**Experiment No. 2:** **Greedy method strategy**

Date:

**Aim:** Write a C program to implement the following program using Greedy method strategy

a. Prims algorithm.

b. Kruskals algorithm.

c. Single source shortest path algorithm.

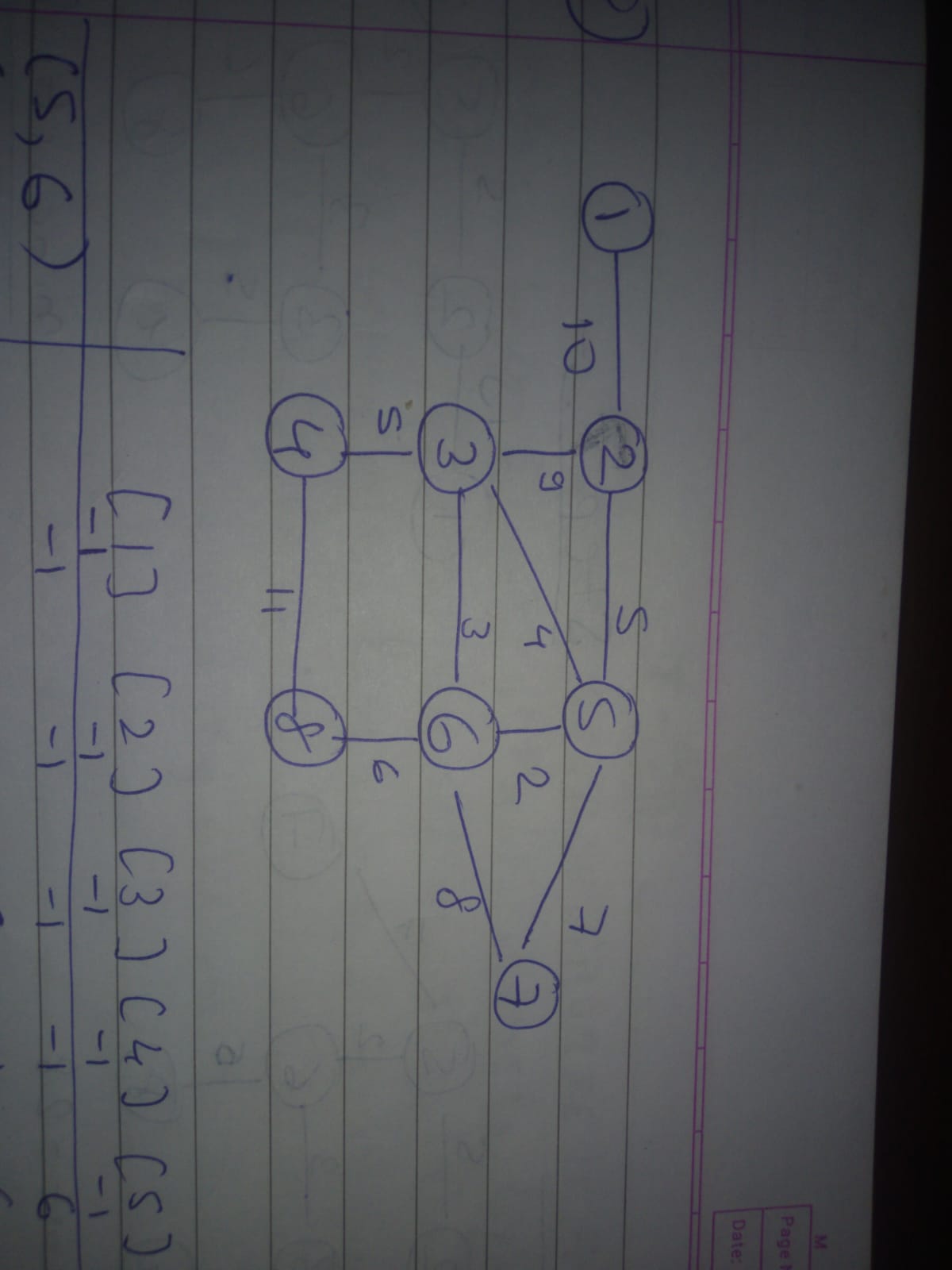
**THEORY:**

* The greedy algorithm is a problem-solving approach that chooses the best available option at each step of the solution process.
* It's based on the principle of making locally optimal choices with the hope of finding a global optimum solution.
* The algorithm starts with an empty solution set and iteratively builds it up by selecting the best available option at each step.
* The selection of the best option is based on a certain criterion, which can be the highest or lowest value, the maximum or minimum weight, or any other relevant metric.
* The algorithm does not revisit the choices made in previous steps, which may result in a suboptimal solution.
* The greedy algorithm is useful for solving optimization problems, such as the Knapsack problem, minimum spanning tree, and shortest path problem.
* It's often fast and efficient, but may not always guarantee the optimal solution, particularly in complex problems.
* The correctness of the greedy algorithm depends on the problem structure and the choice of the criterion for selecting the best option.
* Sometimes, a greedy approach can be combined with other algorithms to improve the quality of the solution

**a) Prim’s algorithm**

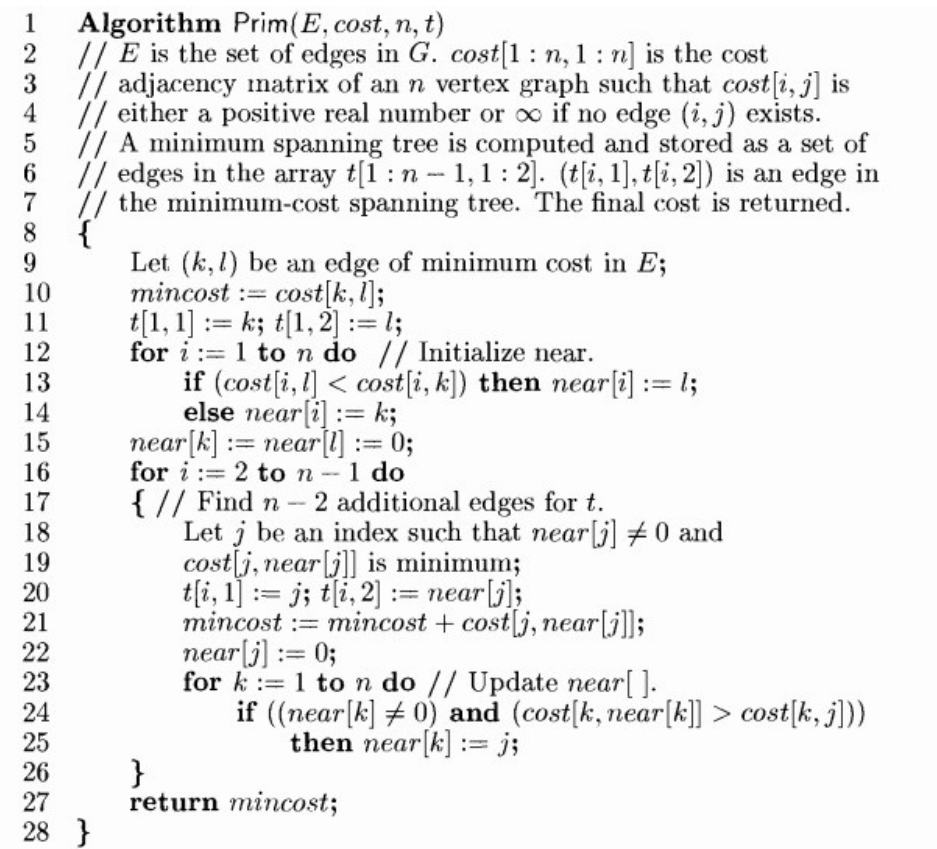
Date:

