4.1. Experiment No. 1

Aim:

Implement depth first search algorithm and Breadth First Search algorithm, Use an undirected graph and develop a recursive algorithm for searching all the vertices of a graph or tree data structure

Prerequisite: Basic knowledge of Arrays, Lists, Stack, Queue, Graph, Tree etc.

Objective:

In this experiment, we will be able to do the following:

- To understand Uninformed Search Strategies and to make use of Graph and Tree Data Structure for implementation of Uninformed Search strategies.
- Study the various Graph traversal algorithms and the difference between them.Understand the BFS Graph traversal using queues.
- Demonstrate knowledge of time complexity and space complexity of performing a BFS on a given graph.

Software and Hardware Requirement: Open Source python Programming tool /java and Ubuntu.

Theory:

Uninformed (Blind search) search

- Uninformed search is a class of general-purpose search algorithms which operates in brute force-way.
- It examines each node of the tree until it achieves the goal node.
- Uninformed search algorithms do not have additional information about state or search space other than how to traverse the tree and how to identify leaf and goal nodes, so it is also called blind search
- Eg. DFS, BFS Search algorithm Graphs Definition A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links.

The interconnected objects are represented by points termed as vertices, and the links that connect the vertices are called edges.

Types of Graphs

- Undirected Graph:- Directions are not given to edges

• Directed Graph:- Directions are given to edges wit

Theory of Graph Traversal Techniques

In computer science, graph traversal (also known as graph search) refers to the process of visiting (checking and/or updating) each vertex in a graph. Such traversals are classified by the order in which the vertices are visited. Tree traversal is a special case of graph traversal. Techniques of Graph Traversal

• **DFS** - A depth-first search (DFS) is an algorithm for traversing a finite graph. DFS visits the child vertices before visiting the sibling vertices; that is, it traverses the depth of any particular path before exploring its breadth. A stack (often the program's call stack via recursion) is generally used when implementing the algorithm.

• **BFS** - A breadth-first search (BFS) is another technique for traversing a finite graph. BFS visits the neighbor vertices before visiting the child vertices, and a queue is used in the search process. This algorithm is often used to find the shortest path from one vertex to another

DFS

• 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 similar to the BFS algorithm.

• Depth First Search algorithm is a recursive algorithm that uses the idea of backtracking.

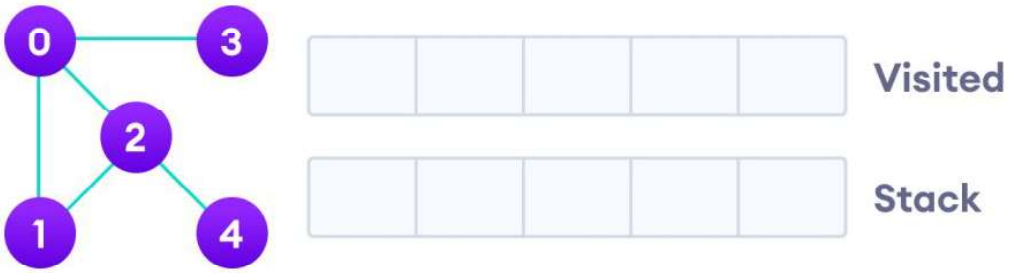
Depth First Search (DFS) Algorithm

Depth first search (DFS) algorithm starts with the initial node of the graph G, and then goes deeper and deeper until we find the goal node or the node which has no children. The algorithm then backtracks from the dead end towards the most recent node that is yet to be completely explored. The data structure which is being used in DFS is stack.

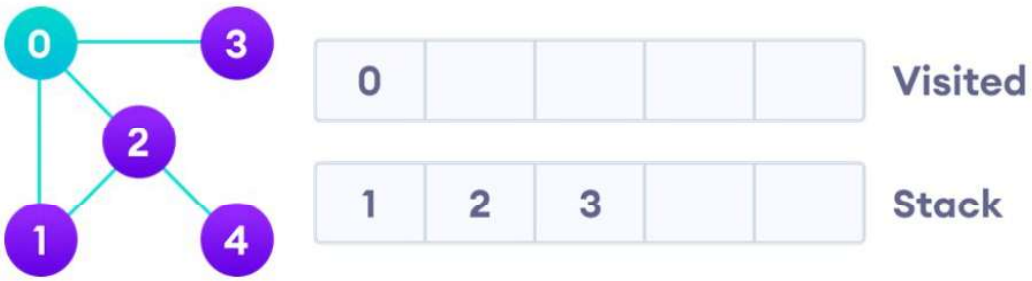
• STEP 1: Start by putting any one of the graph's vertices on top of a stack (acts as source node of DFS).

- STEP 2: Take the top item of the stack and set its visited as 1.
- STEP 3: Create a list of that vertex's adjacent nodes. Add the ones whose visited is 0 to the top of stack.
- STEP 4: Keep repeating steps 2 and 3 until the stack is empty.

