資料結構與論演算法

Title and Author.

Assignment 8

110704008 羅瑋翔

xianglo9121.mg10@nycu.edu.tw

Statement of the Problem.

This assignment requires designing and implementing a data structure for calculating the minimum spanning tree of a weighted graph using Kruskal's algorithm. Preferably, the implementation should involve disjoint set data structures.

The reasons we implemented the Kruskal’s algorithm here are

1. **Efficiency:** Kruskal's algorithm is known for its efficiency in finding the MST of a graph. It is a greedy algorithm that works well for sparse graphs. The time complexity of Kruskal's algorithm is O(E log V), where E is the number of edges and V is the number of vertices. This makes it suitable for practical applications.
2. **Ease of Implementation:** Kruskal's algorithm is relatively easy to understand and implement. It involves sorting the edges based on their weights and then iteratively adding the smallest edge that does not form a cycle in the growing MST. This simplicity makes it a popular choice for educational purposes and practical implementations.

And disjoint set data structures are employed to efficiently detect cycles in the graph during the process of building the MST. Detecting cycles is a crucial step to ensure that the algorithm adds edges without creating cycles in the resulting tree. Disjoint set data structures offer efficient union and find operations, making them well-suited for this purpose.

This assignment includes three types of weighted graphs for experimentation:

1. Four vertices with five edges.
2. Six vertices with twelve edges.
3. Seven vertices with twenty-one edges, forming a complete graph.

The reason for designing these variations is to test whether the program exhibits errors as the complexity of the graphs gradually increases.

Furthermore, output all the weighted graphs as PNG files to confirm the correctness of the minimum spanning tree.

Main Results.

(a)Description of the Program.

I have constructed two structures, one named "Edge" with three parameters representing the two endpoints and the weight of the edge, and the other named "Subset" with two parameters one represents the parent of a particular vertex in the disjoint-set. One keeps track of the height of the tree rooted at a particular vertex. The rank is initially set to 0 for all vertices. During the union operation, if the ranks of the two sets being united are equal, one of the ranks is incremented, indicating that the height of the resulting tree has increased.

Then, I construct a class Graph with 2 variables represent Vertices and Edges of the graph and a vector to store the edges (the struct i construct before).

There are seven functions within this class:

1. A function for adding weighted edges.
2. A function for finding the parent of a subset.
3. A function for merging subsets.
4. A function for comparing edge weights.
5. A function implementing the partition step of QuickSort.
6. A function for visualizing the weighted graph.
7. A function implementing Kruskal's algorithm to find the Minimum Spanning Tree (MST).

Then, in the main function we only input the file that contains the weighted graph. And call the functions to help us find the minimum spanning tree (MST).

(b)Data Structures Used in the Program

The class I construct contains:

1. Disjoint-Set (Union-Find) Data Structure:

The disjoint-set data structure is used to efficiently manage sets of vertices and determine whether adding an edge between two vertices would create a cycle. This is crucial for Kruskal's algorithm, which builds the minimum spanning tree by selecting edges that do not form cycles.

2. Kruskal's Algorithm:

Kruskal's algorithm is a greedy algorithm that builds the minimum spanning tree by selecting edges in non-decreasing order of their weights. The disjoint-set data structure is used to efficiently check for cycles while selecting edges.

In summary, the combination of Kruskal's algorithm and the disjoint-set data structure provides an efficient solution for finding minimum spanning trees in graphs, particularly in scenarios where the graph is represented as a set of edges with associated weights.

