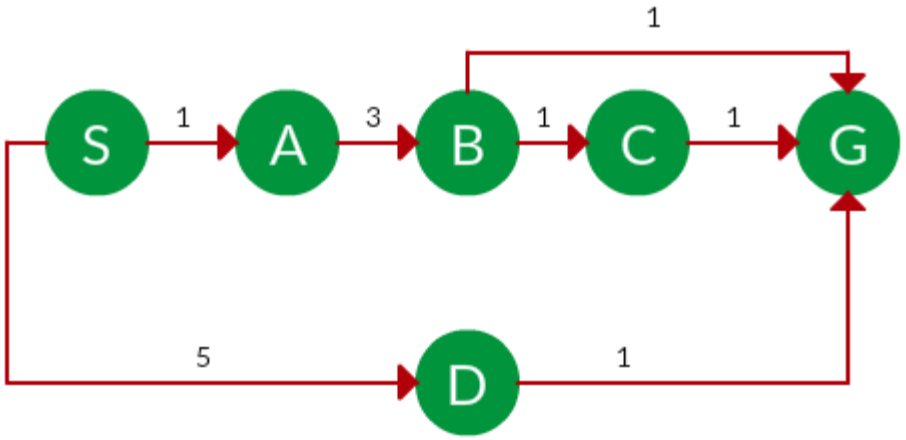
- It can be used to solve any graph/tree where the optimal cost is in demand. A uniform-cost search algorithm is implemented by the priority queue. It gives maximum priority to the lowest cumulative cost. Uniform cost search is equivalent to BFS algorithm if the path cost of all edges is the same.

Example-1: **Uniform Cost Search**

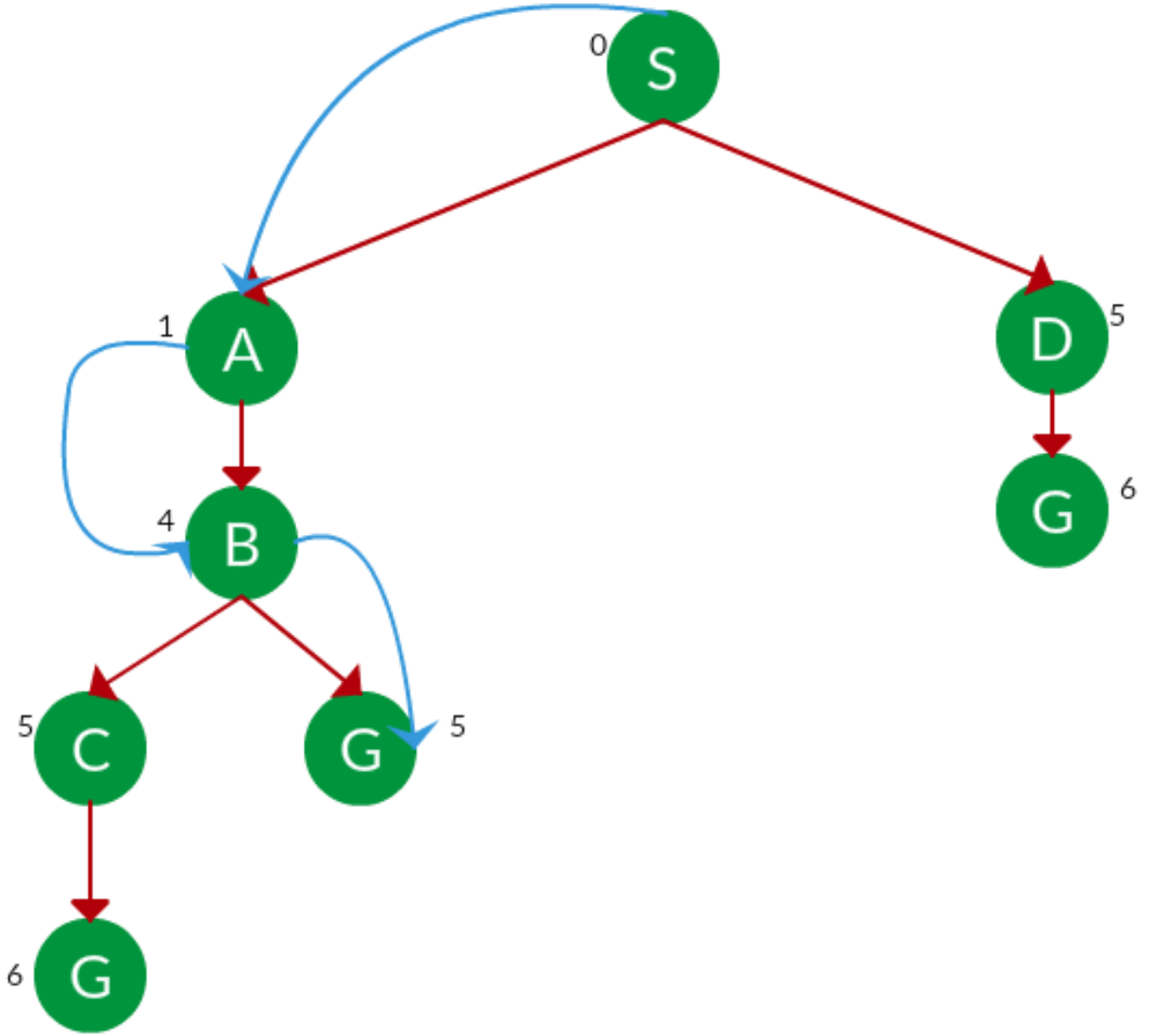


Completeness: Uniform-cost search is complete, such as if there is a solution, UCS will find it.
Time Complexity:
Let C* **is Cost of the optimal solution**, and **ε** is each step to get closer to the goal node. Then the number of steps is = C*/ε+1. Here we have taken +1, as we start from state 0 and end to C*/ε.
Hence, the worst-case time complexity of Uniform-cost search is$\mathbf{O(b^{1+[C^*/\varepsilon]})/}$.
Space Complexity:
The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is $\mathbf{O(b^{1+[C^*/\varepsilon]})}$.
Optimal: Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

- **Example-2:**
  Question. Which solution would UCS find to move from node S to node G if run on the graph below?



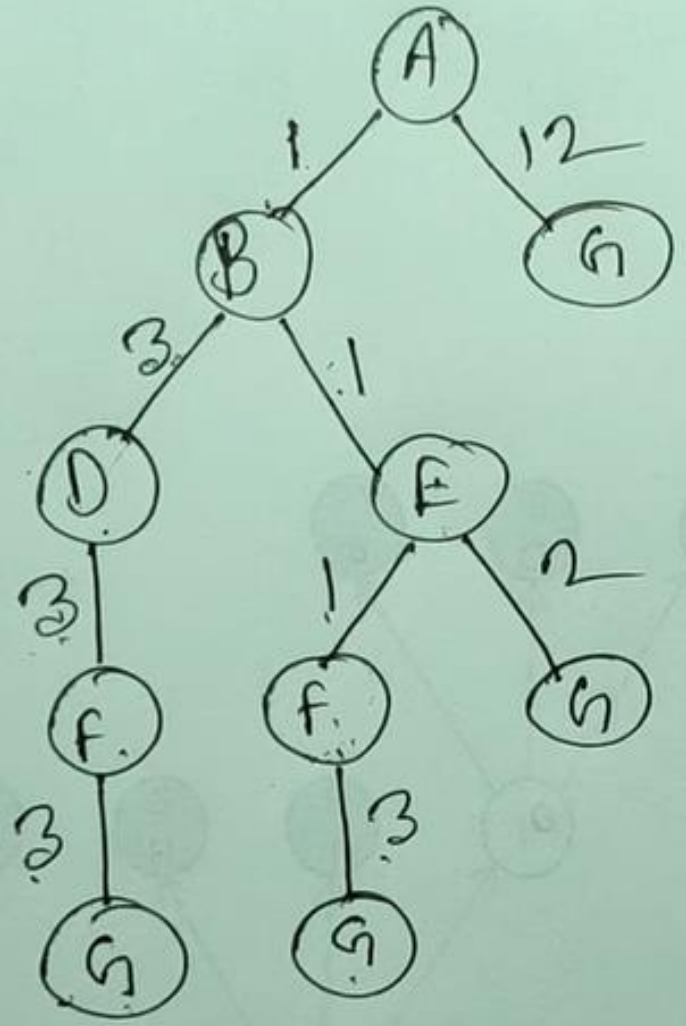
Solution. The equivalent search tree for the above graph is as follows. The cost of each node is the cumulative cost of reaching that node from the root. Based on the UCS strategy, the path with the least cumulative cost is chosen. Note that due to the many options in the fringe, the algorithm explores most of them so long as their cost is low, and discards them when a lower-cost path is found; these discarded traversals are not shown below. The actual traversal is shown in blue.



Path: *S -> A -> B -> G*
Cost: *5*

Example-3:

goal = G



A → B = 1

A → G = 12

A → B → E = 2

A → B → D = 4

A → B → E → F = 3

A → B → E → G = 4

A → B → D → F = 7

A → B → E → F → G = 6

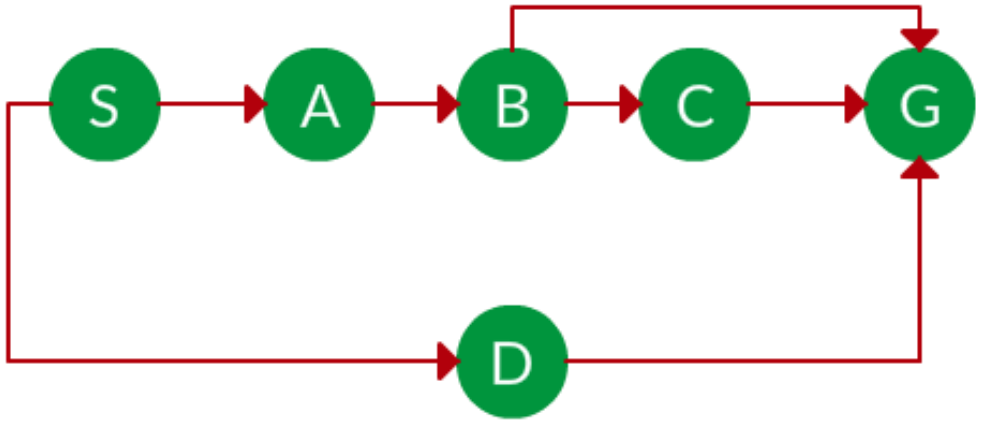
A → B → D → F → G = 10

A → B → E → D → f → G → F

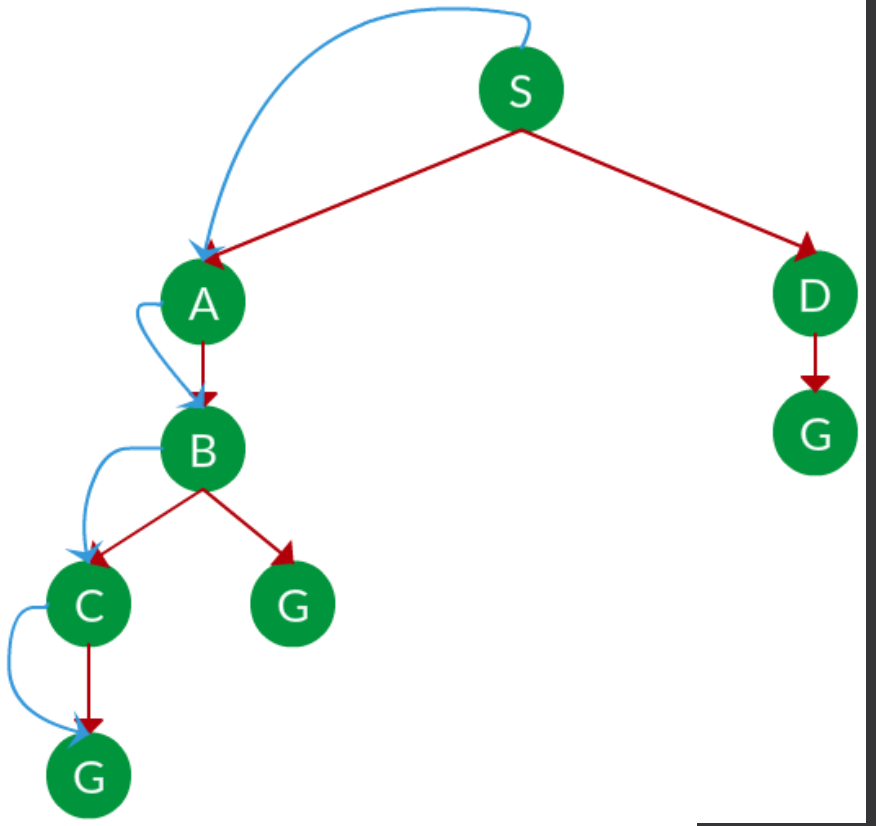
Depth First Search:

- 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-1:** 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.