**Problem Statement:**

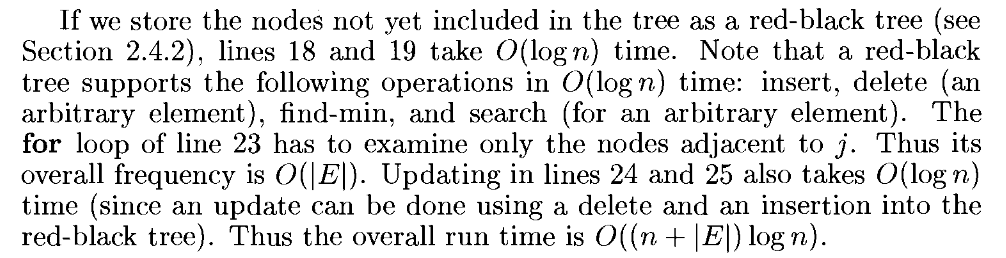
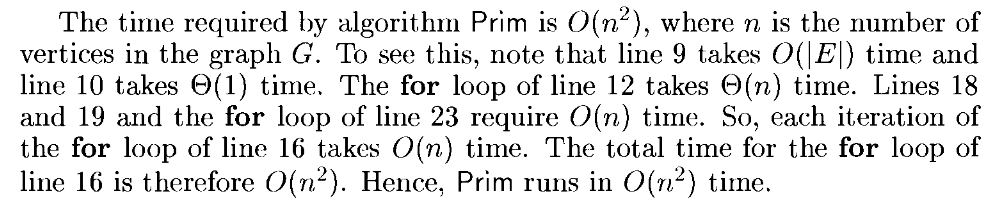


Write a C program to implement a minimum cost spanning tree using prims algorithm on following graph:

**Algorithm:**

****

**Time and Space Complexity:**



When an array is used to represent the graph in Prim's algorithm, the space complexity is O(V^2), where V is the number of vertices in the graph.

The array is used to represent the graph as an adjacency matrix, where each element in the array represents the weight of the edge connecting two vertices. Since the graph is undirected, the matrix is symmetric along the diagonal. Therefore, only half of the matrix needs to be stored.

The size of the array is V x V, which is proportional to the square of the number of vertices in the graph.

**Code:**

#include <stdio.h>

int adj[100][100], cost[100][100], near[100];

void init\_cost(int n)

{

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

{

for (int j = 0; j < n; j++)

cost[i][j] = \_\_INT32\_MAX\_\_;

}

}

void accept\_graph(int n)

{

int max\_edges = n \* (n - 1) / 2;

int origin, destin, edgecost;

printf("Enter the edges of the graph and their respective cost.\n");

for (int i = 0; i < max\_edges; i++)

{

printf("Enter the edge,( 0 0 randval) to quit :");

scanf("%d %d %d", &origin, &destin, &edgecost);

if ((origin == 0) && (destin == 0))

break;

if (origin > n || destin > n || origin <= 0 || destin <= 0)

{

printf("Invalid edge.\n");

i--;

}

else

{

adj[origin - 1][destin - 1] = 1;

cost[origin - 1][destin - 1] = edgecost;

adj[destin - 1][origin - 1] = 1;

cost[destin - 1][origin - 1] = edgecost;

}

}

}

void minedge(int \*k, int \*l, int n)

{

int mincost = cost[0][0];

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

{

for (int j = 0; j < n; j++)

{

if (cost[i][j] < mincost)

{

mincost = cost[i][j];

\*k = i >= j ? j : i;

\*l = i >= j ? i : j;

}

}

}

}

void display\_arr(int a[], int n)

{

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

printf("%d ", a[i] + 1);

printf("\n");

}

int prims(int n, int t[n - 1][2])

{

int k, l;

minedge(&k, &l, n);

t[0][0] = k;

t[0][1] = l;

int mincost = cost[k][l];

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

{

if (cost[i][k] < cost[i][l])

near[i] = k;

else

near[i] = l;

}

near[k] = -1;

near[l] = -1;

printf("Edge considered is %d %d.\n", k + 1, l + 1);

int j;

for (int i = 1; i < n - 1; i++)

{

int min = \_\_INT32\_MAX\_\_;

int minj = -1;

printf("Near array: ");

display\_arr(near, n);

printf("Mincost: %d\n", mincost);

for (j = 0; j < n; j++)

{

if (near[j] != -1 && cost[j][near[j]] < min)

{

min = cost[j][near[j]];

minj = j;

}

}

printf("Edge considered is %d %d.\n", near[minj] + 1, minj + 1);

t[i][0] = minj;

t[i][1] = near[minj];

mincost += cost[minj][near[minj]];

near[minj] = -1;

for (int k = 0; k < n; k++)

{

if (near[k] != -1 && cost[k][near[k]] > cost[k][minj])

near[k] = minj;

}

}

return mincost;

}

int main()

{

int n;

printf("Enter the number of vertices of the graph.\n");

scanf("%d", &n);

int t[n - 1][2];

init\_cost(n);

accept\_graph(n);

int ans = prims(n, t);

