Practical 5

- **Aim :** To implement and analyse time complexity of minimum cost spanning tree (Kruskal's algorithm)

- **Theory :**
  Kruskal's algorithm is a greedy approach to find the minimum spanning tree (MST) of a connected, weighted graph. The algorithm chooses the edge with the minimum weight at each step, ensuring the final MST has the least total weight.

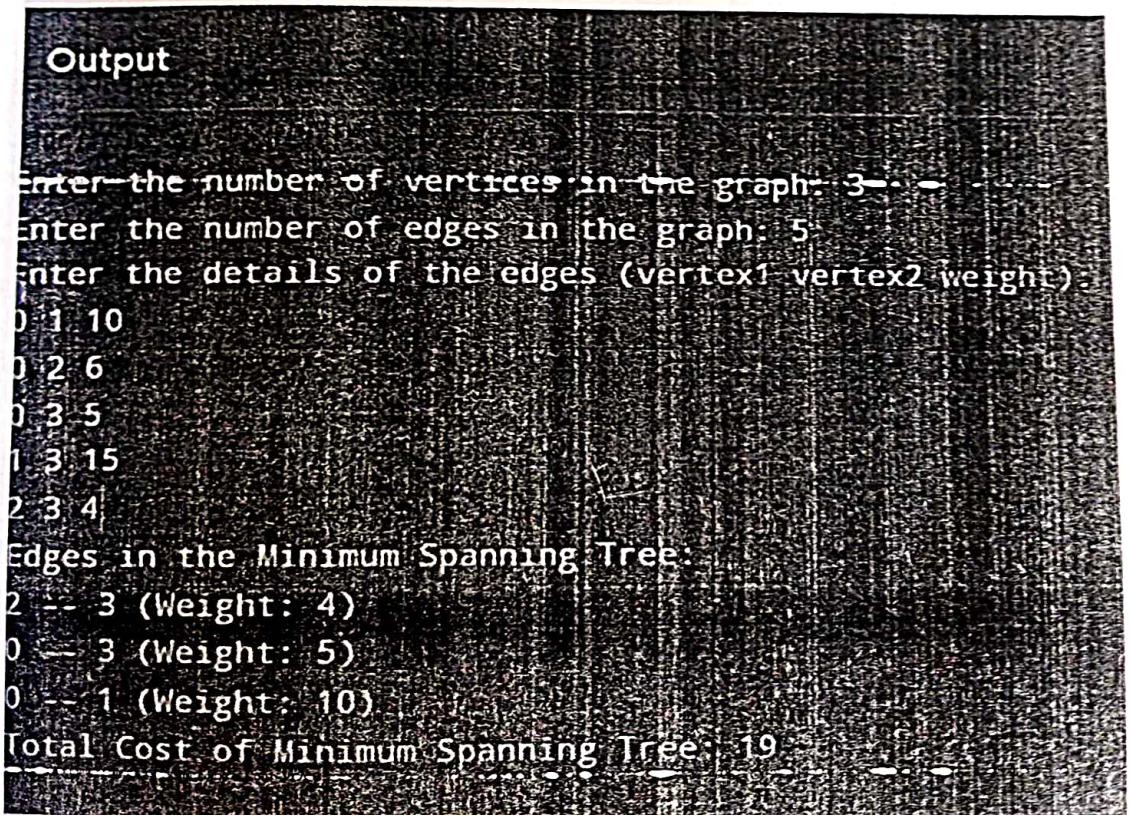
- **MST :** Minimum spanning tree
  A connected, a cycle subgraph of a weighted graph that connects all vertices with the minimum total weight of edges

