Homework Week 11: Implementation Adaptions #2

(Last Submission Date: Tuesday Nov. 11th at 11:59pm)

(Extra Credit for Early Submission…)

Goal: Get more experience adapting an algorithm implementation written by another person to your specific needs. Also, explore an application area for network flow.

Activity: Given an implementation of network flow, adapt it to solve the bigraph matching problem.

Teamwork: None, this is a solo assignment.

Details

Existing Algorithm Implementations (Continued)

See Homework 3 for background information why understanding algorithm implementations is important. However, the motivation for *this* homework is slightly different. In Homework 3 you were given two different implementations of a topological sort, and you were trying to do a topological sort (your data was just in the wrong format). For this homework you have only one implementation, and you also have a problem and algorithm that don't quite match!

Background Information

Let's go back to day one of this semester when we looked at some interview questions and you were asked to match animals to new owners. Here's the summary of that problem:

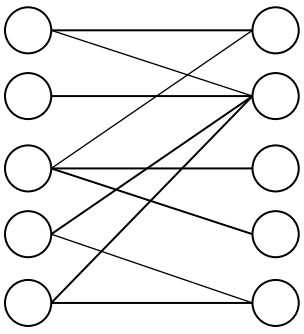
- We have some number of animals at the shelter.
- There are people who want to adopt animals, but they all are looking for different kinds of animal. Some are open to more than one type of animal; some are very picky.
- You want to figure out which animal should go with which person so that the maximum number of animals are adopted.

This is actually a very common problem and not just at animal shelters! Matching medical students to medical schools, matching teams to coaches, matching volunteers to tasks, all these things can be seen as the same problem but with different "things" being matched. Here's another example:

- We have some number of job openings at our company.
- We also have some number of applicants, each of which is qualified for at least one job. But not everyone is qualified for every job, some people are qualified for more than one job, and some people do/don't want certain jobs.
- What is the max number of positions we could fill?

So what *is* this problem? It's called **maximal bipartite graph matching**!

A bipartite graph (or bigraph) is a graph with two "sets" of nodes, where nothing within one set connects to the same set. For example, in the previous problem, a person can be "matched" to a job, but a person can't be matched to a person, or a job matched to a job. This forms graphs that look like the picture on the right:



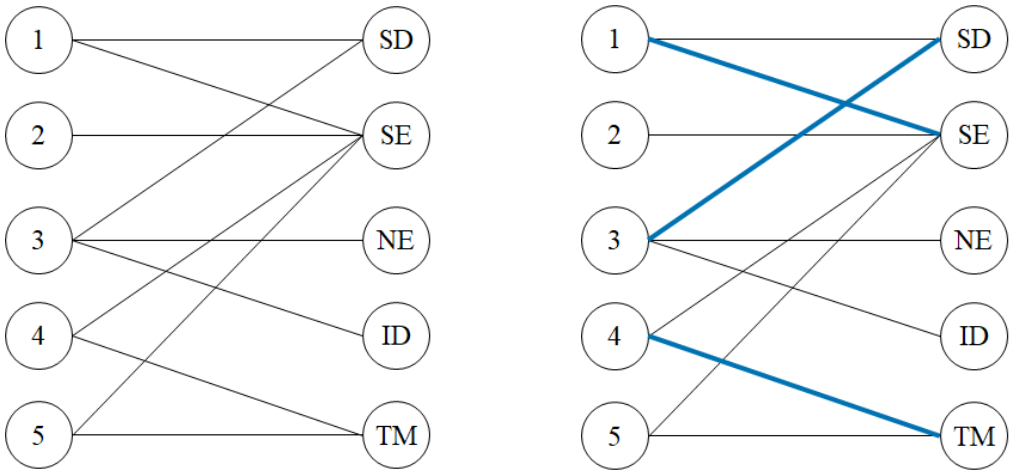
Maximal matching in a bipartite graph is connecting as many nodes as possible from "set 1" (nodes on the left) to a node in "set 2" (nodes on the right) without reusing any nodes in either set.

Let's make a concrete problem to examine from here on…

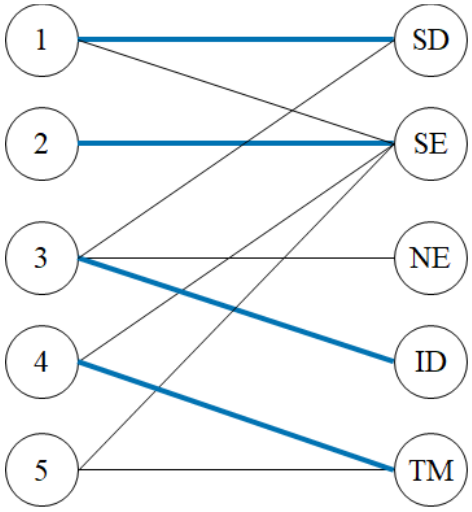
- We have 5 job openings at our company:
  - *Software Developer*
  - *Software Engineer*
  - *Network Engineer*
  - *Interface Developer*
  - *Technical Manager*
- We also have 5 applicants (numbered 1-5 to avoid any bias):

Note: the number of job openings to applicants doesn't have to match evenly, we could have any number of each thing (e.g. 4 people and 6 openings), this is just an example.

