# Automated Pairings in a Fraternal Familial Environment

Meisam Amjad, Prasidh Arora, Megan Moore

## 1 Introduction

In many Greek social organizations, there exists a fabricated familial structure based on shared values and mentorship of younger members. Commonly, this relationship is referred to as a “Big and Little” relationship, where the Big is the older mentor. These relationships continue down multiple generations, creating traceable family trees. Prasidh and Megan are Brothers (members) of a professional engineering fraternity where such a system is utilized.

Traditionally, these relationships are created based on a list of rankings provided by the mentors and mentees, where the mentors rank their top choices for mentee, and vise versa. It is then up to a committee to create each match by hand, while trying to preserve as many natural ranking matches as possible. As the head of such a committee, Megan was frustrated by the difficulty of creating the matches by hand. This process typically involves a complex mapping of the proposed rankings, and requires iteratively considering every potential pairing in the rankings. There is often an uneven distribution of member appearance in the rankings, with some potential mentors being ranked many more times than their peers. Additionally, the fraternal pairings allows for the concept of *twins*, which is a pairing of two Littles to one Big. Even with an input size of less than 60, these matches took weeks to do by hand. Megan saw an opportunity for the introduction of automation to increase speed and create a fairer approach to this complex problem. However, even with computer assistance, our proposed algorithm leaves room for human intervention, as these matches are ultimately human decision.

*First,* the algorithm requires accounting for the web of preferences created by the ranking system. Each *child*, representative of the “Little” described above, should provide the committee with an ordered list of their top five (or any number decided by the committee) rankings of their preferred *parents*, representative of the “Big” described above. Once these rankings have been determined, the data structure can be created, as described in *Section 2.*

*Second*, parents and children are assessed into pairs, based on the rules defined in *Section 3.* Upon completion of these pairings, the algorithm returns a list of the parent-child matches, organized by the name of the child. Any unmatched parents or children are matched arbitrarily, which allows the human committee to make final decisions. Essentially, this paper contributes a proposal of the first (to the best of our knowledge) automated matching program for the traditional Greek organization pair structure.

## 2 Literature Review

Matching theory is used when objects of different sets need to be matched with each other with respect to a list of preferences. This type of algorithm was first proposed by David

Gale and Lloyd Shapley, named the "Marriage Algorithm" (MA). It is used it to solve two sided matching problems, and particularly the Stable Marriage problem (SM) [1]. The SM was also shown by Gale and Shapley to be extendable to other problems such as College Admission (CA) and various other many-to-one problems [1, 2]. This theory is now used to solve various types of problems such as apartment search, roommate search, and matching transplants to patients [3]. This type of algorithm is also used in large-scale assignment processes where manual processing is not possible, such as allocating students to universities in countries with centralized college application, which is done based on student and university preferences, and university capacities [4]. The MA is also known as the Gale-Shapely Algorithm [5].

Matching problems can be either bipartite or non-bipartite. Bipartite problems match two separate groups to each other, while non-bipartite problems consist of a single group whose participants are matched to each other [6]. In a general sense, two sided bipartite problems $\Pi = (U, W)$ include two separate sets of participants, U and W. Each participant will have a list of acceptable participants from the other group, which may include ranking in order of preference. In other words, two participants $m \in U$ and $w \in W$ will be a match $M$ if they both list each other under their preferences, and $M$ is compatible with the restrains specific to the problem. A match is considered stable when no two participants prefer each other over their assignment in $M$ [3]. In this review, we will cover a number of important two-sided Bipartite problems and their related algorithms.