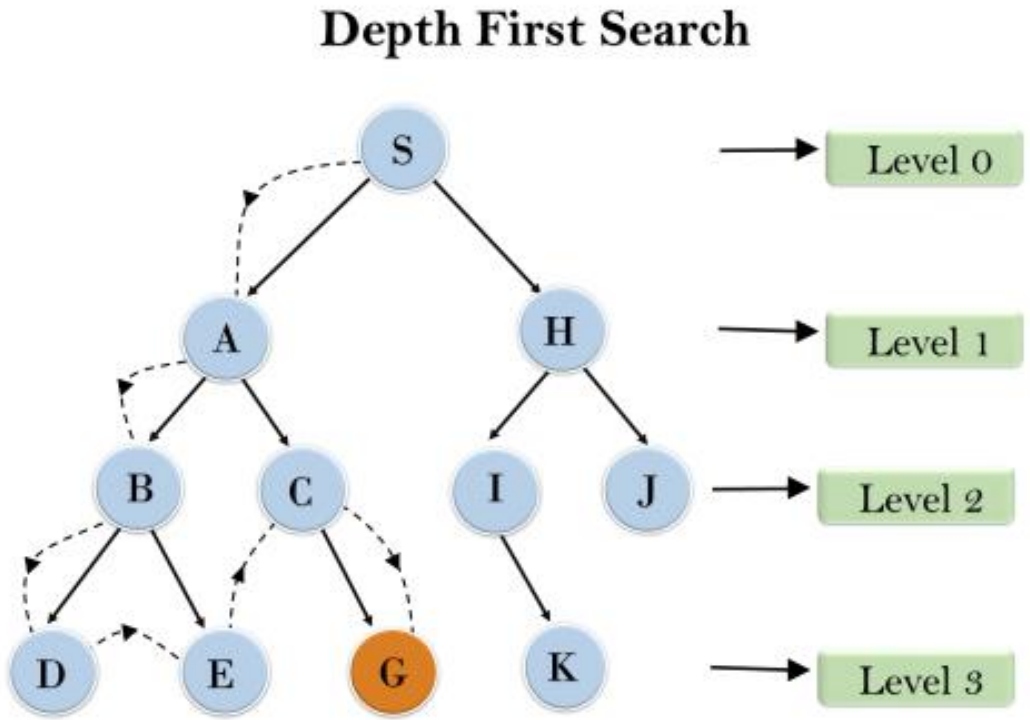


Path: S -> A -> B -> C -> G

- **Example-2:**

  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)**.

Depth-Limited Search

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.

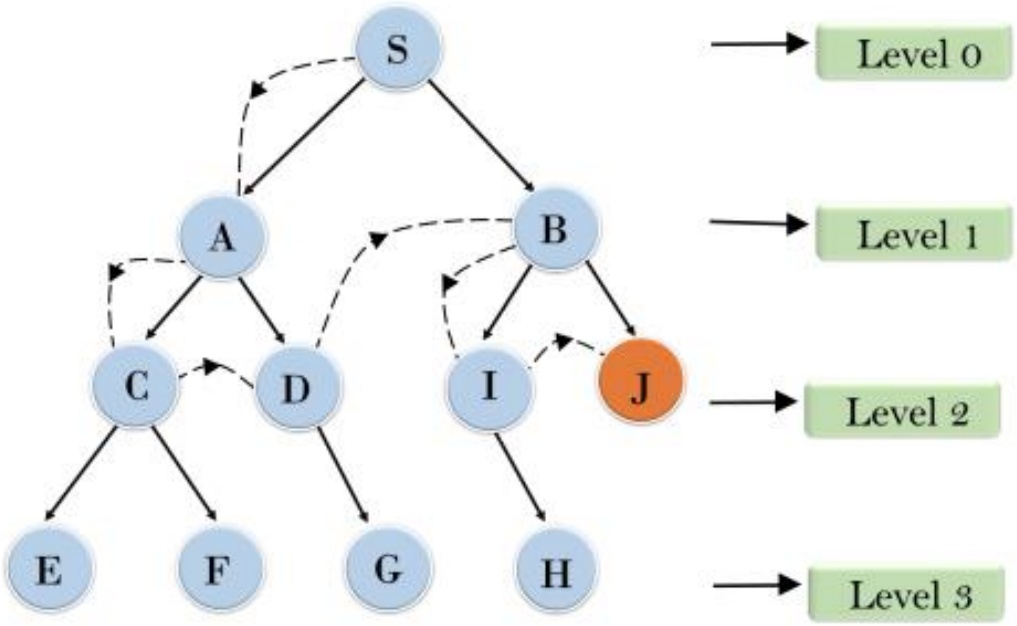
Completeness: DLS search algorithm is complete if the solution is above the depth-limit.

Time Complexity: Time complexity of DLS algorithm is **$O(b^{\ell})$**.

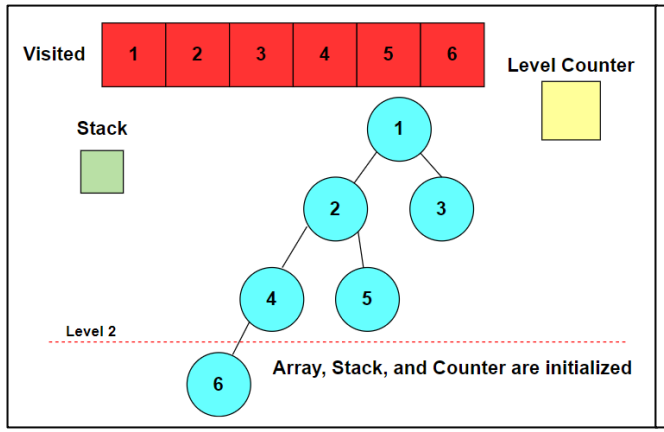
Space Complexity: Space complexity of DLS algorithm is **$O(b \times \ell)$**.

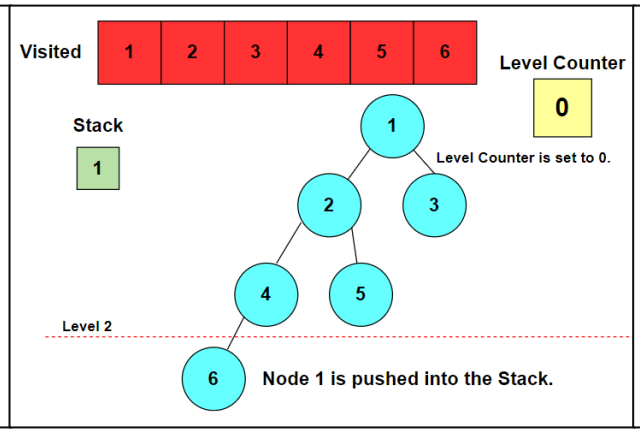
Optimal: Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if $\ell > d$.

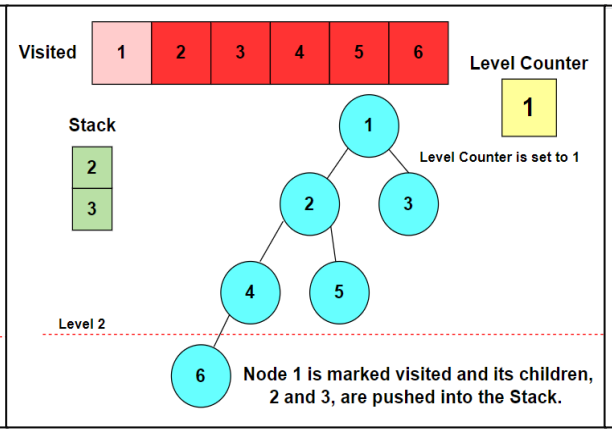
Example-1 Depth Limited Search

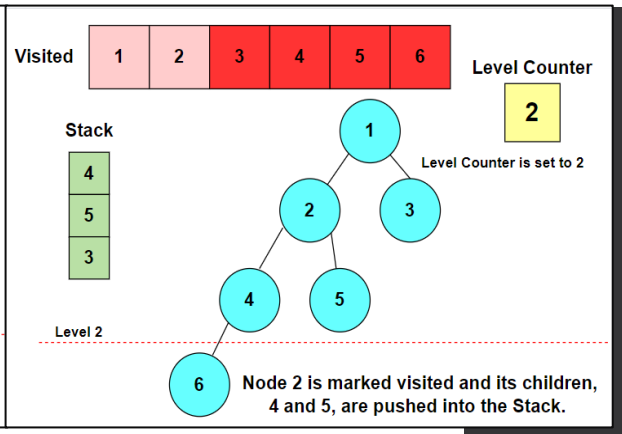


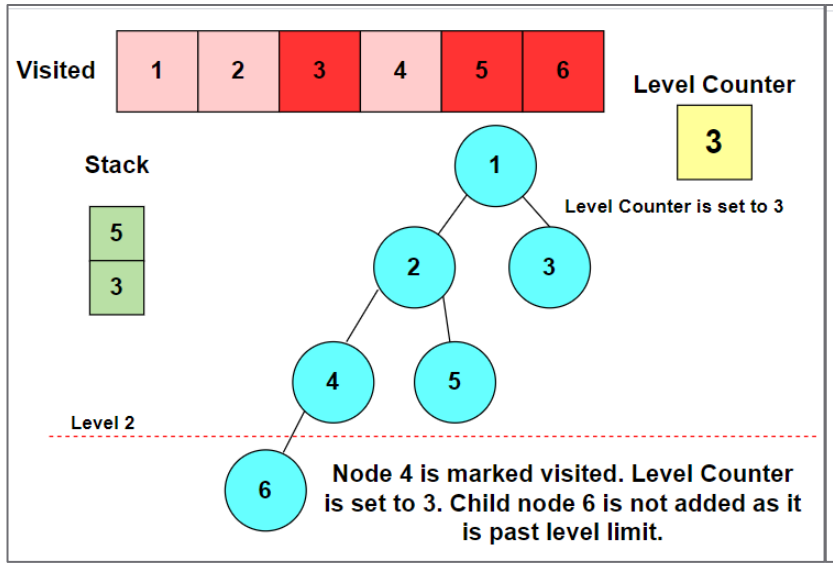
Example-2:- Let's look at the following tree with six nodes. Our target node is 5. For this example, we'll use a level limit of two as we traverse the tree. We use a visited array to mark off nodes we have already traversed through. This keeps us from visiting the same node multiple times

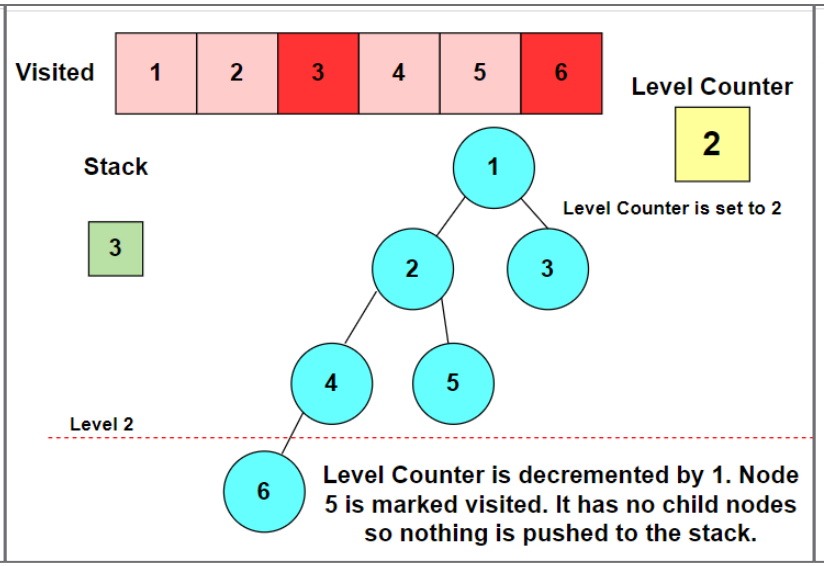


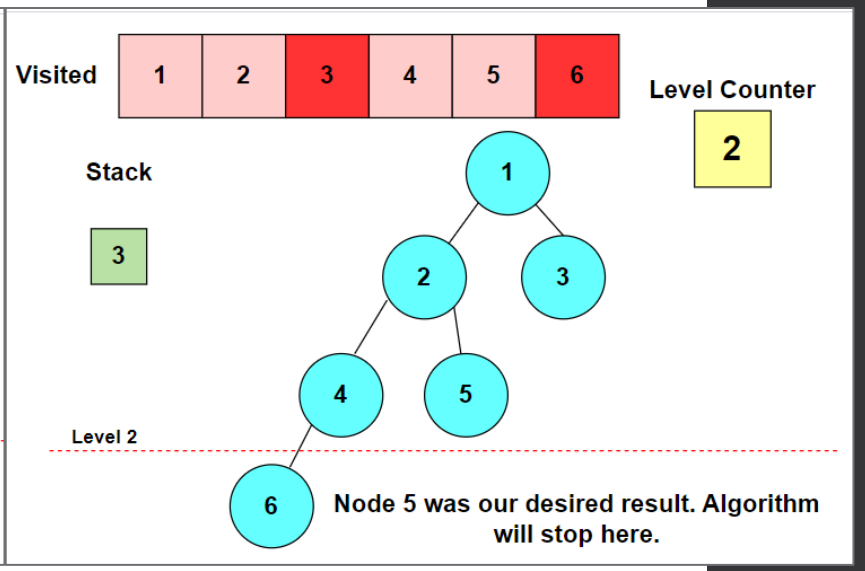








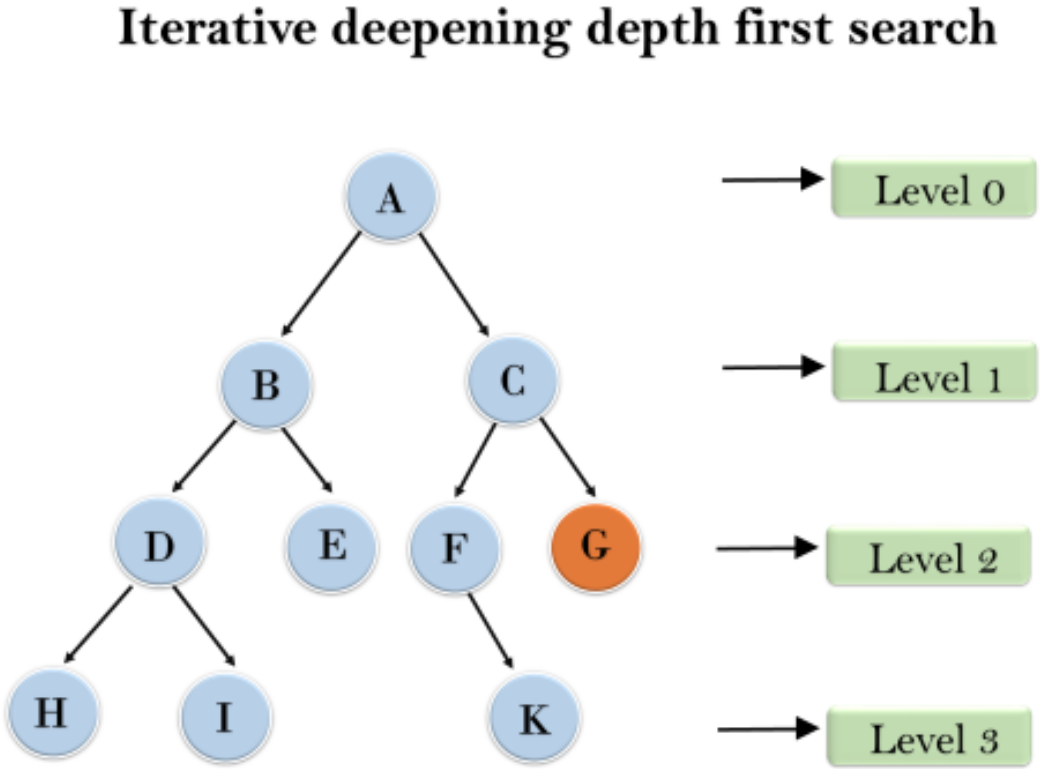




Iterative Deepening Depth-first Search

- 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.

