2018

Mechanism Design, Matching Theory and the Stable Roommates Problem

Yashaswi Mohanty
Colby College

Mechanism Design, Matching Theory and the Stable Roommates Problem

Presented to the Department of Economics in partial fulfillment of the degree of Bachelor of Arts with Honors in Economics

by

Yashaswi Mohanty

Advisor: Timothy Hubbard
Second Reader: Dale Skrien

Department of Economics
Colby College

Abstract

This thesis consists of two independent albeit related chapters. The first chapter introduces concepts from mechanism design and matching theory, and discusses potential applications of this theory, particularly in relation to dorm allocations in colleges. The second chapter investigates a subset of the dorm allocation problem, namely that of matching roommates. In particular, the paper looks at the probability of solvability of random instances of the stable roommates game under the condition that preferences are not completely random and exogenous but endogenously determined through a dependence on room choice. These probabilities are estimated using Monte-Carlo simulations and then compared with probabilities of solving a completely random instance of the stable roommates game. Our results portray a complex relationship between the two probabilities, suggesting avenues for future research.

Acknowledgements

While an honors thesis is ostensibly an individual endeavour, designed to help a student develop his or her ability to conduct independent research, this project has been far from an independent affair. Although I have spent several solitary hours embedded in academic limbo, struggling to figure out a concept or getting my code to work, many of my breakthroughs have been facilitated by the supporters of this project. While a number of people have contributed to the end product that you see here, a few deserve special mention.

First and foremost, I must thank my advisor Prof. Tim Hubbard, who first guided me towards matching theory and its various applications. He introduced me to the possibility of designing a mechanism for college room allotment. Pondering over this problem, I stumbled across the question that I eventually attempt to answer in this thesis. Tim has been instrumental in helping me tackle broad problems in the goal and vision of the project while simultaneously assisting with me with the intricate details of the modeling process. Perhaps his greatest contribution to this project has been suggesting that I use Kendall's tau when I was struggling to come up with a measure of rank correlation. Without this little insight, my project could have possibly collapsed!

Next, I would like thank Prof. Samara Gunter, whose painstaking efforts to ensure that the entire thesis cohort was making consistent progress on their projects was an important motivating force for getting this thesis across the finish line. From making sure that my presentations were not overly technical and opaque to providing critical feedback for my drafts to reassuring me of my abilities when I most doubted my capability to conduct research, Sam's guidance and support has been indispensable. I am very much indebted to her for this finished product.

Third, I must thank Dale Skrien for agreeing to read this paper in the capacity of a second reader and providing helpful suggestions regarding the Python implementation of Irving's algorithm.

I must also acknowledge the incredible debugging skills of Kyle McDonell who spent six hours neglecting his own honors thesis to fix my (rather poorly written) code. Kyle's modifications to the Python implementation of Irving's algorithm allowed me to finally generate the results I had been looking for.

Finally, I would like to acknowledge my fellow honors students, who have shared this incredible experience with me and have experienced the peaks and nadirs of this rather tumultuous journey. This process wouldn't have been half as fun without them.

Contents

Chapter 1

Matching and College Housing Allotment

1.1 An Overview of Matching

While traditional economic theory is concerned with markets where prices play a key role in resource allocation, there exist markets in which there are no explicit prices that determine how agents are "matched" to these resources. This is particularly the case when the resources in question are other agents. Gale and Shapley (1962) provide the classic example of the marriage market, where men and women are matched to each other given that each group has preferences over the other. In such a market, there can be no explicit prices, although agents value other agents differently based on their preferences.

Although a two-sided matching problem such as marriage serves as an excellent example of a market without prices, we can also have a one-sided matching market, where agents are matched to objects. Shapley and Scarf (1974) provide a model for such a market where agents are endowed with an indivisible good such as a house and have preferences over all the houses that are assigned to agents. In such a market, agents may have incentives to exchange houses in order to maximize utility. A real life example of such a situation occurs

on college campuses, where students change dorm rooms at the end of the year, but dorm rooms are usually not bought or sold with money.

The key features which characterize a matching market include the lack of a pricing mechanism to clear the markets, the presence of heterogeneous agents and the problem of allocating indivisible resources to these agents. Economists who study these markets are often interested in designing an algorithm or a "mechanism" that generates allocations which satisfy certain important properties. This can be particularly challenging because of numerous theoretical constraints that prevent a mechanism from having all the desirable properties (see Roth and Sotomayor (1992), and Roth (1982)). We shall discuss the nature of some of these properties in section 1.3.

1.2 Applications of Matching Theory

Matching is not just a theoretical curiosity; while the game theoretical formulations of matching problems provide unique mathematical challenges and insights, economists are interested in the applications of the theory to market design.

One of the first applications for matching was provided by Roth (1984) who looked into the problem of matching medical students to hospitals for internships. This problem can be classified as a **two-sided**, **many-to-one** matching problem. The problem is two-sided because the agents can be split into distinct groups, in this case hospitals and medical students. It is also two-sided in the sense that both hospitals and doctors have preferences over each other. Since many doctors are matched to a single hospital, the problem is characterized as many-to-one matching problem. In the paper, Roth provided a history of the market and its idiosyncrasies while also presenting a game theoretic model that captures the essential elements of the market. Roth and Peranson (1999) actually presented a new mechanism for matching physicians to hospitals which was adopted by the National Resident Matching Program (NRMP).

Matching theory was also successfully applied in the context of school choice. In many U.S. school districts, children (and their parents) are often asked to submit the preferences over which school they would like to attend. Schools too have preferences over children, usually based on factors such as proximity, whether the child has siblings at the school, and his or her grades. Abdulkadiroğlu and Sönmez (2003) also formalized this problem as a two-sided, many-to-one matching problem. They they suggested that districts use a variant of the matching algorithm presented by Gale and Shapley (1962). New York and Boston are two cities which adopted these recommendations and recent empirical evidence suggests that the new mechanisms are effective in increasing the number of "good" matches.

