

A LABORATORY MANUAL FOR
Artificial Intelligence
[310253]



T.E. Computer Engineering(Course-2019)
AS PER THE CURRICULUM OF
SAVITRIBAI PHULE PUNE UNIVERSITY

Name of Faculty: Prof. Afeefa Rafeeqe (Asst. Professor)

Name: Minhaj Ahmed Ansari Faheem Ahmed

Roll No: 29

Exam Seat No: T191164230

Academic Year: 2022-23

VISION

To build a strong research and learning environment producing globally competent professionals and innovators who will contribute to the betterment of the society

MISSION

- To create and sustain an academic environment conducive to the highest level of research and teaching.
- To provide state-of-the-art laboratories which will be up to date with the new developments in the area of computer engineering?
- To organize competitive event, industry interactions and global collaborations in view of providing a nurturing environment for students to prepare for a successful career and the ability to tackle lifelong challenges in global industrial needs.
- To educate students to be socially and ethically responsible citizens in view of national and global development.

CERTIFICATE

*This is to certify that Mr./ Miss Minhaj Ahmed Ansari Faheem Ahmed,
Of Class TE Computer Seat No.T191164230 Has completed all the
practical work in the subject Artificial Intelligence Lab satisfactorily in
the Department of Computer Engineering as prescribed by Savitribai
Phule Pune University, in the academic year **2022 – 23**.*

Staff In-charge

Head of the Department

Principal

INDEX

Sr. No.	Name Of the Experiment	Date Of Start	Date Of Completion	Sign
1	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.			
2	Implement A star Algorithm for any game search problem.			
3	Implement Greedy search algorithm for Prim's minimal spanning tree.			
4	Implement a solution for a Constraint Satisfaction Problem using Branch and Bound and Backtracking for n-queens problem or a graph coloring problem.			
5	Develop an elementary chatbot for any suitable customer interaction application.			

Experiment No:01

Aim: Implement Depth First Search algorithm and breadth first search algorithm.

A) DFS

Theory:

Depth First Traversal (or Search) for a graph is similar to Depth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles (a node may be visited twice). To avoid processing a node more than once, use a Boolean visited array.

Approach:

Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. So, the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them. Finally, print the nodes in the path.

Algorithm:

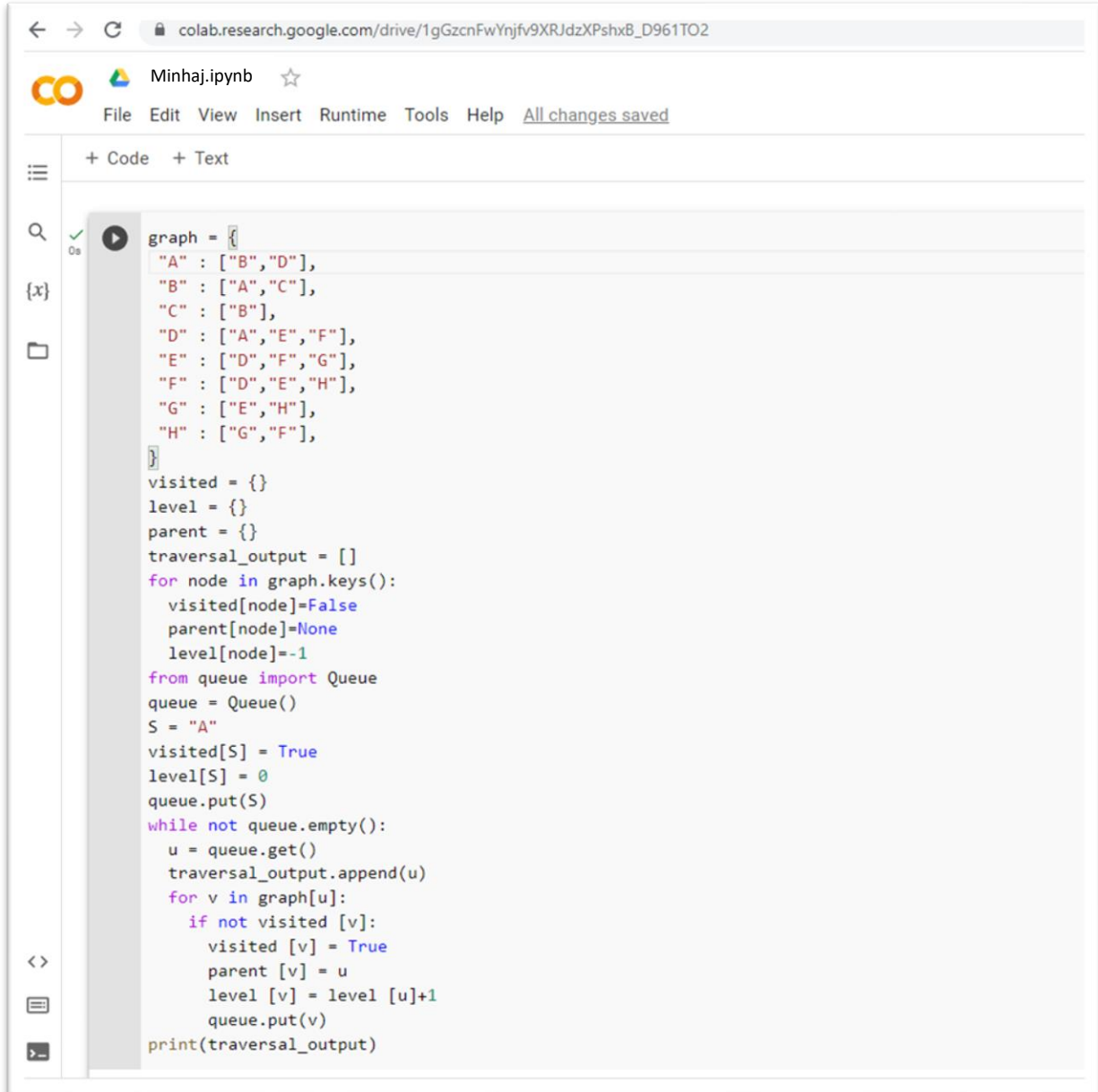
Create a recursive function that takes the index of the node and a visited array.

1. Mark the current node as visited and print the node.
2. Traverse all the adjacent and unmarked nodes and call the recursive function with the index of the adjacent node.