An example which explains DFS Algorithm

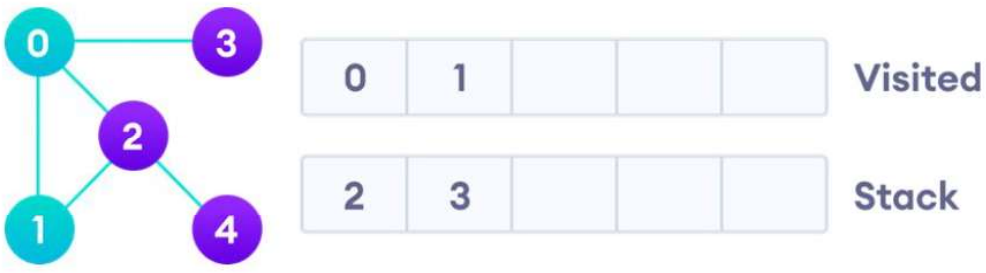


We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



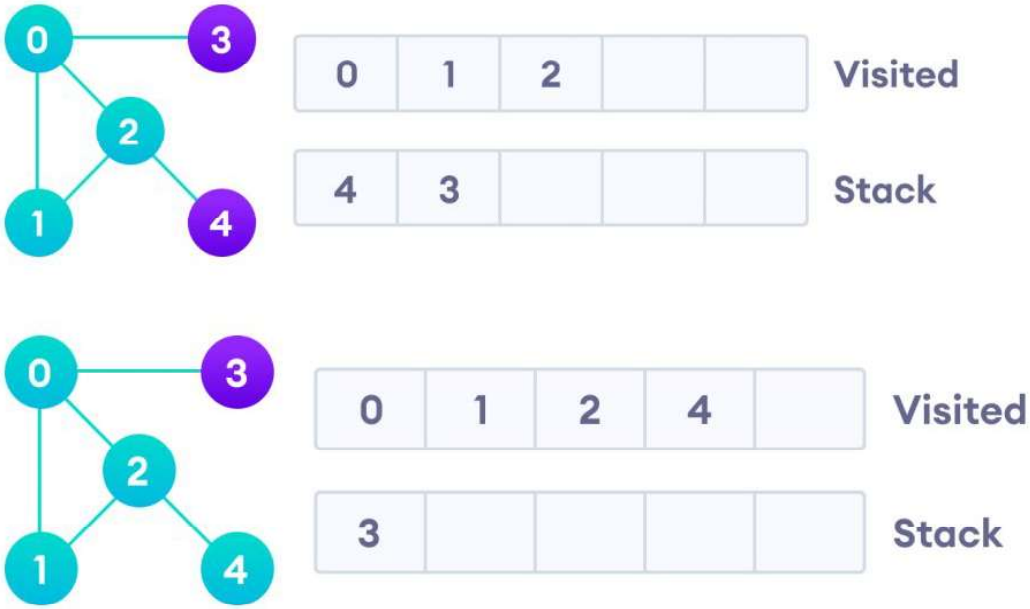
Visit the element and put it in the visited list

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.

Complexity of Depth First Search

The time complexity of the DFS algorithm is represented in the form of O(V + E),where V is the number of nodes and E is the number of edges.

The space complexity of the algorithm is O(V).

Application of DFS Algorithm

For finding the path

To test if the graph is bipartite

For finding the strongly connected components of a graph

For detecting cycles in a graph

DFS Advantage:

- 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

DFS Disadvantage:

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

BFS Queues Definition

A Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO).

- Breadth-first search is the most common search strategy for traversing a tree or graph.
- This algorithm searches breadthwise 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 nodes at the current level before moving to nodes of the 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

BFS Algorithm

The algorithm starts with examining the source node and all of its neighbors. In the next step, the neighbors of the nearest node of the source node are explored. The algorithm then explores all neighbors of all the nodes and ensures that each node is visited exactly once and no node is visited twice.

STEP 1: Set visited as 0 for all nodes in the Graph.

STEP 2: Enqueue the selected source node into the queue.

STEP 3: Dequeue a node N from the queue and update its visited as 1.

STEP 4: Enqueue all the neighbours of node N which are not present in the queue and whose visited

is 0.

STEP 5: Repeat steps 3 and 4 until the queue is empty.

STEP 6: EXIT

BFS Applications

- Path and Minimum Spanning Tree for unweighted graph
- In Peer to Peer Networks like BitTorrent, Breadth First Search is used to find all neighbor nodes.
- GPS Navigation systems Breadth First Search is used to find all neighboring locations.
- Cycle Detection in Undirected Graph In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle. In directed graphs, only depth first search can be used.
- Finding all nodes within one connected component We can either use Breadth First or Depth First Traversal to find all nodes reachable from a given node.

Differences between BFS and DFS

Key Differences Between BFS and DFS:

- BFS is a vertex-based algorithm while DFS is an edge-based algorithm.
- Queue data structure is used in BFS. On the other hand, DFS uses stack or recursion.
- Memory space is efficiently utilized in DFS while space utilization in BFS is not effective.
- BFS is an optimal algorithm while DFS is not optimal.
- DFS constructs narrow and long trees whereas BFS constructs wide and short tree.

Conclusion: In This way we have studied uninformed search strategy, how to implement depth first search algorithm and Breadth First Search algorithm, Use an undirected graph and develop a recursive algorithm for searching all the vertices of a graph or tree data structure.

Outcome: Successfully able to find depth first search algorithm and Breadth First Search algorithm.

Questions:

Q 1: Differentiate between DFS and BFS Algorithms.

Q 2: Write down the Time and Space Complexity of DFS and BFS.

Q 3: Which data structure is used by DFS and BFS?

4.2. Experiment No. 2

Aim:

Implement A* star Algorithm for any game search problem.

Objective:

In this experiment, we will be able to do the following:

- To understand informed Search Strategies.
- To make use of Graph and Tree Data Structure for implementation of informed Search strategies.
- Study the A* algorithms for various graph traversal problem.

