**Name: Gavali Deshabhakt Nagnath Data Structures and Algorithms**

**Roll No. 202CD005**

**Assignment-3**

**Index**

|  |  |  |
| --- | --- | --- |
| Sr. No. | Question | Page No. |
| 1 | Minimum spanning tree  (a) Prim’s Algorithm  (b) Kruskal’s Algorithm | 2  4 |
| 2 | Shortest path (between source vertex to all other vertices)  (a) Dijkstra's Algorithm  (b) Bellman-Ford’s Algorithm | 7  9 |
| 3 | Shortest path between any pair of vertices | 11 |
| 4 | Knapsack problem  (a) Divisible objects  (b) Indivisible objects | 13  15 |
| 5 | Transportation Problem | 17 |

**1. Minimum spanning tree: Prims and Kruskals Algorithm.**

**(a) Prim’s Algorithm**

**Program:**

#include <iostream>

#define I 32767 //setting I to Maximum value

using namespace std;

int main()

{

int cost[8][8] = {{I, I, I, I, I, I, I, I},

{I, I, 25, I, I, I, 5, I},

{I, 25, I, 12, I, I, I, 10},

{I, I, 12, I, 8, I, I, I},

{I, I, I, 8, I, 16, I, 14},

{I, I, I, I, 16, I, 20, 18},

{I, 5, I, I, I, 20, I, I},

{I, I, 10, I, 14, 18, I, I}};

int near[8] = {I, I, I, I, I, I, I, I};

int t[3][6];

int i,j,k,u,v,w,n=7,min=I; //n=7 because we're skipping first row and first column

/\*Finding first min wt edge\*/

for(i=1;i<=n;i++) //scanning upper triangular matrix for minimum weight edge

{

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

{

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

{

min=cost[i][j];

u=i;v=j;w=cost[i][j]; // storing co-ordinates of min wt edge in u,v

}

}

}

near[u]=near[v]=0; // marking corr near indices to 0 i.e marking edge as visited

t[0][0]=u;t[1][0]=v;t[2][0]=w; // storing u,v in t matrix

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

{

if(near[i]!=0) // check if node is not visited

{

if(cost[i][u]<cost[i][v]) // compare wt of near edges and update them in near array

near[i]=u;

else

near[i]=v;

}

}

/\* doing above procedure for all remaining edges \*/

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

{

min=I;

for(j=1;j<=n;j++)

{

if(near[j]!=0 && cost[j][near[j]]<min)

{

k=j;

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

}

}

t[0][i]=k;

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

t[2][i]=min;

near[k]=0;

for(j=1;j<=n;j++)

{

if(near[j]!=0 && cost[j][k]<cost[j][near[j]] )

{

near[j]=k;

}

}

}

cout << "\n\n\tPrim's Algorithm\n";

cout << "\nOutput is printed in following form: (start vertex, end vertex, weight)" << endl;

cout << "\nMinimum Spanning Tree using Prim's Algorithm:\n";

for (i = 0; i < n - 1; i++)

{

cout << "(" << t[0][i] << "," << t[1][i] << "," << t[2][i] <<")";

if(i!=n-2)

{

cout << " --> ";

}

}

int t\_wt=0;

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

{

t\_wt+=t[2][i];

}

cout << "\n\nTotal Minimum Weight: " << t\_wt << endl;

cout << endl;

}

**Output:**

Prim's Algorithm

Output is printed in following form: (start vertex, end vertex, weight)

Minimum Spanning Tree using Prim's Algorithm:

(1,6,5) --> (5,6,20) --> (4,5,16) --> (3,4,8) --> (2,3,12) --> (7,2,10)

Total Minimum Weight: 71

**Complexity:**

1. Time complexity of Prim’s Algorithm is O(n2) i.e. (O(V\*E)

2. If we use Heap to find minimum cost edge then we can reduce time complexity to O(nlogn).

**(b) Krushkal’s Algorithm**

**Program:**

#include<iostream>

using namespace std;

#define I 32767

int edge[9][3]={{1,2,15},{1,6,10},{2,5,25},{2,7,14},{3,4,22},

{4,5,52},{4,7,21},{3,6,25},{5,7,34}};

int s[8]={-1,-1,-1,-1,-1,-1,-1,-1};

int t[3][6];

int included[9]={0};

void unionfunc(int u,int v)

