ARTIFICIAL INTELLIGENCE
UNIT-02

Search Algorithms in Artificial Intelligence

Problem-solving agents:

- In Artificial Intelligence, Search techniques are universal problem-solving methods. **Rational agents** or **Problem-solving agents** in AI mostly use these search strategies or algorithms to solve a specific problem and provide the best result.
- Problem-solving agents are the goal-based agents and use atomic representation. In this topic, we will learn various problem-solving search algorithms.

Search Algorithm Terminologies

- **Search:** Searching is a step-by-step procedure to solve a search-problem in a given search space. A search problem can have three main factors:
  - **Search Space:** Search space represents a set of possible solutions, which a system may have.
  - **Start State:** It is a state from where the agent begins **the search**.
  - **Goal test:** It is a function that observes the current state and returns whether the goal state is achieved or not.
- **Search tree:** A tree representation of a search problem is called a Search tree. The root of the search tree is the root node which corresponds to the initial state.

Search Algorithm Terminologies

- **Actions:** It gives the description of all the available actions to the agent.
- **Transition model:** A description of what each action do, can be represented as a transition model.
- **Path Cost:** It is a function which assigns a numeric cost to each path.
- **Solution:** It is an action sequence which leads from the start node to the goal node.
- **Optimal Solution:** If a solution has the lowest cost among all solutions.

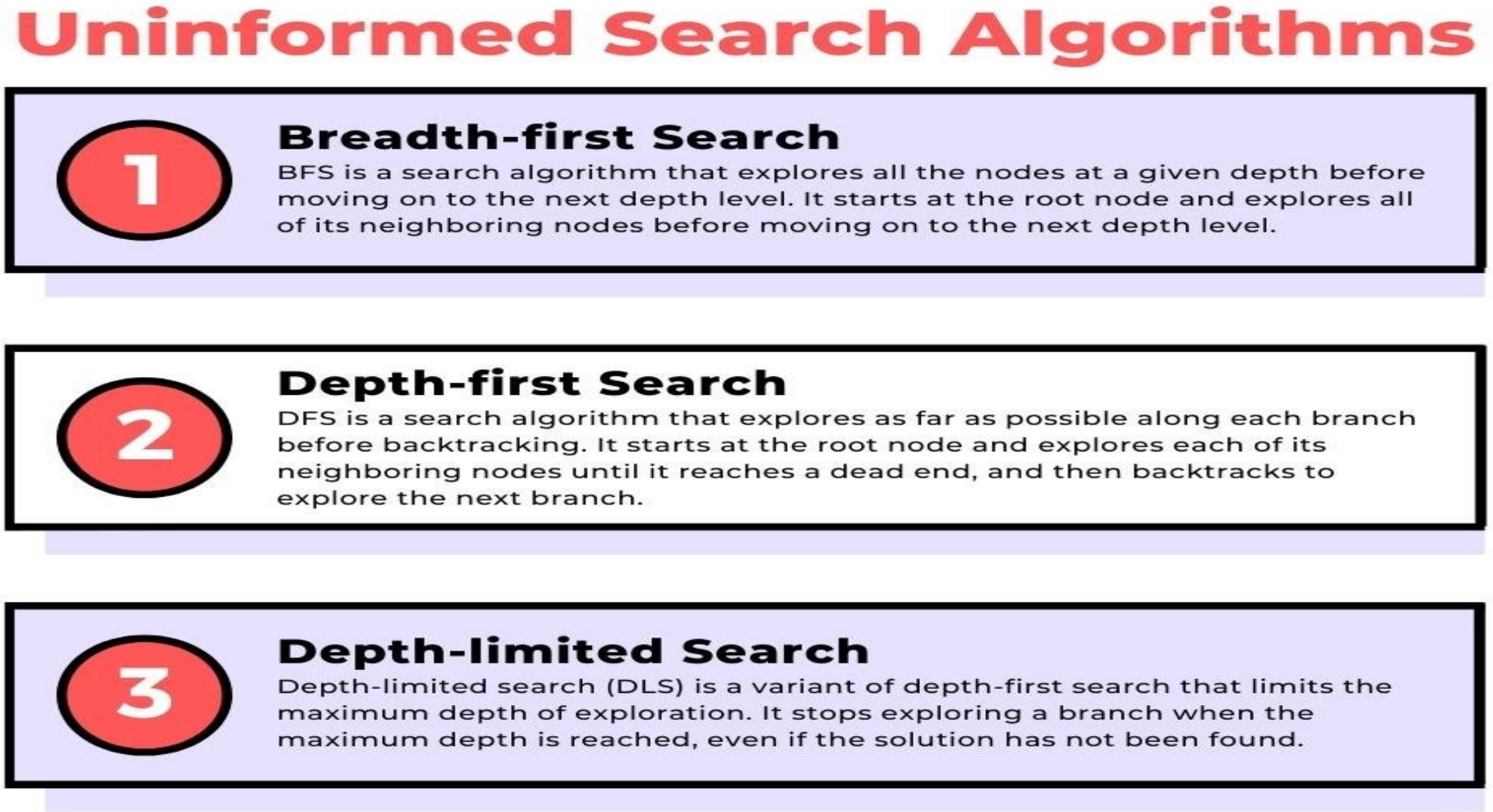
Properties of Search Algorithms

- **Completeness:** A search algorithm is said to be complete if it guarantees to return a solution if at least any solution exists for any random input.
- **Optimality:** If a solution found for an algorithm is guaranteed to be the best solution (lowest path cost) among all other solutions, then such a solution for is said to be an optimal solution.
- **Time Complexity:** Time complexity is a measure of time for an algorithm to complete its task.
- **Space Complexity:** It is the maximum storage space required at any point during the search, as the complexity of the problem.

Application of Search Algorithms

- **Pathfinding:** Pathfinding problems involve finding the shortest path between two points in a given graph or network. BFS or A* search can be used to explore a graph and find the optimal path.
- **Optimization:** In optimization problems, the goal is to find the minimum or maximum value of a function, subject to some constraints. Search algorithms such as hill climbing or simulated annealing are often used in optimization cases.
- **Game Playing:** In game playing, search algorithms are used to evaluate all possible moves and choose the one that is most likely to lead to a win, or the best possible outcome. This is done by constructing a search tree where each node represents a game state and the edges represent the moves that can be taken to reach the associated new game state

Types of search algorithms



Uninformed Search Algorithms

Iterative Deepening Depth-first Search

It is a variant of depth-first search that gradually increases the maximum depth of exploration until the solution is found. It starts with a maximum depth of 1 and increases the depth by 1 in each iteration until the solution is found.

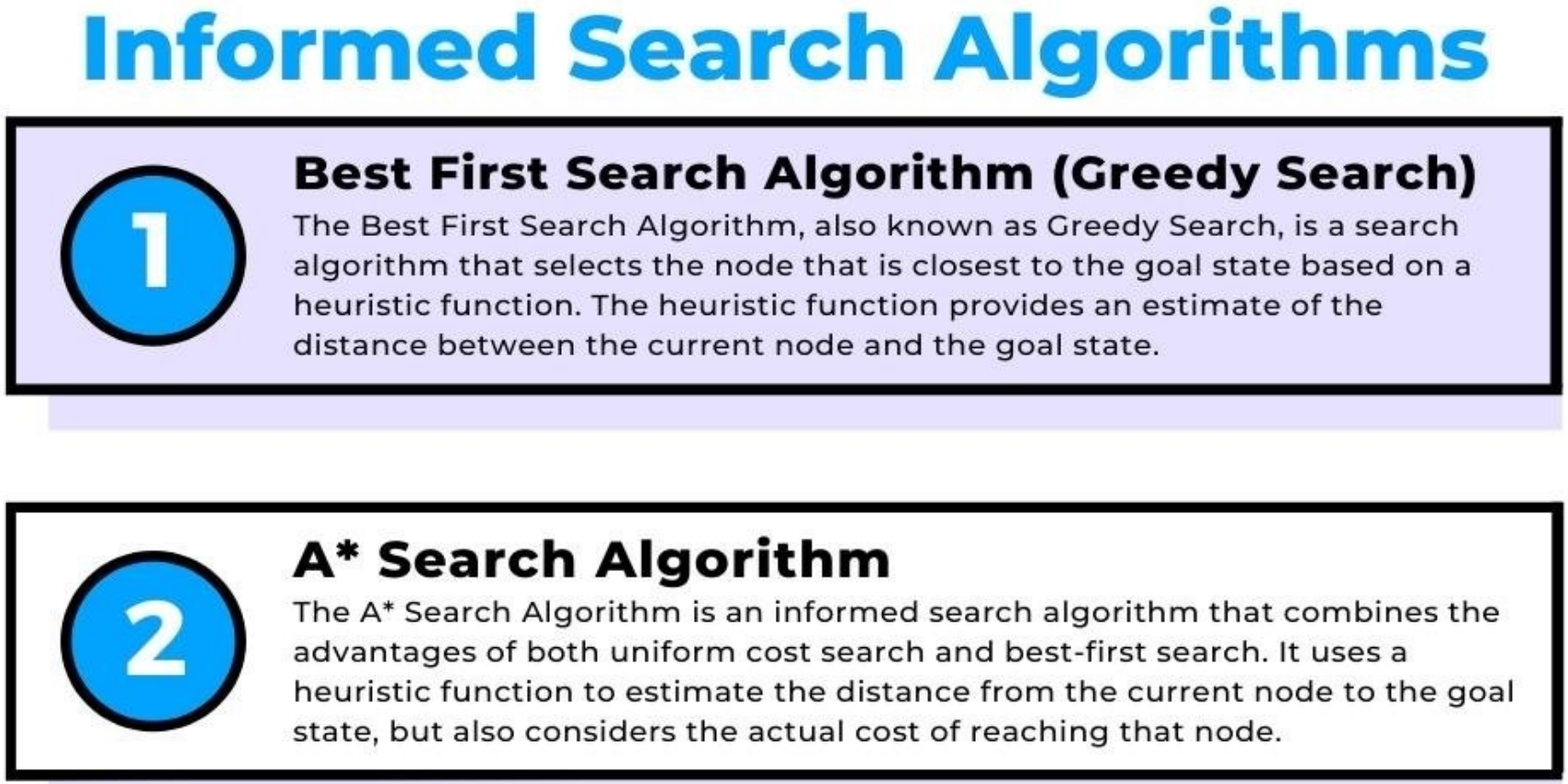
Uniform Cost Search

Uniform cost search (UCS) is a search algorithm that explores the nodes with the lowest cost first. It starts at the root node and explores each neighboring node in order of increasing cost.

Bidirectional Search

Bidirectional search is a search algorithm that starts from both the starting and ending nodes and searches towards the middle. It explores all the neighboring nodes in both directions until they meet at a common node.

Types of search algorithms



3. **AO* algorithm:** Best-first search is what the AO* algorithm does. The AO* method divides any given difficult problem into a smaller group of problems that are then resolved using the AND-OR graph concept.

DIFFERENCE BETWEEN UNINFORMED & INFORMED SEARCH STRATEGIES

Uninformed Search	Informed Search
It is a search strategy with no additional information. It only contains the current state information.	It is a search strategy which carries some additional information with the current state information.
It is less efficient to use uninformed search technique.	It is more efficient search technique.
It may take more time to reach the goal node.	It mostly reaches the goal state in limited time.
May or may not give an optimal solution.	Mostly provides an optimal solution.

