CQUPT – University at Albany

Computer Science – International College

**ICSI 403 --- Design and Analysis of Algorithms**

**Project 4 --- Spring 2025**

**Table of Contents**

1. **System documentation -------------------------------------------------------------1**
   * A high-level data flow diagram for the system-----------------------------1
   * A list of routines and their brief descriptions-------------------------------2
   * Implementation details --------------------------------------------------------3
2. **Test documentation-----------------------------------------------------------------10**
   * How you tested your program ----------------------------------------------11
   * Testing outputs ----------------------------------------------------------------17
3. **User documentation ----------------------------------------------------------------17**
   * How to run your program ---------------------------------------------------17
   * Describe parameter (if any) -------------------------------------------------17
4. **Source Code--------------------------------------------------------------------------17**
   * Correctness --------------------------------------------------------------------19
   * Programming style -----------------------------------------------------------29

**I.** **System documentation**

**1.** **A high-level data flow diagram for the system**

图示

AI 生成的内容可能不正确。

Data flow diagram

**2. A list of routines and their brief descriptions**

| **Routine Name** | **Description** |
| --- | --- |
| main(int argc, char \*argv[]) | Program entry point; reads input file, builds graph, runs Dijkstra and Bellman-Ford algorithms, outputs results. |
| Graph::Graph(int vertices) | Constructor: initializes graph with vertices nodes and empty adjacency lists. |
| Graph::addDirectedEdge(int u, int v, double weight) | Adds a directed edge from u to v with given weight to the graph. |
| Graph::dijkstra(int src, vector<double> &dist, vector<int> &prev) | Runs Dijkstra's algorithm from source src, computes shortest distances and predecessors; returns false if negative edge detected. |
| Graph::bellmanFord(int src, vector<double> &dist, vector<int> &prev) | Runs Bellman-Ford algorithm from source src, detects negative cycles; returns false if negative cycle found. |
| Graph::getPath(int src, int dest, const vector<int> &prev, const vector<double> &dist) | Builds string of shortest path from src to dest (like <0,1,2>); returns empty path if unreachable. |
| Graph::printShortestPaths(int src, const vector<double> &dist, const vector<int> &prev) | Prints shortest distance and path from src to all nodes in specified format. |
| MinHeapPQ::insert(int vertex, double priority) | Inserts a vertex with given priority into the min-heap priority queue. |
| MinHeapPQ::extractMin() | Removes and returns the vertex with minimum priority from the queue. |
| MinHeapPQ::decreaseKey(int vertex, double newPriority) | Decreases the priority of a given vertex and reorders the heap accordingly. |
| MinHeapPQ::contains(int vertex) const | Checks if a vertex is currently in the priority queue. |
| MinHeapPQ::isEmpty() const | Returns whether the priority queue is empty. |
| PriorityQueue::create() | Factory method: creates and returns a new instance of MinHeapPQ. |