(c)Program with Comments

```
#include <iostream>
#include <vector>
#include <string>
#include <fstream>

using namespace std;

struct Edge
{
    int src, dest, weight;
};

struct Subset
{
    int parent, rank;
};

class Graph
{
public:
```

```
    int V, E;
    vector<Edge> edges;
    Graph(int v, int e);
    void addEdge(int src, int dest, int weight);
    int find(Subset subsets[], int i);
    void Union(Subset subsets[], int x, int y);
    void Sort(int low, int high);
    int partition(int low, int high);
    void generateGraphvizDot(const string &filename);
    void KruskalMST();
};

int main()
{
    int numV, numE;
    int V, E, W;
    string filename;
    cin >> filename;
    ifstream inFile(filename);
    inFile >> numV >> numE;
    Graph graph(numV, numE);

    while(!inFile.eof() && inFile >> V >> E >> W){
        graph.addEdge(V, E, W);
    }

    graph.KruskalMST();
    inFile.close();

    return 0;
}

Graph::Graph(int v, int e) : V(v), E(e) {}

void Graph::addEdge(int src, int dest, int weight)
{
    edges.push_back({src, dest, weight});
}
```

```
int Graph::find(Subset subsets[], int i)
{
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);
    return subsets[i].parent;
}

void Graph::Union(Subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    //Attach smaller rank tree under root of high rank tree
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;
    else
    {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Custom sorting function using quicksort
void Graph::Sort(int low, int high)
{
    //Iterate through edges and sort them by weight
    if (low < high)
    {
        int pivot = partition(low, high);
        Sort(low, pivot - 1);
        Sort(pivot + 1, high);
    }
}

int Graph::partition(int low, int high)
```

```
{
    int pivot = edges[high].weight;
    int i = low - 1;

    //Iterate through edges
    for (int j = low; j < high; j++)
    {
        if (edges[j].weight <= pivot)
        {
            i++;
            swap(edges[i], edges[j]);
        }
    }

    swap(edges[i + 1], edges[high]);
    return i + 1;
}

// this part enables me to visualize the graph
void Graph::generateGraphvizDot(const string &filename)
{
    ofstream dotFile(filename);
    if (!dotFile.is_open())
    {
        cerr << "Error opening DOT file for writing." << endl
        return;
    }

    dotFile << "graph MST {\n";
    for (const Edge &edge : edges)
    {
        dotFile << edge.src << " -- " << edge.dest << " [labe
    }
    dotFile << "}\n";

    dotFile.close();
}
```

```
//this function uses Kruskal's algorithm to find the minimum
void Graph::KruskalMST()
{
    Sort(0, E - 1);

    vector<Edge> result;
    Subset *subsets = new Subset[V];

    //Initialize subsets
    for (int i = 0; i < V; i++)
    {
        subsets[i].parent = i;
        subsets[i].rank = 0;
    }

    //Iterate through sorted edges
    for (const Edge &edge : edges)
    {
        int x = find(subsets, edge.src);
        int y = find(subsets, edge.dest);

        if (x != y)
        {
            result.push_back(edge);
            Union(subsets, x, y);
        }
    }

    //Print result
    cout << "Minimum Spanning Tree:\n";
    for (const Edge &edge : result)
    {
        cout << edge.src << " -- " << edge.dest << " : " << e
    }

    generateGraphvizDot("graphviz_output.dot");
```