DIFFERENCE BETWEEN UNINFORMED & INFORMED SEARCH STRATEGIES

Uninformed Search	Informed Search
It is also known as Blind Search because it searches the goal blindly without having the prior knowledge.	It is also known as Heuristic Search as it searches the goal with the help of some prior knowledge.
An uninformed search requires more computation.	An informed search require less computation.
BFS, DFS, Uniform cost search are types of uninformed search.	Best first search, A* search are types of informed search.

Breadth-first Search(BFS)

- Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches breadth wise in a tree or graph, so it is called breadth-first search.
- BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level.
- The breadth-first search algorithm is an example of a general-graph search algorithm.
- Breadth-first search implemented using FIFO queue data structure.

Advantages of BFS

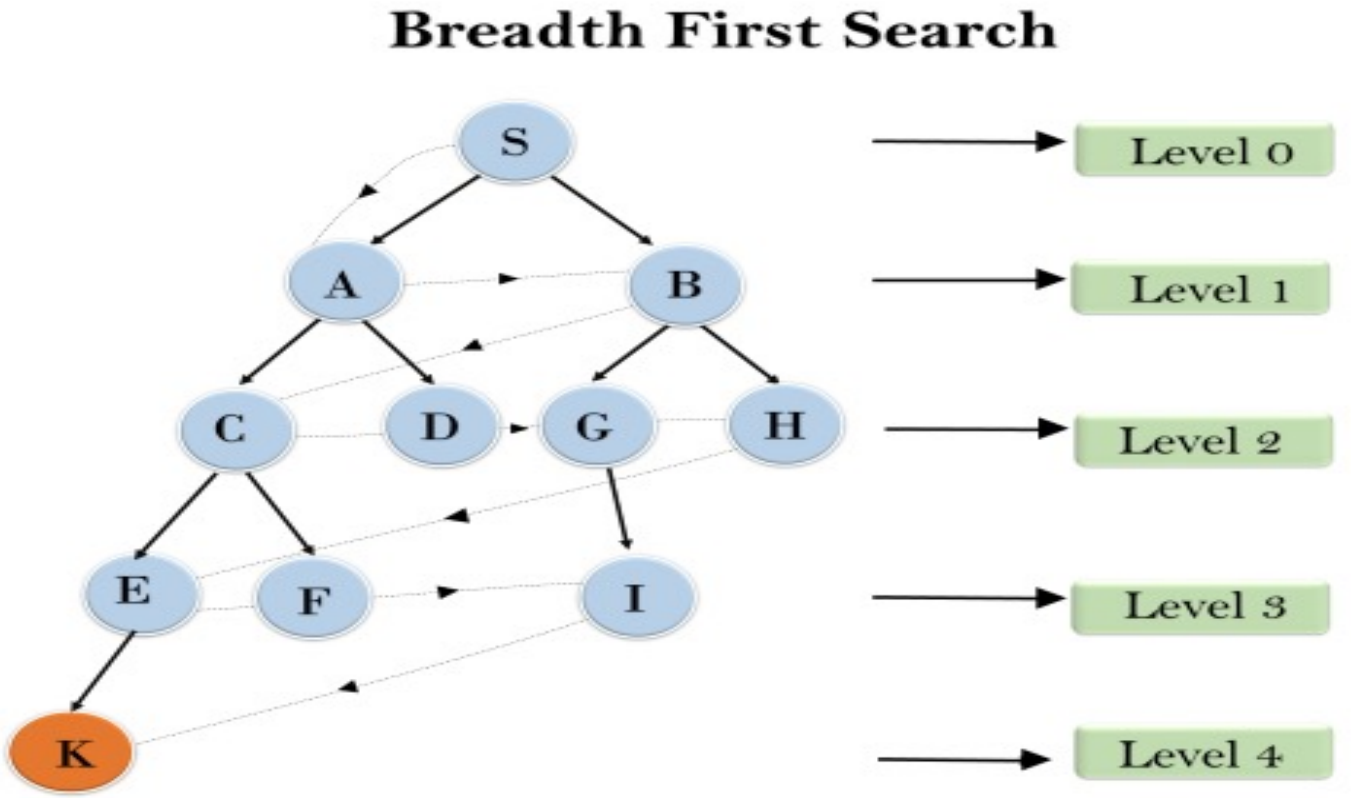
- BFS will provide a solution if any solution exists.
- If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.

Disadvantages of BFS

- It requires lots of memory since each level of the tree must be saved into memory to expand the next level.
- BFS needs lots of time if the solution is far away from the root node.

Example

In the below tree structure, we have shown the traversing of the tree using BFS algorithm from the root node S to goal node K. BFS search algorithm traverse in layers, so it will follow the path which is shown by the dotted arrow, and the traversed path will be:



S---> A--->B---->C--->D---->G--->H--->E---->F---->I---->K

Depth-first Search

- Depth-first search is a recursive algorithm for traversing a tree or graph data structure.
- It is called the depth-first search because it starts from the root node and follows each path to its greatest depth node before moving to the next path.
- DFS uses a stack data structure for its implementation.
- The process of the DFS algorithm is like the BFS algorithm.

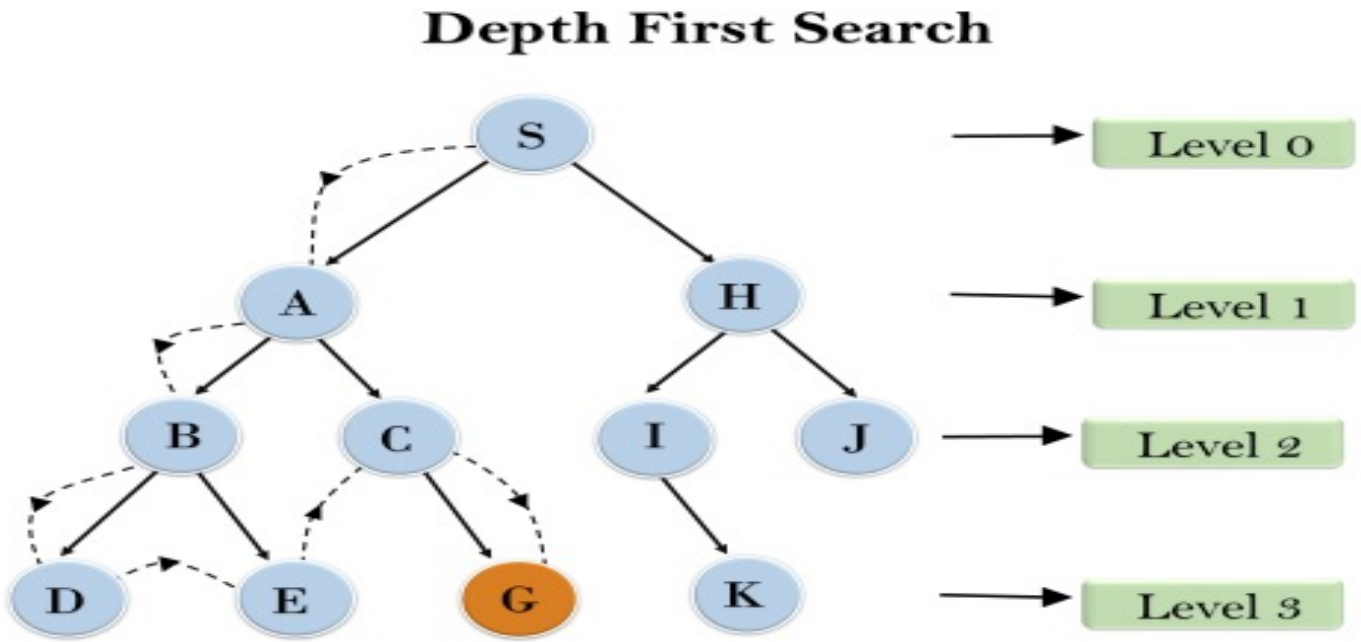
Advantage of DFS

- DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
- It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).

Disadvantage of DFS

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

Example


Depth First Search
S
A
H
B
C
I
J
D
E
G
K
Level 0
Level 1
Level 2
Level 3

- In the below search tree, we have shown the flow of depth-first search, and it will follow the order as:
- Root node--->Left node ----> right node.
- It will start searching from root node S, and traverse A, then B, then D and E, after traversing E, it will backtrack the tree as E has no other successor and still goal node is not found. After backtracking it will traverse node C and then G, and here it will terminate as it found goal node.

Difference between BFS and DFS

BFS	DFS
It extends for Breadth-first search.	It extends for Depth-first search.
It searches a node breadthwise, i.e., covering each level one by one.	It searches a node depthwise, i.e., covering one path deeply.
It uses the queue to store data in the memory.	It uses the stack to store data in the memory.
BFS is a vertex-based algorithm.	DFS is an edge-based algorithm.
The structure of a BFS tree is wide and short.	The structure of a DFS tree is narrow and long.

BFS	DFS
The oldest unexpanded node is its first priority to explore it.	The nodes along the edge are explored first.
BFS is used to examine bipartite graph, connected path as well as shortest path present in the graph.	DFS is used to examine a two-edge connected graph, acyclic graph, and also the topological order.

Depth-Limited Search Algorithm(DLS)

A depth-limited search algorithm is similar to depth-first search with a **predetermined limit**. Depth-limited search can solve the drawback of the **infinite path** in the Depth-first search. In this algorithm, the node at the depth limit will treat as it has no successor nodes further.

Depth-limited search can be terminated with two Conditions of failure:

- **Standard failure value**: It indicates that problem does not have any solution.
- **Cutoff failure value**: It defines no solution for the problem within a given depth limit.

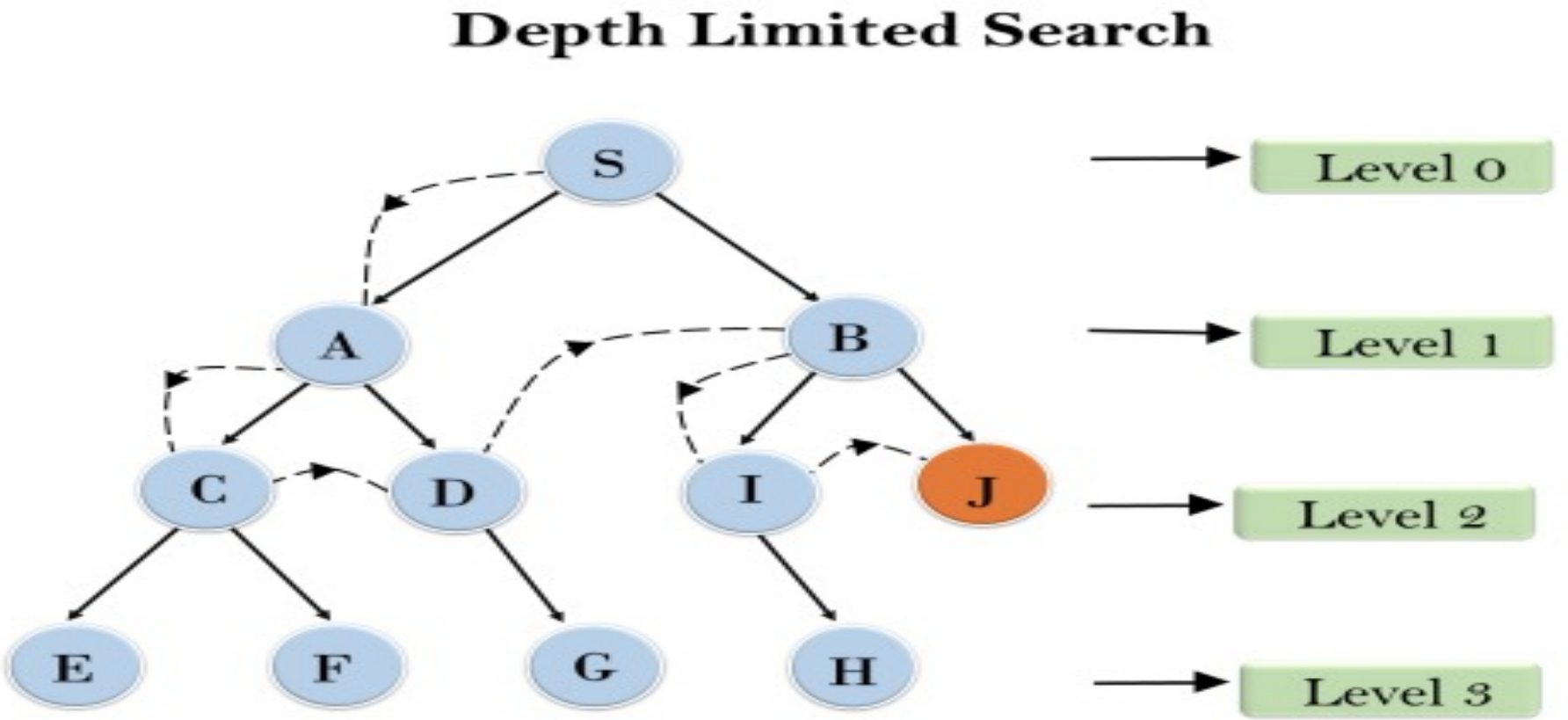
Advantages:

- Depth-limited search is Memory efficient.