- **Algorithm :**

1) Create a disjoint-set forest (DSF) Ds with one set for each vertex
2) Sort all edges in G by their weight in non-decreasing order
3) Initialize MST as an empty set.
4) for each edge (u, v, w) in the sorted list
   - Find the root sets of u and v using DSF, find (u) and DSF. find (v)
   - If the root sets are different
     - Add the edge (u, v, w) to MST
     - Merge the root sets of u and v using DSF union (u, v)
5) return MST

```
#include <stdio.h>
#include <stdlib.h>
int comparator(const void* p1, const void* p2) {
  const int(*x)[3] = p1;
  const int(*y)[3] = p2;
  return (*x)[2] - (*y)[2];}
void makeSet(int parent[], int rank[], int n) {
  for (int i = 0; i < n; i++) {
    parent[i] = i;
    rank[i] = 0;
  }
}

int findParent(int parent[], int component) {
  if (parent[component] == component)
    return component;

  return parent[component] = findParent(parent, parent[component]);
}

void unionSet(int u, int v, int parent[], int rank[], int n) {

  u = findParent(parent, u);
  v = findParent(parent, v);

  if (rank[u] < rank[v]) {
    parent[u] = v;
  } else if (rank[u] > rank[v]) {
    parent[v] = u;
  } else {
    parent[v] = u;
```

```
    int n, m;
    printf("Enter the number of vertices in the graph: ");
    scanf("%d", &n);
    printf("Enter the number of edges in the graph: ");
    scanf("%d", &m);

    int edge[m][3];

    printf("Enter the details of the edges (vertex1 vertex2 weight):\n");
    for (int i = 0; i < m; i++) {
        scanf("%d %d %d", &edge[i][0], &edge[i][1], &edge[i][2]);
    }

    kruskalAlgo(m, edge);

    return 0;
```



: Time Complexity :-

Best Case : The best-case scenario occurs when the graph already forms the minimum spanning tree itself. In this case, all edges have the same weight and no comparison or unions are needed during the sorting or selecting process. Therefore the time complexity becomes
$O(E \log E) = O(E * 1) = O(E)$

Average case : In the average case the edges are randomly distributed in terms of weights. The sorting step dominates the overall complexity.
$O(E \log E)$

Worst case : The worst-case scenario occurs when the sorting algorithm encounters the worst case input, leading to $O(E^2)$ complexity. Efficiently sorting algorithms like merge sort or Quicksort with average $O(n \log n)$ complexity are used. Additionally, the worst case performance can occurs if the graph had multiple edges with the same weight.
$O(E^2)$

Best Case : $O(E)$
Average case : $O(E \log E)$
Worst Case : $O(E^2)$

Result: Successfully implemented and analysed time complexity of minimum cost spanning tree (Kruskal's algorithm)

P	T	D	K	Total
3M	3M	3M	6M	15M
03	03	02	05	13
Sign With date				

19/3/24

Practical 6

- Aim: To implement and analyze time complexi -ty single source shortest path algorithm (Dijkstra Algorithm.)

- Theory:

Dijkstra algorithm finds the shortest path from one vertex to all other vertex It does ~~not~~ so by repeatedly selecting the nearest unvisited vertex and calculating the distance to all the unvisited neighbour vertices

Dijkstra's algorithm is used for solving single source shortest path problem for directed or undirected path.
Single source means that one vertex is chosen to be the start and the algorithm will find the shortest path from the vertex to all other vertices

