ÖZYEĞİN ÜNİVERSİTESİ

2022

CS 333

Project 3

Network Flow

by: Selin Köleş (S021541)

Table of Content

Introduction

This project aims to implement a network flow solution to make decisions to complete the proposed projects by employees to the CEO of a startup company. This report gives a detailed insight into the given problem statement, a brief introduction to Network Flow Graphs and algorithms to solve them, and the steps I followed to crack the given problem, implementation of my code, and time complexity analysis of the solution.

Problem Statement

Employees of a startup company propose a number of new venture projects, represented by the set P = {p1, p2, …, pn}. Each of these projects has an associated outcome, represented by the set O = {o1, o2, …, on}, which can be either positive (earning money) or negative (costing money).

Some of the projects have prerequisites, represented by the set E (p_i,p_j), which means that certain projects (p_i) cannot be carried out unless other projects (p_j) are also completed. A subset of these projects, represented by S, is considered executable if all of the prerequisites for each project in the subset are also included in the subset.

Flow Network Graph

Network flow problems are a class of computational problems. The goal of network flow problems is to construct a flow with numerical values on each edge that respects capacity constraints.

The input for these problems is a flow network i.e. a graph with numerical capacities on its edges. A flow must satisfy the restriction that the amount of flow into a node equals the amount of flow out of it unless it is a source with only outgoing flow or a sink with the only incoming flow. Such graphical representation of networks helps us find the maximum flow along the whole network. There are many algorithms that can be handy to solve such problems such as Dinic's algorithm, the Edmonds–Karp algorithm, the Ford–Fulkerson algorithm, the network simplex algorithm, the out-of-kilter algorithm for minimum-cost flow, and the push–relabel maximum flow algorithm.

A network flow graph can be used to simulate anything that moves via a network of nodes, including computer network traffic, circulation with demand, fluids in pipelines, currents in electrical circuits, and other related phenomena.

Implementation of code

The solution to this given problem statement is implemented in Java programming language, in OpenJDK-19 environment using IntelliJ IDEA Community Edition 2022.3.

In order to implement this solution, we have five classes named:

- `SELİN_KÖLEŞ_S021541`; the one which contains `void main(String[] args)` method,
- `Vertex`; this class creates an object **vertex** for a graph,
- `Edge`; which implements an object **edge** for the network graph,
- `Graph`; it creates an object **graph** with help of vertices and edges,
- and `FordFulkerson`; which solves the graph and evaluates the maximum profit and successfully completed venture projects.

SELİN_KÖLEŞ_S021541 Class

I put this name because since there are multiple classes the file name should be the same as the public class :)
This class contains a void `main(String[] args)` method, where we implement the whole program by instantiating objects from `Graph` and `FordFulkerson` solver.

Initialization of Scanner and variables

First of all a scanner `Scanner sc = new Scanner(System.in).useDelimiter("\n")` is initialized in order to read input from the keyboard. Then we prompt messages for the user helping using to give the inputs using `System.out.print()` method.

The total number of projects is entered by the user through the keyboard and saved in a variable named `int numOfProjects = sc.nextInt()`. Then we declare three variables `String[] namesOfProjects`, `int[] outcomesOfProjects`, and `ArrayList<ArrayList<String>> preReq` in order to store the names of projects, outcomes of projects, and the prerequisites respectively.

```
String[] namesOfProjects;           // array to store the names of project
int[] outcomesOfProjects;           // array to store outcomes of projects
ArrayList<ArrayList<String>> preReq = new ArrayList<>();   // array to
store the prerequisites
```

In order to scan names of projects, a **`do-while`** loop is started. In the loop, the user enters the names of venture projects, in order to write code for all possible inputs and avoid crashing the program because of wrong inputs, program checks if the entered number of projects are same as which user entered earlier.

Similarly, In order to scan the outcomes of projects, a **`do-while`** loop is started, and the program keeps checking if the user entered the correct number of outcomes. If not, the console prompts an error asking the user to enter the right input.

```
// scanner to get names of projects as input and also checks if the user gives the
correct number of inputs
        do {
            String inputProjects = sc.next();
            namesOfProjects = inputProjects.split(" ");
            if (namesOfProjects.length != numOfProjects) {
                System.err.println("The given number of projects are not equal to "
+ numOfProjects + ".");
            }
        } while (namesOfProjects.length != numOfProjects);

        // scanner to get outcomes of projects as an input and also checks if the
user gives the correct number of inputs
        do {
            String inputOutcomes = sc.next();
            outcomesOfProjects = Arrays.stream(inputOutcomes.split("
")).mapToInt(Integer::parseInt).toArray();
            if (outcomesOfProjects.length != numOfProjects) {
                System.err.println("The given number of outcomes are not equal to
the given number of projects.");
            }
        } while (outcomesOfProjects.length != numOfProjects);
```

