Implementing the Gale-Shapley Algorithm with Grouped Preferences

INF421

2018–2019

In this DM, we ask you to implement the Gale-Shapley algorithm in Java. You will write a class `StableMatching` (in the file `StableMatching.java`.) The class must implement the interface `StableMatchingInterface`, whose definition is given to you in the file `StableMatchingInterface.java`. In the ZIP archive provided to you, you will find this interface file as well as other files for testing your code (in the `src` subdirectory), but the file `StableMatching.java` is absent. Your goal is to create and complete this file and submit it to the judging system. **Note: Do not modify or submit any other file. Do not use the 'package' keyword in your code.**

For your submission to be considered acceptable, we emphasize two requirements:

1. **Your code must be accepted by the Java compiler.** There should be no compilation error. To compile the code, you can use Eclipse, or you can use the command-line: just run `make` in the `src` directory.
2. **Your code must pass all the tests provided.** We have provided you with some files (`StableMatchingTest.java`, `Main.java`, etc.) whose job it is to run a series of tests on your code. Your code must pass all these tests. To run the test-suite, you can execute the `Main.java` program from Eclipse, or you can execute `make test` from the command-line.

Problem specification

In the considered variant of the stable matching problem, there are n men and n women. Each man is to marry one woman and each woman is to marry one man. Men are divided into m groups and similarly women are divided into w groups, with groups being possibly of different sizes. The sizes of all groups are given. Within each group, all individuals are alike: they are all equally attractive in the eyes of any person of the other gender, and they all have identical preference ordering with respect to groups of the other gender. The problem is essentially the same as the original stable matching task, but it allows us to represent preference orderings more compactly by a pair of matrices of dimensions $m \times w$ and $w \times m$ (representing the ordering of preferences for the groups of men and women, respectively) instead of a pair of $n \times n$ matrices (representing the ordering of preferences between individuals).

A matching is considered stable if there does not exist a man and a woman who would both (strictly) prefer to be with each other than with the partner they are currently married to. (It is sometimes said that such a matching is "weakly" stable.) The output of your program should be an integer matrix M of dimension $m \times w$, with entry $M_{i,j}$ denoting the number of men from the i-th group of men married to women from the j-th group of women.

Choice of difficulty

Your program will receive partial or full credit depending on the bounds on w, m, and n for which it works correctly. The tests are defined in the following way.

C. Score up to 12/20: All group sizes are 1 (i.e., $n = m = w$) and $n < 10^4$. You can apply the Gale-Shapley algorithm from the lecture directly, achieving $O(n^2)$ complexity.

B. Score up to 15/20: $n < 10^4$. You can expand all groups into individuals, creating a preference matrix of dimension $n \times n$. You can then apply the Gale-Shapley algorithm from the lecture to this matrix, achieving $O(n^2)$ complexity.

A. Score up to 20/20: $\max\{m, w\} < 10^4$ and $n < 10^9$. A runtime complexity of $O(n^2)$ will not be sufficient to pass all tests. You can still apply a variant of the Gale-Shapley algorithm on the group preference matrix, but with some optimizations. For example, try the approach below.

In each iteration, an unengaged man y belonging to some i-th group of men selects a woman to propose to from the j-th group of women, following the rule for the Gale-Shapley algorithm for the current iteration to choose j. The woman x from group j, to whom man y proposes, should have the following relationship status: either she is unengaged, or if there are no unengaged women in group j, she is chosen as a woman from group j having the least attractive fiance from among all current engagements of women in group j. Suppose this fiance comes from some group of men i'.

If the proposal of y to x were to succeed, this means that either x was unengaged, or group i' is less attractive than group i on her preference list. Then, let a denote the number of unengaged men in group i, and let b denote the number of women in group j having exactly the same relationship status as woman x (i.e., unengaged or engaged to a man from group i'). In this case, in the designed algorithm, precisely $\min\{a, b\}$ unengaged men from the i-th group *simultaneously* propose to the corresponding number of women from the j-th group, having exactly the same relationship status as woman x. Clearly, all these proposals lead to new engagements (and also to break-ups with men from group i', if the women were previously engaged).

Otherwise, if the proposal of y to x were to fail, group of men i can permanently discard group of women j from their preference list, and no man from group i will ever propose to a woman from group j in the future.

