Phần D

26.

**Chứng minh:**

1. Giả sử có một đồ thị có hướng với các trọng số không âm và ta muốn tính đường đi ngắn nhất từ đỉnh nguồn *s* đến đỉnh đích *t*.
2. Tạo một đồ thị mới với cùng các đỉnh như đồ thị gốc, nhưng với trọng số cạnh được tính là tổng trọng số của các đỉnh trên đường đi từ đỉnh nguồn *s* đến đỉnh đích *t*. Cụ thể, nếu có một đường đi từ *s* đến *t* là *s*→*v*1​→*v*2​→…→*vk*​→*t*, thì trọng số của cạnh từ *s* đến *t* trong đồ thị mới sẽ là *w*(*s*,*v*1​)+*w*(*v*1​,*v*2​)+…+*w*(*vk*​,*t*).
3. Áp dụng thuật toán Dijkstra trên đồ thị mới với đỉnh nguồn *s* để tính đường đi ngắn nhất từ *s* đến tất cả các đỉnh khác.
4. Kết quả của thuật toán Dijkstra trên đồ thị mới sẽ cho ta trọng số ngắn nhất từ *s* đến *t* trong đồ thị gốc.

Do đó, có thể tính các đường đi ngắn nhất trong một đồ thị có hướng với các trọng số không âm

27.

#include <iostream>

#include <vector>

#include <queue>

#include <limits>

using namespace std;

struct Edge {

int to;

int weight;

Edge(int to, int weight) : to(to), weight(weight) {}

};

class DijkstraMultipleSources {

private:

int numVertices;

vector<vector<Edge>> adjacencyList;

public:

DijkstraMultipleSources(int numVertices) : numVertices(numVertices + 1), adjacencyList(numVertices + 1) {}

void addEdge(int from, int to, int weight) {

adjacencyList[from].push\_back(Edge(to, weight));

}

vector<vector<int>> findShortestPaths(const vector<int>& sources) {

vector<vector<int>> shortestPaths(numVertices, vector<int>(numVertices, numeric\_limits<int>::max()));

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

// Add dummy source with weight 0 to connect all real sources

int dummySource = numVertices;

for (int source : sources) {

addEdge(dummySource, source, 0);

}

// Run Dijkstra's algorithm

for (int source : sources) {

vector<int> distance(numVertices, numeric\_limits<int>::max());

distance[source] = 0;

pq.push({0, source});

while (!pq.empty()) {

int u = pq.top().second;

int dist = pq.top().first;

pq.pop();

if (dist > distance[u]) {

continue;

}

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

int v = edge.to;

int newDist = distance[u] + edge.weight;

if (newDist < distance[v]) {

distance[v] = newDist;

pq.push({newDist, v});

}

}

}

// Update shortest paths for the current source

for (int i = 1; i < numVertices; ++i) {

shortestPaths[source][i] = distance[i];

}

}

return shortestPaths;

}

int getShortestPath(int source, int destination) {

vector<int> sources = {source};

vector<vector<int>> shortestPaths = findShortestPaths(sources);

return shortestPaths[source][destination];

}

};

int main() {

// Example usage

int numVertices = 6;

DijkstraMultipleSources dijkstra(numVertices);

dijkstra.addEdge(1, 2, 2);

dijkstra.addEdge(1, 3, 4);

dijkstra.addEdge(2, 4, 3);

dijkstra.addEdge(3, 4, 1);

dijkstra.addEdge(3, 5, 7);

dijkstra.addEdge(4, 6, 5);

dijkstra.addEdge(5, 6, 2);

vector<int> sources = {1, 3};

vector<vector<int>> shortestPaths = dijkstra.findShortestPaths(sources);

// Output shortest paths

cout << "Shortest Paths:\n";

for (int source : sources) {

for (int destination = 1; destination <= numVertices; ++destination) {

cout << "Shortest path from " << source << " to " << destination << ": ";

if (shortestPaths[source][destination] == numeric\_limits<int>::max()) {

cout << "No path\n";

} else {

cout << shortestPaths[source][destination] << "\n";

}

}

}

return 0;

}

28.

