EE360C Algorithms
University of Texas at Austin — Programming Assignment #1
Dr. Santacruz — Due: Feb 14, 2022 11:59 PM

Programming Assignment #1

Programming assignments are to be done individually. Do not make your code publicly available (such as a Github repo) as this enables others to cheat and you will be held responsible. You may discuss the problem and general concepts with other students, but there should be no sharing of code. You may not submit code other than that which you write yourself or is provided with the assignment. This restriction specifically prohibits downloading code from the Internet. If any code you submit is in violation of this policy, you will receive no credit for the entire assignment.

This programming assignment is due **Monday, February 14th at 11:59 PM.** If you are unable to complete the assignment by this time, you may submit the assignment late until Wednesday, February 16th at 11:59 PM for a 20 point penalty.

The goals of this lab are:

- Familiarize you with programming in Java
- Show an application of the stable matching problem
- Understand the difference between the two optimal stable matchings

Problem Description

In this project, you will implement a variation of the stable matching problem adapted from the textbook Chapter 1, Exercise 4, and write a small report. We have provided Java code skeletons that you will fill in with your own solution. Please read through this document and the documentation in the starter code thoroughly before beginning.

A new international company, Salud del Mundo, sends doctors to cities in need of more medical expertise in their local governments to keep their populations healthy. The doctors are incentivised because they can rank their destination for each quarter and will get to travel the world while improving public health. However, destination cities also rank the doctors based on a variety of factors, including spoken language and their main area of study. Your job is to devise and implement an algorithm to automate this process based off of the Gale Shapley algorithm presented in class.

There are n doctors, each interested in working in one of m cities. Each city has a set number of doctors they seek to hire, which can vary between cities. Every doctor submits their preference list of cities, and every city's government creates a preference list of doctors based on their submitted profiles. We will assume that there are at least as many doctors as the total positions available across all m cities. This means that all positions will be filled, but some doctors may be left unmatched to a city. (They'll get a nice vacation that quarter.) The interest lies in finding a way of assigning each doctor to at most one city in such a way that all available positions are filled.

We say that an assignment of doctors to cities is *stable* if neither of the following situations arises:

- First type of instability: There are doctors i and i', and a city c, such that
  - i is assigned to c, and
  - i' is assigned to no city, and
  - c prefers i' to i
- Second type of instability: There are doctors i and i', and city c and c', so that
  - i is assigned to c, and
  - i' is assigned to c', and
  - c prefers i' to i, and
  - i' prefers c to c'.

So, we basically have the Stable Matching Problem as presented in class, except that (i) a city may want one or more doctors, and (ii) there is potentially a surplus of doctors. There are several parts to this problem.

Part 1: Write a report [20 points]
Write a short report that includes the following information:

(a) Give an algorithm in pseudocode (either an outline or paragraph works) to find a stable assignment that is **city** optimal.

(b) Give the runtime complexity of your algorithm in (a) in Big O notation and explain why. **Note: Full credit will be given to solutions that have a complexity of $O(mn)$.**

(c) Give an algorithm in pseudocode (either an outline or paragraph works) to find a stable assignment that is **doctor** optimal.

(d) Give the runtime complexity of your algorithm in (c) in Big O notation and explain why. **Note: Try to make your algorithm as efficient as you can, but you will get full credit even if it does not match the runtime in (b) as long as you clearly explain your runtime and the difficulty of optimizing it further.**

For the programming assignment, you do not need to submit a proof that your algorithm returns a stable matching, or of doctor/city optimality.

Part 2: Implement a Checker to check stability of any given matching [20 points]
Given a Matching object `problem`, you should implement a boolean function to determine if the pairing of doctors to cities (stored in the variable returned by `problem.getDoctorMatching()`) is stable or not. Your code will go inside a function called `isStableMatching(Matching problem)` inside `Program1.java`. A file named `Matching.java` contains the data structure for a matching. Note that you do not need to optimize the runtime of this function, a brute force approach is sufficient. See the instructions section for more information on how to test this method.

Part 3: Implement Gale Shapley Algorithm [60 points]
Implement both algorithms from parts (a) (city optimal) and (c) (doctor optimal) of your report.

Again, you are provided several files to work with. Implement the function that yields a doctor optimal solution `stableMatchingGaleShapley_doctoroptimal()` and city optimal solution `stableMatchingGaleShapley_cityoptimal()` inside of `Program1.java`.
Of the files we have provided, please only modify `Problem1.java`, so that your solution remains compatible with ours. However, feel free to add any additional Java files (of your own authorship) as you see fit.

Instructions

- Download and import the code into your favorite development environment. We will be grading in Java 1.8. Therefore, we recommend you use Java 1.8 and NOT other versions of Java, as we can not guarantee that other versions of Java will be compatible with our grading scripts. **It is YOUR responsibility to ensure that your solution compiles with Java 1.8.** If you have doubts, email a TA or post your question on Piazza.

- If you do not know how to download Java or are having trouble choosing and running an IDE, email a TA, post your question on Piazza, or visit the TAs during Office Hours.

- **Do not add any package statements to your code.** Some IDEs will make a new package for you automatically. If your IDE does this, make sure that you remove the package statements from your source files before turning in the assignment.

- There are several `.java` files, but you only need to make modifications to `Program1.java`. **Do not modify the other files.** However, you may add additional source files in your solution if you so desire. **Do not add extra imports to** `Program1.java`; the included imports should be all you need for your solution. There is a lot of starter code; carefully study the code provided for you, and ensure that you understand it before starting to code your solution. The set of provided files should compile and run successfully before you modify them.

