DIJKSTRA ADJ LIST

//Dijkstra using adj list

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

// Structure to represent a node in the adjacency list

struct AdjListNode {

int dest;

int weight;

struct AdjListNode\* next;

};

// Structure to represent an adjacency list

struct AdjList {

struct AdjListNode\* head;

};

// Structure to represent a graph

struct Graph {

int V;

struct AdjList\* array;

};

// Structure to represent a node in the priority queue

struct MinHeapNode {

int v;

int dist;

};

// Structure to represent a priority queue

struct MinHeap {

int size;

int capacity;

int\* pos;

struct MinHeapNode\*\* array;

};

// Function prototypes

struct AdjListNode\* newAdjListNode(int dest, int weight);

struct Graph\* createGraph(int V);

void addEdge(struct Graph\* graph, int src, int dest, int weight);

struct MinHeap\* createMinHeap(int capacity);

void swapMinHeapNode(struct MinHeapNode\*\* a, struct MinHeapNode\*\* b);

void minHeapify(struct MinHeap\* minHeap, int idx);

struct MinHeapNode\* extractMin(struct MinHeap\* minHeap);

void decreaseKey(struct MinHeap\* minHeap, int v, int dist);

int isInMinHeap(struct MinHeap\* minHeap, int v);

void printShortestPath(struct Graph\* graph, int src, int dest, int dist[]);

// Dijkstra's algorithm

void dijkstra(struct Graph\* graph, int src, int dest);

int main() {

int V, E;

// Get the number of vertices and edges from the user

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

scanf("%d", &V);

printf("Enter the number of edges: ");

scanf("%d", &E);

struct Graph\* graph = createGraph(V);

// Get edge details from the user

for (int i = 0; i < E; ++i) {

int src, dest, weight;

printf("Enter edge %d (source destination weight): ", i + 1);

scanf("%d %d %d", &src, &dest, &weight);

addEdge(graph, src, dest, weight);

}

int src, dest;

// Get source and destination nodes from the user

printf("Enter the source node: ");

scanf("%d", &src);

printf("Enter the destination node: ");

scanf("%d", &dest);

// Apply Dijkstra's algorithm

dijkstra(graph, src, dest);

return 0;

}

// Function to create a new adjacency list node

struct AdjListNode\* newAdjListNode(int dest, int weight) {

struct AdjListNode\* newNode = (struct AdjListNode\*)malloc(sizeof(struct AdjListNode));

newNode->dest = dest;

newNode->weight = weight;

newNode->next = NULL;

return newNode;

}

// Function to create a graph with V vertices

struct Graph\* createGraph(int V) {

struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));

graph->V = V;

graph->array = (struct AdjList\*)malloc(V \* sizeof(struct AdjList));

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

graph->array[i].head = NULL;

return graph;

}

// Function to add an edge to an undirected graph

void addEdge(struct Graph\* graph, int src, int dest, int weight) {

struct AdjListNode\* newNode = newAdjListNode(dest, weight);

newNode->next = graph->array[src].head;

graph->array[src].head = newNode;

newNode = newAdjListNode(src, weight);

newNode->next = graph->array[dest].head;

graph->array[dest].head = newNode;

}

// Function to create a new Min Heap Node

struct MinHeapNode\* newMinHeapNode(int v, int dist) {

struct MinHeapNode\* minHeapNode = (struct MinHeapNode\*)malloc(sizeof(struct MinHeapNode));

minHeapNode->v = v;

minHeapNode->dist = dist;

return minHeapNode;

}

// Function to create a Min Heap

struct MinHeap\* createMinHeap(int capacity) {

struct MinHeap\* minHeap = (struct MinHeap\*)malloc(sizeof(struct MinHeap));

minHeap->pos = (int\*)malloc(capacity \* sizeof(int));

minHeap->size = 0;

minHeap->capacity = capacity;

minHeap->array = (struct MinHeapNode\*\*)malloc(capacity \* sizeof(struct MinHeapNode\*));

return minHeap;

}

// Function to swap two min heap nodes

void swapMinHeapNode(struct MinHeapNode\*\* a, struct MinHeapNode\*\* b) {

struct MinHeapNode\* t = \*a;

\*a = \*b;

\*b = t;

}

// Function to heapify at given idx

void minHeapify(struct MinHeap\* minHeap, int idx) {

int smallest, left, right;

smallest = idx;

left = 2 \* idx + 1;

right = 2 \* idx + 2;

if (left < minHeap->size && minHeap->array[left]->dist < minHeap->array[smallest]->dist)

smallest = left;

if (right < minHeap->size && minHeap->array[right]->dist < minHeap->array[smallest]->dist)

smallest = right;

if (smallest != idx) {

// The nodes to be swapped in min heap

struct MinHeapNode\* smallestNode = minHeap->array[smallest];

struct MinHeapNode\* idxNode = minHeap->array[idx];

