資料結構與圖論演算法

Title and Author.

Assignment 9

110704008 羅瑋翔

xianglo9121.mg10@nycu.edu.tw

Statement of the Problem.

The assignment instructs us to design and implement an efficient data structure and algorithm for computing the shortest path from a source vertex to a destination vertex in a weighted graph.

And here are the reasons that we choose Dijkstra’s algorithm and priority queue to implement :

1. **Optimality**: Dijkstra's algorithm guarantees the optimality of the solution by ensuring that the shortest path to each vertex is found in a systematic manner. At each step, it selects the vertex with the currently shortest known distance, and the algorithm iteratively explores and updates distances until the shortest paths to all vertices are determined.
2. **Non-negative Edge Weights**: Dijkstra's algorithm is particularly effective when dealing with graphs where all edge weights are non-negative. However, in scenarios where all weights are non-negative, Dijkstra's algorithm provides an efficient and accurate solution.
3. **Greedy Strategy**: The algorithm follows a greedy strategy by always choosing the vertex with the minimum tentative distance at each step. This ensures that the algorithm focuses on the most promising paths, contributing to its efficiency.
4. **Priority Queue for Efficient Operations**: The use of a priority queue allows for efficient retrieval of the vertex with the minimum distance during each iteration. This ensures that the algorithm can quickly identify and process the most promising vertices, reducing overall computation time.
5. **Single-Source Shortest Path**: Dijkstra's algorithm is well-suited for scenarios where the goal is to find the shortest path from a single source vertex to all other

vertices in the graph. This property is valuable in various applications, such as routing algorithms in computer networks or navigation systems.

The main concepct of designing the input data is only depent on the complexity of the graph in this assignment.

Main Results.

(a)Description of the Program.

Program Description:

1. Data Structures and Classes:

- `Edge` **Struct:**
  - Represents an edge in the graph.
  - Contains information about the destination vertex and the weight of the edge.
- `Vertex` **Struct:**
  - Represents a vertex in the graph.
  - Contains an identifier (`id`) and the current distance from the source vertex.
- `CompareVertices` **Struct:**
  - Custom comparison function used for the priority queue. It ensures that vertices with smaller distances come first.
- `MinHeap` **Class:**
  - Manually implemented min-heap used as a priority queue.
  - Supports operations such as `push`, `top`, `pop`, and `empty`.

2. Dijkstra's Algorithm:

- `dijkstra` **Function:**
  - Implements Dijkstra's algorithm to find the shortest path from a source vertex to a destination vertex.
  - Uses a manually implemented min-heap to efficiently extract the vertex with the smallest tentative distance.

    - Performs relaxation steps for each neighboring vertex to update the shortest distances.
    - Outputs the shortest path distance from the source to the destination.

3. `main` Function:

- Takes user input for the number of vertices (`numVertices`), and constructs an empty graph with the specified number of vertices.
- Accepts user input for the graph's edges, where each line represents a vertex and its outgoing edges (destination and weight).
- Takes user input for the source and destination vertices.
- Calls the `dijkstra` function to compute and output the shortest path distance.

Purpose and Construction of Data Structures:

- **Purpose:**
    - The program is designed to find the shortest path in a weighted graph from a specified source vertex to a destination vertex.
- **Data Structures:**
    - `Edge` and `Vertex` structures are used to represent the elements of the graph.
    - The `MinHeap` class is manually implemented to serve as a priority queue for efficiently selecting vertices with the smallest tentative distances during the algorithm.
- **Why Construct the Data Structure:**
    - Dijkstra's algorithm requires maintaining a priority queue to efficiently select vertices with the smallest distances. A min-heap is chosen for this purpose, and since the standard priority queue is not used, a custom min-heap (`MinHeap`) is implemented.
    - The `Edge` and `Vertex` structures are used to organize and store information about the graph elements, facilitating the algorithm's execution and readability of the code.

Overall, the program provides an implementation of Dijkstra's algorithm with a manually implemented min-heap for finding the shortest path in a weighted graph. The data structures are constructed to represent the graph and support the algorithm's efficient execution.