Disadvantages:

- Depth-limited search also has a disadvantage of incompleteness.
- It may not be optimal if the problem has more than one solution.

Example


Depth Limited Search
S
A
B
C
D
I
J
E
F
G
H
Level 0
Level 1
Level 2
Level 3

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 depth limit until a goal is found.
- This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found.
- This Search algorithm combines the benefits of Breadth-first search's fast search and depth-first search's memory efficiency.
- The iterative search algorithm is useful uninformed search when search space is large, and depth of goal node is unknown.

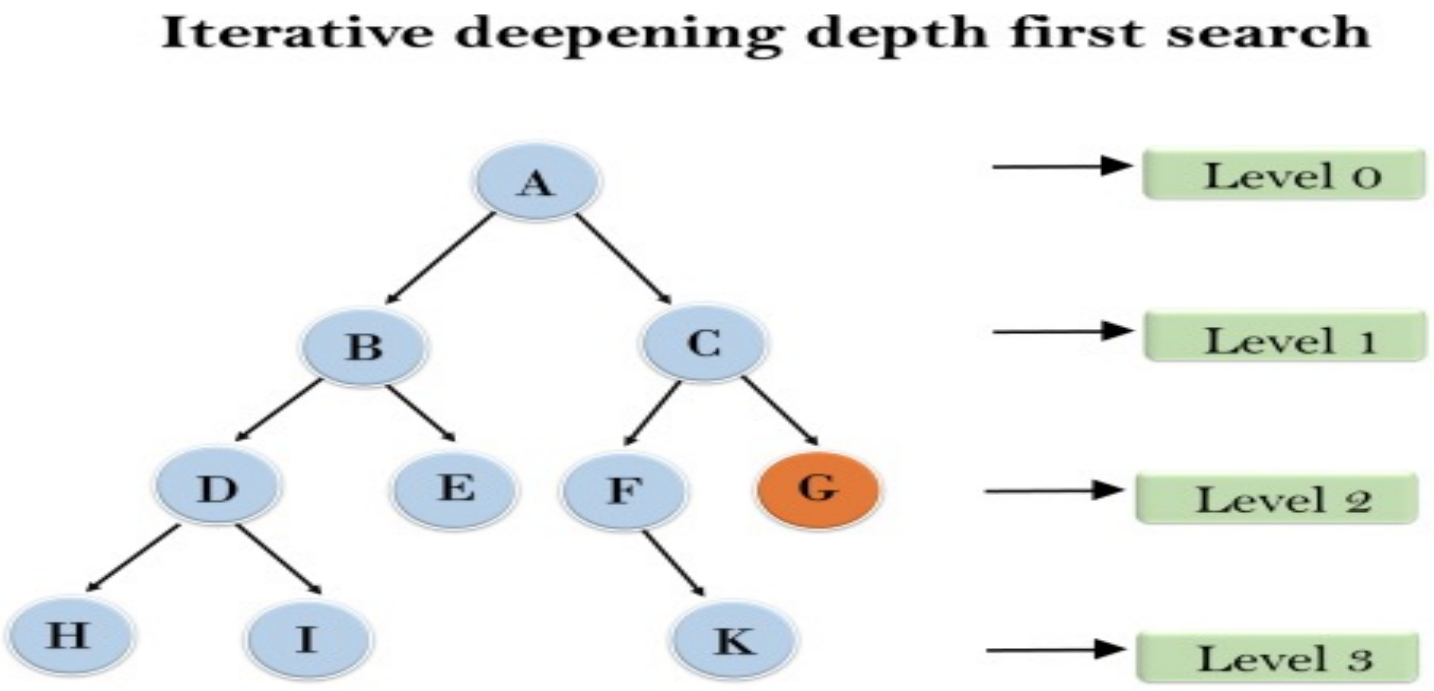
Advantages:

It combines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.

Disadvantages:

The main drawback of IDDFS is that it repeats all the work of the previous phase.

Example


Iterative deepening depth first search
A
B
C
D
E
F
G
H
I
K
Level 0
Level 1
Level 2
Level 3

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.

Uniform-cost Search Algorithm

- Uniform-cost search is a searching algorithm used for traversing a weighted tree or graph.
- This algorithm comes into play when a different cost is available for each edge.
- The primary goal of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost.
- Uniform-cost search expands nodes according to their path costs form the root node. It can be used to solve any graph/tree where the optimal cost is in demand.

- A uniform-cost search algorithm is implemented by the priority queue.
- It gives maximum priority to the lowest cumulative cost.
- Uniform cost search is equivalent to BFS algorithm if the path cost of all edges is the same.

Advantages:

Uniform cost search is optimal because at every state the path with the least cost is chosen.

Disadvantages:

It does not care about the number of steps involve in searching and only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.

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 sub graphs in which one starts the search from an initial vertex and other starts from goal vertex. The search stops when these two graphs intersect each other.
- Bidirectional search can use search techniques such as BFS, DFS, DLS, etc.

Advantages:

Bidirectional search is fast.

Bidirectional search requires less memory

Disadvantages:

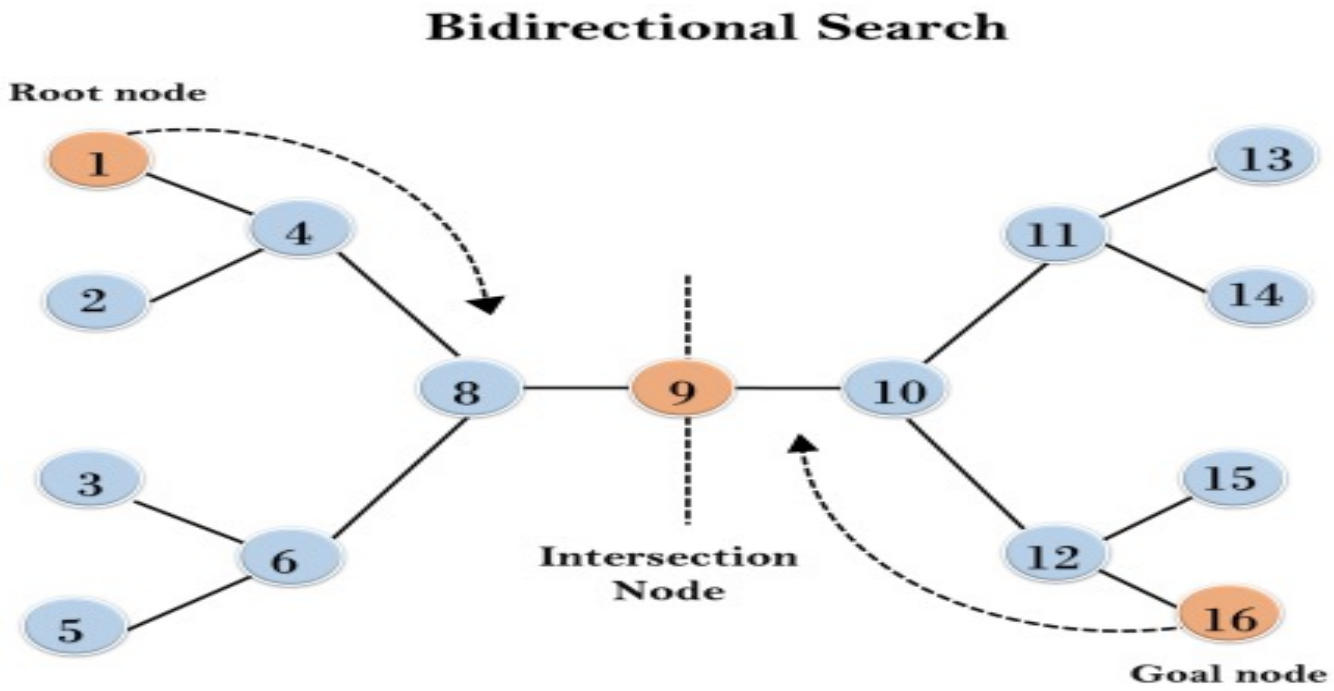
Implementation of the bidirectional search tree is difficult.

In bidirectional search, one should know the goal state in advance.

When to use bidirectional approach?

- Both initial and goal states are unique and completely defined.
- The branching factor is exactly the same in both directions.

Example


Bidirectional Search
Root node
1
2
3
4
5
6
8
9
10
11
12
13
14
15
16
Intersection Node
Goal node

- 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.

Informed Search Strategies

- Informed search algorithm contains an array of knowledge such as how far we are from the goal, path cost, how to reach to goal node, etc.
- This knowledge help agents to explore less to the search space and find more efficiently the goal node.
- The informed search algorithm is more useful for large search space. Informed search algorithm uses the idea of heuristic, so it is also called Heuristic search.

Heuristics Search

- 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.

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.

In the informed search we will discuss three Main algorithms which are given below:

Greedy Best First Search

A* Search Algorithm

AO* Search Algorithm

Best First Search

- 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 evaluation function and search.
- In the best first search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by evaluation function-

$$f(n) = g(n)$$

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 g(n), and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both list, then add it to the OPEN list.

Step 7: Return to Step 2.

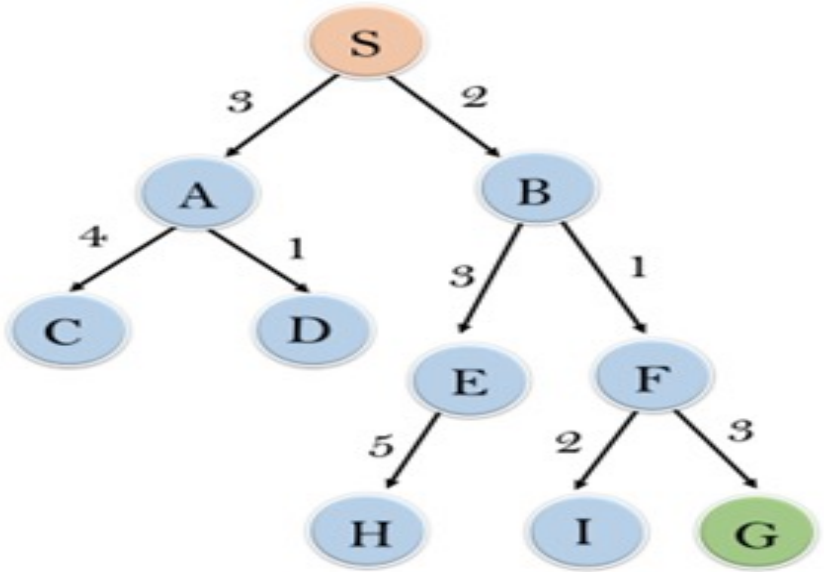
Advantages of BFS:

- It is more efficient than that of BFS and DFS.
- Time complexity of Best first search is much less than Breadth first search.