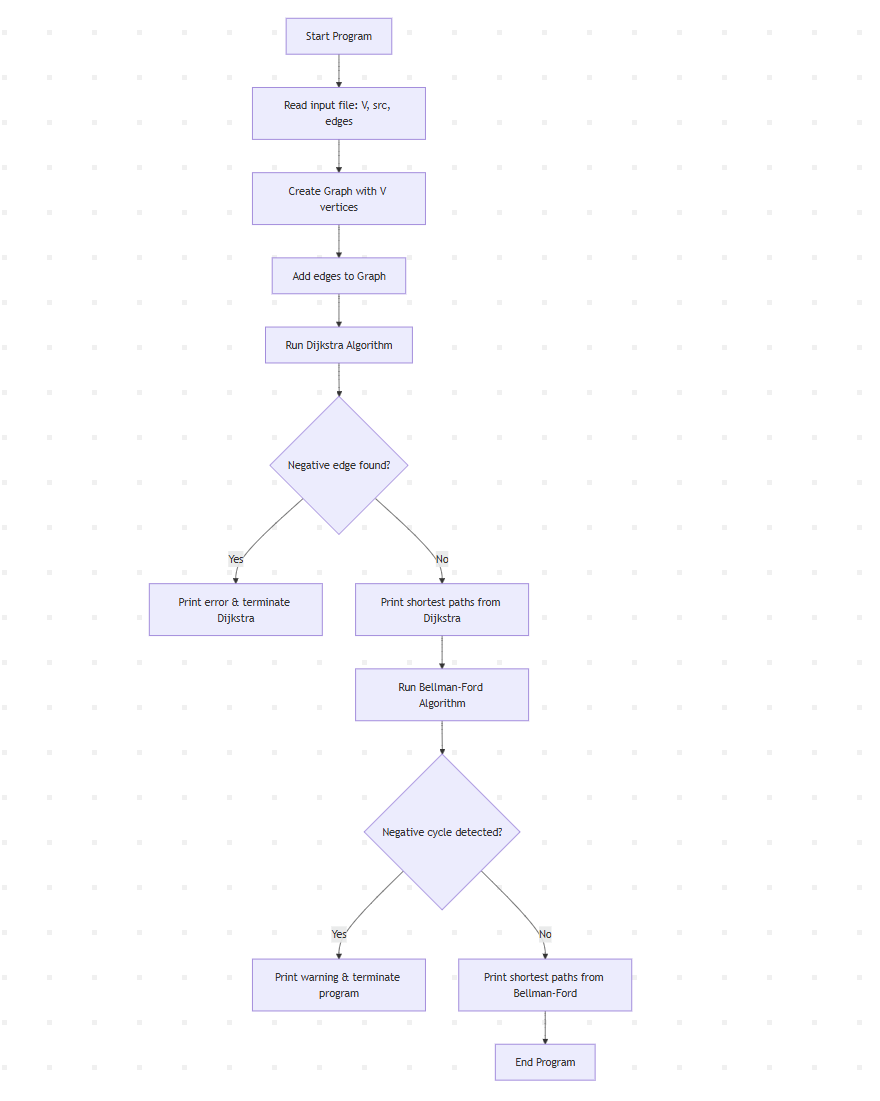
**3. Implementation details.**

**3.1 System Overview**

This system implements two classical shortest path algorithms — **Dijkstra** and **Bellman-Ford** — to compute the shortest distances and corresponding paths from a given source node to all other nodes in a directed weighted graph.

The graph and the source node are read from an input file specified by the user. The program supports graphs with positive and negative edge weights but handles them differently:

* **Dijkstra’s algorithm** is used first. It efficiently computes shortest paths but does not support negative edge weights. If any negative weight edges are encountered, the algorithm terminates early with an error.
* **Bellman-Ford algorithm** is used afterwards. It handles negative edges and detects negative weight cycles. If a negative cycle exists, the program notifies the user and terminates.



Flowchart

**3.2 Key Components**

* **Graph Class:** Represents the graph internally as adjacency lists. It supports adding directed edges and implements both shortest path algorithms.
* **Priority Queue:** A custom min-heap based priority queue (MinHeapPQ) supports Dijkstra’s algorithm for efficient vertex selection and priority updates.
* **Main Program:** Parses input, constructs the graph, executes both algorithms sequentially, and outputs the shortest path distances and paths for each node in a clear format.

**3.3 Output**

For each node, the program outputs the shortest distance from the source and the full path taken (including source and destination nodes) in the format:

node = **distance**;

path = <source, ..., node>

If a node is unreachable, distance is shown as INF and path as empty < >.

**3.4 Graph Class: Graph Representation and Edge Management**

**Design Goals**

* Efficiently store a directed weighted graph using adjacency lists.
* Support fast traversal of all outgoing edges of a node.

**Implementation Highlights**

* Use a vector of vectors of pairs: vector<vector<pair<int, double>>> adj, where each entry holds (neighbor, weight).
* Constructor initializes adjacency list size according to the number of vertices.
* addDirectedEdge appends edges efficiently.

**Key Code**

class Graph

{

private:

    int V;

    vector<vector<pair<int, double>>> adj;

public:

**Graph**(int vertices) : **V**(vertices), **adj**(vertices) {}

    void **addDirectedEdge**(int u, int v, double weight)

    {

        adj[u].**emplace\_back**(v, weight);

    }

};

**Notes**

* Space complexity is O(V + E).
* Edge traversal is linear in the number of neighbors.

**3.5 Dijkstra’s Algorithm Implementation Details**

**Design Goals**

* Compute single-source shortest paths assuming non-negative edge weights.
* Use a custom priority queue supporting decrease-key for efficiency.

