// C++ implementation of above approach

#include <bits/stdc++.h>

using namespace std;

// graph

vector<vector<int> > graph;

// map to store cost of edges

map<pair<int, int>, int> cost;

// returns the minimum cost in a vector( if

// there are multiple goal states)

vector<int> uniform\_cost\_search(vector<int> goal, int start)

{

    // minimum cost upto

    // goal state from starting

    // state

    vector<int> answer;

    // create a priority queue

    priority\_queue<pair<int, int> > queue;

    // set the answer vector to max value

    for (int i = 0; i < goal.size(); i++)

        answer.push\_back(INT\_MAX);

    // insert the starting index

    queue.push(make\_pair(0, start));

    // map to store visited node

    map<int, int> visited;

    // count

    int count = 0;

    // while the queue is not empty

    while (queue.size() > 0) {

        // get the top element of the

        // priority queue

        pair<int, int> p = queue.top();

        // pop the element

        queue.pop();

        // get the original value

        p.first \*= -1;

        // check if the element is part of

        // the goal list

        if (find(goal.begin(), goal.end(), p.second) != goal.end()) {

            // get the position

            int index = find(goal.begin(), goal.end(),

                            p.second) - goal.begin();

            // if a new goal is reached

            if (answer[index] == INT\_MAX)

                count++;

            // if the cost is less

            if (answer[index] > p.first)

                answer[index] = p.first;

            // pop the element

            queue.pop();

            // if all goals are reached

            if (count == goal.size())

                return answer;

        }

        // check for the non visited nodes

        // which are adjacent to present node

        if (visited[p.second] == 0)

            for (int i = 0; i < graph[p.second].size(); i++) {

                // value is multiplied by -1 so that

                // least priority is at the top

                queue.push(make\_pair((p.first +

                cost[make\_pair(p.second, graph[p.second][i])]) \* -1,

                graph[p.second][i]));

            }

        // mark as visited

        visited[p.second] = 1;

    }

    return answer;

}

// main function

int main()

{

    // create the graph

    graph.resize(5);

    // add edge

    graph[0].push\_back(1);

    graph[0].push\_back(2);

    graph[1].push\_back(2);

    graph[1].push\_back(3);

    graph[1].push\_back(4);

    graph[2].push\_back(3);

    graph[3].push\_back(4);

    // add the cost

    cost[make\_pair(0, 1)] = 1;

    cost[make\_pair(0, 2)] = 4;

    cost[make\_pair(1, 2)] = 3;

    cost[make\_pair(1, 3)] = 6;

    cost[make\_pair(1, 4)] = 7;

    cost[make\_pair(2, 3)] = 8;

    cost[make\_pair(3, 4)] = 10;

    // goal state

    vector<int> goal;

    // set the goal

    // there can be multiple goal states

    goal.push\_back(4);

    // get the answer

    vector<int> answer = uniform\_cost\_search(goal, 0);

    // print the answer

    cout << "Minimum cost from 0 to 5 is = "

        << answer[0] << endl;

    return 0;

}