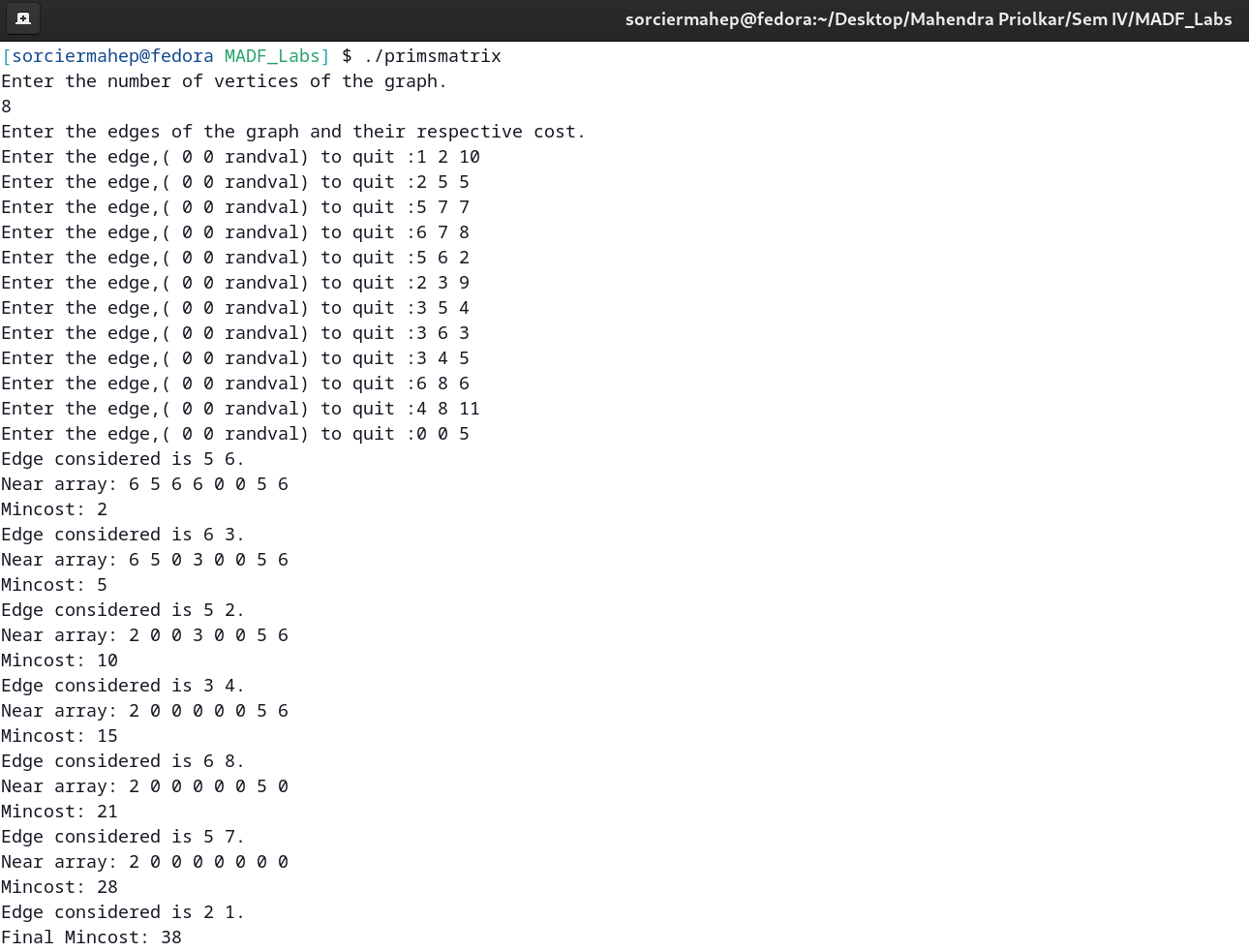
printf("Final Mincost: ");

printf("%d\n", ans);

return 0;