**Core Logic**

* Initialize distance vector dist to infinity and predecessor vector prev to -1.
* Set source node distance to zero and insert it into the priority queue.
* Iteratively extract the vertex with minimum distance and relax its outgoing edges.
* Abort and report error if a negative weight edge is encountered.

**Key Code**

bool **dijkstra**(int src, vector<double> &dist, vector<int> &prev)

{

    dist.**assign**(V, INF);

    prev.**assign**(V, -1);

    dist[src] = 0;

    PriorityQueue \*pq = PriorityQueue::**create**();

    pq->**insert**(src, 0);

    while (!pq->**isEmpty**())

    {

        int u = pq->**extractMin**();

        for (auto &edge : adj[u])

        {

            int v = edge.first;

            double weight = edge.second;

            if (weight < 0)

            {

                cout << "Error: Negative edge encountered at (" << u << "->" << v << ")\n";

                delete pq;

                return false;

            }

            if (dist[v] > dist[u] + weight)

            {

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

                prev[v] = u;

                if (pq->**contains**(v))

                    pq->**decreaseKey**(v, dist[v]);

                else

                    pq->**insert**(v, dist[v]);

            }

        }

    }

    delete pq;

    return true;

}

**Performance**

* Time complexity: O(E log V) due to priority queue operations.
* contains and decreaseKey are optimized by the custom heap implementation.

**3.6 Bellman-Ford Algorithm Details**

**Design Goals**

* Calculate shortest paths even with negative edge weights.
* Detect negative weight cycles and terminate early.

**Algorithm Summary**

* Initialize distances and predecessors as in Dijkstra.
* Perform up to V-1 rounds of edge relaxation.
* Stop early if no updates occur in a round.
* After relaxation, check for any edge that can still relax distances to detect negative cycles.

**Key Code**

bool **bellmanFord**(int src, vector<double> &dist, vector<int> &prev)

{

    dist.**assign**(V, INF);

    prev.**assign**(V, -1);

    dist[src] = 0;

    for (int i = 0; i < V - 1; ++i)

    {

        bool updated = false;

        for (int u = 0; u < V; ++u)

        {

            if (dist[u] == INF)

                continue;

            for (auto &edge : adj[u])

            {

                int v = edge.first;

                double weight = edge.second;

                if (dist[v] > dist[u] + weight)

                {

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

                    prev[v] = u;

                    updated = true;

                }

            }

        }

        if (!updated)

            break; // Early stopping optimization

    }

    // Negative cycle detection

    for (int u = 0; u < V; ++u)

    {

        if (dist[u] == INF)

            continue;

        for (auto &edge : adj[u])

        {

            int v = edge.first;

            double weight = edge.second;

            if (dist[v] > dist[u] + weight)

            {

                return false; // Negative cycle detected

            }

        }

    }

    return true;

}

**Performance**

* Time complexity: O(VE).
* Suitable for graphs with negative edges but inefficient for very large graphs.

**3.7 Path Reconstruction**

**Design Goals**

* Recover the shortest path from the source to a given destination.
* Return a string like <0,1,3> including both source and destination nodes.
* Handle unreachable nodes by returning an empty path < >.

**Implementation Notes**

* Use a stack to reverse the predecessor chain for correct order.
* Verify path is valid (source at the top of the stack).

**Key Code**

string **getPath**(int src, int dest, const vector<int> &prev, const vector<double> &dist)

{

    if (dist[dest] == INF)

        return "< >"; // Destination unreachable

    stack<int> pathStack;

    for (int cur = dest; cur != -1; cur = prev[cur])

        pathStack.**push**(cur);

    if (pathStack.**top**() != src)

        return "< >"; // No valid path from source

    string pathStr = "<";

    while (!pathStack.**empty**())

    {

        pathStr += **to\_string**(pathStack.**top**());

        pathStack.**pop**();

        if (!pathStack.**empty**())

            pathStr += ",";

    }

    pathStr += ">";

    return pathStr;

}

**3.8 Result Printing**

**Design Goals**

* Print shortest distances and paths in a human-readable format.
* Format example:

0=0; path = <0>

1=4; path = <0,1>

2=9; path = <0,3,2>

3=INF; path = < >

**Key Code**

void **printShortestPaths**(int src, const vector<double> &dist, const vector<int> &prev)

{

    cout << "The shortest-length path is:\n";

    for (int i = 0; i < V; ++i)

    {

        if (dist[i] == INF)

            cout << i << "=INF; path = < >\n";

        else

            cout << i << "=" << fixed << **setprecision**(0) << dist[i]

                 << "; path = " << **getPath**(src, i, prev, dist) << "\n";

    }

}

**3.9 Custom Priority Queue Implementation (mypq.h and MinHeapPQ)**

**Design Goals**

* Support typical priority queue operations:
* insert(vertex, priority)
* extractMin()
* decreaseKey(vertex, newPriority)
* contains(vertex)
* isEmpty()
* Maintain min-heap invariant.

**Implementation Details**

* Store heap in a vector of HeapElement {vertex, priority}.
* Maintain a hash map vertexToIndex for O(1) index lookup of any vertex.
* heapifyUp and heapifyDown maintain heap order after insertions or key updates.

**Key Code**

void **decreaseKey**(int vertex, double newPriority) override

{

    if (!**contains**(vertex))

        return;

    size\_t idx = vertexToIndex[vertex];

    if (newPriority >= heap[idx].priority)

        return;

    heap[idx].priority = newPriority;

**heapifyUp**(idx);

}

**7. Main Program Flow**

**Overview**

* Validate command-line arguments and open input file.
* Parse number of vertices and source vertex.
* Build the graph by reading edges.
* Run Dijkstra's algorithm (checks for negative edges).
* Run Bellman-Ford algorithm (detects negative cycles).
* Print shortest paths for both.

**Key Code**

int **main**(int argc, char \*argv[])

{

    if (argc != 2)

    {

        cerr << "Usage: " << argv[0] << " <input\_file>\n";

        return 1;

    }

    ifstream **inputFile**(argv[1]);

    if (!inputFile)

    {

        cerr << "Error: Cannot open input file\n";

        return 1;

    }

    int V, src;

    inputFile >> V >> src;

    Graph **g**(V);

    int u, v;

    double weight;

    while (inputFile >> u >> v >> weight)

    {

        g.**addDirectedEdge**(u, v, weight);

    }

    vector<double> dist;

    vector<int> prev;

    cout << "=== Dijkstra Algorithm ===\n";

    if (g.**dijkstra**(src, dist, prev))

    {

        g.**printShortestPaths**(src, dist, prev);

    }

    else

    {

        cout << "Dijkstra algorithm terminated due to negative edge weight\n";

    }

    cout << "\n=== Bellman-Ford Algorithm ===\n";

    if (g.**bellmanFord**(src, dist, prev))

    {

        g.**printShortestPaths**(src, dist, prev);

    }

    else

    {

        cout << "Warning: Negative cycle detected, program terminating\n";

        return 1;

    }

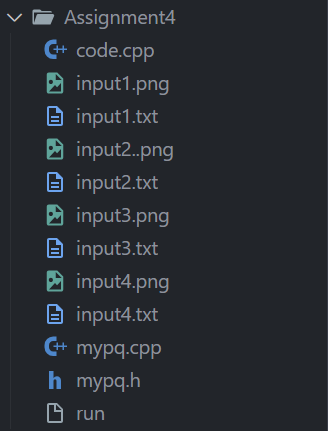
    return 0;

