EXPERIMENT-09

TITLE: N-Queen

AIM: N-Queen is the problem of placing 'N' chess queens on an N×N chessboard. Design a solution for this problem so that no two queens attack each other.
Note: A queen can attack when an opponent is on the same row, column or diagonal

DESCRIPTION:
The N-Queens problem is a classic puzzle that involves placing N queens on an N x N chessboard in such a way that no two queens threaten each other. In chess, a queen can attack in any direction - horizontally, vertically, or diagonally. The goal of the N-Queens problem is to find all possible arrangements of the queens on the board that satisfy this constraint. One common approach to solving the N-Queens problem is by using backtracking. Backtracking is a depth-first search algorithm that systematically explores all possible configurations of queens on the board. It places a queen in one column of the first row and then recursively tries to place the remaining queens in the subsequent rows. If at any point a conflict arises, it backtracks and tries a different position. The N-Queens problem is not only a challenging puzzle but also has practical applications in various fields such as chess, computer vision, and optimization.

ALGOTIHM:

1. Start with an empty chessboard of size N×N.
2. Initialize a variable row to 0, representing the current row being processed.
3. If row is equal to N, it means all queens have been successfully placed on the board. Print or store this configuration as a valid solution and backtrack to find other solutions.
4. Iterate over each column in the current row (col = 0 to N-1).
5. Check if placing a queen at the current position (row, col) is safe. Use a helper function isQueenSafe(board, row, col) to check if the current position is not under attack by any other queen on the board.
6. In the isQueenSafe function, check if there is any queen in the same column, same row, or diagonals as the current position. If a queen is found, return False.
7. If placing a queen at the current position is safe, mark that position as occupied on the board (board[row][col] = 1) to represent the queen.
8. Recursively call the algorithm for the next row (row+1).
9. If the recursive call returns True (indicating that a valid solution was found in the subsequent rows), return True.
10. If the recursive call returns False, it means there is no valid solution for the remaining rows. Backtrack by undoing the current move (board[row][col] = 0) and try the next column in the current row.
11. If none of the columns in the current row lead to a valid solution, return False.
12. Repeat steps 4-10 until all valid solutions have been found.
13. Print or return all the valid solutions found.

CODE:

```python
def print_n_queens(board,ans,row):
    if(row==len(board)):
        print(ans+'.')
        for row in board:
            print(row)
        print ()
        return
    for col in range(len(board)):
        if(board[row][col]==0 and is_queen_safe(board, row, col)==True):
            board[row][col]=1
            print_n_queens(board,ans+str(row)+"-"+str(col)+",",row+1)
            board[row][col]=0


def is_queen_safe(board, row, col):
    # Check upper-left diagonal
    i = row - 1
    j = col - 1
    while i >= 0 and j >= 0:
        if board[i][j] == 1:
            return False
        i -= 1
        j -= 1
    # Check upper row
    i = row - 1
    j = col
    while i >= 0:
        if board[i][j] == 1:
            return False
        i -= 1

    # Check upper-right diagonal
    i = row - 1
    j = col + 1
    while i >= 0 and j < len(board):
        if board[i][j] == 1:
            return False
        i -= 1
        j += 1

    # Check left column
    i = row
    j = col - 1
```

```
    while j >= 0:
       if board[i][j] == 1:
           return False
       j -= 1

    return True

n=int(input("Number  of queens : "))
board=[[0]*n for _ in range(n)]

print_n_queens(board,"",0)
```

OUTPUT:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

PS C:\Users\akash\OneDrive\Desktop\IWT> & c:/Users/akash/OneDrive/Desktop/IWT/akash/Scripts/python.exe c:/Users/akash/OneDrive/Desktop/IWT/myproject/Templates/n
queen.py
Number  of queens : 4
0-1,1-3,2-0,3-2,.
[0, 1, 0, 0]
[0, 0, 0, 1]
[1, 0, 0, 0]
[0, 0, 1, 0]