Performance Comparison for DFS:

- Completeness: No, due to possibility of dead end
- Optimality: No, provided path cost is non- decreasing
- Time complexity: $O(b^m)$ m is the maximum depth of any node
- Space complexity: $O(b*m)$

Code:



```
graph = {
    "A" : ["B", "D"],
    "B" : ["A", "C"],
    "C" : ["B"],
    "D" : ["A", "E", "F"],
    "E" : ["D", "F", "G"],
    "F" : ["D", "E", "H"],
    "G" : ["E", "H"],
    "H" : ["G", "F"],
}

visited = {}
level = {}
parent = {}
traversal_output = []
for node in graph.keys():
    visited[node]=False
    parent[node]=None
    level[node]=-1
from queue import Queue
queue = Queue()
S = "A"
visited[S] = True
level[S] = 0
queue.put(S)
while not queue.empty():
    u = queue.get()
    traversal_output.append(u)
    for v in graph[u]:
        if not visited [v]:
            visited [v] = True
            parent [v] = u
            level [v] = level [u]+1
            queue.put(v)
print(traversal_output)
```

Output:

['A', 'B', 'D', 'C', 'E', 'F', 'G', 'H']

B) BFS

Theory:

Breadth-First Traversal (or Search) for a graph is similar to Breadth-First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a Boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.

Algorithm:

The steps involved in the BFS algorithm to explore a graph are given as follows ;

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until QUEUE is empty

Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).

Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2

(Waiting state)

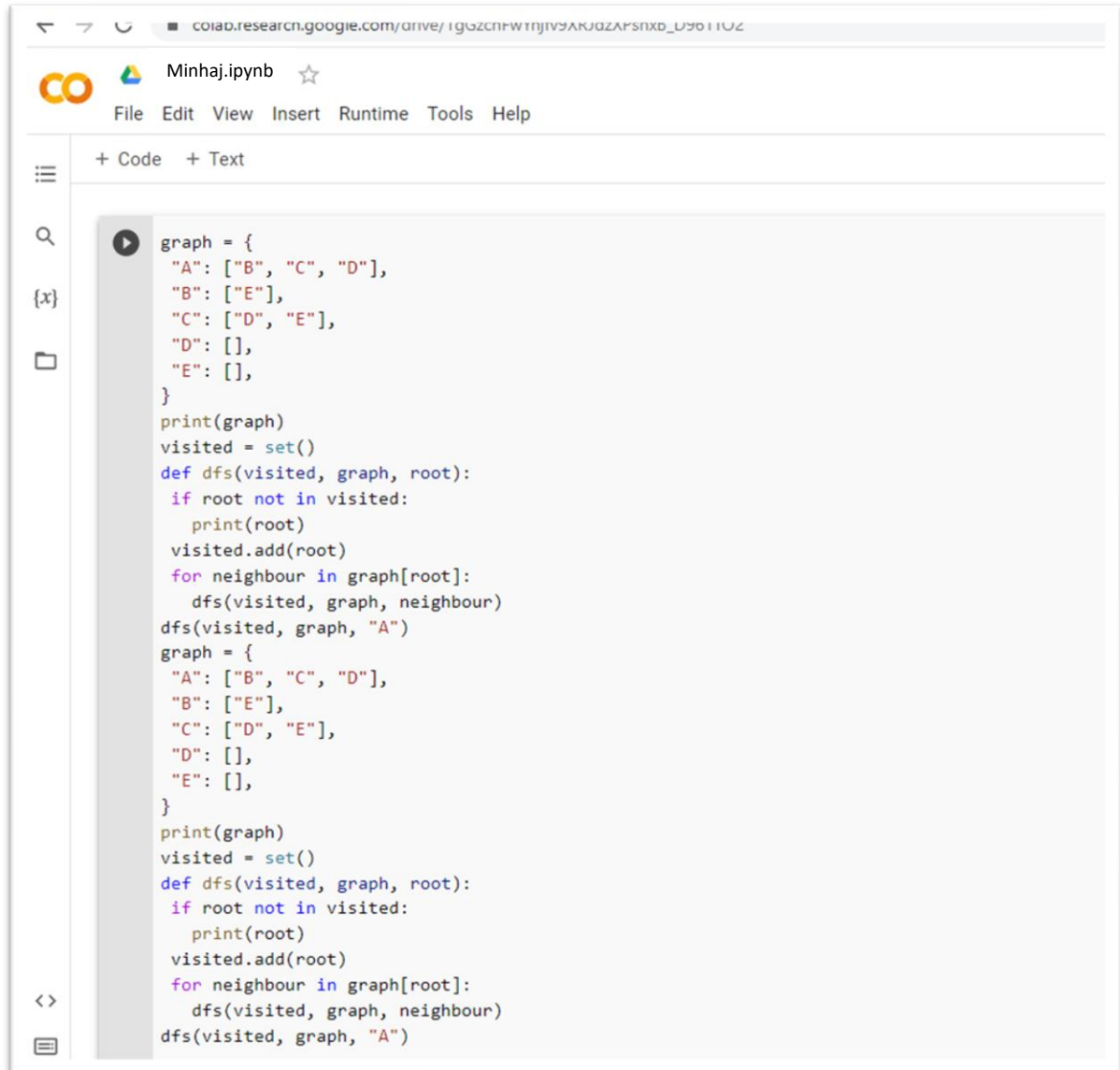
[END OF LOOP]

Step 6: EXIT

Performance Comparison for BFS:

- Completeness: yes, it gives shallowest goal
- Optimality: yes, provided path cost is non- decreasing
- Time complexity: $O(b^d + 1)$
- Space complexity: $O(b^d + 1)$

Code:



```
graph = {
    "A": ["B", "C", "D"],
    "B": ["E"],
    "C": ["D", "E"],
    "D": [],
    "E": []
}
print(graph)
visited = set()
def dfs(visited, graph, root):
    if root not in visited:
        print(root)
        visited.add(root)
        for neighbour in graph[root]:
            dfs(visited, graph, neighbour)
dfs(visited, graph, "A")
graph = {
    "A": ["B", "C", "D"],
    "B": ["E"],
    "C": ["D", "E"],
    "D": [],
    "E": []
}
print(graph)
visited = set()
def dfs(visited, graph, root):
    if root not in visited:
        print(root)
        visited.add(root)
        for neighbour in graph[root]:
            dfs(visited, graph, neighbour)
dfs(visited, graph, "A")
```

Output:

```
['A', 'B', 'D', 'C', 'E', 'F', 'G', 'H']
```

Conclusion: We have learned about DFS algorithm and BFS algorithm along with its implementation using Python programming language.