1.3 College Housing Allotment and Stable Roommates

One interesting application of matching theory is in the context of assigning college students to dorm rooms. Interestingly, matching theory is applicable at two levels in this context. Firstly, the problem of students finding roommates is a two-sided, many-to-many (or one-to-one) matching problem: depending on the type of rooms available, students may be looking for many or just a single roommate. Ensuring optimal roommate matching is key to student happiness while also making sure that the administration is not burdened with mid-year room reassignments.

Next, the student groups need to be assigned to dormitories. This is a one-sided matching problem as the students have preferences over rooms but the rooms do not have preferences over students. Most colleges have a process by which students get to pick rooms; these systems are usually designed to maximize fairness or some other objective. Abdulkadiroğlu and Sönmez (1999) look at some of the mechanisms utilized by colleges and evaluate them on their performance. Economists use a few typical metrics to evaluate such mechanisms. One is **incentive compatibility** or **strategyproofness**, which asks whether students have any incentive to misrepresent their preferences in order to "game" the system. The Colby

College housing allotment system, for example, is famously not incentive-compatible. The Colby dorms are stratified into various mutually exclusive housing types such as substance-free housing, quiet housing and traditional housing. Once students pick which type of housing they want, they can select rooms only from that housing type. However, students who do not intend to be quiet or substance free often opt into these specialty housing programs in order to get better rooms and flout the community rules. A mechanism designer could fix this problem by designing a system in which students have no incentive to lie about their preferences.

Another metric for evaluating mechanisms is **Pareto optimality** or **efficiency**. Matches are not efficient when there exists another assignment of students to rooms where no student is worse off than in the current assignment and at least one student is strictly better off. In this situation, students may have incentive to exchange rooms until an efficient allocation is obtained. In order to avoid the administrative cost of supporting mid-year room re-allocations, colleges should strive to generate Pareto efficient room allocations.

Efficiency is key in matching roommates to roommates as well. In the second chapter of this thesis, we examine a stronger version of Pareto efficiency in a specialized case of the roommate matching problem. In such a specialized (and somewhat unrealistic) setting, students can room with only one person and have preferences over all the other students who are looking for roommates. In this context, we can use **stability** as a metric to evaluate the quality of a matching. A matching (pairing) is *unstable* if two students who are not rooming with each other prefer each other as roommates to their actual roommates. Consequently, a matching is stable if no such two students exist. Abraham and Manlove (2004) proved that any stable matching is also Pareto optimal; the converse is not necessarily true. Gale and Shapley (1962) demonstrated that not every instance of the *stable roommates problem* admits a stable solution. In other words, for certain preference structures, no stable matching can be found.

This fact naturally leads to some questions. What kind of preferences lead to a stable

matching? What is the proportion of stable roommates "games" that admit a solution? Or, equivalently, if students have arbitrary unspecified preferences over one another, what is the probability that a stable matching exists? This last question is important because it lends some insight into the nature of a roommate pairing. How likely are such pairings to fall apart?

In an abstract mathematical setting, all preference structures are considered equally likely to occur. In real life however, students' preferences over one another are guided by a variety of factors, and thus are not completely random. In particular, certain preference structures are more likely to occur than others. The goal of this thesis is to investigate whether the probability of finding a stable solution to the roommates game increases under more realistic settings.

In the next chapter, we formalize the notion of a stable roommates game, restate the research question rigorously and lay out the methods and results of our analysis.

Chapter 2

Stable Roommates with Endogenous Preferences

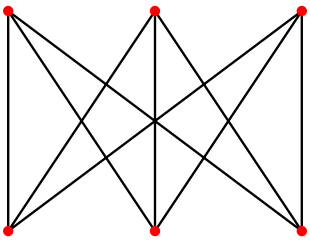
2.1 Introduction

The stable marriage problem is a classic problem in matching theory which seeks to solve the problem of pairing men and women in a marriage market where each group has complete and strict preferences over the other. Formally, the set of agents $I = I_M \cup I_W$ is partitioned such that $|I_M| = |I_W|$ and for any $m \in I_M$ there exists a preference relation $\succ_m$ such that for any $w_1, w_2 \in I_W : w_1 \succ_m w_2$ or $w_2 \succ_m w_1$. The preference relation for any $w \in I_W$ is analogously defined. A solution to an instance of the stable marriage problem involves finding a stable matching. This is a matching in which no pair of men and women who are not matched to each other prefer each other to their partners under the matching.

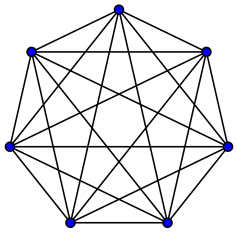
Gale and Shapley (1962) proved that any instance of the stable marriage problem (that is, a game with any set of complete and strict preferences) admits at least one stable matching and supplied an algorithm to find such a matching. In the same paper they discussed the stable roommates problem, a generalization of the marriage problem. In this problem agents are not divided into two groups like in the marriage problem. Instead agents have strict

preferences over every other agent.

The differences between these two problems can be understood in graph theoretic terms. Figures 2.1a and 2.1b depict graph representations of the stable marriage and stable roommates problems respectively. Notice how in the first figure, vertices can be partitioned into two sets such that edges from a vertex in one set can only map to vertices in the other set. This *bipartite* structure captures the nature of the stable marriage problem, where the vertices represent agents and the edges code preferences. In figure 2.1b, there is no such partitioning. In other words, a vertex is mapped to all other vertices through edges, representing an agent's preferences over all other agents. This captures the essence of the stable roommates problem. A real-world example of this problem occurs when college students attempt to find roommates to live in college dormitories.



(a) Complete bipartite graph representing the stable marriage problem. The edges code preferences and the vertices represent agents.



(b) Complete graph representing stable roommates. The edges code preferences and the vertices represent agents.

Figure 2.1: Graphic theoretic interpretation of the stable marriage and roommate problems.

Unlike the stable marriage problem, the stable roommates problem does not automatically admit a solution. We construct a simple example below to demonstrate that solutions may not exist for instances of the problem. To make the analysis of the example more tractable, we can define the notion of a matching and a stable matching formally.

