CS 6901 (Applied Algorithms) – Lecture 1

Antonina Kolokolova

September 13, 2016

In this course we will review, and study more deeply the main algorithm design paradigms and techniques. In particular, we will look at greedy algorithms, dynamic programming, backtracking; we will also consider such famous techniques as Fast Fourier Transform (FFT) and Linear Programming. The goal of this is to get practice converting general descriptions of problems into clean representations such as graphs, and using these representations to design efficient algorithms for the original problem, using a variety of techniques in algorithms and data structures.

Sometimes, the problems do not need to be solved in their full generality, as the input data might have some structure that might help or hinder specific algorithms. For example, MergeSort runs in $O(n \log n)$ time on any input, including an already sorted or nearly sorted one, whereas bubble sort, with its $O(n^2)$ running time in worst case, works in linear (that is, $O(n)$) time when there are only constantly many elements out of order. Throughout the course, we will consider variants of the problems under restrictions on the type of inputs, and analyze how that affects algorithm design.

The assignments will focus mainly on design rather than programming. However, for maybe one problem per assignment I will ask you to implement an algorithm and run experiments to see how it performs.

We will start with a clean, yet quite applied problem of Stable Marriage. Just recently, in 2012, the Nobel Prize in Economics was given to Alvin Roth and Lloyd Shapley, with Shapley's contribution being primarily the algorithm for the very problem we are going to talk about now.

1 Stable matching problem

Imagine a situation where there are two types of entities (say organizations and applicants, for example interns applying to companies for their internships or students applying to professors to be their supervisors for graduate studies). Each of them has a ranking on entities of the opposite type: for example, a student would have a ranking of professors with whom he would prefer more to work, and each professor, in turn, would have a ranking of students that applied to her.) Now, the question is to find the "best" way of matching the two: matching professors with students, companies with interns and so on.

First, let's simplify the problem. Suppose there is exactly the same number of both types, and each of them has to be matched with exactly one of the other type. The classical toy problem description talks about n men and n women, each intending to marry one person of the opposite gender (hence the "stable marriage problem" name). In graph representation, we have a bipartite graph with n vertices on both sides, and looking for a perfect matching in it.

But finding a perfect matching is not quite enough for this problem, because we also want to take into account the rankings. Also, we have not yet defined what it means for the matching to be "good". Would it mean that every student (man) gets his best choice? Or would best be the best for the other side? When economists Gale and Shapley considered this problem, their main concern was not making everybody happy, but, rather, stability: it would have to be a matching such that nobody would like to change their assignment to another person, and have that other person accept the switch. That is, the instability is when there is a student s_i and a professor p_j which both prefer each other to their current assignments. In the language of "stable marriage", this is described as a a married man and a married woman who both prefer each other to their spouses – not a stable situation.

Example 1 Suppose there are two professors p_1 and p_2, and two students s_1 and s_2. Consider the following possibilities. One is when both s_1 and s_2 would prefer p_1, and both p_1 and p_2 would prefer s_1. In that case, the student everybody likes gets matched with the professor everybody likes; neither of them has an incentive to switch to the other choice, so it is stable. If, on the other hand, s_2 were matched to p_1 and s_1 to p_2, it would not be stable since s_1 and p_1 prefer each other to their pairs.

Example 2 Alternatively, suppose that s_1, p_1, p_2 have the same preferences as before, but s_2 prefers p_2 to p_1. In this case, the matching $\{(s_1, p_2), (s_2, p_1)\}$ is still unstable, since as before s_1 and p_1 prefer each other to their assignments.

Example 3 Now suppose s_1 prefers p_1, s_2 prefers p_2, but p_2 would rather be matched with s_1, and p_1 prefers s_2. In this case, there are two possible matchings that are both stable: either students get their choice (and then they have no desire to switch), or the professors are, similarly, happier with their current match.

Note that for this definition of instability they have to prefer each other, rather than just be unhappy with their current partner. To illustrate that, if there are three people on each side, and the matching is $\{(s_1, p_1), (s_2, p_2), (s_3, p_3)\}$ with preferences of s_1 being $[p_2, p_1, p_3]$ and preferences of p_2 being $[s_3, s_2, s_1]$, then s_1 and p_2 would not form an instability, as p_2, though unhappy with the choice of an assignment, would not be tempted to switch to s_1 as s_1 is even lower in her list.

In the last example there were two possible matchings, both of which were stable. This leads to a question:is it always the case that there would be a stable matching? We will show there is always one by producing an algorithm that is guaranteed to find such matching for any rankings. But the existence of such a matching is far from a trivial problem: a slight modification of this problem called "stable roommates" problem, in which the graph is not bipartite (there is a group of $2n$ people, and each of them can be paired with any of the other $2n - 1$) has instances for which no stable matching exists. Consider, for example, the following list of people with their preferences: $A : [B, C, D]$, $B : [C, A, D]$, $C : [A, B, D]$, $D : [A, B, C]$. Then, the person who gets paired with D would always find somebody from another pair to switch to (e.g., if A were with D, then A would entice C to switch.)