Combinatorial Optimization Report

Vo Van Duong - 51702082

Huynh Phi Ho - 51702102

Faculty of Information and Technology

Ton Duc Thang University

This report is submitted for the assignment of

Combinatorics and Graphs Subject

Ton Duc Thang University April 2019

Table of contents

List of figures

Introduction

Combinatorial optimization is use combinatorial techniques to solve discrete optimization problems by searching to determine the best possible solution from a finite set of some possibilities.

In this report, we research about three basic algorithms based on solving the combinatorial optimization, included maximum network flow, shortest path, and minimum spanning tree. We also design the interface for input from the user. Some basic features in this demo version such as Add, Insert, Delete, Dropout, Change parameters,...

The language we use for programming is Java.

Chapter 1

Introduction

1.1 Maximum Network Flow

In optimization theory, maximum flow problems involve finding a feasible flow through a single-source, single-sink flow network that is maximum.[3]

Each edge has a non-negative capacity. If two nodes are not connected then the capacity of them are zero. We have a source s, and a sink t.....

1.1.1 History of Maximum Network Flow

- In 1954, T. E. Harris and F. S. Ross published a simplified model of Soviet railway traffic flow. It is the maximum flow problem was first formulated.[3]
- In 1955, Lester R. Ford, Jr. and Delbert R. Fulkerson created the first known algorithm, the Ford–Fulkerson algorithm.[3]
- Over the years, various improved solutions were discovered: notably the shortest augmenting path algorithm of Edmonds and Karp and independently Dinitz; the blocking flow algorithm of Dinitz; the push-relabel algorithm of Goldberg and Tarjan; the binary blocking flow algorithm of Goldberg and Rao. The algorithms of Sherman and Kelner, Lee, Orecchia and Sidford, respectively, find an approximately optimal maximum flow but only work in undirected graphs.[3]

1.1.2 Definition

Maximum network flow is defined as the maximum amount of flow that the network would allow to flow from source to sink.

1.1.3 Application

There are many ways which maximum network flow is used for, such as:

1.1.4 Solutions

There are many various algorithms for solving the maximum flow problem. But in this report, we only find out about Ford-Fulkerson algorithm.

1.2 Shortest Path

In graph theory, the shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.[5]

1.2.1 Definition

The shortest path problem can be defined for graphs whether undirected, directed, or mixed. It is defined here for undirected graphs; for directed graphs the definition of path requires that consecutive vertices be connected by an appropriate directed edge.

1.2.2 Application

There are many ways which maximum network flow is used for, such as:

1.2.3 Solutions

The most important algorithms for solving this problem are:[5]

- Bellman–Ford algorithm solves the single-source problem if edge weights may be negative.
- Dijkstra's algorithm solves the single-source shortest path problem with non-negative edge weight.
- A* search algorithm solves for single pair shortest path using heuristics to try to speed up the search.
- Floyd–Warshall algorithm solves all pairs shortest paths.

- Johnson's algorithm solves all pairs shortest paths, and may be faster than Floyd–Warshall on sparse graphs.
- Viterbi algorithm solves the shortest stochastic path problem with an additional probabilistic weight on each node.

In this report, we only find out about Dijkstra algorithm.

1.3 Minimum Spanning Tree

A minimum spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. That is, it is a spanning tree whose sum of edge weights is as small as possible. More generally, any edge-weighted undirected graph (not necessarily connected) has a minimum spanning forest, which is a union of the minimum spanning trees for its connected components.[4]

Chapter 2

The State-of-The-Art

2.1 Ford–Fulkerson algorithm

2.1.1 Overview

This is a algorithm that computes the maximum flow in a flow network (as we introduced in section 1.1).
It was published in 1956 by L. R. Ford, Jr. and D. R. Fulkerson. The name "Ford–Fulkerson" is often also used for the Edmonds–Karp algorithm, which is a fully defined implementation of the Ford–Fulkerson method.[2]

2.1.2 Algorithm

2.1.3 Implementation in Java

Source code ord–Fulkerson class in Java language:

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Writer;
import java.util.LinkedList;
import java.util.Queue;
import java.util.Scanner;

public class MaximumNetworkFlow {
    public Writer fileOut;
```

```
private Scanner fileIn;
private FileReader reader;
private int[][] adjacencyMatrix;
private int vertex;

public MaximumNetworkFlow(String outPut) {
    try {
        fileOut = new FileWriter(outPut);
    } catch (IOException e) {
        // TODO: handle exception
        System.out.println(e);
    }
}

private void readerFile(String intput) {
    try {
        reader = new FileReader(intput);
        fileIn = new Scanner(reader);
        vertex = Integer.parseInt(fileIn.nextLine());
        adjacencyMatrix = new int[vertex][vertex];
        // fileIn.hasNextLine();
        // System.out.println(vertex);
        int i = 0;
        while (fileIn.hasNextLine()) {
            String chars = fileIn.nextLine();
            String[] temp = chars.split(" ");
            for (int j = 0; j < temp.length; j++) {
                String replace = temp[j].toUpperCase();
                if (!replace.equals("M"))
                    adjacencyMatrix[i][j] = Integer.parseInt(replace);
                // System.out.println(replace);
                else
                    adjacencyMatrix[i][j] = 0;
            }
            i++;
        }
```

