University of Washington
Department of Computer Science and Engineering
January 21, 2021
CSEP 521, Winter 2021

Homework 3, Due Thursday, January 28, 2021

Problem 1 (10 points):

Mark has a large distributed application. He wishes to record certain user transactions, and store them in a database, and needs to have a unique key for each of these transactions. His solution is to generate a random 128-bit key for each transaction. How long can he record data until there is a 1% chance of having two transactions with the same key? For this problem, you can assume the application has 1 billion users generating 1 thousand transactions per user per day.

You may use the following mathematical result for your analysis. Let $n(p; d)$ denote the number of random integers that need to be drawn from $[1, d]$ to obtain a probability of p that at least two numbers are the same. A good approximation for $n(p; d)$ is

$$n(p; d) \approx \sqrt{2d \cdot \ln\left(\frac{1}{1-p}\right)}.$$

Problem 2 (10 points):

Explain why the following two facts about stable matching are true.

a) Let $I = (M, W)$ be an instance of the stable matching problem where m is ranked first on w's preference list, and w is ranked first in m's preference list. For every stable matching for I, (m, w) must be in the matching.

b) Let $I = (M, W)$ be an instance of the stable matching problem. Suppose that the preference lists of all $m \in M$ are identical, so without loss of generality, m_i has the preference list $[w_1, w_2, \ldots, w_n]$. There is a unique solution to this instance.

Problem 3 (10 points):

For this problem, we explore the issue of *truthfulness* in the Gale-Shapley algorithm for Stable Matching. Show that a participant can improve its outcome by lying about its preferences.
Consider $w \in W$. Suppose w prefers m to m', but m and m' are low on w's preference list. Show that it is possible that by switching the order of m and m' on w's preference list, w achieves a better outcome, e.g., is matched with an m'' higher on the preference list than the one if the actual order was used.

Programming Problem 4 (10 points):

Implement the stable matching algorithm.

Make sure that you test your algorithm on small instance sizes, where you are able to check results by hand. A collection of sample instances are provided.

Run your algorithm on the following instance of size $n = 4$. (You can just hard code this as an input into your program.) The preferences for M's are given by the following matrix (where the i-th row in the ordered list of preferences for m_i.

$$\begin{bmatrix} 2 & 1 & 3 & 0 \\ 0 & 1 & 3 & 2 \\ 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \end{bmatrix}$$

and the preferences for the W's are given by the matrix:

$$\begin{bmatrix} 0 & 2 & 1 & 3 \\ 2 & 0 & 3 & 1 \\ 3 & 2 & 1 & 0 \\ 2 & 3 & 1 & 0 \end{bmatrix}$$

Give the resulting matching that is found, along with the list of proposals performed by the algorithm.

Programming Problem 5 (10 points) :

Write an input generator which creates completely random preference lists, so that each M has a random permutation of the W's for preference, and vice-versa. The purpose of this problem is to explore how "good" the algorithm is with respect to M and W. (There is an interesting meta-point relating to algorithm fairness that can be made with this problem.)

We define "goodness" of a match as the position in the preference list. We will number positions from one (not zero as is standard for array indexing.) Note that lower numbers are good. To be precise, suppose m is matched with w. The $mRank$ of m (written $mRank(m)$) is the position of w in m's preference list, and the $wRank$ of w is the position of m in w's preference list. We define the $MRank$ of a matching to be the sum of all of the $mRank(m)$ and the $WRank$ of w to be the sum of all of the $wRank(w)$. If there are n M's (and n W's), we define the $MGoodness$ to be $MRank/n$ and the $WGoodness$ to be $WRank/n$.

As the size of the problem increases - how does the goodness change for M and W? Submit a short write up about how the goodness varies with the input size based on your experiments. How does the run time of the code vary as a function of n? How do your results relate to result from the coupon collector problem? (You may want to time the permutation generation separately from the main stable matching algorithm.) You will probably need to run your algorithm on inputs with n at least 1,000 to get interesting results. (Instructor: my implementation can handle $n = 15,000$ before running out of memory on my desktop.)

Programming Problem 6 (10 points):

Rewrite your stable matching generator, so that you compute stable matchings for very large, random instances. The key idea is to generate the random permutations *incrementally*, so that you only construct the random preferences when they are needed. You will only need to keep track of preferences of pairs that are involved in proposals. Your implementation should simulate the M's and W's having uniformly random permutations.

Write a summary of your results for large n, including both the statistics for rank/goodness, as well as recorded run-time of the program. What does the estimated growth rate in runtime? You should average over several runs for each value of n, (but there is no need to run a large number of times.) You should run your program with values of n up to at least 1,000,000. (Instructor: my implementation got up to $n = 8,000,000$ but failed at $n = 10,000,000$, running out of memory on my desktop.)

In your write up describe your process for incrementally generating random permutations.