#include <iostream>

#include <vector>

#include <queue>

#include <limits>

using namespace std;

struct Edge {

int to;

int weight;

Edge(int to, int weight) : to(to), weight(weight) {}

};

class DijkstraShortestPath {

private:

int numVertices;

vector<vector<Edge>> adjacencyList;

public:

DijkstraShortestPath(int numVertices) : numVertices(numVertices + 1), adjacencyList(numVertices + 1) {}

void addEdge(int from, int to, int weight) {

adjacencyList[from].push\_back(Edge(to, weight));

}

int findShortestPath(int source, const vector<int>& targets) {

vector<int> distance(numVertices, numeric\_limits<int>::max());

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

for (int target : targets) {

distance[target] = 0;

pq.push({0, target});

}

while (!pq.empty()) {

int u = pq.top().second;

int dist = pq.top().first;

pq.pop();

if (dist > distance[u]) {

continue;

}

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

int v = edge.to;

int newDist = distance[u] + edge.weight;

if (newDist < distance[v]) {

distance[v] = newDist;

pq.push({newDist, v});

}

}

}

// Find the minimum distance from source to any target

int minPath = numeric\_limits<int>::max();

for (int target : targets) {

minPath = min(minPath, distance[target]);

}

return minPath;

}

};

int main() {

// Example usage

int numVertices = 6;

DijkstraShortestPath dijkstra(numVertices);

dijkstra.addEdge(1, 2, 2);

dijkstra.addEdge(1, 3, 4);

dijkstra.addEdge(2, 4, 3);

dijkstra.addEdge(3, 4, 1);

dijkstra.addEdge(3, 5, 7);

dijkstra.addEdge(4, 6, 5);

dijkstra.addEdge(5, 6, 2);

vector<int> sources = {1, 3};

vector<int> targets = {2, 4, 6};

int shortestPath = dijkstra.findShortestPath(sources[0], targets);

// Output shortest path

cout << "Shortest Path: " << shortestPath << "\n";

return 0;

}

29.

#include <iostream>

#include <vector>

#include <queue>

#include <limits>

#include <algorithm>

using namespace std;

struct Point {

int x, y;

Point(int x, int y) : x(x), y(y) {}

};

struct Edge {

int to;

int weight;

Edge(int to, int weight) : to(to), weight(weight) {}

};

class DijkstraShortestPath {

private:

int numVertices;

vector<Point> coordinates;

vector<vector<Edge>> adjacencyList;

public:

DijkstraShortestPath(int numVertices, const vector<Point>& coordinates)

: numVertices(numVertices), coordinates(coordinates), adjacencyList(numVertices + 1) {}

void addEdge(int from, int to, int weight) {

adjacencyList[from].push\_back(Edge(to, weight));

}

int dijkstraShortestPath(int source, int target) {

vector<int> distance(numVertices + 1, numeric\_limits<int>::max());

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

distance[source] = 0;

pq.push({0, source});

while (!pq.empty()) {

int u = pq.top().second;

int dist = pq.top().first;

pq.pop();

if (dist > distance[u]) {

continue;

}

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

int v = edge.to;

int newDist = distance[u] + edge.weight;

if (newDist < distance[v]) {

distance[v] = newDist;

pq.push({newDist, v});

}

}

}

return distance[target];

}

void optimizeVerticesOrder() {

// Sort vertices based on coordinates

vector<int> order(numVertices);

iota(order.begin(), order.end(), 1);

sort(order.begin(), order.end(), [this](int a, int b) {

return coordinates[a].x < coordinates[b].x || (coordinates[a].x == coordinates[b].x && coordinates[a].y < coordinates[b].y);

});

