CS 445: Data Structures
Fall 2015

Assignment 3

**Assigned:** Wednesday, October 28 **Due:** Wednesday, November 11 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 each user should receive one coupon, and no two users who are friends should receive the same coupon (e.g., if Abby and Bryan are friends, they *must* receive different coupons).

# 2 What Do I Need To Do?

Your job is to find an assignment of coupons to users that satisfies the above requirements, or determine that such an assignment is impossible. You *must* do so 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 when developing your program.

## 2.1 Input Format

In order to specify 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:

| | | | |
|---|---|---|---|
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |

In this friends table, user 1 is friends with user 2, user 1 is friends with user 3, and user 3 is friends with user 4. User 2 is **not** friends with user 3 or user 4, and user 1 is **not** friends with user 4.

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 named `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 solution, just one.

If there is no 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 a message indicating as such.

### 2.3 Required Methods

As stated above, you must use the techniques we discussed in Lecture 12 for recursive backtracking. As such, you will 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 step added on.
- `next`, a method that accepts a partial solution and returns another partial solution in which the *most recent* step to be added has been changed to its next option.

### 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.

- `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 step.
- `testNext`, a method that generates partial solutions and ensures that the, in each, `next` method correctly changes the most recent step that was added to its next option.

Each test method should include enough test cases that the correct output convinces you that your method works properly. Your test method should cover as many corner cases as possible and ensure that they are all handled correctly.

### 2.5 Example Inputs

Example friends tables are available at `http://cs.pitt.edu/~bill/445/a/a3examples.zip` for you to test. A summary of these examples is shown below.

| Filename | Users | Coupons |
|---|---|---|
| small.txt | 4 | 2 |
| medium.txt | 11 | 4 |
| large.txt | 64 | 9 |

## 3 What (and how) do I submit?

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. Be sure to test this procedure (unzip, compile, run) before submitting.

All programs will be tested on the command line. If you use an IDE to develop your program, you must export the java files from the IDE and test that they compile and run on the command line. Do not submit the IDE's project files.

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 are working.

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

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