To obtain good running time complexity, it is sufficient that group i is chosen in each iteration so that the number of unengaged men in it (a) is at least a $\frac{1}{2m}$ fraction of the number of all currently unengaged men in the population. So, for example, you can simply choose group i as the one containing the most unengaged men.

Why does this approach work? We can bound its computational complexity by first observing that one of the following statements holds for each iteration:

- No future proposal from group i of men to group j of women will happen in the future. Or;

- The number of currently unengaged men who enter into an engagement in the current iteration is at least a $\frac{1}{2m}$ fraction of all currently unengaged men; Or
- All currently unengaged women from group j become engaged to men from group i, and so all women from group j will always be engaged to some man in the future; Or
- All women from group j who were engaged to men from the group i' directly before the current round of proposal break these engagements. Then, no future engagement between group i' of men and group j of women will happen in the future.

Noting that a man can enter into an engagement at most w times, and that the total number of unengaged men never increases, by a careful analysis, we obtain that the number of unengaged men decreases by at least half every at most $O(mw)$ iterations of proposals. We can then bound the number of iterations, and consequently also the runtime complexity of the algorithm, as $O(\min\{mw \log n, n^2\})$.

Some details

The test-suite will evaluate your `StableMatching` class over a number of instances of the stable marriage problem, both small instances and large. Some of these instances were pseudo-randomly generated; others were chosen to test for special corner cases. All of these instances (within the chosen scope of difficulty of cases) should be solved by your algorithm. We will verify that your code terminates (within a reasonable number of seconds), that it does not throw an exception, and that it produces a valid stable matching solution.

If everything works as expected, executing the test-suite should print out a series of messages that looks something like the following:

```
n = 25, m = 25, w = 25: running...
Elapsed time: 0 milliseconds
SUCCESS!

n = 26, m = 3, w = 4: running...
Elapsed time: 0 milliseconds
SUCCESS!
```

(Note that the time taken by your program may differ from the messages above.)

However, if a test fails, you will see a message that lists input that was tested, the result that your code produced, and why this result was unacceptable. For example, you may see:

```
n = 2, m = 2, w = 2: running...
Elapsed time: 0 milliseconds
FAILURE: NOT A STABLE MATCHING!
The pair formed by a man from group 1 and a woman from group 0 is unstable.
Indeed, some man from group 1 prefers a woman from group 0 to his bride from group 1
and some woman from group 0 prefers man from group 1 to her groom from group 0.
The parameters of this test run were:
...
```

The result of running the full test-suite may be quite long, and it may be useful to save this result in a file that you can study in more detail. If you use the command-line, executing `make test > log.txt` will save the output of the tests in a file called `log.txt`. To achieve the same result with Eclipse, you should first click on the menu `Project`, then click on `Properties`, then on `Run/Debug Settings`, select a «run configurations», then click on the button `Edit`, choose the tab `Common` and choose to send the («*standard output*») not to the «console» but instead to a file of your choice (e.g. `log.txt`).

In some cases, you may want the execution of the test-suite to stop at the first failure. You can obtain this behavior by setting the constant `STOP` in the file `StableMatchingTest.java`.

General Advice

Do not be tempted to prematurely optimize your code. Your code must, most importantly, be clear and correct. Its absolute speed is not very important, provided that it has the required asymptotic time and space complexity (depending on the chosen target score).

If you use Eclipse, you may have to change the «run configurations» to increase the heap size (because our tests take a lot of memory.) To do this, go to «run configurations» (as described above), then choose the tab `Arguments` and insert the phrase `-Xmx3G` in the field titled `VM arguments`. This increases the memory available to your program to 3GB.

We don't require you to comment your code, but it will definitely be useful to you and to us if you insert comments that explain the functioning of the algorithm.

You can also insert assertions into your code using the `assert` statement. The cost of doing so is minimal, but you will find that it can help you detect coding errors more easily. In Eclipse, the `assert` statement is deactivated by default. To enable it, follow the steps described above to access the `Arguments` tab in «run configurations» and add the phrase `-ea` to the `VM arguments` field.

This DM is meant to be completed individually. You are, of course, free to study and discuss the principles of the Gale-Shapley algorithm in groups. However, we require that you write your code by yourself. Plagiarism will be detected!