UNIT-2

Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored.

Random Search:

A random search algorithm refers to an algorithm that uses some kind of randomness or probability may be called a Monte Carlo method or a stochastic algorithm.

Random search involves generating and evaluating random inputs to the objective function. It's effective because it does not assume anything about the structure of the objective function. This can be beneficial for problems where there is a lot of domain expertise that may influence or bias the optimization strategy, allowing non-intuitive solutions to be discovered.

Algorithm

1. Initialize **x** with a random position in the search-space.
2. Until a termination criterion is met (e.g. number of iterations performed, or adequate fitness reached), repeat the following:
    1. Sample a new position **y** from the hypersphere of a given radius surrounding the current position **x**.
    2. If $f(\mathbf{y}) < f(\mathbf{x})$ then move to the new position by setting $\mathbf{x} = \mathbf{y}$

Search by open list and closed list:

Open list helps you in both depth first and breadth first searches to traverse your tree properly. Think about algorithms step by step. You are in a node with many children and your are going to expand one of them. After expansion there should be a mechanism to get back and continue your traversal. Open list performs that for you and tells you what is actually the next node to be expand. And the algorithm only clarify the order of child insertion into the list.

And Closed list generally improves the speed of algorithm. It prevents the algorithm from expanding pre-visited nodes. Maybe you reach node A that was expanded previously through another branch. This will let you cut this branch and try another path.

This was explanation of Open and Closed lists. You asked about when the algorithm finishes. Actually you will repeat the *expand and add to Open list* until you find your goal or Open list goes empty.

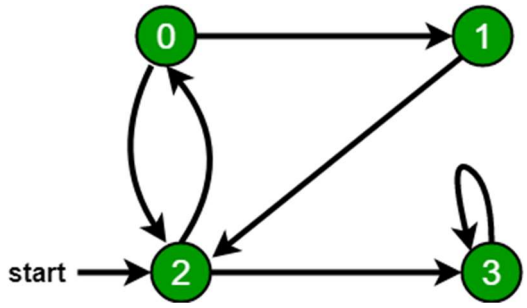
Breadth-First Search

Generate nodes in the tree in order of their distance from the root. That is, all nodes at distance one, followed by all nodes at distance two, etc. The first path to a goal will be of shortest length. The corresponding data structure for storing nodes during search is a queue.

The only catch here is, that, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we divide the vertices into two categories:

i) Visited and
ii) Not visited.

A boolean visited array is used to mark the visited vertices. For simplicity, it is assumed that all vertices are reachable from the starting vertex. BFS uses a queue data structure for traversal.



When we come to vertex 0, we look for all adjacent vertices of it.

2 is also an adjacent vertex of 0.

If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process.

There can be multiple BFS traversals for a graph. Different BFS traversals for the above graph :

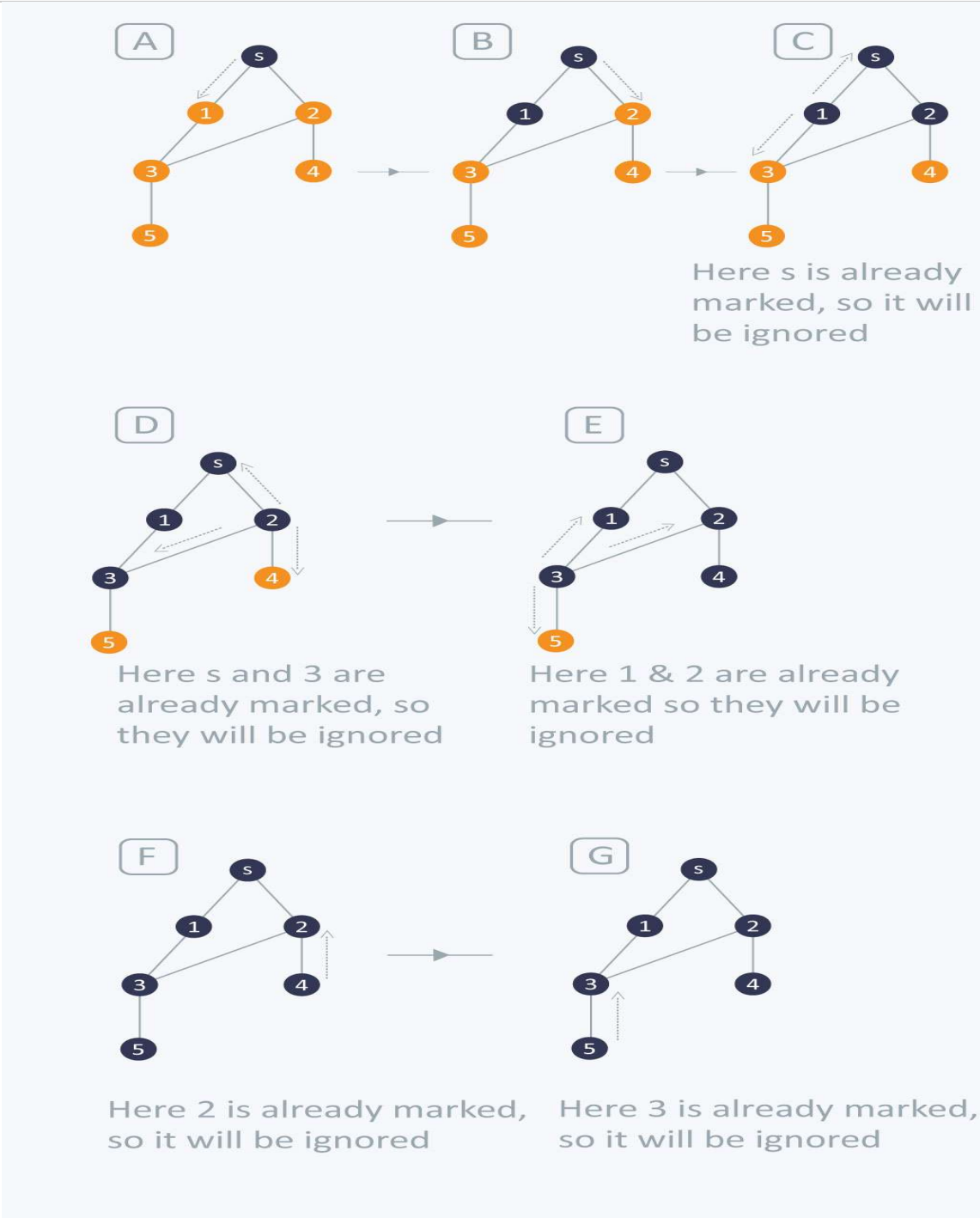
2, 3, 0, 1

2, 0, 3, 1

Algorithm:

```
BFS (G, s)                    //Where G is the graph and s is the source
node
      let Q be queue.
      Q.enqueue( s ) //Inserting s in queue until all its neighbour
vertices are marked.

      mark s as visited.
      while ( Q is not empty)
           //Removing that vertex from queue,whose neighbour will be
visited now
```

```
            v   =   Q.dequeue( )

          //processing all the neighbours of v
          for all neighbours w of v in Graph G
               if w is not visited
                         Q.enqueue( w )              //Stores w in Q to
further visit its neighbour
                         mark w as visited.
```



Program:

```
from collections import defaultdict

# This class represents a directed graph
# using adjacency list representation
class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # Function to print a BFS of graph
    def BFS(self, s):

        # Mark all the vertices as not visited
        visited = [False] * (max(self.graph) + 1)

        # Create a queue for BFS
        queue = []

        # Mark the source node as
        # visited and enqueue it
        queue.append(s)
        visited[s] = True

        while queue:

            # Dequeue a vertex from
            # queue and print it
            s = queue.pop(0)
            print (s, end = " ")

            # Get all adjacent vertices of the
            # dequeued vertex s. If a adjacent
            # has not been visited, then mark it
            # visited and enqueue it
            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True

# Driver code

# Create a graph given in
# the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)
```

```
print ("Following is Breadth First Traversal"
                  " (starting from vertex 2)")
g.BFS(2)
```

Depth-First Search

Follow one path deep into the tree until a goal is found or backtracking is required. It is reasonable when unproductive paths aren't too long. The corresponding data structure for storing nodes during search is a stack.