(b)Data Structures Used in the Program

Manually Implemented Min-Heap (`MinHeap` Class):

- **Purpose:**
  - A min-heap is utilized to maintain a priority queue of vertices based on their tentative distances from the source vertex. This allows the algorithm to quickly extract the vertex with the smallest distance at each step, optimizing the selection process.
- **Implementation:**
  - The `MinHeap` class is designed to represent a binary min-heap.
  - It includes functions for `push`, `top`, `pop`, and `empty` operations.
  - The `push` operation adds a new vertex to the heap and adjusts the heap structure to maintain the min-heap property.
  - The `top` operation returns the vertex with the smallest distance without removing it from the heap.
  - The `pop` operation removes the vertex with the smallest distance from the heap while maintaining the min-heap property.
  - The `empty` operation checks if the heap is empty.
- **Construction:**
  - The `MinHeap` class is constructed using a `vector` to store the vertices.
  - The heap is maintained by adjusting the elements based on their distances.
- **Custom Comparison Function:**
  - The `CompareVertices` struct defines a custom comparison function that ensures the min-heap property. It is used to compare vertices based on their distances.
- **Why Construct the Min-Heap:**
  - Dijkstra's algorithm requires efficient retrieval of the vertex with the smallest tentative distance during each iteration. Using a priority queue, implemented as a min-heap, allows for constant-time retrieval of the minimum element, optimizing the algorithm's overall performance.

In summary, the manually implemented min-heap provides an efficient data structure for organizing and selecting vertices with the smallest distances, crucial for the efficient execution of Dijkstra's algorithm.

(c)Program with Comments

```
#include <iostream>
#include <vector>

using namespace std;

const int INF = 1e9; // Represents infinity

// Define a structure to represent an edge in the graph
struct Edge {
    int destination;
    int weight;

    Edge(int dest, int w) : destination(dest), weight(w) {}
};

// Define a structure to represent a vertex in the graph
struct Vertex {
    int id;
    int distance;

    Vertex(int i) : id(i), distance(INF) {}
};

// Custom comparison function for the binary heap
struct CompareVertices {
    bool operator()(const Vertex& v1, const Vertex& v2) const
        return v1.distance > v2.distance;
    }
};

// Manually implemented min-heap
class MinHeap {
```

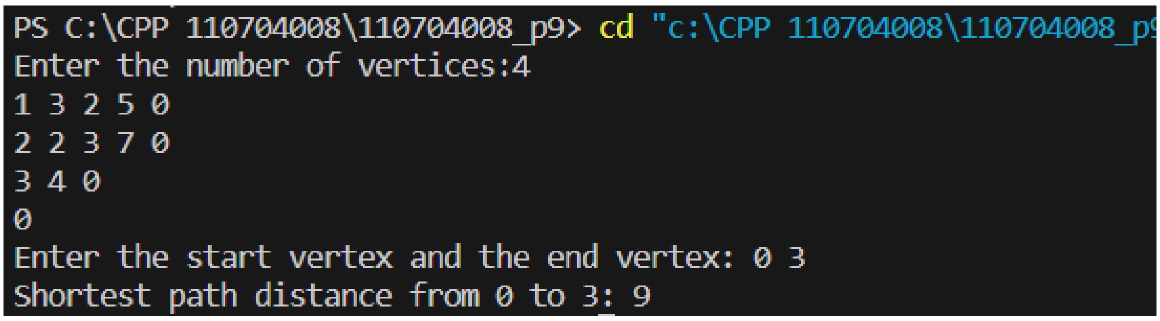
```
public:
    vector<Vertex> heap;

    // Push a vertex onto the heap
    void push(Vertex v) {
        heap.push_back(v);
        int index = heap.size() - 1;
        while (index > 0) {
            int parent = (index - 1) / 2;
            if (heap[parent].distance <= heap[index].distance
                break;
            }
            swap(heap[parent], heap[index]);
            index = parent;
        }
    }

    // Get the vertex with the smallest distance without remo
    Vertex top() const {
        return heap[0];
    }

    // Pop the vertex with the smallest distance from the hea
    void pop() {
        if (heap.empty()) {
            return;
        }
        heap[0] = heap.back();
        heap.pop_back();
        int index = 0;
        while (true) {
            int leftChild = 2 * index + 1;
            int rightChild = 2 * index + 2;
            int smallest = index;
            if (leftChild < heap.size() && heap[leftChild].di
                smallest = leftChild;
            }
            if (rightChild < heap.size() && heap[rightChild].
                smallest = rightChild;
            }
            if (smallest == index) {
                break;
            }
            swap(heap[index], heap[smallest]);
            index = smallest;
        }
    }

    // Check if the heap is empty
    bool empty() const {
        return heap.empty();
    }
};

// Dijkstra's algorithm function with a manually implemented 
void dijkstra(vector<vector<Edge>>& graph, int source, int de
    int numVertices = graph.size();

    // Create a manually implemented min-heap for vertices
    MinHeap pq;

    // Create an array to store distances from source to each
    vector<int> distance(numVertices, INF);

    // Initialize source vertex
    distance[source] = 0;
    pq.push(Vertex(source));

    while (!pq.empty()) {
        Vertex currentVertex = pq.top();
        pq.pop();

        int u = currentVertex.id;
        // Explore neighboring vertices
        for (const Edge& edge : graph[u]) {
            int v = edge.destination;
```