{

if(s[u]<s[v])

{

s[u]=s[u]+s[v];

s[v]=u;

}

else

{

s[v]=s[u]+s[v];

s[u]=v;

}

}

int find(int u)

{

int x=u,v=0;

while(s[x]>0)

{

x=s[x];

}

/\*connecting node to head node\*/

while(u!=x)

{

v=s[u];

s[u]=x;

u=v;

}

return x;

}

int main()

{

int u=0,v=0,i,j,k=0,min=I,n=7,e=9; //e-->number of edges

i=0;

while(i<n-1)

{

min=I;

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

{

if(included[j]==0 && edge[j][2]<min)

{

u=edge[j][0];

v=edge[j][1];

min=edge[j][2];

k=j;

}

}

if(find(u)!=find(v))

{

t[0][i]=u;

t[1][i]=v;

t[2][i]=min;

unionfunc(find(u),find(v));

i++;

}

included[k]=1;

}

cout << "\n\tKrushkal's Algorithm\n";

cout << "\nOutput is printed in following form: (start vertex, end vertex, weight)" << endl;

cout << "\nMinimum Spanning Tree using Krushkal's Algorithm:\n";

for (i = 0; i < n-1; i++)

{

cout << "(" << t[0][i] << "," << t[1][i] << "," << t[2][i] <<")";

if(i!=n-2)

{

cout << " --> ";

}

}

int t\_wt=0;

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

{

t\_wt+=t[2][i];

}

cout << "\n\nTotal Minimum Weight: " << t\_wt << endl;

cout << endl;

}

**Output:**

Krushkal's Algorithm

Output is printed in following form: (start vertex, end vertex, weight)

Minimum Spanning Tree using Krushkal's Algorithm:

(1,6,10) --> (2,7,14) --> (1,2,15) --> (4,7,21) --> (3,4,22) --> (2,5,25)

Total Minimum Weight: 107

**Complexity:**

1. Time complexity of Krushkal’s Algorithm is O(n2) i.e. (O(V\*E)

2. If we use Heap to find minimum cost edge then we can reduce time complexity to O(nlogn).

**2. Shortest path (between source vertex to all other vertices): Dijkstra's Algorithm and Bellman**

**Ford Algorithm**

**(a) Dijkstra’s Algorithm**

**Program:**

#include<iostream>

#include<limits.h>

#include<stdlib.h>

#include<stdio.h>

using namespace std;

#define N 9 //N--> number of vertices

// Function to find vertex with minimum distance from set of vertices not yet included

int minDist(int d[], int included[])

{

int min=INT\_MAX,minIndex;

for(int v=0;v<N;v++)

{

if(included[v]==0 && d[v]<=min)

{

min=d[v];

minIndex=v;

}

}

return minIndex;

}

void printSolution(int d[],int source)

{

cout << "Source: " << source << endl;

cout << "Vertex \t\tDistance from Source"<<endl;

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

if(i!=source)

cout << i <<"\t\t"<< d[i]<<endl;

}

void dijkstrasAlgorithm(int g[N][N],int source)

{

int d[N],included[N]={0};

//Initialize all distances as Infinite and included[v]=0

for(int v=0;v<N;v++)

{

d[v]=INT\_MAX;

included[v]=0;

}

d[source]=0;

// Finding shortest path for all vertices

for(int i=0;i<N-1;i++)

{

int min\_dist=minDist(d,included);

included[min\_dist]=1;

for(int v=0;v<N;v++)

{

if(included[v]==0 && g[min\_dist][v] && d[min\_dist]!=INT\_MAX && d[min\_dist]+ g[min\_dist][v] < d[v])

d[v] = d[min\_dist] + g[min\_dist][v];

}

}

printSolution(d,source);

}

int main()