}

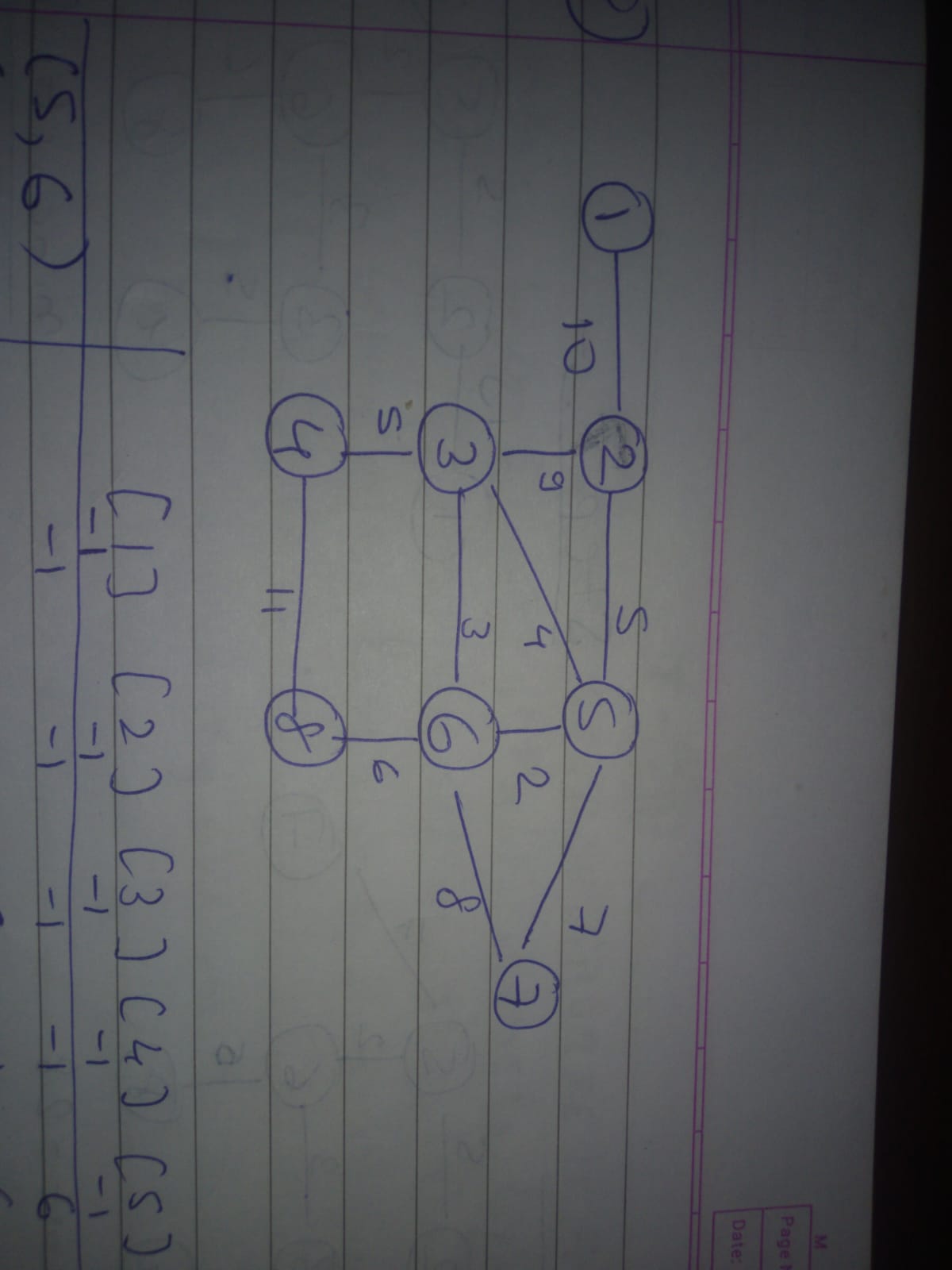
**Output:**



**b) Kruskals algorithm**

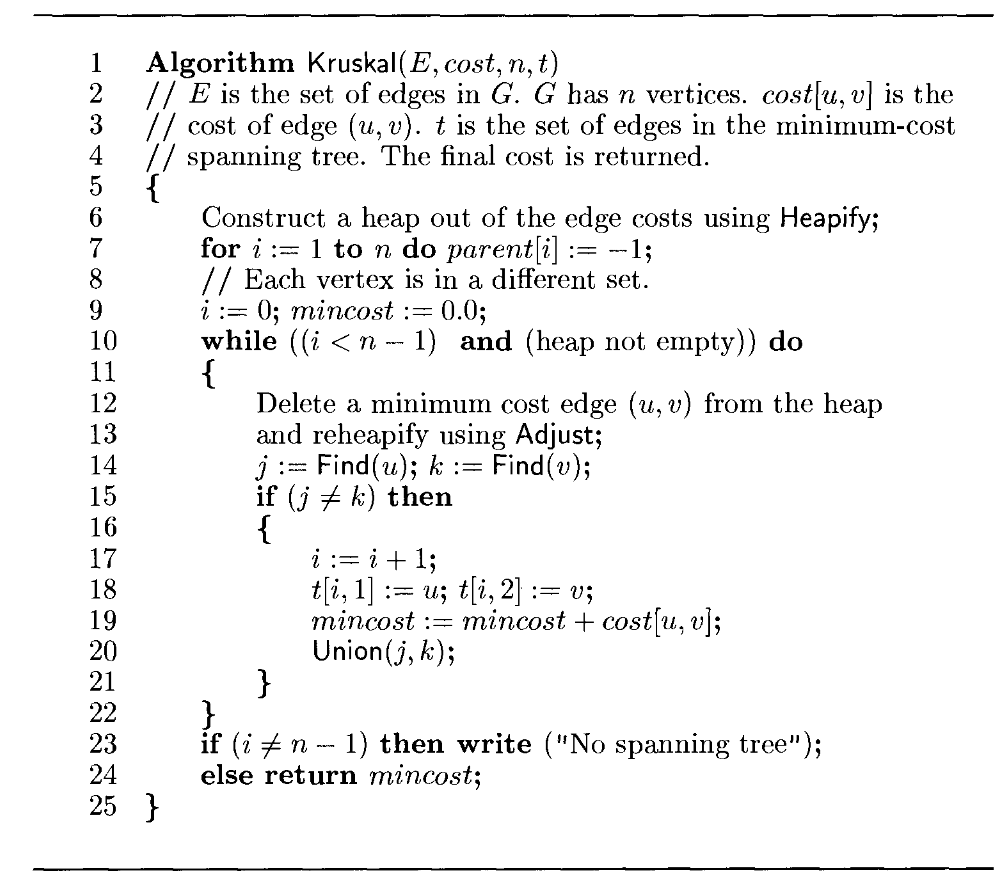
Date:

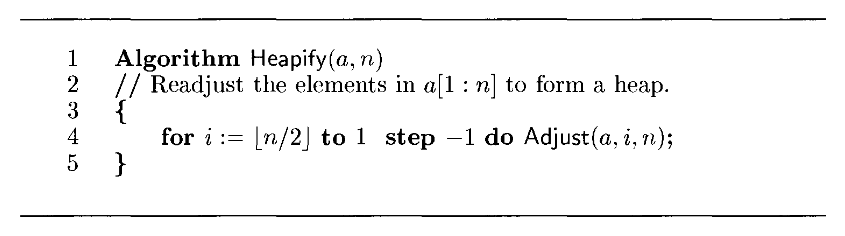
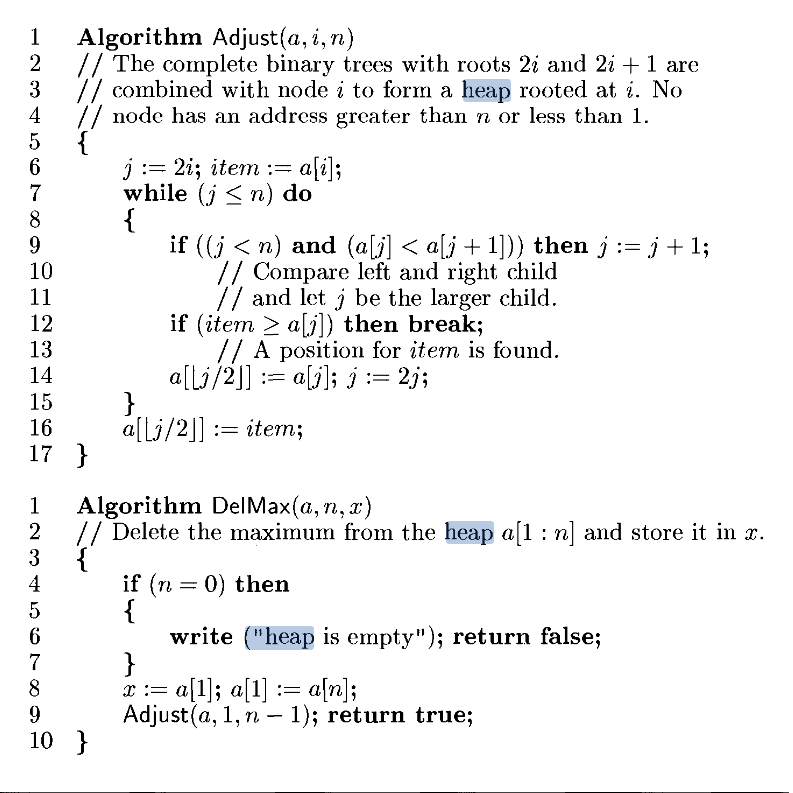
**Problem Statement:**

