EE360C Algorithms
University of Texas at Austin
Prof. Nur Touba
Student Name: Yonatan Teshome
Student EID: YH23572
Programming Assignment #1
Date: July 1, 2025

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 on the **Sunday at 11:59pm** shown in the schedule. To encourage students to get an early start on it, 2% extra credit is given if you submit your assignment by Friday 11:59pm before the Sunday in which it is due. If you are unable to complete the assignment by the due date, you may submit the assignment late until Tuesday at 11:59pm after the due date with a 20 point late 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.

Gale and Shapley published their paper on the Stable Matching Problem in 1962. According to New York Times, in 2004, New York City matched eighth graders to high schools in this way which plummeted the number of unmatched students from 31,000 to about 3,000.

The situation is the following: There are n eighth-grade students who submit applications for high school in a given year, each interested in attending one of m available high schools in New York City, each with a limited number of admission spots. Each school has a ranking of the students in order of preference, and each student has a ranking of the schools in order of preference. We will assume that there are at least as many students graduating as the total admission spots available across all m schools (each school can have a different admission quota). The interest lies in finding a way of assigning each student to at most one school in such a way that all available spots across all schools are filled (since we are assuming a surplus of students, there may be some students who do not get assigned to any school).

We say that an assignment of students to high schools is *stable* if neither of the following situations arises:

- First type of instability: There are students s and s', and a high school h, such that

  - s is assigned to h, and
  - s' is assigned to no high school, and
  - h prefers s' to s
- Second type of instability: There are students s and s', and high school h and h', so that
  - s is assigned to h, and
  - s' is assigned to h', and
  - h prefers s' to s, and
  - s' prefers h to h'.

So we basically have the Stable Matching Problem as presented in class, except that (i) a school generally wants more than one student, and (ii) there is potentially a surplus of eighth-grade students. 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 **high school** 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 **student** 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 running time 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 student/high school optimality.

Part 2: Implement a Checker to check stability of any given matching [20 points]
Given a Matching object m, you should implement a boolean function to determine if the pairing of students to high schools (stored in the variable returned by m.getStudentMatching()) is stable or not. Your code will go inside a function called isStable(Matching x) inside Program1.java. A file named Matching.java contains the data structure for a matching. 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) (high school optimal) and (c) (student optimal) of your report. Again, you are provided several files to work with. Implement the function that yields a student optimal solution stableMatchingGaleShapley_studentoptimal() and high school optimal solution stableMatchingGaleShapley_highschooloptimal() 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.
- 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.
- **DO NOT add a package statement to the starter code**. This will result in your code not compiling, thus receiving a 0 for the code portion of this assignment.
- The main data structure for a matching is defined and documented in Matching.java. A Matching object includes:
  - **m**: Number of high schools
  - **n**: Number of students
  - **highschool preference**: An ArrayList of ArrayLists containing each of the high schools' preferences of students, in order from most preferred to least preferred. The high schools are in order from 0 to $m - 1$. Each high school has an ArrayList that ranks its preferences of students who are identified by numbers 0 through $n - 1$.
  - **student_preference**: An ArrayList of ArrayLists containing each of the student's preferences for high schools, in order from most preferred to least preferred. The students are in order from 0 to $n - 1$. Each student has an ArrayList that ranks its preferences of high schools who are identified by numbers 0 to $m - 1$.
  - **highschool spots**: An ArrayList that specifies how many admission spots each high school has. The index of the value corresponds to which high school it represents.
  - **student_matching**: An ArrayList to hold the final matching. This ArrayList (should) hold the number of the high school each student 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 setStudentMatching(<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 student. The

value at that index indicates to which high school he/she is matched. A value of -1 at that index indicates that the student is not matched up. For example, if student 0 is matched to high school 55, student 1 is unmatched, and student 2 is matched to high school 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_studentoptimal()
  - stableMatchingGaleShapley_highschooloptimal()