// Reorder adjacency list based on the new order

vector<vector<Edge>> newAdjList(numVertices + 1);

for (int i = 1; i <= numVertices; ++i) {

for (const Edge& edge : adjacencyList[order[i - 1]]) {

newAdjList[i].push\_back({lower\_bound(order.begin(), order.end(), edge.to) - order.begin() + 1, edge.weight});

}

}

adjacencyList = move(newAdjList);

}

};

int main() {

// Example usage

int numVertices = 6;

vector<Point> coordinates = {{0, 0}, {1, 2}, {3, 1}, {2, 4}, {5, 3}, {4, 5}};

DijkstraShortestPath dijkstra(numVertices, coordinates);

dijkstra.addEdge(1, 2, 2);

dijkstra.addEdge(1, 3, 4);

dijkstra.addEdge(2, 4, 3);

dijkstra.addEdge(3, 4, 1);

dijkstra.addEdge(3, 5, 7);

dijkstra.addEdge(4, 6, 5);

dijkstra.addEdge(5, 6, 2);

// Optimize the order of vertices based on coordinates

dijkstra.optimizeVerticesOrder();

int shortestPath = dijkstra.dijkstraShortestPath(1, 6);

// Output shortest path

cout << "Shortest Path: " << shortestPath << "\n";

return 0;

}

30.

#include <iostream>

#include <vector>

#include <queue>

#include <limits>

using namespace std;

struct Edge {

int to;

int weight;

Edge(int to, int weight) : to(to), weight(weight) {}

};

class ReverseDijkstra {

private:

int numVertices;

vector<vector<Edge>> adjacencyList;

public:

ReverseDijkstra(int numVertices) : numVertices(numVertices + 1), adjacencyList(numVertices + 1) {}

void addEdge(int from, int to, int weight) {

adjacencyList[to].push\_back(Edge(from, weight));

}

vector<int> findLongestPath(int start) {

vector<int> distance(numVertices, numeric\_limits<int>::min());

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

distance[start] = 0;

pq.push({0, start});

while (!pq.empty()) {

int u = pq.top().second;

int dist = pq.top().first;

pq.pop();

if (dist < distance[u]) {

continue;

}

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

int v = edge.to;

int newDist = distance[u] + edge.weight;

if (newDist > distance[v]) {

distance[v] = newDist;

pq.push({newDist, v});

}

}

}

return distance;

}

};

int main() {

// Example usage

int numVertices = 6;

ReverseDijkstra reverseDijkstra(numVertices);

reverseDijkstra.addEdge(1, 2, 2);

reverseDijkstra.addEdge(1, 3, 4);

reverseDijkstra.addEdge(2, 4, 3);

reverseDijkstra.addEdge(3, 4, 1);

reverseDijkstra.addEdge(3, 5, 7);

reverseDijkstra.addEdge(4, 6, 5);

reverseDijkstra.addEdge(5, 6, 2);

int startVertex = 6;

vector<int> longestPath = reverseDijkstra.findLongestPath(startVertex);

// Output longest path

cout << "Longest Path from vertex " << startVertex << ":\n";

for (int i = 1; i <= numVertices; ++i) {

cout << "To vertex " << i << ": " << longestPath[i] << "\n";

}

return 0;

}

31.

#include <iostream>

#include <vector>

#include <limits>

#include <queue>

using namespace std;

struct Edge {

int from, to, weight;

Edge(int from, int to, int weight) : from(from), to(to), weight(weight) {}

};

class BellmanFord {

private:

int numVertices;

vector<Edge> edges;

public:

BellmanFord(int numVertices) : numVertices(numVertices) {}

void addEdge(int from, int to, int weight) {

edges.push\_back(Edge(from, to, weight));

}

vector<int> computeShortestPaths(int start) {

vector<int> distance(numVertices, numeric\_limits<int>::max());

distance[start] = 0;

for (int i = 0; i < numVertices - 1; ++i) {

for (const Edge& edge : edges) {

if (distance[edge.from] != numeric\_limits<int>::max() &&

distance[edge.from] + edge.weight < distance[edge.to]) {

distance[edge.to] = distance[edge.from] + edge.weight;

