CS 445: Data Structures
Summer 2016

Assignment 2

Assigned: Friday, June 17 **Due:** Wednesday, July 6 11:59 PM

---

1 Motivation

In this assignment, you will revisit the social networking themes from Assignment 1 and implement a coupon distribution algorithm that takes into account the "friends" relationship between users.

Recall that in Assignment 1, we considered the *directed* social relationship in which one profile *followed* another. In this assignment, we consider the *undirected* social relationship in which two users are *friends*. That is, unlike in Assignment 1, if Abby is friends with Bryan, then we know that Bryan is also friends with Abby.

Imagine, in your social network, that you have just been granted a sponsorship by an advertising agency. Under this sponsorship, you are to distribute a series of dining and entertainment coupons to the users of your network. To encourage the redemption of such coupons in groups, each coupon is good for use by several people who purchase together. Therefore, the agency stipulates that *every* user should receive one coupon, but no two users who are friends should receive the *same* type coupon (e.g., if Abby and Bryan are friends, they *must* receive different types of coupons).

2 Description

Your job is to find an assignment of coupons to users, or determine that a valid assignment is impossible. You are given a number of coupons as well as a description of each user's friends. Every user must receive a coupon. You may assign the same coupon to multiple users, but only if they are not friends. You may also assign less than the total number of coupons, if all users can receive a coupon without using them all.

You *must* accomplish this using the backtracking techniques discussed and demonstrated in class. That is, you need to build up a solution recursively, one assignment at a time, until you determine that the current coupon assignment is impossible to complete (in which case you will backtrack and try another assignment), or that the current coupon assignment is complete and valid. It may be helpful to review the *8 Queens* solution available `https://cs.pitt.edu/~bill/445/code/Queens.java`, when developing your program.

2.1 Input Format

In order to receive the friendship relation, your program should read in a square, space-separated table from a file, where value (`i`, `j`) is `1` if users `i` and `j` are friends, and `0` otherwise. For instance, consider the following table:

$$\begin{matrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{matrix}$$

In this friends table, users 1 and 2 are friends, users 1 and 3 are friends, and users 3 and 4 are friends. Users 2 and 3 are **not** friends, and users 1 and 4 are **not** friends.

Note that, because the friendship relation is unordered, the table must be a square and must be symmetrical: the values in `(i, j)` and `(j, i)` must be the same. Also, users should not be friends with themselves: `(i, i)` must be 0 for all `i`. If the table is invalid, your program should print an error and quit.

2.2 Arguments and Output Format

Your class should be named `FriendsCoupon`, and therefore should be in a file with the name `FriendsCoupon.java`. Your program must be usable from the command line using the following format for arguments:

```
java FriendsCoupon table_file.txt number_of_coupons
```

If there is a way to assign `number_of_coupons` coupons to the users specified in `table_file.txt` while following the rules specified above, your program should output which coupon (lettered $A, B, C, \ldots$) is assigned to each user (numbered $1, 2, 3, \ldots$). You do not need to find every possible solution, just one.

If there is no way to assign every user a coupon with only `number_of_coupons` coupons while following the rules specified above, your program should output a message indicating as such.

2.3 Required Methods

As stated above, you **must** use the techniques we discussed in week 5 of lecture for recursive backtracking. As such, you will first need to determine a format for your partial solution. You will then need to write the following methods to support your backtracking algorithm.

- `isFullSolution`, a method that accepts a partial solution and returns `true` if it is a complete, valid solution.
- `reject`, a method that accepts a partial solution and returns `true` if it should be rejected because it can never be extended into a complete solution.
- `extend`, a method that accepts a partial solution and returns another partial solution that includes one additional choice added on. This method will return `null` if there are no more choices to add to the solution.
- `next`, a method that accepts a partial solution and returns another partial solution in which the *most recent* choice to be added has been changed to its next option. This method will return `null` if there are no more options for the most recent choice that was made.

2.4 Test Methods

In addition to the backtracking-supporting methods above, you will be required to test your methods as you develop them. Re-read starting at Chapter 2.16 to review the concepts behind writing test methods. To test each of the methods above, you need to write the following test methods. In each one, you should consider a wide variety of partial solutions that fit as many corner cases as you can think of, **including examples with different numbers of users**. Include enough test cases that the correct output **convinces** you that your method works properly in all situations.

- `testIsFullSolution`, a method that generates partial solutions and ensures that the `isFullSolution` method correctly determines whether each is a complete solution.
- `testReject`, a method that generates partial solutions and ensures that the `reject` method correctly determines whether each should be rejected.
- `testExtend`, a method that generates partial solutions and ensures that the `extend` method correctly extends each with the correct next choice.
- `testNext`, a method that generates partial solutions and ensures that the, in each, `next` method correctly changes the most recent choice that was added to its next option.

2.5 Example Inputs

Example friends tables are available at `https://cs.pitt.edu/~bill/445/a/a2examples.zip` for you to test.

Filename	Users	Minimum coupons
small.txt	4	2
medium.txt	14	3
large.txt	49	7

2.6 Example Outputs

```
$ java FriendsCoupon small.txt 2
A, B, B, A

$ java FriendsCoupon small.txt 3
A, B, B, A

$ java FriendsCoupon small.txt 1
No assignment possible

$ java FriendsCoupon medium.txt 3
A, A, A, A, B, C, C, C, C, B, B, B, B, A

$ java FriendsCoupon medium.txt 2
No assignment possible
```

2.7 Grading

Your grade for this assignment will be based on your program's success at solving instances of the coupon distribution problem (50%), the thoroughness of your test methods (40%), and your error checking of the input file (10%).

3 Submission

Create a zip file containing *only* java files (no class files!). Include `FriendsCoupon.java` and any other necessary java files, so that the TA can unzip your submission, then compile and run your program without any additional changes. All programs will be tested on the command line, so if you use an IDE to develop your program, you must export the java files from the IDE and ensure that they compile and run on the command line. Do not submit the IDE's project files.

Before uploading your final zip file, test that it works! Unzip your submission, compile your code, and run `FriendsCoupon` from the command line to ensure the TA will be able to complete these steps with the file you are turning in.

In addition to your code, you may wish to include a `README.txt` file that describes features of your program that are not working as expected to assist the TA in grading the portions that do work as expected.

Submit your zip file according to the instructions at `https://cs.pitt.edu/~bill/445/#submission`

Your project is due at 11:59 PM on Wednesday, July 6. Be sure to test the submission procedure in advance of this deadline: no late assignments will be accepted.