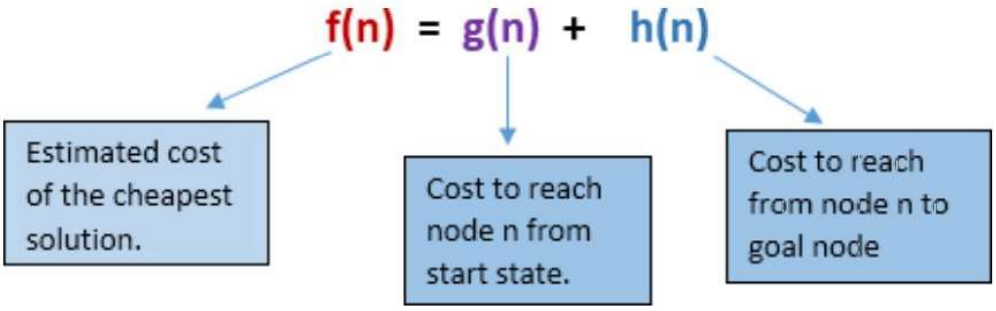
Theory:

A* Search

A* search is the most commonly known form of best-first search. Ituses 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 theheuristic 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**.



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

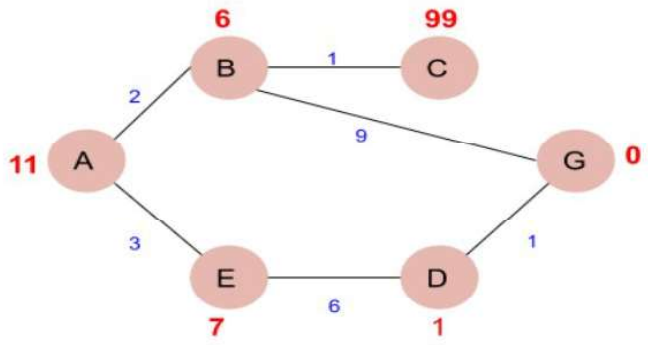
Step 4: Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.

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.

Implementation with Python

In this section, we are going to find out how the A* search algorithm can be used to find the most cost-effective path in a graph. Consider the following graph below.



The numbers written on edges represent the distance between the nodes, while the numbers written on nodes represent the heuristic values. Let us find the most cost-effective path to reach from start state A to final state G using the A* Algorithm.

Let's start with node A. Since A is a starting node, therefore, the value of g(x) for A is zero, and from the graph, we get the heuristic value of A is 11, therefore

g(x) + h(x) = f(x)

0+ 11 =11

Thus for A, we can write A=11

Now from A, we can go to point B or point E, so we compute f(x) for each of them

A → B = 2 + 6 = 8

A → E = 3 + 6 = 9

Since the cost for A → B is less, we move forward with this path and compute the f(x) for the children nodes of B

Since there is no path between C and G, the heuristic cost is set to infinity or a very high value

A → B → C = (2 + 1) + 99= 102

A → B → G = (2 + 9) + 0 = 11

Here the path A → B → G has the least cost but it is still more than the cost of A → E, thus we explore this path further

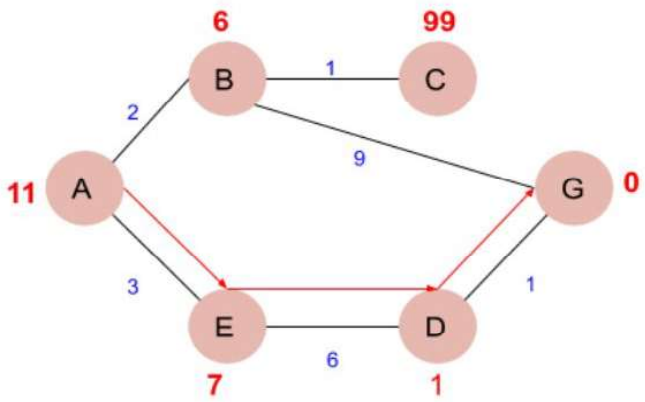
A → E → D = (3 + 6) + 1 = 10

Comparing the cost of A → E → D with all the paths we got so far and as this cost is least of all we move forward with this path. And compute the f(x) for the children of D

A → E → D → G = (3 + 6 + 1) +0 =10

Now comparing all the paths that lead us to the goal, we conclude that A → E → D

→ G is the most cost-effective path to get from A to G.



Input:

```
# heuristic distance for all nodes
def heuristic(n):
        H_dist = {
            'A': 11,
            'B': 6,
            'C': 99,
            'D': 1,
            'E': 7,
            'G': 0,

        }

        return H_dist[n]

#Describe your graph here
Graph_nodes = {
    'A': [('B', 2), ('E', 3)],
    'B': [('C', 1),('G', 9)],
    'C': None,
    'E': [('D', 6)],
    'D': [('G', 1)],

}
```

Output:

Path Found: ['A','E','D','G']

Advantages:

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

Disadvantages:

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

Applications:

A- Star algorithm is used to determine the closest path the enemy can go to the tree.

Furthermore, the player must try to protect the tree from the enemy by touching the enemy or using a barrier to keep the tree alive.

Conclusion:

Successfully able to implement graph traversal problem using A* Algorithm.