// Swap positions

minHeap->pos[smallestNode->v] = idx;

minHeap->pos[idxNode->v] = smallest;

// Swap nodes

swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);

minHeapify(minHeap, smallest);

}

}

// Function to check if the given vertex is in the min heap or not

int isInMinHeap(struct MinHeap\* minHeap, int v) {

if (minHeap->pos[v] < minHeap->size)

return 1;

return 0;

}

// Function to extract the minimum node from the heap

struct MinHeapNode\* extractMin(struct MinHeap\* minHeap) {

if (minHeap->size == 0)

return NULL;

// Extract the root

struct MinHeapNode\* root = minHeap->array[0];

// Replace the root with the last node

struct MinHeapNode\* lastNode = minHeap->array[minHeap->size - 1];

minHeap->array[0] = lastNode;

// Update position of last node

minHeap->pos[root->v] = minHeap->size - 1;

minHeap->pos[lastNode->v] = 0;

// Reduce heap size and heapify the root

minHeap->size--;

minHeapify(minHeap, 0);

return root;

}

// Function to decrease the dist value of a given vertex v

void decreaseKey(struct MinHeap\* minHeap, int v, int dist) {

// Get the index of v in the heap array

int i = minHeap->pos[v];

// Get the node and update its dist value

minHeap->array[i]->dist = dist;

// Travel up while the complete tree is not heapified

while (i && minHeap->array[i]->dist < minHeap->array[(i - 1) / 2]->dist) {

// Swap this node with its parent

minHeap->pos[minHeap->array[i]->v] = (i - 1) / 2;

minHeap->pos[minHeap->array[(i - 1) / 2]->v] = i;

swapMinHeapNode(&minHeap->array[i], &minHeap->array[(i - 1) / 2]);

// Move to the parent index

i = (i - 1) / 2;

}

}

// Function to print the shortest path from source to destination

void printShortestPath(struct Graph\* graph, int src, int dest, int dist[]) {

printf("\nShortest Path from Node %d to Node %d:\n", src, dest);

printf("Node\tDistance from Source\n");

for (int i = 0; i < graph->V; i++) {

printf("%d\t%d\n", i, dist[i]);

}

}

// Dijkstra's algorithm to find the shortest path in a graph

void dijkstra(struct Graph\* graph, int src, int dest) {

int V = graph->V;

int dist[V]; // The output array dist[i] holds the shortest distance from src to i

// minHeap represents the priority queue

struct MinHeap\* minHeap = createMinHeap(V);

// Initialize the heap, source vertex distance is 0, all others are INFINITE

for (int v = 0; v < V; ++v) {

dist[v] = INT\_MAX;

minHeap->array[v] = newMinHeapNode(v, dist[v]);

minHeap->pos[v] = v;

}

// Make dist value of src vertex as 0 so that it is extracted first

minHeap->array[src] = newMinHeapNode(src, dist[src]);

minHeap->pos[src] = src;

dist[src] = 0;

decreaseKey(minHeap, src, dist[src]);

// Initially, the heap size is equal to the number of vertices

minHeap->size = V;

// Dijkstra's algorithm

while (minHeap->size > 0) {

// Extract the vertex with the minimum distance value

struct MinHeapNode\* minHeapNode = extractMin(minHeap);

int u = minHeapNode->v; // Store the extracted vertex number

// Traverse through all adjacent vertices of u and update their distance values

struct AdjListNode\* crawl = graph->array[u].head;

while (crawl != NULL) {

int v = crawl->dest;

// If the shortest path to v is not finalized yet, and the new distance is smaller

if (isInMinHeap(minHeap, v) && dist[u] != INT\_MAX && crawl->weight + dist[u] < dist[v]) {

dist[v] = dist[u] + crawl->weight;

// Update distance value in the min heap

decreaseKey(minHeap, v, dist[v]);

}

crawl = crawl->next;

}

}

// Print the shortest distance

printf("\nShortest Distance from Node %d to Node %d: %d\n", src, dest, dist[dest]);

// Print the shortest path if it exists

if (dist[dest] != INT\_MAX)

printShortestPath(graph, src, dest, dist);

else

printf("No path found from Node %d to Node %d\n", src, dest);

// Free allocated memory

free(minHeap->array);

free(minHeap->pos);

free(minHeap);

}

input

Enter the number of vertices: 5

Enter the number of edges: 8

Enter edge 1 (source destination weight): 0 1 2

Enter edge 2 (source destination weight): 0 2 4

Enter edge 3 (source destination weight): 1 2 1

Enter edge 4 (source destination weight): 1 3 7

Enter edge 5 (source destination weight): 2 3 3

Enter edge 6 (source destination weight): 2 4 5

Enter edge 7 (source destination weight): 3 4 1

Enter edge 8 (source destination weight): 4 0 8

Enter the source node: 0

Enter the destination node: 3