```
            fileIn.close();
        } catch (IOException e) {
            // TODO: handle exception
            System.out.println(e);
        }
    }

    private void fordFulkerson(int src, int des){
        int u, v;

        int [][]tempGraph = new int[vertex][vertex]; // change weight in adjacency

        for (u = 0; u < vertex; u++) {
            for (v = 0; v < vertex; v++) {
                tempGraph[u][v] = adjacencyMatrix[u][v];
            }
        }

        int []parent = new int[vertex]; // Root
        int max_flow = 0;

        while(bfs(tempGraph, src, des, parent)){ // true if visit(des) == true and
            int pathFlow = Integer.MAX_VALUE;
            for (v = des; v != src; v= parent[v]) { // stop when 5 != 0
                //System.out.println(v);
                //System.out.println(parent[v]);
                u = parent[v];
               // System.out.println(u);
                pathFlow = Math.min(pathFlow, tempGraph[u][v]); // set pathFlow.
            }

            for (v=des; v != src; v=parent[v])
            {
                u = parent[v];
               // System.out.println(u + " " + v + "  " + tempGraph[u][v] + " " + p
                tempGraph[u][v] -= pathFlow;
```

```
                tempGraph[v][u] += pathFlow;
            }
            max_flow += pathFlow;
        }
        try {
            fileOut.write("maximum value: " + max_flow + System.lineSeparator());
            fileOut.write("Minimum Cut: " + System.lineSeparator());
            boolean[] isVisited = new boolean[vertex];
            dfs(tempGraph, src, isVisited);
            for (int i = 0; i < vertex; i++) {
                for (int j = 0; j < vertex; j++) {
                    if(adjacencyMatrix[i][j] > 0 && isVisited[i] && !isVisited[j]){
                      //  System.out.println(i +" " + j);
                      fileOut.write("\t" + i + " -- " + j + System.lineSeparator());
                    }
                }
            }
        } catch (IOException e) {
            //TODO: handle exception
        }

    }

    private boolean bfs(int [][]tempGraph, int source, int destination, int []parent)
        boolean []visited = new boolean[vertex];
        Queue<Integer> q = new LinkedList<>();
        q.add(source);
        visited[source] = true;
        parent[source] = -1;

        while(!q.isEmpty()){
            int v = q.poll();
            for (int i = 0; i < vertex; i++) {
                if(tempGraph[v][i] > 0 && !visited[i]){
                    q.offer(i);
                    visited[i] = true;
```

```
                    parent[i] = v;
                }
            }
        }
        return (visited[destination] == true);
    }

    private void dfs(int[][]tempGraph, int s, boolean[]visited){
        visited[s] = true;
        for (int i = 0; i < vertex; i++) {
            if(tempGraph[s][i] > 0 && !visited[i]){
                dfs(tempGraph, i, visited);
            }
        }
    }

    public static void main(String[] args) {

        String inPut = args[0];
        String outPut = args[1];

        try {
            Scanner sc = new Scanner(System.in);
            System.out.print("Enter source: ");
            int source = sc.nextInt();
            System.out.print("Enter Destination: ");
            int destination = sc.nextInt();
            MaximumNetworkFlow MF = new MaximumNetworkFlow(outPut);
            MF.readerFile(inPut);
            //System.out.println();
            MF.fordFulkerson(source, destination);
            MF.fileOut.close();
        } catch (IOException e) {
                // TODO: handle exception
                System.out.println(e);
        }catch (Exception e) {
```

```
            //TODO: handle exception
            System.out.println(e);
        }
    }
}
```

File input:

```
6
0 16 13 0 0 0
0 0 10 12 0 0
0 4 0 0 14 0
0 0 9 0 0 20
0 0 0 7 0 4
0 0 0 0 0 0
```

File output:

```
maximum value: 23
Minimum Cut:
        1 -- 3
        4 -- 3
        4 -- 5
```

2.2 Dijkstra algorithm

2.2.1 Overview

Dijkstra's algorithm (or Dijkstra's Shortest Path First algorithm, SPF algorithm)[1] is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.[1]

2.2.2 Algorithm

2.2.3 Implementation in Java

Source code Dijkstra class in Java language:

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Writer;
import java.util.Scanner;

public class ShortestPath {
    public Writer fileOut;
    private Scanner fileIn;
    private FileReader reader;
    private int[][] adjacencyMatrix;
    private int vertex;

    public ShortestPath(String outPut) {
        try {
            fileOut = new FileWriter(outPut);
        } catch (IOException e) {
            // TODO: handle exception
            System.out.println(e);
        }
    }

    private void readerFile(String intput) {
        try {
            reader = new FileReader(intput);
            fileIn = new Scanner(reader);
            vertex = Integer.parseInt(fileIn.nextLine());
            adjacencyMatrix = new int[vertex][vertex];
            // fileIn.hasNextLine();
            // System.out.println(vertex);
            int i = 0;
            while (fileIn.hasNextLine()) {
                String chars = fileIn.nextLine();
                String[] temp = chars.split(" ");
                for (int j = 0; j < temp.length; j++) {
                    String replace = temp[j].toUpperCase();
```

```
                    if (!replace.equals("M"))
                        adjacencyMatrix[i][j] = Integer.parseInt(replace);
                    // System.out.println(replace);
                    else
                        adjacencyMatrix[i][j] = 0;
                }
                i++;
            }
            fileIn.close();
        } catch (IOException e) {
            // TODO: handle exception
            System.out.println(e);
        }
    }

    private void print(int dist[], int src, int destination, int[] pathTree) {
        try {

            // System.out.println(i);
            fileOut.write("Predicting the shortest path: ");
            printPath(destination, pathTree);
            fileOut.write(System.lineSeparator());
            fileOut.write("Shorest Path form " + src + " to "+ destination+ ": " + di
        } catch (Exception e) {
            // TODO: handle exception
            System.out.println(e);
        }

    }

    private void printPath(int i, int[] pathTree) {
        // System.out.println(i);
        try {
            if (i == -1)
                return;
            printPath(pathTree[i], pathTree);
```

```
            fileOut.write( i + " ");
        } catch (Exception e) {
            // TODO: handle exception
        }
    }

    private int minDistance(int dist[], Boolean checkVertex[]) {
        int min = Integer.MAX_VALUE;
        int min_index = -1;
        for (int i = 0; i < vertex; i++) {
            if (!checkVertex[i] && dist[i] <= min) {
                min = dist[i];
                min_index = i;
            }
        }
        return min_index;
    }

    private void dijkstra(int src, int destination) {
        int[] dist = new int[vertex]; // shortest distance form src -> i
        Boolean[] checkVertex = new Boolean[vertex];

        for (int i = 0; i < vertex; i++) {
            dist[i] = Integer.MAX_VALUE;
            checkVertex[i] = false;
        }

        dist[src] = 0;
        int[] pathTree = new int[vertex];
        pathTree[src] = -1;
        int temp = 0;
        for (int i = 0; i < vertex; i++) {
            temp = minDistance(dist, checkVertex); // min edge and return nearest v

            checkVertex[temp] = true; // vertex did complete
            for (int j = 0; j < vertex; j++) {
```

```
                if (!checkVertex[j] && adjacencyMatrix[temp][j] != 0 && dist[temp] !=
                    pathTree[j] = temp;
                    dist[j] = dist[temp] + adjacencyMatrix[temp][j];
                }
            }
        }
        print(dist, src, destination, pathTree);
    }

    public static void main(String[] args) {

        String inPut = args[0];
        String outPut = args[1];
        try {
            ShortestPath spt = new ShortestPath(outPut);
            Scanner sc = new Scanner(System.in);
            System.out.print("Enter source: ");
            int source = sc.nextInt();
            System.out.print("Enter Destination: ");
            int destination = sc.nextInt();
            spt.readerFile(inPut);
            spt.dijkstra(source, destination);
            spt.fileOut.close();
        } catch (IOException e) {
            // TODO: handle exception
            System.out.println(e);
        }catch(Exception e){
            System.out.println(e);
        }
    }
}
```

File input:

```
9
0 4 0 0 0 0 0 8 0
4 0 8 0 0 0 0 11 0
0 8 0 7 0 4 0 0 2
```

```
0 0 7 0 9 14 0 0 0
0 0 0 9 0 10 0 0 0
0 0 4 0 10 0 2 0 0
0 0 0 14 0 2 0 1 6
8 11 0 0 0 0 1 0 7
0 0 2 0 0 0 6 7 0
```

File output:

2.3 Prim algorithm

2.3.1 Overview

2.3.2 Algorithm

2.3.3 Implementation in Java

Source code Prim class in Java language:

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Writer;
import java.util.Scanner;

public class MST {
    public Writer fileOut;
    private Scanner fileIn;
    private FileReader reader;
    private int[][] adjacencyMatrix;
    private int vertex;

    public MST(String outPut) {
        try {
            fileOut = new FileWriter(outPut);
        } catch (IOException e) {
```