0-2,1-0,2-3,3-1,.
[0, 0, 1, 0]
[1, 0, 0, 0]
[0, 0, 0, 1]
[0, 1, 0, 0]
```

CONCLUSION AND ANALYSIS:

Time Complexity:
The time complexity of the N-Queens problem using backtracking is typically expressed in terms of the number of recursive function calls and the number of operations performed within each call. In worst case, where all valid configurations/solutions are considered, the total number of recursive calls made by the algorithm is exponential in the size of the chessboard (N). It can be approximated as $O(N!)$. This is because for each row, we have N choices (columns) to place the queen, resulting in N possibilities at each level of recursion. Therefore, the overall time complexity of the N-Queens problem is approximately $O(N!)$.

Space Complexity:
The space complexity of the N-Queens problem primarily depends on the space used to represent the chessboard and the recursive call stack. Space required to store the chessboard is $O(N^2)$ since it is a 2D array of size N×N. The space complexity of the recursive algorithm is $O(N)$. Therefore the overall space complexity is approximately $O(N^2)$.

EXPERIMENT-10:

TITLE: Graph Coloring

AIM: CSE department of CBIT want to generate a timetable for "N" subjects. The following information is given- subject name, subject code and list of subject's code which clashes with this subject. The problem is to identify the list of subjects which can be scheduled on the same timeline such that clashes among them do not exist.

DESCRIPTION:

The Graph Coloring Problem is a classic problem in graph theory and computer science. It involves assigning colors to the vertices of a graph such that no two adjacent vertices share the same color. The goal is to find the minimum number of colors required to color the graph. The problem has various applications in real-world scenarios, such as scheduling, map coloring, register allocation in compilers, and solving Sudoku puzzles.
The Graph Coloring Problem is known to be NP-complete, which means that there is no known polynomial-time algorithm that can solve all instances of the problem. Therefore, various heuristics and approximation algorithms are commonly used to find feasible or near-optimal solutions. The algorithm recursively explores different color assignments for the vertices and backtracks whenever it encounters a conflict. The backtracking algorithm explores the search space of possible colorings until a valid coloring is found or all possibilities are exhausted.

ALGORITHM:

1. Initialize an empty dictionary called clashes to store the subject codes and their corresponding neighbors.
2. Iterate over each subject in the subjects list.
3. Retrieve the subject code and the list of neighbors from the subject tuple.
4. Add the subject code as a key in the clashes dictionary and set its value to the set of neighbors.
5. Initialize an empty dictionary called timetable to store the timetable with colors as keys and subject codes as values.
6. Initialize an empty dictionary called colors to store the assigned color for each subject code.
7. Iterate over each subject in the subjects list again.
8. Retrieve the subject code.
9. Create a set called available_colors by iterating over the neighbors of the subject code and retrieving their assigned colors from the colors dictionary.
10. Assign a color to the subject by finding the smallest positive integer that is not present in the available_colors set.
11. Store the assigned color in the colors dictionary for the subject code.

12. If the assigned color is not already present in the timetable dictionary, add it as a key
    with an empty list as its value.
13. Append the subject code to the list of subjects for the assigned color in the timetable
    dictionary.
14. Finally, the function returns the generated timetable.

CODE:

```
def generate_timetable(subjects):
    clashes = {}
    for subject in subjects:
        subject_code = subject[1]
        clashes[subject_code] = set(subject[2])  # Fix: Use subject[2] for the list of neighbors

    timetable = {}
    colors = {}

    for subject in subjects:
        subject_code = subject[1]
        available_colors = set(colors.get(neighbour, 0) for neighbour in clashes[subject_code])
# Fix: Use colors.get(neighbour, 0) to default to 0

        color = 1
        while color in available_colors:
            color += 1
        colors[subject_code] = color

        if color not in timetable:
            timetable[color] = []
        timetable[color].append(subject_code)

    return timetable

subjects = [
    ("A", "C1", ["C2", "C3"]),
    ("B", "C2", ["C1"]),
    ("C", "C3", ["C1", "C4"]),
    ("D", "C4", ["C3"]),
    ("E", "C5", []),
]