}

}

}

return distance;

}

void reweightGraph(vector<int>& pi) {

for (Edge& edge : edges) {

if (pi[edge.from] != numeric\_limits<int>::max()) {

edge.weight = edge.weight + pi[edge.from] - pi[edge.to];

}

}

}

};

class DijkstraShortestPath {

private:

int numVertices;

vector<vector<pair<int, int>>> adjacencyList;

public:

DijkstraShortestPath(int numVertices) : numVertices(numVertices), adjacencyList(numVertices + 1) {}

void addEdge(int from, int to, int weight) {

adjacencyList[from].push\_back({to, weight});

}

vector<int> findShortestPaths(int start) {

vector<int> distance(numVertices + 1, numeric\_limits<int>::max());

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

distance[start] = 0;

pq.push({0, start});

while (!pq.empty()) {

int u = pq.top().second;

int dist = pq.top().first;

pq.pop();

if (dist > distance[u]) {

continue;

}

for (const auto& neighbor : adjacencyList[u]) {

int v = neighbor.first;

int newDist = distance[u] + neighbor.second;

if (newDist < distance[v]) {

distance[v] = newDist;

pq.push({newDist, v});

}

}

}

return distance;

}

};

int main() {

// Example usage

int numVertices = 6;

BellmanFord bellmanFord(numVertices);

bellmanFord.addEdge(1, 2, 2);

bellmanFord.addEdge(1, 3, 4);

bellmanFord.addEdge(2, 4, 3);

bellmanFord.addEdge(3, 4, 1);

bellmanFord.addEdge(3, 5, 7);

bellmanFord.addEdge(4, 6, 5);

bellmanFord.addEdge(5, 6, 2);

int startVertex = 1;

// Compute pi[v]

vector<int> pi = bellmanFord.computeShortestPaths(startVertex);

// Reweight the graph

bellmanFord.reweightGraph(pi);

// Use Dijkstra to find all shortest paths in the reweighted graph

DijkstraShortestPath dijkstra(numVertices);

for (const Edge& edge : bellmanFord.getEdges()) {

dijkstra.addEdge(edge.from, edge.to, edge.weight);

}

vector<int> shortestPaths = dijkstra.findShortestPaths(startVertex);

// Output shortest paths

cout << "Shortest Paths:\n";

for (int i = 1; i <= numVertices; ++i) {

cout << "To vertex " << i << ": " << shortestPaths[i] << "\n";

}

return 0;

}

32.

#include <iostream>

#include <vector>

using namespace std;

const int INF = numeric\_limits<int>::max();

class ShortestPathsPreprocessing {

private:

vector<vector<int>> distanceMatrix;

public:

ShortestPathsPreprocessing(vector<vector<int>>& graph) {

int n = graph.size();

distanceMatrix.resize(n, vector<int>(n, INF));

// Initialize the distance matrix with direct edges

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

for (int j : graph[i]) {

distanceMatrix[i][j] = 1; // Assume unit weights

distanceMatrix[j][i] = 1; // Assuming undirected graph

}

}

// Apply Floyd-Warshall algorithm

for (int k = 0; k < n; ++k) {

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

for (int j = 0; j < n; ++j) {

if (distanceMatrix[i][k] != INF && distanceMatrix[k][j] != INF &&

distanceMatrix[i][k] + distanceMatrix[k][j] < distanceMatrix[i][j]) {

distanceMatrix[i][j] = distanceMatrix[i][k] + distanceMatrix[k][j];

}

}

}

}

}

int getShortestDistance(int u, int v) {

return distanceMatrix[u][v];

}

};

int main() {

// Example usage

int n = 6; // Number of vertices

vector<vector<int>> graph(n);

// Add edges to the graph

graph[0] = {1};

graph[1] = {0, 2};

graph[2] = {1, 3};

graph[3] = {2, 4};

graph[4] = {3, 5};

graph[5] = {4};

// Preprocess the graph to compute all shortest paths

ShortestPathsPreprocessing preprocessing(graph);

// Get the shortest distance between vertices 1 and 4

int shortestDistance = preprocessing.getShortestDistance(1, 4);

cout << "Shortest Distance between vertices 1 and 4: " << shortestDistance << endl;

return 0;

}

33.