Declaration and initialization of `outcomesMap`

Then a hashmap variable `outcomesMap` having String as keys and Integer as values in order to store names of projects and outcomes of projects in a single data structure.

```
// hash-map to store the given outcomes of venture projects
HashMap<String, Integer> outcomesMap = new HashMap<>();
// storing the venture projects and their outcomes in outcomesMap<>
for (int i = 0; i < namesOfProjects.length; i++) {
    outcomesMap.put(namesOfProjects[i], outcomesOfProjects[i]);
}
```

Scanning the prerequisites set E from the keyboard

Program reads input from a single line and then stores the array of prerequisites in two different arrays `ArrayList<ArrayList<String>> preReq` and `preReqCopy`.

```
// scanner to get input for the prerequisites
String[] str;
String inputPreReq = sc.next();
String[] inputPreReqArray = inputPreReq.split(" ");
ArrayList<ArrayList<String>> preReq = new ArrayList<>();    // array to
store the prerequisites
ArrayList<ArrayList<String>> preReqCopy = new ArrayList<>();    // holds
copy of ArrayList<ArrayList<String>> preReq

// storing the input prerequisites in an 2D array
for (String s : inputPreReqArray) {
      str = s.substring(1, 4).split(",");
      preReq.add(new ArrayList<>(Arrays.asList(str[0], str[1])));
            preReqCopy.add(new ArrayList<>(Arrays.asList(str[0], str[1])));
}
```

Declaration of `projectsNeedPreReq` and `seqProjects`

Then the program stores projects which require some prerequisite in `projectsNeedPreReq` variable and the ones which do not need any prerequisite in `seqProjects` variable respectively.

```
// a set which contains the projects which require some prerequisite
Set<String> projectsNeedPreReq=new HashSet<>();
// storing the projects need some prerequisite
for(ArrayList<String> project: preReq){
    projectsNeedPreReq.add(project.get(0));
}

// a set which contains the projects which do not need any prerequisite
Set<String> seqProjects=new HashSet<>();
// storing the projects which do not require any prerequisite
for(String project:namesOfProjects){
    if(!projectsNeedPreReq.contains(project)){
        seqProjects.add(project);
    }
}
```

Declaration of `profitMap` variable and storing data

In lines 229-283 comes the magic! We declare and store the ultimate profit of projects because the projects have prerequisites hence the final outcomes of the venture projects changes. This part will be explained in the upcoming heading.

Creation of network flow graph

We create a network flow graph by creating an instance of the `Graph()` class. First of all, we add vertices `source` and `sink` as `"S"` and `"t"` respectively, and then names of projects one by one in a loop. Then we create edges from source `"S"` to all projects which don't need any prerequisites with capacity `INF = Integer.MAX_VALUE`. Also, the edges from prerequisites leading to the successor are created with capacity `INF` using the `preReqCopy` array. And at the end, we complete the graph by creating edges from all vertices falling to `sink` with the capacity of their optimal profit copying from the `profitMap` hash table.

```java
// we started to create our network flow graph
Graph networkGraph = new Graph();

networkGraph.addVertex("S");
networkGraph.addVertex("t");

for (String proj : namesOfProjects) {
      networkGraph.addVertex(proj);
}

for (String proj : seqProjects) {
        networkGraph.addEdge("S", proj, INF);
}

for (ArrayList<String> projArray: preReqCopy) {
        networkGraph.addEdge(projArray.get(1), projArray.get(0), INF);
}

for (String proj: namesOfProjects) {
        networkGraph.addEdge(proj, "t", profitMap.get(proj));
}
```

Declaration of Ford Fulkerson Solver and “Decide” keyword

At the end we instantiate a Ford Fulkerson Solver using the FordFulkerson class. To solve the problem `solve()` method is called, but wait for the user to enter input `Decide`. After displaying the result on the console, the program ends by closing the `scanner` instance `sc`.

Edge Class

This class gives an instance of an edge in a graph. This class contains four attributes as follows:

- `String from`
- `String to`
- `int capacity`
- and `int flow`

It has a constructor `public Edge(String from, String to, int capacity) {}` and also two methods `int remainingCapacity()` and `void augmentEdge()` which return remaining capacity of edge and augment the flow while traversal respectively.