timetable = generate_timetable(subjects)
```
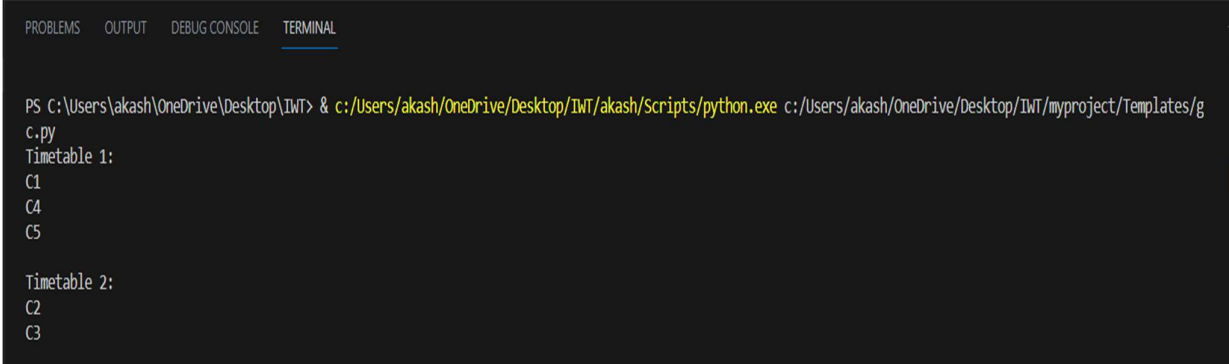
```
for color, subjects in timetable.items():
  print(f"Timetable {color}:")
  for subject in subjects:
    print(subject)
  print()
```

OUTPUT:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

PS C:\Users\akash\OneDrive\Desktop\IWT> & c:/Users/akash/OneDrive/Desktop/IWT/akash/Scripts/python.exe c:/Users/akash/OneDrive/Desktop/IWT/myproject/Templates/g
c.py
Timetable 1:
C1
C4
C5

Timetable 2:
C2
C3
```

CONCLUSION AND ANALYSIS:

Time Complexity:
In the greedy approach to the graph colouring problem, we are greedily assigning a colour to each vertex of the graph as well as checking if the assigned colour meets the criteria or not (no two adjacent vertexes have the same colour). In the greedy approach to the graph colouring problem, the time complexity is $O(V^2+E)$ in the worst case.

Space Complexity:
As the algorithm needs to store the colors assigned to each vertex it is $O(V)$.

EXPERIMENT-11:

TITLE: Shortest Path

AIM: You are given the task of choosing the optimal path to connect "N" devices. The devices are connected with the minimum required N-1 wires into a tree structure, and each device is connected with the other with a wire of length "L" i.e.; "D1" connected to "D2" with a wire of length "L1". This information will be available for all "N" devices.
a) Determine the minimum length of the wire which consists of N-1 wires that will connect all devices.
b) Determine the minimum length of the wire which connects Di and Dj
c) Determine the minimum length of the wire which connects Di to all other devices.
d) Determine the minimum length of the wire which connects Di to all other devices where $1 \leq i \leq N$.

DESCRIPTION:

The problem described is about connecting a given number of devices in an optimal way using a minimum number of wires. The devices are represented as vertices in a tree structure, where each device is connected to the other devices with wires of different lengths.

a) The task is to determine the minimum length of the wire required to connect all the devices using N-1 wires, where N is the total number of devices. This can be solved by finding the minimum spanning tree (MST) of the graph formed by the devices and their connections. The MST algorithm finds a subset of edges that connects all the vertices with the minimum total edge weight.

b) The goal is to find the minimum length of the wire that directly connects two specific devices, Di and Dj, in the tree structure. This can be solved by applying Dijkstra's algorithm, which finds the shortest path between two vertices in a weighted graph. By setting Di as the source vertex and Dj as the target vertex, the algorithm determines the shortest path and its length, which represents the minimum wire length between Di and Dj.

