7. Single source shortest paths using greedy method when the graph is represented by adjacency matrix using C.

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

void dijkstra(int G[MAX][MAX],int n,int startnode);

int main()
{
   int G[MAX][MAX],i,j,n,u;
   printf("Enter no. of vertices:");
   scanf("%d",&n);
   printf("\nEnter the adjacency matrix:\n");

   for(i=0;i<n;i++)
      for(j=0;j<n;j++)
         scanf("%d",&G[i][j]);

   printf("\nEnter the starting node:");
   scanf("%d",&u);
   dijkstra(G,n,u);

   return 0;
}

void dijkstra(int G[MAX][MAX],int n,int startnode)
{

   int cost[MAX][MAX],distance[MAX],pred[MAX];
   int visited[MAX],count,mindistance,nextnode,i,j;
```

```
   //pred[] stores the predecessor of each node
   //count gives the number of nodes seen so far
   //create the cost matrix
   for(i=0;i<n;i++)
      for(j=0;j<n;j++)
         if(G[i][j]==0)
            cost[i][j]=INFINITY;
         else
            cost[i][j]=G[i][j];

   //initialize pred[],distance[] and visited[]
   for(i=0;i<n;i++)
   {
      distance[i]=cost[startnode][i];
      pred[i]=startnode;
      visited[i]=0;
   }

   distance[startnode]=0;
   visited[startnode]=1;#include<stdio.h>
#define INFINITY 9999
#define MAX 10

void dijkstra(int G[MAX][MAX],int n,int startnode);

int main()
{
   int G[MAX][MAX],i,j,n,u;
   printf("Enter no. of vertices:");
   scanf("%d",&n);
```

```
    printf("\nEnter the adjacency matrix:\n");

    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            scanf("%d",&G[i][j]);

    printf("\nEnter the starting node:");
    scanf("%d",&u);
    dijkstra(G,n,u);

    return 0;
}

void dijkstra(int G[MAX][MAX],int n,int startnode)
{

    int cost[MAX][MAX],distance[MAX],pred[MAX];
    int visited[MAX],count,mindistance,nextnode,i,j;

    //pred[] stores the predecessor of each node
    //count gives the number of nodes seen so far
    //create the cost matrix
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            if(G[i][j]==0)
                cost[i][j]=INFINITY;
            else
                cost[i][j]=G[i][j];

    //initialize pred[],distance[] and visited[]
    for(i=0;i<n;i++)
```

```
{
   distance[i]=cost[startnode][i];
   pred[i]=startnode;
   visited[i]=0;
}

distance[startnode]=0;
visited[startnode]=1;
count=1;

while(count<n-1)
{
   mindistance=INFINITY;

   //nextnode gives the node at minimum distance
   for(i=0;i<n;i++)
      if(distance[i]<mindistance&&!visited[i])
      {
         mindistance=distance[i];
         nextnode=i;
      }

      //check if a better path exists through nextnode
      visited[nextnode]=1;
      for(i=0;i<n;i++)
         if(!visited[i])
            if(mindistance+cost[nextnode][i]<distance[i])
            {
               distance[i]=mindistance+cost[nextnode][i];
               pred[i]=nextnode;
            }
```

```
        count++;
    }

    //print the path and distance of each node
    for(i=0;i<n;i++)
        if(i!=startnode)
        {
            printf("\nDistance of node%d=%d",i,distance[i]);
            printf("\nPath=%d",i);

            j=i;
            do
            {
                j=pred[j];
                printf("<-%d",j);
            }while(j!=startnode);
     }
}
    count=1;

    while(count<n-1)
    {
        mindistance=INFINITY;

        //nextnode gives the node at minimum distance
        for(i=0;i<n;i++)
            if(distance[i]<mindistance&&!visited[i])
            {
                mindistance=distance[i];
                nextnode=i;
            }
```

```
        //check if a better path exists through nextnode
        visited[nextnode]=1;
        for(i=0;i<n;i++)
           if(!visited[i])
              if(mindistance+cost[nextnode][i]<distance[i])
              {
                 distance[i]=mindistance+cost[nextnode][i];
                 pred[i]=nextnode;
              }
      count++;
   }

   //print the path and distance of each node
   for(i=0;i<n;i++)
      if(i!=startnode)
      {
         printf("\nDistance of node%d=%d",i,distance[i]);
         printf("\nPath=%d",i);

         j=i;
         do
         {
            j=pred[j];
            printf("<-%d",j);
         }while(j!=startnode);
      }
}
```

