**II: The Greedy Method Date:**

**Aim:-**Write algorithm and C program to implement the following problems using the greedy method

a. Prim’s Algorithm

b. Kruskal’s Algorithm

c. Single Source Shortest paths

**THEORY:**

The greedy method suggests that one can devise an algorithm that works in stages,

considering one input at a time. At each stage, a decision is made regarding whether a

particular input is an optimal solution. This is done by considering the inputs in an order

determined by some selection procedure. If the inclusion of the next input into the partially

constructed optimal solution will result in an infeasible solution, then this input is not added

to the partial solution. Otherwise, it is added. The selection procedure itself is based on some

optimization measure. This measure may be the objective function. In fact, several different

optimization measures may be plausible for a given problem. Most of these, however, will

result in algorithms that generates unoptimal solutions. This version of the greedy techniques

called the subset paradigm

The function Select selects an input from a[ ] and removes it. The selected input's value is

assigned to x. Feasible is a Boolean-valued function that determines whether x can be

included into the solution vector. The function Union combines x with the solution and

updates the objective function. The function Greedy describes the essential way that a greedy

algorithm will look, once a particular problem is chosen and the functions Select, Feasible,

and Union are properly implemented.

1 **Algorithm** **Greedy**(a, n)

2 // a[l : n] contains the n inputs.

3 {

4 solution:=0; // Initialize the solution.

5 for i :=1to n do

6 {

7 x :=Select(a);

8 if Feasible(solution, x) then

9 solution:= Union (solution, x);

10 }

11 return solution;

12 }

**Greedy Algorithms:**

**Definition:**

A greedy algorithm is a problem-solving approach that makes locally optimal choices at each step with the aim of finding a global optimum solution. It selects the best immediate option without considering future consequences. Greedy algorithms are characterized by their greedy choice property, optimal substructure, and lack of backtracking.

**Features:**

1. Greedy Choice Property: Locally optimal choices are made at each step without considering future consequences.

2. Optimal Substructure: The optimal solution to the problem can be constructed from optimal solutions to its subproblems.

3. No Backtracking: Decisions are made once and not revisited or revised during the algorithm's execution.

**Advantages:**

1. Efficiency: Greedy algorithms are simple and efficient to implement, often involving straightforward local decisions.

2. Quick Solutions: They can find solutions rapidly, particularly for problems with numerous potential solutions.

3. Space Efficiency: Greedy algorithms typically require less memory space compared to other techniques due to their localized decision-making.

**Disadvantages:**

1. Suboptimal Solutions: Greedy algorithms may not always produce the optimal solution, potentially leading to suboptimal outcomes.

2. Risk of Local Optima: They can get stuck in local optima, especially if the problem space lacks the greedy choice property or optimal substructure.

3. Correctness: Without careful analysis, there's no guarantee of correctness, as certain cases may lead to incorrect solutions.

**Applications:**

1. Minimum Spanning Trees: Prim's and Kruskal's algorithms use greedy strategies to find the minimum spanning tree of a graph by adding minimum-weight edges iteratively.

2. Shortest Path Algorithms: Dijkstra's algorithm, for example, employs a greedy approach by selecting the shortest path from the source node iteratively.

3. Job Scheduling: Greedy algorithms can prioritize tasks based on criteria like earliest deadline first (EDF) or shortest processing time (SPT).

4. Fractional Knapsack Problem: They are useful in solving the fractional knapsack problem, maximizing total value within weight constraints.

5. Huffman Coding: Used in lossless data compression, Huffman coding constructs a variable-length prefix coding scheme based on character frequencies, using a greedy algorithm to build an optimal prefix-free binary tree.

Greedy algorithms find applications in diverse problem domains. However, their suitability depends on the problem characteristics, and thorough analysis is essential to ensure correctness and optimality. While they offer efficiency and quick solutions, caution must be exercised to address their limitations and validate their outcomes carefully.