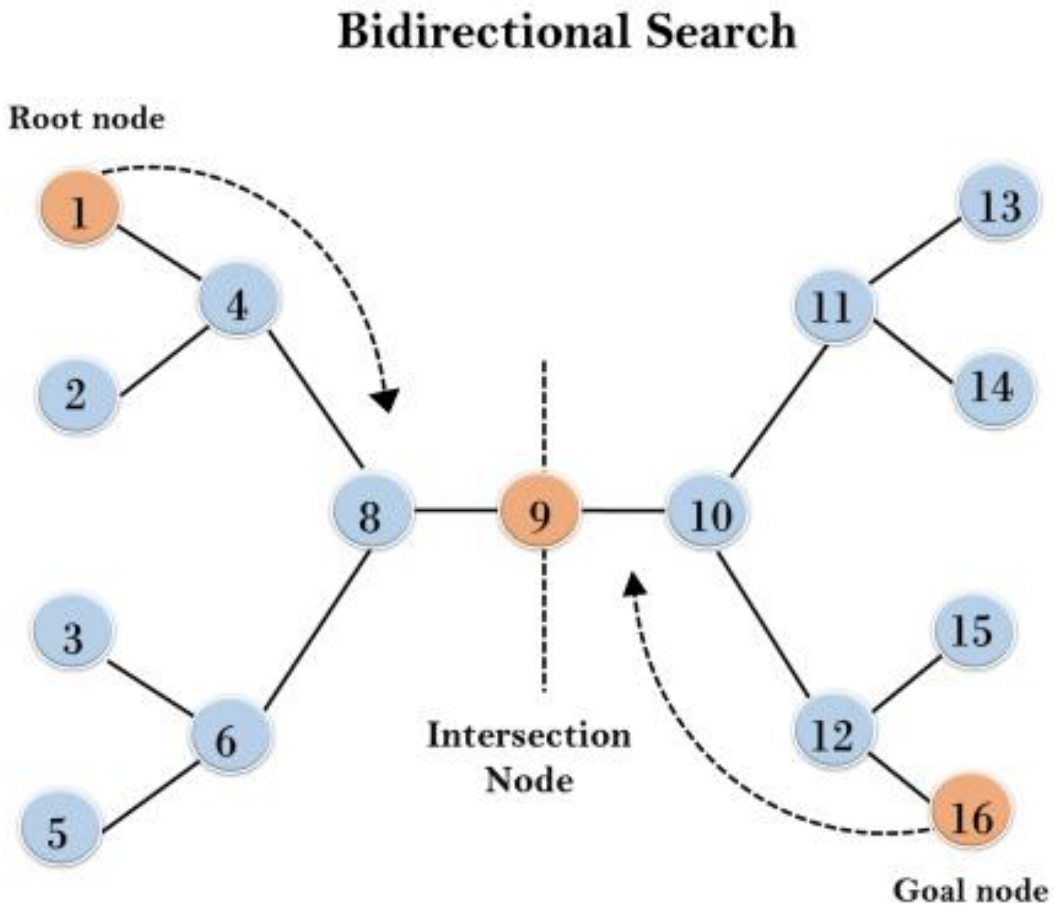
Example-1:
Following tree structure is showing the iterative deepening depth-first search. IDDFS algorithm performs various iterations until it does not find the goal node. The iteration performed by the algorithm is given as:
1'st Iteration-----> A
2'nd Iteration----> A, B, C
3'rd Iteration------>A, B, D, E, C, F, G
4'th Iteration------>A, B, D, H, I, E, C, F, K, G
In the fourth iteration, the algorithm will find the goal node.



completeness: This algorithm is complete is if the branching factor is finite.
Time Complexity: Let's suppose b is the branching factor and depth is d then the worst-case time complexity is O(bd).
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.
- **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.



Informed Search Algorithms:

- Here, the algorithms have information on the goal state, which helps in more efficient searching. This information is obtained by something called a *heuristic*.
  Search Heuristics: In an informed search, a heuristic is a *function* that estimates how close a state is to the goal state. For example – Manhattan distance, Euclidean distance, etc. (Lesser the distance, closer the goal.) Different heuristics are used in different informed algorithms discussed below. 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:**

- $h(n) <= h^*(n)$

- **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.**

- Pure 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.

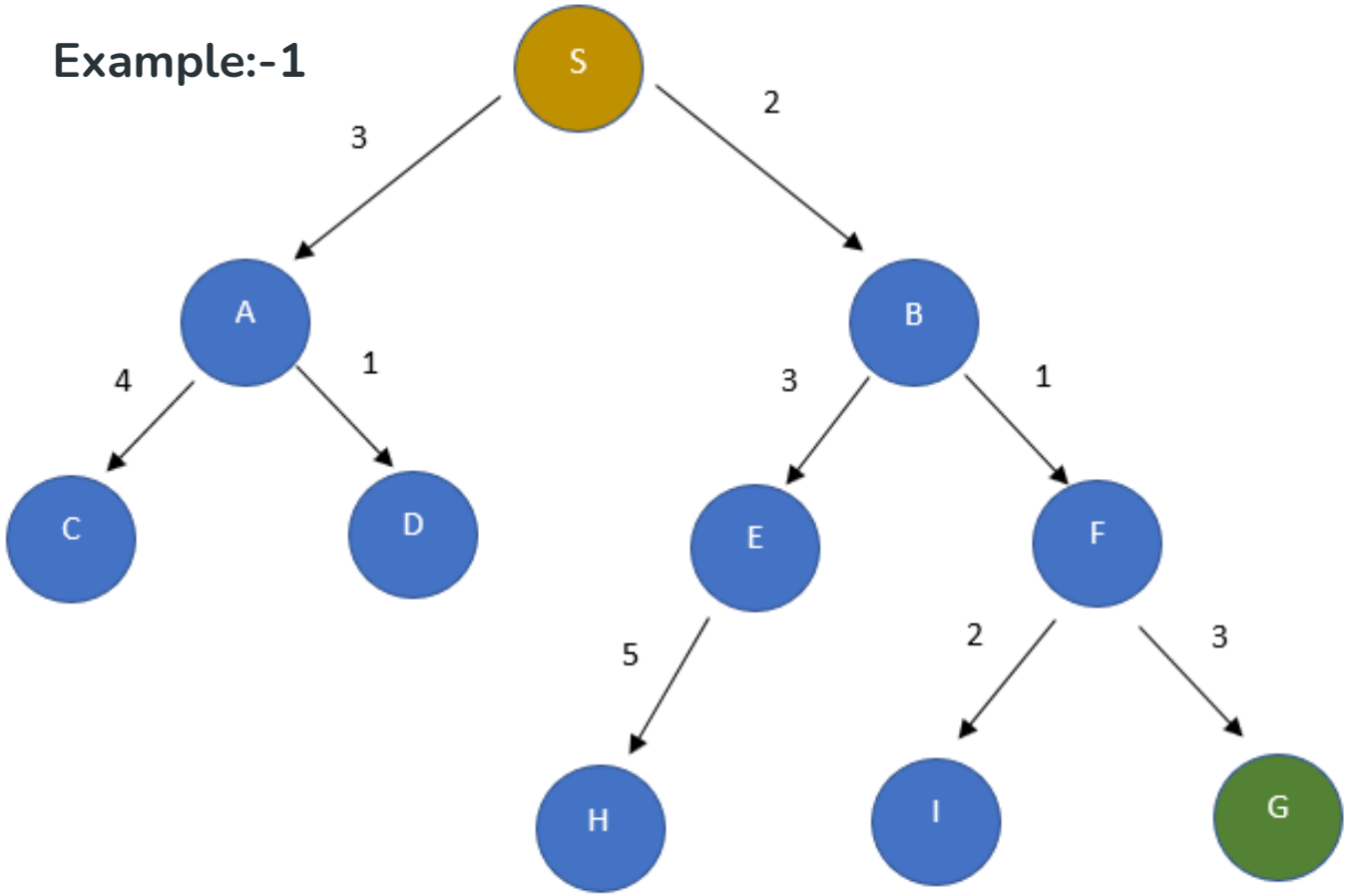
Uninformed Searching	VS	Informed Searching
1. Searching without information.		1. Searching with information.
2. No knowledge, no guide.		2. use knowledge to find steps to solution.
3. Time Consuming.		3. Quick solution.
4. More Complexity (Time, Space)		4. Less Complexity (Time, Space)
5. BFS, DFS, etc.		5. A*, Heuristic DFS, Best First Search

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.
- 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.
- Time Complexity: The worst case time complexity of Greedy best first search is O(bm).
- Space Complexity: The worst case space complexity of Greedy best first search is O(bm). 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.

- 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

Example:-1



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 utilize two lists: OPEN and CLOSED Lists. These lists help in managing and tracking the exploration of nodes during the search process. Expand the nodes of S and put them 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]

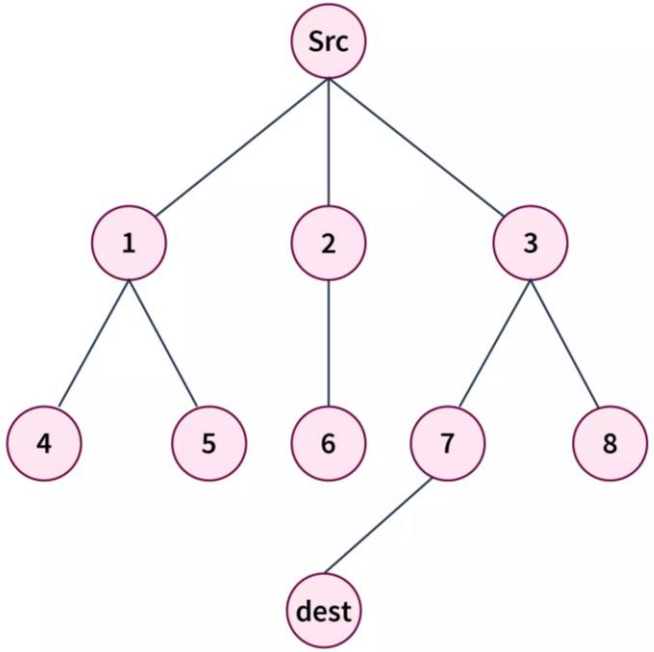
: Open [E, A], Closed [S, B, F]