Definition 2.1.1. A matching $\mu : I \to I$ is a symmetric 2-cycle permutation such that if $\mu(m) = w$ then $\mu(w) = m$.

Definition 2.1.2. A matching μ is said to be *unstable* if and only if there exist $i_1, i_2 \in I$ such that $\mu(i_1) \neq i_2$ and $i_2 \succ_{i_1} \mu(i_1)$ and $i_1 \succ_{i_2} \mu(i_2)$. A matching μ is said to be *stable* if it

is not unstable.

Example 2.1.1. (Gale-Shapley) Let A,B,C and D be agents with the following preferences-

A	B	C	D
B	C	A	A
C	A	B	B
D	D	D	C

Proposition 2.1.1. There is no stable matching for this problem instance.

Proof. Let μ be one of the $\binom{4}{2}$ possible matchings. For any $X \in \{A, B, C\} \setminus \mu(D) : X \succ_{\mu(D)} D$. Since no two members rank each other the highest $\exists Y \in \{A, B, C\} \setminus \mu(D)$ $s.t$ $\mu(D) \succ_Y \mu(Y)$. ■

Irving (1985) provided an algorithm that takes in agents' preferences as an input and returns a stable matching if one exists. If a stable matching does not exist then the algorithm informs the user that no stable solutions exist. In this paper, we use this algorithm in order to determine whether a given instance of the stable roommates problem has a solution.

2.1.1 Solvability of stable roommates instances

The possibility of unsolvable instances of the roommates problem raises many questions but two are particularly pertinent. Qualitatively, we would like to know what features of a problem instance lead to it being solvable. Quantitatively, we would like to know the proportion of problem instances that are solvable for any n-person game. The former question was resolved by Tan (1991) who established necessary and sufficient conditions for the solvability of a stable roommates instance. In fact Chung (2000) provides a sufficient condition for the existence of stable matchings when agents have weak (not necessarily strict) preferences.

The latter question is still unresolved although non-trivial progress has been made in that direction. In particular, Pittel (1993) proved that $P_n \geq \sqrt{4e^3/\pi n}$, where P_n denotes the

probability that an n-person stable roommates instance is solvable. In the same paper, Pittel also provided an analytical expression for P_n, a version of which is presented in section 2.2.4. Further, Pittel and Irving (1994) provided a loose upper bound for the limiting behavior of P_n by establishing that $\lim_{n\to\infty} P_n \leq \sqrt{e}/2$. Mertens (2005) supplied some empirical evidence for conjectures about the asymptotic behavior of P_n.

2.1.2 Partially endogenous preferences

Research on the stable roommates problem traditionally treats preferences as exogenous. While such a simplification is important to understand the theoretical properties of the stable roommates game, particularly with respect to solvability, it is hardly an assumption that holds in practical applications. For example, if we consider the case of an actual room selection problem, we know that roommate choice is inextricable from the choice of room itself. In particular, preferences over roommates are generally not independent of preferences over rooms and ceteris paribus one would expect two agents with similar room rankings to rank each other higher on their preference lists.[1]

Pittel (1993) asserts that "stable partners are very likely to be relatively close to the tops of each other's preference lists". Given this observation, it is highly plausible that the probability of finding a stable solution for any given problem with partially endogenous preferences is higher than the probability of finding a stable solution for a random instance of stable roommates with exogenous preferences. Note we use the phrase *partially endogenous* because the roommate preferences have both a stochastic and a deterministic component; the similarity of dorm room preferences between agents acts like a shock to otherwise idiosyncratic roommate preferences.

The goal of this chapter is to consider statistical evidence for the claim that the probability of finding a stable solution for a given problem instance is higher in the case where we have endogenous preferences vis-a-vis exogenous preferences.

[1]Notice that the dependence goes both ways i.e. room rankings affect roommate preferences and vice-versa. For simplicity, we assume room rankings to be exogenous.

2.2 Methods

2.2.1 Setting up the model.

For very small instances of the roommates problem, the proportion of stable instances can be calculated using exhaustive enumeration or through a multidimensional integral formulation (see section 2.2.4). However, since the number of possible preference sets for an n-person game is given by $[(n-1)!]^n$, even for modest values of n, exhaustive enumeration becomes practically impossible. Instead, Monte Carlo simulations are used to estimate the probability P_n of an n-person game instance having a stable solution.

In order to generate preference lists for the agents, we can use an $n \times n$ matrix X of independently and uniformly distributed random variables on $[0, 1]$. For this to make sense, we must first formalize the set of agents by coding them into natural numbers that correspond with the rows and columns of the matrix X.

Definition 2.2.1. The set of agents is given by $I = \{1, 2, \dots n\}$.

Next we can define the preference relation $\succ_i$ as follows

Definition 2.2.2. $\forall j_1, j_2 \in I \setminus \{i\} : j_1 \succ_i j_2 \iff X_{i,j_1} > X_{i,j_2}$.

This process yields a complete and strict preference list for each agent $i \in I$. We provide an example to illustrate this fact.

Example 2.2.1. $I = \{1, 2, 3, 4\}$.

$$X = \begin{bmatrix} 0.17 & 0.22 & 0.75 & 0.20 \\ 0.95 & 0.74 & 0.57 & 0.26 \\ 0.60 & 0.87 & 0.51 & 0.45 \\ 0.61 & 0.25 & 0.95 & 0.08 \end{bmatrix}$$

Note that the diagonal entries are irrelevant and so, using definition 2.2.2, we can define the preference relation for each agent $i \in I$.

1: $3 \succ_1 2 \succ_1 4$

2: $1 \succ_2 3 \succ_2 4$

3: $2 \succ_3 1 \succ_3 4$

4: $3 \succ_4 1 \succ_4 2$

As one can see, the preference list above is complete and strict. Interestingly, this preference list is similar to the one found in example 2.1.1 and thus does not admit a stable solution.

2.2.2 Adding exogenous room preferences

In order to construct a model with endogenous roommate preferences, we first need to add exogenous room preferences for every agent in the game. Let I be the set of agents in the model.