1. **Prims**

#include<stdio.h>

#include<stdlib.h>

#include<limits.h>

void MinCostEdge(int \*\*cost, int n, int \*x, int \*y)

{

    int smallest = cost[0][0];

    \*x = \*y = 0;

    for(int i = 0; i < n; i++)

    {

        for(int j = i; j < n ;j++)

        {

            if(cost[i][j] < smallest)

            {

                smallest = cost[i][j];

                \*x = i; \*y = j;

            }

        }

    }

}

int NearestEdge(int \*near,int \*\*cost, int n)

{

    int nearest = -1,minCost = INT\_MAX;

    for(int i = 0; i < n; i++)

    {

        if(cost[i][near[i]] < minCost && near[i] != -1)

        {

            nearest = i;

            minCost = cost[i][near[i]];

        }

    }

    return nearest;

}

void PrintMCSTree(int \*\*MCSTree,int\*\*cost,int n)

{

    printf("\n    src   dest   cost\n");

    for(int i = 0; i < n; i++)

    {

        printf("%d]  ",i+1);

        printf("%d     ",MCSTree[i][0]+1);

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

        printf("%d\n",cost[MCSTree[i][0]][MCSTree[i][1]]);

    }

}

void PrintNearCost(int \*near, int \*\*cost, int n)

{

    printf("\nnear  ");

    for(int i= 0; i<n; i++)

    {

        if(near[i] == -1)

            printf("  0  ");

        else

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

    }

    printf("\ncost  ");

    for(int i= 0; i<n; i++)

    {

        if(cost[i][near[i]] == INT\_MAX)

            printf(" INF ");

        else if(near[i] == -1)

            printf(" --- ");

        else

            printf("  %d  ",cost[i][near[i]]);

    }printf("\n");

}

int Prims(int \*\*edge, int \*\*cost, int \*\*MCSTree, int n)

{

    int minCost = 0;

    int x,y;

    int \*near;

    near = malloc(n\*sizeof(int));

    MinCostEdge(cost,n,&x,&y);

    minCost = cost[x][y];

    MCSTree[0][0] = x;

    MCSTree[0][1] = y;

    for(int i = 0; i < n; i++)

    {

        if(cost[i][x] < cost[i][y])

            near[i] = x;

        else

            near[i] = y;

    }

    near[x] = near[y] = -1;

    printf("\nindex ");

    for(int i = 0; i <n; i++)

    {

        printf(" [%d] ",i+1);

    }printf("\n");

    PrintNearCost(near,cost,n);

    PrintMCSTree(MCSTree,cost,1);

    for(int i = 1; i < n-1; i++)

    {

        int minEdge = NearestEdge(near,cost,n);

        MCSTree[i][0] = minEdge;

        MCSTree[i][1] = near[minEdge];

        minCost = (minCost + cost[minEdge][near[minEdge]]);

        near[minEdge] = -1;

        for(int j = 0; j < n; j ++)

        {

            if(near[j] != -1 && (cost[j][near[j]] > cost[j][minEdge]))

                near[j] = minEdge;

        }

        PrintNearCost(near,cost,n);

        PrintMCSTree(MCSTree,cost,i+1);

    }

    return minCost;

}

void CreateWeightedGraph(int \*\*edge, int \*\*cost, int n)

{

    for(int i = 0; i < n; i++)

    {

        for(int j = 0; j < n; j++)

        {

            edge[i][j] = 0;

            cost[i][j] = INT\_MAX;

        }

    }

    int x,y,weight;

    while(1)

    {

        printf("Enter edge (x, y) Enter (-1 -1) to exit: \n");

        scanf("%d%d",&x,&y);

        if(--x >= n || --y >= n)

        {

            printf("Node exceeds size limit\nEnter again\n");

        }

        else if(x < 0 && y < 0 )

        {

            break;

        }

        else{

            if(edge[x][y] == 1)

            {

                printf("An edge already exists at (%d , %d)",x,y);

                continue;

            }

            printf("Enter Weight(cost) of edge : ");

            scanf("%d",&weight);

            edge[x][y] = 1;

            edge[y][x] = 1;

            cost[x][y] = weight;

            cost[y][x] = weight;

        }

    }

}

int main()

{

    int n;

    printf("Enter number of nodes : ");

    scanf("%d",&n);

    int \*\*edge, \*\*cost, \*\*MCSTree;

    edge = malloc(n\*sizeof(\*edge));

    cost = malloc(n \* sizeof(\*cost));

    MCSTree = malloc(n \* sizeof(\*MCSTree));

    for(int i =0; i < n; i++)

    {

        edge[i] = malloc(n\* sizeof(\*edge[i]));

        cost[i] = malloc(n\* sizeof(\*cost[i]));

        MCSTree[i] = malloc(2\* sizeof(\*edge[i]));