Depth First Search (popularly abbreviated as DFS) is a recursive graph searching technique.

The Algorithm

While doing a DFS, we maintain a set of visited nodes. Initially this set is empty.
When DFS is called on any vertex (say), first that vertex is marked as visited and then for every edge going out of that vertex, , such that is unvisited, we call DFS on .
Finally, we return when we have exhausted all the edges going out from .

```
void dfs(int node)  {
 seen[node] = true;
 for(int next: graph[node])  {
  if(not seen[next])  {
   dfs(next);
  }
 }
}
```

A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

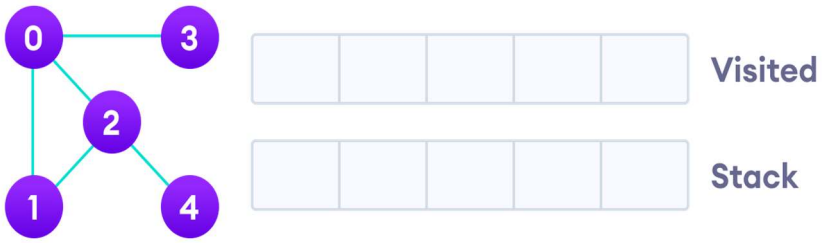
The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.

4. Keep repeating steps 2 and 3 until the stack is empty.

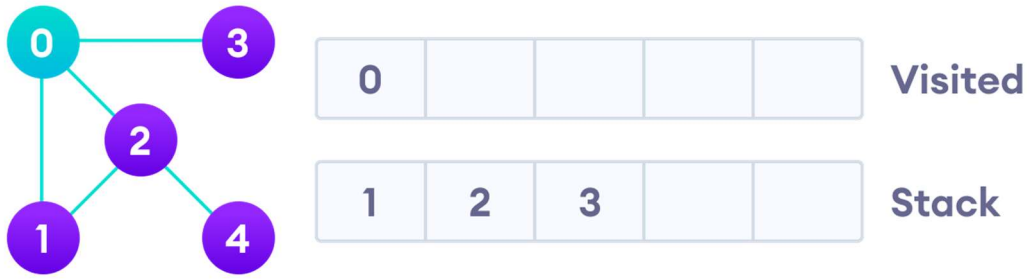
Example

Let's see how the Depth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



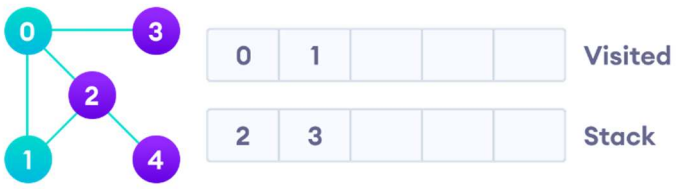
Undirected graph with 5 vertices

We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



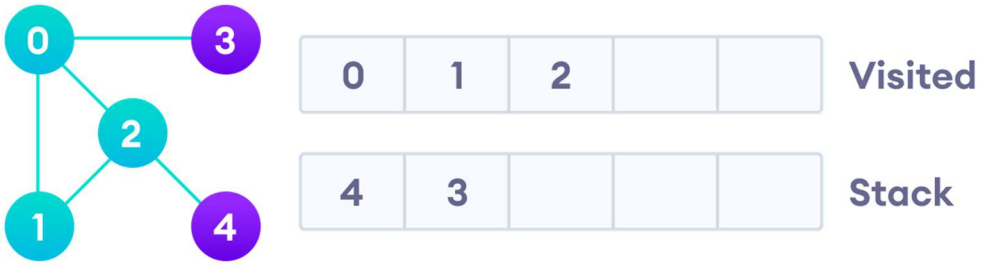
Visit the element and put it in the visited list

Next, we visit the element at the top of stack i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

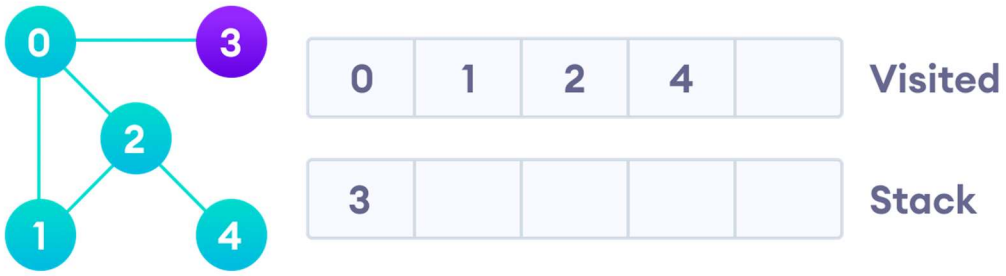


Visit the element at the top of stack

Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

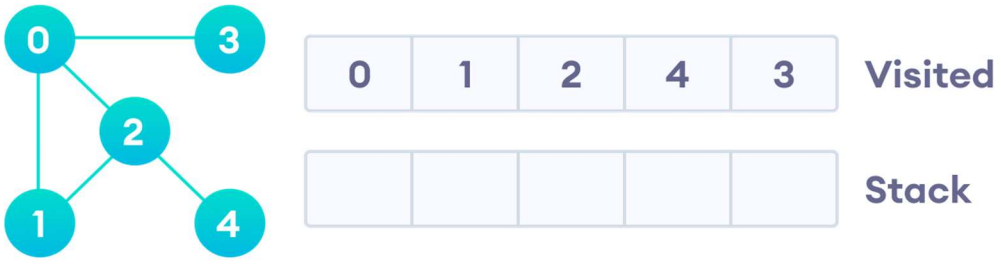


Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.



Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.