}

**II Test documentation**

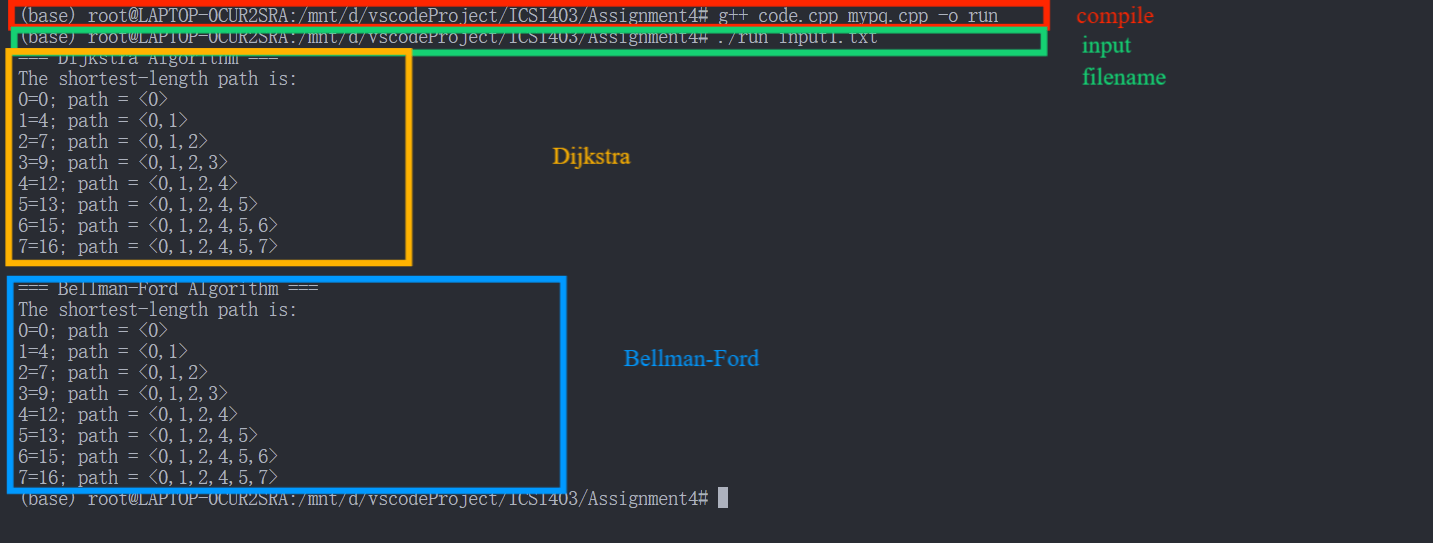
**1.** **How you tested your program**

I execute my program in **Vscode**, the program structure as follows



The input files are: **input1.txt, input2.txt,input3.txt,input4.txt，**The answer files are: **answer1.txt, answer2.txt, answer3.txt, answer4.txt**

1. Compiling and running **code.c**, then **input** the testcase file’s name
2. The console will show the result
3. Results have **three parts**, organe show the **Dijkstra Algorithm**, blue show the **Bellman-Ford Algorithm**



