EE360C: Algorithms
University of Texas at Austin
Dr. Christine Julien

Programming Assignment #1
Due: February 14, 2017 11:59pm

Programming Assignment #1

Programming assignments are to be done individually. 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.

The goals of this lab are:

- Familiarize you with programming in Java
- Compare the run time of an efficient algorithm against an inefficient algorithm
- Show an application of the stable matching problem

Problem Description

In this project, you will implement a variation of the stable marriage problem 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.

Consider a workforce economy made up of n available jobs and n people looking for work. Each worker has a list of preferences for jobs, and each job ranks all of the available workers. There are no ties in these lists. Some of the jobs are full-time jobs and remaining jobs are part-time jobs. We assume that every worker prefers any full-time job over any part-time job. Further, some of the workers are "hard workers" and the remaining are "lazy." We assume that every employer prefers any hard worker over any lazy worker. That is, an employer's ranked list of workers first ranks all of the hard workers, followed by a rank ordering of the lazy workers. Similarly each worker's list of preferred jobs begins with a rank ordering of the full-time jobs, followed by a rank ordering of the part-time jobs.

We assume a definition of stability identical to that we proved for stable marriage: a matching of workers to jobs is stable if it is perfect (i.e., every worker gets a job and every job gets filled) and there are no instabilities (i.e., there does not exist two pairs (j, w), (j', w') such that the employer offering job j prefers worker w' to w and w' also prefers job j to j'.

There are several parts to the problem.

Part 1: Write a report [30 points]

Write a short report that includes the following information:

(a) If half of the jobs are full-time jobs, and half of the workers are hard-workers, prove that, in every stable matching of workers to jobs, every hard worker gets a full time job.

(b) Consider a Brute Force Implementation of the algorithm where you find all combinations of possible matchings and verify if they are a stable marriage one by one. Give the runtime

complexity of this brute force algorithm in Big O notation and explain why it correctly captures the runtime of the brute force algorithm.

(c) Give an efficient algorithm in pseudocode (either an outline or paragraph works) to find a stable assignment. The efficient algorithm should be significantly more efficient than a brute force algorithm.

(d) Give a proof of your efficient algorithm's correctness. Remember that you must prove both that your algorithm terminates and gives a correct result.

(e) Give the runtime complexity of your efficient algorithm in Big O notation and explain why it correctly captures the runtime of your algorithm. *You will be graded on the efficiency of your algorithm.*

(f) In the following two sections you will implement code for a brute force solution and an efficient solution. In your report, use the provided data files to plot the number of jobs (x-axis) against the time in ms it takes for your code to run (y-axis). There are four small data files and four large data files included in the input provided. The large data files may be too large for the brute force algorithm to finish running on your machine. If that is the case, do not worry about plotting the brute force results for the large data files. Your plot should therefore contain 8 points from your efficient algorithm and 4-8 points from the brute force algorithm. Please make sure the points from different algorithms are distinct so that you can easily compare the run times from the brute force algorithm and your efficient algorithm. Scale the plot so that the comparisons are easy to make *(we recommend a logarithmic scaling)*. Also take note of the trend in run time as the number of jobs increases.

Part 2: Implement a Brute Force Solution [20 points]

A brute force solution to this problem involves generating all possible permutations of workers and jobs, and checking whether each one is a stable matching, until a stable matching is found. For this part of the assignment, you are to implement a function that verifies whether or not a given matching is stable. We have provided most of the brute force solution already, including input, function skeletons, abstract classes, and a data structure. Your code will go inside a skeleton file called `Program1.java`. A file named `Matching.java` contains the data structure for a matching. See the instructions section for more information.

Inside `Program1.java` is a placeholder for a verify function called `isStableMatching()`. Implement this function to complete the Brute Force algorithm (the rest of the code for the brute force algorithm is already provided).

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.

Part 3: Implement an Efficient Algorithm [35 points]