Disadvantages of BFS:

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

Example



node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0

In this search example, we are using two lists which are **OPEN** and **CLOSED** Lists. Following are the iteration for traversing the above example.

Expand the nodes of S and put in the CLOSED list

Initialization: Open [A, B], Closed [S]

Iteration 1: Open [A], Closed [S, B]

Iteration 2: Open [E, F, A], Closed [S, B]

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

Iteration 3: Open [I, G, E, A], Closed [S, B, F]

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

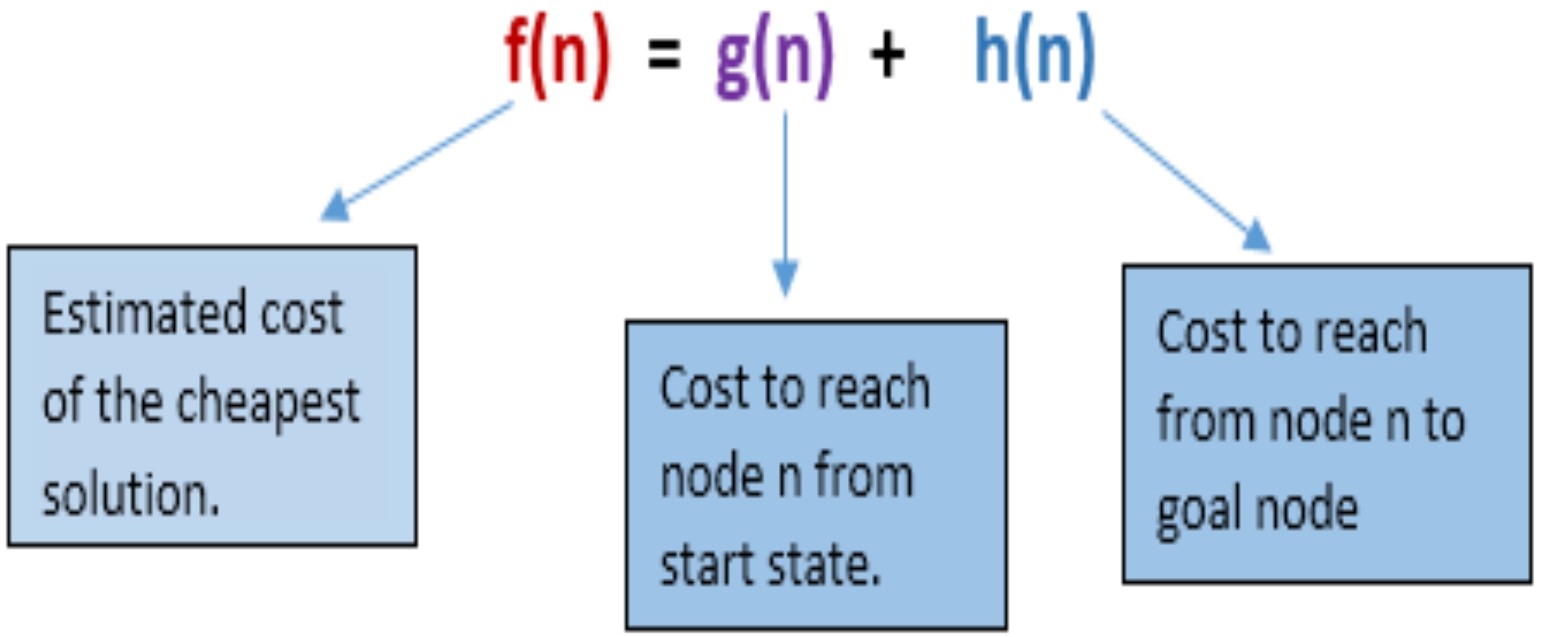
Hence the final solution path will be:

S----> B----->F----> G

A* Search Algorithm

- A* search is the most commonly known form of best-first search.
- It uses heuristic function h(n), and cost to reach the node n from the start state g(n).
- It has combined features of UCS and greedy best-first search, by which it solve the problem efficiently.
- A* search algorithm finds the shortest path through the search space using the heuristic function.

- This search algorithm expands less search tree and provides optimal result faster.
- A* algorithm is similar to UCS except that it uses g(n)+h(n) instead of g(n).
- In A* search algorithm, we use search heuristic as well as the cost to reach the node.
- Hence we can combine both costs as following, and this sum is called as a **fitness number**.


f(n) = g(n) + h(n)
Estimated cost of the cheapest solution.
Cost to reach node n from start state.
Cost to reach from node n to goal node

Algorithm of A* search

- **Step1:** Place the starting node in the OPEN list.
- **Step 2:** Check if the OPEN list is empty or not, if the list is empty then return failure and stops.
- **Step 3:** Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise

Step 4: Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.

Step 5: Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.

Step 6: Return to **Step 2**.

Advantages:

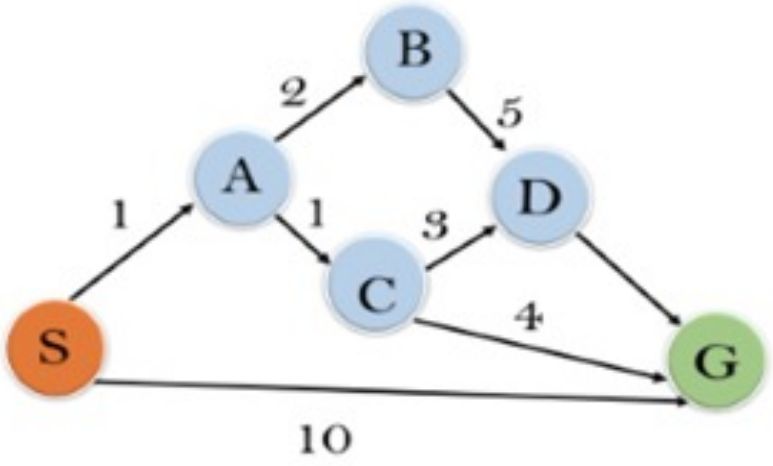
- A* search algorithm is the best algorithm than other search algorithms.
- A* search algorithm is optimal and complete.
- This algorithm can solve very complex problems.

Disadvantages:

- It does not always produce the shortest path as it mostly based on heuristics and approximation.
- A* search algorithm has some complexity issues.
- The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

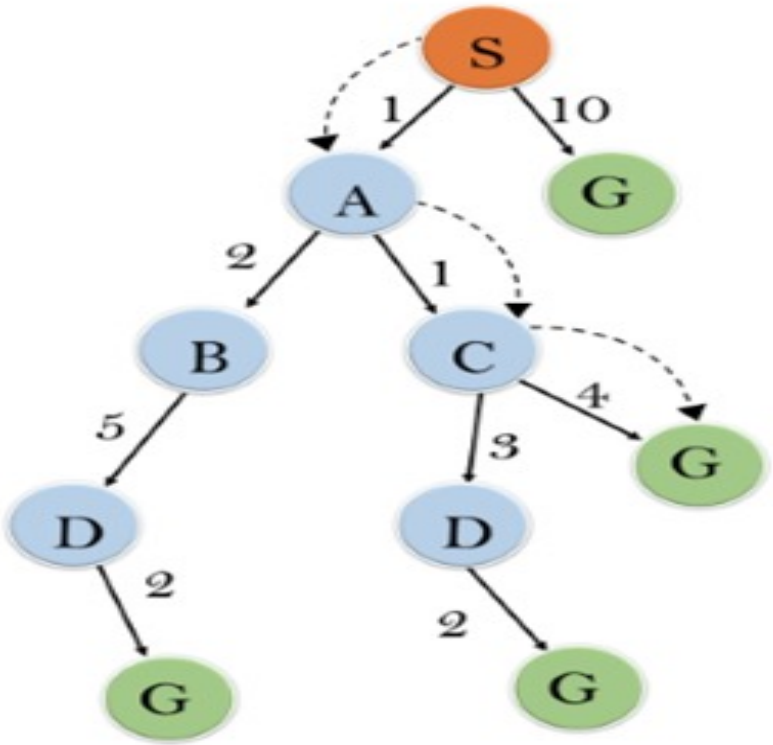
Example

- 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.

Local Search Algorithms

A local search algorithm completes its task by traversing on a single current node rather than multiple paths and following the neighbors of that node generally.

Although local search algorithms are not systematic, still they have the following two advantages:

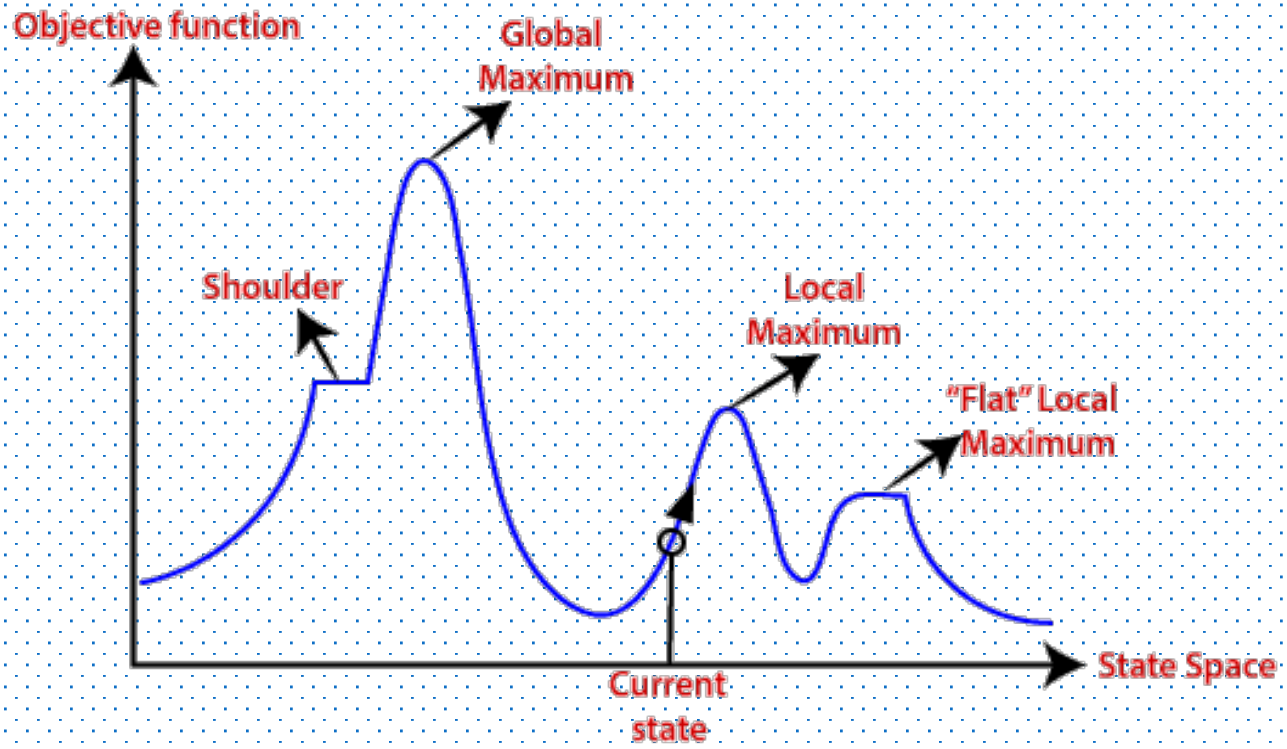
- Local search algorithms use a very little or constant amount of memory as they operate only on a single path.
- Most often, they find a reasonable solution in large or infinite state spaces where the classical or systematic algorithms do not work.
- Local search algorithm works for pure optimized problems. A pure optimization problem is one where all the nodes can give a solution. But the target is to find the best state out of all according to the **objective function**. Unfortunately, the pure optimization problem fails to find high-quality solutions to reach the goal state from the current state.