*1. The Stable Marriage problem (SM)*

Two important factors that should be considered in a SM mechanism's algorithm are its computational efficiency and strategy [5]. The efficiency is significant due to the high number of factors involved in matching mechanisms, and the used strategy is related to designing the process in a way that it's in the best interest of users to behave truthfully, while reducing misconception [5]. The sets in SM are women $W$ and men $U$ where each $|U| = |W| = n > 0$ is an instance $I$ of SM. In this algorithm, each person $p \in U \cup W$ will list all participant of the opposite sex ranking them based on their preference. In this case, a match would by M as a bijection of $U \times W$, thus in the case of $(m, w) \in M$, $m$ is matched to $w$, and $w$ to $m$. This $M$ will be stable unless $(m, w) \notin M$, and $w$ or $m$ prefer $m$ and $w$ to $M(m)$ and $M(w)$ respectively. Figure 1 shows the Gale-Shapely Algorithm for SM, which returns stable matching. While Gale and Shapely were able to prove each man is matched ($M$) with their most preferred woman [1], McVitie and Wilson showed woman will be matched with their worst partner in this algorithm [7]. $M$ is thus called $Mo$ (man-optimal) in this algorithm, although exchanging the roles can make $M$ in to $Mz$ which is woman optimal. The stable matches formed in $I$ will ultimately form a lattice in time complexity of $O(n^2)$, which has been used in previous studies to find fair stable matches, adjusting happiness in both men and women [7].

*2. The Hospital/Resident problem*

The two groups in this problem instance $I$ are the residents $R$ and hospitals $H$. Similar to SM, each resident $ri$ and $hj$ list their preferred hospitals and residents respectively, in order of their preferences. $ri$ and $hj$ find each other acceptable if they are in each other's lists, while $hj$ also incorporates a capacity of $cj$. In other words, an $M$ in $I$ is if $(ri, hj) \in M$ where $ri$ and $hj$ are said to be assigned to each other. $ri$ is assigned to a hospital unless $ri \in R$ and $M(ri) = \Phi$, where $ri$ will remain unassigned, with a similar case for $hj$; thus, a hospital is undersubscribed or full if $|M(hj)| \leq cj$. Furthermore, $M(ri)$ is an assignment for $M$ in $I$ if $|M(ri)| \leq 1$. $M$ is stable unless is $ri$ unassigned or prefers $hj$ to $M(ri)$, and $hj$ is not at full capacity or prefers $ri$ to at least of member in $M(hj)$. In other words, $ri$ and $hj$ are called stable partners if they are a stable $M$ in $I$. This model was first proposed by Gale and Shapley with time complexity of O($n^2$) [1], and later amended by Irving and Roth for very stable scenario for time complexity of O($n^4$) [7]. Additionally, linear-time algorithms of HR are developed in which they determine the existence of strong matching [8].

*3. The School choice problem (SC)*

This is a problem similar to HR where the two groups are students and schools, documented by Michel Balinski and Tayfun Sönmez [8]. In this problem, the school's preference dominates over the students' preferences which may be due to their policies such as school district, or other preferences including student grades or waiting lists. These different circumstances give rise to different concepts of stability in SC. One of such concepts in the "justified envy" in which a student has a higher priority but another student with lower priority is assigned to their school of choice. SC utilizes the Gale–Shapley student optimal mechanism to overcome issues created by Gale and Shapley's college admissions model [9].

*4. The Stable Roommates problem (SR)*

In the SR problem, an instance $I$ is a set of residents $R$, where each resident $r$ will list their preferred roommates in order of preference. Similar to HR, if $(ri, rj) \in M$, then it is said that $ri$ and $rj$ are matched, and $M$ is stable unless one roommate prefers another resident to their match or if $(ri, rj) \notin M$. The difference of SR with SM is that in some instances, no stable match will be found. Irving has attended to this issue and proposed an algorithm that defines if $I$ in SR will result in a stable $M$ [9].

A good example of a real-life problem solved by MA is the National Resident Matching Program (NRMP) in the US, which assigns resident doctors to available hospital positions, and was used by approximately 43,000 residents for 31, 757 available position in 2017 (NRMP website, 2017). This model is based on the Gale-Shapely Algorithm in solving CA problem [10]. Similarly, schools in Boston and New York assign students to schools based on the student's and school's preferences such as distance, or going to the same school as siblings [11]. Another use of this algorithm is in kidney exchange where patients can swap willing donors in cases where they have donors but are not compatible [12, 13]. Such matching requires information to match patient and donor compatibility, and are also used to match living donors to patients in need instead of only using deceased donors, which increases the number of lives saved [12, 13]. Many countries currently use centralized kidney exchange such as the US, Netherlands, and the UK [12–14]. The impact of researches lead to Alvin Roth and Lloyd Shapley to win the Nobel prize in 2012 for their work and contributions in this area including mentioned resident doctor allocation, student-school matching, and kidney donations [12, 13]. The increased engagement in the use of SM models and the subsequent Nobel Prize awarded in this field demonstrates the increasingly important nature of this algorithm with its use in real life problems.