Experiment No:02

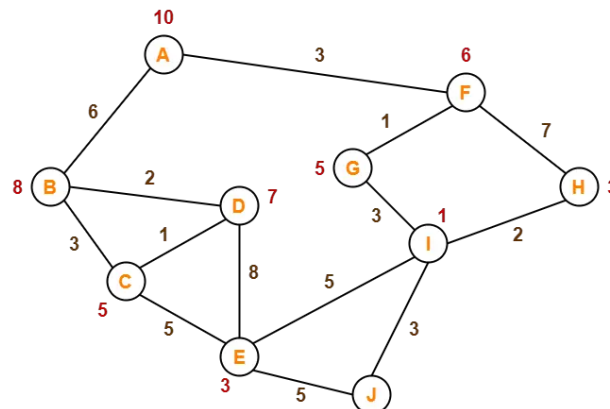
Aim: A* algorithm Implementation.

Theory:

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.

A major drawback of the A* 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.

Consider the following graph;



The numbers written on edges represent the distance between the nodes.

The numbers written on nodes represent the heuristic value.

Step-01:

We start with node A.

Node B and Node F can be reached from node A.

A* Algorithm calculates $f(B)$ and $f(F)$.

$$f(B) = 6 + 8 = 14$$

$$f(F) = 3 + 6 = 9$$

Since $f(F) < f(B)$, so it decides to go to node F.

Path- $A \rightarrow F$

Step-02:

Node G and Node H can be reached from node F.

A* Algorithm calculates $f(G)$ and $f(H)$.

$$f(G) = (3+1) + 5 = 9$$

$$f(H) = (3+7) + 3 = 13$$

Since $f(G) < f(H)$, so it decides to go to node G.

Path- $A \rightarrow F \rightarrow G$

Step-03:

Node I can be reached from node G.

A* Algorithm calculates $f(I)$.

$$f(I) = (3+1+3) + 1 = 8$$

It decides to go to node I.

Path- $A \rightarrow F \rightarrow G \rightarrow I$

Step-04:

Node E, Node H and Node J can be reached from node I.

A* Algorithm calculates $f(E)$, $f(H)$ and $f(J)$.

$$f(E) = (3+1+3+5) + 3 = 15$$

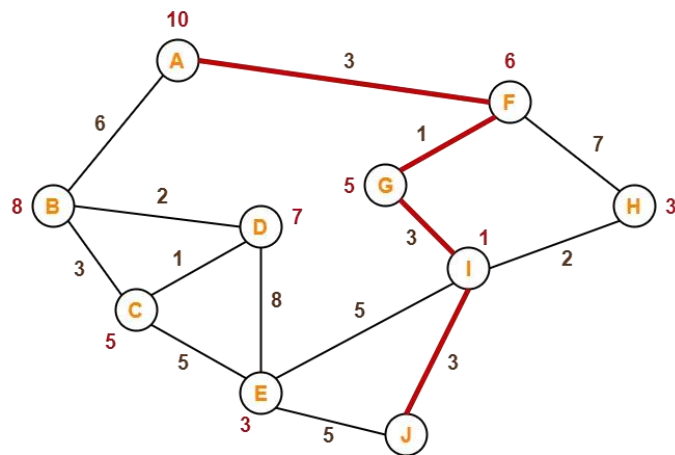
$$f(H) = (3+1+3+2) + 3 = 12$$

$$f(J) = (3+1+3+3) + 0 = 10$$

Since $f(J)$ is least, so it decides to go to node J.

Path- $A \rightarrow F \rightarrow G \rightarrow I \rightarrow J$

This is the required shortest path from node A to node J.



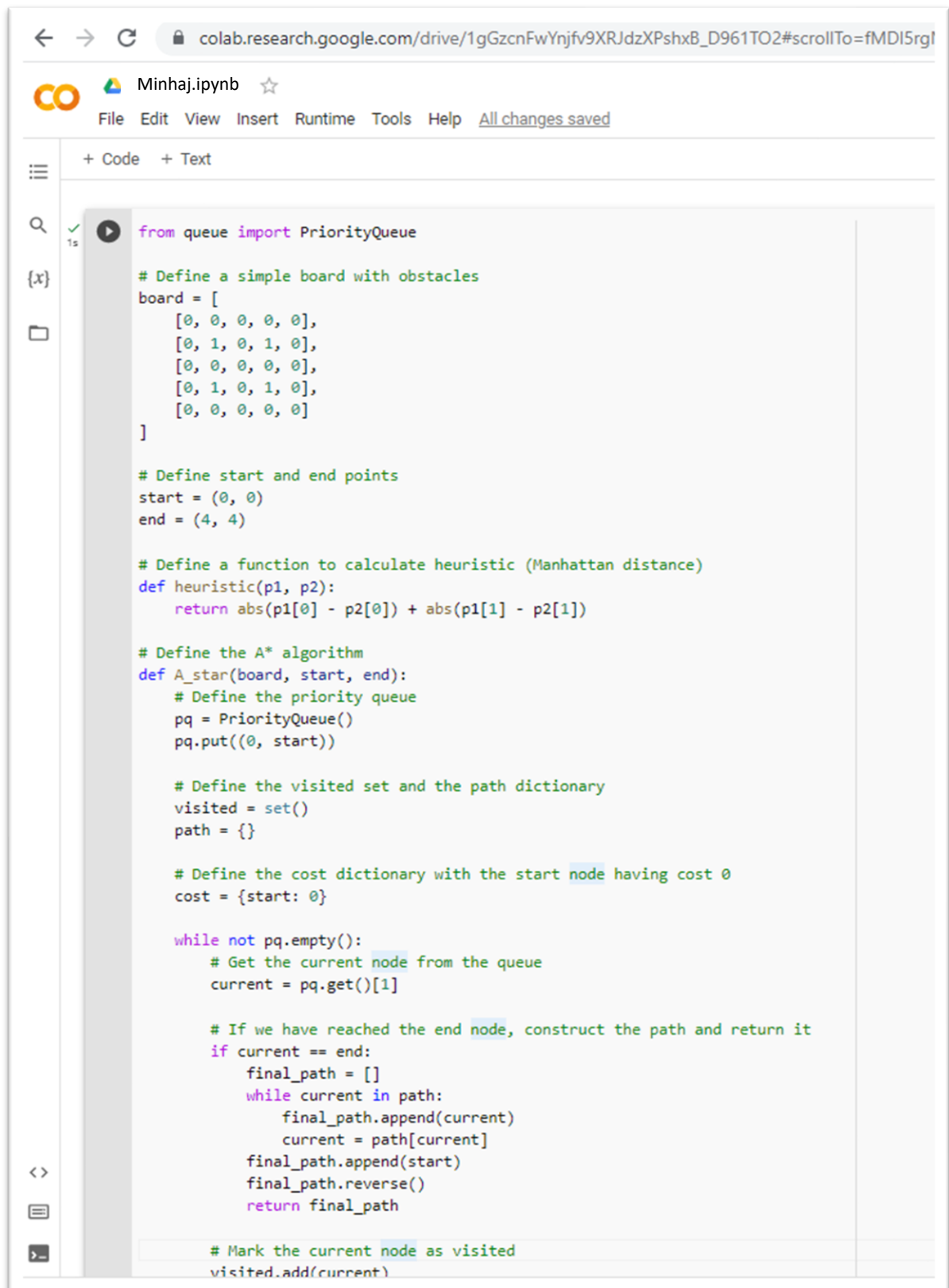
Applications of A* algorithm

- It is commonly used in web-based maps and games to find the shortest path at the highest possible efficiency.
- A* is used in many artificial intelligence applications, such as search engines.
- It is used in other algorithms such as the Bellman-Ford algorithm to solve the shortest path problem.
- The A* algorithm is used in network routing protocols, such as RIP, OSPF, and BGP, to calculate the best route between two nodes.