c) The objective is to find the minimum length of the wire that connects a particular device Di to all other devices in the tree structure. This can be solved using Bellman Ford algorithm by setting Di as the source vertex and finding the shortest paths to all other vertices. The sum of the lengths of these shortest paths gives the minimum wire length connecting Di to all other devices.

d) The task is to find the minimum length of the wire connecting each device Di to all other devices in the tree structure, where $1 \leq i \leq N$. This can be solved by applying Floyd Warshall algorithm for each device Di in the tree. By setting Di as the source vertex and finding the shortest paths to all other vertices, the algorithm determines the minimum wire lengths connecting each device to all other devices.

a)
ALGORITHM:

1. Sort all the edges (wires) in ascending order based on their lengths.
2. Initialize an empty set to store the selected edges of the MST.
3. Iterate through the sorted edges, starting from the smallest length:
4. If adding the current edge to the MST set does not create a cycle, add it to the set.
5. To check for a cycle, you can use a disjoint-set data structure (such as the Union-Find algorithm) to keep track of connected components.
6. Continue the iteration until you have N-1 edges in the MST set or you have processed all the edges.
7. The total length of the N-1 selected edges in the MST set will be the minimum length of the wire required to connect all devices.

CODE:

```
import heapq

def minimum_wire_length(N,connections):
    graph=[[] for _ in range(N)]
    for D1,D2,L in connections:
        graph[D1].append((D2,L))
        graph[D2].append((D1,L))
    visited=[False]*N
    min_heap = [(0, 0)]  # (length, device)
    total_length = 0
    while min_heap:
        length, device = heapq.heappop(min_heap)
        if visited[device]:
            continue
        visited[device] = True
        total_length += length
        for neighbor, neighbor_length in graph[device]:
            if not visited[neighbor]:
                heapq.heappush(min_heap, (neighbor_length, neighbor))
    return total_length

def main():
    # Example input
    N = 5  # Number of devices
    connections = [
        (0, 1, 2),  # D0 connected to D1 with wire length 2
        (0, 2, 3),  # D0 connected to D2 with wire length 3
        (1, 3, 1),  # D1 connected to D3 with wire length 1
```
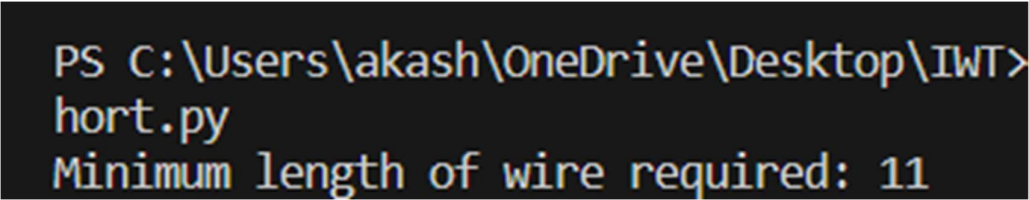
```
    (2, 3, 4),  # D2 connected to D3 with wire length 4
    (2, 4, 5),  # D2 connected to D4 with wire length 5
  ]

  minimum_length = minimum_wire_length(N, connections)
  print("Minimum length of wire required:", minimum_length)

if __name__ == "__main__":
  main()
```

OUTPUT:

```
PS C:\Users\akash\OneDrive\Desktop\IWT>
hort.py
Minimum length of wire required: 11
```

CONCLUSION AND ANALYSIS:

Time Complexity:
The time complexity of Kruskal's algorithm is O(E log E), where E is the number of edges. Sorting the edges initially takes O(E log E) time, and each edge is processed once in the iteration.