## 3 Problem Definition

In many Greek social organizations, there exists a fabricated familial structure consisting of a parent and child. Each year, this structure is added to based on approximately five preferences given by the parents and children joining the organization that year. The problem is creating an efficient manner such that each child's top preference is matched with each parent's top preference as best as possible.

We define the first input list as *P*, or a list of *Parent* objects, using a quintuple (String identifier, int rollNum, String name, int availChild, arrayList<Pointer> rankedChildren), where:

- rollNum: identifier of the Parent which is defined by their role in the Fraternity.
- name : string holding the name of the Parent.
- availChild: representing the maximum number of Children, they are willing to be paired with (0, 1, 2).
- rankedChildren : an arrayList of Pointers [0 to 4] to Child objects, where 0 is the most preferred Child and 4 is the lowest ranked.

We also define the second input list as a C, or a list of *Child* objects, using a quadruple (String identifier, int id, String name, arrayList<Pointer> rankedParents), where:

- id: identifier of the Child which is defined as a constant integer where $1 \leq \text{id} \leq |C|$.
- name : string holding the name of the Child.
- rankedParents : an arrayList of Pointers [0 to 4] to Parent objects, where 0 is most the preferred Parent and 4 is the lowest ranked.

With these inputs, we want to produce an output *M* as a set of $\{(P,C)\}$ pairings, organized by *C.id*. Our constraints are such that each *C* is matched with exactly one *P*, and each *P* is paired with 0, 1, or 2 *C*. Assume $p1 \in \{P\}$ and $c1 \in \{C\}$ and are arbitrary *P* and *C* objects. A pair (*P,C*) should be made with the highest rank match possible, with the ideal case that $p_1[0] = c_1[0]$.

At a minimum, a member of $p_1$ should appear in $c_1$, and vise-versa. The number of pairs created by final random match should be minimized. Thus, the following will be true at the end of the algorithm: $\forall c \in C\ \exists p \in P((c,p) \in M)$.