```
[student@localhost S]$ gcc SingleSourceShortestPath.c
[student@localhost S]$ ./a.out
Enter no. of vertices:5

Enter the adjacency matrix:
0 10 0 30 100
10 0 50 0 0
0 50 0 20 10
30 0 20 0 60
100 0 10 60 0

Enter the starting node:0

Distance of node1=10
Path=1<-0
Distance of node2=50
Path=2<-3<-0
Distance of node3=30
Path=3<-0
Distance of node4=60
Path=4<-2<-3<-0[student@localhost S]$
```

7. Single source shortest paths using greedy method when the graph is represented by adjacency List using C.

```
#include <stdio.h>

#include <stdlib.h>

#include <limits.h>


typedef struct node {

   int dest;

   int cost;

   struct node* next;

} node;


typedef struct Graph {

   int n;

   node** adjList;  // Array of pointers to linked lists

} Graph;
```

```
// Function to create a new node
node* createNode(int dest, int cost) {
   node* newNode = (node*)malloc(sizeof(node));
   newNode->dest = dest;
   newNode->cost = cost;
   newNode->next = NULL;
   return newNode;
}

// Function to initialize the graph
Graph* createGraph(int n) {
   Graph* graph = (Graph*)malloc(sizeof(Graph));
   graph->n = n;
   graph->adjList = (node**)malloc(n * sizeof(node*));

   for (int i = 0; i < n; i++) {
      graph->adjList[i] = NULL;  // Initialize the adjacency list for each vertex
   }

   return graph;
}

// Function to add an edge to the graph
void addEdge(Graph* graph, int source, int dest, int cost) {
   node* newNode = createNode(dest, cost);
   newNode->next = graph->adjList[source];
   graph->adjList[source] = newNode;  // Add at the beginning of the list
}

// Function to display the edges of the graph
void displayEdges(Graph* graph) {
```

```
    for (int i = 0; i < graph->n; i++) {
        node* temp = graph->adjList[i];
        printf("Adjacency list of vertex %d\n", i);
        while (temp) {
            printf("(%d)---(%d|%d) ", i, temp->dest, temp->cost);
            temp = temp->next;
        }
        printf("\n");
    }
}

// Helper function to find the vertex with the minimum distance
int findMinVertex(int* dist, int* Q, int n) {
    int minDist = INT_MAX;
    int minIndex = -1;
    for (int i = 0; i < n; i++) {
        if (Q[i] && dist[i] < minDist) {
            minDist = dist[i];
            minIndex = i;
        }
    }
    return minIndex;
}

// Dijkstra's Algorithm for shortest path
void dijkstra(Graph* graph, int* dist, int* prev, int start) {
    int n = graph->n;
    int* Q = (int*)malloc(n * sizeof(int));  // Set of unvisited nodes

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

```
    dist[i] = INT_MAX;  // Set all distances to infinity
    prev[i] = -1;       // Set all previous vertices to undefined
    Q[i] = 1;           // All vertices are unvisited initially
}

dist[start] = 0;  // Distance to the start vertex is 0

// Main Dijkstra loop
while (1) {
    int u = findMinVertex(dist, Q, n);  // Find the vertex with the minimum distance

    if (u == -1) {
        break;  // All reachable vertices have been processed
    }

    Q[u] = 0;  // Mark u as visited

    node* temp = graph->adjList[u];
    while (temp) {
        int v = temp->dest;
        int weight = temp->cost;

        // Relaxation step
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
            dist[v] = dist[u] + weight;
            prev[v] = u;
        }

        temp = temp->next;
    }
}
```

```
    free(Q);  // Clean up
}

int main() {
    int n = 7;  // Number of vertices
    Graph* g = createGraph(n);
    int dist[n], prev[n];
    int start = 0;

    // Adding edges to the graph
    addEdge(g, 0, 1, 3);
    addEdge(g, 0, 2, 6);
    addEdge(g, 1, 0, 3);
    addEdge(g, 1, 2, 2);
    addEdge(g, 1, 3, 1);
    addEdge(g, 2, 1, 6);
    addEdge(g, 2, 1, 2);
    addEdge(g, 2, 3, 1);
    addEdge(g, 2, 4, 4);
    addEdge(g, 2, 5, 2);
    addEdge(g, 3, 1, 1);
    addEdge(g, 3, 2, 1);
    addEdge(g, 3, 4, 2);
    addEdge(g, 3, 6, 4);
    addEdge(g, 4, 2, 4);
    addEdge(g, 4, 3, 2);
    addEdge(g, 4, 5, 2);
    addEdge(g, 4, 6, 1);
    addEdge(g, 5, 2, 2);
    addEdge(g, 5, 4, 2);
```

```
    addEdge(g, 5, 6, 1);
    addEdge(g, 6, 3, 4);
    addEdge(g, 6, 4, 1);
    addEdge(g, 6, 5, 1);

    // Run Dijkstra's algorithm from the start vertex
    dijkstra(g, dist, prev, start);

    // Print shortest paths and distances
    for (int i = 0; i < n; i++) {
        if (i != start) {
            printf("Start %d to %d, Cost: %d, Previous: %d\n", start, i, dist[i], prev[i]);
        }
    }

    return 0;
}
```