After the company interviews each applicant, they can determine how every candidate *could* be match and try out different combinations. Below is the bigraph showing who is qualified for each position (left) and one sample matching (right):

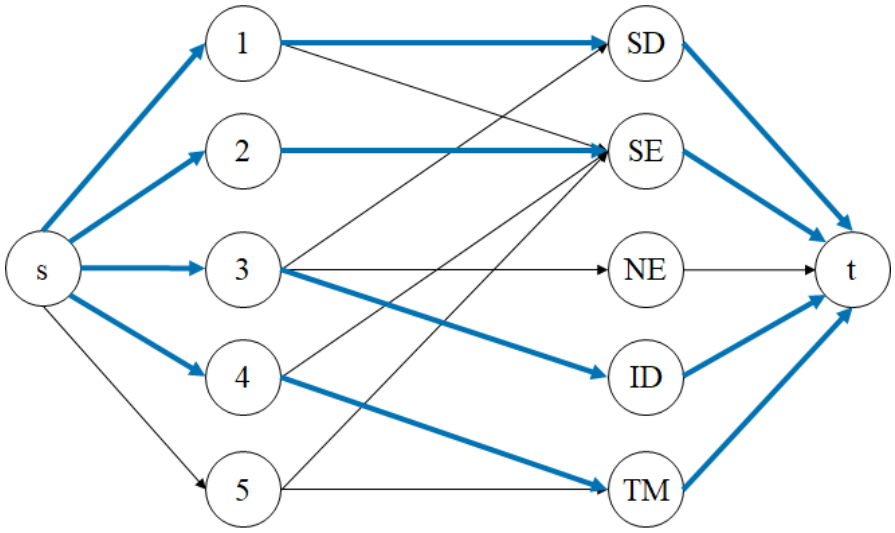


The matching shown on the right is Person 1 ➔ Software Engineer, Person 3 ➔ Software Developer, Person 4 ➔ Technical Manager. Unfortunately, while there are no more matches possible with these choices, it isn't the *maximum* number of matches. That would look something like this:



We could theoretically try every possible combination and see what happens, but we could also be "clever" and convert this into a well-understood problem (in this case, network flow!).

So how do you convert a bigraph matching problem into a network flow problem? It's actually quite simple, you make the graph directed (from set 1 to set 2) and then create a "fake" source node and a "fake" sink node like so:



Now, matches just "flow" through the graph! In other words, after you run your max-flow algorithm, the flow graph will tell you which nodes from set 1 are "matched" to set 2 (because there is flow between them). The max flow will also be the number of matches.

So, what will you be doing? You've been given you a bigraph matching problem, and a max flow algorithm implementation, and you just need to put those two things together to get your answer to the maximal matching problem.

Setup and File Structure

The zip file you were given contains multiple files and a folder called "codeGoesHere". This is your "code directory" for the remainder of this document, the folder above this is your "project directory". Do not change this directory structure unless you're really sure you know what you're doing.

In your *code* directory there are four files: **`Convert.java`**, **`Node.java`**, **`Edge.java`**, and **`Flow.java`**. **`Convert`** is where you will write your code. **`Node`** and **`Edge`** are helper classes for **`Flow`** which, in turn, contains an implementation of Ford-Fulkerson's Max Flow algorithm. There are some comments and notes in the code which you should read, and some methods that return things you might find useful (like the max flow and the flow graph itself) in **`Flow`**.

Additionally, **`Flow`** has a main method setup to display a demo of max flow step-by-step (you can uncomment more demos if you want), and **`Convert`** has a main method which outputs the answer you arrived at. There are no provided JUnit tests since **`Convert`** demonstrates how your code will be used.

Input to and Output from Your Program

You will receive bigraph information from a text file as the string representation of the potential matches in the following format:

```
[Single node from set 1]>[Comma separated list of nodes in set 2]
[Single node from set 1]>[Comma separated list of nodes in set 2]
[Single node from set 1]>[Comma separated list of nodes in set 2]
[Single node from set 1]>[Comma separated list of nodes in set 2]
...
```

This is similar to an adjacency list, but only nodes from set 1 on the left and only nodes from set 2 in the lists. For the example we've been working with, there is a sample file (**`example.txt`**) in your project directory which looks like this:

```
Person 1>Software Developer,Software Engineer
Person 2>Software Engineer
Person 3>Software Developer,Network Engineer,Interface Developer
Person 4>Software Engineer,Technical Manager
Person 5>Technical Manager
```

You may assume that nodes do not "share" string names, i.e. node "banana" cannot be in set one and set two, and "banana" in two places on the right side refers to the same "banana" node.