#include <iostream>

#include <vector>

#include <climits>

using namespace std;

int minPathSum(vector<vector<int>>& grid) {

int rows = grid.size();

int cols = grid[0].size();

// Tạo một ma trận dp để lưu trữ độ dài đường đi ngắn nhất đến từng ô

vector<vector<int>> dp(rows, vector<int>(cols, 0));

// Khởi tạo dp[0][0] là giá trị tại ô (0,0) của ma trận grid

dp[0][0] = grid[0][0];

// Khởi tạo giá trị đầu hàng và đầu cột của dp

for (int i = 1; i < rows; ++i) {

dp[i][0] = dp[i - 1][0] + grid[i][0];

}

for (int j = 1; j < cols; ++j) {

dp[0][j] = dp[0][j - 1] + grid[0][j];

}

// Tính toán dp cho các ô còn lại

for (int i = 1; i < rows; ++i) {

for (int j = 1; j < cols; ++j) {

// Tại mỗi ô (i, j), chọn giá trị nhỏ nhất giữa dp[i-1][j] (đi từ trên xuống) và dp[i][j-1] (đi từ trái sang)

dp[i][j] = min(dp[i - 1][j], dp[i][j - 1]) + grid[i][j];

}

}

// Kết quả là giá trị tại ô (N-1, N-1)

return dp[rows - 1][cols - 1];

}

int main() {

// Example usage

vector<vector<int>> grid = {

{1, 3, 1},

{1, 5, 1},

{4, 2, 1}

};

int result = minPathSum(grid);

cout << "Shortest Path Sum: " << result << endl;

return 0;

}

34.

#include <iostream>

#include <vector>

#include <queue>

#include <algorithm>

#include <limits>

using namespace std;

struct Edge {

int to;

int weight;

Edge(int to, int weight) : to(to), weight(weight) {}

};

class MonotonicShortestPath {

private:

int numVertices;

vector<vector<Edge>> adjacencyList;

public:

MonotonicShortestPath(int numVertices) : numVertices(numVertices), adjacencyList(numVertices + 1) {}

void addEdge(int from, int to, int weight) {

adjacencyList[from].push\_back(Edge(to, weight));

}

vector<int> dijkstra(int start, vector<int>& dist) {

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

vector<int> visited(numVertices + 1, false);

dist[start] = 0;

pq.push({0, start});

while (!pq.empty()) {

int u = pq.top().second;

int cost = pq.top().first;

pq.pop();

if (visited[u]) {

continue;

}

visited[u] = true;

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

int v = edge.to;

int weight = edge.weight;

if (!visited[v] && dist[u] + weight < dist[v]) {

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

pq.push({dist[v], v});

}

}

}

return dist;

}

int monotonicShortestPath(int start) {

vector<int> increasingDist(numVertices + 1, numeric\_limits<int>::max());

vector<int> decreasingDist(numVertices + 1, numeric\_limits<int>::max());

// Dijkstra with edges sorted in increasing order of weight

dijkstra(start, increasingDist);

// Reverse the edges in the adjacency list

vector<vector<Edge>> reversedAdjList(numVertices + 1);

for (int u = 1; u <= numVertices; ++u) {

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

reversedAdjList[edge.to].push\_back(Edge(u, edge.weight));

}

}

// Dijkstra with edges sorted in decreasing order of weight

dijkstra(start, decreasingDist);

// Find the minimum monotonic path length

int minMonotonicPath = numeric\_limits<int>::max();

for (int u = 1; u <= numVertices; ++u) {

minMonotonicPath = min(minMonotonicPath, max(increasingDist[u], decreasingDist[u]));

