Project 5: Stable marriage problem v3

Contents

New concepts: Inheritance

1 Overview

This project builds upon the previous project, where a program was built to match students and schools using principles from the stable marriage problem (SMP). This version of the project will generalize the situation so that this code serves as base to match any two groups, not just students and schools. Participants in the matching will also be allowed to have more than one match (e.g., schools can have more than one opening for students). The remainder of this section provides an overview of SMP and the student-school matching problem from the previous project.

1.1 The stable marriage problem

SMP is described as follows. Say you have a group of n men and n women, and they need to be matched together in marriage.The men have ranked all the women in order of preference (1 is most preferred, n is least preferred), and the women have similarly ranked all the men in order of preference. The matching solution should make sure that each man is married to exactly one woman and vice versa, and that the marriages are **stable**. A stable matching solution means that there is no one person who would rather be with someone else who would also prefer to be with them. In other words, no two people would want to have an affair with each other.

1.2 Students, schools and matching

The program will store students' names, grade point averages (GPAs), extracurricular scores, and ratings of schools. GPAs are on a scale of 0-4 (4 being highest), and can be fractional. Extracurricular scores measure

how involved a student has been in extracurricular activities, and are integers on a scale 0-5 (with 5 being most involved). Students' rankings of school are provided by the user.

The program will store schools' names, GPA weights (α), and rankings of students. The GPA weight is a value in $[0, 1]$ that indicates how much a school emphasizes GPA score over extracurricular score. For example, if $\alpha = 0.90$, then the school puts 90% of its assessment of a student on GPA, and 10% on extracurriculars. Each school creates a composite score for each student using the following formula:

$$\text{composite score} = \alpha G + (1 - \alpha)E$$

where G is the student's GPA and E is the student's extracurricular score. Each school has its own α, and ranks the students according the composite scores (where higher scores are higher ranked). Thus, schools' rankings of students are not entered manually, but are automatically calculated based on α. Ties are broken in the same manner as in the previous project.

Matching is done using the Gale-Shapley algorithm [2]. The resulting solution is guaranteed to be stable and **suitor-optimal**. The quality of matching is defined by whether or not the matching is stable and the total **regret** of the participants. Regret is the difference in rank between a participant's top choice and the match they ended up with.

2 Program description

In this project, we will do three things:

1. Make the scenario more realistic by allowing schools to accept multiple students
2. Generalize the code using inheritance so that the matching framework can be used for any two groups of participants, not just students and schools
3. Compare student-optimal and school-optimal solutions

Loading students and schools is the same as in the previous project, but there is an extra integer at the end of each school line to indicate how many openings the schools has. This number must be ≥ 1; if it is not, then the school is rejected.

2.1 Many-to-one Gale-Shapley algorithm

The Gale-Shapley algorithm [2] can be easily extended to allow one group of participants to accept multiple matches (in this case, schools accept multiple students, while students can only accept one school). Using the student-school example with students as suitors (though the process is the same with schools as suitors), the many-to-one Gale-Shapley algorithm can again be simply described as just two steps:

1. Each free student proposes to the most preferred school to whom s/he has not yet proposed.
2. Each school accepts the proposal from the most preferred student, to whom it gets "engaged."
    - If a school still has spots available, it must accept the new student.
    - If a full school receives a proposal and this new student is preferred over its least-preferred matched student, it kicks out the least-preferred student and accepts the new student. The least-preferred student is now free.

These two steps are repeated for every free student $1, \ldots, n$ in order until everyone is engaged (i.e., matched to a school). Note that each student makes at most one proposal per iteration (one proposal if not engaged, no proposals if already engaged). For simplicity, we will require that the total number of openings at all the schools be equal to the number of students. Thus, the algorithm ends when every student is matched, which is the same as when every school is full. Alternatively, you could think of the matching being done when all suitors or receivers are full. The regret of the matching is the sum of all regret over all matches.