Types of local searches

- Hill-climbing Search
- Simulated Annealing
- Local Beam Search

Hill-climbing Search

- The topographical regions shown in the figure can be defined as:
- **Global Maximum:** It is the highest point on the hill, which is the goal state.
- **Local Maximum:** It is the peak higher than all other peaks but lower than the global maximum.
- **Flat local maximum:** It is the flat area over the hill where it has no uphill or downhill. It is a saturated point of the hill.
- **Shoulder:** It is also a flat area where the summit is possible.
- **Current state:** It is the current position of the person.



A one-dimensional state-space landscape in which elevation corresponds to the objective function

Types of Hill climbing search algorithm

There are following types of hill-climbing search:

- Simple hill climbing
- Steepest-ascent hill climbing
- Stochastic hill climbing

Simple hill climbing search

- Simple hill climbing is the simplest technique to climb a hill. The task is to reach the highest peak of the mountain.
- Here, the movement of the climber depends on his move/steps. If he finds his next step better than the previous one, he continues to move else remain in the same state. This search focus only on his previous and next step.

Simple hill climbing Algorithm

- Create a **CURRENT** node, **NEIGHBOUR** node, and a **GOAL** node.
- If the **CURRENT node=GOAL node**, return **GOAL** and terminate the search.
- Else **CURRENT node<= NEIGHBOUR node,** move ahead.
- Loop until the goal is not reached or a point is not found.

Steepest-ascent hill climbing

- Steepest-ascent hill climbing is different from simple hill climbing search. Unlike simple hill climbing search, It considers all the successive nodes, compares them, and choose the node which is closest to the solution.
- Steepest hill climbing search is similar to **best-first search** because it focuses on each node instead of one.

Steepest-ascent hill climbing algorithm

- Create a **CURRENT** node and a **GOAL** node.
- If the **CURRENT node=GOAL** node, return **GOAL** and terminate the search.
- Loop until a better node is not found to reach the solution.
- If there is any better successor node present, expand it.
- When the **GOAL** is attained, return **GOAL** and terminate.

Stochastic hill climbing

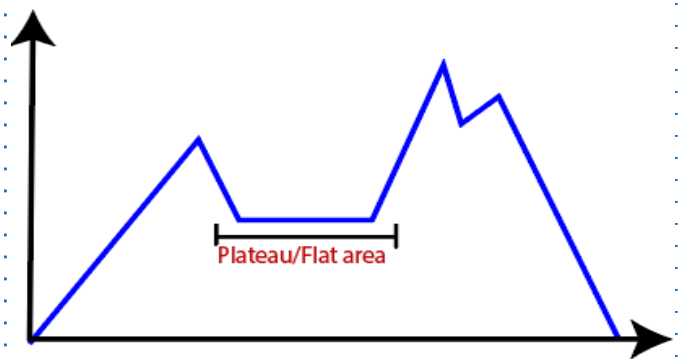
- Stochastic hill climbing does not focus on all the nodes. It selects one node at random and decides whether it should be expanded or search for a better one.

Limitations of Hill climbing algorithm

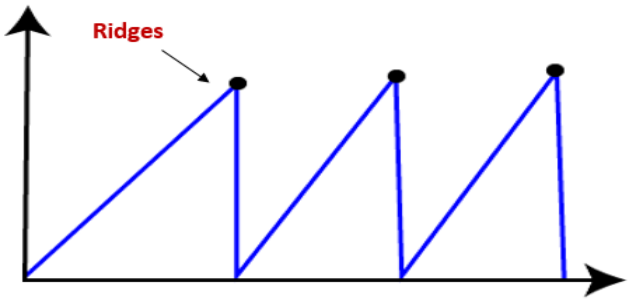
Hill climbing algorithm is a fast and furious approach. It finds the solution state rapidly because it is quite easy to improve a bad state. But, there are following limitations of this search:

- **Local Maxima:** It is that peak of the mountain which is highest than all its neighboring states but lower than the global maxima. It is not the goal peak because there is another peak higher than it.

- **Plateau:** It is a flat surface area where no uphill exists. It becomes difficult for the climber to decide that in which direction he should move to reach the goal point. Sometimes, the person gets lost in the flat area.



- **Ridges:** It is a challenging problem where the person finds two or more local maxima of the same height commonly. It becomes difficult for the person to navigate the right point and stuck to that point itself.



Features of Hill Climbing

- **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.

Local Beam Search

- Local beam search is quite different from random-restart search. It keeps track of **k** states instead of just one.
- It selects **k** randomly generated states, and expand them at each step.
- If any state is a goal state, the search stops with success. Else it selects the best **k** successors from the complete list and repeats the same process.
- In random-restart search where each search process runs independently, but in local beam search, the necessary information is shared between the parallel search processes.

Adversarial Search

- Adversarial search is a **game-playing** technique where the agents are surrounded by a competitive environment.
- A conflicting goal is given to the agents (multiagent).
- These agents compete with one another and try to defeat one another in order to win the game.
- Such conflicting goals give rise to the adversarial search. Here, game-playing means discussing those games where **human intelligence** and **logic factor** is used, excluding other factors such as **luck factor**. **Tic-tac-toe, chess, checkers**, etc., are such type of games where no luck factor works, only mind works.

Techniques required to get the best optimal solution

- **Pruning:** A technique which allows ignoring the unwanted portions of a search tree which make no difference in its result.
- **Heuristic Evaluation Function:** It allows to approximate the cost value at each level of the search tree, before reaching the goal node.

Elements of Game Playing search

- **S_0:** It is the initial state from where a game begins.
- **PLAYER (s):** It defines which player is having the current turn to make a move in the state.
- **ACTIONS (a):** It defines the set of legal moves to be used in a state.
- **RESULT (s, a):** It is a transition model which defines the result of a move.

TERMINAL-TEST (p): It defines that the game has ended and returns true.

UTILITY (s,p): It defines the final value with which the game has ended. This function is also known as **Objective function** or **Payoff function**. The price which the winner will get i.e.

(-1): If the PLAYER loses.

(+1): If the PLAYER wins.

(0): If there is a draw between the PLAYERS.

Types of algorithms in Adversarial search

An **adversarial search**, the result depends on the players which will decide the result of the game.

- **Minimax Algorithm**
- **Alpha-beta Pruning.**

Minimax

Minimax is a **decision-making** strategy under **game theory,** which is used to minimize the losing chances in a game and to maximize the winning chances. This strategy is also known as '**Minmax,**' '**MM,**' **or 'Saddle point'.**

MINIMAX Algorithm

MINIMAX algorithm is a backtracking algorithm where it backtracks to pick the best move out of several choices. MINIMAX strategy follows the **DFS (Depth-first search)** concept. Here, we have two players **MIN and MAX,** and the game is played alternatively between them, i.e., when **MAX** made a move, then the next turn is of **MIN.** It means the move made by MAX is fixed and, he cannot change it. The same concept is followed in DFS strategy, i.e., we follow the same path and cannot change in the middle. That's why in MINIMAX algorithm, instead of BFS, we follow DFS.

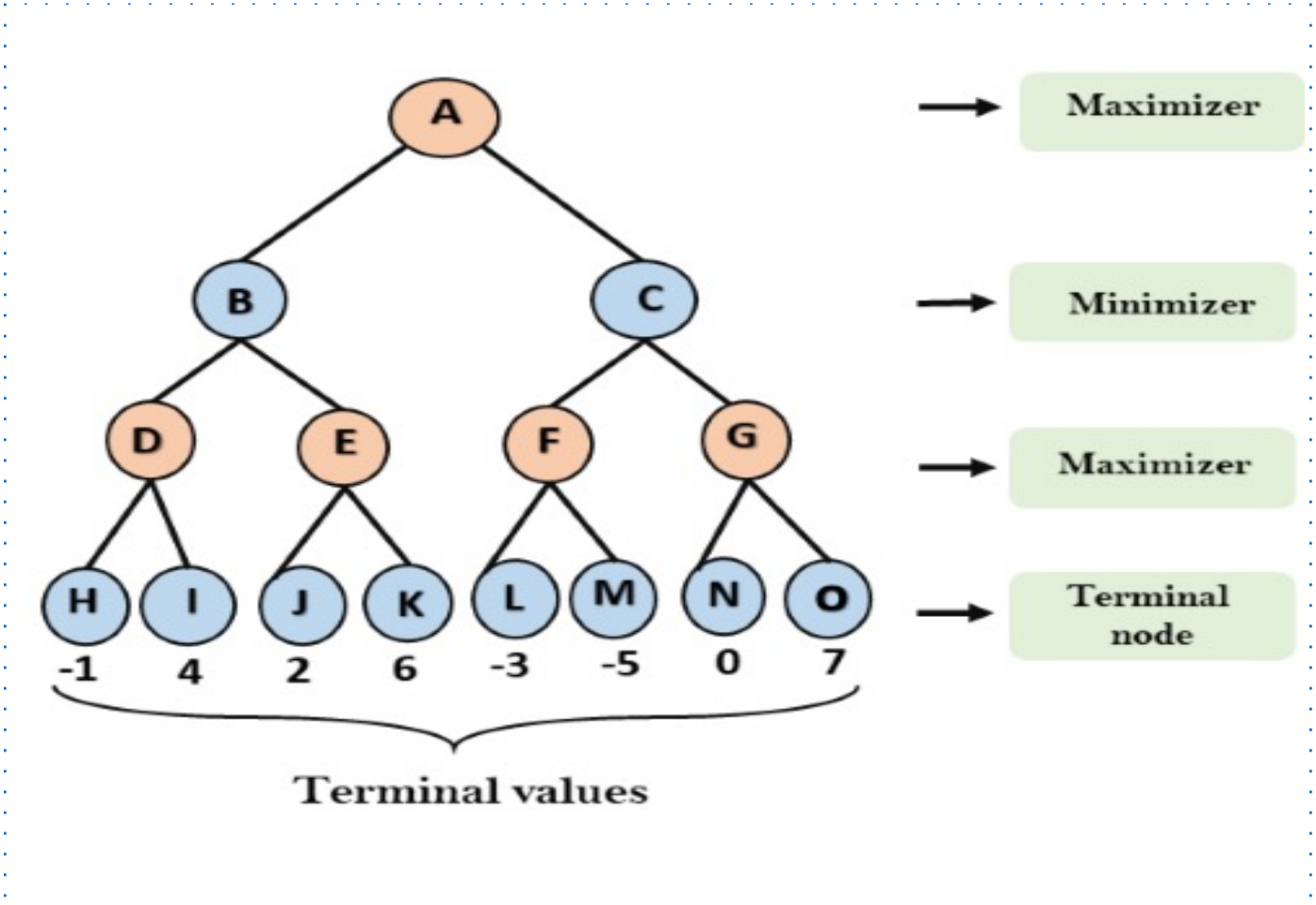
- Keep on generating the game tree/ search tree till a limit **d.**
- Compute the move using a heuristic function.
- Propagate the values from the leaf node till the current position following the minimax strategy.
- Make the best move from the choices.