Questions:

1. How is the heuristic function used in the A* algorithm, and what is its significance?

2. What is the role of a priority queue in the A* algorithm?

3. How does A* guarantee finding the optimal solution, and under what conditions does it hold true?

4. What is an admissible heuristic, and why is it important in the context of the A* algorithm?

5. Explain the similarities and differences between A* and the Best-First Search algorithm.

4.3. Experiment No. 3

Aim:

Implement a solution for a Constraint Satisfaction Problem using Branch and Bound and Backtracking for n-queens problem or a graph coloring problem

Objective:

Implement a solution for a Constraint Satisfaction Problem using Branch and Bound and

Backtracking for n-queens problem or a graph coloring problem

Theory:

8 Queens Problem using Branch and Bound

The N-Queens problem is a puzzle of placing exactly N queens on an NxN chessboard, such that no two queens can attack each other in thatconfiguration. Thus, no two queens can lie in the same row, column ordiagonal.

The branch and bound solution is somehow different, it generates a partial solution until it figures that there's no point going deeperas we would ultimately lead to a dead end.

In the backtracking approach, we maintain an 8x8 binary matrix for keeping track of safe cells (by eliminating the unsafe cells, those that are likely to be attacked) and update it each time we place a new queen. However, it required O(n^2) time to check safe cell and update the queen.

In the 8 queens problem, we ensure the following:

1. no two queens share a row
2. no two queens share a column
3. no two queens share the same left diagonal
4. no two queens share the same right diagonal

ensure that the queens do not share the same column by theway we fill out our auxiliary matrix (column by column). Hence, onlythe left out 3 conditions are left out to be satisfied.

Applying the branch and bound approach:

The branch and bound approach suggest that we create a partial solution and use it to ascertain whether we need to continue in a particular direction or not. For this problem, we create 3 arrays to checkfor conditions 1,3 and 4. The Boolean arrays tell which rows and diagonals are already occupied. To achieve this, we need a numbering system to specify which queen is placed.

The indexes on these arrays would help us know which queen we are analysing.

Preprocessing - create two NxN matrices, one for top-left to bottom- right diagonal, and other for top-right to bottom-left diagonal. We need to fill these in such a way that two queens sharing same top- left bottom-right diagonal will have same value in slash Diagonal and two queens sharing same top-right bottom-left diagonal will have same value in back Slash Diagonal.

slashDiagnol(row)(col) = row + col

backSlashDiagnol(row)(col) = row - col

+ (N-1) { N = 8 }

{ we added (N-1) as we do not need negative values in backSlashDiagnol }

For placing a queen *i* on row *j*, check the following :

1. whether row 'j' is used or not
2. whether slashDiagnol 'i+j' is used or not
3. whether backSlashDiagnol 'i-j+7' is used or not

If the answer to any one of the following is true, we try another location for queen **i** on row **j**, mark the row and diagnols; and recur for queen **i+1**.

Applications:

Constraint satisfaction problem in AI has a wide range of applications, including scheduling, resource allocation, and automated reasoning.

Branch and bound is an algorithm used to solve combinatorial optimization problems

Input: n=8

Output- for (n = 8)

1	**0**	**0**	**0**	**0**	**0**	**0**	**0**
0	**0**	**0**	**0**	**0**	**0**	**1**	**0**
0	**0**	**0**	**0**	**1**	**0**	**0**	**0**
0	**0**	**0**	**0**	**0**	**0**	**0**	**1**
0	**1**	**0**	**0**	**0**	**0**	**0**	**0**
0	**0**	**0**	**1**	**0**	**0**	**0**	**0**
0	**0**	**0**	**0**	**0**	**1**	**0**	**0**
0	**0**	**1**	**0**	**0**	**0**	**0**	**0**

Graph coloring problem's solution using backtracking algorithmGraph coloring

The **graph coloring problem** is to discover whether the nodes of the graph G can be covered in such a way, that no two adjacent nodes have the same color yet only m colors are used. This graph coloring problem is also known as M- colorability decision problem.

The M – colorability optimization problem deals with the smallest integer m for which the graph G can be colored. The integer is known as a chromatic number of the graph.

Here, it can also be noticed that if d is the degree of the given graph, then it can be colored with d+ 1 color.

A graph is also known to be planar if and only if it can be drawn in a planar in such a way that no two edges cross each other. A special case is the 4 - colors problem for planar graphs. The problem is to color the region in a map in such away that no two adjacent regions have the same color. Yet only four colors are needed. This is a problem for which graphs are very useful because a map can beeasily transformed into a graph. Each region of the map becomes the node, and iftwo regions are adjacent, they are joined by an edge.

Graph coloring problem can also be solved using a state space tree, whereby applying a backtracking method required results are obtained.

For solving the **graph coloring problem**, we suppose that the graph

7	6	5	4	3	2	1	0
8	7	6	5	4	3	2	1
9	8	7	6	5	4	3	2
10	9	8	7	6	5	4	3
11	10	9	8	7	6	5	4
12	11	10	9	8	7	6	5
13	12	11	10	9	8	7	6
14	13	12	11	10	9	8	7

slash diagnol[row][col] = row + col

0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10
4	5	6	7	8	9	10	11
5	6	7	8	9	10	11	12
6	7	8	9	10	11	12	13
7	8	9	10	11	12	13	14

backslash diagnol[row][col] = row-col+(N-1)

is represented by its adjacency matrix G[1:n, 1:n], where, G[i, j]= 1 if (i, j) is an edge of G, and G[i, j] = 0 otherwise.

