CSCI 6620 Data Structures Fall 2018

P9: Stable Matching, a Graph Algorithm

1 Goals

- Define and use a data structure that implements the adjacency list representation of a graph.
- Implement the stable-matching algorithm on this data structure.

2 The Reality Show

Imagine you are writing code to help implement a new television reality show called "Truth and Algorithms". On this show, there will be 20 contestants, 10 female and 10 male, all selected to be more-or-less compatible. On the first week, the 20 young people will be introduced to each other and to the audience. Audience members are urged to choose their favorite person, male or female, and send tweets favoring or panning one or another of the opposite-gender contestants.

During the next week, each one of the 10 young men will spend half a day with each one of the 10 young women. On show night, videos of their meetings will be shown on TV and the audience will vote for the five men they like best.

The next week each one of the men (in order of popularity with the audience) gets to choose five people to date again for a whole day (there are five days). The third week, each one of the women (in order of popularity with the audience) gets to choose five men to date again for a whole day.

At the end of the third week, each contestant writes down the names of the 10 people of the other gender, in preference order. Then the 20 lists of names are put into the Stable Matching algorithm, and it selects "mates" for everyone. Anyone who actually gets married within 3 months, gets a $1000 wedding dowry from the show. These are the people who knew who they were and what they wanted.

The whole group will get together for a party once a month for two years, and the TV audience gets to watch, tweet, and try to cause trouble. A jackpot of $20,000 will be divided equally among the couples who are still married two years later.

3 Instructions: Due December 6

These instructions are written for men doing the bidding and women receiving bids. The input file contains 10 of each, but you should write your code in terms of N, not 10, so that you can test the code first with a small number of people. (I suggest 4). Use a `#define` for N.

3.1 The Graph Class.

Data members. Implement a Graph data structure, with these data members:

- An array named `people` of 2*N Nodes (requires a default constructor in the Node class).
- A `queue<Node*>` to store the names of all the men who do not yet have mates.

The constructor and the input file. The Graph constructor will read the input file. The first line lists the names of 10 men, separated by whitespace. The second line lists the names of 10 women, separated by whitespace. Read these 20 names, and store them in the Nodes in the `people` array.

Then read the rest of the file, which contains the preferences. Each remaining line gives one person's name, followed by N preferences. The constructor must search for the node that contains the name at the beginning of the line. Call the result `current`. Next, read a preference-name from the input file, search for the Node that matches the preference-name, and call `current.addPref()` to store a pointer to the preference-node in the next available slot of the the current node. Repeat for the N preferences. Then repeat for the remaining people in the file.

Function members.

- `void doMatch( )` Carry out the matching algorithm on the fully-initialized Graph. (Described below.)
- `ostream& print( ostream& out )`: Print out all the people in the Graph. Delegate to `Node::print()`.
- `Node* getNode( string name )`: a private function that searches for a name in the array of Nodes and returns a pointer to it.
- `void addPrefs( istream& in, Node* np )` A private function that reads all the preferences for node `np` and adds them to `np`'s list by calling `Node::addPref()`.

3.2 The Node Class.

A Node class is needed but not an Edge class, because each node contains all the information about its connections. Node must give friendship to Graph. Data members should be:

- name (a length-3 string, defaulting to `""`),
- currentMate (a Node*, initially nullptr),
- the preference-rank of the current mate (initially N),
- the number of bids that have already been made (an int, initially 0),
- a preference list (an array of 10 Node* for the 10 people of the other gender, in order of preference, initially empty). Subscript 0 in the preference array will be the most desirable mate, subscript 9 will be the least desirable.
- One more data member is needed during construction of the preference list: the number of prefs that have already been stored in the preference array.

The Node functions. These functions are executed by both men and women:

- `void addPref ( Node* prefNode )`: Add this to the preference list.
- `print( ostream& )`: print the Node's name, the currentMate, and the names on this Node's preference list. Sample output:

```
    Xin = Ann :  Bet Dee Ann Kat Cin Ida Flo Eva Gay Jan
```

These functions are executed only by women:

- `int getRank ( Node* bidder )`: search for the subscript in the woman's own preference list of the man making the current bid.
- `void befriend ( Node* bidder )`: Let the man know you have accepted his bid. Send him your Node* to save in his Node.
- `void unfriend ( )`: Tell your currentMate that you don't want him any more. He needs to "let it go", that is, erase his pointer to you. He now has no mate. The Graph will put him back on the unmated list.
- `Node* receiveBid ( Node* bidder )`. The woman accepts or rejects:
  - She accepts if she has no current mate. Return nullptr meaning that nobody was rejected.
  - She accepts if she likes the bidder better than the mate she has. She `befriends()` the bidder. Return the Node* for the former mate.
  - She rejects the bid if she likes the mate she has better than the bidder. Return the Node* for the bidder.

This function is executed only by men:

- `int makeBid`. To make a bid, the man call the makeBid() function with a pointer to himself as the parameter. The object used to execute the call is the woman he wants to bid for. Example: If Tom wants to bid for Flo, the second person on his list, he calls —tt preferenceList(2)-¿makeBid(self); . He must also increment his bid counter.