Output layout

I check the correctness of the program through **manual simulation**

**2. Testing outputs**

There are four testcases

**2.1 Testcase1**

There is no negative weight

电脑萤幕画面

AI 生成的内容可能不正确。

Input1.txt

图片包含 风筝, 项链, 挂, 飞行

AI 生成的内容可能不正确。

Input1 Illustration

文本

AI 生成的内容可能不正确。

My output1

**2.2 Testcase2**

There is a negative weight but no negative loop

手机屏幕的截图

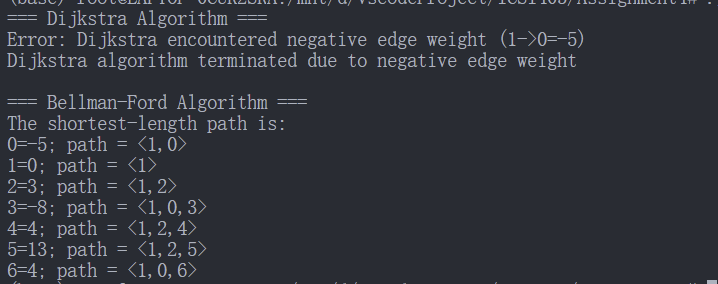
AI 生成的内容可能不正确。

Input2.txt

图片包含 项链, 挂, 桌子, 各种

AI 生成的内容可能不正确。

Input2 Illustration



My output2

**2.3 Testcase3**

There is a negative loop

手机屏幕的截图

AI 生成的内容可能不正确。

Input3.txt

图片包含 物体, 滑雪, 男人, 挂

AI 生成的内容可能不正确。

Input3 Illustration

屏幕的截图

AI 生成的内容可能不正确。

My output3

**2.4 Testcase4**

A large sample with a negative loop

屏幕上有字

AI 生成的内容可能不正确。

Input4.txt

图片包含 风筝, 飞行, 人, 华美

AI 生成的内容可能不正确。

Input4 Illustration

文本

AI 生成的内容可能不正确。

My output4

**III. User documentation**

1. **How to run your program**

Compile

g++ code.cpp mypq.cpp - o run

then, run it with filename

./ run input1.txt

**2.** **Describe parameter (if any)**

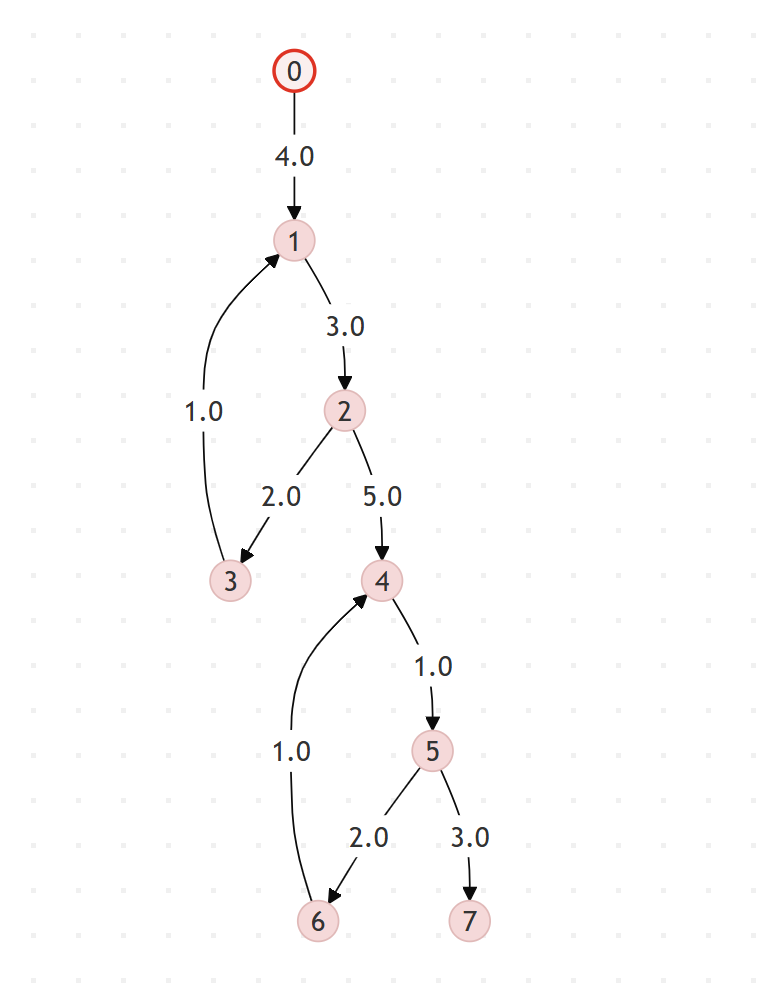
The program receive a filename to find the input file

**IV. Source Code**

1. **Correctness**

I check the correctness of the program through **manual simulation**

**1.1 Testcase1**

文本

AI 生成的内容可能不正确。

Through manual simulation, the answer is **Correct!**

**1.2 Testcase2**

图片包含 项链, 挂, 桌子, 各种

AI 生成的内容可能不正确。文本

AI 生成的内容可能不正确。

There is a negative weight in graph, Therefore Dijkstra terminated

Through manual simulation, the answer is **Correct!**

**1.3 Testcase3、**

图片包含 物体, 滑雪, 男人, 挂

AI 生成的内容可能不正确。屏幕的截图

AI 生成的内容可能不正确。

There is a negative weight in graph, Therefore Dijkstra terminated

Bellman-Ford find the negative cycle in graph, so it terminated

Through manual simulation, the answer is **Correct!**

**1.4 Testcase4**

图片包含 风筝, 飞行, 人, 华美

AI 生成的内容可能不正确。文本

AI 生成的内容可能不正确。

There is a negative weight in graph, Therefore Dijkstra terminated

Through manual simulation, the answer is **Correct!**

1. **Programming style**

**Layering | Readability | Comments | Efficiency** are showing as follows

**mypq.h**

#ifndef **MYPQ\_H**

#define **MYPQ\_H**

class PriorityQueue

{

public:

    virtual **~PriorityQueue**() {}

    virtual void **insert**(int vertex, double priority) = 0;

    virtual int **extractMin**() = 0;

    virtual void **decreaseKey**(int vertex, double newPriority) = 0;

    virtual bool **contains**(int vertex) const = 0;

    virtual bool **isEmpty**() const = 0;