Limitation of the minimax Algorithm

- The main drawback of the minimax algorithm is that it gets really slow for complex games such as Chess, go, etc.
- This type of games has a huge branching factor, and the player has lots of choices to decide.
- This limitation of the minimax algorithm can be improved from **alpha-beta pruning** which we have discussed in the next topic.

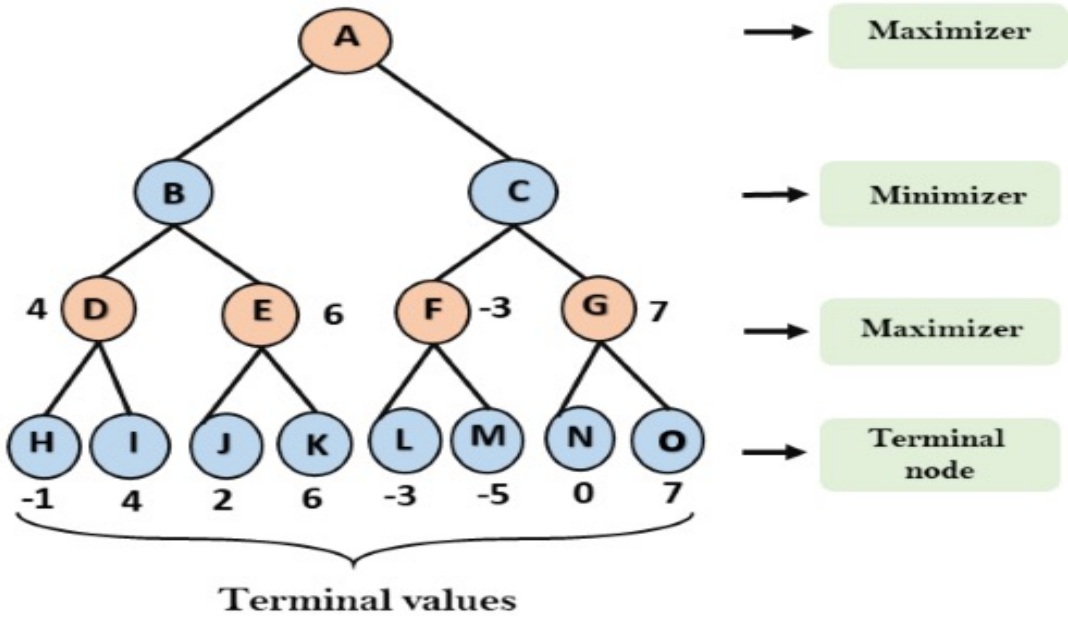
Example

- **Step-1:** In the first step, the algorithm generates the entire game-tree and apply the utility function to get the utility values for the terminal states. In the below tree diagram, let's take A is the initial state of the tree. Suppose maximizer takes first turn which has worst-case initial value =- infinity, and minimizer will take next turn which has worst-case initial value = +infinity.


A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
-1
4
2
6
-3
-5
0
7
Terminal values
Maximizer
Minimizer
Maximizer
Terminal node

Step 2: Now, first we find the utilities value for the Maximizer, its initial value is -∞, so we will compare each value in terminal state with initial value of Maximizer and determines the higher nodes values. It will find the maximum among the all.

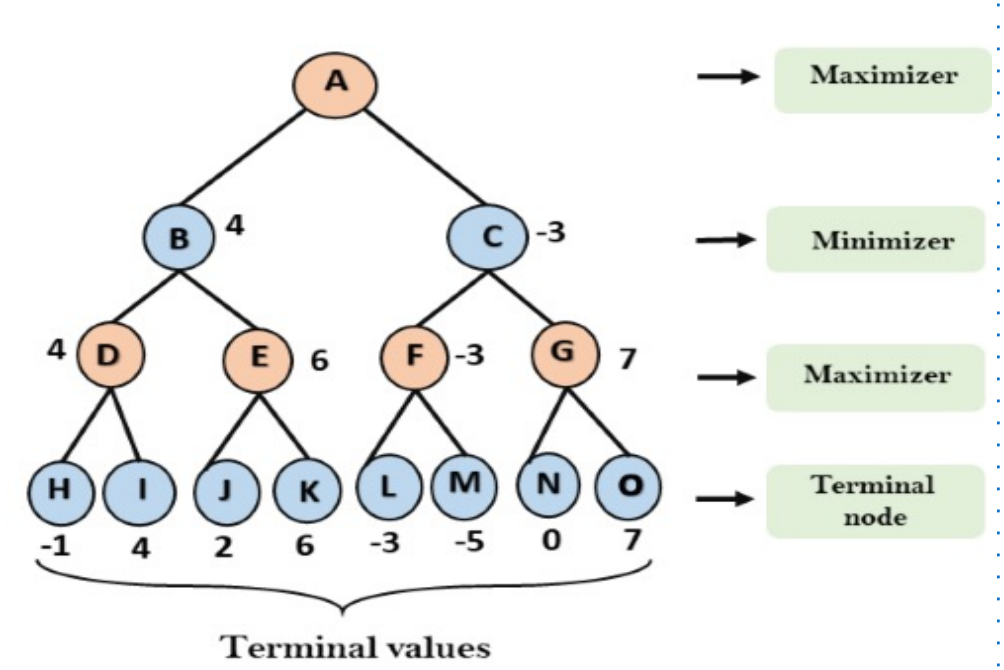
For node D max(-1,- -∞) => max(-1,4)= 4
For Node E max(2, -∞) => max(2, 6)= 6
For Node F max(-3, -∞) => max(-3,-5) = -3
For node G max(0, -∞) = max(0, 7) = 7



Step 3: In the next step, it's a turn for minimizer, so it will compare all nodes value with +∞, and will find the 3rd layer node values.

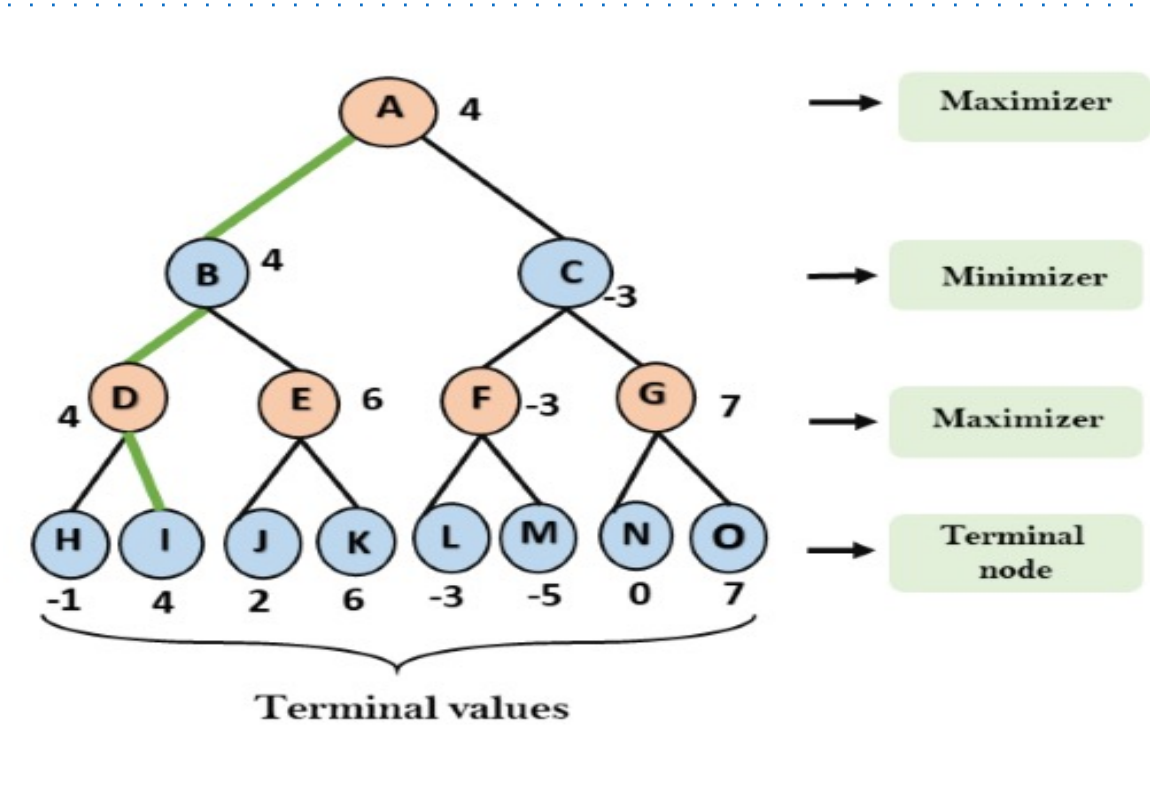
For node B= min(4,6) = 4

For node C= min (-3, 7) = -3



Step 4: Now it's a turn for Maximizer, and it will again choose the maximum of all nodes value and find the maximum value for the root node. In this game tree, there are only 4 layers, hence we reach immediately to the root node, but in real games, there will be more than 4 layers.

For node A max(4, -3)= 4



Alpha-Beta Pruning

- Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.
- As we have seen in the minimax search algorithm that the number of game states it has to examine are exponential in depth of the tree. Since we cannot eliminate the exponent, but we can cut it to half. Hence there is a technique by which without checking each node of the game tree we can compute the correct minimax decision, and this technique is called **pruning**. This involves two threshold parameter **Alpha** and **beta** for future expansion, so it is called **alpha-beta pruning**. It is also called as **Alpha-Beta Algorithm**.
- Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prune the tree leaves but also entire sub-tree.

The two-parameter can be defined as:

Alpha: The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is -∞.

Beta: The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is +∞.

The Alpha-beta pruning to a standard minimax algorithm returns the same move as the standard algorithm does, but it removes all the nodes which are not really affecting the final decision but making algorithm slow. Hence by pruning these nodes, it makes the algorithm fast.

Condition for Alpha-beta pruning

- The main condition which required for alpha-beta pruning is:

$$\alpha >= \beta$$

- The Max player will only update the value of alpha.
- The Min player will only update the value of beta.
- While backtracking the tree, the node values will be passed to upper nodes instead of values of alpha and beta.
- We will only pass the alpha, beta values to the child nodes.