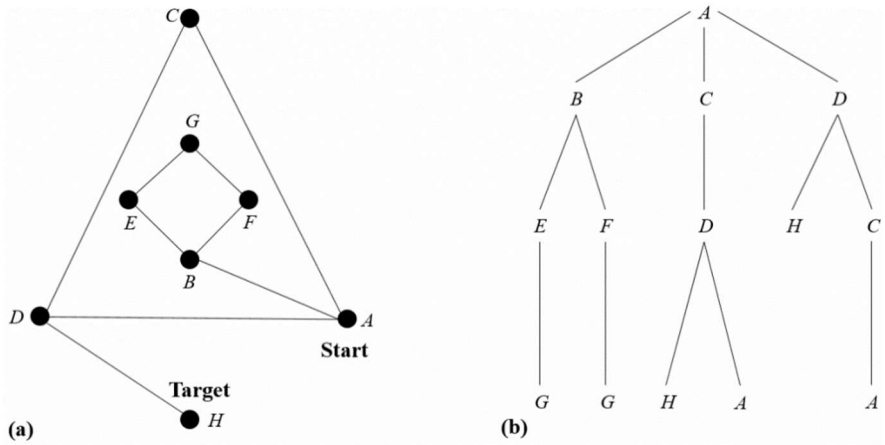
After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.

```
# DFS algorithm in Python
# DFS algorithm
def dfs(graph, start, visited=None):
   if visited is None:
      visited = set()
   visited.add(start)
   print(start)
   for next in graph[start] - visited:
      dfs(graph, next, visited)
   return visited

graph = {'0': set(['1', '2']),
       '1': set(['0', '3', '4']),
       '2': set(['0']),
       '3': set(['1']),
       '4': set(['2', '3'])}

dfs(graph, '0')
```

Heuristic Search:

Heuristic search is class of method which is used in order to search a solution space for an optimal solution for a problem. The heuristic here uses some method to search the solution space while assessing where in the space the solution is most likely to be and focusing the search on that area. It is often possible to model the process of solving a problem as a search through a solution space starting from an initial possible solution, with a method or set of rules dictating how to move from one possible solution to another. This method or rule set must be applied repeatedly in order to eventually satisfy some goal condition which indicates that a solution has been found.

In general, the exploration of a discrete solution space can be visualised as searching a graph with each vertex representing a possible solution, and an edge representing that possible solutions are adjacent to each other.



At times there is no graph representation available for the problem at the start of the process of finding a solution. In this case, whilst search is performed, a partial picture of the graph is gradually built up from nodes which have been explored. Here, for every iteration, all nodes adjacent to the one being explored according to set of transition rules are added to the graph (along with edges connecting them to the node being explored), this is called expanding the node. Practically, this is the application of all possible actions to that state. When doing this,

it is important to keep track of nodes which have already been generated in the search. Here, every node must be explored at least once in order for it to be present. As a result of this, it is possible to sort the set of all nodes which have been reached into one set of nodes which have been expanded, and another set which have been added to the graph and not yet expanded. The rest of these is known as the open list or search frontier, and the second set is known as the closed list. The set of all paths from the node at which search began up to the set of open nodes gives the the search tree of problem, which gives an illustration of the part of the solution space which has been explored by the search algorithm at a given time. Possible search roots for up to three moves on the graph in figure (a) above are shown on the search tree for the solution of the problem in figure (b). There are multiple different methods for performing searches on graphs, one distinction between methods is the order in which the solution space is searched, this is commonly either breadth first or depth first which shall be discussed further here.

Step	Exploring	Open	Closed
1		$\{A\}$	$\{\}$
2	A	$\{B, C, D\}$	$\{A\}$
3	B	$\{C, D, E, F\}$	$\{A, B\}$
4	C	$\{D, E, F\}$	$\{A, B, C\}$
5	D	$\{E, F, H\}$	$\{A, B, C, D\}$
6	E	$\{F, H, G\}$	$\{A, B, C, D, E\}$
7	F	$\{H, G\}$	$\{A, B, C, D, E, F\}$
8	H	$\{G\}$	$\{A, B, C, D, E, F, H\}$
9	G	$\{\}$	$\{A, B, C, D, E, F, H, G\}$

When performing breadth first search, the open set is considered to be a first in first out queue. So when a node is added to the open set, it is added to the bottom of the list. When a node is explored, the node at the top of the list is expanded and moved from the open set to the closed set. The result of this is that nodes are explored layer by layer and the graph is expanded from the start node.

When performing depth first search, the open set is considered to be a last in first out queue. This means that when a node is added to the open set, it is added to the top of the list. It also means that when a node is explored, the node at the top of the list is expanded and moved from the open set to the closed set. The result of this is that every iteration generates a node connected to the last node expanded (unless there are none, in this situation the algorithm backtracks to the last node with an unexplored node adjacent to it). This means the search effectively explores a path until that path has been completely explored, then goes back to the point at which that path branches off and explores other paths from here, repeating this process until all paths have been explored.

Step	Exploring	Open	Closed
1		$\{A\}$	$\{\}$
2	A	$\{B, C, D\}$	$\{A\}$
3	B	$\{E, F, C, D\}$	$\{A, B\}$
4	E	$\{G, F, C, D\}$	$\{A, B, E\}$
5	G	$\{F, C, D\}$	$\{A, B, E, G\}$
6	F	$\{C, D\}$	$\{A, B, E, G, F\}$
7	C	$\{D\}$	$\{A, B, E, G, F, C\}$
8	D	$\{H\}$	$\{A, B, E, G, F, C, D\}$
9	H	$\{\}$	$\{A, B, E, G, F, C, D, H\}$

Best first search:

Best first search is a traversal technique that decides which node is to be visited next by checking which node is the most promising one and then check it. For this it uses an evaluation function to decide the traversal.

This best first search technique of tree traversal comes under the category of heuristic search or informed search technique.

The cost of nodes is stored in a priority queue. This makes implementation of best-first search is same as that of breadth First search. We will use the priority queue just like we use a queue for BFS

Algorithm for implementing Best First Search

Step 1 : Create a priorityQueue pqueue.

Step 2 : insert ‘start’ in pqueue : pqueue.insert(start)

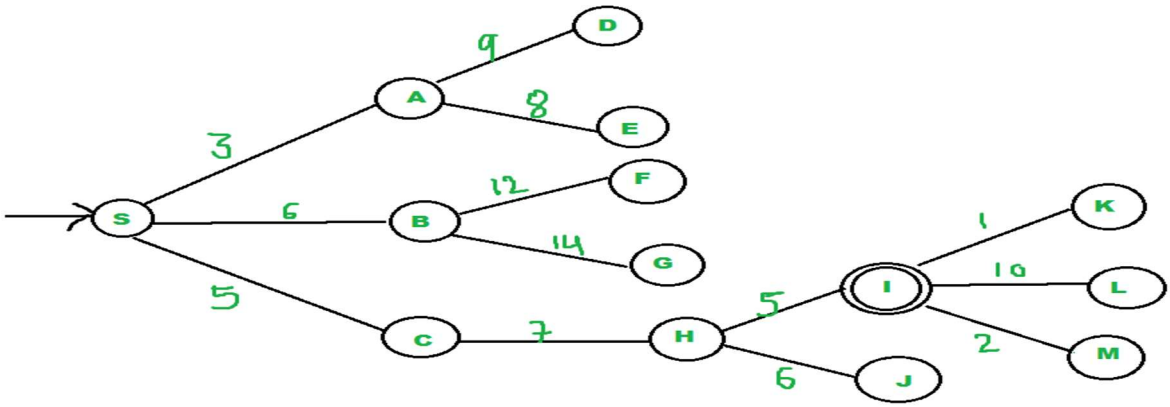
Step 3 : delete all elements of pqueue one by one.

Step 3.1 : if, the element is goal . Exit.

Step 3.2 : else, traverse neighbours and mark the node examined.

Step 4 : End.