Space Complexity:
The space complexity is O(V + E), where V is the number of vertices and E is the number of edges.

b)
ALGORITHM:

1. Define a function dijkstra(N, graph) to implement Dijkstra's algorithm:

   -Initialize an array distances of size N with all elements set to infinity.
   -Set distances[0] = 0 to mark the starting device.
   -Create a min heap min_heap and push the tuple (0, 0) into it.
   -While min_heap is not empty:
      -Pop the minimum distance device (distance, device) from min_heap.
      -If distance is greater than distances[device], continue to the next iteration.
      -Iterate over the neighbors of device in the graph:
      -Calculate the new distance new_distance as the sum of the current distance and the -
      length of the edge to the neighbor.
      -If new_distance is less than the current distance to the neighbor, update
      distances[neighbor] and push (new_distance, neighbor) into min_heap.
2. Define the main function minimum_wire_length(N, connections):

   -Create an empty graph graph with N empty lists.
   -Iterate over the connections (D1, D2, L):
   -Append (D2, L) to the list at index D1 in the graph.
   -Append (D1, L) to the list at index D2 in the graph.
   -Call the dijkstra function with N and graph to get the shortest distances.
   -Calculate the sum of all distances in the distances array to get the minimum wire
   length.
   -Return the minimum wire length.
3. In the main program, specify the number of devices N and the list of connections.
4. Call the minimum_wire_length function with N and connections.
5. Print the minimum length of wire required.

CODE:

```
import heapq


def dijkstra(N, graph):
    distances = [float('inf')] * N  # Initialize distances with infinity
    distances[0] = 0  # Start from device 0
    min_heap = [(0, 0)]  # (distance, device)

    while min_heap:
        distance, device = heapq.heappop(min_heap)

        if distance > distances[device]:
```

```
        continue

    for neighbor, neighbor_length in graph[device]:
        new_distance = distance + neighbor_length
        if new_distance < distances[neighbor]:
            distances[neighbor] = new_distance
            heapq.heappush(min_heap, (new_distance, neighbor))

    return distances


def minimum_wire_length(N, connections):
    graph = [[] for _ in range(N)]

    for D1, D2, L in connections:
        graph[D1].append((D2, L))
        graph[D2].append((D1, L))

    distances = dijkstra(N, graph)
    minimum_length = sum(distances)

    return minimum_length


# Example usage
N = 5  # Number of devices
connections = [
    (0, 1, 2),  # D0 connected to D1 with wire length 2
    (0, 2, 3),  # D0 connected to D2 with wire length 3
    (1, 3, 1),  # D1 connected to D3 with wire length 1
    (2, 3, 4),  # D2 connected to D3 with wire length 4
    (2, 4, 5),  # D2 connected to D4 with wire length 5
]

minimum_length = minimum_wire_length(N, connections)
print("Minimum length of wire required:", minimum_length)
```
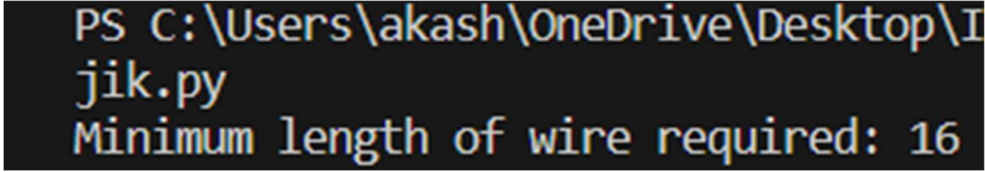
OUTPUT:

```
PS C:\Users\akash\OneDrive\Desktop\I
jik.py
Minimum length of wire required: 16
```

CONCLUSION AND ANALYSIS:

Time Complexity:
Building the graph: $O(E)$, where E is the number of connections (edges) between devices.
Dijkstra's algorithm: $O((V + E) \log V)$, where V is the number of devices (vertices) and E is the number of connections (edges).
Calculating the sum of distances: $O(V)$, as we iterate over all devices.
Therefore, the overall time complexity is $O((V + E) \log V)$.