Algorithm:

- 1: Firstly, Place the starting node into OPEN and find its $f(n)$ value.
- 2: Then remove the node from OPEN, having the smallest $f(n)$ value. If it is a goal node, then stop and return to success.
- 3: Else remove the node from OPEN, and find all its successors.
- 4: Find the $f(n)$ value of all the successors, place them into OPEN, and place the removed node into CLOSE.
- 5: Goto Step-2.
- 6: Exit.

Code:



The image shows a Google Colab notebook interface. At the top, the URL is `colab.research.google.com/drive/1gGzcnFwYnjfv9XRJdzXPshxB_D961TO2#scrollTo=fMDI5rgI`. The notebook is titled "Minhaj.ipynb". Below the title bar, there are tabs for "File", "Edit", "View", "Insert", "Runtime", "Tools", "Help", and "All changes saved". The main area of the notebook contains a single code cell with the following Python code:

```
from queue import PriorityQueue

# Define a simple board with obstacles
board = [
    [0, 0, 0, 0, 0],
    [0, 1, 0, 1, 0],
    [0, 0, 0, 0, 0],
    [0, 1, 0, 1, 0],
    [0, 0, 0, 0, 0]
]

# Define start and end points
start = (0, 0)
end = (4, 4)

# Define a function to calculate heuristic (Manhattan distance)
def heuristic(p1, p2):
    return abs(p1[0] - p2[0]) + abs(p1[1] - p2[1])

# Define the A* algorithm
def A_star(board, start, end):
    # Define the priority queue
    pq = PriorityQueue()
    pq.put((0, start))

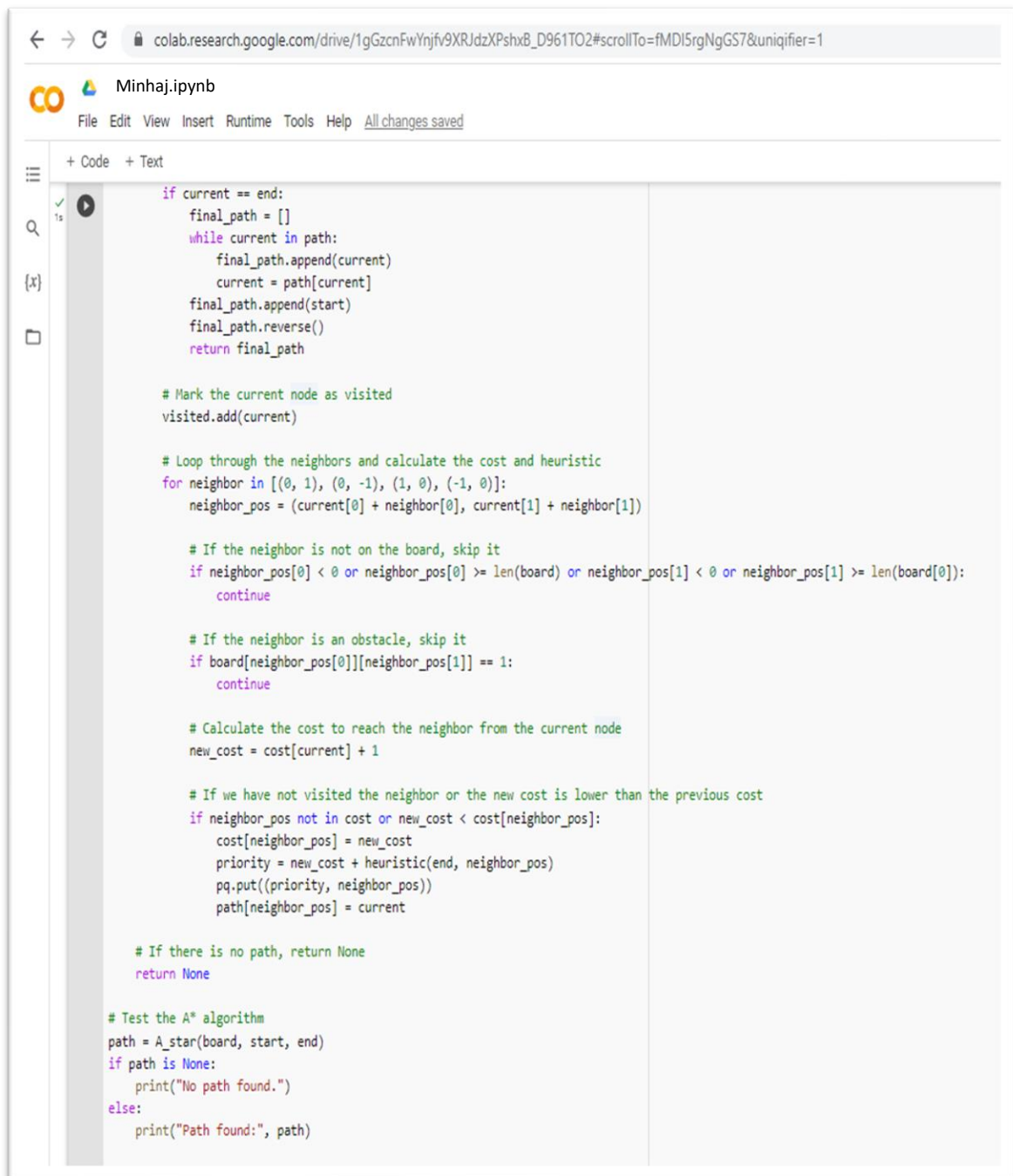
    # Define the visited set and the path dictionary
    visited = set()
    path = {}

    # Define the cost dictionary with the start node having cost 0
    cost = {start: 0}

    while not pq.empty():
        # Get the current node from the queue
        current = pq.get()[1]

        # If we have reached the end node, construct the path and return it
        if current == end:
            final_path = []
            while current in path:
                final_path.append(current)
                current = path[current]
            final_path.append(start)
            final_path.reverse()
            return final_path

        # Mark the current node as visited
        visited.add(current)
```



```
if current == end:
    final_path = []
    while current in path:
        final_path.append(current)
        current = path[current]
    final_path.append(start)
    final_path.reverse()
    return final_path

# Mark the current node as visited
visited.add(current)

# Loop through the neighbors and calculate the cost and heuristic
for neighbor in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
    neighbor_pos = (current[0] + neighbor[0], current[1] + neighbor[1])

    # If the neighbor is not on the board, skip it
    if neighbor_pos[0] < 0 or neighbor_pos[0] >= len(board) or neighbor_pos[1] < 0 or neighbor_pos[1] >= len(board[0]):
        continue

    # If the neighbor is an obstacle, skip it
    if board[neighbor_pos[0]][neighbor_pos[1]] == 1:
        continue

    # Calculate the cost to reach the neighbor from the current node
    new_cost = cost[current] + 1

    # If we have not visited the neighbor or the new cost is lower than the previous cost
    if neighbor_pos not in cost or new_cost < cost[neighbor_pos]:
        cost[neighbor_pos] = new_cost
        priority = new_cost + heuristic(end, neighbor_pos)
        pq.put((priority, neighbor_pos))
        path[neighbor_pos] = current

# If there is no path, return None
return None

# Test the A* algorithm
path = A_star(board, start, end)
if path is None:
    print("No path found.")
else:
    print("Path found:", path)
```