Output:

```
Start 0 to 1, Cost: 3, Previous: 0
Start 0 to 2, Cost: 5, Previous: 1
Start 0 to 3, Cost: 4, Previous: 1
Start 0 to 4, Cost: 6, Previous: 3
Start 0 to 5, Cost: 7, Previous: 2
Start 0 to 6, Cost: 7, Previous: 4
```

8. Implement job sequencing with deadlines using greedy method

```
#include <stdio.h>

#define MAX 100

typedef struct Job {
  char id[5];
  int deadline;
  int profit;
} Job;

void jobSequencingWithDeadline(Job jobs[], int n);

int minValue(int x, int y) {
  if(x < y) return x;
  return y;
}

int main(void) {
  //variables
  int i, j;

  //jobs with deadline and profit
  Job jobs[5] = {
    {"j1", 2,  60},
    {"j2", 1, 100},
    {"j3", 3,  20},
    {"j4", 2,  40},
    {"j5", 1,  20},
  };
```

```
  //temp
  Job temp;

  //number of jobs
  int n = 5;

  //sort the jobs profit wise in descending order
  for(i = 1; i < n; i++) {
    for(j = 0; j < n - i; j++) {
      if(jobs[j+1].profit > jobs[j].profit) {
        temp = jobs[j+1];
        jobs[j+1] = jobs[j];
        jobs[j] = temp;
      }
    }
  }

  printf("%10s %10s %10s\n", "Job", "Deadline", "Profit");
  for(i = 0; i < n; i++) {
    printf("%10s %10i %10i\n", jobs[i].id, jobs[i].deadline, jobs[i].profit);
  }

  jobSequencingWithDeadline(jobs, n);

  return 0;
}

void jobSequencingWithDeadline(Job jobs[], int n) {
  //variables
  int i, j, k, maxprofit;
```

```
//free time slots
int timeslot[MAX];

//filled time slots
int filledTimeSlot = 0;

//find max deadline value
int dmax = 0;
for(i = 0; i < n; i++) {
  if(jobs[i].deadline > dmax) {
    dmax = jobs[i].deadline;
  }
}

//free time slots initially set to -1 [-1 denotes EMPTY]
for(i = 1; i <= dmax; i++) {
  timeslot[i] = -1;
}

printf("dmax: %d\n", dmax);

for(i = 1; i <= n; i++) {
  k = minValue(dmax, jobs[i - 1].deadline);
  while(k >= 1) {
    if(timeslot[k] == -1) {
      timeslot[k] = i-1;
      filledTimeSlot++;
      break;
    }
    k--;
```

```
    }

    //if all time slots are filled then stop
    if(filledTimeSlot == dmax) {
      break;
    }
  }

  //required jobs
  printf("\nRequired Jobs: ");
  for(i = 1; i <= dmax; i++) {
    printf("%s", jobs[timeslot[i]].id);

    if(i < dmax) {
      printf(" --> ");
    }
  }

  //required profit
  maxprofit = 0;
  for(i = 1; i <= dmax; i++) {
    maxprofit += jobs[timeslot[i]].profit;
  }
  printf("\nMax Profit: %d\n", maxprofit);
}
```

Output

```
  Job   Deadline     Profit
   j2         1        100
   j1         2         60
   j4         2         40
```

j3	3	20
j5	1	20

dmax: 3

Required Jobs: j2 --> j1 --> j3

Max Profit: 180