Definition 2.2.3. Let H be the set of *rooms* such that $|H| = |I|/2$.[2]

Just like in section 2.2.1, we can construct preferences over houses using a matrix of independently and uniformly distributed random variables, except now the dimensions of the matrix are $n \times n/2$. Given that we now have two sets of preferences for each agent we need to introduce some new notation-

Definition 2.2.4. Let $\succ_{i,h}$, $i \in I, h \in H$ denote the preference relation of agent i over rooms.

Thus $h_1 \succ_{i,h} h_2$ denotes that agent i prefers room h_1 to room h_2

2.2.3 Endogenizing roommate preferences

In order to partially endogenize the roommate preferences of the agents in our model using room choice, we need to find a way to measure the similarity between room rankings that the agents reveal. One good measure of ordinal association is Kendall's Tau (see Abdi (2007)

[2] I has even cardinality.

or Appendix A for details), a function that takes two rankings and returns the ordinal correlation between the two. A correlation of 1 indicates that the rankings are perfectly aligned; in our case, since the rooms ranked are the same for all agents, a correlation of 1 indicates that the rankings are identical. A correlation of -1 indicates that the ranking is reversed.

This correlation can be used to shock the model in a way which allows us to generate preferences which incorporate room choice similarity. The best way to demonstrate how we achieve this in our model is through an example.

Example 2.2.2. Let $I = \{1, 2, 3, 4\}$, $H = \{h_1, h_2\}$.

We are given the following rooming preferences-

1: $h_1 \succ_{1,h} h_2$

2: $h_2 \succ_{2,h} h_1$

3: $h_2 \succ_{3,h} h_1$

4: $h_1 \succ_{4,h} h_2$

Let $\tau_{i,j}$ indicate the Kendall ordinal association between the room preferences of agents i and j. Since we only have two rooms, the number of possible permutations of room preferences is just two. This means that the only possible values for τ are 1 or -1.

Table 2.1: Ordinal association between room preference lists.

Kendall's τ			
$\tau_{1,2}$	-1	$\tau_{2,3}$	1
$\tau_{1,3}$	-1	$\tau_{2,4}$	-1
$\tau_{1,4}$	1	$\tau_{3,4}$	-1

Now we can use the same $n \times n$ matrix X of uniform random variables on (0,1) as seen in example 2.2.1 and add the τ values as shocks to this matrix in the following sense.

We have the original matrix-

$$X = \begin{bmatrix} 0.17 & 0.22 & 0.75 & 0.20 \\ 0.95 & 0.74 & 0.57 & 0.26 \\ 0.60 & 0.87 & 0.51 & 0.45 \\ 0.61 & 0.25 & 0.95 & 0.08 \end{bmatrix}$$

Next we construct an augmented matrix $\bar{X}$ in which $\bar{X}_{i,j} = X_{i,j} + \tau_{i,j}$ wherever $\tau_{i,j}$ is defined – i.e. the off diagonal values. Remember that since Kendall's Tau is a correlation coefficient, $\tau_{i,j} = \tau_{j,i}$ and thus this addition operation is performed symmetrically on the matrix X.

$$\bar{X} = \begin{bmatrix} 0.17 & 0.22 - 1 & 0.75 - 1 & 0.20 + 1 \\ 0.95 - 1 & 0.74 & 0.57 + 1 & 0.26 - 1 \\ 0.60 - 1 & 0.87 + 1 & 0.51 & 0.45 - 1 \\ 0.61 + 1 & 0.25 - 1 & 0.95 - 1 & 0.08 \end{bmatrix} = \begin{bmatrix} 0.17 & -0.78 & -0.25 & 1.20 \\ -0.05 & 0.74 & 1.57 & -0.74 \\ -0.40 & 1.87 & 0.51 & -0.55 \\ 1.61 & -0.75 & -0.05 & 0.08 \end{bmatrix}$$

Now we use the augmented matrix to construct the preferences:

1: $4 \succ_1 3 \succ_1 2$

2: $3 \succ_2 1 \succ_2 4$

3: $2 \succ_3 1 \succ_3 4$

4: $1 \succ_4 3 \succ_4 2$

It is clear here that $\mu = \{(1,4),(2,3)\}$ is a stable matching as the partner for each agent is at the top of their preference list.

2.2.4 Finding theoretical values for the solvability probability

The purpose of this paper is to determine if the additional information about room choice leads to a greater proportion of stable outcomes for roommate games. In order to conduct such a study, we use Monte-Carlo simulations of random instances of the stable roommates game. However, the probability of finding a stable solution for a random n-game can also be computed exactly, using a method first elucidated by Knuth (1976). This method was

adapted by Pittel (1993) for the stable roommates problem. We present a slightly modified version of the latter's results here.

Consider the framework employed in previous sections. Random preferences are generated using an $n \times n$ matrix X of uniform random variables on $[0, 1]$.

Definition 2.2.5. Let the complete matching space $M = \{\{i, j\} | 1 \leq i \neq j \leq n\}$.

Definition 2.2.6. Let the standard matching $M_0 = \{\{i, i + \frac{n}{2}\} | 1 \leq i \leq \frac{n}{2}\}$.

Definition 2.2.7. Let $A_{i,j}$ denote the event that $(X_{i,j} > X_{i,i+\frac{n}{2}}) \bigcap (X_{j,i} > X_{j,j+\frac{n}{2}})$. [3]

Definition 2.2.8. Let $C = \{(x_1, \ldots, x_n) | 0 \leq x_i \leq 1\}$ be the n-dimensional cube.