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

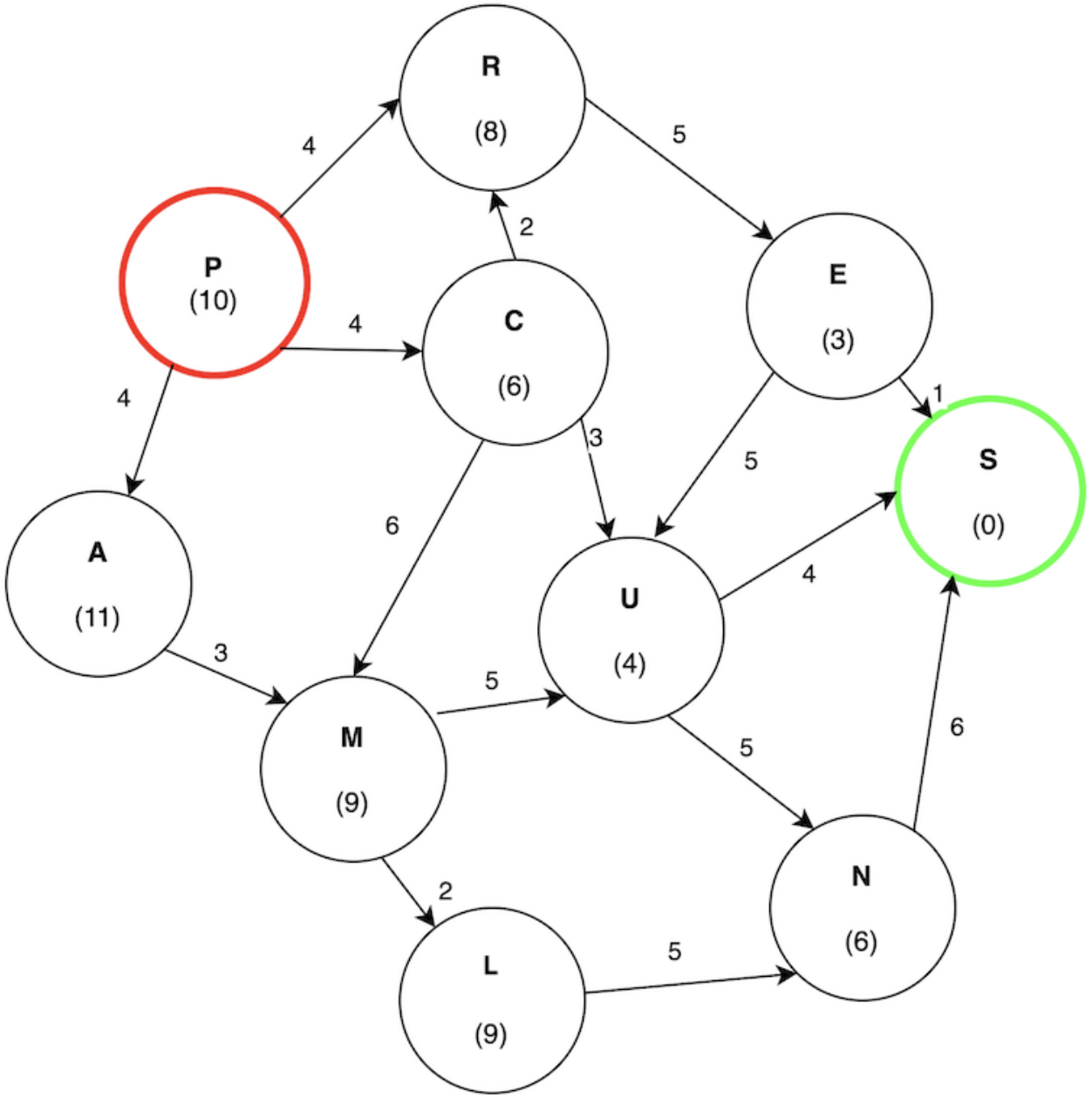
Example:-2



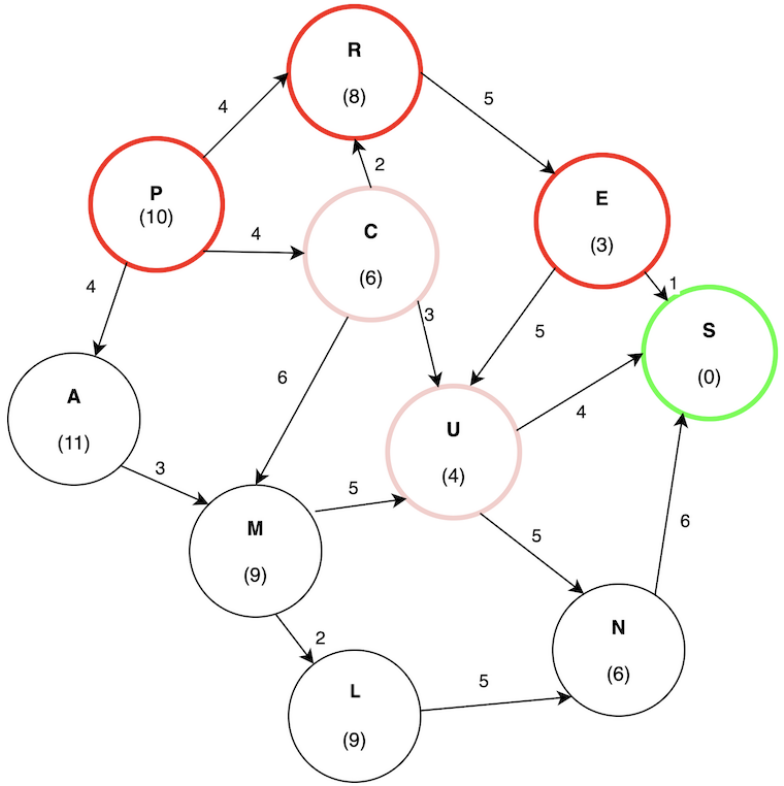
Src	20
1	22
2	21
3	10
4	25
5	24
6	30
7	5
8	12
dest	0

- *step 1* - Initialize two empty lists *i.e.i.e.* openList and closeList.
- *Step 2*- Insert src in the openList, after which we have - openList = [src] closeList = []
- *Step 3* - (Iterating while the open list is not empty) -
    - **Iteration 1 -**
        - After this operation we will have - openList = [1, 2, 3] closeList = [src]
    - **Iteration 2** -
        - After this operation we will have - openList = [1, 2, 7, 8] closeList = [src, 3]
    - **Iteration 3** -
        - Upon exploring the adjacent of node 7 we found that dest is one of them. Hence, we return that search result in success, and print the path src→3→7→dest

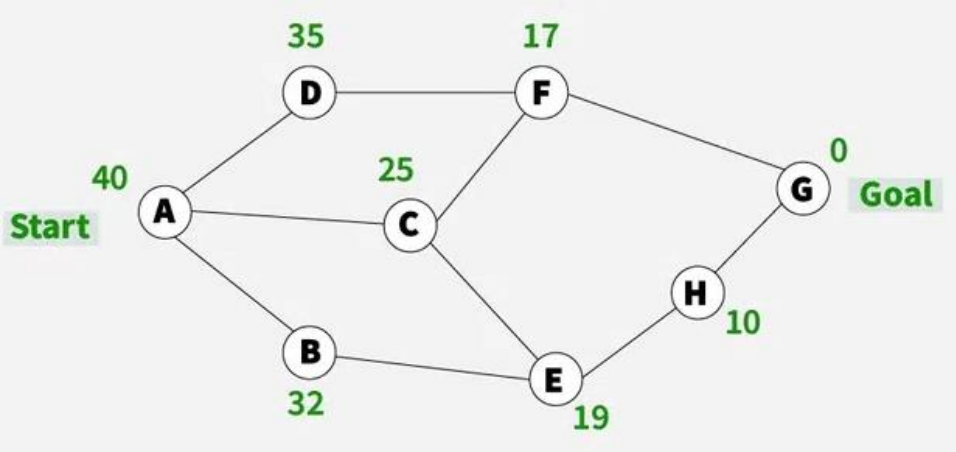
Example:-3



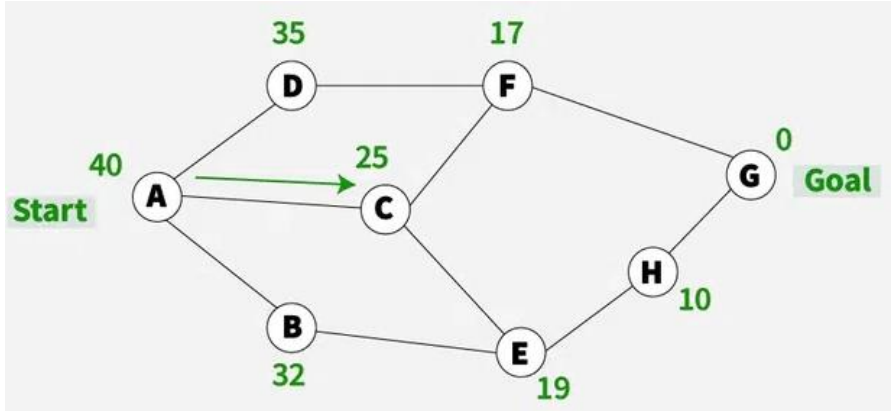
Consider finding the path from P to S in the following graph:
In this example, the cost is measured strictly using the heuristic value. In other words, how close it is to the target.
C has the lowest cost of 6.
U has the lowest cost compared to M and R, so the search will continue by exploring U. Finally, S has a heuristic value of 0 since that is the target node:
The total cost for the path (P -> C -> U -> S) evaluates to 11. The potential problem with a greedy best-first search is revealed by the path (P -> R -> E -> S) having a cost of 10, which is lower than (P -> C -> U -> S). Greedy best-first search ignored this path because it does not consider the edge weights.



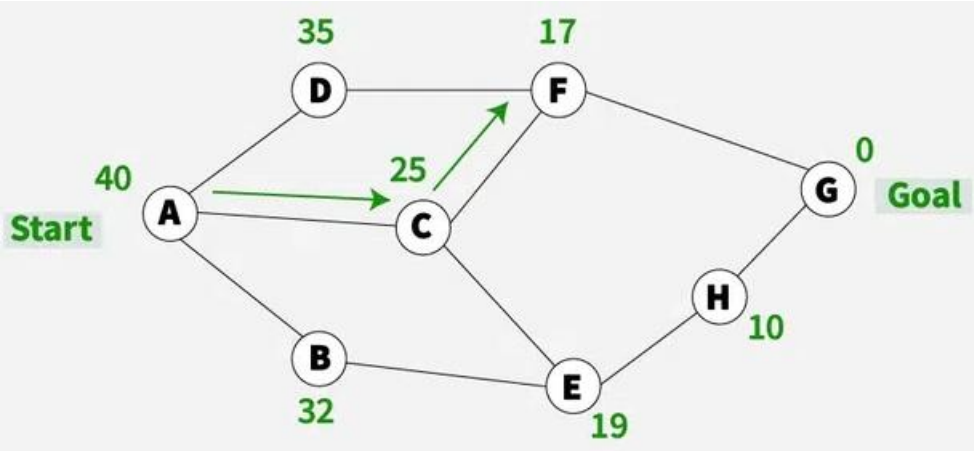
Example-4:- An example of the best-first search algorithm is below graph, suppose we have to find the path from A to G



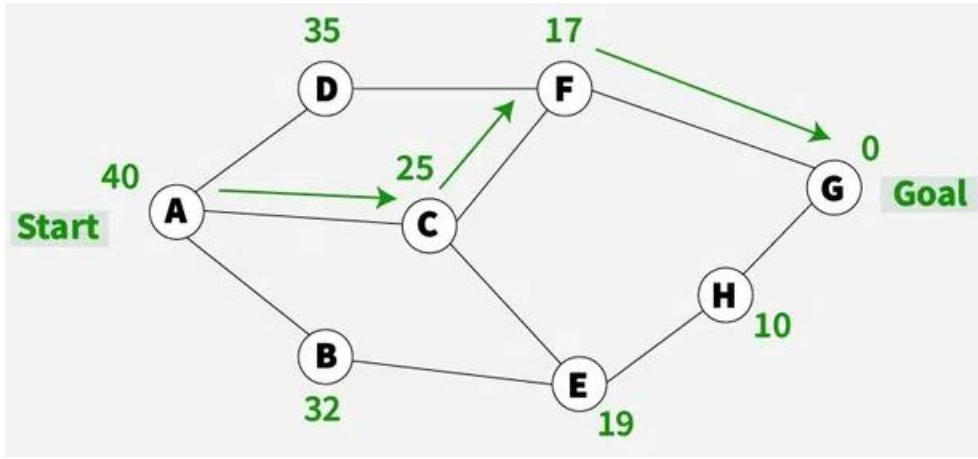
1) We are starting from A , so from A there are direct path to node B(with heuristics value of 32) , from A to C (with heuristics value of 25) and from A to D(with heuristics value of 35) .
2) So as per best first search algorithm choose the path with lowest heuristics value , currently C has lowest value among above node . So we will go from A to C.



3) Now from C we have direct paths as C to F(with heuristics value of 17) and C to E(with heuristics value of 19) , so we will go from C to F.