Pseudo-code for Alpha-beta Pruning

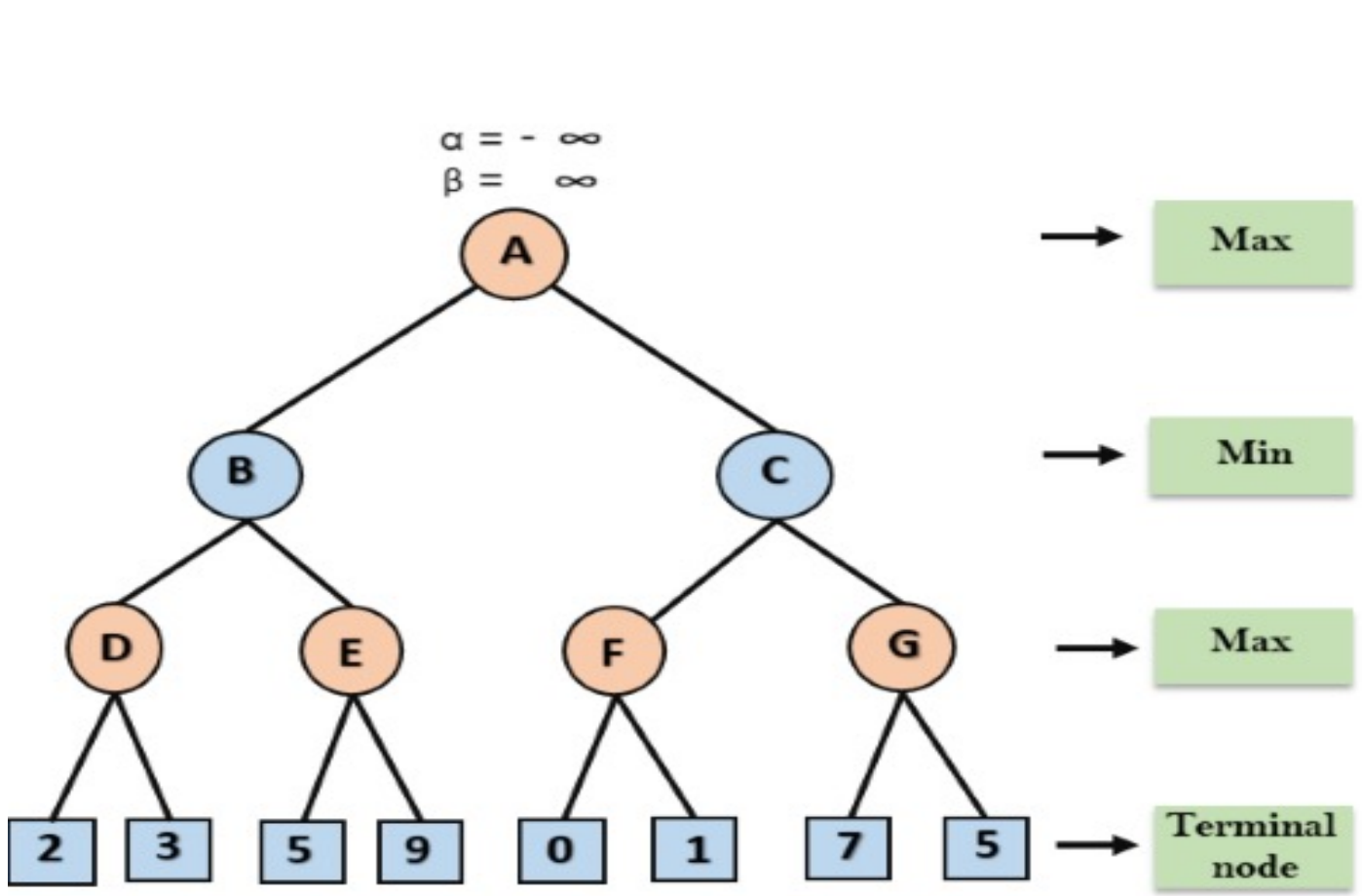
```
function minimax(node, depth, alpha, beta, maximizingPlayer) is
if depth ==0 or node is a terminal node then
return static evaluation of node

if MaximizingPlayer then      // for Maximizer Player
  maxEva= -infinity
  for each child of node do
  eva= minimax(child, depth-1, alpha, beta, False)
 maxEva= max(maxEva, eva)
 alpha= max(alpha, maxEva)
  if beta<=alpha
```

```
 break
 return maxEva
else                        // for Minimizer player
  minEva= +infinity
  for each child of node do
  eva= minimax(child, depth-1, alpha, beta, true)
  minEva= min(minEva, eva)
  beta= min(beta, eva)
   if beta<=alpha
 break
return minEva
```

Working of Alpha-Beta Pruning

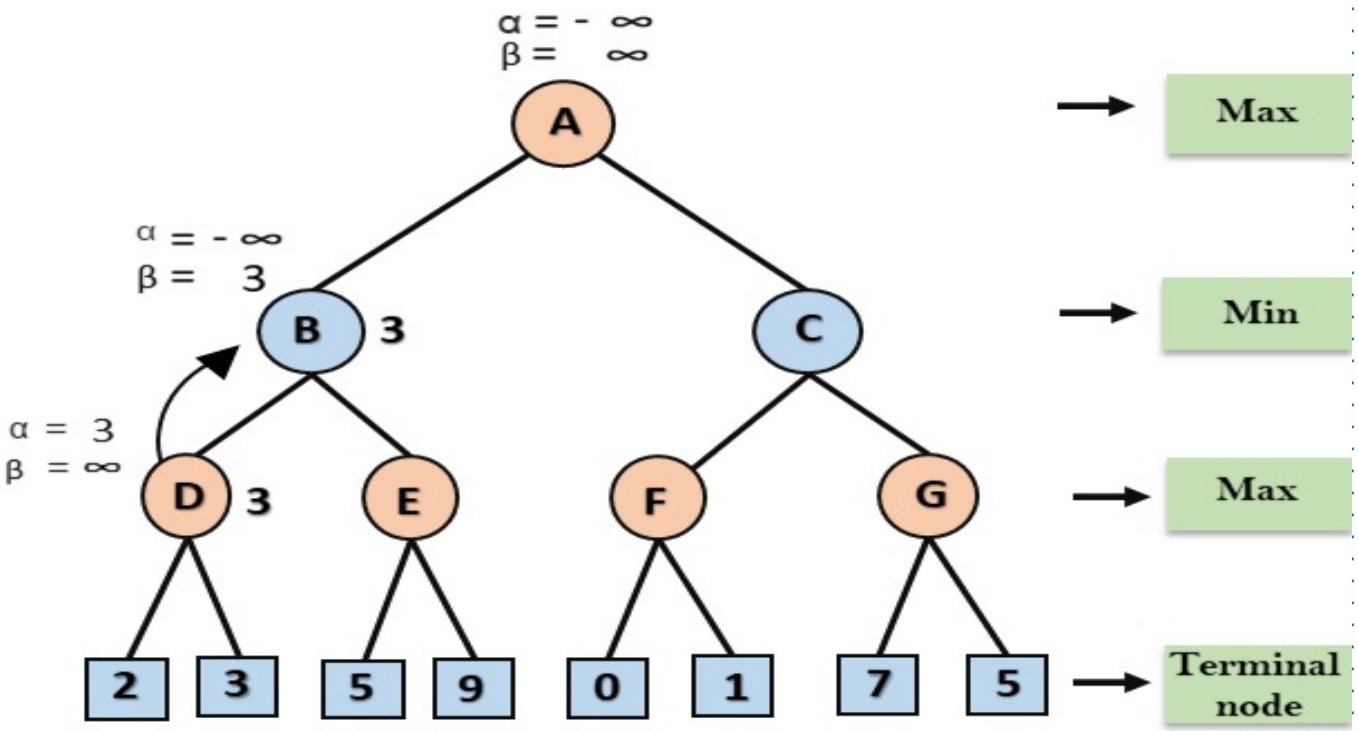
Let's take an example of two-player search tree to understand the working of Alpha-beta pruning:

- **Step 1:** At the first step the, Max player will start first move from node A where $\alpha = -\infty$ and $\beta = +\infty$, these value of alpha and beta passed down to node B where again $\alpha = -\infty$ and $\beta = +\infty$, and Node B passes the same value to its child D.


α = - ∞
β = ∞
A
B
C
D
E
F
G
2
3
5
9
0
1
7
5
Max
Min
Max
Terminal node

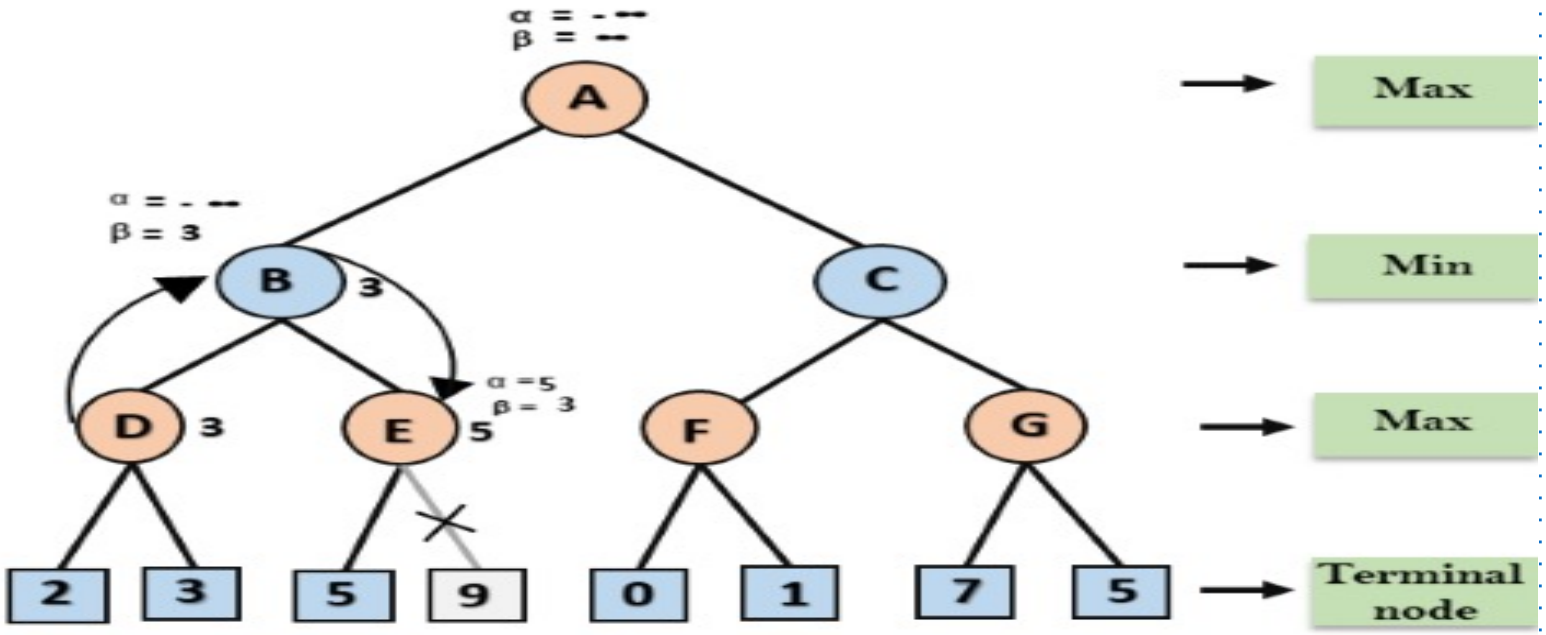
Step 2: At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.

Step 3: Now algorithm backtrack to node B, where the value of β will change as this is a turn of Min, Now β= +∞, will compare with the available subsequent nodes value, i.e. min (∞, 3) = 3, hence at node B now α= -∞, and β= 3.