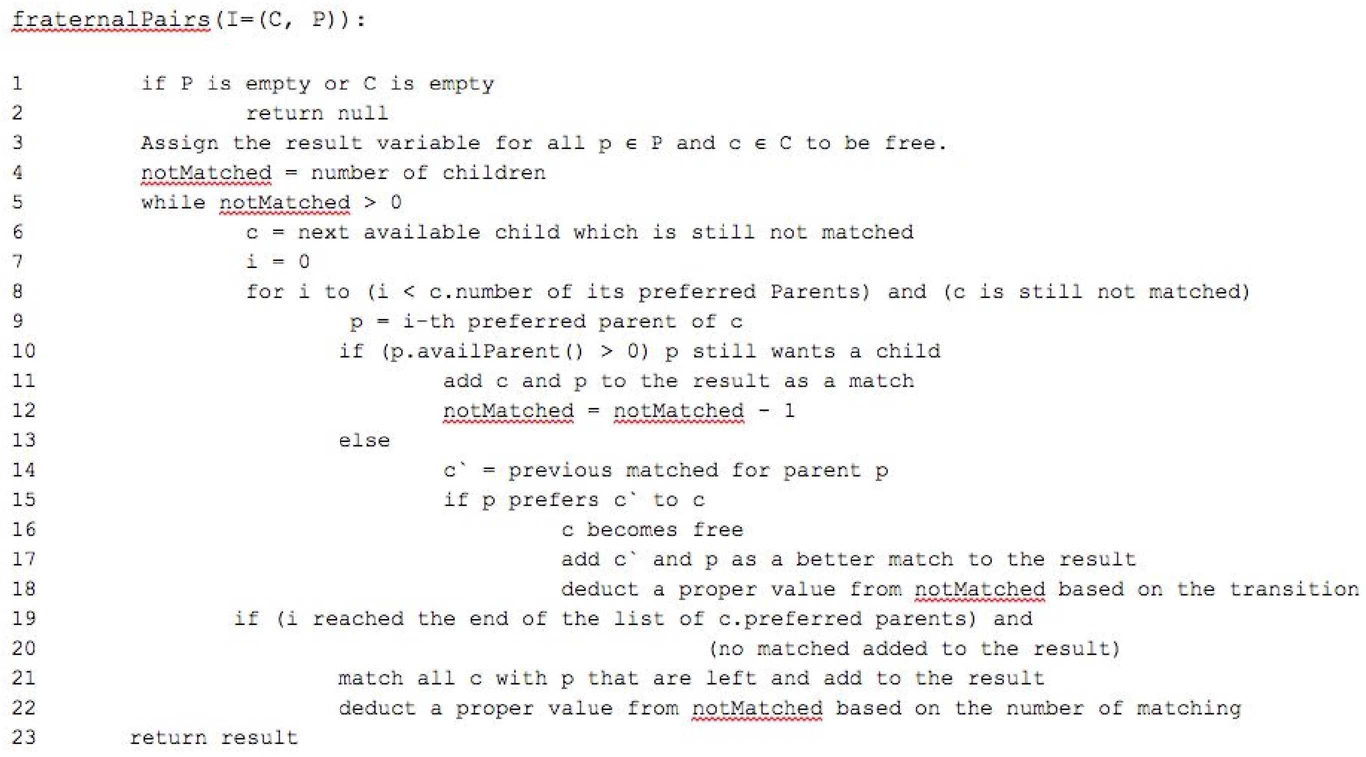
## 4 Proposed Algorithm

### *4.1 Algorithm Introduction*

This algorithm works by considering each Child in the Input set of Children, *C*. So long as there is at least one Child still unmatched, the algorithm continues. A secondary counter variable is set to force the algorithm into it main mode of function for two loops.

This first matching function works as follows: A Child is matched with the first Parent on the Child's preference list that still has availableChild spots remaining. This pair is added to the set of Matches, and the algorithm moves on to the next Child. This continues until the case where a Child's first choice is a parent who has already been matched. At this point, the algorithm looks at the preference list of the disputed parent, and determines which of the two Children are higher prefered by the parent. If the originally matched Child is higher preferred, the Match stays. If the new Child is higher preferred, the Match switches, and the original Child is freed back into the algorithm. This repeats until every Child is matched, or until two complete loops have finished. At the end, all remaining Children are matched with a remaining Parent at random. The algorithm completes by returning the set of matches *M*.

### *4.2 Pseudocode*



```
fraternalPairs(I=(C, P)):

        if P is empty or C is empty
                return null
        Assign the result variable for all p ∊ P and c ∊ C to be free.
        notMatched = number of children
        while notMatched > 0
                c = next available child which is still not matched
                i = 0
                for i to (i < c.number of its preferred Parents) and (c is still not matched)
                        p = i-th preferred parent of c
                        if (p.availParent() > 0) p still wants a child
                                add c and p to the result as a match
                                notMatched = notMatched - 1
                        else
                                c` = previous matched for parent p
                                if p prefers c` to c
                                        c becomes free
                                        add c` and p as a better match to the result
                                        deduct a proper value from notMatched based on the transition
               if (i reached the end of the list of c.preferred parents) and
                                                    (no matched added to the result)
                        match all c with p that are left and add to the result
                        deduct a proper value from notMatched based on the number of matching
       return result
```

*4.3 Example Case*

*See Appendix A.*

*4.4 Cost Analysis*

The worst case time complexity of this algorithm is $O(2c^2)$. However, this is unlikely to occur, as it would require a worst-case input where every match was undone by another match later in the algorithm, on both loops. While this input technically can occur, the complexity of a more normal input would be comfortably under $O(2c^2)$, as a normal input would loop through the child list twice, but would only re-consider a fraction of those Children again. This makes the total time complexity of the algorithm $O(c^2)$.