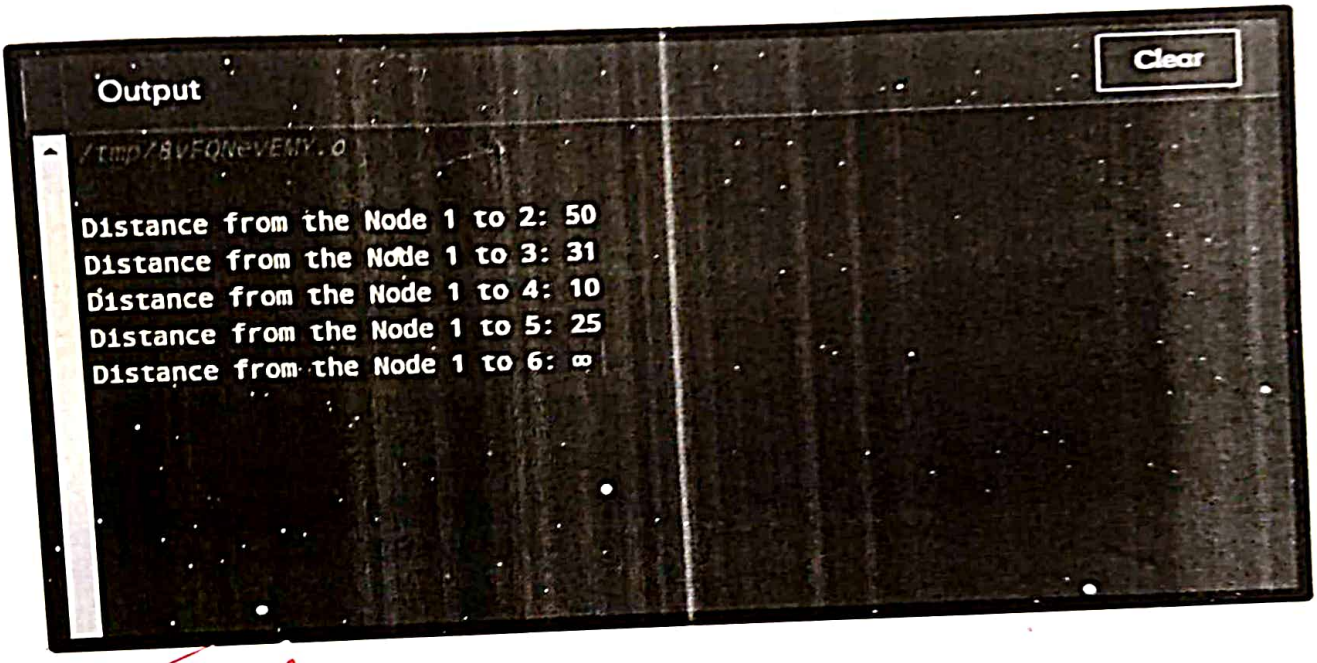
Dijkstra's algorithm does not work for graph with negative edges. for negative edges the bellman ford algorithm that is described.

Algorithm:

```
Dijkstra (Graph , source)
    create vertex set Q
    for each vertex v in graph
    dist [v]- ∞
    add v to Q
    dist [source] =0
```

```
#include <stdio.h>
#define INF 9999
#define MAX 10


void DijkstraAlgorithm(int Graph[MAX][MAX], int size, int start) {
  int cost[MAX][MAX], distance[MAX], previous[MAX];
  int visited_nodes[MAX], counter, minimum_distance, next_node, i, j;
  for (i = 0; i < size; i++)
    for (j = 0; j < size; j++)
      if (Graph[i][j] == 0)
        cost[i][j] = INF;
      else
        cost[i][j] = Graph[i][j];

    for (i = 0; i < size; i++) {
      distance[i] = cost[start][i];
      previous[i] = start;
      visited_nodes[i] = 0;
    }

    distance[start] = 0;
    visited_nodes[start] = 1;
    counter = 1;

    while (counter < size - 1) {
      minimum_distance = INF;

      for (i = 0; i < size; i++)
        if (distance[i] < minimum_distance && !visited_nodes[i]) {
          minimum_distance = distance[i];
          next_node = i;
          }
        visited_nodes[next_node] = 1;
        for (i = 0; i < size; i++)
          if (!visited_nodes[i])
            if (minimum_distance + cost[next_node][i] < distance[i]) {
              distance[i] = minimum_distance + cost[next_node][i];
              previous[i] = next_node;
            }
        counter++;
      }
      for (i = 0; i < size; i++)
        if (i != start) {
          if(distance[i]!=9999){
          printf("\nDistance from the Node %d to %d: %d",start+1, i+1, distance[i]); }
          else{
            printf("\nDistance from the Node %d to %d: ∞ ", start+1,i+1);
          }
        }
      }
```

```
int main() {
 int Graph[MAX][MAX], i, j, size, source;
 size = 6;
 Graph[0][0] = 0;
 Graph[0][1] = 50;
 Graph[0][2] = 0;
 Graph[0][3] = 10;
 Graph[0][4] = 0;
 Graph[0][5] = 0;

 Graph[1][0] = 0;
 Graph[1][1] = 0;
 Graph[1][2] = 10;
 Graph[1][3] = 20;
 Graph[1][4] = 0;
 Graph[1][5] = 0;

 Graph[2][0] = 0;
 Graph[2][1] = 0;
 Graph[2][2] = 0;
 Graph[2][3] = 0;
 Graph[2][4] = 30;
 Graph[2][5] = 0;

 Graph[3][0] = 15;
 Graph[3][1] = 0;
 Graph[3][2] = 0;
 Graph[3][3] = 0;
 Graph[3][4] = 15;
 Graph[3][5] = 0;

 Graph[4][0] = 0;
 Graph[4][1] = 35;
 Graph[4][2] = 6;
 Graph[4][3] = 0;
 Graph[4][4] = 0;
 Graph[4][5] = 0;

 Graph[5][0] = 0;
 Graph[5][1] = 0;
 Graph[5][2] = 0;
 Graph[5][3] = 0;
 Graph[5][4] = 3;
 Graph[5][5] = 0;

 source = 0;
 DijkstraAlgorithm(Graph, size, source);

 return 0;
}
```