Output:

Path found: [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 4), (2, 4), (3, 4), (4, 4)]

Conclusion: A good example of heuristic search is A* search algorithm. The performance of such heuristic search algorithm depends upon the quality of heuristic function. A* algorithm is executed and the path with minimum cost from source node to destination node is calculated.

Experiment No:03

Aim: Implement Greedy search algorithm for Prim's Minimal Spanning Tree Algorithm

Theory:

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.

Algorithm:

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

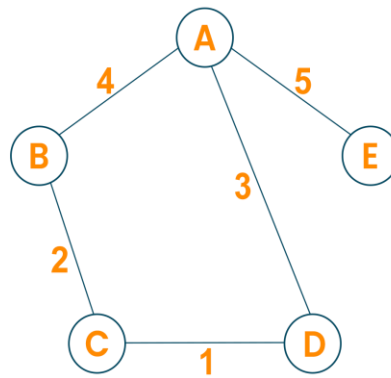
Pseudocode:

```
T =  $\emptyset$ ;  
M = { 1 };  
while (M  $\neq$  N)  
    let (m, n) be the lowest cost edge such that m  $\in$  M and n  $\in$  N - M;  
    T = T  $\cup$  { (m, n) }  
    M = M  $\cup$  {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:

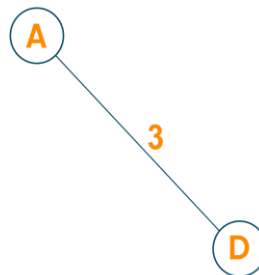
Let us consider the below-weighted graph



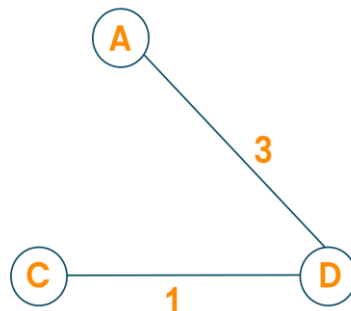
Later we will consider the source vertex to initialize the algorithm



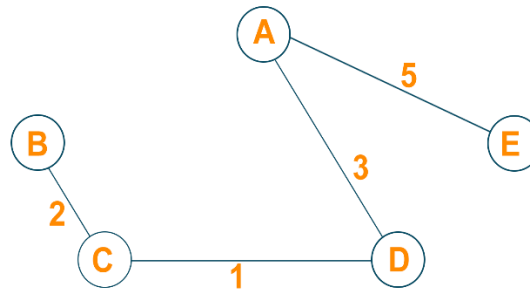
Now, we will choose the shortest weight edge from the source vertex and add it to finding the spanning tree.



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



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



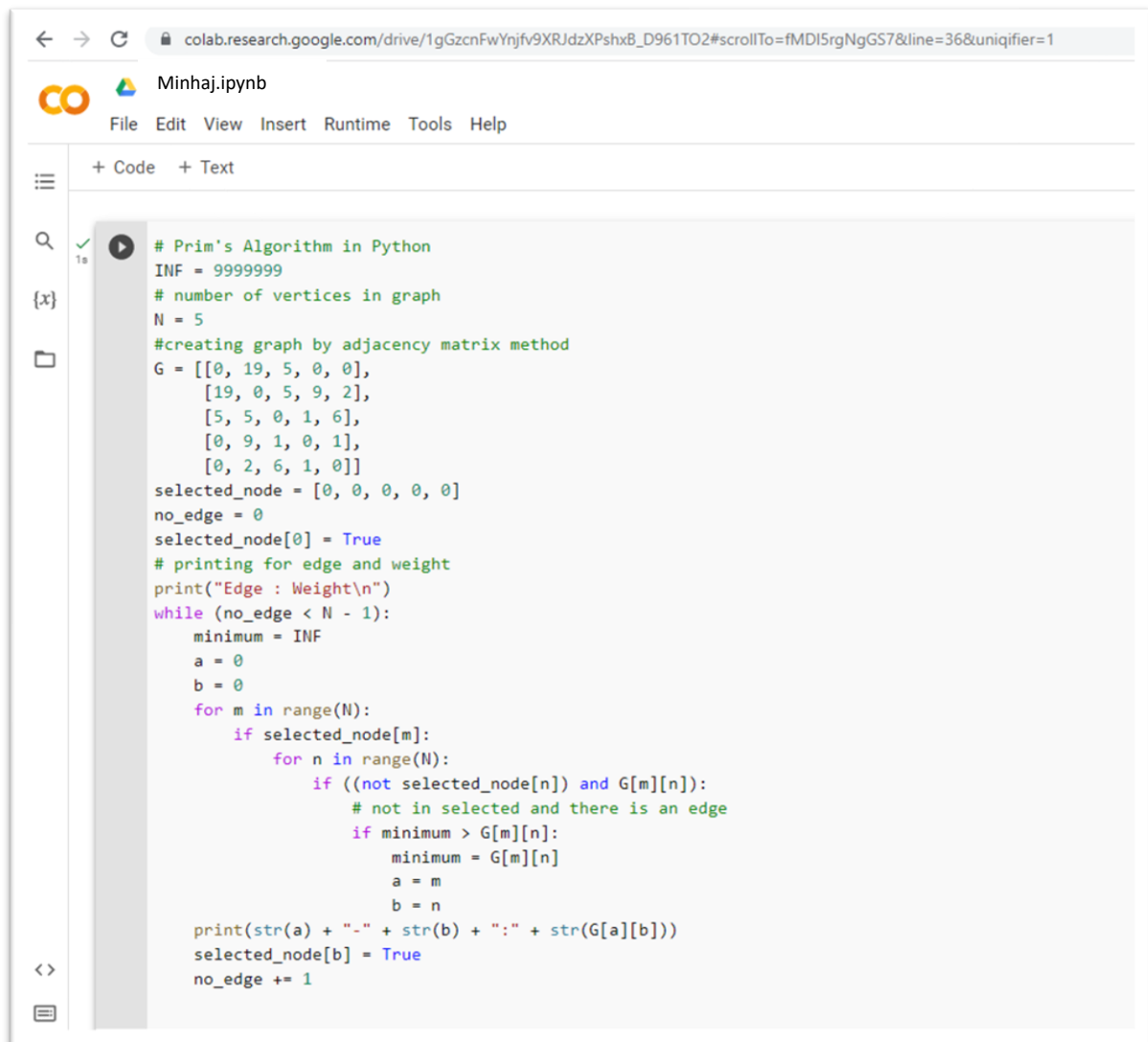
Time Complexity

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

Applications

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

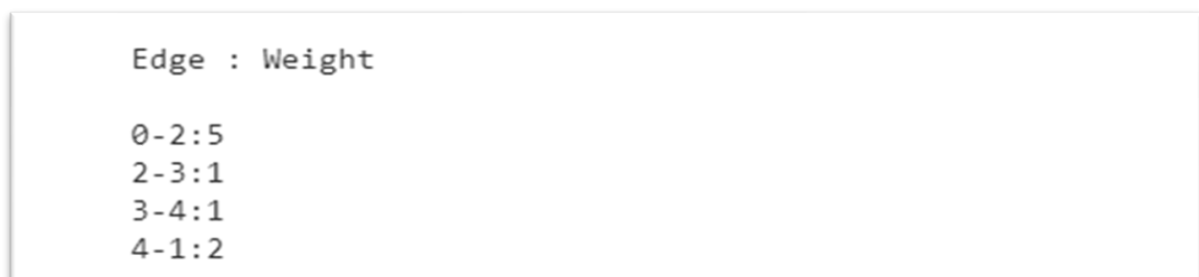
Code:



The screenshot shows a Google Colab notebook interface. The browser address bar displays the URL: `colab.research.google.com/drive/1gGzcnFwYnjfv9XRJdzXPshxB_D961TO2#scrollTo=fMDI5rgNgGS7&line=36&uniqifier=1`. The notebook is titled "Minhaj.ipynb". The menu bar includes "File", "Edit", "View", "Insert", "Runtime", "Tools", and "Help". Below the menu bar, there are tabs for "+ Code" and "+ Text". The code editor contains the following Python code:

```
# 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
```

Output:



The output of the code is displayed in a text box. It shows the edges and their weights selected by the algorithm:

```
Edge : Weight

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

Conclusion: Prim's algorithm is a greedy algorithm which refers to the class of algorithm that use a greedy approach to find the optimal solution to some optimization problem.

Experiment No:04

Aim: 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 queen problem using branch and bound

The n queen problem is a puzzle of placing exactly N Queens on an in N x N chessboard, such that no to queens can lie in the same row column or diagonal.

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

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

Applying the branch and bound approach :

The branch and bound approach suggests 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 check for conditions 1,3 and 4

The 21 diagonal arrays tell which rows and diagonals are already occupied. To achieve this, we need a numbering system to specify which queen is placed.

The 21 diagonal arrays tell which rows and diagonals are already occupied. To achieve this, we need a numbering system to specify which queen is placed.

The 21 diagonal arrays tell which rows and diagonals are already occupied. To achieve this, we need a numbering system to specify which queen is placed.

The 21 diagonal arrays tell which rows and diagonals are already occupied. To achieve this, we need a numbering system to specify which queen is placed.

The 21 diagonal arrays tell which rows and diagonals are already occupied. To achieve this, we need a numbering system to specify which queen is placed.

The 21 diagonal arrays tell which rows and diagonals are already occupied. To achieve this, we need a numbering system to specify which queen is placed.

The diagonal 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.

Pre-processing – create two $N \times N$ 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 21 diagonal will have same value in slash Diagonal and two queens sharing same top-right bottom-left 21 diagonal will have same value in backslash diagonal.

Slash Diagonal(row)(col) = row + col backslash Diagonal (row)(col) = row – col + (N-1) { N = 8 } { we added (N-1) as we do not need negative values in backslash Diagonal }

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)

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

1. whether row 'j' is used or not
2. whether slash Diagonal 'i + j' is used or not
3. whether backslash Diagonal '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 diagonals; and recur for queen i+1.

Algorithm:

Following is the backtracking algorithm for solving the N-Queen problem

1. Initialize an empty chessboard of size N x N.
2. Start with the leftmost column and place a queen in the first row of that column.
3. Move to the next column and place a queen in the first row of that column.
4. Repeat step 3 until either all N queens have been placed or it is impossible to place a queen in the current column without violating the rules of the problem.
5. If all N queens have been placed, print the solution.
6. If it is not possible to place a queen in the current column without violating the rules of the problem, backtrack to the previous column.
7. Remove the queen from the previous column and move it down one row.
8. Repeat steps 4-7 until all possible configurations have been tried.



+ Code + Text

```
[1] # Python program to solve N Queen
    # Problem using backtracking

    global N
    N = 4

    def printSolution(board):
        for i in range(N):
            for j in range(N):
                print (board[i][j],end=' ')
            print()

    # A utility function to check if a queen can
    # be placed on board[row][col]. Note that this
    # function is called when "col" queens are
    # already placed in columns from 0 to col -1.
    # So we need to check only left side for
    # attacking queens
    def isSafe(board, row, col):

        # Check this row on left side
        for i in range(col):
            if board[row][i] == 1:
                return False

        # Check upper diagonal on left side
        for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
            if board[i][j] == 1:
                return False

        # Check lower diagonal on left side
        for i, j in zip(range(row, N, 1), range(col, -1, -1)):
```



+ Code + Text



```
# Check lower diagonal on left side
for i, j in zip(range(row, N, 1), range(col, -1, -1)):
    if board[i][j] == 1:
        return False

return True

def solveNQUtil(board, col):
    # base case: If all queens are placed
    # then return true
    if col >= N:
        return True

    # Consider this column and try placing
    # this queen in all rows one by one
    for i in range(N):

        if isSafe(board, i, col):
            # Place this queen in board[i][col]
            board[i][col] = 1

            # recur to place rest of the queens
            if solveNQUtil(board, col + 1) == True:
                return True

            # If placing queen in board[i][col]
            # doesn't lead to a solution, then
            # queen from board[i][col]
            board[i][col] = 0

    # if the queen can not be placed in any row in
    # this column col then return false
    return False
```