The colors are represented by the integers 1, 2, ..., m and the solutions are given by the n-tuple (x1, x2, x3, ..., xn), where x1 is the color of node i.

Algorithm for finding the m - colorings of graph

```
{
Repeat
{
// generate all legal assignments
for x[k], Next value (k); // assign
to x[k] a legal color.
If ( x[k] = 0 ) then return; // no new color
possible
If (k = n) then // at most m colors have been
used to color the n
vertices.
 Write (x[1 : n ]);
Else mcoloring (k
+ 1);
}
Until (false);
}
```

Conclusion:

Thus we have implemented pre-processing of a text document such as stop word removal, stemming.

Outcome:

Upon completion of this experiment, students will be able to: Implement a solution for a Constraint Satisfaction Problem using Branch and Bound and Backtracking for n-queens problem or a graph coloring problem

Questions:

Q.1) Which are the constraints required to solve N Queen problem?

Q.2) Compare backtracking and branch and bound method

Q.3) What do mean by constraints satisfaction problem.

4.4. Experiment No. 4

Aim:

Implement Alpha-beta Tree search for any game search problem

Objective:

Students will be able to understand how Alpha-beta Tree search is used in any game search problem

Theory:

- 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:
    a. **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 $-\infty$.
    b. **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 $+\infty$.
- 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.
- The main condition which required for alpha-beta pruning is:

  $\alpha >= \beta$

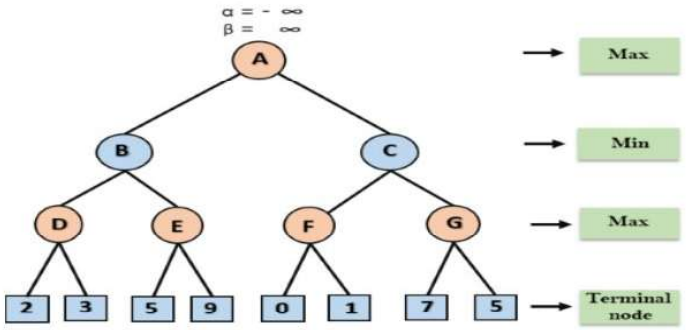
- The Max player will only update the value of alpha.
- The Min player will only update the value of beta.

- 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.

Working of Alpha-Beta Pruning:

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

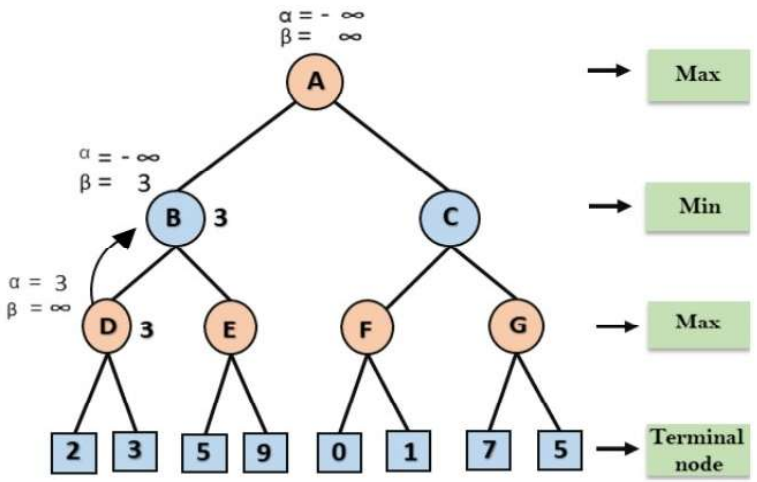
Step 1: At the first step the, Max player will start first move from node A where α= -∞ and β= +∞, these value of alpha and beta passed down to node B where again α= -∞ and β= +∞, and Node B passes the same value to its child D.



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.

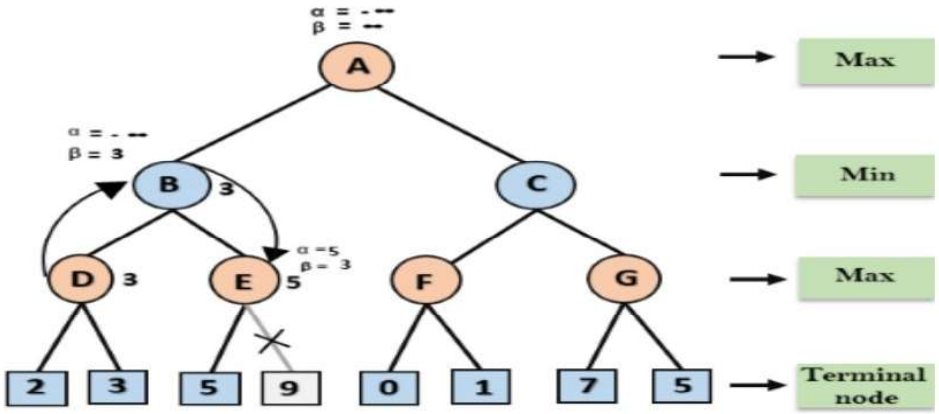
Step 3: Now algorithm backtrack to node B, where the value of β will change as this is a turn of Min, Now β= +∞, will compare with the available subsequent

nodes value, i.e. min (∞, 3) = 3, hence at node B now α= -∞, and β= 3.