Output
Clear
/tmp/8VFQNeVEMY.o
Distance from the Node 1 to 2: 50
Distance from the Node 1 to 3: 31
Distance from the Node 1 to 4: 10
Distance from the Node 1 to 5: 25
Distance from the Node 1 to 6: ∞

lude <stdio.h>

while Q is not empty
u = Extract_Min[Q]
for each neighbour v of u
Relax (u, v)

* Time complexity : 1) Best Case :
- The time complexity is determined by the graph's number of vertices (V) and edges (E)
- In the scenario, the algorithm efficiently find the shortest path, with the priority queue operation optimized, leading to the overall complexity of O(E log V)
- This scenario is typically encountered when the graph is sparse, meaning it has relatively few edges compared to vertices.

2) Average Case :
- The average-case time complexity of Dijkstra algorithm is typically same as the best case scenario O(E log V)
- This is because Dijkstra algorithm perform well on most real-world graph which are often neither extremly spurse nor fully connected.
- The algorithm efficiently find shortest paths in graph with varying densities finding a balance between the quantity of edges and vertices

clude <stdio.h>

3) Worst Case: $O((V^2) \log V)$

- In the worst case scenario Dijkstra algorithm operates less efficiently, typically when using a simple priority queue or array base implementation
- This occurs when the graph is dense with many edges and the priority queue operation become less efficient due to the lacks of optimization

- Result : In this practical we have successfully implemented and analyzed time complexity of single source shortest path (Dijkstra's algorithm).

P	T	D	K	Total
3M	3M	3M	6M	15M
03	03	03	05	14
Sign With date				

19/3/24

Practical 7

RAISONI GROUP
a vision beyond

- **Aim :** To implement and analyze all time complexity all pair shortest path algorithm (Floyd Warshall's algorith)

- **Theory :**

The Floyd - Warshall algorithm is used to find the shortest paths between all pairs of vertices in a weighted graph, including negative -weight edges. it works for both directed and undirected graphs. The algorithm maintains a matrix where each entry represents the shortest distance between two vertices.