α = - ∞
β = ∞
A
Max
α = - ∞
β = 3
B
3
C
Min
α = 3
β = ∞
D
3
E
F
G
Max
2
3
5
9
0
1
7
5
Terminal node

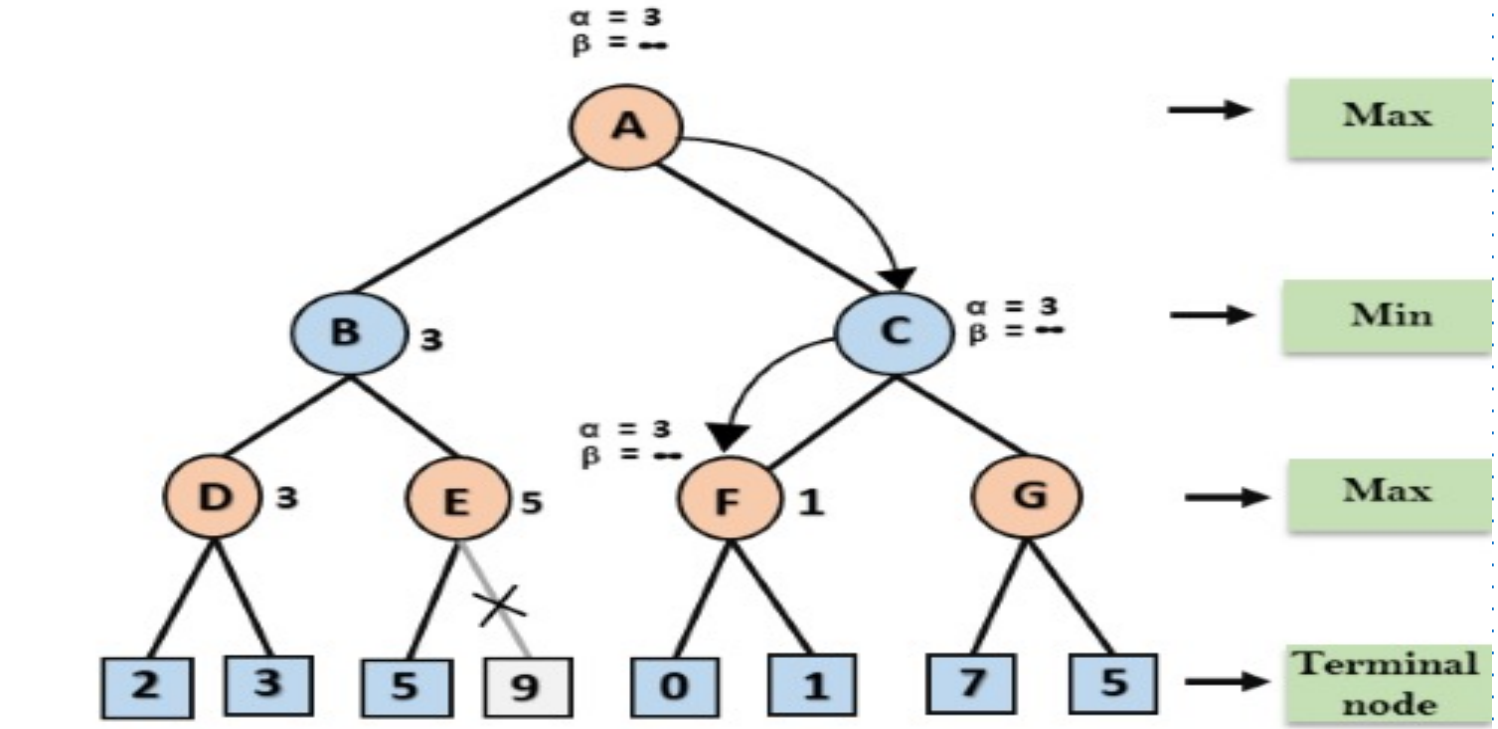
In the next step, algorithm traverse the next successor of Node B which is node E, and the values of $\alpha = -\infty$, and $\beta = 3$ will also be passed.

Step 4: At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so max $(-\infty, 5) = 5$, hence at node E $\alpha = 5$ and $\beta = 3$, where $\alpha >= \beta$, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will be 5.

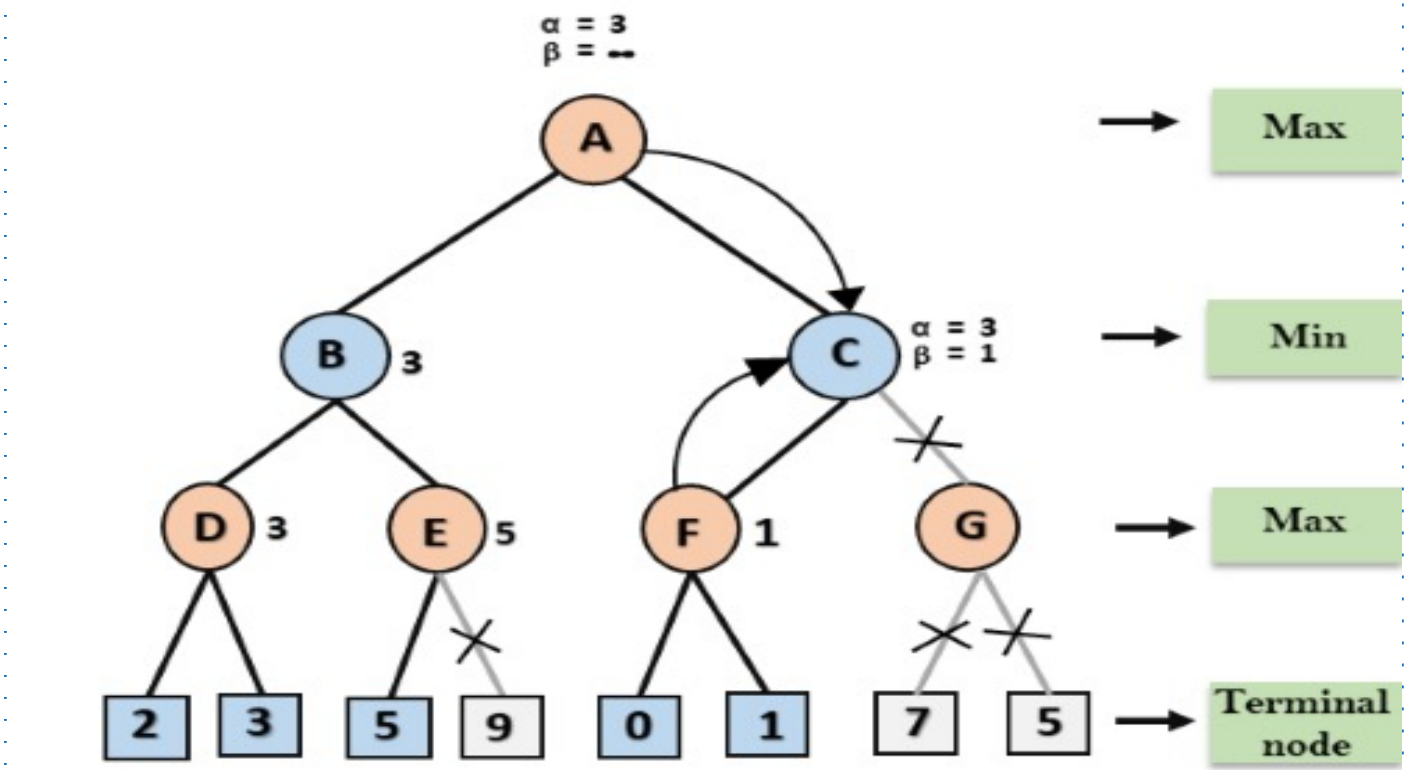

α = -∞
β = ∞
A
Max
α = -∞
β = 3
B
3
C
Min
α =5
β = 3
D
3
E
5
F
G
Max
2
3
5
9
0
1
7
5
Terminal node

Step 5: At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C.
At node C, α=3 and β= +∞, and the same values will be passed on to node F.

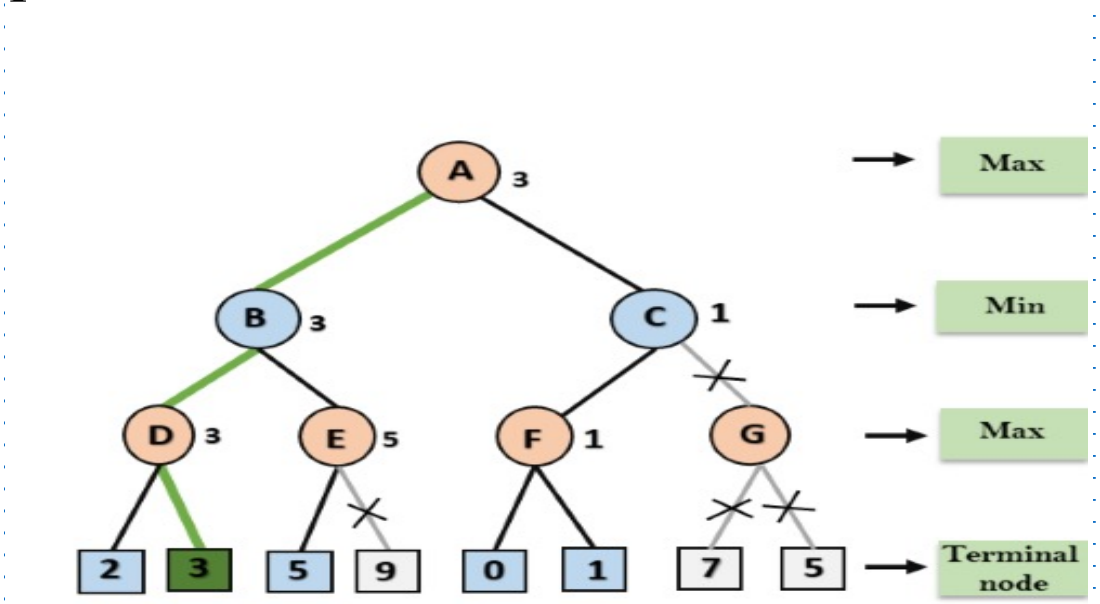
Step 6: At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become 1.


α = 3
β = ∞
A
Max
B
3
C
α = 3
β = ∞
Min
α = 3
β = ∞
D
3
E
5
F
1
G
Max
2
3
5
9
0
1
7
5
Terminal node

Step 7: Node F returns the node value 1 to node C, at C $\alpha = 3$ and $\beta = +\infty$, here the value of beta will be changed, it will compare with 1 so min $(\infty, 1) = 1$. Now at C, $\alpha=3$ and $\beta = 1$, and again it satisfies the condition $\alpha >= \beta$, so the next child of C which is G will be pruned, and the algorithm will not compute the entire sub-tree G.


α = 3
β = ∞
A
B
3
C
α = 3
β = 1
D
3
E
5
F
1
G
2
3
5
9
0
1
7
5
Max
Min
Max
Terminal node

Step 8: C now returns the value of 1 to A here the best value for A is max (3, 1) = 3. Following is the final game tree which is the showing the nodes which are computed and nodes which has never computed. Hence the optimal value for the maximizer is 3 for this example.



Move Ordering in Alpha-Beta pruning

The effectiveness of alpha-beta pruning is highly dependent on the order in which each node is examined. Move order is an important aspect of alpha-beta pruning. It can be of two types:

- **Worst ordering:** In some cases, alpha-beta pruning algorithm does not prune any of the leaves of the tree, and works exactly as minimax algorithm. In this case, it also consumes more time because of alpha-beta factors, such a move of pruning is called worst ordering. In this case, the best move occurs on the right side of the tree. The time complexity for such an order is $O(b^m)$.

Ideal ordering: The ideal ordering for alpha-beta pruning occurs when lots of pruning happens in the tree, and best moves occur at the left side of the tree. We apply DFS hence it first search left of the tree and go deep twice as minimax algorithm in the same amount of time. Complexity in ideal ordering is $O(b^{m/2})$.

Example