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

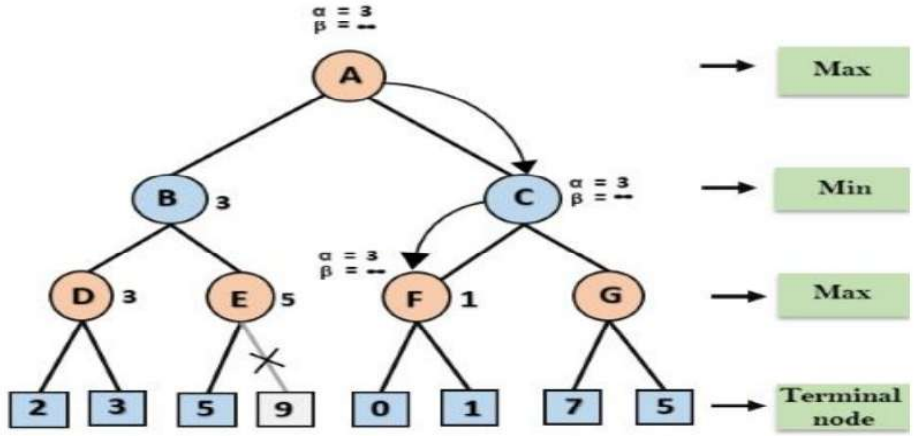
Step 4: At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so max (-∞, 5) = 5, hence at node E α= 5 and β= 3, where α>=β, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will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



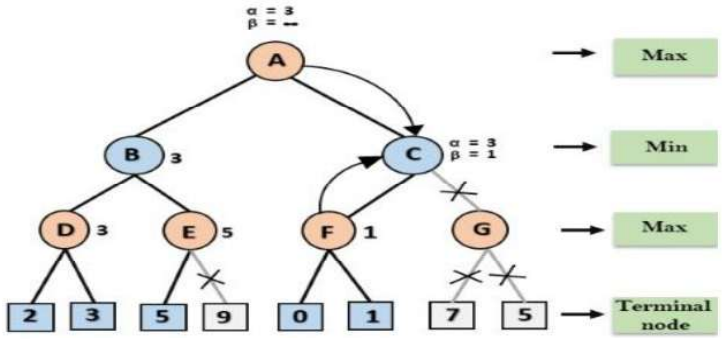
Step 5: At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C.

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.

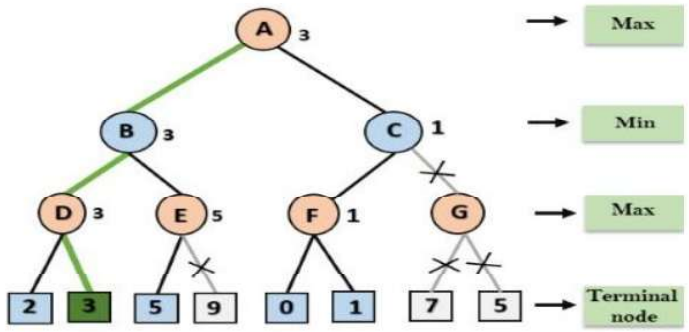


Step 7: Node F returns the node value 1 to node C, at C α= 3 and β= +∞, here the value of beta will be changed, it will compare with 1 so min (∞, 1) = 1. Now at C, α=3 and β= 1, and again it satisfies the condition α>=β, so the next child of C which is G will be pruned, and the algorithm will not compute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



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 $O(b^{m/2})$.

Rules to find good ordering:

Following are some rules to find good ordering in alpha-beta pruning:

- Occur the best move from the shallowest node.
- Order the nodes in the tree such that the best nodes are checked first.
- Use domain knowledge while finding the best move. Ex: for Chess, try order: captures first, then threats, then forward moves, backward moves.
- We can bookkeep the states, as there is a possibility that states may repeat.

Algorithm:

1. First, generate the entire game tree starting with the current position of the game all the way up to the terminal states.
2. Apply the utility function to get the utility values for all the terminal states.
3. Determine the utilities of the higher nodes with the help of the utilities of the terminal nodes. For instance, in the diagram below, we have the utilities for the terminal states written in the squares.
4. Calculate the utility values with the help of leaves considering one layer at a time until the root of the tree.
5. Eventually, all the backed-up values reach to the root of the tree, i.e., the topmost point. At that point, MAX has to choose the highest value.

Applications:

Alpha Beta Pruning is an optimization technique of the Minimax algorithm. This algorithm solves the limitation of exponential time and space complexity in the case of the Minimax algorithm by pruning redundant branches of a game tree using its parameters Alpha(α) and Beta(β).

Input: `{ 3, 5, 6, 9, 1, 2, 0, -1 }`

```
Output: The optimal value is : 5
```

Conclusion:

Thus we have implemented pre-processing of a text document such as stop word removal, stemming.

Outcome:

Upon completion of this experiment, students will be able to:

Experiment level outcome (SL-1): Implementation of alpha beta tree search for any game search problem.