Space Complexity:
Graph representation: $O(V + E)$, as we store the connections in the graph.
distances array: $O(V)$, as we store the shortest distances from the starting device to all other devices.
min_heap: $O(V)$, as the maximum number of devices in the min heap can be V.
Therefore, the overall space complexity is $O(V + E)$.

c)
ALGORITHM:

1. Initialize the distance array, distances, with infinity for all vertices except the source vertex, which is set to 0.
2. Iterate N-1 times, where N is the number of vertices.
3. For each edge (D1, D2, L) in the list of connections:
   -If distances[D1] is not infinity and distances[D1] + L is less than distances[D2], update distances[D2] to distances[D1] + L.
4. After N-1 iterations, the distances array will contain the shortest distances from the source vertex to all other vertices.
5. If there is a negative weight cycle in the graph, an additional iteration will further update the distances. If any distance is updated in this iteration, it indicates the presence of a negative weight cycle.
6. The minimum wire length required to connect all devices is the sum of the distances in the distances array.

CODE:

```
def bellman_ford(N, connections, source):
    distances = [float('inf')] * N
    distances[source] = 0

    for _ in range(N - 1):
        for D1, D2, L in connections:
            if distances[D1] != float('inf') and distances[D1] + L < distances[D2]:
                distances[D2] = distances[D1] + L

    return distances


def minimum_wire_length(N, connections, Di):
    distances = bellman_ford(N, connections, Di)
    minimum_length = sum(distances)
    return minimum_length
# Example usage
N = 5  # Number of devices
connections = [
    (0, 1, 2),  # D0 connected to D1 with wire length 2
    (0, 2, 3),  # D0 connected to D2 with wire length 3
    (1, 3, 1),  # D1 connected to D3 with wire length 1
    (2, 3, 4),  # D2 connected to D3 with wire length 4
    (2, 4, 5),  # D2 connected to D4 with wire length 5
```
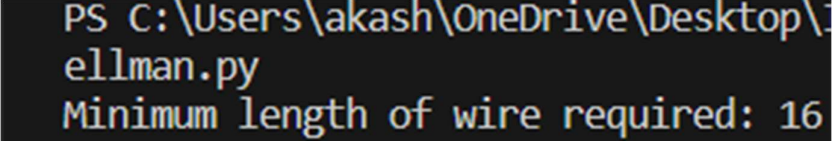
]
Di = 0 # Device to connect to all others

minimum_length = minimum_wire_length(N, connections, Di)
print("Minimum length of wire required:", minimum_length)
OUTPUT:

```
PS C:\Users\akash\OneDrive\Desktop\
ellman.py
Minimum length of wire required: 16
```

CONCLUSION AND ANALYSIS:

Time Complexity:
The time complexity of the Bellman-Ford algorithm is O(N * M), where N is the number of vertices (devices) and M is the number of edges (connections). In the worst case, the algorithm performs N-1 iterations, each iteration checking all M edges.

Space Complexity:
The space complexity of the algorithm is O(N), as it requires an array to store the distances for each device. Additionally, it requires some additional space for variables and temporary storage, but their space requirements are typically negligible compared to the size of the graph.

d)
ALGORITHM:

1. Create a 2D array distances of size N × N and initialize all entries to infinity.
2. For each direct connection (D1, D2, L) in the list of connections, update distances[D1][D2] and distances[D2][D1] with the respective wire lengths.
3. Apply the Floyd-Warshall algorithm to update the distances between all pairs of vertices.
   - Iterate over a middle vertex k from 0 to N-1.
   - For each pair of vertices (i, j) from 0 to N-1, update distances[i][j] by taking the minimum of the current distance distances[i][j] and the sum of the distances through vertex k (distances[i][k] + distances[k][j]).
4. After the algorithm finishes, distances[i][j] represents the minimum wire length to connect device i and device j.
5. Compute the sum of distances for each device, sum(distances[i]), and find the minimum value among them.
6. The minimum length of wire required to connect all devices is the obtained minimum value.