Vertex Class

This class creates an object vertex of a graph where multiple edges either bring in the flow or take the flow outwards. This class has two attributes:

- `String id`
- and `ArrayList<Edge> edges`

It has just one constructor `public Vertex(String id) {}` and no other method.

Graph Class

This class creates a flow network graph which we used in our `static void main method()` in line 136 to create our flow graph. This class has two attributes, which are as follows:

- `ArrayList<Vertex> vertices`
- and `ArrayList<Edge> edges`

There are two methods implemented `void addVertex()` and `void addEdge()` which add vertices and edges to those vertices respectively.

FoldFolkerson Class

This class acts as a solver for our graph which uses DFS traversal of the graph with time complexity O(f*E) and gives maximum profit and the venture projects as a result.

Network Flow Programming Solution

In this section the determined network flow graph is described. Let's consider the given example to elaborate the flow graph.

In the given problem we are provided with three sets:

- $P = \{A, B, C, D, E, F, G\}$
- $O = \{10, -8, 2, 4, -5, 3, 2\}$
- $E = \{(A, B), (C, A), (C, B), (C, E), (D, B), (D, F), (G, C)\}$

We created a hash table `profitMap` which stores the net profit of the projects, for example, in the given example B is a prerequisite of A, therefore, A cannot be executed till the time we complete project B. Ultimately, the profit we earn after completing project A will be 2 because it's collective outcome of $B + A = -8 + 10 = 2$.

Step 1: Evaluating net profit of the projects which don't need any prerequisites

In the given problem B, E and F are the only projects which can be started without waiting for the other projects. So, we add the outcomes of these projects to the `profitMap`.

So, `profitMap` becomes as follows:

	A	B	C	D	E	F	G
profitMap = {	__	-8	__	__	-5	4	__}

Then we check if there is any project having the net profit greater than 0, if **Yes!** then we exclude the dependency of that particular project on other projects in E set. Because they are not dependent anymore because we are going to complete such a project anyhow because the review is positive. In our case, F has positive profit value so we will remove the tuple (D, F) from E set and set changes:

$$E = \{(A, B), (C, A), (C, B), (C, E), (D, B), (D, F), (G, C)\}$$

$$\Downarrow$$

$$E = \{(A, B), (C, A), (C, B), (C, E), (D, B), (G, C)\}$$

Implementation of the logic is shown in following snippet:

```
for (String seq : seqProjects) {
      profitMap.put(seq, outcomesMap.get(seq));
      if (profitMap.get(seq) > 0) {
            for (int i = 0; i < preReq.size(); i++) {
                  if (Objects.equals(preReq.get(i).get(1), seq)) {
                              preReq.remove(i--);
                  }
            }
      }
}
```

Step 2: Evaluating net profit of the projects which have prerequisites

We can find net profit of such projects using following formula:

Netprofit = Outcome of project + Sum of the net profits of their prerequisites

and similarly if net profit greater than 0, then we not only exclude the dependency of the project but also dependency of their prerequisites too. It is because, since profit is positive then the project is completed anyways after its prerequisites are completed. So, if we consider evaluation of net profit of A project, then:

$$\begin{array}{lccccccc} & A & B & C & D & E & F & G \\ \text{profitMap} = \{ & 2 & \underline{-8} & \_ & \_ & \underline{-5} & \underline{4} & \_\} \end{array}$$

$$E = \{(A, B), (C, A), (C, B), (C, E), (D, B), (G, C)\}$$

$$\Downarrow$$

$$E = \{(A, B), (C, E), (G, C)\}$$

Implementation of the logic is shown in following code snippet:

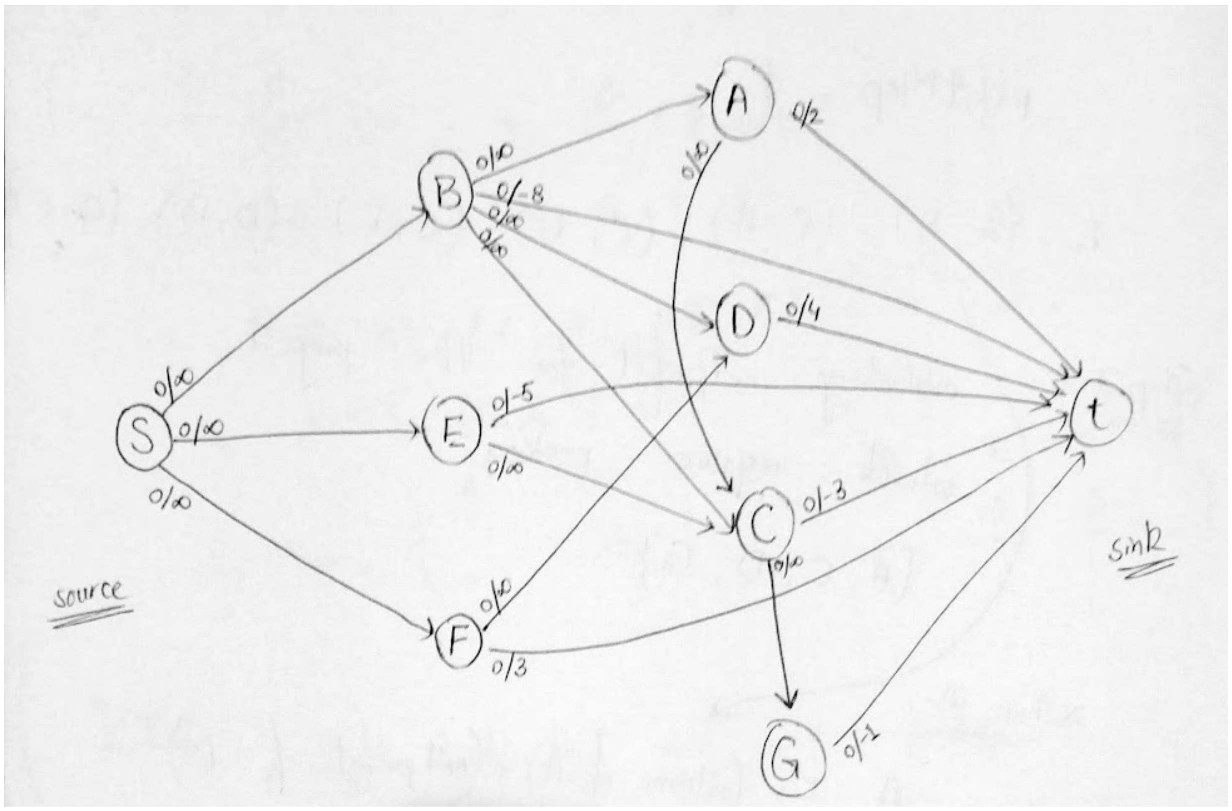
```
for (String proj : projectsNeedPreReq) {
      for (ArrayList<String> p : preReq) {
            if (Objects.equals(p.get(0), proj)) {
                  prerequisites.add(p.get(1));
                }
      }
      for (String pre : prerequisites) {
                projectProfit += profitMap.get(pre);
        }
        projectProfit += outcomesMap.get(proj);
        profitMap.put(proj, projectProfit);

        if (profitMap.get(proj) > 0) {
            for (int i = 0; i < preReq.size(); i++) {
                  if (Objects.equals(preReq.get(i).get(1), proj)) {
                              preReq.remove(i--);
                        }
                }
                for (String pre : prerequisites) {
                        for (int i = 0; i < preReq.size(); i++) {
                              if (Objects.equals(preReq.get(i).get(1),
pre)) {
                                    preReq.remove(i--);
                              }
                        }
                }
      }
      projectProfit = 0;
      prerequisites.clear();
}
```

Step 3: Creating Flow Network Graph

As I have mentioned in the previous section, the vertices are created with the name of projects. Edges of the graph are created as follows:

- Edges from source to the projects which don't need any prerequisite having capacity infinity.
- Edges from prerequisite to the venture project having capacity infinity.
- Edges from each project to sink with capacity of net profit of each venture project respectively.

Here is the demonstration of directed acyclic graph:



Step 4: Calculating maxflow and venture projects using Ford Fulkerson Algorithm

The maximum profit earned from the venture projects is maxflow of the given graph. The Maxflow problem is solved using the Ford Fulkerson Algorithm. This solver method is defined in the FordFulkerson class. Implementation of this algorithm is given in `FordFulkerson` class from line 93-149.

Ford Fulkerson Algorithm traverses the network graph through a DFS search in a manner to find a path which leads from source to the sink and where all of the edges have remaining capacity more than zero. After finding the path, we evaluate the bottleneck value of the path which is equal to the minimum remaining capacity among the edges. Then we augment the path with a bottleneck value, in the Fold Fulkerson algorithm we augment the residual edge too, but in this case we can surpass it. The program terminates when there is no remaining augmenting path. The sum of the bottleneck values gives maximum flow among the graphs.