}

```
            int w = edge.weight;

            // Relaxation step
            if (distance[u] + w < distance[v]) {
                distance[v] = distance[u] + w;
                pq.push(Vertex(v));
            }
        }
    }

    // Output the shortest path distance from source to desti
    cout << "Shortest path distance from " << source << " to 
}

int main() {
    int numVertices, vertex_1 = 0, vertex_2, weight, source, 
    cout << "Enter the number of vertices:";
    cin >> numVertices;
    // Create a vector of vectors to represent the graph
    vector<vector<Edge>> graph(numVertices);

    // Input the edges and its weight of the graph
    for(int i = 0; i < numVertices; i++){
        while(cin >> vertex_2 && vertex_2 != 0){
            cin >> weight;
            graph[vertex_1].push_back(Edge(vertex_2, weight))
        }
        vertex_1++;
    }

    cout << "Enter the start vertex and the end vertex: ";
    cin >> source >> destination;

    dijkstra(graph, source, destination);

    return 0;
}
```

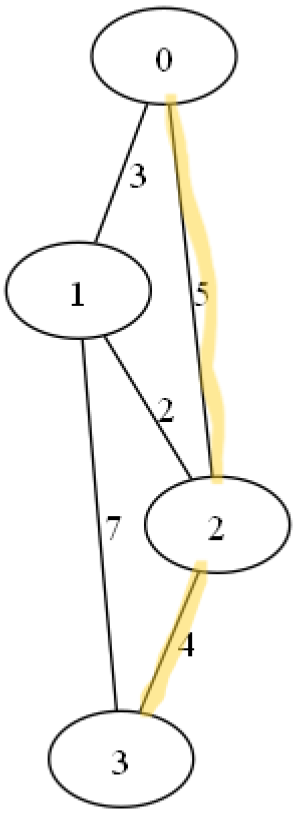
(d)Outputs and the Executions

1st input data and its output

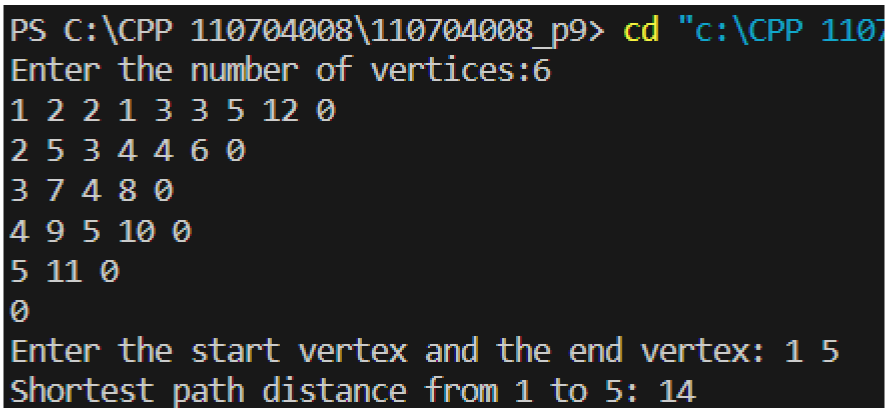


```
PS C:\CPP 110704008\110704008_p9> cd "c:\CPP 110704008\110704008_p9
Enter the number of vertices:4
1 3 2 5 0
2 2 3 7 0
3 4 0
0
Enter the start vertex and the end vertex: 0 3
Shortest path distance from 0 to 3: 9
```