Questions:

1. What is the Minimax algorithm?

2. Why is the time complexity of the Minimax algorithm so high?

3. What are the best case and worst case time complexity of alpha-beta pruning?

4. Can alpha-beta pruning be used for applications apart from games?

5.1.Experiment No. 6

Aim: **Implement Greedy search algorithm for any ofthe following application:**
Prim's Algorithm | Minimum Spanning Tree (Python Code)

Objective:

Theory: **What is a Minimum Spanning Tree?**

As we all know, the graph which does not have edges pointing to any direction in a graph is called an undirected graph and the graph always has a path from a vertex to any other vertex. A spanning tree is a subgraph of the undirected connected graph where it includes all the nodes of the graph with the minimum possible number of edges. Remember, the subgraph should contain each and every node of the original graph. If any node is missed out then it is not a spanning tree and also, the spanning tree doesn't contain cycles. If the graph has n number of nodes, then the total number of spanning trees created from a complete graph is equal to n^(n-2). In a spanning tree, the edges may or may not have weights associated with them. Therefore, the spanning tree in which the sum of edges is minimum as possible then that spanning tree is called the minimum spanning tree. One graph can have multiple spanning-tree but it can have only one unique minimum spanning tree. There are two different ways to find out the minimum spanning tree from the complete graph i.e Kruskal's algorithm and Prim's algorithm. Let us study prim's algorithm in detail below:

What is Prim's Algorithm?

Prim's algorithm is a minimum spanning tree algorithm which helps to find out the edges of the graph to form the tree including every node with the minimum sum of weights to form the minimum spanning tree. Prim's algorithm starts with the single source node and later explore all the adjacent nodes of the source node with all the connecting edges. While we are exploring the graphs, we will choose the edges with the minimum weight and those which cannot cause the cycles in the graph.

Prim's Algorithm for Minimum Spanning Tree

Prim's algorithm basically follows the greedy algorithm approach to find the optimal solution. To find the minimum spanning tree using prim's algorithm, we will choose a source node and keep adding the edges with the lowest weight.

The algorithm is as given below:

- Initialize the algorithm by choosing the source vertex
- Find the minimum weight edge connected to the source node and another node and add it to the tree
- Keep repeating this process until we find the minimum spanning tree

Step 1: Determine an arbitrary vertex as the starting vertex of the MST.
Step 2: Follow steps 3 to 5 till there are vertices that are not included in the MST (known as fringe vertex).
Step 3: Find edges connecting any tree vertex with the fringe vertices.
Step 4: Find the minimum among these edges.
Step 5: Add the chosen edge to the MST if it does not form any cycle.
Step 6: Return the MST and exit

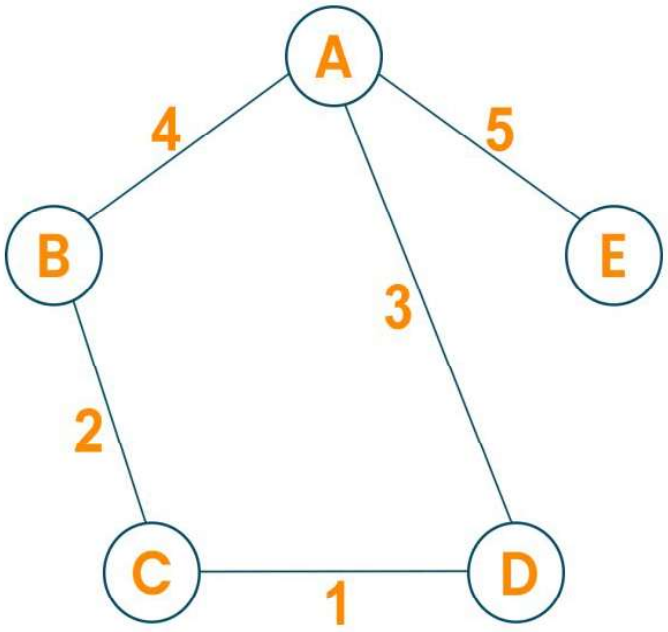
Pseudocode

```
T = Ø;

M = { 1 };

while (M ≠ N)

  let (m, n) be the lowest cost edge such that m
  ∈ M and n ∈ N - M;T = T U {(m, n)}

  M = M U {n}
```

Here we create two sets of nodes i.e M and M-N. M set contains the list of nodes that have been visited and the M-N set contains the nodes that haven't been visited. Later, we will move each node from M to M-N after each step by connecting the least weight edge.