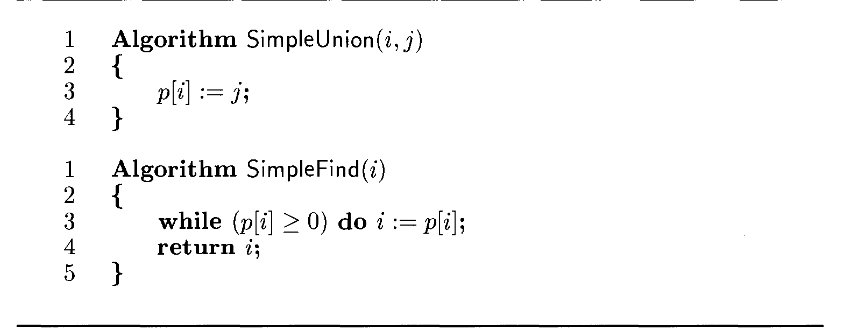


Write a c program to implement a minimum cost spanning tree using Kruskals algorithm on following graph:

**Algorithm**







**Time and Space Complexity :**

If the edges are maintained as a minheap, then the next

edge to consider can be obtained in 0(log |E|) time. The construction of the

heap itself takes 0(|E|) time. Since there are E edges, the time required to insert all the edges into the min-heap is O(E log E).

When edges are extracted from the min-heap, they are checked to see if they connect two trees. This can be done efficiently using the disjoint-set data structure, which maintains a set of trees and supports operations to find the root of a tree and to merge two trees. Finding the root of a tree takes O(log V) time, where V is the number of vertices in the graph. Merging two trees takes constant time. Since there are at most V trees in the forest, the total time required to process all edges is O(E log V).

Therefore, the overall time complexity of Kruskal's algorithm using a min-heap is O(E log E + E log V) = O(E log E). This is because the first term dominates the second term, as E is typically much larger than V in practice.

The space complexity of Kruskal's algorithm using a min-heap is O(E + V), where E is the number of edges and V is the number of vertices in the graph.The algorithm requires space to store the edges of the graph, which can be stored in an array or a linked list. The size of the array is proportional to the number of edges, which is E. Therefore, the space required to store the edges is O(E).

In addition to the edges, the algorithm also requires space to store the disjoint-set data structure. This can be implemented using an array or a linked list, where each element represents a tree in the forest. The size of the array or the linked list is proportional to the number of vertices, which is V. Therefore, the space required to store the disjoint-set data structure is O(V).

Finally, the min-heap data structure also requires space to store the edges that have not yet been added to the tree. The size of the min-heap is at most E, which is the number of edges in the graph. Therefore, the space required to store the min-heap is O(E).

Combining all these components, the space complexity of Kruskal's algorithm using a min-heap is O(E + V + E) = O(E + V).

**Code:**

#include <stdio.h>

int adj[100][100], cost[100][100];

struct EDGE

{

int origin, destin, edgecost;

};

void init\_adj(int n)

{

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

for (int j = 0; j < n; j++)

adj[i][j] = 0;

}

void init\_cost(int n)

{

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

for (int j = 0; j <= n; j++)

cost[i][j] = \_\_INT32\_MAX\_\_;

}

// int accept\_graph(FILE \*fp, int n)

int accept\_graph(int n)

{

int max\_edges = n \* (n - 1) / 2;

int origin, destin, edgecost;

int e = 0;

printf("Enter the edges of the graph and their respective cost.\n");

for (int i = 0; i < max\_edges; i++)

{

printf("Enter the edge,( 0 0 randval) to quit :");

scanf("%d %d %d", &origin, &destin, &edgecost);

// fscanf(fp, "%d %d %d", &origin, &destin, &edgecost);

if ((origin == 0) && (destin == 0))

break;

if (origin > n || destin > n || origin <= 0 || destin <= 0)

{

printf("Invalid edge.\n");

i--;

}

else

{

adj[origin - 1][destin - 1] = 1;

cost[origin - 1][destin - 1] = edgecost;

adj[destin - 1][origin - 1] = 1;

cost[destin - 1][origin - 1] = edgecost;

e++;

}

}

return e;

}

void display\_arr(int a[], int n)

{

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

printf("%d ", a[i] + 1);

printf("\n");

}

void swap(struct EDGE \*a, struct EDGE \*b)

{

struct EDGE temp = \*a;

\*a = \*b;

\*b = temp;

}

void heapify(struct EDGE adj[], int e, int i)

{

int smallest = i;

int l = 2 \* i + 1, r = 2 \* i + 2;

if (l < e && adj[l].edgecost <= adj[smallest].edgecost)

smallest = l;

if (r < e && adj[r].edgecost <= adj[smallest].edgecost)

smallest = r;

if (smallest != i)

{

swap(&adj[i], &adj[smallest]);

heapify(adj, e, smallest);

}

}

int Find(int parent[], int u)

{

while (parent[u] != -1)

u = parent[u];

return u;

}

void Union(int parent[], int u, int v)

{

parent[u] = v;

}

int kruskal(int n, int e, int t[][2])

{

struct EDGE heap[e];

int in = 0;

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

{

for (int j = 0; j < n; j++)

{

if (j > i && cost[i][j] != \_\_INT32\_MAX\_\_)

{

heap[in].origin = i;

heap[in].destin = j;

heap[in++].edgecost = cost[i][j];

}

}

}

for (int i = e / 2 - 1; i >= 0; i--)

heapify(heap, e, i);

int parent[n];

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

parent[i] = -1;

int i = 1, mincost = 0;

int u, v, j, k;

while (i < n)

{

swap(&heap[0], &heap[e - 1]);

struct EDGE minedge = heap[e - 1];

e--;

heapify(heap, e, 0);

u = minedge.origin;

v = minedge.destin;

j = Find(parent, u);

k = Find(parent, v);

if (j != k)

{

t[i][0] = u, t[i][1] = v, mincost += cost[u][v];

Union(parent, j, k);

printf("(%d, %d) ", u + 1, v + 1);

for (int l = 0; l < n; l++)

{

int val = 1;

if (parent[l] == -1)

val = 0;

printf("%d ", parent[l] + val);

}

printf("Cost: %d", mincost);

printf("\n");

i++;