    }

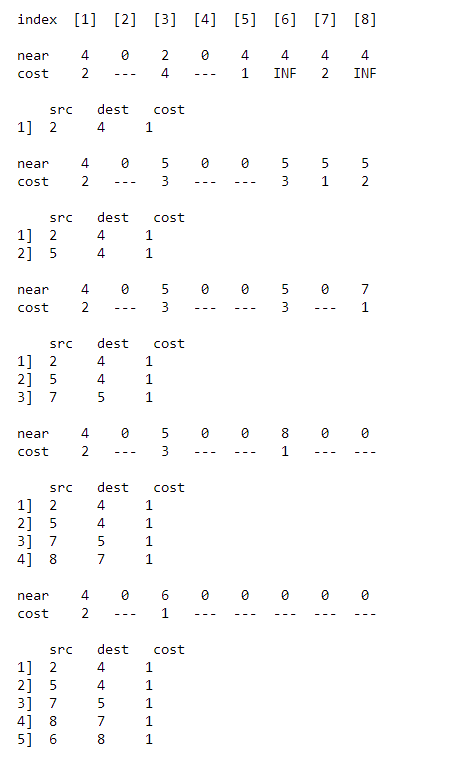
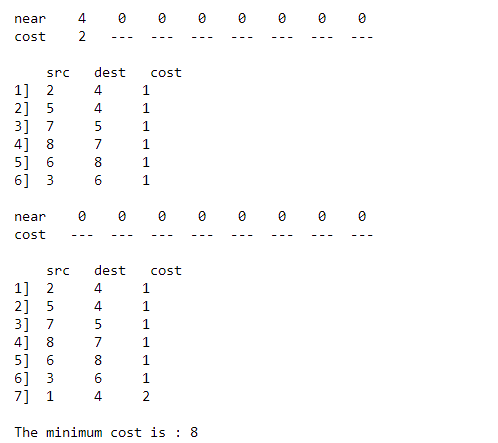
    CreateWeightedGraph(edge,cost,n);

    int minCost = Prims(edge,cost,MCSTree,n);

    printf("\nThe minimum cost is : %d\n",minCost);

}

**Output**

****

1. **Kruskals**

#include<stdio.h>

#include<stdlib.h>

#include<limits.h>

#include<math.h>

typedef struct node

{

    int x, y, cost;

} unit;

void PrintHeap(unit \*heap, int n)

{

    for(int i =0; i<n; i++)

    {

        printf("\nx : %d  y : %d  cost : %d",heap[i].x,heap[i].y,heap[i].cost);

    }

}

void PrintParent(int\* parent, int n, unit min, int x)

{

    printf("\nstep%d (%d,%d) ",x,min.x,min.y);

    for(int i = 0; i<n; i++)

    {

        if(parent[i] < 0)

        printf("%d   ",parent[i]);

        else

        printf(" %d   ",parent[i]);

    }

}

void PrintMCSTree(unit \*MCSTree, int n)

{

    printf("\n\nsrc dest  cost\n");

    for(int i =0; i<n; i++)

    {

        printf("%d   %d  :  %d\n",MCSTree[i].x,MCSTree[i].y,MCSTree[i].cost);

    }

}

int InsertinHeap(unit \*heap, int n)

{

    int i = n;

    unit item = heap[n];

    while((i > 0) && (heap[i/2].cost > item.cost))

    {

        heap[i] = heap[i/2];

        i = i/2;

    }

    heap[i] = item;

    return 1;

}

void CreateHeap(unit \*heap, int n, int n1)

{

    int x, y, cost, pos = 0;

    while(pos < n)

    {

        printf("\nEnter an Edge (x y) and its cost, (-1 -1 to exit) : \n");

        scanf("%d%d%d",&x,&y,&cost);

        if(x < 0 && y < 0)

            break;

        else if(x < 0 || y < 0 || x >n1 || y>n1)

        {

            printf("\nEnter a valid edge");

        }

        else{

            heap[pos].x = x;

            heap[pos].y = y;

            heap[pos].cost = cost;

            InsertinHeap(heap, pos);

            pos++;

            //PrintHeap(heap,pos);

        }

    }

}

void Adjust(unit \*heap, int i, int n)

{

    int j = (2 \* i) + 1;

    unit item = heap[i];

    while(j < n)

