```
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
using namespace std;

// Data structure to store a graph edge
struct Edge {
	int source, dest, weight;
};

// Data structure to store a heap node
struct Node {
	int vertex, weight;
};

// A class to represent a graph object
class Graph
{
public:
	// a vector of vectors of `Edge` to represent an adjacency list
	vector<vector<Edge>> adjList;

	// Graph Constructor
	Graph(vector<Edge> const &edges, int n)
	{
		// resize the vector to hold `n` elements of type vector<Edge>
		adjList.resize(n);

		// add edges to the directed graph
		for (Edge const &edge: edges)
		{
			// insert at the end
			adjList[edge.source].push_back(edge);
		}
	}
};

void printPath(vector<int> const &prev, int i, int source)
{
	if (i < 0) {
		return;
	}
	printPath(prev, prev[i], source);
	if (i != source) {
		cout << ", ";
	}
	cout << i;
}

// Comparison object to be used to order the heap
struct comp
{
	bool operator()(const Node &lhs, const Node &rhs) const {
		return lhs.weight > rhs.weight;
	}
};

// Run Dijkstra’s algorithm on the given graph
```

```
void findShortestPaths(Graph const &graph, int source, int n)
{
      // create a min-heap and push source node having distance 0
      priority_queue<Node, vector<Node>, comp> min_heap;
      min_heap.push({source, 0});

      // set initial distance from the source to `v` as infinity
      vector<int> dist(n, INT_MAX);

      // distance from the source to itself is zero
      dist[source] = 0;

      // boolean array to track vertices for which minimum
      // cost is already found
      vector<bool> done(n, false);
      done[source] = true;

      // stores predecessor of a vertex (to a print path)
      vector<int> prev(n, -1);

      // run till min-heap is empty
      while (!min_heap.empty())
      {
            // Remove and return the best vertex
            Node node = min_heap.top();
            min_heap.pop();

            // get the vertex number
            int u = node.vertex;

            // do for each neighbor `v` of `u`
            for (auto i: graph.adjList[u])
            {
                  int v = i.dest;
                  int weight = i.weight;

                  // Relaxation step
                  if (!done[v] && (dist[u] + weight) < dist[v])
                  {
                        dist[v] = dist[u] + weight;
                        prev[v] = u;
                        min_heap.push({v, dist[v]});
                  }
            }

            // mark vertex `u` as done so it will not get picked up again
            done[u] = true;
      }

      for (int i = 0; i < n; i++)
      {
            if (i != source && dist[i] != INT_MAX)
            {
                  cout << "Path (" << source << " —> " << i << "): Minimum cost = "
                         << dist[i] << ", Route = [";
                  printPath(prev, i, source);
                  cout << "]" << endl;
            }
      }
```

```
}

int main()
{
	// initialize edges as per the above diagram
	// (u, v, w) represent edge from vertex `u` to vertex `v` having weight `w`
	vector<Edge> edges =
	{
		{0, 1, 10}, {0, 4, 3}, {1, 2, 2}, {1, 4, 4}, {2, 3, 9},
		{3, 2, 7}, {4, 1, 1}, {4, 2, 8}, {4, 3, 2}
	};

	// total number of nodes in the graph (labelled from 0 to 4)
	int n = 5;

	// construct graph
	Graph graph(edges, n);

	// run the Dijkstra’s algorithm from every node
	for (int source = 0; source < n; source++) {
		findShortestPaths(graph, source, n);
	}

	return 0;
}
```