```
            // TODO: handle exception
            System.out.println(e);
        }
    }

    private void readerFile(String intput) {
        try {
            reader = new FileReader(intput);
            fileIn = new Scanner(reader);
            vertex = Integer.parseInt(fileIn.nextLine());
            adjacencyMatrix = new int[vertex][vertex];
            // fileIn.hasNextLine();
            // System.out.println(vertex);
            int i = 0;
            while (fileIn.hasNextLine()) {
                String chars = fileIn.nextLine();
                String[] temp = chars.split(" ");
                for (int j = 0; j < temp.length; j++) {
                    String replace = temp[j].toUpperCase();
                    if (!replace.equals("M"))
                        adjacencyMatrix[i][j] = Integer.parseInt(replace);
                    // System.out.println(replace);
                    else
                        adjacencyMatrix[i][j] = 0;
                }
                i++;
            }
            fileIn.close();
        } catch (IOException e) {
            // TODO: handle exception
            System.out.println(e);
        }
    }

    private void print(int []pathTree, int sum){
        try {
```

```
            fileOut.write("Predict the minimum: " + System.lineSeparator());
            fileOut.write("Edge\tweight" + System.lineSeparator());
            for (int i = 1; i < vertex; i++) {
                fileOut.write(pathTree[i] + "-" + i + "\t" + adjacencyMatrix[i][pat
            }
            fileOut.write("Minimal total weight: " + sum);
        } catch (Exception e) {
            // TODO: handle exception
            System.out.println(e);
        }

    }

    private int minDistance(int dist[], Boolean checkVertex[]) {
        int min = Integer.MAX_VALUE;
        int min_index = -1;
        for (int i = 0; i < vertex; i++) {
            if (!checkVertex[i] && dist[i] < min) {
                min = dist[i];
                min_index = i;
            }
        }
        return min_index;
    }

    private void prim() {
        int[] dist = new int[vertex]; // Shorest distance from 0 to i
        int[] pathTree = new int[vertex];
        Boolean[] checkVertex = new Boolean[vertex];

        for (int i = 0; i < vertex; i++) {
            dist[i] = Integer.MAX_VALUE;
            checkVertex[i] = false;
        }

        dist[0] = 0;
```

```
        pathTree[0] = -1;

        int sum = 0;
        for (int i = 0; i < vertex; i++) {
            int temp = minDistance(dist, checkVertex); // min edge and return nearest
            //System.out.println(temp);

            checkVertex[temp] = true; // vertex did complete
            for (int j = 1; j < vertex; j++) {
                if (adjacencyMatrix[temp][j] != 0 && !checkVertex[j] &&  adjacencyMat
                    pathTree[j] = temp;
                    dist[j] = adjacencyMatrix[temp][j];
                    sum += adjacencyMatrix[temp][j];
                }
            }
        }
        print(pathTree, sum);
    }

    public static void main(String[] args) {
        String inPut = args[0];
        String outPut = args[1];
        try {
            MST mst = new MST(outPut);
            mst.readerFile(inPut);
            mst.prim();
            mst.fileOut.close();
        } catch (IOException e) {
            // TODO: handle exception
            System.out.println(e);
        }
    }
}
```

File input:

```
5
0 2 0 6 0
```

```
2 0 3 8 5
0 3 0 0 7
6 8 0 0 9
0 5 7 9 0
```

File output:

```
Predict the minimum:
Edge        weight
0-1         2
1-2         3
0-3         6
1-4         5
Minimal total weight: 16
```

Chapter 3

Approach

3.1 General Architecture

3.2 Technology stacks

Chapter 4

Experiment and Results

4.1 Layout

Software Layout

4.2 Layout

Chapter 5

Conclusion

Conclusion

In this assignment, we have explored three basic algorithms based on solving the combinatorial optimization:

1. Maximum network flow
2. Shortest path
3. Minimum spanning tree

We also design a demo program to visualize the networks.

References

[1] Dijkstra's algorithm. https://en.wikipedia.org/wiki/Dijkstra's_algorithm. Accessed: April 6th, 2019.

[2] Ford–fulkerson algorithm. https://en.wikipedia.org/wiki/Ford\T1\textendashFulkerson_algorithm. Accessed: April 6th, 2019.

[3] Maximum flow problem. https://en.wikipedia.org/wiki/Maximum_flow_problem. Accessed: March 24th, 2019.

[4] Minimum spanning tree. https://en.wikipedia.org/wiki/Minimum_spanning_tree. Accessed: March 24th, 2019.

[5] Shortest path problem. https://en.wikipedia.org/wiki/Shortest_path_problem. Accessed: April 6th, 2019.