CPSC 320 Notes, The Stable Matching Problem, Part I

For this and all worksheets in the course, work in groups of three or four. To start with, form a group and get to know each other by telling the story of your best experience in a course.

Once you're done with this activity, we'll explore the stable matching problem (SMP). Following the historical literature, the text formulates the problem in terms of marriages between men and women. We'll avoid the gender binaries inherent in that literature and use employers and job applicants instead. We focus on a scenario in which each employer has exactly one full-time position. Imagine for example the task faced by UBC's co-op office each semester, which seeks to match hundreds of student applicants to employer internships. To keep the problem as simple as possible for now, assume (perhaps unrealistically) that every applicant has a full ranking of employers and vice versa.

1 Trivial and Small Instances

1. Write down all the **trivial** instances of SMP. We think of an instance as "trivial" roughly if its solution requires no real reasoning about the problem.

2. Write down two **small** instances of SMP. One should have three employers and three applicants:

   The other can be even smaller, but not trivial:

People tend to stop at the end of the page instead of going on. **GO ON UNTIL YOU'RE STUCK!**

2 Represent the Problem

1. What quantities or data (such as numbers, sets, lists, etc.) matter in representing an instance of SMP? Give them short, usable names.

2. For one of your trivial and one of your small instances, write down which quantities or data correspond with your names.

3. Using your names, write down what a valid instance looks like, as input parameters to an algorithm or procedure that solves the problem:

```
procedure SMP-Algorithm(                                        )
...
end procedure
```

(**Still and always** go to the next page if you finish early! There are challenge problems at the end.)

3 Represent the Solution

1. What are the quantities or data that matter in a solution to the problem? Give them names.

2. Using your names, describe what makes a solution **valid**.

3. Describe what you think makes a solution **good**. Develop notation to help with your description.

4. For one of your trivial and one of your small instances, write down one or more valid solutions.

5. Draw at least one good solution for your instance of size three. (We would normally ask you to draw an instance as well, but SMP isn't an especially graphical problem.)

4 Similar Problems

As the course goes on, we'll have more and more problems we can compare against, but you've already learned some. So, give at least one problem that seems related to SMP in terms of its surface features ("story"), instance or solution structure, or representation:

5 Brute Force?

You should usually start on any algorithmic problem by using "brute force": generate all possible **valid** solutions and test each one to see if it is, in fact, a **good** solution.

1. A valid SMP solution takes the form of a perfect matching: a pairing of each employer with exactly one applicant. We'll call a perfect matching a "valid" (but not necessarily good) solution.

   It's more difficult than the usual brute force algorithm to produce all possible perfect matchings; instead, we'll count how many there are. Imagine that the applicants are arranged in some order. How many different ways we can arrange (permute) the employers next to them? This is the number of valid solutions.

2. Once we have a valid solution, we must test whether it's a good solution.

   A valid solution, i.e., a perfect matching, is a good solution if it has no instabilities. Design a (brute force!) algorithm that—given an instance of SMP and a valid solution (perfect matching)—determines whether that perfect matching contains an instability. As always, it helps to **give a name** to your algorithm as well as its input parameters, *especially* if your algorithm is recursive. For brute force: generate each potential instability and then test whether it actually is an instability. Be careful: a potential instability is an (employer, applicant) pair who would rather be matched with each other than their actual matches, not an already-matched pair.

3. Exactly or asymptotically, how long does your brute force algorithm take? (What named quantity pertaining to an instance is most useful in expressing this?)

4. Brute force would generate each valid solution and then test whether it's good. Will brute force be efficient for this problem on instances of interest to us?

6 Promising Approach

Describe—in as much detail as you can—an approach that looks more efficient than brute force.

7 Challenge Your Approach

1. **Carefully** run your algorithm on one or two of your instances above. (Don't skip steps or make assumptions; you're debugging!) Is your algorithm correct on these instances?

2. Design an instance of size three on which your algorithm is as slow as possible, or incorrect.

3. Can you generalize from your example of part 3, to say what is the running time of your algorithm?

8 Repeat!

Hopefully, we've already bounced back and forth between these steps in today's worksheet. You usually *will* have to. Especially repeat the steps where you generate instances and challenge your approach(es).

9 Challenge Problems

These are just for fun, some are easier than others.

1. Design an algorithm to generate each possible perfect matching between n employers and n applicants. (As always, it will help tremendously to start by giving your algorithm and its parameters names! Your algorithm will almost certainly be recursive.)

2. A "local search" algorithm might pick a matching and then "repair" instabilities one at a time by matching the pair causing the instability and also matching their partners. Use the smallest possible instance to show how bad this algorithm can get.

3. Design a scalable SMP instance that forces the Gale-Shapley algorithm to take its maximum possible number of iterations. How many is that? (A "scalable instance" is really an algorithm that takes a size and produces an instance of that size, just like the "input" in worst case analysis is scalable to any n.)