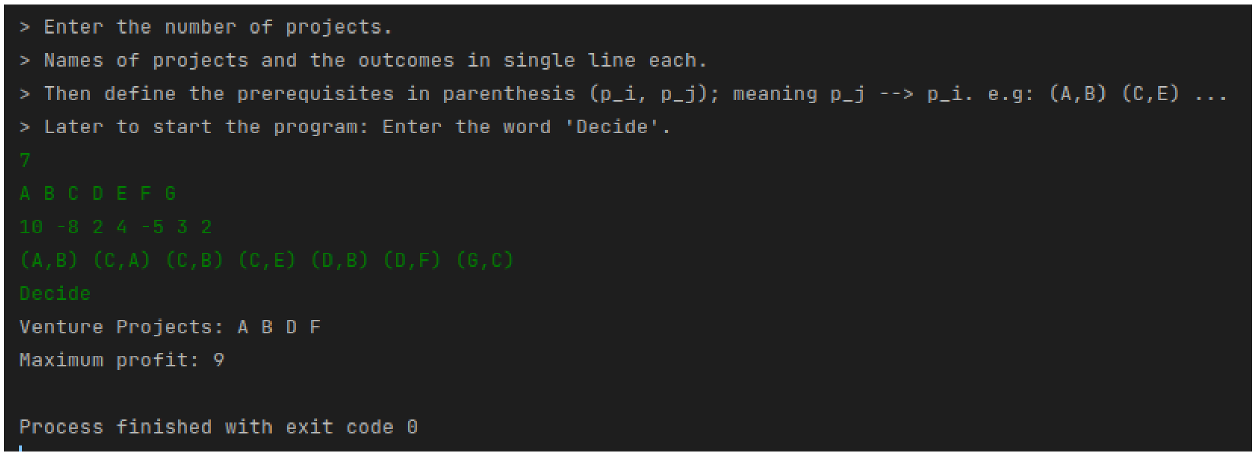
Time Complexity

This section gives an in-depth time analysis of the whole implementation.

Initialization of Arraylist, arrays, HashMap, variables, getting inputs from the user takes O(1) time complexity. Checking the input of the user for names and outcomes of the projects takes O(N) times by using String.split() for each (line: 174-189). Creating Sets for the projects that require some prerequisites and the projects that don't require any prerequisites takes O(N) time complexity for each (line: 213-226). The nested loops (line: 236-245) takes O(N^2) time complexity. Between the lines 254 and 283 there are many for loops where some of them are nested. The dominant time complexity will be (N^3). Adding a vertex to the ArrayList takes O(1) time complexity where for n projects takes O(N) in a for loop. Between the lines 295 and 305 there are 3 separate for loops where I am adding edges according to the function I wrote addEdge() each for loop takes O(N) times. Since they are not nested we are not going to multiply. Creating a Ford Fulkerson solver at line 308 takes O(N) time complexity since I used a foreach loop in its constructor. Then applying Ford Fulkerson method utilizing Depth First Search takes O(Max flow * Edges) because the algorithm tries to find an augmenting path in the graph to add a constant amount of flow to each edge in that specific path. This process is repeated until there are no more augmenting paths. At the end, we get the max flow. Since I used DFS to find augmenting paths it takes O(E). It iterates to find new augmenting paths which can be done at most F iterations (max flow) where each iteration takes O(E+V). So the worst running time becomes O((E+V)f) which is O(E*F).

Results

The suggested solution to the described problem statement is extremely effective and produces the right outcome in all usage scenarios. The result of a program run is shown in the following picture as an illustration:



```
> Enter the number of projects.
> Names of projects and the outcomes in single line each.
> Then define the prerequisites in parenthesis (p_i, p_j); meaning p_j --> p_i. e.g: (A,B) (C,E) ...
> Later to start the program: Enter the word 'Decide'.
7
A B C D E F G
10 -8 2 4 -5 3 2
(A,B) (C,A) (C,B) (C,E) (D,B) (D,F) (G,C)
Decide
Venture Projects: A B D F
Maximum profit: 9

Process finished with exit code 0
```

References

- https://en.wikipedia.org/wiki/Flow_network
- https://en.wikipedia.org/wiki/Network_flow_problem