Proposition 2.2.1. Let P_n denote the probability that M_0 is stable. Then

$$P_n = \int_C \prod_{\{i,j\} \in M_0^c} 1 - (1 - x_i)(1 - x_j) dx, \qquad dx = dx_1 dx_2 \ldots dx_n.$$

Proof. Note that under M_0, $\{i, i + \frac{n}{2}\}$ and $\{j, j + \frac{n}{2}\}$ are two matched pairs. If event $A_{i,j}$ occurs, however, players i and j would have incentive to match with each other rather than their partners under the matching. This would render the matching unstable. Thus M_0 would be stable if and only if this event did not occur for any agents i and j who are not matched to each other under M_0. We can formalize this as follows. Pairs which are not matched to each other under M_0 belong in the set $M_0^c = M \setminus M_0$. For M_0 to be stable we need $A_{i,j}$ to not occur for all $\{i, j\} \in M_0^c$. Suppose that S is the event that M_0 is stable. Then

$$S = \bigcap_{\{i,j\} \in M_0^c} A_{i,j}^c.$$

[3] Note that these addition operations are modulo n.

Let $X_{\alpha,\alpha+\frac{n}{2}} = x_\alpha$ be observed for all $\alpha \in \{1, 2, \ldots, n\}$. Then

$$\begin{aligned} P(A_{i,j}|X_{i,i+\frac{n}{2}} = x_i, X_{j,j+\frac{n}{2}} = x_j) &= P((X_{i,j} > x_i) \bigcap (X_{j,i} > x_j)) \\ &= P(X_{i,j} > x_i)P(X_{j,i} > x_j) \\ &= (1 - x_i)(1 - x_j) \end{aligned}$$

where the second equality is due to the independence of $X_{i,j}$ and $X_{j,i}$ and the third equality is due to the fact that all elements of X are uniformly distributed on [0,1]. Taking complements, we have that

$$P(A^c_{i,j}|X_{i,i+\frac{n}{2}} = x_i, X_{j,j+\frac{n}{2}} = x_j) = 1 - (1 - x_i)(1 - x_j).$$

Now in order to deduce the expression for $P(S)$, we first look at the conditional probability $P(S|X)$. At this point we switch notations to accommodate the use of probability density functions

$$\begin{aligned} P(S|X_{1,1+\frac{n}{2}} = x_1, \ldots, X_{n+\frac{n}{2}} = x_n) &= f(S|X_{1,1+\frac{n}{2}} = x_1, \ldots, X_{n+\frac{n}{2}} = x_n) \\ &= f\left(\bigcap_{(i,j)\in M_0^c} A^c_{i,j} \middle| X_{1,1+\frac{n}{2}} = x_1, \ldots, X_{n+\frac{n}{2}} = x_n \right) \\ &= \prod_{\{i,j\}\in M_0^c} f\left(A^c_{i,j}|X_{1,1+\frac{n}{2}} = x_1, \ldots, X_{n+\frac{n}{2}} = x_n\right) \quad \text{[Ind. Events]} \\ &= \prod_{\{i,j\}\in M_0^c} 1 - (1 - x_i)(1 - x_j). \end{aligned}$$

Using rules of conditional probability, we have that

$$\begin{aligned} & f\left(S, X_{1,1+\frac{n}{2}} = x_1, \ldots, X_{n+\frac{n}{2}} = x_n\right) \\ =& f\left(S|X_{1,1+\frac{n}{2}} = x_1, \ldots, X_{n+\frac{n}{2}} = x_n\right) f\left(X_{1,1+\frac{n}{2}} = x_1, \ldots, X_{n+\frac{n}{2}} = x_n\right). \end{aligned}$$

Since $X_{1,1+\frac{n}{2}}, \ldots X_{n,n+\frac{n}{2}} \overset{\text{i.i.d}}{\sim} U(0,1)$

$$f\left(X_{1,1+\frac{n}{2}} = x_1, \ldots, X_{n+\frac{n}{2}} = x_n\right) = \prod_{i=1}^{n} f(X_{i,i+\frac{n}{2}}) = \prod_{i=1}^{n} 1 = 1.$$

Therefore

$$\begin{aligned} f\left(S, X_{1,1+\frac{n}{2}} = x_1, \ldots, X_{n+\frac{n}{2}} = x_n\right) &= f\left(S | X_{1,1+\frac{n}{2}} = x_1, \ldots, X_{n+\frac{n}{2}} = x_n\right) \\ &= \prod_{\{i,j\} \in M_0^c} 1 - (1-x_i)(1-x_j). \end{aligned}$$

Integrating out the other variables, we can find the marginal density

$$P_n = f(S) = \int_C \prod_{\{i,j\} \in M_0^c} 1 - (1-x_i)(1-x_j) dx, \qquad dx = dx_1 dx_2 \ldots dx_n.$$

■

Note that while we have ostensibly found P_n associated with a specific matching M_0, the "order" of the agents in the game can be rearranged so that M_0 will represent any arbitrary matching. Thus we have found an analytic expression for the probability that an arbitrary matching is stable in a random n-game.

Unfortunately, the integral is quite difficult to evaluate, both numerically and analytically. As such, for large values of n, the integral expression yields little insight into the actual value for P_n. Nevertheless, we present some values of P_n computed from this integral by Mertens (2005) in table 2.2. These values can be used as a validity check for our Monte-Carlo simulations.

Table 2.2: Probability of solvability for stable roommates instances of size n

P_n			
P_4	0.963	P_8	0.910
P_6	0.933	P_{10}	0.891

2.2.5 Monte-Carlo methods

The main results of this paper are derived computationally, using Monte-Carlo simulations in order to compute the probability of finding a stable solution for a game of size n. Two probabilities are of importance here, described in table 2.3.

Table 2.3: The two kinds of probabilities.

Parameters	
$P_{n,o}$	Probability of finding a stable solution for a game of size n with no information about room choice
$P_{n,r}$	Probability of finding a stable solution for a game of size n with partially endogenous preferences

In order to derive estimates for these probabilities, we implement the procedure outlined in sections 2.2.1 and 2.2.3 along with Irving's algorithm (see an overview in appendix B) to determine whether a random instance of the stable roommates game under the appropriate conditions has a stable solution. This can be described by the Bernoulli random variable

$$B = \begin{cases} 1 & \text{The instance has a stable solution} \\ 0 & \text{The instance has no stable solution.} \end{cases}$$

If we conduct m such trials, then

$$Y = \sum_{i=1}^{m} B_i \sim \text{Binom}(m, p)$$

where $p = P_{n,r}$ or $P_{n,o}$ depending on whether we are simulating a model with or without partially endogenous preferences respectively. Since p is not observed, the goal of conducting this simulation is to find a good estimator $\hat{p}$ of p. We use the standard estimator

$$\hat{p} = \frac{Y}{m}$$

as it is unbiased and consistent (Wackerly et al., 2007) with variance $V(\hat{p}) = p(1-p)/m$.