}

}

if (i != n)

return -1;

else

return mincost;

}

int main()

{

int e, n;

// FILE \*fp;

// fp = fopen("graph1.txt", "r+");

printf("Enter the number of vertices of the graph.\n");

scanf("%d", &n);

int t[n][2];

init\_adj(n);

init\_cost(n);

// e = accept\_graph(fp, n);

// fclose(fp);

e = accept\_graph(n);

int mincost = kruskal(n, e, t);

if (mincost != -1)

printf("Mincost: %d\n", mincost);

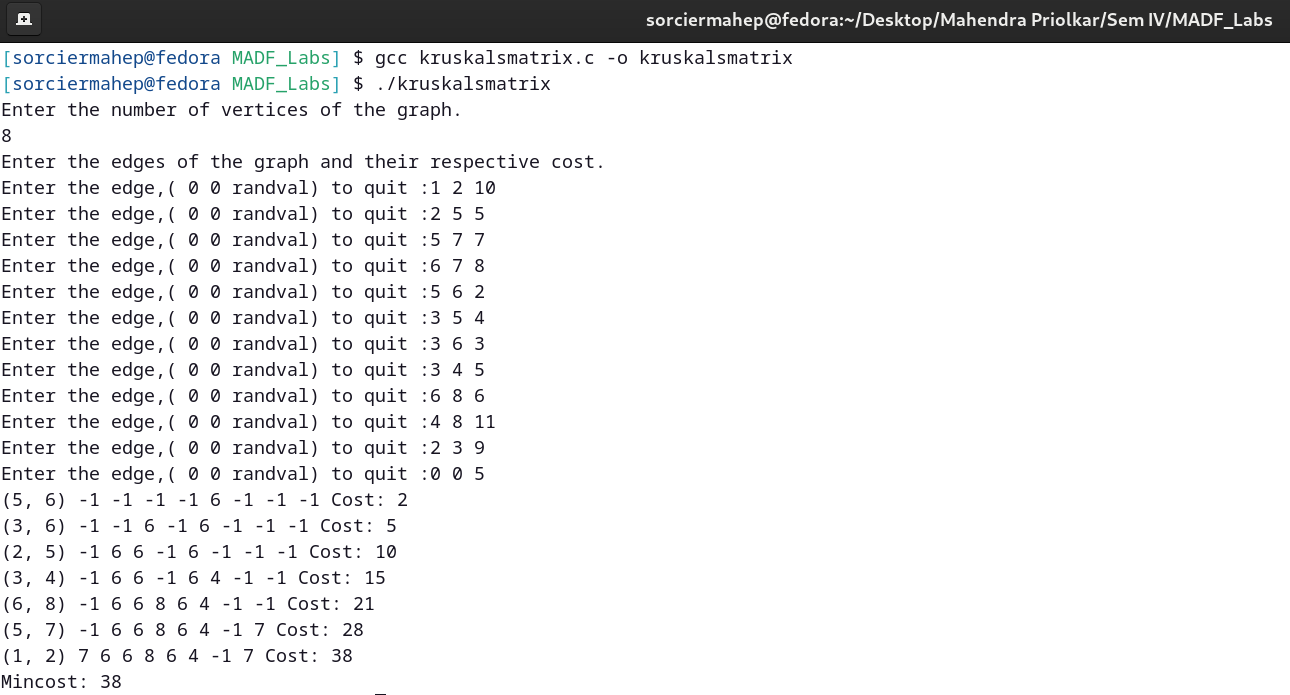
else

printf("No MST.\n");

return 0;