We can do better than the above using an algorithm similar to the Gale-Shapley algorithm. Implement the efficient algorithm you devised in your report. Again, you are provided several files to work with. Implement the function `stableHiringGaleShapley()` 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 favourite development environment. We will be grading in Java 1.8. Therefore, we recommend you use Java 1.8. **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.
- 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. 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**: The number of jobs
  - **n**: Number of workers
  - **job_preference**: An ArrayList of ArrayLists containing each of the job's preferences of workers, in order from most preferred to least preferred. The jobs are in order from 0 to $n-1$. Each job has an ArrayList that ranks its preferences of workers who are identified by numbers 0 through $n-1$. Hard workers are ranked higher than lazy workers.
  - **worker_preference**: An ArrayList of ArrayLists containing each of the worker's preferences for jobs, in order from most preferred to least preferred. The workers are in order from 0 to $n-1$. Each worker has an ArrayList that ranks its preferences of jobs who are identified by numbers 0 to $n-1$. Full-time jobs are ranked higher than part-time jobs.
  - **job_is_fulltime**: An ArrayList representing whether each job is full time or half time.
  - **worker_is_hardworking**: An ArrayList representing whether each worker is a hard worker or a lazy worker.
  - **worker_matching**: An ArrayList to hold the final matching. This ArrayList (should) hold the number of the job each worker 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 `setWorkerMatching(<your_solution>)` or constructing a `new Matching(marriage, <your_solution>)`, where `marriage` is the Matching we pass into the function. The index of this ArrayList corresponds to each worker. The value at that index indicates to which jobs he/she is matched. A value of -1 at that index indicates that the worker is not matched up. For example, if worker 0 is matched to job 55, worker 1 is unmatched, and worker 2 is matched to job 3, the ArrayList should contain {55, -1, 3}.

- The inputs to the program are to be provided in the following way:
  ⟨ number_of_jobs ⟩ ⟨ number_of_workers ⟩
  ⟨ preference list of first job ⟩
  ⟨ preference list of second job ⟩
  ...
  ⟨ preference list of first worker ⟩
  ⟨ preference list of second worker ⟩
  ...
  In the preference list of each job, the first value represents whether that job is full-time or not. If the value is 1, then the job is full-time, else if the value is 0, the job is part-time.
  In the preference list of each worker, the first value represents whether that worker is hard-working or not. If the value is 1, then the worker is hard-working, else if the value is 0, the worker is lazy.

- You must implement the methods `isStableMaching()` and `stableHiringGaleShapley()` 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.

- `Driver.java` is the main driver program. Use command line arguments to choose between brute force and your efficient algorithm and to specify an input file. Use -b for brute force, -g for the efficient algorithm, and input file name for specified input (i.e. `java -classpath . Driver [-g] [-b] <filename>` on a linux machine).

- When you run the driver, it will tell you if the results of your efficient algorithm 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 `verify()` 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).

- It is **highly recommended** that you do comment your code and follow good programming style. It will be very difficult for TAs to justify giving partial credit when they cannot understand what your code is/should be doing.

- If you are unsure how to do the plot in the report, we recommend the following resources. If you are on linux: `http://www.gnuplot.info/`. If you are using Windows: `http://www.online-tech-tips.com/ms-office-tips/excel-tutorial-how-to-make-a-simple-graph-or-chart-in-excel/`.

- We suggest using System.nanoTime() to calculate your runtime.

- Before you submit, be sure to turn your report into a PDF and name your PDF file `eid_lastname_firstname.pdf`.

Part 4: Implement with Ties [15 points]
In this part of the problem, we allow ties in the preference lists. That is, each of the worker's preference lists can assign the same rank to more than one jobs. Similarly, each of the employer's preference lists can assign the same rank to more than one workers. However, each worker still prefers any full-time job over any part-time job, and each employer still prefers any hard worker over any lazy worker. We will say a worker w prefers job j over a job j' if j is ranked higher than j' on $w's$ preference list(they are not tied). With indifferences in rankings, there could be two natural notions for instability as follows.

(a) A *strong instability* in a perfect matching S consists of a worker w and a job j such that each of w and j prefer the other to their partners in S.

(b) A *weak instability* in a perfect matching S consists of a worker w and a job j, such that their partners in S are j' and w' respectively, and one of the following holds:

- w prefers j to j', and j either prefers w to w' or is indifferent between the two choices, or
- j prefers w to w', and w either prefers j to j' or is indifferent between the two choices.

In other words, the pairing between w and j is either preferred by both, or preferred by one while the other is indifferent.

Modify your report to answer the following:

(a) Does there always exist a perfect matching with no *weak instability*? Prove, or disprove with a counterexample.

(b) Does there always exist a perfect matching with no *strong instability*? Prove, or disprove with a counterexample.

(c) How will you modify your algorithm from **Part 1(c)** to find a stable matching (*you need to figure out what stable means here*)?

(10 points) Implement your modified algorithm by modifying your `stableHiringGaleShapely()` method. You will also need to modify the `isStableMatching()` method to verify if the matching stable according to the new definition of stability.

What To Submit
You should submit a single zip file titled `eid_lastname_firstname.zip` that contains all of your java files and pdf report `eid_lastname_firstname.pdf`. Do not put these files in a folder before you zip them (i.e. the files should be in the root of the ZIP archive). Your solution must be submitted via Canvas BEFORE 11:59 pm on February 14, 2017.