Note that

$$\hat{p} = \frac{Y}{m} = \frac{1}{m}\sum_{i=1}^{m} B_i = \overline{B} \sim N\left(p, \frac{p(1-p)}{m}\right)$$

by the Central Limit Theorem. From this fact, and by using the method of moment generating functions, we can deduce that for independent estimators $\hat{p}_1$ and $\hat{p}_2$ of p_1 and p_2,

$$\hat{p}_1 - \hat{p}_2 \sim N\left(p_1 - p_2, \frac{p_1(1-p_1)}{m_1} + \frac{p_2(1-p_2)}{m_2}\right)$$

where m_1, m_2 denote sample sizes. Now we are ready to present our hypothesis

$$H_0 : P_{n,o} = P_{n,r} \quad \forall n \in 2\mathbb{N} \setminus \{2\}$$
$$H_a : P_{n,o} < P_{n,r}.$$

Since this is a one-tailed test (our conjecture entails a direction for the difference), we shall use relatively small significance levels in order to minimize the possibility of a Type I error. Given this framework, we have the test statistic

$$Z_t = \frac{\hat{p}_{n,r} - \hat{p}_{n,o}}{\sqrt{\frac{\hat{p}_{n,r}(1-\hat{p}_{n,r})}{m_{n,r}} + \frac{\hat{p}_{n,o}(1-\hat{p}_{n,o})}{m_{n,o}}}} \sim N(0,1).$$

We can reject the null hypothesis when

$$P(Z > Z_t) \leq 0.025.$$

2.3 Results

Running the Monte-Carlo simulation in Python (see appendix D for the code), we can generate estimates for $P_{n,o}$ and $P_{n,r}$ and use the hypothesis test described in the previous section to deduce whether $P_{n,r} > P_{n,o}$. Figure 2.2 shows the graph of such a simulation where $m_1 = m_2 = 1000$. The *dotted* line in the figure depicts the actual values of $P_{n,o}$,

calculated using the multidimensional integral formula described in section 2.2.4. The *solid* line indicates estimates of the probability of solvability without any information about room preferences i.e. $\hat{p}_{n,o}$. The *dashed* line indicates estimates for the probability of solvability with the additional informaton about room preferences; i.e $\hat{p}_{n,r}$.

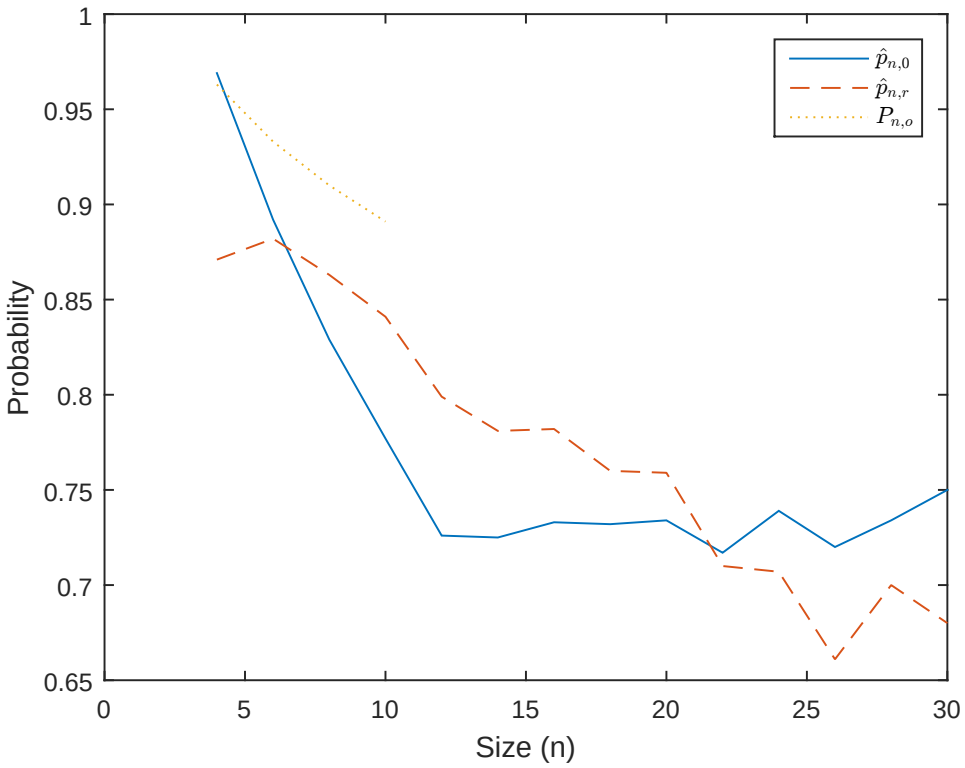


Figure 2.2: Graph of probabilities estimated using Monte-Carlo simulations with sample sizes of 1000.