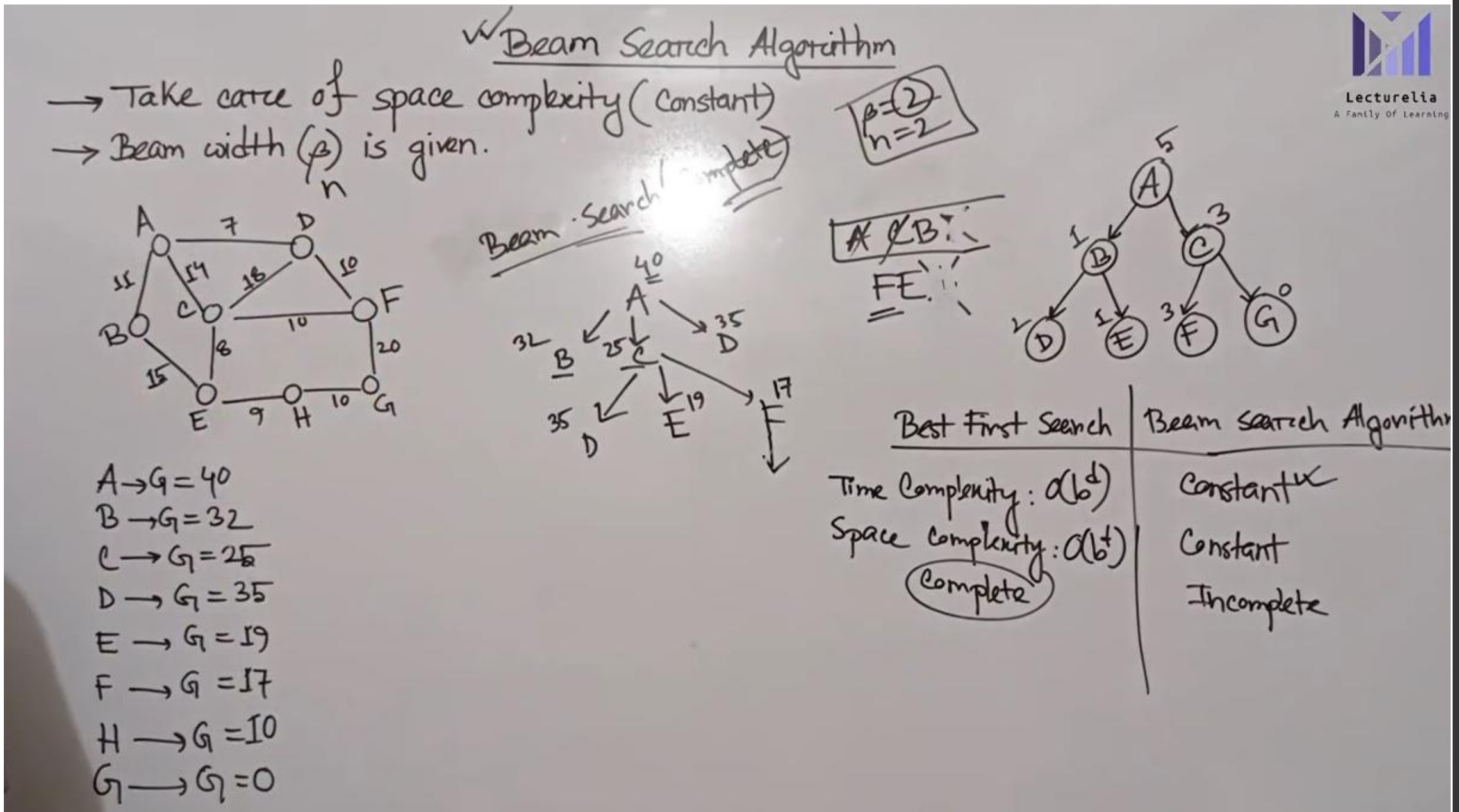


4) Now from F we have direct path to go to the goal node G (with heuristics value of 0) , so we will go from F to G.



*5) So now the goal node G has been reached and the path we will follow is **A->C->F->G** .*

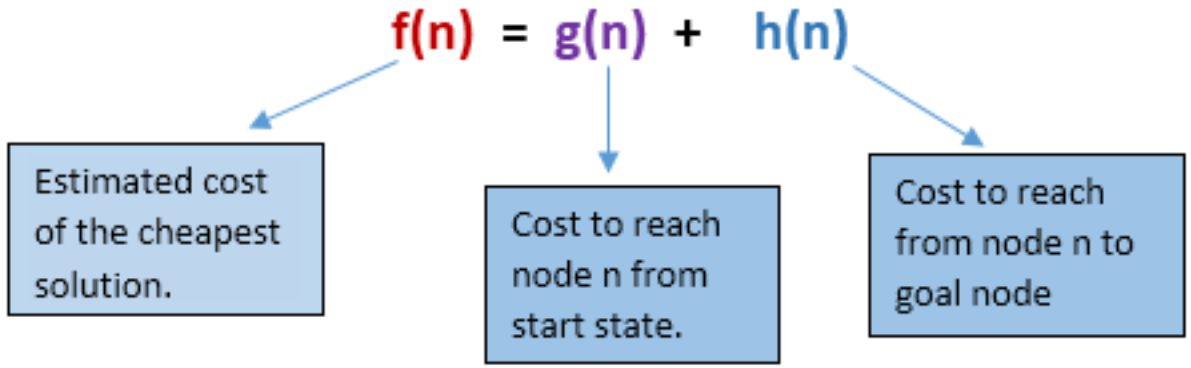
Beam search


Beam Search Algorithm
→ Take care of space complexity (Constant)
→ Beam width (β) is given.
β = 2
n = 2
Beam search
Lecturelia
A Family Of Learning
A 7 D
11 14 16 10
B C F
10
8 20
15
E 9 H 10 G
A→G = 40
B→G = 32
C→G = 25
D→G = 35
E→G = 19
F→G = 17
H→G = 10
G→G = 0
40
A
32 25 35
B C D
35 19 17
D E F
Best First Search
Beam search Algorithm
Time Complexity : O(b^d)
Space Complexity : O(b^d)
Complete
Constant
Constant
Incomplete

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**.



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.

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="">

Algorithm-

The implementation of A* Algorithm involves maintaining two lists- OPEN and CLOSED.

- ✓ OPEN contains those nodes that have been evaluated by the heuristic function but have not been expanded into successors yet.
- ✓ CLOSED contains those nodes that have already been visited.

The algorithm is as follows-

Step-01:- Define a list OPEN.
Initially, OPEN consists solely of a single node, the start node S.
Step-02:- If the list is empty, return failure and exit.
Step-03:- Remove node n with the smallest value of f(n) from OPEN and move it to list CLOSED.
If node n is a goal state, return success and exit.
Step-04: - Expand node n.
Step-05:- If any successor to n is the goal node, return success and the solution by tracing the path from goal node to S. Otherwise, go to Step-06.
Step-06:- For each successor node,
Apply the evaluation function f to the node.
If the node has not been in either list, add it to OPEN.
Step-07:- Go back to Step-02.

Yes, UCS is a special case of A*.

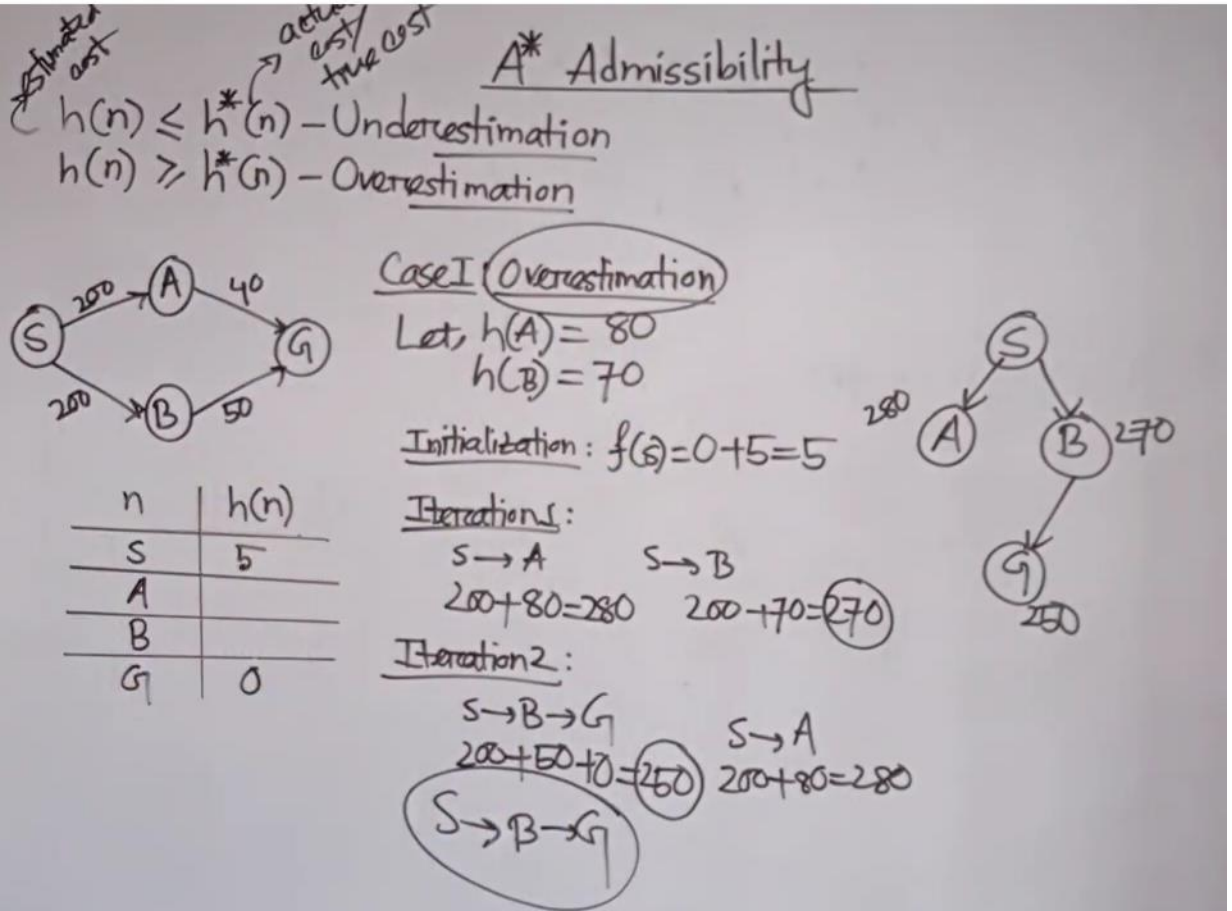
UCS uses the evaluation function $f(n) = g(n)$, where $g(n)$ is the length of the path from the starting node to n, whereas A* uses the evaluation function $f(n) = g(n) + h(n)$, where $g(n)$ means the same thing as in UCS and $h(n)$, called the "heuristic" function, is an estimate of the distance from n to the goal node. In the A* algorithm, $h(n)$ must be admissible.

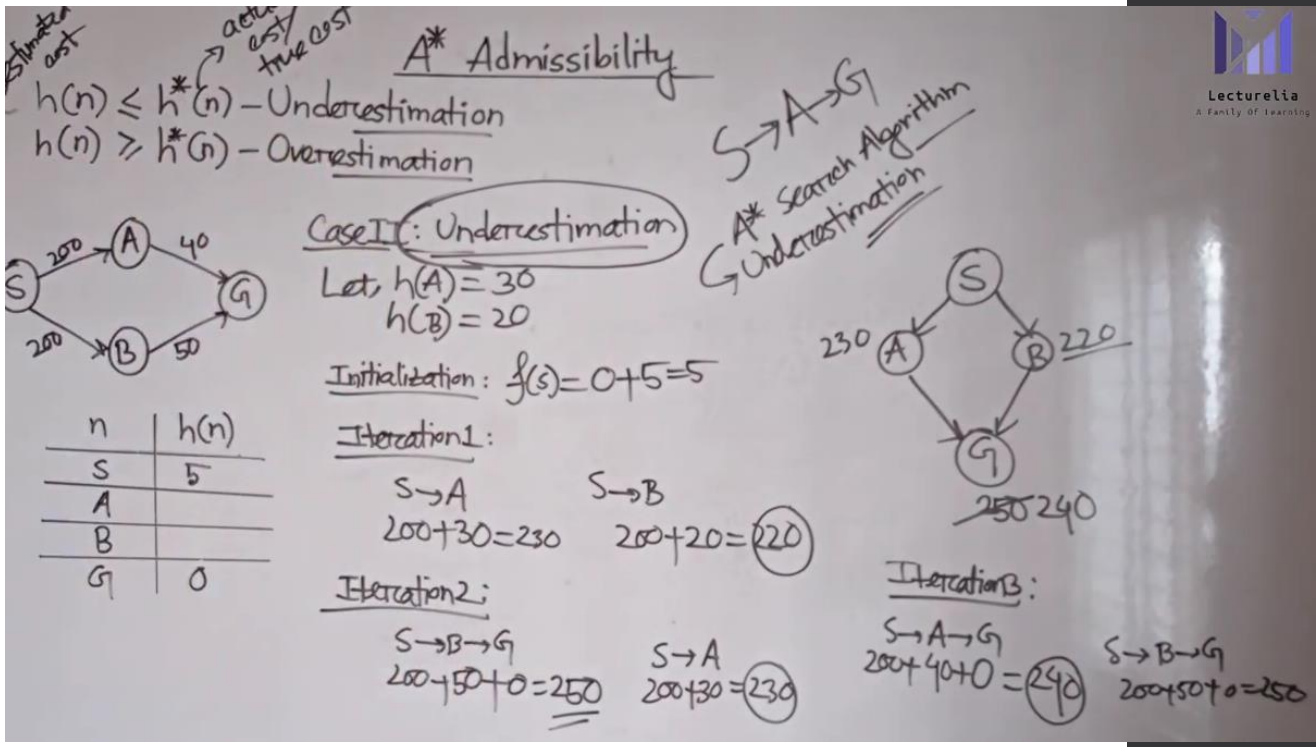
UCS is a special case of A* which corresponds to having $h(n) = 0, \forall n$. A heuristic function h which has $h(n) = 0$, $\forall n$, is clearly admissible, because it always "underestimates" the distance to the goal, which cannot be smaller than 0, unless you have negative edges (but I assume that all edges are non-negative). So, indeed, UCS is a special case of A*, and its heuristic function is even admissible!

To see this with an example, just draw a simple graph, and apply the A* algorithm using $h(n) = 0$, for all n, and then apply UCS to the same graph. You will obtain the same results.

Admissible A*

- The heuristic function h(n) is called admissible if h(n) is never larger than h*(n), namely h(n) is always less or equal to true cheapest cost from n to the goal.
- A* is admissible if it uses an admissible heuristic, and h(goal) = 0.
- If the heuristic function, h always underestimates the true cost (h(n) is smaller than h*(n)), then A* is guaranteed to find an optimal solution.





Problem-01:

Given an initial state of a 8-puzzle problem and final state to be reached-

Find the most cost-effective path to reach the final state from initial state using A* Algorithm.

Consider g(n) = Depth of node and

h(n) = Number of misplaced tiles.