{

int graph[N][N] = { { 0, 24, 0, 0, 0, 0, 0, 8, 0 },

{ 24, 0, 18, 0, 0, 0, 0, 11, 0 },

{ 0, 18, 0, 17, 0, 40, 0, 0, 20 },

{ 0, 0, 17, 0, 9, 2, 0, 0, 0 },

{ 0, 0, 0, 9, 0, 42, 0, 0, 0 },

{ 0, 0, 40, 2, 42, 0, 12, 0, 0 },

{ 0, 0, 0, 0, 0, 12, 0, 15, 36 },

{ 8, 11, 0, 0, 0, 0, 15, 0, 27 },

{ 0, 0, 20, 0, 0, 0, 36, 27, 0 } };

cout << "\n\tDijkstra's Algorithm Implementation\t\n\n";

dijkstrasAlgorithm(graph, 0);

}

**Output:**

Dijkstra's Algorithm Implementation

Source: 0

Vertex Distance from Source

1 19

2 54

3 37

4 46

5 35

6 23

7 8

8 35

**Complexity:**

The time complexity of Dijkstra’s Algorithm is O(n2).

**(b) Bellman-Ford’s Algorithm**

**Program:**

#include <iostream>

#include <limits.h>

using namespace std;

void printSolution(int d[], int V, int source)

{

cout << "Source: " << source << endl;

cout << "Vertex \t\tDistance from Source" << endl;

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

{

if (i != source)

{

if (d[i] < INT\_MAX)

cout << i << "\t\t" << d[i] << endl;

else if (d[i] > 32767) //INT\_MAX==32767 largest 16 bit integer value

cout << i << "\t\t"<< "INFINITE" << endl;

}

}

}

void BellmanFord(int edge[][3], int V, int E, int src)

{

int dist[V];

// Step 1: Initialize distances from src to all other vertices as INFINITE

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

dist[i] = INT\_MAX;

dist[src] = 0;

/\*Step 2: Relax all edges |V| - 1 times. A simple shortest path from src to any other vertex can have at-most |V| - 1 edges \*/

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

{

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

{

int u = edge[j][0];

int v = edge[j][1];

int weight = edge[j][2];

if (dist[u] != INT\_MAX && dist[u] + weight < dist[v])

dist[v] = dist[u] + weight;

}

}

/\*Step 3: check for negative-weight cycles. The above step guarantees shortest distances if graph doesn't contain

negative weight cycle. If we get a shorter path, then there is a cycle.\*/

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

{

int u = edge[j][0];

int v = edge[j][1];

int weight = edge[j][2];

if (dist[u] != INT\_MAX && dist[u] + weight < dist[v])

{

printf("Graph contains negative weight cycle");

return; // If negative cycle is detected, simply return

}

}

printSolution(dist, V, src);

return;

}

int main()

{

int V = 5; // Number of vertices in graph

int E = 7; // Number of edges in graph

int edge[][3] = {{0, 1, -1}, {0, 2, 4}, {1, 2, 3}, {1, 3, 2}, {1, 4, 2}, {3, 2, 5}, {3, 1, 1}, {4, 3, -3}};

cout << "\n\tImplementation of Bellman-Ford's Algorithm\t\n\n";

BellmanFord(edge, V, E, 3);

return 0;

}

**Output:**

Implementation of Bellman-Ford's Algorithm

Source: 3

Vertex Distance from Source

0 INFINITE

1 1

2 4

4 3

**Complexity:**

For a complete graph, the time complexity of Bellman-Ford Algorithm is ϴ(n3).

**3.** **Shortest path between any pair of vertices (Floyd-Warshall Algorithm).**

**Program:**

#include <bits/stdc++.h>

using namespace std;

#define I INT\_MAX

void flyodWarshall(vector<vector<int>> g)

{

int V = g.size();

vector<vector<int>> dist = g;

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

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

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

if (dist[i][k] != I && dist[k][j] != I && dist[i][k] + dist[k][j] < dist[i][j])

dist[i][j] = dist[i][k] + dist[k][j];

cout << endl;

cout << "The Shortest path between any vertices is:\n";

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

{

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

{

if (dist[i][j] == INT\_MAX)

{

cout << "INFINITE\t";

continue;

}

cout << dist[i][j] << "\t";

}

cout << endl;

}

}

int main()