Both probabilities appear to be decreasing functions of n; this is consistent with the literature (Gusfield and Irving 1989; Mertens 2005). Worryingly, the values of $\hat{p}_{n,o}$ appear to be biased with respect to $P_{n,o}$. In particular, our estimator seems to consistently underestimate the probability of solving a stable roommates instance. While inferring the existence of a bias based on single samples is highly speculative, particularly when we only have four values for $P_{n,o}$, robustness checks conducted in section 2.4.1 reveal that the bias indeed exists. However, it is reasonable to believe that if any bias exists, it is systematic and thus affects

our estimates $\hat{p}_{n,o}$ and $\hat{p}_{n,r}$ in the same fashion, leaving our hypothesis tests valid.[4]

Table 2.4: Z-test for proportions with a sample size of 10000

	n	$\hat{p}_{n,o}$	$\hat{p}_{n,r}$	p-value
1	4	0.969	0.871	1
2	6	0.892	0.882	0.738
3	8	0.829	0.863	0.020*
4	10	0.777	0.841	0.0002*
5	12	0.726	0.799	0.0001*
6	14	0.725	0.781	0.002*
7	16	0.733	0.782	0.006*
8	18	0.732	0.760	0.083
9	20	0.734	0.759	0.109
10	22	0.717	0.710	0.617
11	24	0.739	0.707	0.939
12	26	0.720	0.661	0.997
13	28	0.734	0.700	0.949
14	30	0.750	0.680	1.000

Table 2.4 shows the results of conducting the hypothesis test. As one can see, we can reject the null hypothesis only for a small fraction of the games. In particular, we can reject the null hypothesis for games of size 10,12,14. Indeed, for some games, the inverse hypothesis appears to be true i.e. $P_{n,o} > P_{n,r}$.

These results are borne out in the graph where we see the dashed line intersect the solid line at two points. Between these two points we have that $\hat{p}_{n,r} > \hat{p}_{n,o}$, the only points where our alternative hypothesis could hold. This suggests that endogenizing roommate preferences using exogenous room preferences does not significantly increase the probability of solvability of roommate problems for general n.

In order to confirm these results, we conduct a number of robustness exercises.

[4]For a more formal discussion of this claim, see appendix C.

2.4 Robustness Checks

While one can imagine a plethora of modifications that can be utilized as robustness checks, there are a few that are particularly important in order to establish the veracity of the results.

- **Increasing the sample size**: We have conducted our simulation with a sample size of 1000 for both probabilities for each value of n. Increasing the sample size to 10,000 may yield more robust results.
- **Looking at larger values of** n: One important way to establish if our results hold for general n is to conduct the simulation for larger values of n.

These are fairly straightforward extensions, and we present results from the new simulations in the following sections.

2.4.1 Increasing the sample size

In this simulation, we choose $m_1 = m_2 = 10000$. Figure 2.3 shows the graph of this simulation. This graph can be interpreted akin to figure 2.2, although the color scheme employed is different. Notice that we see the continuing inverse relationship between n and P_n. This confirms the result we got in section 2.3. Again, we see that $\hat{p}_{n,o}$ is a biased estimator of $P_{n,o}$, consistently underestimating the true value of the probability of solving a stable roommates instance. Just like in figure 2.2, the graph for $\hat{p}_{n,r}$ crosses the graph of $\hat{p}_{n,o}$ twice. For values of n between those two points, it is possible that our alternative hypothesis holds. Table 2.5 shows the result of the hypothesis test.

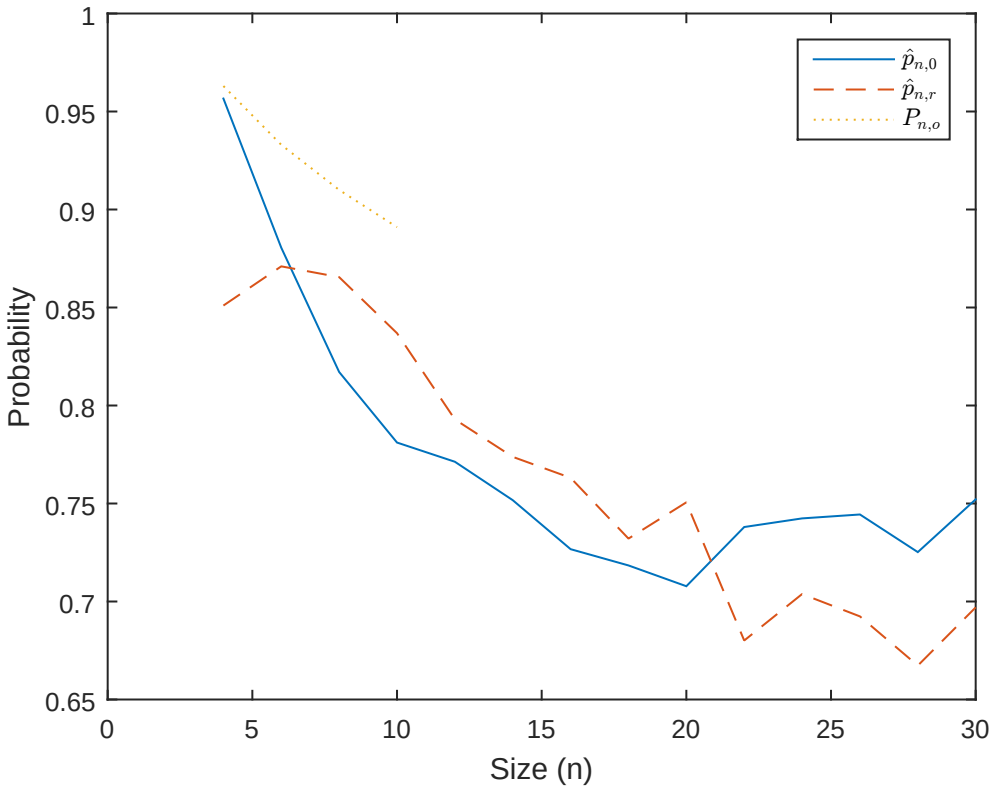


Figure 2.3: Graph of probabilities estimated using Monte-Carlo simulations with a sample size of 10000

In a somewhat sharp contrast with the result found in section 2.3, we see a number of significant tests. This is possibly due to the fact that a large sample size decreases the size of the standard error of our estimator, leading to more extreme test statistics and thus smaller p-values. In the case of a one-sided test, such as this one, extreme test statistics can also lead to extremely large p-values. Indeed, the number of outcomes which are particularly not significant have surfaced as well.

These results essentially confirm the results we arrived at in section 2.3, albeit with greater precision. This is the expected outcome of increasing the sample size of the experiment.