Solution-

2	8	3
1	6	4
7		5

Initial State

1	2	3
8		4
7	6	5

Final State

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

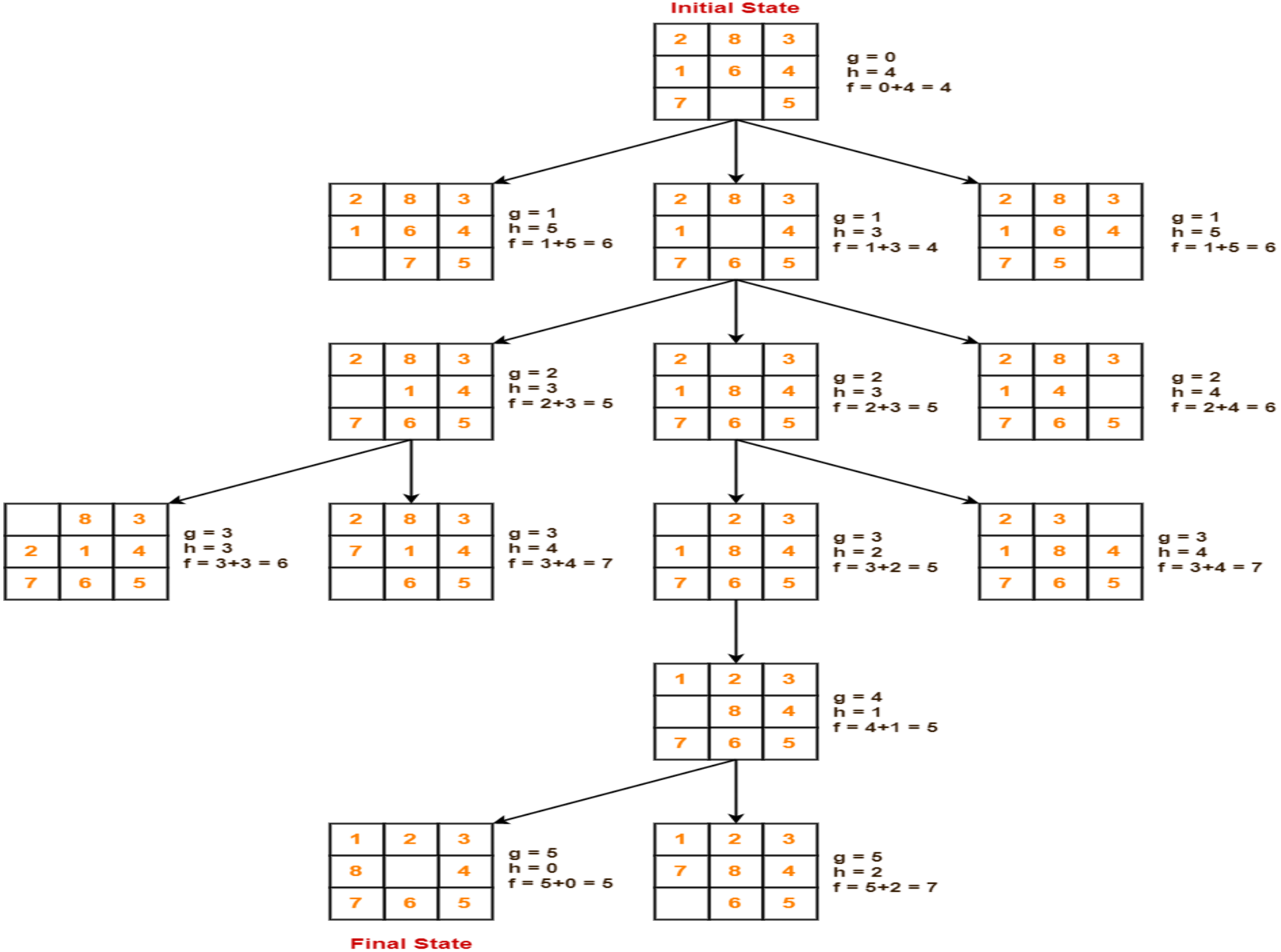
Optimal: A* search algorithm is optimal if it follows below two conditions:

- **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.

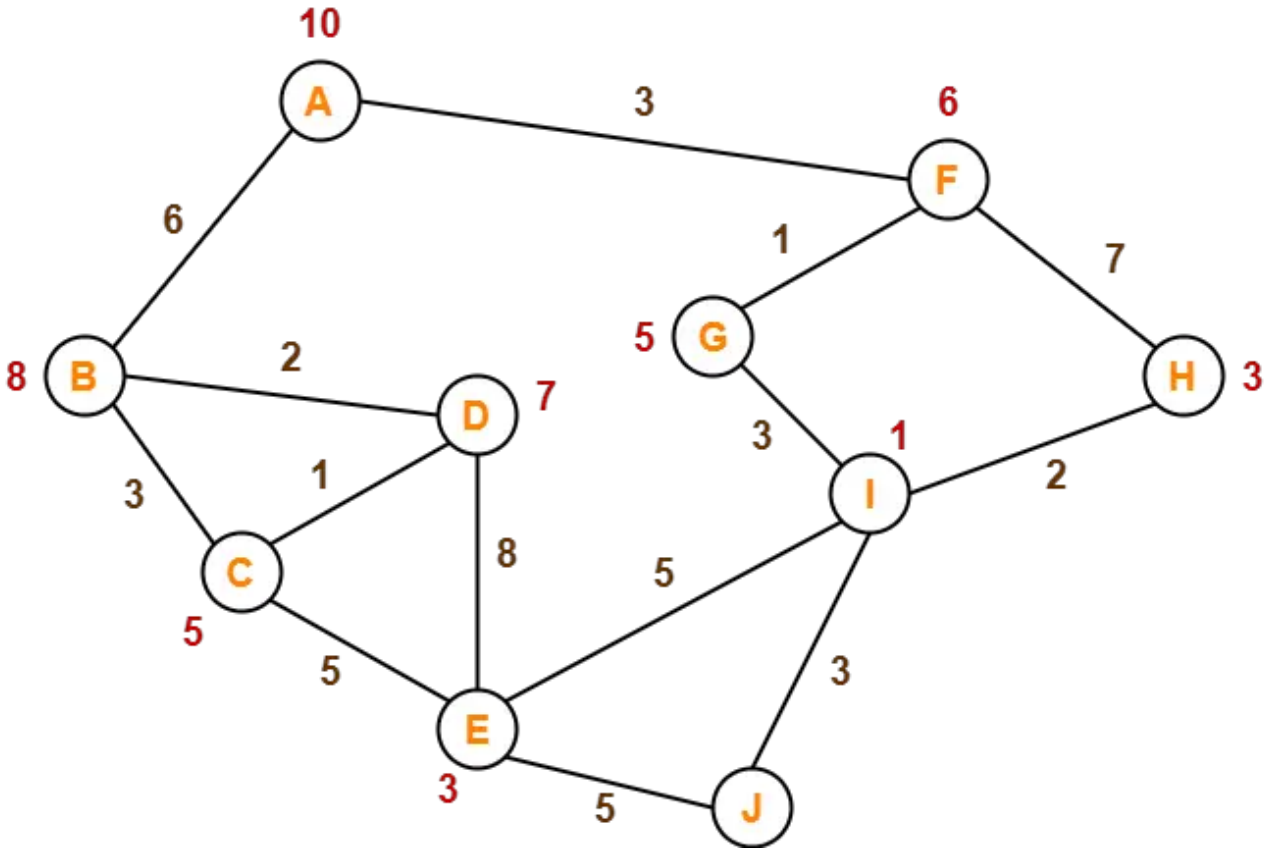
Time Complexity: The time complexity of A* search algorithm depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution d. So the time complexity is O(b^d), where b is the branching factor.

Space Complexity: The space complexity of A* search algorithm is O(b^d)


Initial State
2 8 3
1 6 4
7 5
g = 0
h = 4
f = 0+4 = 4
2 8 3
1 6 4
7 5
g = 1
h = 5
f = 1+5 = 6
2 8 3
1 4
7 6 5
g = 1
h = 3
f = 1+3 = 4
2 8 3
1 6 4
7 5
g = 1
h = 5
f = 1+5 = 6
2 8 3
1 4
7 6 5
g = 2
h = 3
f = 2+3 = 5
2 3
1 8 4
7 6 5
g = 2
h = 3
f = 2+3 = 5
2 8 3
1 4
7 6 5
g = 2
h = 4
f = 2+4 = 6
8 3
2 1 4
7 6 5
g = 3
h = 3
f = 3+3 = 6
2 8 3
7 1 4
6 5
g = 3
h = 4
f = 3+4 = 7
2 3
1 8 4
7 6 5
g = 3
h = 2
f = 3+2 = 5
2 3
1 8 4
7 6 5
g = 3
h = 4
f = 3+4 = 7
1 2 3
8 4
7 6 5
g = 4
h = 1
f = 4+1 = 5
1 2 3
8 4
7 6 5
g = 5
h = 0
f = 5+0 = 5
Final State
1 2 3
7 8 4
6 5
g = 5
h = 2
f = 5+2 = 7

Problem-02:

- The numbers written on edges represent the distance between the nodes.
- The numbers written on nodes represent the heuristic value.
- Find the most cost-effective path to reach from start state A to final state J using A* Algorithm.

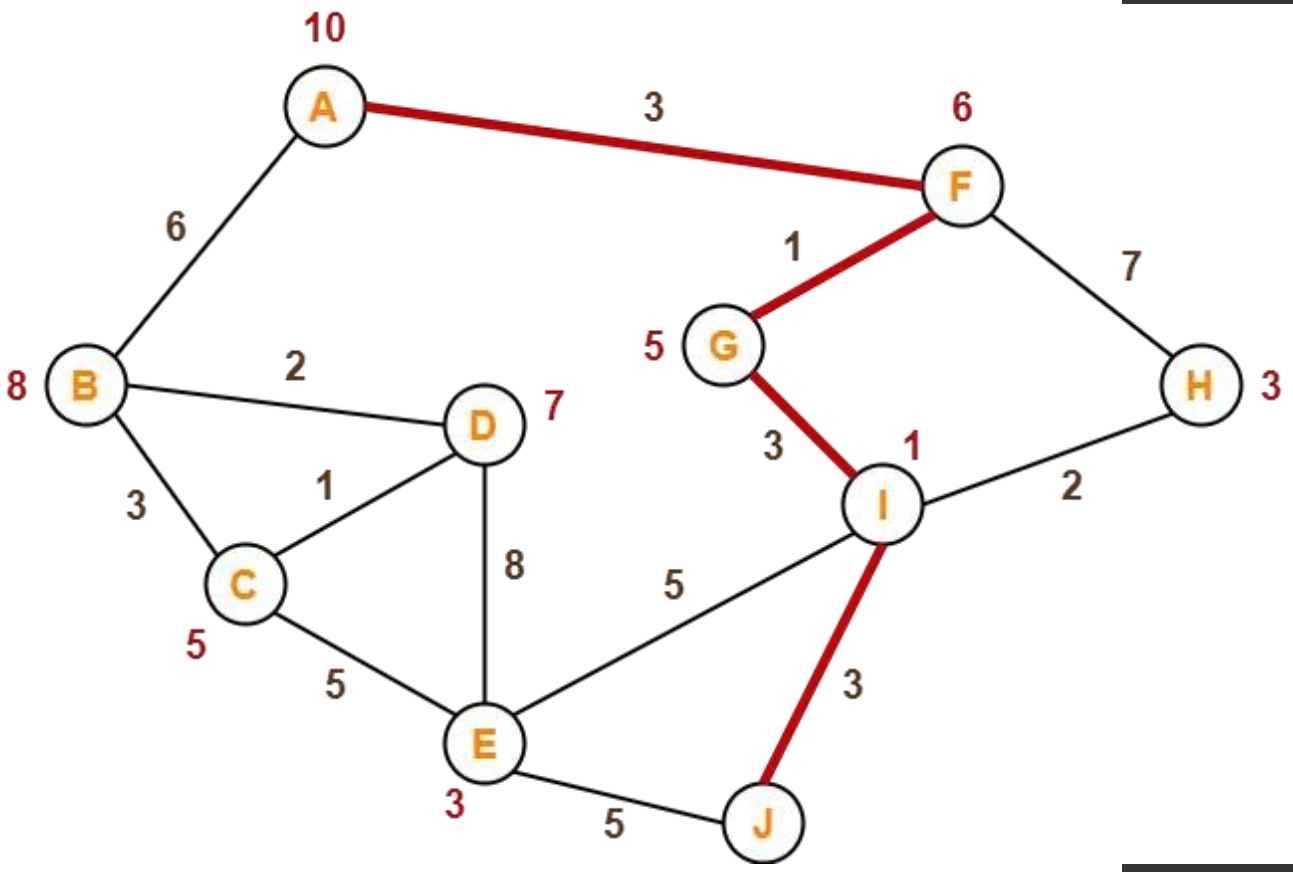


Step-01:- We start with node A.
Node B and Node F can be reached from node A.
A* Algorithm calculates f(B) and f(F).
f(B) = 6 + 8 = 14
f(F) = 3 + 6 = 9
Since f(F) < f(B), so it decides to go to node F.
Path- A → F

Step-02:- Node G and Node H can be reached from node F.
A* Algorithm calculates f(G) and f(H).
f(G) = (3+1) + 5 = 9
f(H) = (3+7) + 3 = 13
Since f(G) < f(H), so it decides to go to node G.
Path- A → F → G

Step-03:- Node I can be reached from node G.
A* Algorithm calculates f(I).
f(I) = (3+1+3) + 1 = 8
It decides to go to node I.
Path- A → F → G → I

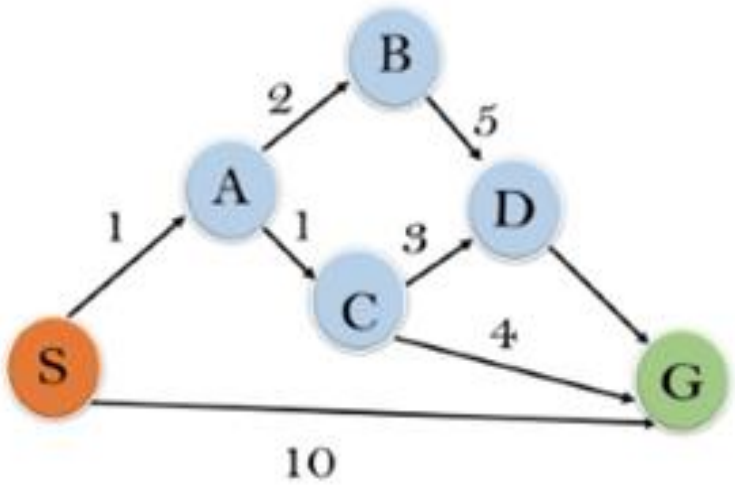
Step-04:- Node E, Node H and Node J can be reached from node I.
A* Algorithm calculates f(E), f(H) and f(J).
f(E) = (3+1+3+5) + 3 = 15
f(H) = (3+1+3+2) + 3 = 12
f(J) = (3+1+3+3) + 0 = 10
Since f(J) is least, so it decides to go to node J.
Path- A → F → G → I → J
This is the required shortest path from node A to node J.