2.2 Generalizing the participants

Hopefully, by now you have realized that there is an incredible amount of redundancy between the `Student` and `School` classes. They both have names, an array of rankings, and an index to indicate their match, and almost all the methods are the same. The only uniqueness between students and schools is that students have a GPA and extracurricular score (and thus a unique print method) and their rankings can be manually edited, while schools have a GPA weight α (and thus a unique print method) and their rankings are automatically calculated. In fact, any possible participant in any matching will have the same info that is common to both students and schools (names, rankings, match), so we should have a general class that describes a basic participant instead of reinventing the wheel every time we have a new type of participant.

This new general participant class will be called `Participant`, and it will hold all the values that are common to any possible SMP participant: names, rankings, and matches. It will also hold all the common functionality (e.g., getters/setters, printing matches). Note that we want to be able to store more than one match in case a participant allows multiple matches (e.g., schools can accept multiple students). The `Student` and `School` classes will extend the `Participant` class. The `SMPSolver` class will be similarly generalized to hold two sets of generic `Participant`s instead of specifically only holding one set of `School`s and one set of `Student`s. The generic nature of the improved `SMPSolver` class means that we can easily switch between students as suitors and schools as suitors! Note that any messages printed by the `SMPSolver` must be updated to generically say "suitors" and "recipients" instead of "students" and "schools".

From the perspective of someone writing the main program, nothing will have changed (aside from modifying the file loading for schools to allow for the number of openings to be read in). However, if the person writing this program wants to match US Navy sailors to ships [3] or employers to employees [1], s/he will not have to start from scratch. S/he can just inherit from the `Participant` class if needed to define unique classes (which may not even be necessary) and then directly use the `SMPSolver` without any modification. Thus, any future matching problems can be programmed and solved with minimal effort.

2.3 Comparing student-optimal and school-optimal solutions

Since the new generic `SMPSolver` class will allow us to set either of the two participant groups as the suitor, we will try doing the matching first with students and suitors, and then with schools as suitors, so we can see how different the solutions are. Solutions will be measured according to four metrics:

1. Average student regret
2. Average school regret
3. Average total regret
4. Computation time

The best performing solution will be identified for each category.

3 Menu options

- `L - Load students and schools from file`
  Get file names from the user, then load the files.
- `E - Edit students and schools`
  Go to a sub-area with its own menu where students and schools can be edited.
- `P - Print students and schools`
  Print student and school information, including any existing matches and rankings.
- `M - Match students and schools using Gale-Shapley algorithm`
  Perform matching using the Gale-Shapley algorithm first with students as suitors, then with schools as suitors.

- `D - Display matches`
  Display the matches for both the student-optimal and school-optimal solutions.
- `X - Compare student-optimal and school-optimal matches`
  Display a table comparing the student-optimal and school optimal performance metrics.
- `R - Reset database`
  Clear out the students and schools.
- `Q - Quit`
  Quit the program.

4 Error messages

- `ERROR: Invalid menu choice!`
  when the user enters an invalid menu choice.
- `ERROR: No students are loaded!`
  when the user attempts to edit students before students have been loaded.
- `ERROR: No schools are loaded!`
  when the user attempts to edit schools before schools have been loaded.
- `ERROR: No suitors are loaded!`
  when the user attempts to perform matching before suitors have been loaded into the solver.
- `ERROR: No receivers are loaded!`
  when the user attempts to perform matching before schools have been loaded into the solver.
- `ERROR: The number of suitor and receiver openings must be equal!`
  when the user attempts to perform matching, but the number of suitor and receiver openings are not equal.
- `ERROR: No matches exist!`
  when the user attempts to display match statistics without performing matching.
- `ERROR: Choice must be 'y' or 'n'!`
  when the user enters an invalid choice for editing a student's ranking of schools.
- `ERROR: Rank <r> already used!`
  when the user enters a rank that has already been used while editing a student's ranking of schools.
- `ERROR: Input must be an integer in [<LB>, <UB>]!`
  where `LB` and `UB` are lower and upper bounds. If UB is equal to the maximum storable value of an `int`, then "`infinity`" is displayed instead of the actual UB value. Similarly, if LB is equal to the negative of the maximum storable value of an `int`, then "`-infinity`" is displayed instead of the actual LB value.
- `ERROR: Input must be a real number in [<LB>, <UB>]!`
  where `LB` and `UB` are lower and upper bounds. If UB is equal to the maximum storable value of a `double`, then "`infinity`" is displayed instead of the actual UB value. Similarly, if LB is equal to the negative of the maximum storable value of a `double`, then "`-infinity`" is displayed instead of the actual LB value.