  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 spots available in each high school. The first integer represents the number of open spots in high school 0, the next integer represents the number for high school 1, and so on.
  - The next m lines are the preference lists of the high schools, where each space-separated integer represents a student. The list goes from left to right, from most to least desirable. The first of these m lines is the preference list for high school 0, the next line is for high school 1, and so on.
  - The next n lines are the preference lists of the students, where each space-separated integer represents a high school. The list goes from left to right, from most to least desirable. The first of these n lines is the preference list for student 0, the next line is for student 1, and so on.
  - Last line (optional): n space separated integers representing a student-high-school matching. If the first integer is *x*, then the first student is assigned to high school *x*, the second student 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 high school optimal or student optimal algorithms and to specify an input file. Use -gh for high school optimal, -gs for student optimal, and -bf for importing an existing matching (to check correctness of isStableMatching()). (i.e. java -classpath . Driver [-gh] [-gs] [-bf] <filename> on a Linux machine). As a test, the 3-10-3.in input file should output the following for both a student and high school optimal solution:
  - Student 0 high school -1

  - Student 1 high school 1
  - Student 2 high school -1
  - Student 3 high school -1
  - Student 4 high school -1
  - Student 5 high school -1
  - Student 6 high school -1
  - Student 7 high school 2
  - Student 8 high school 0
  - Student 9 high school -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.
- 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).

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. Failure to follow this naming format will result in a penalty of up to 5 points.

Your PDF report should be legibly scanned and submitted to Gradescope. Both your zipped code and PDF report must be submitted BY 11:59pm on Sunday as shown in the class schedule. If you are unable to complete the assignment by this time, you may submit the assignment late until Tuesday at 11:59pm for a 20 point penalty.

Report Section: Stable Matching

a) Pseudocode High School Optimal Matching

Proposer: Schools
Input: School preference, Student preference, and available spots at school
Output: "Gale-Shapley high school Optimal: stable? (true/false)"

Solve:

1. **Initialize students as unmatched**
2. **For each school we track:**
    - **a. # of available spots**
    - **b. Index of the next student to propose to**
3. **Add all schools with available spots to a que**
4. **While the queue is empty:**
    - **a. Pop school h from the queue**
    - **b. While h has open spots and has not proposed to all students;**
        - **i. Let student s = h's next most preferred student**
        - **ii. If s is unmatched:**
            1. **Match s to h**
        - **iii. Else if s prefers h over their current match h_1:**
            1. **Unmatch s from h_1**
            2. **Match s to h**
            3. **Add h_1 back to the ques if it has open spots now**
        - **iv. Else:**
            1. **S rejects the proposal**
    - **c. If h still has open spots, readd h to the queue**
5. **Return the final student to school matches**

b) Runtime Complexity High School Optimal

The outer loop runs while there are schools unfilled quotas each school can propose to each student at most once.

N = number of students
M = Number of schools

Each proposal takes $O(1)$ time with a rank lookup
At most $O(M*N)$ proposals are made

Total Runtime = $O(MN)$

c) Pseudocode Student Optimal Matching

Input: School preference, Student preference, and available spots at school
Output: "Gale-Shapley Student Optimal: stable? (true/false)"

Solve:

6. **Initialize students as unmatched**
7. **For each school we track:**
    a. **The index of the next school to propose to**
8. **Add all students with available spots to a que**
9. **For each school, initialize an empty set of accepted students**
10. **While free queue is NOT empty:**
    a. **Pop student s from the queue**
    b. **If s has no schools left to propose to**
        i. **Skip**
    c. **Let h = s's next preferred school**
    d. **If h has an available spot:**
        i. **Accept student s**
    e. **Else:**
        i. **Find the worst ranked student s_1 in h's accepted list**
        ii. **If h prefers s over s_1:**
            1. **Replace s_1 with s**
            2. **Add s_1 back to the queue**
        iii. **Else:**
            1. **Reject student s**
            2. **Add student s back to the queue**
11. **Return the final student to school matches**

d) Runtime Complexity Student optimal

Each student proposes to each school at most once

N= number of students
M= number of schools

Each proposal takes $O(1)$ time with a rank lookup
At most $O(n*m)$ proposals are made

Total Runtime: $O(N*M)$