}

return minMonotonicPath;

}

};

int main() {

// Example usage

int numVertices = 5;

MonotonicShortestPath msp(numVertices);

msp.addEdge(1, 2, 1);

msp.addEdge(1, 3, 4);

msp.addEdge(2, 3, 2);

msp.addEdge(2, 4, 5);

msp.addEdge(3, 4, 1);

msp.addEdge(3, 5, 3);

msp.addEdge(4, 5, 1);

int startVertex = 1;

int result = msp.monotonicShortestPath(startVertex);

cout << "Minimum Monotonic Shortest Path: " << result << endl;

return 0;

}

35.

#include <iostream>

#include <vector>

#include <queue>

#include <algorithm>

#include <limits>

using namespace std;

struct Edge {

int to;

int weight;

Edge(int to, int weight) : to(to), weight(weight) {}

};

class BitonicShortestPath {

private:

int numVertices;

vector<vector<Edge>> adjacencyList;

public:

BitonicShortestPath(int numVertices) : numVertices(numVertices), adjacencyList(numVertices + 1) {}

void addEdge(int from, int to, int weight) {

adjacencyList[from].push\_back(Edge(to, weight));

}

vector<int> dijkstra(int start, vector<int>& dist) {

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

vector<int> visited(numVertices + 1, false);

dist[start] = 0;

pq.push({0, start});

while (!pq.empty()) {

int u = pq.top().second;

int cost = pq.top().first;

pq.pop();

if (visited[u]) {

continue;

}

visited[u] = true;

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

int v = edge.to;

int weight = edge.weight;

if (!visited[v] && dist[u] + weight < dist[v]) {

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

pq.push({dist[v], v});

}

}

}

return dist;

}

int bitonicShortestPath(int start, int destination) {

// Step 1: Find bitonic shortest path to intermediate vertices

vector<int> increasingDist(numVertices + 1, numeric\_limits<int>::max());

dijkstra(start, increasingDist);

// Step 2: Find bitonic shortest path from intermediate vertices to destination

vector<int> reversedDist(numVertices + 1, numeric\_limits<int>::max());

// Reverse the edges in the adjacency list

vector<vector<Edge>> reversedAdjList(numVertices + 1);

for (int u = 1; u <= numVertices; ++u) {

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

reversedAdjList[edge.to].push\_back(Edge(u, edge.weight));

}

}

dijkstra(destination, reversedDist);

// Find the minimum bitonic path length

int minBitonicPath = numeric\_limits<int>::max();

for (int u = 1; u <= numVertices; ++u) {

minBitonicPath = min(minBitonicPath, increasingDist[u] + reversedDist[u]);

}

return minBitonicPath;

}

};

int main() {

// Example usage

int numVertices = 5;

BitonicShortestPath bsp(numVertices);

bsp.addEdge(1, 2, 1);

bsp.addEdge(1, 3, 4);

bsp.addEdge(2, 3, 2);

bsp.addEdge(2, 4, 5);

bsp.addEdge(3, 4, 1);

bsp.addEdge(3, 5, 3);

bsp.addEdge(4, 5, 1);

int startVertex = 1;

int endVertex = 5;

int result = bsp.bitonicShortestPath(startVertex, endVertex);

cout << "Minimum Bitonic Shortest Path from " << startVertex << " to " << endVertex << ": " << result << endl;

return 0;

}

36.

