CS51 Final Project Report

A Collection of Matching Algorithms: Bipartite and Stable

Samuel Cheng, John Gee, Angela Ma, Emily Wang

May 1, 2015

Overview

We implemented solutions to several bipartite matching problems, including the general hospital resident problem as an extension to the stable marriage problem, the stable roommates problem, maximum cardinality bipartite matching, max weight bipartite matching, and min cost max flow matching.

Planning

Our original planning for this project was surprisingly quite accurate. While we didn't fulfill much in the way of the front end, we completed not only the algorithms we expected to, but also two more difficult algorithms that we had originally intended to be extensions to our set of problems. We decided that we would do more than what we had set out to do after we met our first milestone early. This decision turned out to also have its negative consequences, as we'll mention later.

Design and implementation

A key choice in design for this project was sharing code for the similar stable marriage and hospital resident problems. We recognized that the main difference between the problems was that hospitals can have a capacity different from one, but that men and women in the stable marriage problem each have a "capacity" of 1 (monogamy). By having the user indicate "hospital resident" or "stable marriage" at the top of the preferences.csv file, we can explicitly set all capacities to 1 only if given data for the stable marriage problem and efficiently share code for the stable marriage and hospital resident problems. After learning how the Gale-Shapley algorithm solves the stable marriage problem, we extended the algorithm to more generally solve the hospital resident problem.

For the stable roommates problem, we read a paper from the Cornell University Library titled "Stable Roommates Problems with Random Preferences" by Stephan Mertens, which described

an efficient version of the Irving's algorithm for solving the stable roommates problem. We read and understood the descriptions and pseudocode of the algorithm on phases, rotations, and cycles, and implemented Irving's algorithm in Python using lists and dictionaries. Following Merten's model, we have PhaseI and PhaseII functions, as well as a helper function seekCycle that aids PhaseII in rotations.

Reflection

While we expected difficulty in implementing algorithms based on theory from papers and textbooks, most of our troubles came with putting together each person's work at the end. We didn't have much trouble learning a new language; we actually found that Python was very easy to learn and use, given our past experience with C and Java. In hindsight, while our implementations of algorithms were acceptable, our design choices were not very good, because they did not allow for the most reduced code, and did not produce good interfaces for the user to interact with. We did not have enough time to work on this part of the project because we near-sightedly decided to try to implement two extra algorithms once we found that we had reached our first goal early. Instead, it may have been better to work with what we had and spent much more time cleaning up and unifying our code. This showed us just how important it is to be very careful when writing code, trying as much as possible to write cleanly and beautifully along the way.

Advice for future students

The most important thing we learned was not so much about implementing algorithms, but rather teamwork and how to collaborate for a programming project. Because we split the work among the four of us by the family of problems the algorithms solve, our code turned out to clearly be written by several different authors. When we got together at the end to link our algorithms to a master control script, we found that each of our implementations of the algorithms and their design was quite different, and while we tried to modularize within our assigned set of algorithms, it was hard to do it as a whole. While the setup of our control script does allow for some independence in modularization, it was not the best practice in design, and our code at the end turned out a bit messy, with no clear hierarchy between the different files.

We should have communicated much more as a group, and planned much more clearly what the greatest common factor for our code would be. Especially with CS51, the goal is to write beautiful, modular, reusable code, which should have been our first priority after attaining an acceptable amount of algorithmic rigor.

Teamwork

Angela Ma worked on solving the hospital residents and stable marriages problem using a version of the Gale-Shapley Algorithm, writing the master control script (menu.py), formatting and taking in input from preferences.csv, and writing and debugging the stable roommates problem.

Emily Wang worked on the bulk of the stable roommates problem (reading research, writing code, debugging), organizing the video, writing the script, and making the video (pulling together all audio files, photos, and video segments).

Samuel Cheng researched and implemented the graph algorithms Edmond-Karp, Hopcroft-Karp, and Cycle-Canceling.

John Gee researched and implemented the blossom algorithm for general graphs (as opposed to just bipartite graphs).

Video

https://www.youtube.com/watch?v=BonDiqActjc

Original Specs

Spec 1: https://tinyurl.com/naxatas

Spec 2: https://tinyurl.com/qx8zckq