    {

        if(j+1 < n && heap[j].cost > heap[j+1].cost)

            j++;

        if(item.cost <= heap[j].cost)

            break;

        heap[(int)ceil(((float)j)/2)-1] = heap[j];

        j = (j\*2)+1;

    }

    heap[(int)ceil((float)j/2)-1] = item;

}

void Heapify(unit \*heap, int n)

{

    for(int i =(n/2)-1; i > 0; i--)

    {

        Adjust(heap,i,n);

    }

}

void DelMin(unit \*heap, int n, unit \* item)

{

        if(n < 0)

        {

            printf("\nThe Heap is Empty !");

            return;

        }

    \*item = heap[0];

    heap[0] = heap[n-1];

    Adjust(heap,0,n-1);

}

int Find(int i, int\* parent)

{

    while(parent[i] >= 0)

        i = parent[i];

    return i;

}

void Union(int i, int j, int \*parent)

{

    parent[i] = j;

}

int Kruskals(unit \*heap, int nVer,int nEdges, unit \*MCSTree)

{

    int \*parent = malloc(sizeof(int) \* nVer);

    for(int i = 0; i<nVer; i++)

    {

        parent[i] = -1;

    }

    int i = 0, j, k, nEdgeCount = nEdges;

    int minCost = 0;

    unit min;

    printf("\n\n\t   ");

    for(int c = 0; c<nVer; c++)

        printf(" [%d] ",c);

    printf("\nparent\t   ");

    for(int c = 0; c<nVer; c++)

        printf(" %d  ",parent[c]);

    while(i < nVer - 1 && i < nEdges)

    {

        DelMin(heap,nEdgeCount--,&min);

        Heapify(heap,nEdgeCount);

        j = Find(min.x-1, parent);

        k = Find(min.y-1,parent);

        if(j != k)

        {

            MCSTree[i] = min;

            i++;

            minCost += min.cost;

            Union(j,k,parent);

            PrintParent(parent,nVer,min, i);

        }

    }

    PrintMCSTree(MCSTree,nVer-1);

    if(i != nVer-1)

        printf("\nNo Spanning Tree");

    else return minCost;

    return -1;

}

int main()

{

    int nEdges, nVert;

    printf("Enter number of edges : ");

    scanf("%d",&nEdges);

    printf("Enter number of vertices : ");

    scanf("%d",&nVert);

    unit \*heap = malloc(sizeof(unit)\*nEdges);

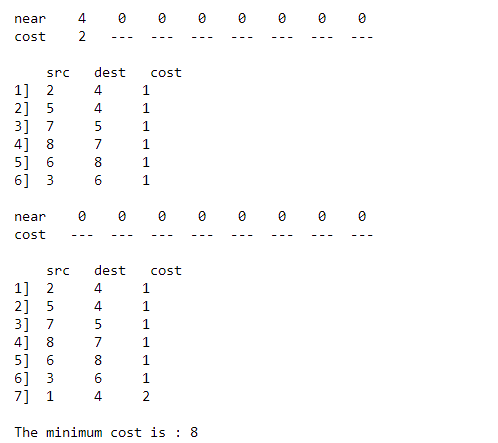
    unit \*MCSTree = malloc(sizeof(unit)\*(nVert-1));

    CreateHeap(heap,nEdges,nVert);

    int minCost = Kruskals(heap,nVert,nEdges,MCSTree);

    printf("\nThe minimum cost is : %d\n",minCost);

}

**Output:**

1. **Single Source Shortest Path**

#include<stdio.h>

#include<stdlib.h>

#include<math.h>

#include<limits.h>

#define false 0

#define true 1

int iter = 0;

void CreateWeightedGraph(int \*\*cost, int n)

{

    for(int i = 0; i < n; i++)

    {

        for(int j = 0; j < n; j++)

        {

            cost[i][j] = INT\_MAX;

        }

    }

    int x,y,weight;

    while(1)

    {

        printf("Enter edge (x, y) Enter (-1 -1) to exit: \n");

        scanf("%d%d",&x,&y);

        if(--x >= n || --y >= n)

        {

            printf("Node exceeds size limit\nEnter again\n");

        }

        else if(x < 0 && y < 0 )

        {

            break;

        }

        else{

            if(cost[x][y] != INT\_MAX)

            {

                printf("An edge already exists at (%d , %d)",x,y);

                continue;