CODE:

```
def floyd_marshall(N, connections):
    INF = float('inf')
    distances = [[INF] * N for _ in range(N)]

    # Initialize distances with direct connections
    for D1, D2, L in connections:
        distances[D1][D2] = L
        distances[D2][D1] = L

    # Update distances using the Floyd-Warshall algorithm
    for k in range(N):
        for i in range(N):
            for j in range(N):
                distances[i][j] = min(distances[i][j], distances[i][k] + distances[k][j])

    return distances


def minimum_wire_length(N, connections):
    distances = floyd_marshall(N, connections)
    min_distances = [sum(distances[i]) for i in range(N)]
    minimum_length = min(min_distances)
```
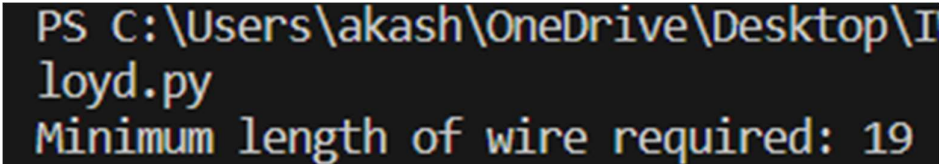
```
    return minimum_length


# Example usage
N = 5  # Number of devices
connections = [
    (0, 1, 2),  # D0 connected to D1 with wire length 2
    (0, 2, 3),  # D0 connected to D2 with wire length 3
    (1, 3, 1),  # D1 connected to D3 with wire length 1
    (2, 3, 4),  # D2 connected to D3 with wire length 4
    (2, 4, 5),  # D2 connected to D4 with wire length 5
]

minimum_length = minimum_wire_length(N, connections)
print("Minimum length of wire required:", minimum_length)
```

OUTPUT:

```
PS C:\Users\akash\OneDrive\Desktop\I
loyd.py
Minimum length of wire required: 19
```

CONCLUSION AND ANALYSIS:

Time Complexity:
The time complexity of the Floyd-Warshall algorithm is $O(N^3)$, where N is the number of devices. This is because the algorithm iterates over all pairs of vertices and considers all possible intermediate vertices.

Space Complexity:
The space complexity of the algorithm is $O(N^2)$ since it requires a 2D array of size $N \times N$ to store the distances between all pairs of vertices.

Overall, the time and space complexity of the Floyd-Warshall algorithm is $O(N^3)$ and $O(N^2)$ respectively.

EXPERIMENT-12:

TITLE: Bi-Connected Graphs

AIM: Bi-connected graphs are used in the design of power grid networks. Consider the nodes as cities and the edges as electrical connections between them, you would like the network to be robust and a failure at one city should not result in a loss of power in other cities.

DESCRIPTION:

An undirected graph is called Biconnected if there are two vertex-disjoint paths between any two vertices. In a Biconnected Graph, there is a simple cycle through any two vertices.
By convention, two nodes connected by an edge form a biconnected graph, but this does not verify the above properties. For a graph with more than two vertices, the above properties must be there for it to be Biconnected.
Or in other words:
A graph is said to be Biconnected if:
- It is connected, i.e., it is possible to reach every vertex from every other vertex, by a simple path.
- Even after removing any vertex the graph remains connected.

To design a robust power grid network, we can model it as a bi-connected graph, where each city is represented as a node, and the electrical connections between cities are represented as edges. By ensuring that the graph is bi-connected, we can guarantee that there are at least two distinct paths between any pair of cities, thereby providing redundancy and minimizing the impact of a single city failure on the overall network.

ALGORITHM:
1.Create a class PowerGridGraph with the following attributes:

- num_cities: The total number of cities in the power grid network.
- adj_list: An adjacency list to store the connections between cities.
- visited: A boolean array to keep track of visited cities during graph traversal.
- disc: An array to store the discovery time of each city during the bi-connectedness check.
- low: An array to store the lowest discovery time of each city during the bi-connectedness check.
- time: A variable to keep track of the current time during the bi-connectedness check.

2.Implement the add_connection method to add a connection between two cities in the power grid network. This method should update the adjacency list.