    // 创建优先队列实例的工厂方法

    static PriorityQueue \***create**();

};

#endif // MYPQ\_H

**mypq.cpp**

#include "mypq.h"

#include <vector>

#include <unordered\_map>

#include <limits>

#include <algorithm>

using namespace std;

class MinHeapPQ : public PriorityQueue

{

private:

    struct HeapElement

    {

        int vertex;

        double priority;

    };

    vector<HeapElement> heap;

    unordered\_map<int, size\_t> vertexToIndex;

    void **heapifyUp**(size\_t index)

    {

        while (index > 0)

        {

            size\_t parent = (index - 1) / 2;

            if (heap[parent].priority <= heap[index].priority)

                break;

**swap**(heap[parent], heap[index]);

            vertexToIndex[heap[parent].vertex] = parent;

            vertexToIndex[heap[index].vertex] = index;

            index = parent;

        }

    }

    void **heapifyDown**(size\_t index)

    {

        size\_t left, right, smallest;

        while (true)

        {

            left = 2 \* index + 1;

            right = 2 \* index + 2;

            smallest = index;

            if (left < heap.**size**() && heap[left].priority < heap[smallest].priority)

                smallest = left;

            if (right < heap.**size**() && heap[right].priority < heap[smallest].priority)

                smallest = right;

            if (smallest == index)

                break;

**swap**(heap[index], heap[smallest]);

            vertexToIndex[heap[index].vertex] = index;

            vertexToIndex[heap[smallest].vertex] = smallest;

            index = smallest;

        }

    }

public:

**~MinHeapPQ**() override = default;

    void **insert**(int vertex, double priority) override

    {

        heap.**push\_back**({vertex, priority});

        vertexToIndex[vertex] = heap.**size**() - 1;

**heapifyUp**(heap.**size**() - 1);

    }

    int **extractMin**() override

    {

        if (heap.**empty**())

            return -1;

        int minVertex = heap[0].vertex;

        vertexToIndex.**erase**(minVertex);

        if (heap.**size**() > 1)

        {

            heap[0] = heap.**back**();

            vertexToIndex[heap[0].vertex] = 0;

        }

        heap.**pop\_back**();

        if (!heap.**empty**())

**heapifyDown**(0);

        return minVertex;

    }

    void **decreaseKey**(int vertex, double newPriority) override

    {

        if (!**contains**(vertex))

            return;

        size\_t index = vertexToIndex[vertex];

        if (newPriority >= heap[index].priority)

            return;

        heap[index].priority = newPriority;

**heapifyUp**(index);

    }

    bool **contains**(int vertex) const override

    {

        return vertexToIndex.**find**(vertex) != vertexToIndex.**end**();

    }

    bool **isEmpty**() const override

    {

        return heap.**empty**();

    }

};

// 工厂方法定义

PriorityQueue \*PriorityQueue::**create**()

{

    return new **MinHeapPQ**();

}

**code.cpp**

#include <iostream>

#include <vector>

#include <queue>

#include <limits>

#include <fstream>

#include <iomanip>

#include <stack>

#include <string>

#include "mypq.h"

using namespace std;

const double INF = numeric\_limits<double>::**infinity**();

class Graph

{

private:

    int V;

    vector<vector<pair<int, double>>> adj;

public:

**Graph**(int vertices) : **V**(vertices), **adj**(vertices) {}

    void **addDirectedEdge**(int u, int v, double weight)

    {

        adj**[**u**]**.**emplace\_back**(v, weight);

}

    bool **dijkstra**(int src, vector<double> &dist, vector<int> &prev)

    {

        dist.**assign**(V, INF);

        prev.**assign**(V, -1);

        dist**[**src**]** = 0;

        PriorityQueue \*pq = PriorityQueue::**create**();

        pq->**insert**(src, 0);

        while (!pq->**isEmpty**())

        {

            int u = pq->**extractMin**();

            for (const auto &edge : adj**[**u**]**)