Example:



We start from source “S” and search for goal “I” using given costs and Best First search.

pq initially contains S
We remove s from and process unvisited neighbors of S to pq.
pq now contains {A, C, B} (C is put before B because C has lesser cost)

We remove A from pq and process unvisited neighbors of A to pq.
pq now contains {C, B, E, D}

We remove C from pq and process unvisited neighbors of C to pq.
pq now contains {B, H, E, D}

We remove B from pq and process unvisited neighbors of B to pq.
pq now contains {H, E, D, F, G}
We remove H from pq.
Since our goal “I” is a neighbor of H, we return.

Program:

```
from queue import PriorityQueue
v = 14
graph = [[] for i in range(v)]

# Function For Implementing Best First Search
# Gives output path having lowest cost


def best_first_search(actual_Src, target, n):
    visited = [False] * n
    pq = PriorityQueue()
    pq.put((0, actual_Src))
    visited[actual_Src] = True

    while pq.empty() == False:
        u = pq.get()[1]
        # Displaying the path having lowest cost
        print(u, end=" ")
        if u == target:
            break

        for v, c in graph[u]:
```

```
            if visited[v] == False:
                visited[v] = True
                pq.put((c, v))
    print()

# Function for adding edges to graph


def addedge(x, y, cost):
    graph[x].append((y, cost))
    graph[y].append((x, cost))


# The nodes shown in above example(by alphabets) are
# implemented using integers addedge(x,y,cost);
addedge(0, 1, 3)
addedge(0, 2, 6)
addedge(0, 3, 5)
addedge(1, 4, 9)
addedge(1, 5, 8)
addedge(2, 6, 12)
addedge(2, 7, 14)
addedge(3, 8, 7)
addedge(8, 9, 5)
addedge(8, 10, 6)
addedge(9, 11, 1)
addedge(9, 12, 10)
addedge(9, 13, 2)

source = 0
target = 9
best_first_search(source, target, v)
```

A* algorithm:

It is a searching algorithm that is used to find the shortest path between an initial and a final point.

It is a handy algorithm that is often used for map traversal to find the shortest path to be taken. A* was initially designed as a graph traversal problem, to help build a robot that can find its own course. It still remains a widely popular algorithm for graph traversal.

It searches for shorter paths first, thus making it an optimal and complete algorithm. An optimal algorithm will find the least cost outcome for a problem, while a complete algorithm finds all the possible outcomes of a problem.

Another aspect that makes A* so powerful is the use of weighted graphs in its implementation. A weighted graph uses numbers to represent the cost of taking each path or course of action. This means that the algorithms can take the path with the least cost, and find the best route in terms of distance and time.

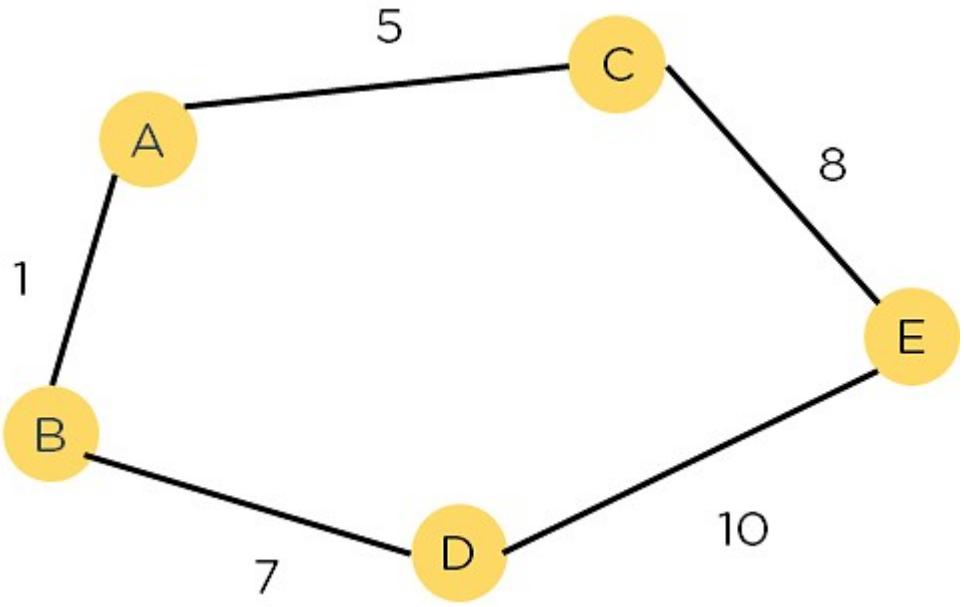


Figure 1: Weighted Graph

A major drawback of the algorithm is its space and time complexity. It takes a large amount of space to store all possible paths and a lot of time to find them.

A* Search Algorithm is a simple and efficient search algorithm that can be used to find the optimal path between two nodes in a graph. It will be used for the shortest path finding. It is an extension of Dijkstra's shortest path algorithm (Dijkstra's Algorithm). The extension here is that, instead of using a priority queue to store all the elements, we use heaps (binary trees) to store them. The A* Search Algorithm also uses a heuristic function that provides additional information regarding how far away from the goal node we are. This function is used in conjunction with the f-heap data structure in order to make searching more efficient.

Let us now look at a brief explanation of the A* algorithm.

Explanation

In the event that we have a grid with many obstacles and we want to get somewhere as rapidly as possible, the A* Search Algorithms are our savior. From a given starting cell, we can get to the target cell as quickly as possible. It is the sum of two variables' values that determines the node it picks at any point in time.

At each step, it picks the node with the smallest value of 'f' (the sum of 'g' and 'h') and processes that node/cell. 'g' and 'h' is defined as simply as possible below:

- 'g' is the distance it takes to get to a certain square on the grid from the starting point, following the path we generated to get there.

- ‘h’ is the heuristic, which is the estimation of the distance it takes to get to the finish line from that square on the grid.

Heuristics are basically educated guesses. It is crucial to understand that we do not know the distance to the finish point until we find the route since there are so many things that might get in the way (e.g., walls, water, etc.). In the coming sections, we will dive deeper into how to calculate the heuristics.

Let us now look at the detailed algorithm of A*.

Algorithm:

Initial condition - we create two lists - Open List and Closed List.

Now, the following steps need to be implemented -

- The open list must be initialized.
- Put the starting node on the open list (leave its f at zero). Initialize the closed list.
- Follow the steps until the open list is non-empty:

1. Find the node with the least f on the open list and name it “q”.
2. Remove Q from the open list.
3. Produce q's eight descendants and set q as their parent.
4. For every descendant:

i) If finding a successor is the goal, cease looking

ii)Else, calculate g and h for the successor.

successor.g = q.g + the calculated distance between the successor and the q.

successor.h = the calculated distance between the successor and the goal. We will cover three heuristics to do this: the Diagonal, the Euclidean, and the Manhattan heuristics.

successor.f = successor.g plus successor.h

iii) Skip this successor if a node in the OPEN list with the same location as it but a lower f value than the successor is present.

iv) Skip the successor if there is a node in the CLOSED list with the same position as the successor but a lower f value; otherwise, add the node to the open list end (for loop).

- Push Q into the closed list and end the while loop.