+ Code + Text



{x}



```
# This function solves the N Queen problem using
# Backtracking. It mainly uses solveNQutil() to
# solve the problem. It returns false if queens
# cannot be placed, otherwise return true and
# placement of queens in the form of 1s.
# note that there may be more than one
# solutions, this function prints one of the
# feasible solutions.
def solveNQ():
    board = [ [0, 0, 0, 0],
              [0, 0, 0, 0],
              [0, 0, 0, 0],
              [0, 0, 0, 0]
            ]
    if solveNQutil(board, 0) == False:
        print ("Solution does not exist")
        return False
    printSolution(board)
    return True

# driver program to test above function
solveNQ()
```

```
➞ 0 0 1 0
   1 0 0 0
   0 0 0 1
   0 1 0 0
   True
```

Conclusion: Thus we have implement a solution for a Constraint Satisfaction Problem using Branch and Bound and Backtracking for n-queens problem.

Experiment No:05

Aim: Develop an elementary chatbot for any suitable customer interaction application.

Theory:

A chatbot is software that simulates human-like conversations with users via text messages on chat. Its key task is to help users by providing answers to their questions. Chatbots are programs built to automatically engage with received messages. Chatbots can be programmed to respond the same way each time, to respond differently to messages containing certain keywords and even to use machine learning to adapt their responses to fit the situation.

Chatbots leverage chat mediums like SMS text, website chat windows and social messaging services across platforms like Facebook and Twitter to receive and respond to messages.

Why add a chatbot to website?

When businesses add a chatbot to their support offerings, they're able to serve more customers, improve first response time, and increase agent efficiency. Chatbots help mitigate the high volume of rote questions that come through via email, messaging, and other channels by empowering customers to find answers on their own and guiding them to quick solutions.

When chatbots take simple, repetitive questions off a support team's plate, they give agents time back to provide more meaningful support—nothing kills team productivity like forcing employees to do work that could be automated. Bots can also integrate into global support efforts and ease the need for international hiring and training. They're a cost-effective way to deliver instant support that never sleeps—over the weekends, on holidays, and in every time zone.

The Value of Chatbots:

One way to stay competitive in modern business is to automate as many of your processes as possible. Evidence of this is seen in the rise of self-checkout at grocery stores and ordering kiosks at restaurants

Amazon just opened a store without any cashiers or self-checkouts, limiting human interactions to those only absolutely necessary. The value in chatbots comes from their ability to automate conversations throughout the organization.

Below are five key benefits businesses realize when using chatbots.

1. Save Time & Money
2. Generate Leads & Revenue
3. Guide Users to Better Outcomes
4. Provide 'After Hours' Support
5. Engage Users in a Unique Way

Limitations With A Chatbot

With increasing advancements, there also comes a point where it becomes fairly difficult to work with the chatbots. Following are a few limitations we face with the chatbots.

- **Domain Knowledge** – Since true artificial intelligence is still out of reach, it becomes difficult for any chatbot to completely fathom the conversational boundaries when it comes to conversing with a human.
- **Personality** – Not being able to respond correctly and fairly poor comprehension skills has been more than frequent errors of any chatbot, adding a personality to a chatbot is still a benchmark that seems far away. But we are more than hopeful with the existing innovations and progress-driven approaches.

The chatbots can be defined into two categories; following are the two categories of chatbots:

- 1. Rule-Based Approach** – In this approach, a bot is trained according to rules. Based on this a bot can answer simple queries but sometimes fails to answer complex queries.
- 2. Self-Learning Approach** – These bots follow the machine learning approach which is rather more efficient and is further divided into two more categories.
 - o **Retrieval-Based Models** – In this approach, the bot retrieves the best response from a list of responses according to the user input.
 - o **Generative Models** – These models often come up with answers than searching from a set of answers which makes them intelligent bots as well.

Few applications across Industries:

According to a new survey, 80% of businesses want to integrate chatbots in their business model by 2020. According to a chatbot, these major areas of direct-to-consumer engagement are prime:

Chatbots in Restaurant and Retail Industries

Famous restaurant chains like Burger King and Taco bell has introduced their Chatbots to stand out of competitors of the Industry as well as treat their customers quickly. Customers of these restaurants are greeted by the resident Chatbots, and are offered the menu options- like a counter order, the Buyer chooses their pickup location, pays, and gets told when they can head over to grab their food. Chatbots also works to accept table reservations, take special requests and go take the extra step to make the evening special for your guests. Chatbots are not only good for the restaurant staff in reducing work and pain but can provide a better user experience for the customers.

Chatbots in Hospitality and Travel

For hoteliers, automation has been held up as a solution for all difficulties related to productivity issues, labor costs, a way to ensure consistently, streamlined production processes across the system. Accurate and immediate delivery of information to customers is a major factor in running a successful online Business, especially in the price sensitive and competitive Travel and Hospitality industry. Chatbots particularly have gotten a lot of attention from the hospitality industry in recent months.

Chatbots in Health Industry

Chatbots are a much better fit for patient engagement than Standalone apps. Through these Health-Bots, users can ask health related questions and receive immediate responses. These responses are either original or based on responses to similar questions in the database. The impersonal nature of a bot could act as a benefit in certain situations, where an actual Doctor is not needed. Chatbots ease the access to healthcare and industry has favorable chances to serve their customers with personalized health tips. It can be a good example of the success of Chatbots and Service Industry combo.

Chatbots in E-Commerce

Mobile messengers- connected with Chatbots and the E-commerce business can open a new channel for selling the products online. E-commerce Shopping destination “Spring” was the early adopter. E-commerce future is where brands have their own Chatbots which can interact with their customers through their apps. Chatbots in Fashion Industry Chatbots, AI and Machine Learning pave a new domain of possibilities in the Fashion industry, from Data Analytics to Personal Chatbot Stylists. Fashion is such an industry where luxury goods

can only be bought in a few physical boutiques and one to one customer service is essential. The Internet changed this dramatically, by giving the customers a seamless but a very impersonal experience of shopping. This particular problem can be solved by Chatbots. Customers can be treated personally with bots, which can exchange messages, give required suggestions and information. Famous fashion brands like Burberry, Tommy Hilfiger have recently launched Chatbots for the London and New York Fashion Week respectively. Sephora a famous cosmetics brand and H&M— a fashion clothing brand have also launched their Chatbots.