- **Algorithm :**

n= no. of vertices
A= matrix of dimension n*n

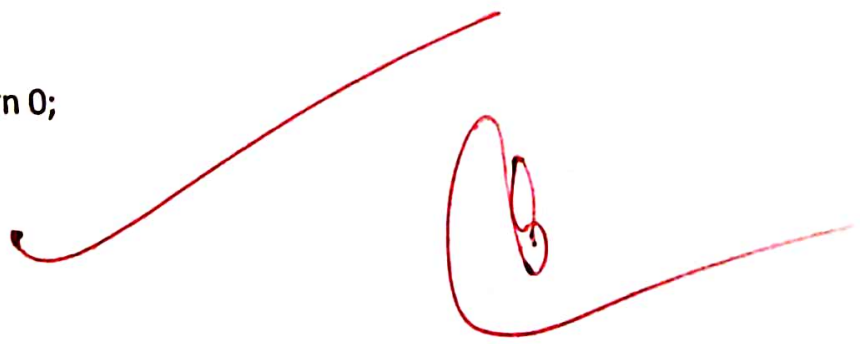
```
for k=1 to n
    for i= 1 to n
        for j = 1 to n
```

$A^k[i,j] = \min(A^{k-1}[i,j], A^{k-1}[i,k] + A^{k-1}[k,j])$

return A

```
le <stdio.h>

e INF 99999
ne MAX_VERTICES 100

djMatrix[MAX_VERTICES][MAX_VERTICES];

floydWarshall(int numVertices) {
t dist[numVertices][numVertices];
nt i, j, k;

/ Initialize dist matrix with graph matrix
for (i = 0; i < numVertices; i++) {
  for (j = 0; j < numVertices; j++) {
    dist[i][j] = adjMatrix[i][j];
  }
}

// Apply Floyd-Warshall algorithm
for (k = 0; k < numVertices; k++) {
  for (i = 0; i < numVertices; i++) {
    for (j = 0; j < numVertices; j++) {
      if (dist[i][k] + dist[k][j] < dist[i][j])
        dist[i][j] = dist[i][k] + dist[k][j];
    }
  }
}

// Print the shortest distances
printf("Shortest distances between every pair of vertices:\n");
for (i = 0; i < numVertices; i++) {
  for (j = 0; j < numVertices; j++) {
    if (dist[i][j] == INF)
      printf("%7s", "INF");
```

```
else
    printf("%7d", dist[i][j]);

rintf("\n");



nain() {
t numVertices, i, j;

rintf("Enter the number of vertices: ");
scanf("%d", &numVertices);

printf("Enter the adjacency matrix (enter INF for no edge):\n");
 for (i = 0; i < numVertices; i++) {
    for (j = 0; j < numVertices; j++) {
       scanf("%d", &adjMatrix[i][j]);
       if (adjMatrix[i][j] == INF && i != j)
           adjMatrix[i][j] = INF;
     }
  }

   floydWarshall(numVertices);

    return 0;
}
```



```
mber of vertices: 4
jacency matrix (enter INF for no edge):

9999

99 0
stances between every pair of vertices:
  3    INF      5
  0    INF      7
  7      0      1
  9    INF      0


xecution Successful ===
```

- Time complexity :

- Best Case $O(V^3)$:

In the best-case scenario the algorithm completes all iterations of the nested loops without any relaxation steps needed.
This implies that the shortest paths between all pairs of vertices are already determined and no updates are necessary
Because of this the three nested loops each of which iterates through every vertex with V vertices each loop iterates V times resulting in a time complexity of $O(V^3)$ for the best case.

- Average Case : $O(V^3)$

The average case time complexity of the floyd warshall algorithm is also $O(V^3)$. This complexity holds true across various graph structures and ~~dest~~ densities as the algorithm's performance primarily depends on the number of vertices and the number of iterations needed to compute shortest paths between all pairs of vertices.

- Worst Case : $O(V^3)$

Conversely, in the worst-case scenario the algorithm performs relaxation steps for each pair of vertices during all iterations of the nested loops.
This means that the algorithm needs to update distances between vertices

multiple times until the shortest paths are determined. with each loop iterates v times resulting in a time complexity of $O(v^3)$ for the worst case as well.

- **Result** : Hence, we have successfully implemented shortest path algorithm (Floyd warshall) and analyzed time complexity of it.