{

int V;

cout << "\n\tShortest Path between Any Pair of Vertices\n\n";

cout << "Enter the number of vertex in the g: ";

cin >> V;

vector<vector<int>> g(V, vector<int>(V));

cout << "\nINFINITE: " << I << endl;

cout << "\nEnter the weights of the each Edges:\n";

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

{

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

{

cout << ">";

cin >> g[i][j];

}

}

flyodWarshall(g);

}

**Output:**

Shortest Path between Any Pair of Vertices

Enter the number of vertex in the g: 4

INFINITE: 2147483647

Enter the weights of the each Edges:

>0

>3

>2147483647

>7

>8

>0

>2

>2147483647

>5

>2147483647

>0

>1

>2

>2147483647

>2147483647

>0

The Shortest path between any vertices is:

0 3 5 6

5 0 2 3

3 6 0 1

2 5 7 0

**4. Solve Knapsack problem (for divisible and indivisible objects)**

**(a) Divisible Objects**

**Program:**

#include <bits/stdc++.h>

using namespace std;

struct Item

{

public:

int value, weight;

Item(int value, int weight): value(value),weight(weight){}

};

// comparator used to sort Item according to val/weight ratio

bool cmp(struct Item a, struct Item b)

{

double r1 = (double)a.value / (double)a.weight;

double r2 = (double)b.value / (double)b.weight;

return r1 > r2;

}

void fractionalKnapsack(int W, Item arr[], int n)

{

cout << "\nWeight Limit: " << W << endl << endl;

// sorting Item on basis of ratio

sort(arr, arr + n, cmp);

// Printing items added to knapsack along with their weights

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

{

cout << "Item " << i+1 <<"\tValue: " << arr[i].value << "\tWeight: " << arr[i].weight << "\tQuantity: "\

<< ((double)arr[i].value / arr[i].weight) << endl;

}

int curWeight = 0; // Current weight in knapsack

double finalvalue = 0.0; // Result (value in Knapsack)

// Looping through all Items

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

{

// If adding Item won't overflow, add it completely

if (curWeight + arr[i].weight <= W)

{

curWeight += arr[i].weight;

finalvalue += arr[i].value;

}

// If we can't add current Item, add fractional part of it

else

{

int remain = W - curWeight;

finalvalue

+= arr[i].value

\* ((double)remain / (double)arr[i].weight);

break;

}

}

cout << "\nMaximum Value that can be obtained for weight " << W << " is " << finalvalue << endl;

}

int main()

{

int W = 100; // Weight of knapsack

Item arr[] = { { 60, 10 }, { 100, 20 }, { 120, 30 }, { 10, 25}, { 55, 22} };

int n = sizeof(arr) / sizeof(arr[0]);

cout <<"\n\tKnapsack Problem Implementation\n\n";

cout <<"\t\t(Divisible Objects)\n\n";

fractionalKnapsack(W, arr, n);

return 0;

}

**Output:**

Knapsack Problem Implementation

(Divisible Objects)

Weight Limit: 100

Item 1 Value: 60 Weight: 10 Quantity: 6

Item 2 Value: 100 Weight: 20 Quantity: 5

Item 3 Value: 120 Weight: 30 Quantity: 4

Item 4 Value: 55 Weight: 22 Quantity: 2.5

Item 5 Value: 10 Weight: 25 Quantity: 0.4

Maximum Value that can be obtained for weight 100 is 342.2

**Complexity:**

As Sorting takes maximum of time, so the time complexity of Knapsack algorithm for divisible objects is O(nlogn).

**(b) Indivisible Objects**

**Program:**

#include <iostream>

using namespace std;

int knapSackRec(int W, int wt[],int val[], int i,int \*\*dp)

{

if (i < 0)

return 0;

if (dp[i][W] != -1)

return dp[i][W];

if (wt[i] > W)

{

dp[i][W] = knapSackRec(W, wt,val, i - 1,dp);

return dp[i][W];

}

else

{

dp[i][W] = max(val[i] + knapSackRec(W - wt[i], wt, val, i - 1, dp), knapSackRec(W, wt, val, i - 1, dp));

return dp[i][W];

}

}

int knapSack(int W, int wt[], int val[], int n)

{

int \*\*dp;

dp = new int \*[n];

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

dp[i] = new int[W + 1];