5 Required elements

Your project must use the each of the elements described in this section. You can have more functions, variables, and objects if you want, but you must at least use the elements described in this section.

Failure to correctly implement any of the elements in this section may result in a zero on the project.

5.1 Variables

- An array or ArrayList of `Student` objects, as defined in Section 5.2
- An array or ArrayList of `School` objects, as defined in Section 5.2
- Two `SMPSolver` objects, as defined in Section 5.2
- Global `public static BufferedReader` as **the only** `BufferedReader` object in the entire program for reading user input
  - Although your programs will compile and run just fine on your own computer with multiple `BufferedReader` objects, they will not run correctly on a web server with multiple `BufferedReader` objects. Since your programs will be graded on a web server, please don't have more than one `BufferedReader` for reading input from the console.

5.2 Classes

You are required to write three classes:

1. `Participant` class to store basic participant data
2. `Student` class inherited from `Participant` to store student-specific data
3. `School` class inherited from `Participant` to store school-specific data
4. `SMPSolver` class to run the Gale-Shapley algorithm and store matching results

You can write more classes, and you can add to these classes, but you must at least implement the classes defined in this section.

```
public class Participant {
    private String name;     // name
    private int[] rankings;  // rankings of participants
    private ArrayList<Integer> matches = new ArrayList<Integer>();// match indices
    private int regret;      // total regret
    private int maxMatches;  // max # of allowed matches/openings

    // constructors
    public Participant()
    public Participant(String name, int maxMatches, int nParticipants)

    // getters
    public String getName()
    public int getRanking(int i)
    public int getMatch(int i)
    public int getRegret()
    public int getMaxMatches()
    public int getNMatches()
    public int getNParticipants() // return length of rankings array
    public boolean isFull()

    // setters
    public void setName(String name)
    public void setRanking(int i, int r)
    public void setMatch(int m)
    public void setRegret(int r)
    public void setNParticipants(int n) // set rankings array size
    public void setMaxMatches(int n)

```

```
    // methods to handle matches
    public void clearMatches() // clear all matches
    public int findRankingByID(int k) // find rank of participant k
    public int getWorstMatch() // find the worst-matched participant
    public void unmatch(int k) // remove the match with participant k
    public boolean matchExists(int k) // check if match to participant k exists
    public int getSingleMatchedRegret(int k) // get regret from match with k
    public void calcRegret() // calculate total regret over all matches

    // methods to edit data from the user
    public void editInfo(ArrayList<? extends Participant> P)
    public void editRankings(ArrayList<? extends Participant> P)

    // print methods
    public void print(ArrayList<? extends Participant> P)
    public void printRankings(ArrayList<? extends Participant> P)
    public String getMatchNames(ArrayList<? extends Participant> P)

    // check if this participant has valid info
    public boolean isValid()
}
```

Participant_Template.java

```
public class Student extends Participant {
  private double GPA;      // GPA
  private int ES;          // extracurricular score

  // constructors
  public Student()
  public Student(String name, double GPA, int ES, int nSchools)

  // getters and setters
  public double getGPA()
  public int getES()
  public void setGPA(double GPA)
  public void setES(int ES)

  public void editInfo(ArrayList<School> H, boolean canEditRankings) // user info
  public void print(ArrayList<? extends Participant> H) // print student row
  public boolean isValid() // check if this student has valid info
}
```

Student_Template.java

```
public class School extends Participant {
  private double alpha; // GPA weight

  // constructors
  public School()
  public School(String name, double alpha, int maxMatches, int nStudents)

  // getters and setters
  public double getAlpha() { return this.alpha; }
  public void setAlpha(double alpha) { this.alpha = alpha; }

  // get new info from the user; cannot be inherited or overridden from parent
  public void editSchoolInfo(ArrayList<Student> S, boolean canEditRankings)

  public void calcRankings(ArrayList<Student> S) // calc rankings from alpha
```

```
  public void print(ArrayList<? extends Participant> S) // print school row
  public boolean isValid()  // check if this school has valid info
}
```