#include <iostream>

#include <vector>

#include <limits>

using namespace std;

struct Edge {

int from;

int to;

int weight;

Edge(int from, int to, int weight) : from(from), to(to), weight(weight) {}

};

class CriticalEdgeFinder {

private:

int numVertices;

vector<Edge> edges;

public:

CriticalEdgeFinder(int numVertices) : numVertices(numVertices) {}

void addEdge(int from, int to, int weight) {

edges.push\_back(Edge(from, to, weight));

}

int bellmanFord(int source, int destination) {

vector<int> distance(numVertices + 1, numeric\_limits<int>::max());

distance[source] = 0;

for (int i = 1; i <= numVertices - 1; ++i) {

for (const Edge& edge : edges) {

int u = edge.from;

int v = edge.to;

int weight = edge.weight;

if (distance[u] != numeric\_limits<int>::max() && distance[u] + weight < distance[v]) {

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

}

}

}

return distance[destination];

}

Edge findCriticalEdge(int source, int destination) {

int originalDistance = bellmanFord(source, destination);

Edge criticalEdge = {-1, -1, -1};

for (const Edge& edge : edges) {

int originalWeight = edge.weight;

edge.weight = numeric\_limits<int>::max();

int newDistance = bellmanFord(source, destination);

if (newDistance > originalDistance) {

originalDistance = newDistance;

criticalEdge = edge;

}

edge.weight = originalWeight;

}

return criticalEdge;

}

};

int main() {

// Example usage

int numVertices = 5;

CriticalEdgeFinder cef(numVertices);

cef.addEdge(1, 2, 3);

cef.addEdge(1, 3, 2);

cef.addEdge(2, 3, 5);

cef.addEdge(2, 4, 1);

cef.addEdge(3, 4, 4);

cef.addEdge(3, 5, 2);

cef.addEdge(4, 5, 1);

int source = 1;

int destination = 5;

Edge criticalEdge = cef.findCriticalEdge(source, destination);

if (criticalEdge.from != -1) {

cout << "Critical Edge: (" << criticalEdge.from << " -> " << criticalEdge.to << ") with weight " << criticalEdge.weight << endl;

} else {

cout << "No critical edge found." << endl;

}

return 0;

}

37.

#include <iostream>

#include <vector>

#include <limits.h>

#include <algorithm>

using namespace std;

const int INF = INT\_MAX;

struct Edge {

int src, dest, weight;

};

class Graph {

public:

int V, E;

vector<Edge> edges;

Graph(int V, int E) : V(V), E(E) {

edges.resize(E);

}

void addEdge(int src, int dest, int weight) {

edges.push\_back({src, dest, weight});

}

void optimizedBellmanFord(int src, int C) {

vector<int> distance(C + 1, INF);

distance[0] = 0;

for (int i = 0; i < V - 1; ++i) {

for (const Edge& edge : edges) {

if (distance[edge.src] != INF && distance[edge.src] + edge.weight < distance[edge.dest]) {

distance[edge.dest] = distance[edge.src] + edge.weight;

}

}

}

// Print the shortest distances

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

cout << "Shortest distance to vertex " << i << " is " << distance[i] << endl;

}

}

};

int main() {

int V = 5; // số đỉnh

int E = 8; // số cạnh

Graph graph(V, E);

// Thêm các cạnh của đồ thị

graph.addEdge(0, 1, 5);

graph.addEdge(0, 2, 2);

graph.addEdge(1, 3, 4);

graph.addEdge(1, 4, 1);

graph.addEdge(2, 1, 3);

graph.addEdge(2, 4, 7);

graph.addEdge(3, 4, 2);

graph.addEdge(4, 3, 1);

int source = 0; // đỉnh nguồn

// Giả sử trọng số không vượt quá hằng số C

int C = 10;

graph.optimizedBellmanFord(source, C);

return 0;

}

38.