- The main data structure for a matching is defined and documented in `Matching.java`. A Matching object includes:

  - **m**: Number of cities
  - **n**: Number of doctors
  - **city_preference**: An ArrayList of ArrayLists containing each of the city's preferences of doctors, in order from most preferred to least preferred. The cities are in order from 0 to $m - 1$. Each city has an ArrayList that ranks its preferences of doctors who are identified by numbers 0 through $n - 1$.
  - **doctor_preference**: An ArrayList of ArrayLists containing each of the doctor's preferences for cities, in order from most preferred to least preferred. The doctors are in order from 0 to $n - 1$. Each doctor has an ArrayList that ranks its preferences of cities who are identified by numbers 0 to $m - 1$.
  - **city_positions**: An ArrayList that specifies how many positions each city has. The index of the value corresponds to which city it represents.

  - **doctor_matching**: An ArrayList to hold the final matching. This ArrayList (should) hold the number of the city each doctor is assigned to. This field will be empty in the `Matching` which is passed to your functions. The results of your algorithm should be stored in this field either by calling `setDoctorMatching(<your_solution>)` or constructing a `new Matching(data, <your_solution>)`, where `data` is the Matching we pass into the function. The index of this ArrayList corresponds to each doctor. The value at that index indicates to which city they are matched. A value of -1 at that index indicates that the doctor is not matched up. For example, if doctor 0 is matched to city 55, doctor 1 is unmatched, and doctor 2 is matched to city 3, the ArrayList should contain {55, -1, 3}. If using the flag [-bf], an input with an existing matching can be given to check correctness of the `isStableMatching()` function.

- You must implement the methods
  - `isStableMatching()`
  - `stableMatchingGaleShapley_doctoroptimal()`
  - `stableMatchingGaleShapley_cityoptimal()`

  in the file `Program1.java`. You may add methods to this file if you feel it necessary or useful. You may add additional source files if you so desire.

- Test cases take the format of text files, which either have the file extension of `.in` or `.extended.in`. Here's how to interpret each test case, line by line:
  - Line 1: `m n`
  - Line 2: `m` space separated integers, denoting the number of positions available in each city. The first integer represents the number of open positions in city 0, the next integer represents the number for city 1, and so on.
  - The next `m` lines are the preference lists of the cities, where each space-separated integer represents a doctor. The list goes from left to right, from most to least desirable. The first of these `m` lines is the preference list for city 0, the next line is for city 1, and so on.
  - The next `n` lines are the preference lists of the doctors, where each space-separated integer represents a city. The list goes from left to right, from most to least desirable. The first of these `n` lines is the preference list for doctor 0, the next line is for doctor 1, and so on.
  - Last line (optional): `n` space separated integers representing a doctor-city matching. If the first integer is x, then the first doctor is assigned to city x, the second doctor to the second integer, and so on. This is a way of hard coding in a matching to test your implementation of `isStableMatching()` in Part 2 before you complete Part 3. To see examples, see the last lines of the test cases with the file extension `.extended.in`.

- `Driver.java` is the main driver program. Use command line arguments to choose between your checker and your city optimal or doctor optimal algorithms and to specify an input file. Use -gc for city optimal, -gi for doctor optimal, and -bf for importing an existing matching (to check correctness of `isStableMatching()`). (i.e. `java -classpath . Driver [-gc]`

`[-gi] [-bf] <filename>` on a linux machine). As a test, the 3-10-3.in input file should output the following for both a doctor and city optimal solution:

  – Doctor 0 city -1
  – Doctor 1 city 1
  – Doctor 2 city -1
  – Doctor 3 city -1
  – Doctor 4 city -1
  – Doctor 5 city -1
  – Doctor 6 city -1
  – Doctor 7 city 2
  – Doctor 8 city 0
  – Doctor 9 city -1

- When you run `Driver.java`, it will tell you if the results of your algorithm(s) pass the `isStableMatching()` function that *you coded* for this particular set of data. When we grade your program, however, we will use *our* implementation of `isStableMatching()` to verify the correctness of your solutions.
- Make sure your program compiles on the LRC machines before you submit it.
- We will be checking programming style. A penalty of up to 10 points will be given for poor programming practices (e.g. do not name your variables foo1, foo2, int1, and int2).

NOTE: To avoid receiving a 0 for the coded portion of this assignment, you MUST ensure that your code correctly compiles with the original, unmodified starter files on Java 1.8. Do not modify the signatures of or remove existing methods of `Program1.java`. Do not add package statements. Do not add extra imports. You must zip your code using the exact format described below with no spaces. We recommend testing compilation of your code using the ECE LRC Linux machines (using "javac *.java" and "java Driver inputfile.in") after redownloading the starter files from Canvas. We will not be allowing regrades on this assignment, so please be careful and double check that your final submission is correct.

What To Submit (please read carefully)

You should submit to Canvas a single ZIP file titled `pa1_eid_lastname_firstname.zip` that contains `Program1.java` and any extra `.java` files you added. Do not submit `AbstractProgram1.java`, `Driver.java`, or `Matching.java`. Do not put your `.java` files in a folder before you zip them (i.e. the files should be in the root of the ZIP archive). Your ZIP file name MUST have the exact format: `pa1_eid_lastname_firstname.zip`. Be certain that there are no spaces in your zip file name. Failure to follow these instructions will result in a penalty of up to 10 points.

Your PDF report should be legibly scanned and submitted to Gradescope. Both your zipped code and PDF report must be submitted BY 11:59 PM on Monday, February 14th, 2022. If you are unable to complete the assignment by this time, you may submit the assignment late until Wednesday, February 16th at 11:59 PM for a 20 point penalty.