            {

                int v = edge.first;

                double weight = edge.second;

                if (weight < 0)

                {

                    cout **<<** "Error: Dijkstra encountered negative edge weight (" **<<** u **<<** "->" **<<** v **<<** "=" **<<** weight **<<** ")\n";

                    delete pq;

                    return false;

                }

                if (dist**[**v**]** > dist**[**u**]** + weight)

                {

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

                    prev**[**v**]** = u;

                    if (pq->**contains**(v))

                        pq->**decreaseKey**(v, dist**[**v**]**);

                    else

                        pq->**insert**(v, dist**[**v**]**);

                }

            }

        }

        delete pq;

        return true;

    }

    bool **bellmanFord**(int src, vector<double> &dist, vector<int> &prev)

    {

        dist.**assign**(V, INF);

        prev.**assign**(V, -1);

        dist**[**src**]** = 0;

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

        {

            bool updated = false;

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

            {

                if (dist**[**u**]** == INF)

                    continue;

                for (const auto &edge : adj**[**u**]**)

                {

                    int v = edge.first;

                    double weight = edge.second;

                    if (dist**[**v**]** > dist**[**u**]** + weight)

                    {

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

                        prev**[**v**]** = u;

                        updated = true;

                    }

                }

            }

            if (!updated)

                break;

        }

        for (int u = 0; u < V; ++u)

        {

            if (dist**[**u**]** == INF)

                continue;

            for (const auto &edge : adj**[**u**]**)

            {

                int v = edge.first;

                double weight = edge.second;

                if (dist**[**v**]** > dist**[**u**]** + weight)

                {

                    return false;

                }

            }

        }

        return true;

    }

    // Return path string like "<0,1,2>" including src and dest

    string **getPath**(int src, int dest, const vector<int> &prev, const vector<double> &dist)

    {

        if (dist**[**dest**]** == INF)

            return "< >";

        stack<int> pathStack;

        for (int cur = dest; cur != -1; cur = prev**[**cur**]**)

            pathStack.**push**(cur);

        if (pathStack.**top**() != src)

            return "< >";

        string pathStr = "<";

        while (!pathStack.**empty**())

        {

            pathStr **+=** **to\_string**(pathStack.**top**());

            pathStack.**pop**();

            if (!pathStack.**empty**())

                pathStr **+=** ",";

        }

        pathStr **+=** ">";

        return pathStr;

    }

    void **printShortestPaths**(int src, const vector<double> &dist, const vector<int> &prev)

    {

        cout **<<** "The shortest-length path is:\n";

        for (int i = 0; i < V; ++i)

        {

            if (dist**[**i**]** == INF)

                cout **<<** i **<<** "=INF; path = < >\n";

            else

            {

                cout **<<** i **<<** "=" **<<** **fixed** **<<** **setprecision**(0) **<<** dist**[**i**]** **<<** "; path = " **<<** **getPath**(src, i, prev, dist) **<<** "\n";

            }

        }

    }

};

int **main**(int argc, char \*argv[])

{

    if (argc != 2)

    {

        cerr **<<** "Usage: " **<<** argv[0] **<<** " <input\_file>\n";

        return 1;

    }

    ifstream **inputFile**(argv[1]);

    if (**!**inputFile)

    {

        cerr **<<** "Error: Cannot open input file\n";

        return 1;

    }

    int V, src;

    inputFile **>>** V **>>** src;

    Graph **g**(V);

    int u, v;

    double weight;

    while (inputFile **>>** u **>>** v **>>** weight)

    {

        g.**addDirectedEdge**(u, v, weight);

    }

    vector<double> dist;

    vector<int> prev;

    cout **<<** "=== Dijkstra Algorithm ===\n";

    if (g.**dijkstra**(src, dist, prev))

    {

        g.**printShortestPaths**(src, dist, prev);

    }

    else

    {

        cout **<<** "Dijkstra algorithm terminated due to negative edge weight\n";

    }

    cout **<<** "\n=== Bellman-Ford Algorithm ===\n";

    if (g.**bellmanFord**(src, dist, prev))

    {

        g.**printShortestPaths**(src, dist, prev);

    }

    else

    {

        cout **<<** "Warning: Negative cycle detected, program terminating\n";

        return 1;

    }

    return 0;

}