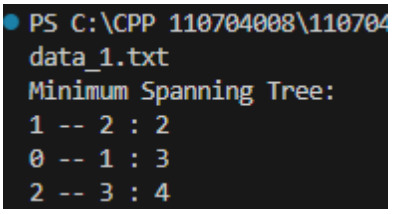
```
    delete[] subsets;
}
```

(d)Outputs and the Executions

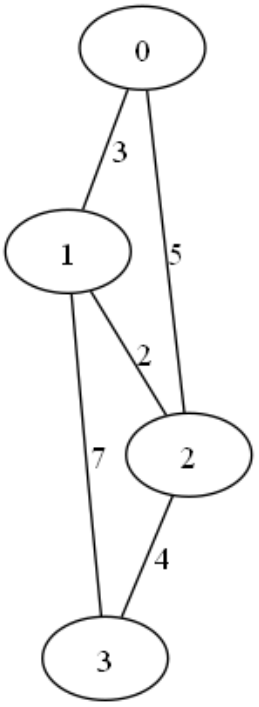
1st input data

4 5
0 1 3
0 2 5
1 2 2
1 3 7
2 3 4

Output



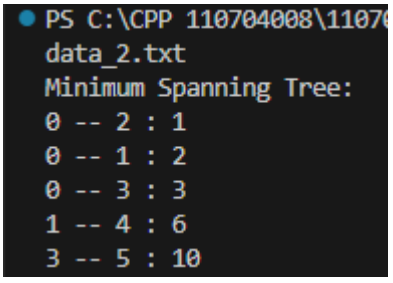
Weighted Graph



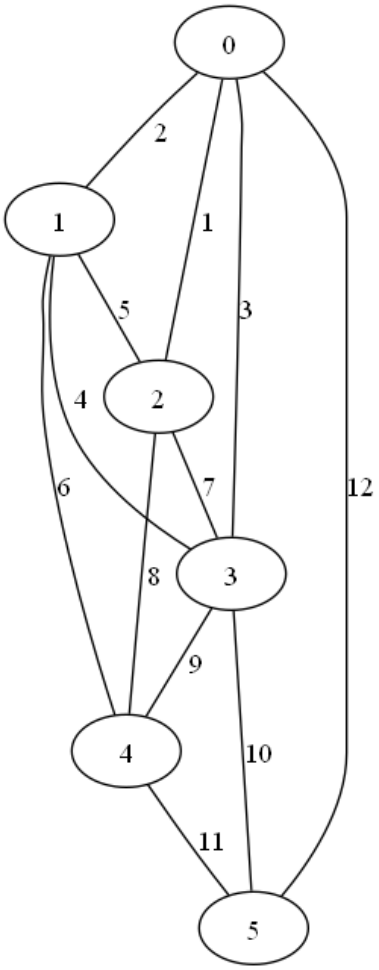
2nd input data

6 12
0 1 2
0 2 1
0 3 3
1 2 5
1 3 4
1 4 6
2 3 7
2 4 8
3 4 9
3 5 10
4 5 11
5 0 12

Output



Weighted Graph



3rd input data

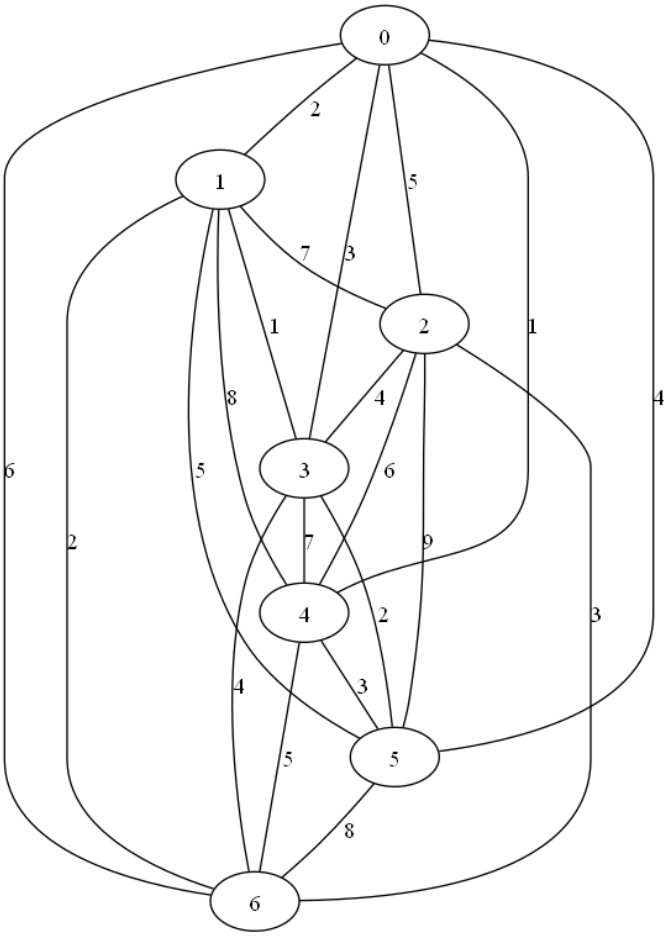
7 21
0 1 2
0 2 5
0 3 3
0 4 1
0 5 4
0 6 6
1 2 7
1 3 1
1 4 8
1 5 5
1 6 2
2 3 4
2 4 6
2 5 9
2 6 3

3 4 7
3 5 2
3 6 4
4 5 3
4 6 5
5 6 8

Output

```
PS C:\CPP 110704008\1107040
data_3.txt
Minimum Spanning Tree:
0 -- 4 : 1
1 -- 3 : 1
0 -- 1 : 2
1 -- 6 : 2
3 -- 5 : 2
2 -- 6 : 3
```

Weighted Graph



Conclusions.

Here are some notable aspects of my implementation:

1. **File Input/Output:** You've designed your program to read the graph information from an input file and generate the minimum spanning tree and its visualization in the DOT format in an output file. This makes your program flexible and allows you to use different graphs without modifying the code.
2. **Graph Representation:** Your program represents the graph using an edge list, where each edge is defined by its source, destination, and weight. This representation is suitable for Kruskal's algorithm, which involves sorting the edges based on their weights.
3. **Disjoint Set Data Structure:** You've used a disjoint-set data structure to efficiently keep track of connected components and perform union-find operations. This is crucial for Kruskal's algorithm to determine whether adding an edge forms a cycle in the spanning tree or not.
4. **Quicksort:** You've implemented a custom sorting function using the Quicksort algorithm to sort the edges based on their weights. Quicksort is a suitable choice for this scenario, considering the average-case time complexity.
5. **Graph Visualization with Graphviz:** You've included a function to generate a DOT file for the minimum spanning tree, allowing for easy visualization using tools like Graphviz. This is helpful for understanding the structure of the resulting spanning tree.
6. **Error Handling:** You've included basic error handling, such as checking if the input file is successfully opened and closed. This improves the robustness of your program.

My program demonstrates the importance of algorithmic efficiency, data structure choice, and the significance of representing the graph appropriately for the specific algorithm being employed. Kruskal's algorithm, with the help of disjoint-set operations, efficiently finds the minimum spanning tree by considering edges in non-decreasing order of weight. The inclusion of visualization adds an extra layer of understanding and makes the algorithm's output more accessible. Overall, this assignment contains several good practices in software design and algorithm implementation, this makes me learned a lot of things that I do not know before, such as the disjoint set data structure.