Important Note-
It is important to note that-
A Algorithm is one of the best path finding algorithms.*
But it does not produce the shortest path always.
This is because it heavily depends on heuristics.

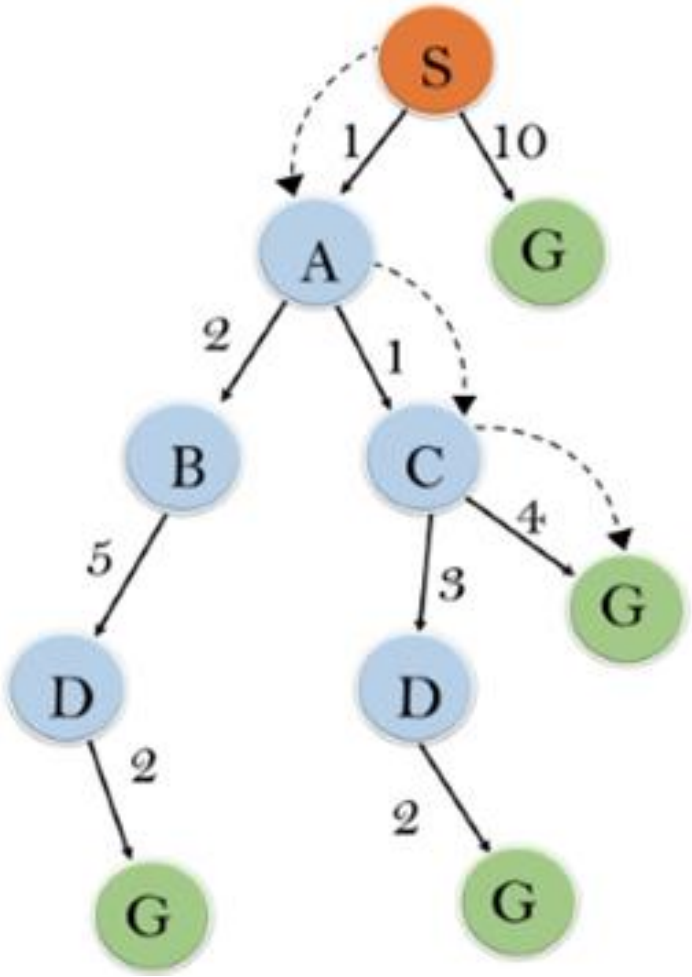
Problem-3:- In this example, we will traverse the given graph using the A* algorithm. The heuristic value of all states is given in the below table so we will calculate the f(n) of each state using the formula f(n)= g(n) + h(n), where g(n) is the cost to reach any node from start state.

Here we will use OPEN and CLOSED list.



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

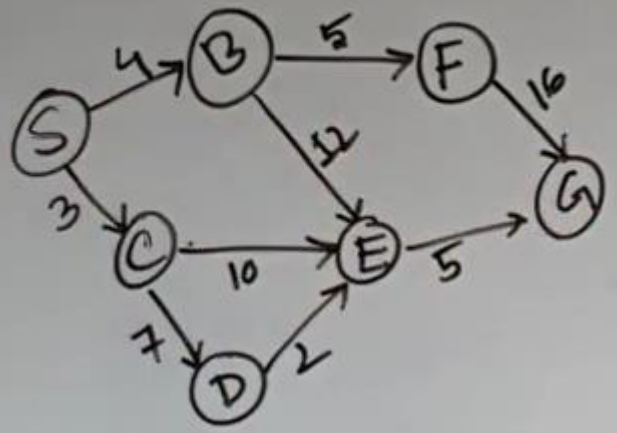
Solution:



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.

Example-4:

A* Search Algorithm



n	h(n)
S	14
B	12
C	11
D	6
E	4
F	11
G	0

Initialization: $f(S) = 0+14 = 14$

Iteration 1: S→B: 4+12=16 ; S→C: 3+11=14

Iteration 2: S→C→D: 3+7+6=16 ; S→C→E: 3+10+4=17

S→B: 4+12=16

Iteration 3:

S→C→D→E: 3+7+2+4=16

S→B→E: 4+12+4=20

S→B→F: 4+5+11=20

S→C→E: 3+10+4=17

Iteration 4:

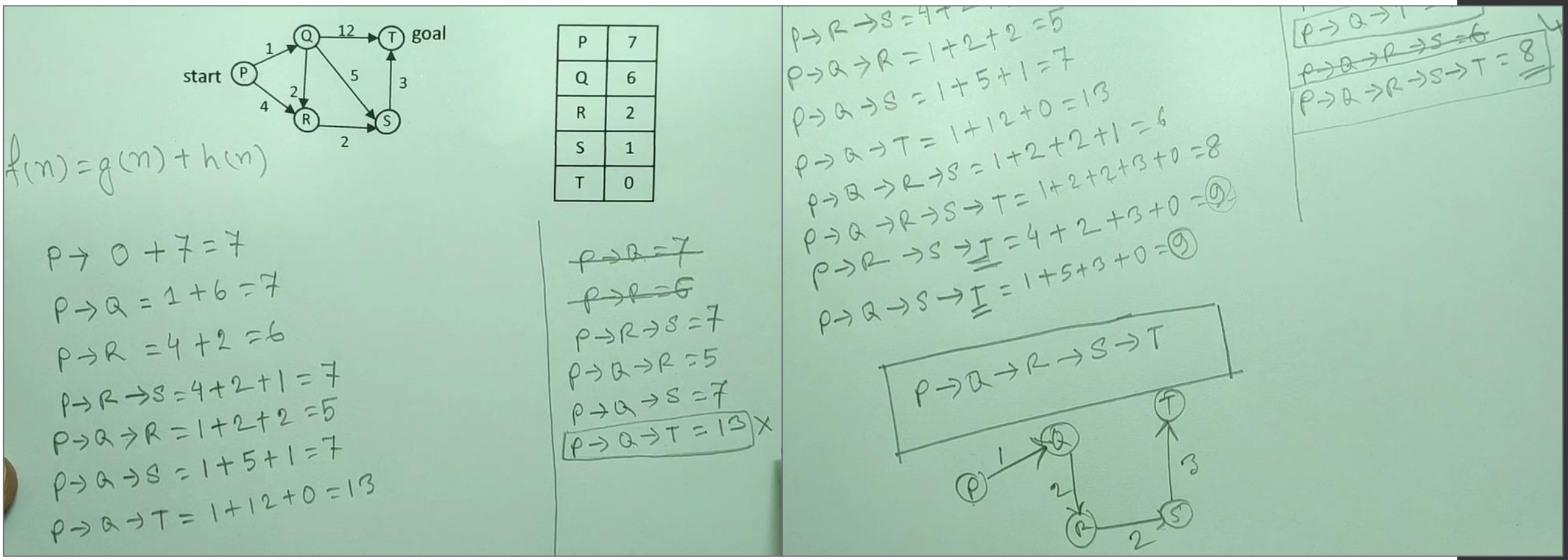
S→C→D→E→G: 3+7+2+5+0=17

S→B→E→G: 4+12+5+0=21

S→B→F→G: 4+5+16+0=25

S→C→E→G: 3+10+5+0 = 18

Example-5:


start
P
Q
R
S
T
goal
1
12
5
3
2
4
2
f(n) = g(n) + h(n)
P → 0 + 7 = 7
P → Q = 1 + 6 = 7
P → R = 4 + 2 = 6
P → R → S = 4 + 2 + 1 = 7
P → Q → R = 1 + 2 + 2 = 5
P → Q → S = 1 + 5 + 1 = 7
P → Q → T = 1 + 12 + 0 = 13
P 7
Q 6
R 2
S 1
T 0
P → R = 7
P → R = 6
P → R → S = 7
P → Q → R = 5
P → Q → S = 7
P → Q → T = 13 X
P → R → S = 4 + 2 = 5
P → Q → R = 1 + 2 + 2 = 5
P → Q → S = 1 + 5 + 1 = 7
P → Q → T = 1 + 12 + 0 = 13
P → Q → R → S = 1 + 2 + 2 + 1 = 6
P → Q → R → S → T = 1 + 2 + 2 + 3 + 0 = 8
P → R → S → T = 4 + 2 + 3 + 0 = 9
P → Q → S → T = 1 + 5 + 3 + 0 = 9
P → Q → R → S → T = 8
P → Q → R → S → T

Parameters	Best First Search	A* Search
Past knowledge	No prior knowledge.	Past knowledge involved
Completeness	Not complete	Complete
Optimal	May not optimal	Always optimal
Evaluation Function	f(n)=h(n)Where h(n) is heuristic function	f(n)=h(n)+g(n)Where h(n) is heuristic function and g(n) is past knowledge acquired
Time Complexity	O(bm,,,) where b is branching and m is search tree's maximum depth	O(bm,,,) where b is branching and m is search tree's maximum depth
Space Complexity	Polynomial	O(bm,,) where b is branching and m is search tree's maximum depth
Nodes	When searching, all the fridges or border nodes are kept in memory	All nodes are present in memory while searching
Memory	Need less memory	Need more memory

Hill Climbing

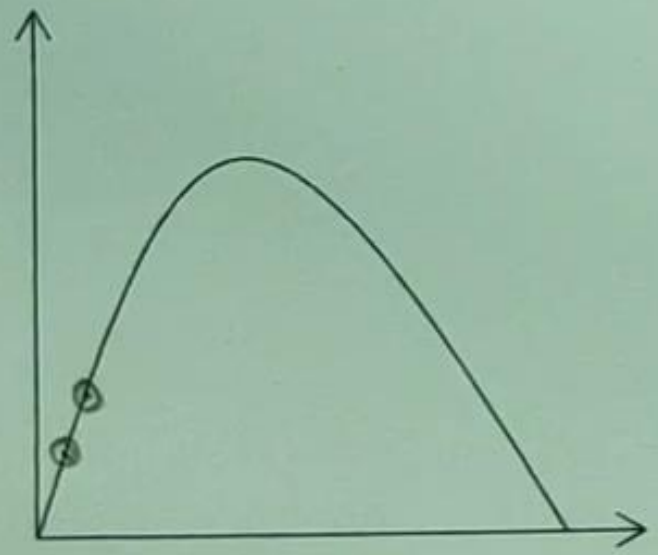
- *Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem. It terminates when it reaches a peak value where no neighbor has a higher value.*
- Hill climbing algorithm 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.

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.

Hill Climbing Search & it's Drawbacks

Hill Climbing Search



1 . **local variables:** *current*, **a node.**
neighbor, **a node.**

2. *current* ← MAKE-NODE(INITIAL-STATE)

3. Evaluate the initial state and if it is goal state then return and stop.

4. **loop**
until a solution is found or a new node is found.

5. Select and apply new node

6. Evaluate the new state and if it is goal state then return and stop.

7. If it is better then the current state then make it new current state.

8. If it is not better then the current state then go to step 4 .

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:

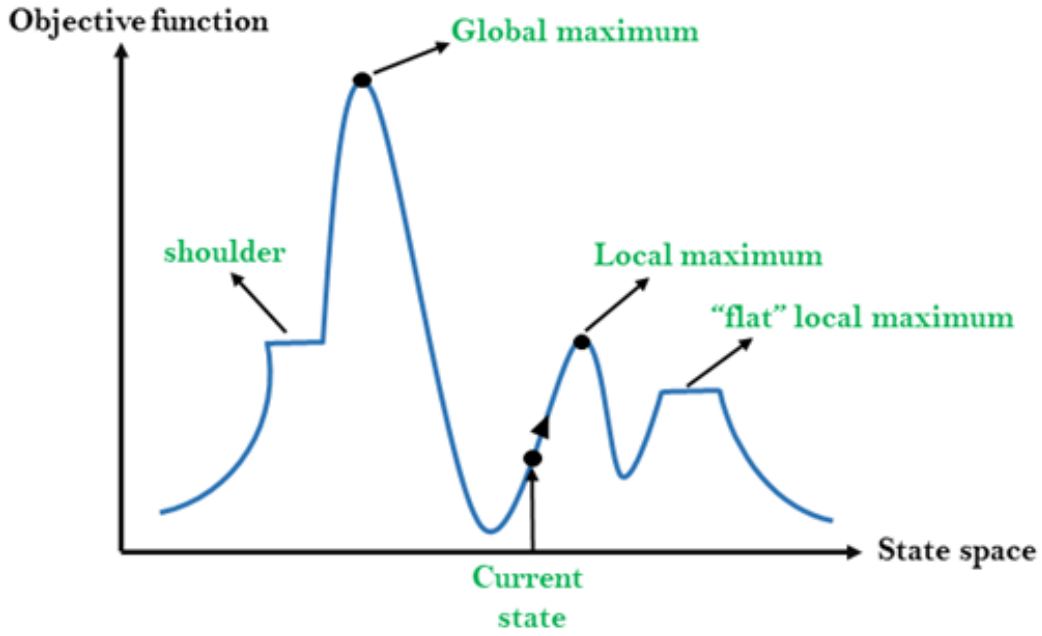
Local Maximum: Local maximum is a state which is better than its neighbour states, but there is also another state which is higher than it.