## 5 Description of Inputs

Example *Child* Input

- Column 1: "C", indicating Child
- Column 2: Child's unique identifying number
- Column 3: Child's anonymized name string
- Column 4+: List of preferred Parents

```
C, 1, SWO, 16, 14,
C, 2, DIW, 17,
C, 3, HYI, 12, 16, 14, 13
C, 4, JFB, 11, 17, 15, 12
C, 5, NGU, 12, 15, 17
C, 6, GBK, 12, 13, 14, 11
C, 7, RKT, 12, 14, 11, 13
C, 8, VJE, 16, 12, 11, 15
C, 9, BHE, 13, 14, 15
C, 10, ALP, 11, 15, 17, 12
```

Example *Parent* Input

- Column 1: "P", indicating Parent
- Column 2: Parent's unique identifying number
- Column 3: Parent's anonymized name string
- Column 4: Parent's count of availableChildren ("2" indicates a willingness to take twins)
- Column 5+: List of preferred Children

```
P, 11, NVE, 2, 5, 8, 9, 2
P, 12, HVY, 2, 6, 7, 1, 10
P, 13, XLH, 1, 4, 10, 1
P, 14, LKV, 2, 7, 9, 6
P, 15, XCE, 1, 10, 7, 3, 6
P, 16, SBG, 1, 5, 3, 8, 1
P, 17, FWX, 1, 6, 7, 4, 9
```

In order to guarantee proper function of our algorithm, there are some constraints the inputs must follow:

- The number of *Children* must be less than or equal to the total number of *availableChildren* between all *Parents*
- Every *Parent* and every *Child* must have at least one preference

The algorithm will still work properly with varied numbers of Parents and Children, and varied number of preferences for each Parent and Child.

## 6 Performance Metrics

The objective of our work was not just create an algorithm to work as our need but, we also wanted to create an algorithm as efficient as possible. Therefore, the new algorithm needed to be tested to prove this hypothesis itself. Since, our main work was mainly using the Gale Shapley algorithm and improvement of that, we decided to do this comparison by creating similar environment and test both algorithm together performance and demonstrate the result. For that, we needed a data input that first can be accepted by Gale Shapley algorithm and as we mentioned earlier our algorithm also can handle the same data as Gale Shapley. Therefore, we started our experiment by producing some random data with same size for Parent's preferences and Children's preferences and we decided to use that data for as our test case. Then, we created fair environment to run both codes in a server using one thread one core and recorded the elapsed time for both codes.

### Experimental Platform:

The experiments were conducted on the following platform:

| Components | Details |
|---|---|
| CPU Model | Intel(R) Xeon(R) CPU X5550 @ 2.67GHz |

|  |  |
|---|---|
| CPU/Core Speed | 2661.000 MHz |
| Main Memory (RAM) size | 24591648 kB |
| Operating system used | Linux mualhpcp01.hpc.muohio.edu 2.6.32-<br>642.el6.x86_64 #1 SMP Tue May 10 17:27:01 UTC 2016 x86_64 x86_64 x86_64<br>GNU/Linux |
| Interconnect type & speed (if applicable) | n/a |
| Was machine dedicated to task (yes/no) | Yes (via a qsub job) |
| Name and version of Java compiler | javac 1.7.0_13 |
| software tools & components (if used) | n/a |

**Running the experiment:**

We created a main method to get the input as below for having our code gets run using either Gale Shapley algorithm or Matching algorithm :
Java main.java {1 or 2} {name of the input file} {number of repetition}
If the user uses number 1 that means the Matching algorithm will be used for running the code, otherwise the code gets run by using Gale Shapley algorithm.