Example:

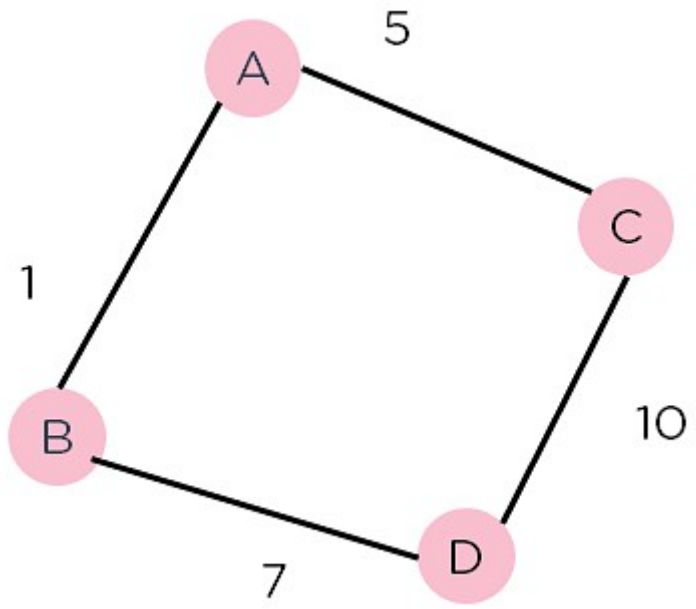


Figure 2: Weighted Graph 2

Consider the weighted graph depicted above, which contains nodes and the distance between them. Let's say you start from A and have to go to D.
Now, since the start is at the source A, which will have some initial heuristic value.
Hence, the results are
f(A) = g(A) + h(A)
f(A) = 0 + 6 = 6
Next, take the path to other neighbouring vertices :
f(A-B) = 1 + 4
f(A-C) = 5 + 2
Now take the path to the destination from these nodes, and calculate the weights :
f(A-B-D) = (1+ 7) + 0
f(A-C-D) = (5 + 10) + 0
It is clear that node B gives you the best path, so that is the node you need to take to reach the destination.

Game Search

Game playing is an important domain of artificial Intelligence. Games don't require much knowledge the only knowledge we need to provide is the rules, legal moves and the conditions of winning and losing the game.

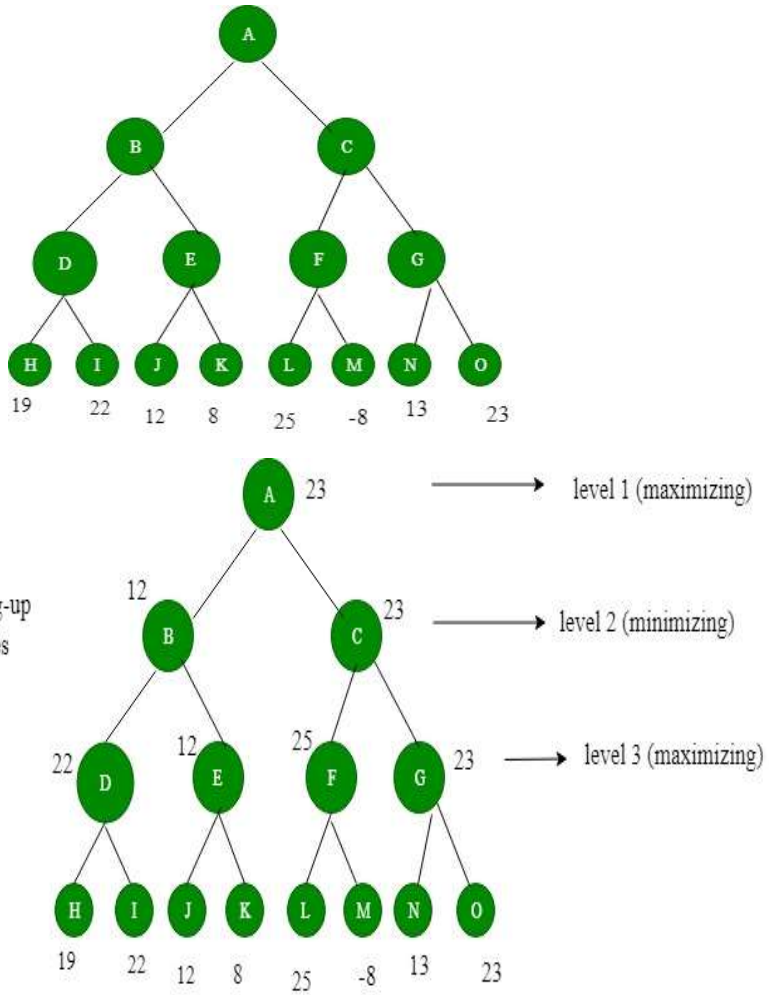
Both players try to win the game So both of them try to make the best moves possible at each turn. The most common technique in game playing is Minimax search algorithm, it is subset of depth first search algorithm. It is used in games like chess tic-tac-toe.

Minimax algorithm uses two functions:

MOVEGEN: Generates all possible moves that are generated from the current position

StaticEvaluation: It returns a value depending on the goodness from the viewpoint of two player.

This algorithm is for a two player game. So called player1 and player 2 the value of each node is backed up from its children. For player1 the backed up value is the maximum value of its children and for the player 2 the backed up value is the minimum value of its children it provides most promising move to player 1 assuming that player2 has make the best move.



Game Tree:

The different states of the game are represented by the game tree, In game tree the nodes are arranged in levels that corresponds to each players turn in the game so that the root node of the tree is the beginning position of the game. Assume Tic Tac Toe game the empty grid with no Xs and

Os is the starting and beginning position of the game. The next possible states results from first players move may be X and O. We call them as children for root node.

Each node in the second level the nodes are the predictive steps for the second player . This continued level by level until reaching states where the game is over.