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 neighbour states of current states have the same value.

Shoulder: It is a plateau region which has an uphill edge.



X-axis: denotes the state space i.e. states or configuration our algorithm may reach.
Y-axis: denotes the values of objective function corresponding to a particular state.

Types of 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:

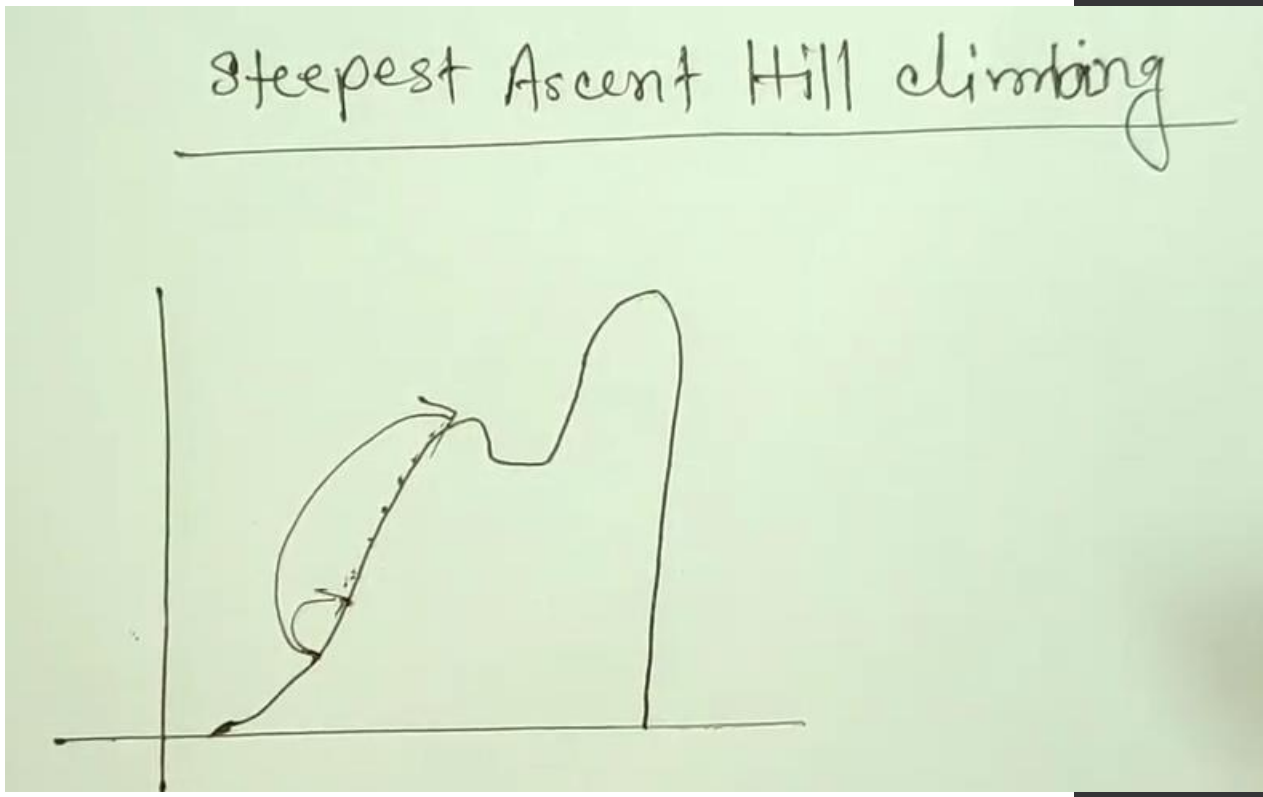
- **Step 1:** Evaluate the initial state, if it is goal state then return success and Stop.
- **Step 2:** Loop Until a solution is found or there is no new operator left to apply.
- **Step 3:** Select and apply an operator to the current state.
- **Step 4:** Check new state:
    - If it is goal state, then return success and quit.
    - Else if it is better than the current state then assign new state as a current state.
    - 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.
    - Let SUCC be a state such that any successor of the current state will be better than it.
    - For each operator that applies to the current state:
        - Apply the new operator and generate a new state.
        - Evaluate the new state.
        - If it is goal state, then return it and quit, else compare it to the SUCC.
        - If it is better than SUCC, then set new state as SUCC.
        - If the SUCC is better than the current state, then set current state to SUCC.
- **Step 5:** Exit.



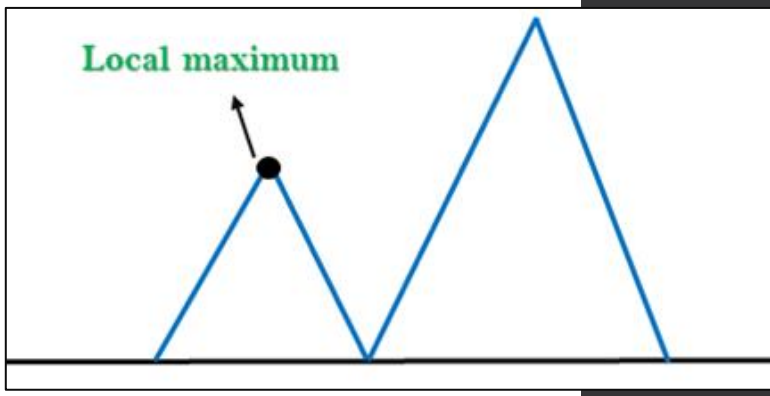
3. Stochastic hill climbing:

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:

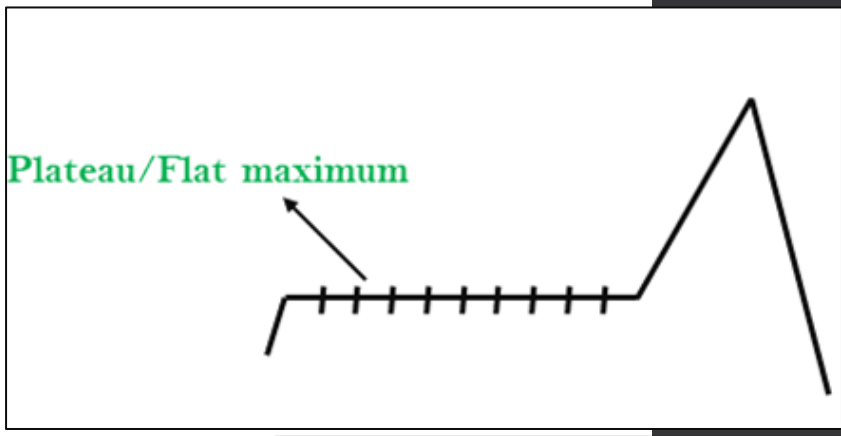
1. Local Maximum: A local maximum is a peak state in the landscape which is better than each of its neighboring states, but there is another state also present which is higher than the local maximum.

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.



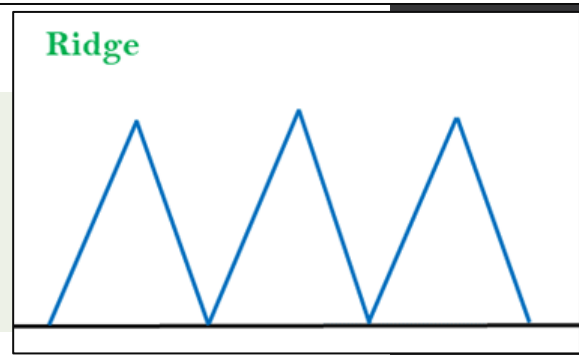
2. Plateau: A plateau is the flat area of the search space in which all the neighbor states of the current state contains the same value, because of this algorithm does not find any best direction to move. A hill-climbing search might be lost in the plateau area.

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.

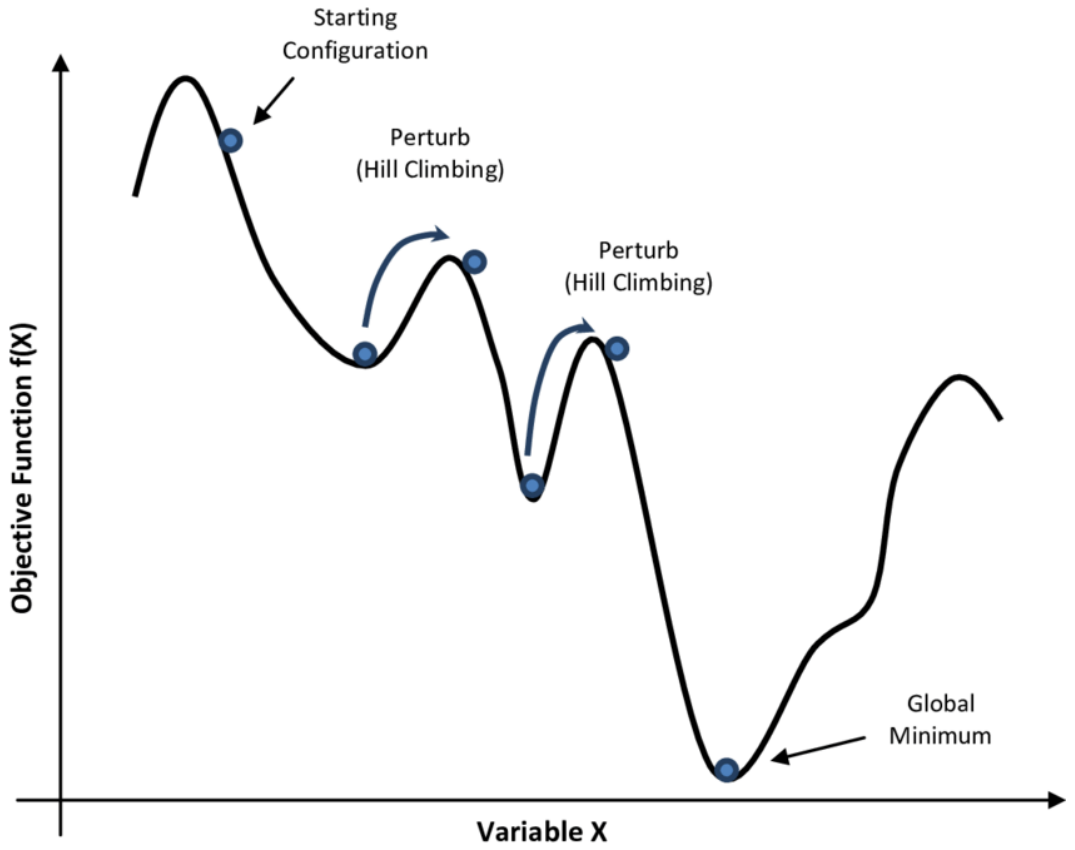
Solution: With the use of bidirectional search, or by moving in different directions, we can improve this problem.



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.



Constraint Satisfaction Problem in AI

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.*

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, 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 Problem (CSP)

A **Constraint Satisfaction Problem** in artificial intelligence involves a set of variables, each of which has a domain of possible values, and a set of constraints that define the allowable combinations of values for the variables. The goal is to find a value for each variable such that all the constraints are satisfied.
The goal of a CSP is to find an assignment of values to the variables that satisfies all the constraints. This assignment is called a solution to the CSP.

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.

➢ Types of Constraints in CSP

1. Unary Constraints:- A unary constraint is a constraint on a single variable. For example, Variable A not equal to “Red”.

2. 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.

3. 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.

➢ Real-world Constraint Satisfaction Problems (CSP):

- **Scheduling:** The constraints in this domain specify the availability and capacity of each resource, whereas the variables indicate the time slots or resources.
- **Vehicle routing:** In this domain, the constraints specify each vehicle’s capacity, delivery locations, and time windows, while the variables indicate the routes taken by the vehicles.
- **Assignment:** In this field, the variables stand in for the tasks, while the constraints specify the knowledge, capacity, and workload of each person or machine.
- **Sudoku:** The well-known puzzle game Sudoku can be modelled as a CSP problem, where the variables stand in for the grid’s cells and the constraints specify the game’s rules.
- **Constraint-based image segmentation:** The segmentation of an image into areas with various qualities (such as color, texture, or shape) can be treated as a CSP issue in computer vision.

➢ Constraint Satisfaction Problems (CSP) benefits:

1. conventional representation patterns
2. generic successor and goal functions
3. Standard heuristics (no domain-specific expertise).

Important Question From This Slide:

1. Explain 8-puzzle Problem Without Heuristic
2. UCS Example-3
3. Iterative Deepening DFS Ex-1
4. Best-first Search Example:1 - 4
5. A* Search: Example 3,4,5***
6. Admissibility Of A* Search***
7. Different Regions In The State Space For Hill Climbing
8. Explain:- Steepest Hill Climbing
9. Problems In Hill Climbing Algorithm***
10. Simulated Annealing(definition + Fig)
11. What Is Constraint Satisfaction Problem (CSP)? How To Represent It?