Weighted Graph with shortest path from source to destination

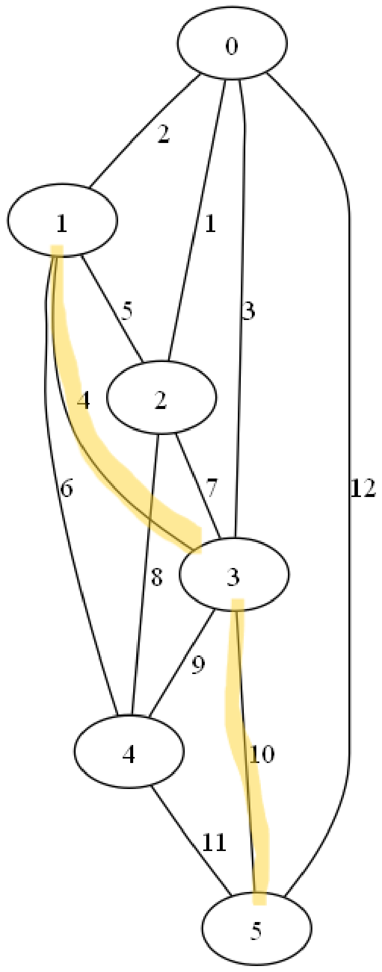


2nd input data and its output

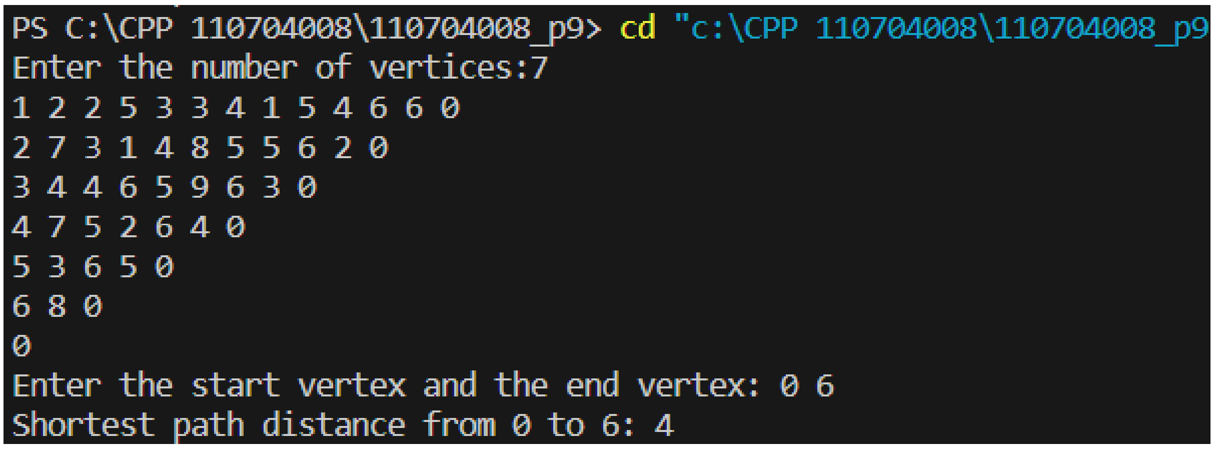


```
PS C:\CPP 110704008\110704008_p9> cd "c:\CPP 1107
Enter the number of vertices:6
1 2 2 1 3 3 5 12 0
2 5 3 4 4 6 0
3 7 4 8 0
4 9 5 10 0
5 11 0
0
Enter the start vertex and the end vertex: 1 5
Shortest path distance from 1 to 5: 14
```