The player gets line of three wins or if the board is full, game ends and tie

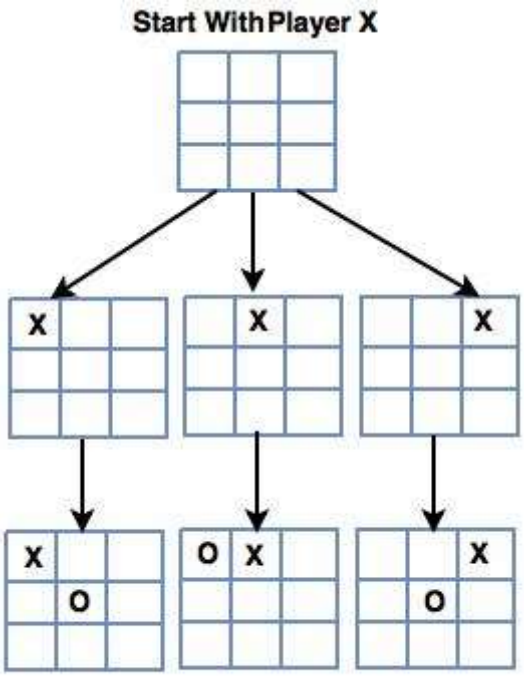
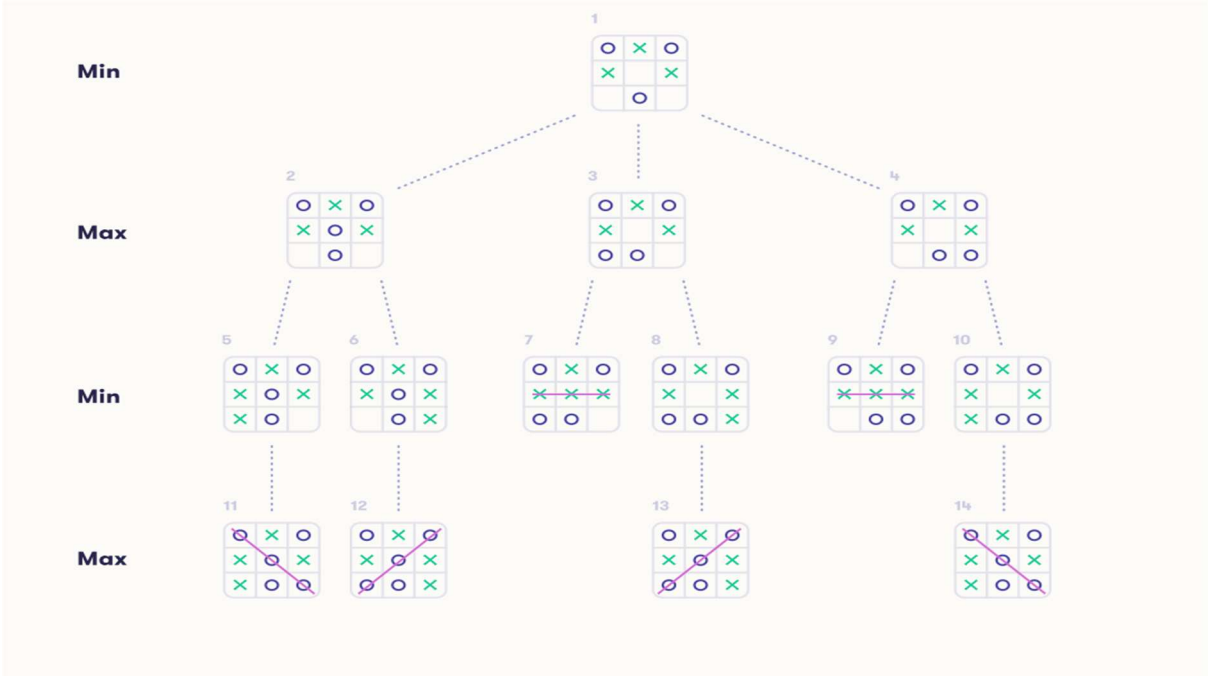


Fig (a): Initial Moves

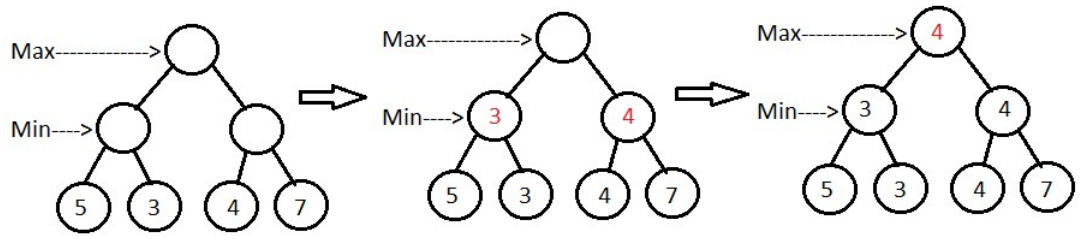


Minimizing and Maximizing value:

In order to be able to create game AI that attempts to win the game, we attach a numerical value to each possible end result. X has line of three we attach +1 where O has line of three attach -1. For the other positions use neutral value 0

MiniMax Algorithm:

Minimax is a recursive and backtracking algorithm which is used in decision making and game theory. It provides an optimal move for the player assuming that opponent is also playing optimally. The algorithm computes the minimax decision for the current state It solve the two player games



Maximizer starts with the root node and chooses the move with the maximum score. Unfortunately, only leaves have evaluation scores with them, and hence the algorithm has to reach leaf nodes recursively. In the given game tree, currently it's the minimizer's turn to **choose a move from the leaf nodes**, so the nodes with minimum scores (here, node 3 and 4) will get selected. It keeps picking the best nodes similarly, till it reaches the root node.

Now, let's formally define steps of the algorithm:

1. Construct the complete game tree
2. Evaluate scores for leaves using the evaluation function
3. Back-up scores from leaves to root, considering the player type:
    - For max player, select the child with the maximum score
    - For min player, select the child with the minimum score
4. At the root node, choose the node with max value and perform the corresponding move

Program:

```
def make_best_move():
    bestScore = -math.inf
    bestMove = None
    for move in ticTacBoard.get_possible_moves():
        ticTacBoard.make_move(move)
        score = minimax(False, aiPlayer, ticTacBoard)
        ticTacBoard.undo()
        if (score > bestScore):
            bestScore = score
            bestMove = move
    ticTacBoard.make_move(bestMove)

def minimax(isMaxTurn, maximizerMark, board):
    state = board.get_state()
    if (state is State.DRAW):
        return 0
    elif (state is State.OVER):
        return 1 if board.get_winner() is maximizerMark else -1

    scores = []
    for move in board.get_possible_moves():
        board.make_move(move)
        scores.append(minimax(not isMaxTurn, maximizerMark, board))
        board.undo()

    return max(scores) if isMaxTurn else min(scores)
```

Alpha Beta Pruning in AI:

Alpha beta pruning is an optimization technique for minimax algorithm. The word pruning means cutting down branches and leaves. Alpha-beta pruning is an optimization technique for the minimax algorithm. Alpha beta pruning is used which saves the computer from having to examine the entire tree.

Alpha: At any point along the maximizer path, alpha is the best option or the highest value. The initial value for alpha is –infinity

Beta: Beta is best choice or the lowest value that we have found at any instance along the path of minimizer the intial value for alpha is +infinity

The condition for alpha beta pruning is $\alpha >= \beta$.

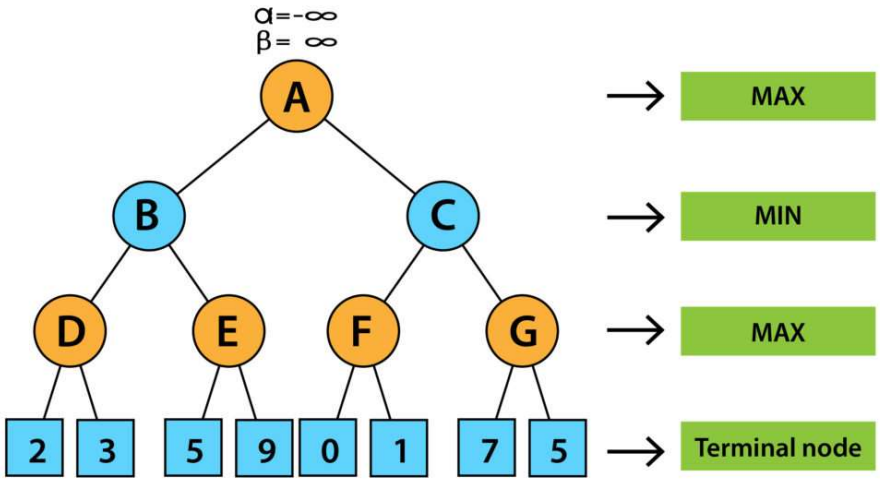
Alpha values only updates at MAX's time and beta values can only

Condition for Alpha-beta pruning

- Alpha: At any point along the Maximizer path, Alpha is the best option or the highest value we've discovered. The initial value for alpha is $-\infty$.
- Beta: At any point along the Minimizer path, Beta is the best option or the lowest value we've discovered.. The initial value for alpha is $+\infty$.
- The condition for Alpha-beta Pruning is that $\alpha >= \beta$.
- The alpha and beta values of each node must be kept track of. Alpha can only be updated when it's MAX's time, and beta can only be updated when it's MIN's turn.
- MAX will update only alpha values and the MIN player will update only beta values.
- The node values will be passed to upper nodes instead of alpha and beta values during going into the tree's reverse.
- Alpha and Beta values only are passed to child nodes.

