#include <iostream>

#include <vector>

#include <queue>

#include <limits>

#define INF std::numeric\_limits<int>::max()

// Structure to represent a weighted edge

struct Edge {

int destination;

int weight;

};

// Graph class

class Graph {

int numVertices;

std::vector<std::vector<Edge>> adjacencyList;

public:

Graph(int vertices) : numVertices(vertices), adjacencyList(vertices) {}

void addEdge(int source, int destination, int weight) {

Edge edge;

edge.destination = destination;

edge.weight = weight;

adjacencyList[source].push\_back(edge);

}

std::vector<int> dijkstra(int source) {

std::vector<int> distance(numVertices, INF); // Initialize distances as infinity

std::priority\_queue<std::pair<int, int>, std::vector<std::pair<int, int>>, std::greater<std::pair<int, int>>> pq;

distance[source] = 0; // Distance from source to source is 0

pq.push(std::make\_pair(0, source));

while (!pq.empty()) {

int u = pq.top().second;

pq.pop();

for (const Edge& edge : adjacencyList[u]) {

int v = edge.destination;

int weight = edge.weight;

if (distance[u] != INF && distance[u] + weight < distance[v]) {

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

pq.push(std::make\_pair(distance[v], v));

}

}

}

return distance;

}

};

// Example usage

int main() {

int numVertices = 6;

Graph graph(numVertices);

// Adding edges to the graph

graph.addEdge(0, 1, 4);

graph.addEdge(0, 2, 3);

graph.addEdge(1, 3, 2);

graph.addEdge(1, 2, 1);

graph.addEdge(2, 3, 4);

graph.addEdge(2, 4, 3);

graph.addEdge(3, 4, 2);

graph.addEdge(3, 5, 1);

graph.addEdge(4, 5, 6);

int source = 0;

std::vector<int> distances = graph.dijkstra(source);

std::cout << "Shortest distances from source vertex " << source << ":\n";

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

std::cout << "Vertex " << i << ": " << distances[i] << "\n";

}

return 0;

}

//Abhishek

|  |
| --- |
| // C++ program for Dijkstra's single source shortest path |
|  | // algorithm. The program is for adjacency matrix |
|  | // representation of the graph |
|  | #include <iostream> |
|  | using namespace std; |
|  | #include <limits.h> |
|  |  |
|  | // Number of vertices in the graph |
|  | #define V 9 |
|  |  |
|  | // A utility function to find the vertex with minimum |
|  | // distance value, from the set of vertices not yet included |
|  | // in shortest path tree |
|  | int minDistance(int dist[], bool sptSet[]) |
|  | { |
|  |  |
|  | // Initialize min value |
|  | int min = INT\_MAX, min\_index; |
|  |  |
|  | for (int v = 0; v < V; v++) |
|  | if (sptSet[v] == false && dist[v] <= min) |
|  | min = dist[v], min\_index = v; |
|  |  |
|  | return min\_index; |
|  | } |
|  |  |
|  | // A utility function to print the constructed distance |
|  | // array |
|  | void printSolution(int dist[]) |
|  | { |
|  | cout << "Vertex \t Distance from Source" << endl; |
|  | for (int i = 0; i < V; i++) |
|  | cout << i << " \t\t\t\t" << dist[i] << endl; |
|  | } |
|  |  |
|  | // Function that implements Dijkstra's single source |
|  | // shortest path algorithm for a graph represented using |
|  | // adjacency matrix representation |
|  | void dijkstra(int graph[V][V], int src) |
|  | { |
|  | int dist[V]; // The output array. dist[i] will hold the |
|  | // shortest |
|  | // distance from src to i |
|  |  |
|  | bool sptSet[V]; // sptSet[i] will be true if vertex i is |
|  | // included in shortest |
|  | // path tree or shortest distance from src to i is |
|  | // finalized |
|  |  |
|  | // Initialize all distances as INFINITE and stpSet[] as |
|  | // false |
|  | for (int i = 0; i < V; i++) |
|  | dist[i] = INT\_MAX, sptSet[i] = false; |
|  |  |
|  | // Distance of source vertex from itself is always 0 |
|  | dist[src] = 0; |
|  |  |
|  | // Find shortest path for all vertices |
|  | for (int count = 0; count < V - 1; count++) { |
|  | // Pick the minimum distance vertex from the set of |
|  | // vertices not yet processed. u is always equal to |
|  | // src in the first iteration. |
|  | int u = minDistance(dist, sptSet); |
|  |  |
|  | // Mark the picked vertex as processed |
|  | sptSet[u] = true; |
|  |  |
|  | // Update dist value of the adjacent vertices of the |
|  | // picked vertex. |
|  | for (int v = 0; v < V; v++) |
|  |  |
|  | // Update dist[v] only if is not in sptSet, |
|  | // there is an edge from u to v, and total |
|  | // weight of path from src to v through u is |
|  | // smaller than current value of dist[v] |
|  | if (!sptSet[v] && graph[u][v] |
|  | && dist[u] != INT\_MAX |
|  | && dist[u] + graph[u][v] < dist[v]) |
|  | dist[v] = dist[u] + graph[u][v]; |
|  | } |
|  |  |
|  | // print the constructed distance array |
|  | printSolution(dist); |
|  | } |
|  |  |
|  | // driver's code |
|  | int main() |
|  | { |
|  |  |
|  | /\* Let us create the example graph discussed above \*/ |
|  | int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 }, |
|  | { 4, 0, 8, 0, 0, 0, 0, 11, 0 }, |
|  | { 0, 8, 0, 7, 0, 4, 0, 0, 2 }, |
|  | { 0, 0, 7, 0, 9, 14, 0, 0, 0 }, |
|  | { 0, 0, 0, 9, 0, 10, 0, 0, 0 }, |
|  | { 0, 0, 4, 14, 10, 0, 2, 0, 0 }, |
|  | { 0, 0, 0, 0, 0, 2, 0, 1, 6 }, |
|  | { 8, 11, 0, 0, 0, 0, 1, 0, 7 }, |
|  | { 0, 0, 2, 0, 0, 0, 6, 7, 0 } }; |
|  |  |
|  | // Function call |
|  | dijkstra(graph, 0); |
|  |  |
|  | return 0; |
|  | } |