To avoid issues with writing to files, we're going to have your code wrapped by a single method:

```
Answer youCodeGoesInThisMethod(File inputFile)
```

This method accepts a Java **`File`** and returns an **`Answer`**. An **`Answer`** has two parts, the max flow and a set of matches from nodes in set 1 to nodes in set 2:

```
class Answer {
      int maxFlow;
      TreeMap<String,String> matches;     }
```

For example, running **`youCodeGoesInThisMethod()`** on the file **`example.txt`** would return an **`Answer`** with the **`maxFlow`** set to **`4`** and a set of four valid matches, such as:

```
{ Person 1 : Software Developer
  Person 2 : Software Engineer
  Person 3 : Network Engineer
  Person 4 : Technical Manager }
```

The **`Answer`** class and the **`youCodeGoesInThisMethod()`** can both be found in the **`Convert.java`**. There is also a main method which you can run like so:

```
> java Convert ../example.txt
************************************************
Max Flow: 4
Matches:
        Person 1-->Software Developer
        Person 2-->Software Engineer
        Person 3-->Network Engineer
        Person 4-->Technical Manager
************************************************
>
```

At the end of the day, your program must compile with the following command:

```
javac *.java
```

And **`Convert.youCodeGoesInThisMethod(...)`** must return a valid **`Answer`** for the *given* input file.

Note that you ***can*** modify, remove, change, or otherwise alter any of the provided code except the **`Answer`** class and the method signature of **`youCodeGoesInThisMethod()`**. You can also add classes, methods, and other components to your heart's content. However, you cannot make your program part of a package, or import libraries that require downloading from the internet, or anything that requires compiling/running with a jarred library (such as JUNG).

Self-Testing Your Code

As mentioned, there are no provided JUnit tests for this homework. The instance of the **`Answer`** class returned by **`youCodeGoesInThisMethod()`** is what will be graded. However, please note that you still must be using a max flow algorithm to solve your problem, brute force solutions will receive a 0.

Also, make sure to test with more than just "example.txt". There's another example provided, but you should also be making your own and checking you're getting the expected results.

What should I be careful of / test with static methods? Remember that calling a method twice with two different inputs should work! This is still the case for static methods and you should test this. For example, a main method which had:

```
Answer a1 = Convert.youCodeGoesInThisMethod(new File("example.txt"));
Answer a2 = Convert.youCodeGoesInThisMethod(new File("differentExample.txt"));
```

Should be valid (and work)!

Homework Challenges Laid Out

Here is a summary of the challenges of this homework and how they relate to topics covered this semester (either by reinforcing earlier topics or introducing new topics):

- **Working with graphs**
    - Your input is in the form of an adjacency list, but the algorithm requires an adjacency matrix – you'll either need to convert to a matrix or convert the algorithm to work with different input (first option is *way* easier). **[Reinforcement Topic: Basic Graph Storage]**
    - You have non-integer nodes, but the algorithm requires integer nodes – you'll either need to use an index map or to convert the max flow algorithm to work with different input (first option is way easier). **[Reinforcement Topic: Advanced Graph Storage]**
- **Working with max flow**
    - You need to convert a bigraph matching problem into a max flow problem using graph manipulation. **[New Topic: Max Flow Applications]**
    - You need to be able to read the output from the max flow algorithm and "interpret" it as matchings in the bigraph. **[New Topic: Max Flow Applications]**

Grading Rubric

No credit will be given for non-submitted assignments, assignments submitted after the last submission date, non-compiling assignments, non-independent work, or brute force solutions. Otherwise your score will be proportional to the number of grading tests you pass. We will definitely check the sample inputs we gave you, but also we'll make up some new inputs. What you submit will be what is graded, so test before submission!

Extra credit for early submissions: 1% extra credit rewarded for every 24 hours your submission made before the last submission date (up to 5% extra credit). Your latest submission before the due time will be used for grading and extra credit checking. You CANNOT choose which one counts.

Submission Instructions

- Go to the Homework 11 submission link on GradeScope.
- Submit ALL Java files from your CODE directory.
- You can resubmit as many times as you'd like, only the last submission will be graded.
- See common autograder feedback (and whether or not you should care about it) below
- 5% will be subtracted if you don't submit correctly to GradeScope as listed in the Submission Instructions (because it will slow down grading for the entire class).

Autograder Feedback Upon Submission

Go to GradeScope and submit ***just the Java files from your CODE directory***. The GradeScope autograder will give you some immediate feedback to ensure you have submitted everything correctly:

- Cannot Find Java Files
  If your results say anything like:

  ```
  **********************
  Results
  **********************
  Cannot find java files Issue: cannot locate java files in submission
  folder
  ```

  Then something is wrong with your submission, and you should re-check what you've uploaded. Common mistakes: uploading a folder rather than just the Java files, uploading a zip, rar, or other compressed file, uploading .class files.

- Fail "Compile Alone"
  You are probably missing a file, make sure you submit ***all*** your Java files from your ***CODE*** directory.

- Fail "Compile With Tests (Preliminary Check)"
  Somehow you are not compiling with the JUnit tests. Check to make sure you didn't change the **`Answer`** class or the method signature of **`youCodeGoesInThisMethod()`** – Note that adding "throws" is a change in the method signature, so don't add that.

- Fail "Checkstyle Code" or failed "Checkstyle Comments"
  This is ok for CS483 (we are not grading you on style or documentation!).