For submitting and running our experiment through the server we created the test.bash file with following context:

```
#!/bin/bash

#PBS -N test
#PBS -l walltime=0:30:00
#PBS -l mem=128MB
#PBS -l nodes=1:ppn=1
#PBS -S /bin/bash
#PBS -j oe

# Change to directory from where PBS job was submitted
cd $PBS_O_WORKDIR
```

```
for threads in 1 1 1 1 1;
do
        # Timing with single thread
        echo "---------------------[ Matching ]---------------------"
        export OMP_NUM_THREADS=${threads}
        /usr/bin/time -v java Group_Project 1 data04.txt 1000000
done

for threads in 1 1 1 1 1;
do
        # Timing with single thread
        echo "---------------------[ Gale Shapley ]---------------------"
        export OMP_NUM_THREADS=${threads}
        /usr/bin/time -v java Group_Project 2 data04.txt 1000000
done
```

This bash file first run the Matching algorithm 5 times by using only 1 thread and 1 node and repeat it 1,000,000 times. Then repeat the whole procedure by using the Gale Shapley algorithm. For submitting our job to the cluster we used the command:

```
$ qsub test.bash
```

**Runtime Observations:**

| observations | Elapsed time for Matching | Elapsed time for Gale Shapley |
|---|---|---|
| 1 | 248.20 sec | 299.43 sec |
| 2 | 253.43 sec | 309.13 sec |
| 3 | 253.01 sec | 308.30 sec |
| 4 | 248.41 sec | 304.28 sec |
| 5 | 251.33 sec | 308.85 sec |
| Average($\mu$) | 250.87 sec | 305.99 |

**7 Result graphs and their analysis and description:**

Now it is time to collect statistics from the Job output and analyze the result.As you can see the average elapsed time of the Matching algorithm was less than Gale Shapley. However, to ensure that numbers are comparable we must calculate 95% confidence interval (CI).
The 95% CI value essentially defines a range around the average that essentially indicates that we are confident that 95% of the time, the observed execution time would lie within the range we report. In other words there is a 5% chance that the numbers we report are wrong. For doing that we used following equations:

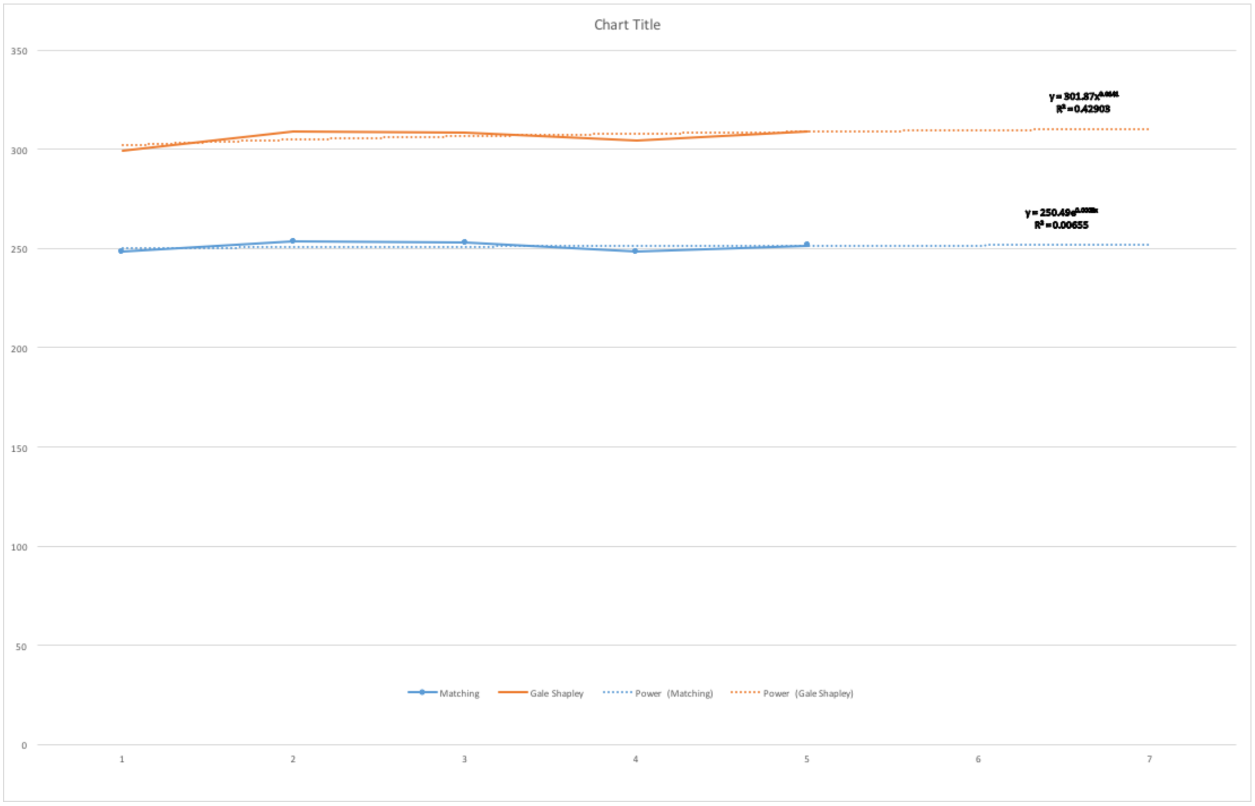
$$\mu = \sum_{1}^{5} t_i / n$$

$$\sigma = \sqrt{\sum_{1}^{n} (t_i - \mu)^2 / n}$$

When n = 5 which is the number of our observation, from the t-distribution table, 95%CI is computed using the formula below:

$$95\%\ CI = (2.776 * \sigma) / \sqrt{n}$$

Time for Matching algorithm ( $\mu \pm 95\% CI$ ) => 250.87 ± 2.75 sec
Fastest time : 248.12 sec
Slowest time: 253. 62 sec
Time for Gale Shapley algorithm ( $\mu \pm 95\% CI$ ) => 305.99 ± 4.62 sec
Fastest time : 301.37 sec
Slowest time: 310.61 sec



Having both run times as O( $n^2$ ) we recorded that the Matching algorithm gained 21.97% or the speed up of 1.2X faster performance over the Gale Shapley algorithm which is an endorsement for proving the Matching algorithm is more efficient and faster.

## 8 Overall performance analysis of the proposed algorithm and how the results corroborate with the theoretical time complexity analysis.

As displayed in the results of a series of tests, our algorithm consistently performed under the O($c^2$) runtime we analyzed, as we had expected. Not only did it perform well under the worst-case complexity, it also showed significant speed improvements over the original matching solution, which was to make matches by hand. Additionally, this algorithm performed faster than, or at worse equivalent to, the Gale-Shapley algorithm when run with equivalent inputs. This is a significant achievement, as our algorithm adds complexity to the Gale-Shapley algorithm without increasing complexity or reducing scalability.

## 9 References

1. Gale D, Shapley LS (1962) College admissions and the stability of marriage. Am Math

Mon 69:9–15

2. Gale D, Shapley LS (2013) College admissions and the stability of marriage. Am Math Mon 120:386–391
3. Manlove DF (2013) Algorithmics of matching under preferences. World Scientific
4. Manlove DF (2014) Algorithmics of matching under preferences. Bull EATCS 1:
5. Farczadi L, Georgiou K, Könemann J (2016) Stable marriage with general preferences. Theory Comput Syst 59:683–699
6. Klaus B, Manlove DF, Rossi F (2016) Matching under preferences
7. McVitie DG, Wilson LB (1971) The stable marriage problem. Commun ACM 14:486–490
8. Roth AE (1986) On the allocation of residents to rural hospitals: a general property of two-sided matching markets. Econom J Econom Soc 425–427
9. Balinski M, Sönmez T (1999) A tale of two mechanisms: student placement. J Econ Theory 84:73–94
10. Roth AE (1984) The evolution of the labor market for medical interns and residents: a case study in game theory. J Polit Econ 92:991–1016
11. Abdulkadiroğlu A, Sönmez T (2003) School choice: A mechanism design approach. Am Econ Rev 93:729–747
12. Roth AE, Sönmez T, Ünver MU (2005) Pairwise kidney exchange. J Econ Theory 125:151–188
13. Roth AE, Sönmez T, Ünver MU (2004) Kidney exchange. Q J Econ 119:457–488
14. Keizer KM, de Klerk M, Haase-Kromwijk B, Weimar W (2005) The Dutch algorithm for allocation in living donor kidney exchange. In: Transplantation proceedings. Elsevier, pp 589–591