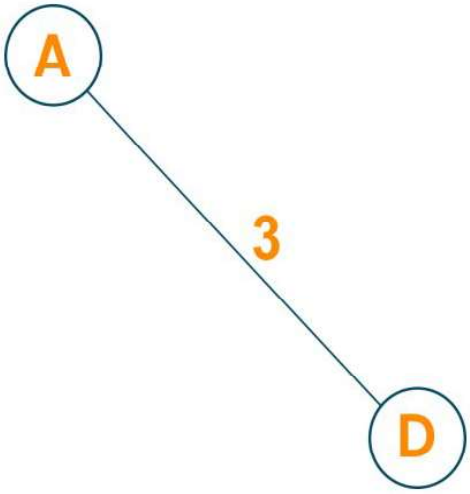
Example

Let us consider the below-weighted graph

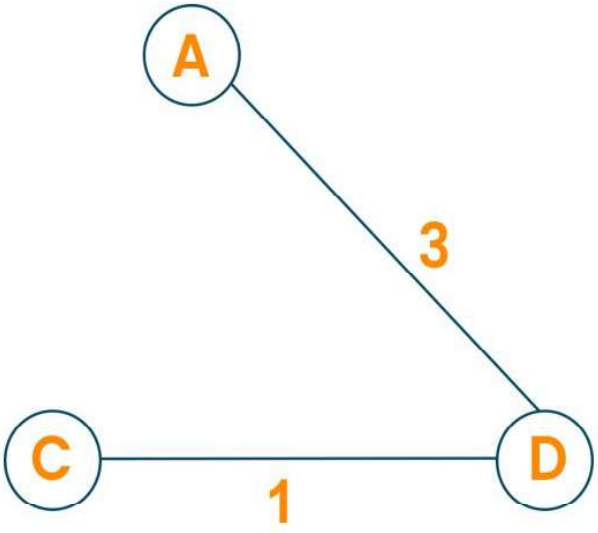


Later we will consider the source vertex to initialize the algorithm

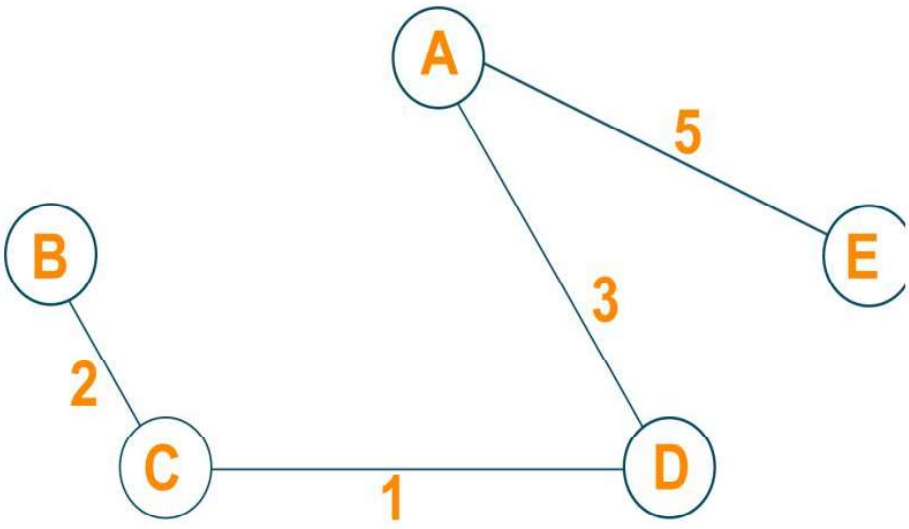
Now, we will choose the shortest weight edge from the source vertex and add it to finding thespanning tree.



Then, choose the next nearest node connected with the minimum edge and add it to thesolution. If there are multiple choices then choose anyone.



Continue the steps until all nodes are included and we find the minimum spanning tree.



Time Complexity:

The running time for prim's algorithm is O(VlogV + ElogV) which is equal to O(ElogV) because every insertion of a node in the solution takes logarithmic time. Here, E is the numberof edges and V is the number of vertices/nodes. However, we can improve the running time complexity to O(E + logV) of prim's algorithm using Fibonacci Heaps.

```
# Prim's Algorithm in Python

INF = 9999999
# number of vertices in graph
N = 5
#creating graph by adjacency matrix method
G = [[0, 19, 5, 0, 0],
     [19, 0, 5, 9, 2],
     [5, 5, 0, 1, 6],
     [0, 9, 1, 0, 1],
     [0, 2, 6, 1, 0]]

selected_node = [0, 0, 0, 0, 0]

no_edge = 0

selected_node[0] = True

# printing for edge and weight
print("Edge : Weight\n")
while (no_edge < N - 1):
```

```
    minimum = INF
    a = 0
    b = 0
    for m in range(N):
        if selected_node[m]:
            for n in range(N):
                if ((not selected_node[n]) and G[m][n]):
                    # not in selected and there is an edge
                    if minimum > G[m][n]:
                        minimum = G[m][n]
                        a = m
                        b = n
    print(str(a) + "-" + str(b) + ":" + str(G[a][b]))
    selected_node[b] = True
    no_edge += 1
```

Applications:

- Prim's algorithm is used in network design
- It is used in network cycles and rail tracks connecting all the cities
- Prim's algorithm is used in laying cables of electrical wiring
- Prim's algorithm is used in irrigation channels and placing microwave towers
- It is used in cluster analysis
- Prim's algorithm is used in gaming development and cognitive science
- Pathfinding algorithms in artificial intelligence and traveling salesman problems make use of prim's algorithm.

Input:

```
g.graph = [[0, 2, 0, 6, 0],
           [2, 0, 3, 8, 5],
           [0, 3, 0, 0, 7],
           [6, 8, 0, 0, 9],
           [0, 5, 7, 9, 0]]
Or
G = [[0, 19, 5, 0, 0],
     [19, 0, 5, 9, 2],
```

```
    [5, 5, 0, 1, 6],
    [0, 9, 1, 0, 1],
    [0, 2, 6, 1, 0]]
```

Output:

```
Edge     Weight
0 - 1     2
1 - 2     3
0 - 3     6
1 - 4     5
Or
Edge : Weight

0-2:5
2-3:1
3-4:1
4-1:2
```

Conclusion: Successfully able to implement Greedy search Algorithm for Prims Minimum Spanning tree.

Outcome: The minimum spanning tree has its own importance to learn the prim's algorithm which leads to find the solution to many problems. When it comes to finding the minimum spanning tree for the dense graphs, prim's algorithm isthe first choice.

Questions:

1. How does Prim's Algorithm Work?
2. What Is a Minimum Spanning Tree?
3. What Is Prim's Minimum Spanning Tree Algorithm?
4. How to Implement the Prim's Algorithm
5. Does Prim's algorithm run on a negative edge weight graph?
6. Can there be multiple MST for a graph?