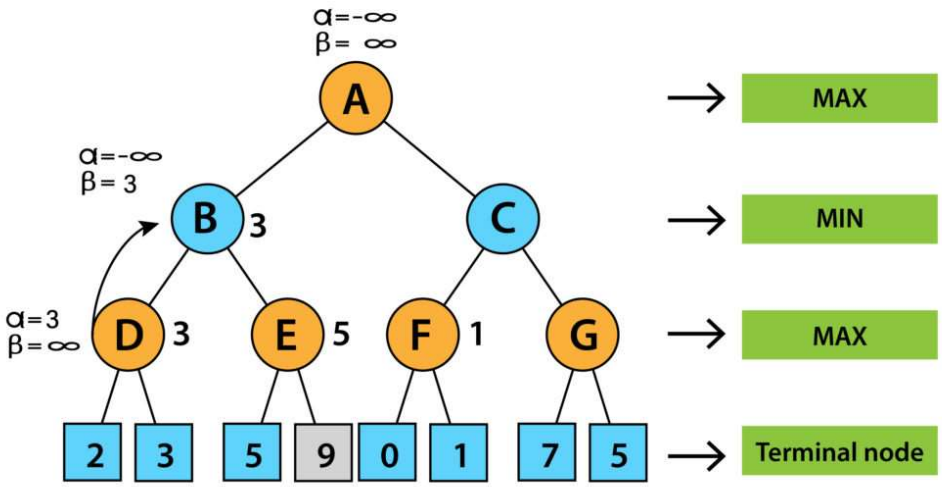
Working of Alpha-beta Pruning

1. We will first start with the initial move. We will initially define the alpha and beta values as the worst case i.e. $\alpha = -\infty$ and $\beta = +\infty$. We will prune the node only when alpha becomes greater than or equal to beta.



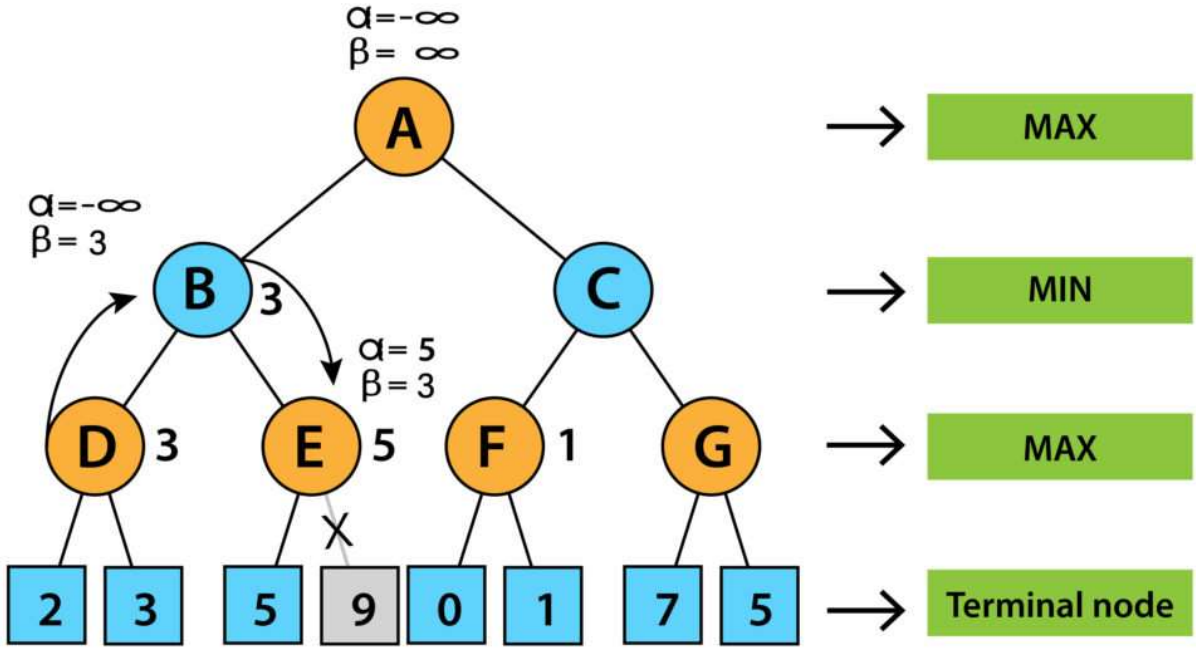
2. Since the initial value of alpha is less than beta so we didn't prune it. Now it's turn for MAX. So, at node D, value of alpha will be calculated. The value of alpha at node D will be max (2, 3). So, value of alpha at node D will be 3.

3. Now the next move will be on node B and its turn for MIN now. So, at node B, the value of alpha beta will be min (3, ∞). So, at node B values will be alpha= – ∞ and beta will be 3.



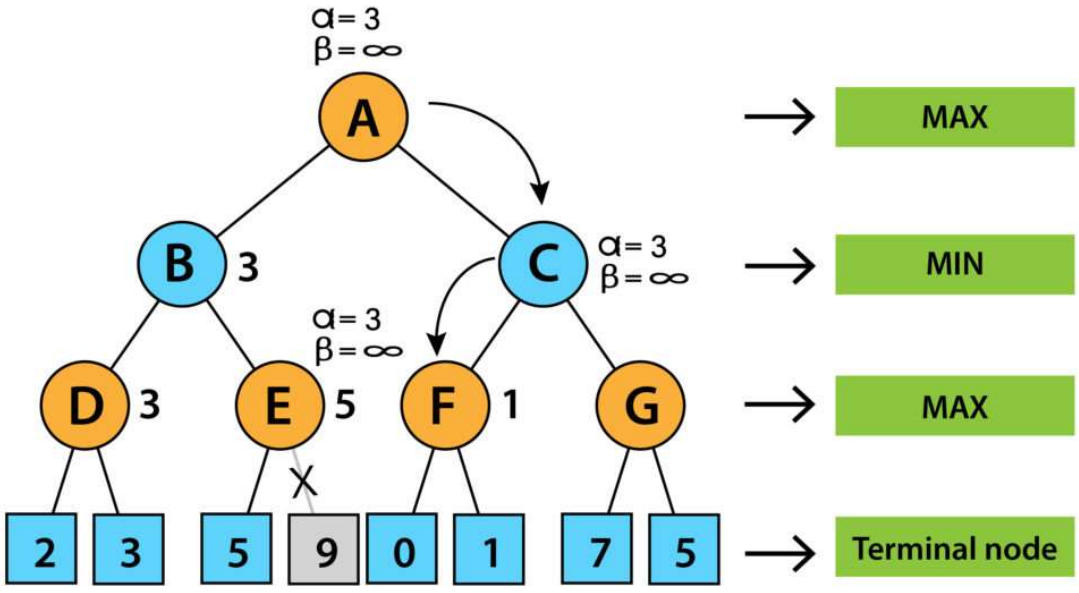
In the next step, algorithms traverse the next successor of Node B which is node E, and the values of α= -∞, and β= 3 will also be passed.