School_Template.java

```
public class SMPSolver {
  private ArrayList<Participant> S = new ArrayList<Participant>(); // suitors
  private ArrayList<Participant> R = new ArrayList<Participant>(); // receivers
  private double avgSuitorRegret;   // average suitor regret
  private double avgReceiverRegret; // average receiver regret
  private double avgTotalRegret;    // average total regret
  private boolean matchesExist;     // whether or not matches exist
  private boolean stable;           // whether or not matching is stable
  private long compTime;            // computation time
  private boolean suitorFirst;      // whether to print suitor stats first

  public SMPSolver()  // constructor

  // getters
  public double getAvgSuitorRegret() { return this.avgSuitorRegret; }
  public double getAvgReceiverRegret() { return this.avgReceiverRegret; }
  public double getAvgTotalRegret() { return this.avgTotalRegret; }
  public boolean matchesExist()
  public boolean isStable()
  public long getTime()
  public int getNSuitorOpenings()
  public int getNReceiverOpenings()

  // setters
  public void setMatchesExist(boolean b)
  public void setSuitorFirst(boolean b)
  public void setParticipants(ArrayList<? extends Participant> S, ArrayList<?
      extends Participant> R)

  // methods for matching
  public void clearMatches() // clear out existing matches
  public boolean matchingCanProceed() // check that matching rules are satisfied
  public boolean match() // Gale-Shapley algorithm to match; students are suitors
  private boolean makeProposal(int suitor, int receiver) // suitor proposes
  private void makeEngagement(int suitor, int receiver, int oldSuitor) // make
      suitor-receiver engagement, break receiver-oldSuitor engagement
  public void calcRegrets()  // calculate regrets
  public boolean determineStability()  // calculate if a matching is stable

  // print methods
  public void print()  // print the matching results and statistics
  public void printMatches()  // print matches
  public void printStats()  // print matching statistics
  public void printStatsRow(String rowHeading) // print stats as row
}
```

SMPSolver_Template.java

5.3 Functions

Function prototypes and short descriptions are provided below. It is your job to determine exactly what should happen inside each function, though it should be clear from the function and argument names and

function descriptions. You can write more functions (and are strongly recommended to), but you must at least implement the functions defined in this section.

Only ArrayLists are shown in the function prototypes, but you may always use an array instead of an ArrayList. If you use arrays, you may pass extra arguments to and from the functions to act as counters.

- `public static void displayMenu()`
  Display the menu.
- `public static int loadStudents(ArrayList<Student> S, ArrayList<School> H)`
  Load student information from a user-provided file and return the number of new students. New students are added to the list of existing students.
- `public static int loadSchools(ArrayList<School> H)`
  Load school information from a user-provided file and return the number of new schools. New schools are added to the list of existing schools.
- `public static void editData(ArrayList<Student> S, ArrayList<School> H)`
  Sub-area menu to edit students and schools.
- `public static void editStudents(ArrayList<Student> S, ArrayList<School> H)`
  Sub-area to edit students. The edited student's regret is updated if needed. Any existing school rankings and regrets are re-calculated after editing a student.
- `public static void editSchools(ArrayList<Student> S, ArrayList<School> H)`
  Sub-area to edit schools. Any existing rankings and regret for the edited school are updated.
- `public static void printStudents(ArrayList<Student> S, ArrayList<School> H)`
  Print students to the screen, including matched school (if one exists).
- `public static void printSchools(ArrayList<Student> S, ArrayList<School> H)`
  Print schools to the screen, including matched student (if one exists).
- `public static void printComparison(SMPSolver GSS, SMPSolver GSH)`
  Print comparison of student-optimal and school-optimal solutions.
- `public static int getInteger(String prompt, int LB, int UB)`
  Get an integer in the range [LB, UB] from the user. Prompt the user repeatedly until a valid value is entered.
- `public static double getDouble(String prompt, double LB, double UB)`
  Get a real number in the range [LB, UB] from the user. Prompt the user repeatedly until a valid value is entered.