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

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

dp[i][j] = -1;

return knapSackRec(W, wt, val, n - 1, dp);

}

int main()

{

int val[] = {10, 20, 30};

int wt[] = {10, 18, 20};

int W ;

int n = sizeof(val) / sizeof(val[0]);

cout <<"\nKnapsack Problem Implementation"<<

"\n (Indivisible Objects)\n";

cout << "\nValues: ";

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

{

cout << " " << val[i];

}

cout << "\nWeights:";

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

{

cout << " " << wt[i];

}

cout << "\nEnter Weight limit:\n>";

cin >> W;

cout << "\nWeight limit: " << W << endl;

cout << "Max Value that can be obtained for given weight limit: " << knapSack(W, wt, val, n) << endl;

return 0;

}

**Output:**

Knapsack Problem Implementation

(Indivisible Objects)

Values: 10 20 30

Weights: 10 18 20

Enter Weight limit:

>35

Weight limit: 35

Max Value that can be obtained for given weight limit: 40

**Complexity:**

Time complexity is O(N\*W) i.e. O(n2).

**5. Solve the transportation Problem**

**Program:**

#include <iostream>

using namespace std;

int main()

{

int flag = 0, flag1 = 0;

int s[10], d[10], sn, eop = 1, dm, a[10][10];

int i, j, sum = 0, min, x[10][10], k, fa, fb;

/\* Getting The Input For the Problem\*/

cout << "\n\t Transporation Problem \t\n" <<endl;

cout << "Enter the number of Sources:\n>";

cin >> sn;

cout << "\nEnter the number of Destinations\n>";

cin >> dm;

cout << "\nEnter the Supply Values:";

for (i = 0; i < sn; i++)

{

cout << "\nSource " << (i+1) << ": ";

cin >> s[i];

}

cout << "\nEnter the Demand Values: ";

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

{

cout << "\nDestination " << (j+1) <<": ";

cin >> d[j];

}

cout << "\nEnter costs row wise :\n";

for (i = 0; i < sn; i++)

{

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

{

cout << ">";

cin >> a[i][j];

}

}

/\* Calculation For the Transportation \*/

i = 0;

j = 0;

for (i = 0, j = 0; i < sn, j < dm;)

{

if (s[i] < d[j]) // Check supply less than demand

{

x[i][j] = a[i][j] \* s[i]; // Calculate amount \* supply

d[j] = d[j] - s[i]; // Calculate demand - supply

i++; // Increment i for the deletion of the row or column

}

else if (s[i] >= d[j]) //Check the supply greater than equal to demand

{

x[i][j] = a[i][j] \* d[j]; // Calculate amount \* demand

s[i] = s[i] - d[j]; // Calculate supply - demand

j++; // Increment j for the deletion of the row or column

}

}

cout << "\nGiven Cost Matrix is :\n";

for (fa = 0; fa < sn; fa++)

{

for (fb = 0; fb < dm; fb++)

{

cout << a[fa][fb] << "\t";

}

cout << endl;

}

cout << "\nAllocated Cost Matrix is \n";

for (fa = 0; fa < sn; fa++)

{

for (fb = 0; fb < dm; fb++)

{

if(x[fa][fb]!=0)

{

cout << x[fa][fb] << "\t";

}

else

{

cout <<"\t";

}

sum = sum + x[fa][fb];

}

cout << endl;

}

cout << "\nTransportation cost: " << sum << endl<<endl;

}

**Output:**

Transporation Problem

Enter the number of Sources:

>4

Enter the number of Destinations

>4

Enter the Supply Values:

Source 1: 10

Source 2: 20

Source 3: 30

Source 4: 40

Enter the Demand Values:

Destination 1: 40

Destination 2: 30

Destination 3: 20

Destination 4: 10

Enter costs row wise :

>1

>2

>3

>7

>9

>3

>4

>6

>8

>1

>5

>4

>6

>9

>8

>4

Given Cost Matrix is :

1 2 3 7

9 3 4 6

8 1 5 4

6 9 8 4

Allocated Cost Matrix is

10

180

80 20

90 160 40

The Transportation cost: 580

==================================================================================================================================================

**THE END**