P	T	D	K	Total
3M	3M	3M	3M	15M
03	03	03	04	13

Sign with

02/4/24

RAISONI GROUP
a vision beyond

Practical 8

- Aim: To implement and analyse backtracking algorithm (N queens problem)

- Theory:

The N-queens problem is a classical puzzle in computer science and mathematics. It involves placing N queens on an N×N chessboard in such a way that no two queens threaten each other

Constraints:

1) Each row can contain only one queen
2) Each column can contain only one queen.
3) No two queens can be placed on the same diagonal.

The objective is to find all possible arrangements of N queens on the N×N chessboard that satisfy the constraints.

Algorithm:

```
Queen(n) {
for column=1 to n do
[ if(place(row, column))
  {
  board[row] = column ;
if (row=n) then
print_board(n)
else
  Queen (row+1,n)
}}
```

```
#include <stdio.h>
#include <stdbool.h>
#define MAX_SIZE 20
int board[MAX_SIZE][MAX_SIZE];
int solutions_count = 0;
bool isSafe(int row, int col, int n) {
  int i, j;
  for (i = 0; i < col; i++)
    if (board[row][i])
      return false;
  for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
    if (board[i][j])
      return false;
  for (i = row, j = col; j >= 0 && i < n; i++, j--)
    if (board[i][j])
      return false;
  return true;
}
bool solveNQueens(int col, int n) {
  if (col >= n) {
    solutions_count++;
    if (solutions_count > 2)
      return true;
    printf("Solution %d:\n", solutions_count);
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++){
        if(board[i][j]==1)
          printf("Q");
        else
```

```
            printf(". ");
        }
        printf("\n");
    }
    printf("\n");
    return false;
  }
  for (int i = 0; i < n; i++) {
    if (isSafe(i, col, n)) {
      board[i][col] = 1;
      if (solveNQueens(col + 1, n))
        return true;
      board[i][col] = 0;
    }
  }
  return false;
}

int main() {
  int n;
  printf("Enter the board size (maximum %d): ", MAX_SIZE);
  scanf("%d", &n);

  if (n <= 0 || n > MAX_SIZE) {
    printf("Invalid board size.\n");
    return 1;
  }

  for (int i = 0; i < n; i++)
```

```
        for (int j = 0; j < n; j++)
            board[i][j] = 0;

    solveNQueens(0, n);
    if (solutions_count == 0)
        printf("No solutions found.\n");

    return 0;
}
```

```
Enter the board size (maximum 20): 8
Solution 1:
Q. . . . . .
. . . . . Q.
. . . . Q. . .
. . . . . Q
. Q. . . . . .
. . . Q. . . .
. . . . . Q. .
. . Q. . . . .

Solution 2:
Q. . . . . .
. . . . . Q.
. . . Q. . . .
. . . . . Q. .
. . . . . Q
. Q. . . . . .
. . . . Q. . .
. . Q. . . . .

PS C:\Users\ASUS\OneDrive\Desktop\practice code>
```

```
place (row, column)
for i=1 to row-1 do
  if (board[i] = column]) then
        return 0
else if (abs(board[i] - column) = abs (i-row)) then
        return 0
  }
return 1.
```

Time complexity :

- **Best Case Time Complexity :**

The best-case scenario occurs when the first arrangement of queens tried leads to a solution. The best case complexity is usually considered to be $O(N!)$, where N is the size of the chessboard.

- **Average Case Time complexity :**

The average case time complexity can be more complex as it depends on various factors such as the specific implementation of algorithm, the size of the chessboard, and the distribution of valid solutions. average, time complexity is $O(N!)$

- **Worst Case Time complexity:**
  The worst-case scenario happens when the algorithm needs to explore all possible arrangements of queen on the board before finding a solution. occurs when the last arrangement is tried before finding a valid solution. The worst case time complexity is O(N!)

- **Result:** Hence, we have successfully implemented backtracking algorithm (N queens problem) and analysed its time complexity.

P	T	D	K	Total
3M	3M	3M	6M	15M
03	03	03	04	13
Sign With date				

08/4/24