}

**Output:**

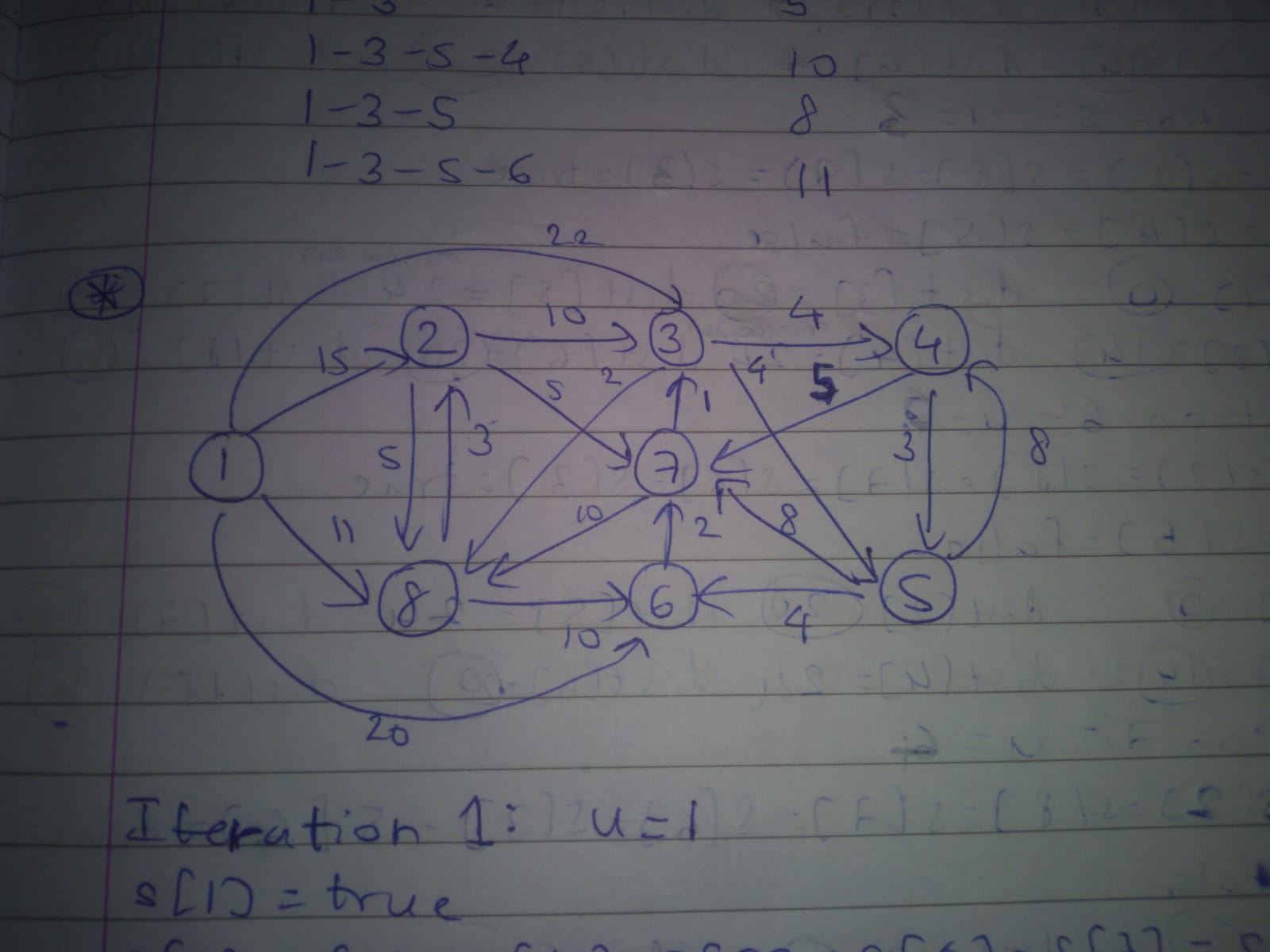


**c) Single source shortest path algorithm**

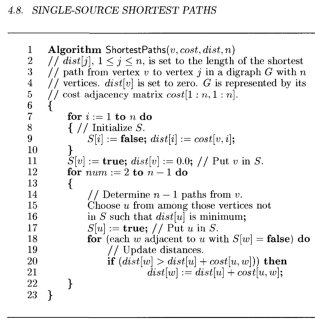
Date:

**Problem Statement:**

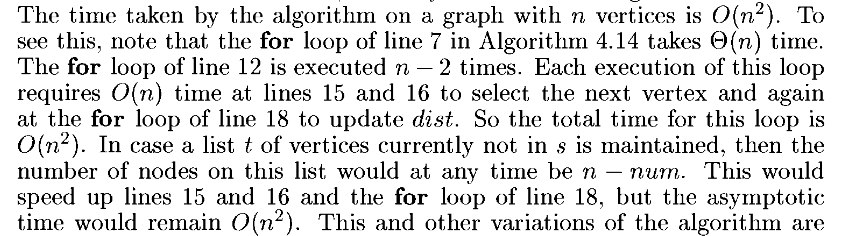
Write a c program to find shortest paths from source vertex (1) to other vertices in the following graph:



**Algorithm:**



**Time and Space Complexity:**



When an adjacency matrix is used to represent the graph in Dijkstra's algorithm, the space complexity is O(|V|^2), where |V| is the number of vertices in the graph.In addition to the space required for the adjacency matrix, Dijkstra's algorithm also requires space for the array used to store the vertices with the minimum distance,but this will be dominated by |V|^2.

**Code:**

#include <stdio.h>

#include <stdbool.h>

int adj[100][100], cost[100][100], path[100][100];

// void accept\_graph(FILE \*fp, int n)

void accept\_graph(int n)

{

int max\_edges = n \* (n - 1);

int origin, destin, edgecost;

printf("Enter the edges of the graph and their respective cost.\n");

for (int i = 0; i < max\_edges; i++)

{

printf("Enter the edge,( 0 0 randval) to quit :");

scanf("%d %d %d", &origin, &destin, &edgecost);

// fscanf(fp, "%d %d %d", &origin, &destin, &edgecost);

if ((origin == 0) && (destin == 0))

break;

if (origin > n || destin > n || origin <= 0 || destin <= 0)

{

printf("Invalid edge.\n");

i--;

}

else

{

adj[origin - 1][destin - 1] = 1;

cost[origin - 1][destin - 1] = edgecost;

}

}

}

void init\_cost(int n)

{

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

{

for (int j = 0; j < n; j++)

cost[i][j] = \_\_INT32\_MAX\_\_;

}

}

void init\_dist(int dist[], int n)

{

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

dist[i] = 0;

}

void init\_path(int n, int v)

{

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

for (int j = 0; j < 1; j++)

path[i][j] = v - 1;

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

for (int j = 1; j < 100; j++)

path[i][j] = -1;

}

void display(int arr[], int n)

{

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

{

if (arr[i] == \_\_INT32\_MAX\_\_)

printf("∞ ");

else

printf("%d ", arr[i]);

}

printf("\n");

}

void rec\_dijkstra(int v, int dist[], int n, int S[], int counter)

{

if (counter < n)

{

int min = \_\_INT32\_MAX\_\_, minj = -1;

for (int j = 0; j < n; j++)

{

if (S[j] != true && dist[j] < min) // Finds vertex corresponding to minimum distance

{

min = dist[j];

minj = j;

}

}

printf("u=%d\n", minj + 1);

S[minj] = true;

for (int k = 0; k < n; k++)

{

if (adj[minj][k] == 1 && S[k] == false)

{

if (dist[k] > (dist[minj] + cost[minj][k])) // Updates distances

{

int l;

dist[k] = dist[minj] + cost[minj][k];

for (l = 0; path[k][l] != -1; l++)

;

if (dist[minj] == cost[v][minj]) // Direct path

{

path[k][l] = path[k][l - 1];

path[k][l - 1] = minj;

}

else // Indirect path through other nodes

{

int m;

for (m = 1; path[minj][m] != -1; m++)

;

int o = l + m - 2;

path[k][o] = path[k][l - 1];

for (int p = m - 1; p != 0; p--)

{

path[k][--o] = path[minj][p];

}

for (int p = 0; path[k][p] != -1; p++) // Removing redundant nodes

{

for (int q = 1; path[k][q] != -1; q++)

{

int count = 0;

while (path[k][p] == path[k][q] && p != q)

{

count++;

p++;

q++;

}

p -= count;

q -= count;

if (count > 0)

{

int r;

for (r = p; path[k][r] != -1; r++)

path[k][r] = path[k][r + count];

for (int s = r; s < n; s++)

path[k][s] = -1;

}

}

}

}

}

}

}

display(dist, n);

rec\_dijkstra(v, dist, n, S, ++counter);

}

}

void init\_dijkstra(int v, int dist[], int n)

{

int S[n];

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

{

S[i] = false;

dist[i] = cost[v - 1][i];

}

S[v - 1] = true;

dist[v - 1] = 0;

printf("u=%d\n", v);

display(dist, n);

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

{

for (int j = 1; j < 2; j++)

{

path[i][j] = i;

}

}

rec\_dijkstra(v, dist, n, S, 1);

}

int main()

{

// FILE \*fp;

// fp = fopen("graph2.txt", "r+");

int n;

printf("Enter the number of vertices of the graph.\n");

scanf("%d", &n);

int dist[n];

init\_cost(n);

init\_dist(dist, n);

accept\_graph(n);