Weighted Graph with shortest path from source to destination

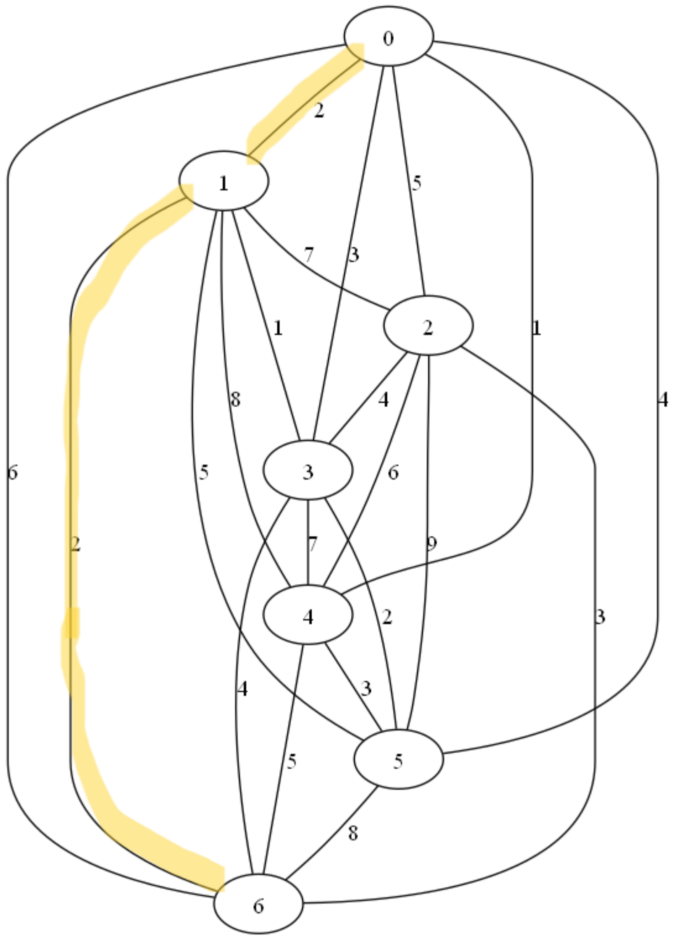


3rd input data and its output



```
PS C:\CPP 110704008\110704008_p9> cd "c:\CPP 110704008\110704008_p9
Enter the number of vertices:7
1 2 2 5 3 3 4 1 5 4 6 6 0
2 7 3 1 4 8 5 5 6 2 0
3 4 4 6 5 9 6 3 0
4 7 5 2 6 4 0
5 3 6 5 0
6 8 0
0
Enter the start vertex and the end vertex: 0 6
Shortest path distance from 0 to 6: 4
```

Weighted Graph with shortest path from source to destination



Conclusions.

Interesting Observations:

- The manual implementation of the min-heap illustrates the core operations of a priority queue, enhancing the understanding of heap-based data structures.

- Dijkstra's algorithm, when combined with a priority queue, efficiently finds the shortest path in weighted graphs.
- The program provides a good hands-on experience in implementing fundamental graph algorithms and data structures.

I have learned :

1. **Dijkstra's Algorithm:**
    - Reinforced understanding of Dijkstra's algorithm, a widely used algorithm for finding the shortest paths between nodes in a graph.
    - Emphasized the importance of maintaining a priority queue to efficiently select the next vertex with the smallest tentative distance.
2. **Priority Queue Implementation:**
    - Demonstrated the manual implementation of a min-heap, a fundamental data structure used as a priority queue in Dijkstra's algorithm.
    - Enhanced understanding of heap operations such as insertion, extraction, and maintaining the heap property.

Overall, this assignment provides insights into the mechanics of priority queues, heap operations, and the application of Dijkstra's algorithm for finding shortest paths in weighted graphs.