3.Implement the is_bi_connected method:

- Check if the graph is connected by calling the is_connected method. If it returns False, the power grid network is not robust, and we can return False.
- Initialize the visited, disc, and low arrays to keep track of city visits, discovery time, and lowest discovery time.
- Perform a Depth-First Search (DFS) starting from the first city in the power grid network.
- During the DFS, mark each visited city and update the discovery time and lowest discovery time for each city.
- Track the number of children in the DFS tree.
- If the current city is the root (parent is -1) and has more than one child, it is an articulation point, and the power grid network is not robust. Return False.
- If the current city is not the root and its lowest discovery time is greater than or equal to its parent's discovery time, it is an articulation point, and the power grid network is not robust. Return False.
- If the DFS completes without finding any articulation points, return True.

4.Implement the is_connected method:

- Perform a Depth-First Search (DFS) starting from the first city in the power grid network.
- Mark each visited city during the DFS.
- If all cities are visited, return True.
- If there are unvisited cities after the DFS, return False.

5.Create an instance of the PowerGridGraph class.
6.Add connections between cities using the add_connection method.
7.Call the is_bi_connected method to check if the power grid network is robust.
8.If the method returns True, print "The power grid network is bi-connected." Otherwise, print "The power grid network is not bi-connected."

CODE:

```
class Graph:
    def __init__(self, num_vertices):
        self.num_vertices = num_vertices
        self.adj_list = [[] for _ in range(num_vertices)]
        self.visited = [False] * num_vertices
        self.discovery_time = [float('inf')] * num_vertices
        self.low = [float('inf')] * num_vertices
        self.parent = [-1] * num_vertices
        self.articulation_points = set()
        self.time = 0

    def add_edge(self, u, v):
        self.adj_list[u].append(v)
        self.adj_list[v].append(u)
```

```python
    def find_articulation_points(self):
        for v in range(self.num_vertices):
            if not self.visited[v]:
                self.dfs(v)

    def dfs(self, v):
        children = 0
        self.visited[v] = True
        self.discovery_time[v] = self.time
        self.low[v] = self.time
        self.time += 1

        for u in self.adj_list[v]:
            if not self.visited[u]:
                self.parent[u] = v
                children += 1
                self.dfs(u)
                self.low[v] = min(self.low[v], self.low[u])

                if self.parent[v] == -1 and children > 1:
                    self.articulation_points.add(v)

                if self.parent[v] != -1 and self.low[u] >= self.discovery_time[v]:
                    self.articulation_points.add(v)

            elif u != self.parent[v]:
                self.low[v] = min(self.low[v], self.discovery_time[u])

    def print_articulation_points(self):
        print("Articulation Points:")
        for point in self.articulation_points:
            print(point)


# Example usage:
g = Graph(7)  # Create a graph with 7 vertices

# Add edges
g.add_edge(0, 1)
g.add_edge(1, 2)
g.add_edge(2, 0)
g.add_edge(1, 3)
g.add_edge(1, 4)
g.add_edge(1, 6)
g.add_edge(3, 5)
g.add_edge(4, 5)

g.find_articulation_points()
```

g.print_articulation_points()

OUTPUT:

```
Articulation Points:
1
>
```

CONCLUSION AND ANALYSIS:

Time Complexity:
The time complexity of the algorithm is determined by the Depth-First Search (DFS) performed in the is_bi_connected method. In the worst-case scenario, where all cities are connected to each other, the time complexity is $O(V + E)$, where V is the number of cities (vertices) and E is the number of connections (edges) in the power grid network. This is because the DFS visits each city once and explores all its adjacent connections.

Space Complexity:
The space complexity of the algorithm is determined by the storage of various data structures. In the PowerGridGraph class, the space complexity is $O(V)$, where V is the number of cities, as it requires arrays to store the visited status, discovery time, and lowest discovery time for each city. Additionally, the adjacency list requires $O(E)$ space to store the connections between cities. Overall, the general space complexity of the bi-connected graph algorithm is typically $O(V^2)$ or $O(V + E)$, depending on the graph representation, along with some additional constant space.