4. Now it's turn for MAX. So, at node E we will look for MAX. The current value of alpha at E is – ∞ and it will be compared with 5. So, MAX (- ∞, 5) will be 5. So, at node E, alpha = 5, Beta = 5. Now as we can see that alpha is greater than beta which is satisfying the pruning condition so we can prune the right successor of node E and algorithm will not be traversed and the value at node E will be 5.

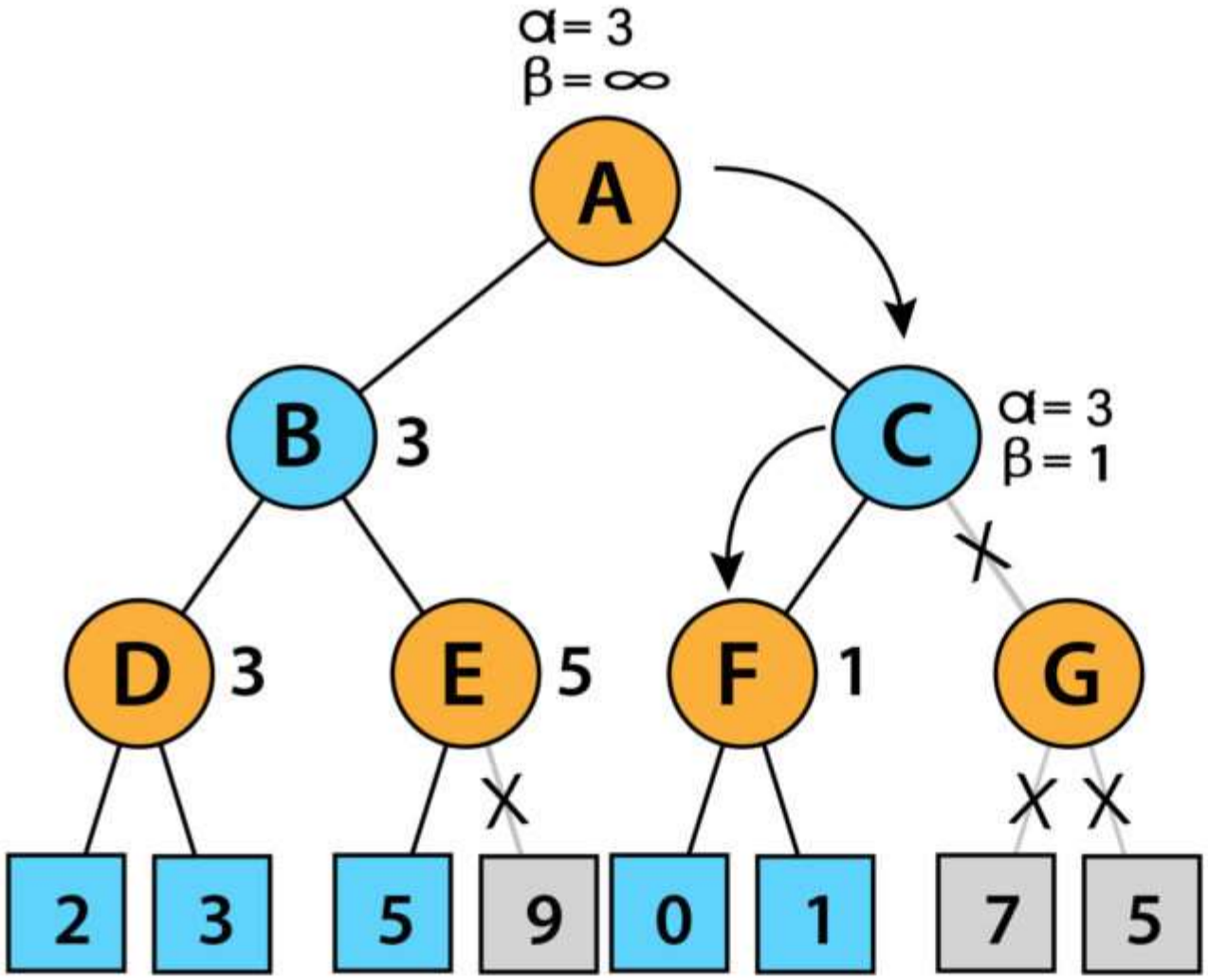


6. In the next step the algorithm again comes to node A from node B. At node A alpha will be changed to maximum value as MAX (- ∞, 3). So now the value of alpha and beta at node A will be (3, + ∞) respectively and will be transferred to node C. These same values will be transferred to node F.

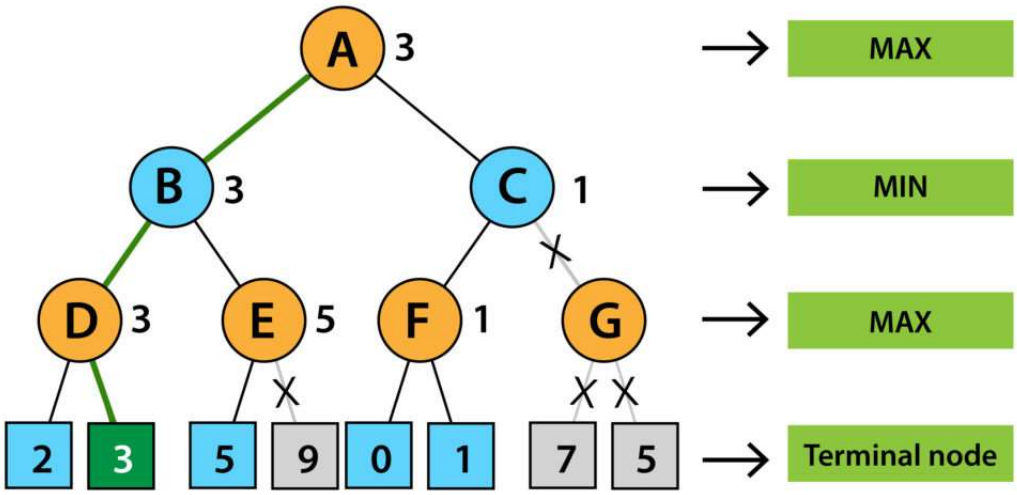
7. At node F the value of alpha will be compared to the left branch which is 0. So, MAX (0, 3) will be 3 and then compared with the right child which is 1, and MAX (3,1) = 3 still α remains 3, but the node value of F will become 1.



8. Now node F will return the node value 1 to C and will compare to beta value at C. Now its turn for MIN. So, MIN (+ ∞, 1) will be 1. Now at node C, α= 3, and β= 1 and alpha is greater than beta which again satisfies the pruning condition. So, the next successor of node C i.e. G will be pruned and the algorithm didn't compute the entire subtree G.



Now, C will return the node value to A and the best value of A will be MAX (1, 3) will be 3.



The above represented tree is the final tree which is showing the nodes which are computed and the nodes which are not computed. So, for this example the optimal value of the maximizer will be 3.