6 Helpful hints

- All the hints from the previous project are still useful.
- In the new classes, there is some new syntax in the input arguments in some of the methods. For example, in the `print` method of the `Participant` class, we have

```
1 public void print(ArrayList <? extends Participant > P)
```

  The `?` means that any object that is a child of the `Participant` class can be passed into the function. This syntax is necessary because for these functions, sometimes we will want to pass in `School` objects, and other times we will want to pass in `Student` objects, both of which are child classes of `Participant`.

- You should create two `SMPSolver` objects: one with students as suitors, one with schools as suitors. Remember that if you pass the ArrayList/arrays of students and schools into the two objects, you are only passing *references* to the ArrayLists/arrays. So, any changes to participants in one `SMPSolver` object will result in the same changes in the other `SMPSolver` object (which we do not want). Therefore, for one or both of these objects, you should make a copy of each participant ArrayList/array, and pass the copies into the `SMPSolver` object(s). While there are built-in Java functions to make copies of ArrayLists and arrays, they won't be effective here since the objects we are copying (`Participants`) have ArrayLists/arrays inside them, and those internal ArrayLists/arrays will not be duplicated when using the built-in Java copy functions. Instead, you should manually copy the entire participant ArrayList/array into a new ArrayList/array, and then pass the new ArrayList/array into one of the `SMPSolvers`. Here is an example for ArrayLists, with only the function for copying schools shown:

```
// in the main function after getting new Schools (H) and Students (S)
ArrayList<School> H2 = copySchools(H);
ArrayList<Student> S2 = copyStudents(S);
GSH.reset(H2, S2); // SMPSolver with high schools as suitors

// create independent copy of School ArrayList
public static ArrayList<School> copySchools(ArrayList<School> P) {
  ArrayList<School> newList = new ArrayList<School>();
  for (int i = 0; i < P.size(); i++) {
    String name = P.get(i).getName();
    double alpha = P.get(i).getAlpha();
    int maxMatches = P.get(i).getMaxMatches();
    int nStudents = P.get(i).getNParticipants();
    School temp = new School(name,alpha,maxMatches,nStudents);
    for (int j = 0; j < nStudents; j++) {
      temp.setRanking(j, P.get(i).getRanking(j));
    }
    newList.add(temp);
  }
  return newList;
}
```

- Since either the suitors or the receivers are allowed to have multiple matches, when breaking up an existing engagement between Suitor s and Receiver r, remember to remove s from r's matches *and* remove remove r from s's matches.

- The changes to your main program should be minimal. You are just adding one new function for comparison and one new `SMPSolver` object. Everything done in the previous project with the single `SMPSolver` object is now just done to both objects.

- If you were wondering, the reason why the `School` class cannot override its parent's `editInfo` method is due to the fact that Java strips away ArrayList types when it compiles, which means we get the following effective identical methods:

```
// parent method in Participant class
public void editInfo(ArrayList P)

// child method in School class
public void editInfo(ArrayList P)
```

  But, the child class (`School`) requires that the input exactly be an ArrayList of `Students` (in other words, the input argument must explicitly be `ArrayList<Student> P`), since GPAs are needed to calculate rankings, and generic `Participant` objects (or any other type of object) won't have the needed `getGPA()` method. So, this method cannot be overridden, and instead we just create a brand new method for the child class. Note that this problem does not exist for arrays.

- **Don't worry if your computation times are not exactly the same as the solution times.** The computation time is not only dependent on the algorithm, but also on the coding style: efficiency of loops, efficiency of data structures, pre-processing, etc. If your algorithms run faster than the solution program, Prof. Aleman wants to know, and wants to know if you are interested in doing research!

References

[1] V.P. Crawford and E.M. Knoer. Job matching with heterogeneous firms and workers. *Econometrica*, 49 (2):437–450, 1981.

[2] D. Gale and L. S. Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.

[3] D. Garrett, J. Vannucci, R. Silva, D. Dasgupta, and J. Simien. Genetic algorithms for the sailor assignment problem. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, GECCO '05, pages 1921–1928, New York, NY, USA, 2005. ACM. ISBN 1-59593-010-8.