Dayoung Lee
DL29923

EE 360C Programming Assignment 1 Report

(a) The fatal flaw with this algorithm is that the result will not be a perfect matching. There may be some advisors with no students after three rounds. The key difference between this algorithm and Gale Shapley algorithm is how the advisors give offers to students. In this algorithm, advisors always make at most three offers. In the Gale Shapley algorithm, one advisor would give an offer to a student, and is not limited with how many offers they can give.

(b) To fix this fatal flaw, I will change the algorithm so there are multiple rounds of offers until all advisors get a student.

(c) Proof by contradiction. Assume that using both GPA and unique location distances to decide the advisor's preference does not give a total ordering. This means that student a and student b are tied. If they are tied, they have the same GPA and same location distance, but this contradicts our statement that each student has a unique location distance, therefore using both GPA and unique location distances to decide the advisor's preference list still gives total ordering.

(d) In the Gale-Shapley algorithm, the person who is proposing gets a better matching than if they were the one being proposed to. In our case, we can have the students "propose", This way, they get the best possible outcome for themselves and should have no incentive to lie.

(e) Each advisor makes a preference list, ranking students higher if they have a higher GPA. If there are ties, then the advisors break the tie by choosing the student closer to them. The students also make preference lists. Initially, all advisors and students are free. There is a list of advisors who are free. While the list of advisors who are free is not empty and that advisor hasn't given an offer to every student, we pick the first advisor and pick a student who is highest on their preference list that they haven't given an offer to yet. If that student is free, then the student takes the advisor's offer and the advisor is no longer free. Else, if the current advisor the student is matched to is higher on their preference list, the new advisor stays free. Otherwise, if the current advisor the student is matched to is lower on their preference list, then the student takes the offer from the new advisor and the old advisor is free. If there are no free advisors left, then return the list of student-advisor matchings.

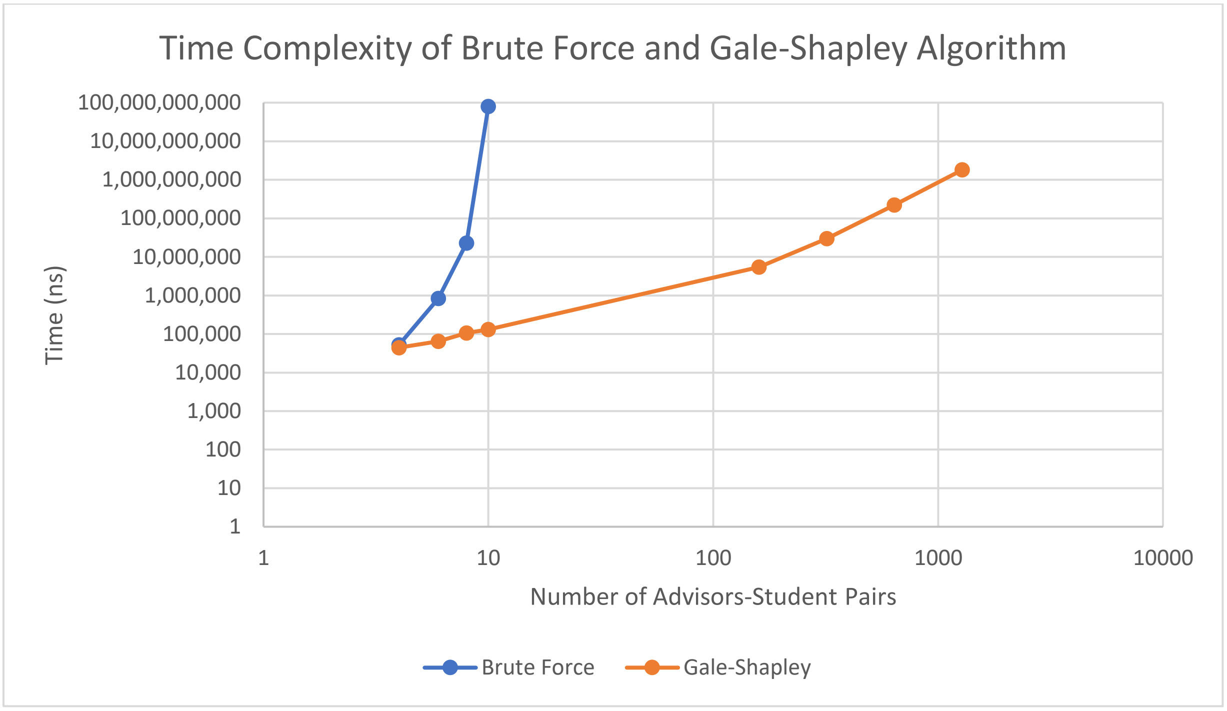
(f) First, we will prove that the algorithm terminates. There are only n^2 total offers that could be given. In each iteration of the loop, one advisor gives an offer to a student, so the number of proposals increases by 1. If the loop runs n^2 times, then there will have been n^2 proposals, and every advisor will have proposed to every student, so the program must terminate. Now we will prove that the algorithm returns the correct result, which is a stable matching. Suppose there is an instability. Then there exist two pairs (*a*, *s*) and (*a'*, *s'*) in the set such that *a* prefers *s'* to *s* and *s'* prefers *a* to *a'*. The final offer that *a* made was to *s*. If *a* did not give an offer to *s'* before, then *s* must be higher than *s'* on *a*'s preference list, but this contradicts our assumption that *a* prefers *s'* to *s*. If *a* did give an offer to *s'* before, then *a* was rejected by *s'* for some other advisor *a''*. Then *a''* = *a'* or *s'* prefers *a'* to *a''*, which means *s'* prefers *a'* to *a*, which contradicts our assumption, therefore our algorithm is correct and results in a stable matching.

(g) First, we make the advisors' preference list. I create an ArrayList of n Student objects, which takes O(n). Then I sort them based on the advisor's preference using Collections.sort, which takes O(nlog(n)). I do that for n advisors, so it takes O(n*(n + nlog(n)), which simplifies to

$O(n^2\log(n))$. Then I make an ArrayList to hold the student matchings and fill it with initial values (n values), which takes O(n). Then I make an ArrayList to hold the free advisors and fill it with initial values (n values), which takes O(n). Then I have the typical Gale-Shapley algorithm, in which n advisors give offers to at most n students before termination, so it takes $O(n^2)$. In the Gale-Shapely algorithm, there were only array indexing, ArrayList set, get, and indexOf functions called, which are all constant time. So the total time it takes for the function to run is $O(n^2\log(n) + n + n + n^2)$, which simplifies to $O(n^2\log(n))$.

(h) Using the brute force algorithm to get all the permutations of matchings takes O(n!). For n students, we need to check the advisors that rank higher in the student's preference list than the advisor the student is currently with, worst case is n advisors. When we check an advisor, we look at which student they are matched with, which takes constant time, and check if that student is higher preference than the other student, which also takes constant time. So verifying a weakly stable marriage for each permutation takes $O(n^2)$. So the overall runtime complexity of the brute force algorithm with verification of stable marriage is $O(n^2*n!)$.

(i) We can see below that the Brute Force algorithm takes much longer to terminate than the Gale-Shapley algorithm.


Time Complexity of Brute Force and Gale-Shapley Algorithm
Time (ns)
Number of Advisors-Student Pairs
Brute Force
Gale-Shapley