// accept\_graph(fp, n);

// fclose(fp);

int v;

printf("Enter the source vertex of the graph.\n");

scanf("%d", &v);

init\_path(n, v);

init\_dijkstra(v, dist, n);

printf("PATH%44cLENGTH\n", ' ');

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

{

int elems = 0;

if (dist[i] != \_\_INT32\_MAX\_\_ && dist[i] != 0)

{

for (int j = 0; path[i][j] != -1; j++, elems++)

printf("%d ", path[i][j] + 1);

printf("%\*d", 50 - (2 \* (elems - 1)), dist[i]);

printf("\n");

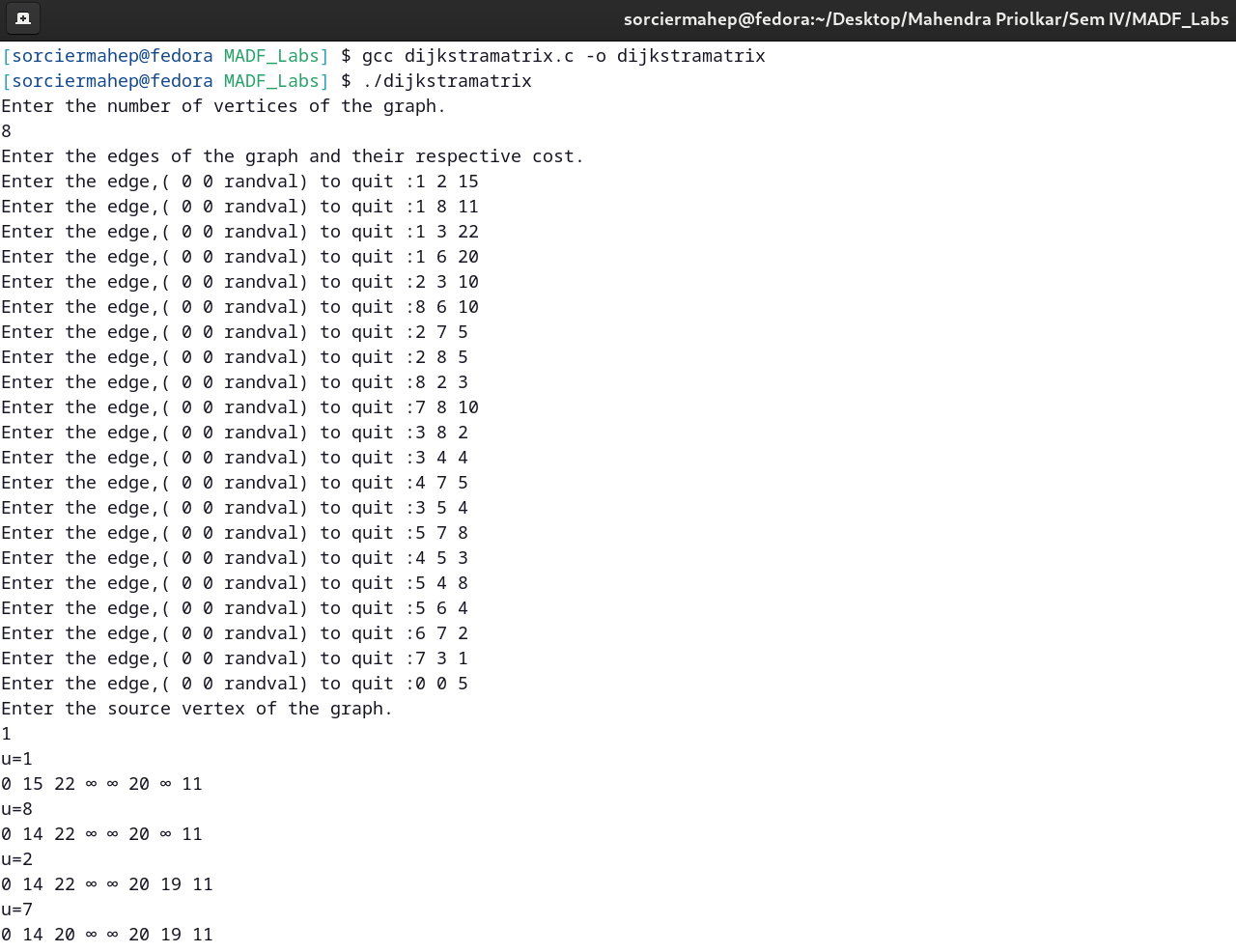
}

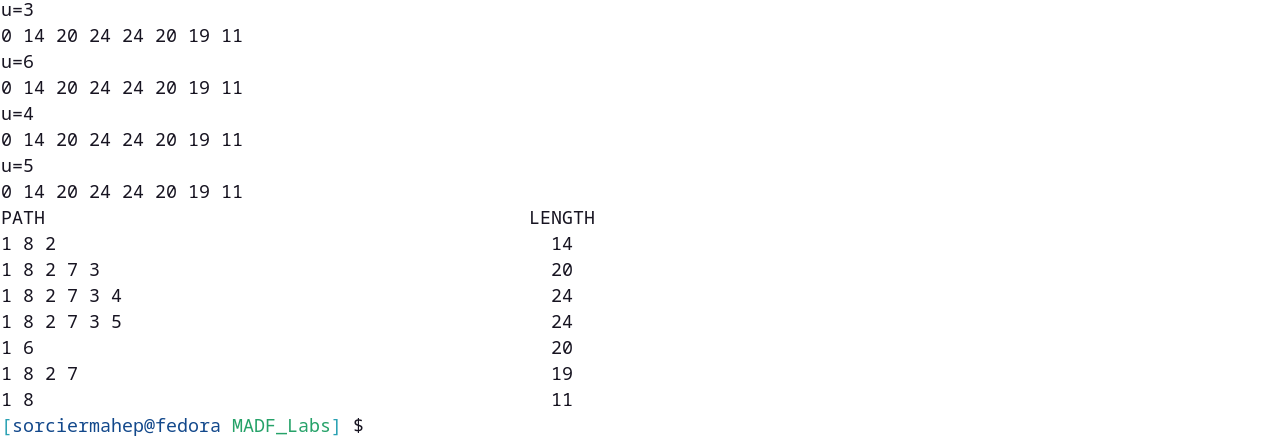
}

return 0;

}

**Output:**





**CONCLUSION:**

Greedy method strategy was studied. The programs for (a) Prims algorithm, (b) Kruskals algorithm, (c) Single source shortest path algorithm were studied and implemented successfully.