#include <iostream>

#include <SFML/Graphics.hpp>

#include <unordered\_map>

#include <vector>

#include <queue>

class DijkstraVisualizer {

public:

DijkstraVisualizer(std::unordered\_map<char, std::vector<std::pair<char, int>>> graph, char source);

void visualize();

private:

std::unordered\_map<char, std::vector<std::pair<char, int>>> graph;

char source;

std::unordered\_map<char, int> distances;

std::unordered\_map<char, char> predecessors;

std::vector<char> visited;

sf::RenderWindow window;

void initializeGraph();

void drawGraph();

void drawPath(char destination);

};

DijkstraVisualizer::DijkstraVisualizer(std::unordered\_map<char, std::vector<std::pair<char, int>>> graph, char source)

: graph(graph), source(source) {

initializeGraph();

window.create(sf::VideoMode(800, 600), "Dijkstra's Algorithm Visualization");

}

void DijkstraVisualizer::initializeGraph() {

for (const auto& node : graph) {

distances[node.first] = std::numeric\_limits<int>::max();

predecessors[node.first] = '\0';

}

distances[source] = 0;

}

void DijkstraVisualizer::drawGraph() {

window.clear(sf::Color::White);

sf::Font font;

if (!font.loadFromFile("arial.ttf")) {

std::cerr << "Font not found." << std::endl;

return;

}

for (const auto& node : graph) {

sf::CircleShape circle(30);

circle.setPosition(50 + (node.first - 'A') \* 100, 200);

circle.setFillColor(sf::Color::LightBlue);

sf::Text text;

text.setFont(font);

text.setString(node.first);

text.setCharacterSize(20);

text.setPosition(50 + (node.first - 'A') \* 100 + 10, 200 + 10);

window.draw(circle);

window.draw(text);

for (const auto& neighbor : node.second) {

sf::Vertex line[] = {

sf::Vertex(sf::Vector2f(50 + (node.first - 'A') \* 100 + 30, 200 + 30)),

sf::Vertex(sf::Vector2f(50 + (neighbor.first - 'A') \* 100, 200 + 30))

};

window.draw(line, 2, sf::Lines);

sf::Text weight;

weight.setFont(font);

weight.setString(std::to\_string(neighbor.second));

weight.setCharacterSize(15);

weight.setPosition((line[0].position.x + line[1].position.x) / 2, (line[0].position.y + line[1].position.y) / 2);

window.draw(weight);

}

}

window.display();

}

void DijkstraVisualizer::drawPath(char destination) {

visited.push\_back(destination);

char current = destination;

while (predecessors[current] != '\0') {

visited.push\_back(predecessors[current]);

current = predecessors[current];

}

for (int i = visited.size() - 1; i >= 0; --i) {

sf::CircleShape circle(30);

circle.setPosition(50 + (visited[i] - 'A') \* 100, 400);

circle.setFillColor(sf::Color::Green);

sf::Text text;

text.setFont(window.getDefaultFont());

text.setString(visited[i]);

text.setCharacterSize(20);

text.setPosition(50 + (visited[i] - 'A') \* 100 + 10, 400 + 10);

window.draw(circle);

window.draw(text);

window.display();

sf::sleep(sf::milliseconds(500));

}

}

void DijkstraVisualizer::visualize() {

while (window.isOpen()) {

sf::Event event;

while (window.pollEvent(event)) {

if (event.type == sf::Event::Closed)

window.close();

}

drawGraph();

char current = source;

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

pq.push({0, source});

while (!pq.empty()) {

int uDist = -pq.top().first;

char u = pq.top().second;

pq.pop();

if (uDist > distances[u])

continue;

for (const auto& neighbor : graph[u]) {

char v = neighbor.first;

int weight = neighbor.second;

if (distances[v] > distances[u] + weight) {

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

predecessors[v] = u;

pq.push({-distances[v], v});

drawGraph();

}

}

sf::sleep(sf::milliseconds(500));

drawPath(u);

}

}

}

int main() {

std::unordered\_map<char, std::vector<std::pair<char, int>>> graph = {

{'A', {{'B', 1}, {'D', 3}}},

{'B', {{'A', 1}, {'D', 4}, {'E', 2}}},

{'C', {{'E', 5}}},

{'D', {{'A', 3}, {'B', 4}, {'E', 1}}},

{'E', {{'B', 2}, {'C', 5}, {'D', 1}}}

};

char source = 'A';

DijkstraVisualizer visualizer(graph, source);

visualizer.visualize();

return 0;

}