            }

            printf("Enter Weight(cost) of edge : ");

            scanf("%d",&weight);

            cost[x][y] = weight;

        }

    }

}

void Print(int \*dist, int \*S, int n, int u)

{

    if(iter == 0)

    {

        printf("\nIteration   Vertex                Distance");

        printf("\n            Selected   ");

        for(int i=1; i<=n; i++)

            printf("[%d]  ",i);

        printf("\nInitial         --     ");

        for(int i=0; i<n; i++){

            if(dist[i] == INT\_MAX)

                printf("INF  ");

            else

                printf("%3d  ",dist[i]);

        }

    }

    else

    {

        printf("\n%7d      \t %d   ",iter,u+1);

        for(int i=0; i<n; i++){

            if(dist[i] == INT\_MAX)

                printf("  INF");

            else

                printf("%5d",dist[i]);

        }

        printf("  S : {");

        for(int j=0; j<n; j++)

            if(S[j] == true)

                printf("%d,",j+1);

        printf("}");

    }

}

int NearestVertex(int \*S, int \*dist,int n)

{

    int shortestDistance = INT\_MAX;

    int shortest = -1;

    for(int i = 0; i<n; i++)

    {

        if(S[i] == true)

            continue;

        else{

            if(dist[i] < shortestDistance)

            {

                shortestDistance = dist[i];

                shortest = i;

            }

        }

    }

    return shortest;

}

void PrintPath(int \*pred, int j)

{

    if(j == 0)

        printf("1");

    else

    {

        PrintPath(pred,pred[j]);

        printf("->%d",j+1);

    }

}

void PrintPaths(int\*dist, int\*pred, int n)

{

    printf("src  dest  path\n");

    for(int i=0; i<n; i++)

    {

        printf(" 1    %d    ",i+1);

        PrintPath(pred,i);

        printf(" = %d\n",dist[i]);

    }

}

void SingleSourceShortestPath(int \*\*cost, int n, int\* dist, int source,int \*pred)

{

    int \*S = malloc(sizeof(int)\*n);

    int shortest;

    for(int i=0; i<n; i++)

    {

        S[i] = false;

        dist[i] = cost[source][i];

    }

    S[source] = true;

    dist[source] = 0;

    pred[0] = 0;

    Print(dist,S,n,0);

    for(int i = 1; i<n-1; i++)

    {

        shortest = NearestVertex(S,dist,n);

        S[shortest] = true;

        for(int j=0; j<n; j++)

        {

            if(cost[shortest][j] != INT\_MAX && S[j] == false)

            {

                if (dist[j] > dist[shortest] + cost[shortest][j])

                {

                    dist[j] = dist[shortest] + cost[shortest][j];

                    pred[j] = shortest;

                }

            }

        }

        iter++;

        Print(dist,S,n,shortest);

    }printf("\n");

    PrintPaths(dist,pred,n);

}

int main()

{

    int n;

    int\*\* cost, \*dist, \*pred;

    printf("Enter the number of vertices in the graph : ");

    scanf("%d",&n);

    cost = malloc(sizeof(\*cost)\*n);

    dist = malloc(sizeof(int)\*n);

    pred = malloc(sizeof(\*pred)\*n);

    for(int i=0; i<n; i++)

        cost[i]=malloc(sizeof(\*cost[i])\*n);

    CreateWeightedGraph(cost,n);

    for (int i = 1; i <= n; i++)

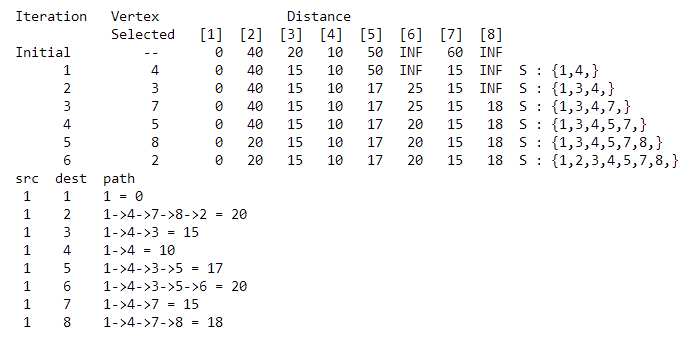
        pred[i] = 0;

    SingleSourceShortestPath(cost,n,dist,0,pred);

    printf("\n");

    return 0;

}

**Output**