Chatbots in Finance

Chatbots have already stepped in Finance Industry. Chatbots can be programmed to assist the customers as Financial Advisor, Expense Saving Bot, Banking Bots, Tax bots, etc. Banks and Fintech have ample opportunities in developing bots for reducing their costs as well as human errors. Chatbots can work for customer's convenience, managing multiple accounts, directly checking their bank balance and expenses on particular things. Further about Finance and Chatbots have been discussed in our earlier blog: Chatbots as your Personal Finance Assistant.

Chatbots in Fitness Industry

Chat based health and fitness companies using Chatbot, to help their customers get personalized health and fitness tips. Tech based fitness companies can have a huge opportunity by developing their own Chatbots offering huge customer base with personalized services. Engage with your fans like never before with news, highlights, game-day info, roster and more. Chatbots and Service Industry together have a wide range of opportunities and small to big all size of companies using chatbots to reduce their work and help their customers better.

Chatbots in Media

Big publisher or small agency, our suite of tools can help your audience chatbot experience rich and frictionless. Famous News and Media companies like The Wall Street Journal, CNN, Fox news, etc. have launched their bots to help you receive the latest news on the go.

Chatbot in Celebrity

With a chatbot you can now have one-on-one conversation with millions of fans.

Languages and technologies:

Good knowledge of back-end technologies and analytics.

Languages used for developing chatbots are Java, C#, Python, and Node JS.

To be able to answer arbitrary questions and to develop these smart robots, a deep understanding of machine learning, artificial intelligence, Natural Language Understanding (NLU), and Google Cloud Natural Language API (Application Programming Interface) is required.

Code and Output:



```
[1] def greet(bot_name, birth_year):
    print("Hello! My name is {}".format(bot_name))
    print("I was created in {}".format(birth_year))

[2] def remind_name():
    print('Please, remind me your name.')
    name = input()
    print("What a great name you have, {}".format(name))

[3] def guess_age():
    print('Let me guess your age.')
    print('Enter remainders of dividing your age by 3, 5 and 7.')

[6] rem3 = int(input())
    rem5 = int(input())
    rem7 = int(input())
    age = (rem3 * 70 + rem5 * 21 + rem7 * 15) % 105

    0
    21
    0

print("Your age is {}; that's a good time to start programming!".format(age))
Your age is 21; that's a good time to start programming!
```



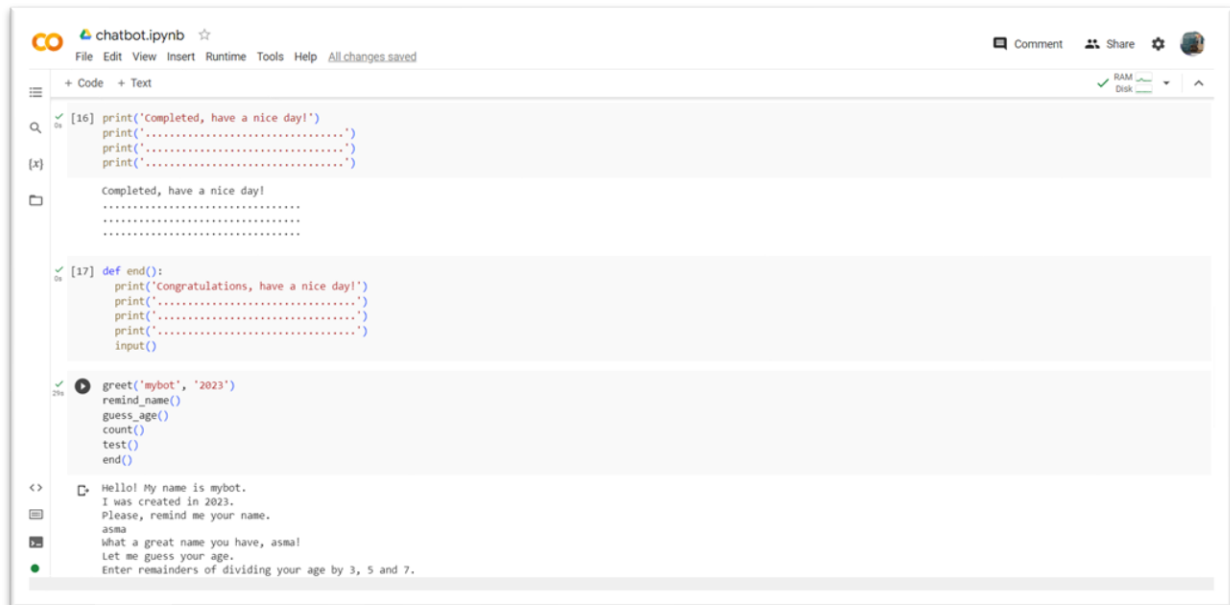
```
[13] def count():
    print("Now I will prove to you that I can count to any number you want.")
    num = int(input())
    counter = 0
    while counter <= num:
        print("{}!".format(counter))
        counter += 1

[14] def test():
    print("Let's test your programming knowledge.")
    print("Why do we use methods?")
    print("1. To repeat a statement multiple times.")
    print("2. To decompose a program into several small subroutines.")
    print("3. To determine the execution time of a program.")
    print("4. To interrupt the execution of a program.")

answer = 2
guess = int(input())
while guess != answer:
    print("Please, try again.")
    guess = int(input())

2

[16] print('Completed, have a nice day!')
    print('.....')
    print('.....')
    print('.....')
```



```
chatbot.ipynb
File Edit View Insert Runtime Tools Help All changes saved


+ Code + Text
[16] print('completed, have a nice day!')
print('.....')
print('.....')
print('.....')

Completed, have a nice day!
.....

[17] def end():
    print('congratulations, have a nice day!')
    print('.....')
    print('.....')
    print('.....')
    input()

greet('mybot', '2023')
remind_name()
guess_age()
count()
test()
end()

Hello! My name is mybot.
I was created in 2023.
Please, remind me your name.
asma
What a great name you have, asma!
Let me guess your age.
Enter remainders of dividing your age by 3, 5 and 7.
```



```
greet('mybot', '2023')
remind_name()
guess_age()
count()
test()
end()

Hello! My name is mybot.
I was created in 2023.
Please, remind me your name.
asma
What a great name you have, asma!
Let me guess your age.
Enter remainders of dividing your age by 3, 5 and 7.
Now I will prove to you that I can count to any number you want.
0
0 !
Let's test your programming knowledge.
Why do we use methods?
1. To repeat a statement multiple times.
2. To decompose a program into several small subroutines.
3. To determine the execution time of a program.
4. To interrupt the execution of a program.
Congratulations, have a nice day!
.....
.....
2
```

Conclusion: We have understood concept of chat bot and implemented an elementary chatbot application for customer interaction.