Table 2.5: Z-test for proportions with a sample size of 10000

	n	$\hat{p}_{n,o}$	$\hat{p}_{n,r}$	p-value
1	4	0.957	0.851	1
2	6	0.880	0.871	0.978
3	8	0.817	0.866	0*
4	10	0.781	0.837	0*
5	12	0.771	0.793	0.0001*
6	14	0.752	0.774	0.0001*
7	16	0.727	0.763	0*
8	18	0.718	0.732	0.016*
9	20	0.708	0.751	0*
10	22	0.738	0.680	1
11	24	0.742	0.704	1
12	26	0.744	0.692	1
13	28	0.725	0.667	1
14	30	0.752	0.697	1

2.4.2 Looking at larger values of n

While we have observed the behavior of both $\hat{p}_{n,r}$ and $\hat{p}_{n,o}$ for n up to 30, it is reasonable to question whether the outcomes we observe persist for higher values of n. In particular, we would like to verify if we can reject the null hypothesis in favor of our alternative hypothesis for higher values of n. Looking at figure 2.4, it appears that for larger values of n, the opposite hypothesis holds true. In other words, it appears as if $P_{n,o} > P_{n,r}$. Indeed, examining table 2.6, we see that the p-values are extremely high, suggesting that if we inverted the hypothesis test, we would have significant results. This strongly suggests that our hypothesis does not hold true for values of n greater than 20.

While this appears to be a negative result, it spawns numerous interesting questions. Since the process outlined in section 2.2.3 generates preferences that should ideally lead to more players being ranked highly in each others' preference lists, one would expect that on average more matches would be stable but this is not the case.

An avenue of future research could be investigating why we see the counter-intuitive results that we have generated here.

Table 2.6: Z-test for proportions with a sample size of 1000 for n up to 94. Most values are omitted due to redundancy.

	n	$\hat{p}_{n,o}$	$\hat{p}_{n,r}$	p-value
1	32	0.780	0.570	1
2	34	0.760	0.642	1
3	36	0.754	0.644	1.000
4	38	0.710	0.641	0.999
5	40	0.720	0.581	1
6	42	0.842	0.652	1
7	54	0.818	0.521	1
8	56	0.771	0.598	1
9	58	0.791	0.543	1
10	84	0.810	0.566	1
11	86	0.818	0.592	1
12	88	0.902	0.563	1
13	90	0.870	0.603	1
14	92	0.923	0.571	1
15	94	0.824	0.557	1

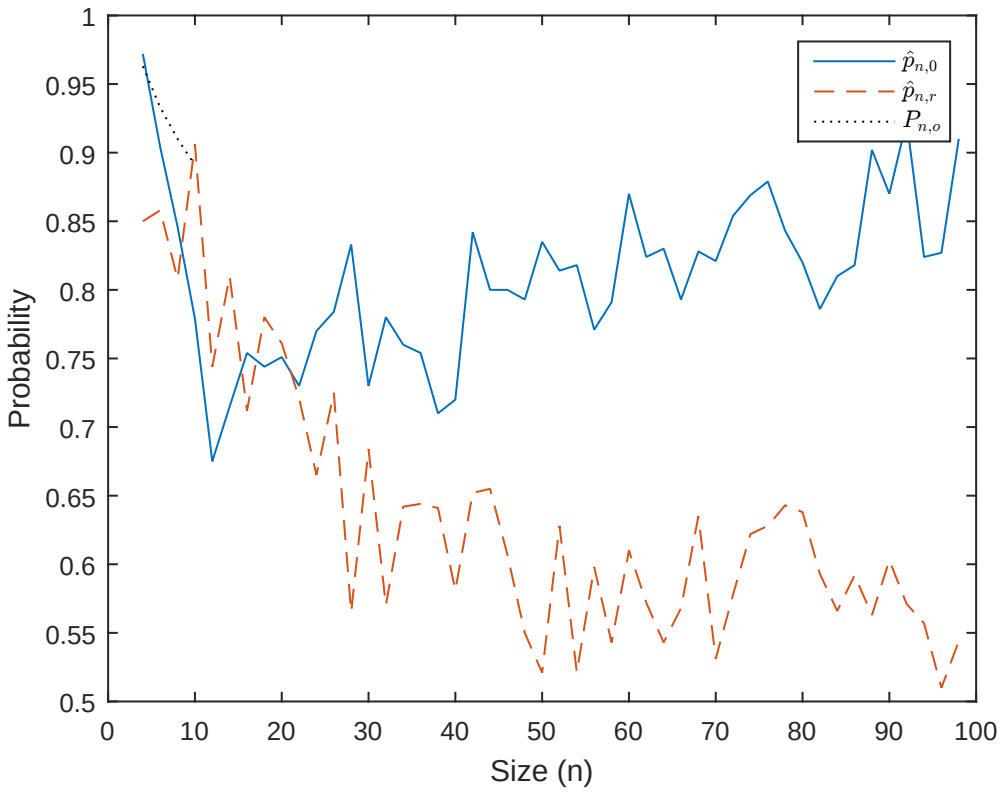


Figure 2.4: Graph of probabilities estimated for $n \leq 100$ using Monte-Carlo simulations with a sample size 1000

The next two robustness checks involve modifying the manner in which the Kendall's tau values are used to generate the partially endogenous preferences. We conduct these checks to ensure that an arbitrary choice of functional form does not have a significant effect on the results.

2.4.3 Modifying the k-factor

Refer back to section 2.2.3. Remember that we construct the augmented matrix $\bar{X}$ from the matrix X of independent, uniform random variables by using the Kendall's tau values by the formula

$$\bar{X}_{i,j} = X_{i,j} + \tau_{i,j}$$

where $\tau_{i,j}$ represents the rank correlation between the room preferences of agents i and j. Since the random variables are uniform on $[0, 1]$ and the values $\tau_{i,j} \in [-1, 1]$, this addition can have a potentially large effect on the ordinality of the random variables in each row of the matrix, thus changing the preference lists generated from the rows. Since ordinality is of paramount importance (as opposed to magnitude), it is worth wondering if a larger change in the values of the random variables would lead to different preference lists and thus different outcomes for stability. This is important as we do not want the fact that τ is bounded in [-1,1] to affect the results of our model. Thus we can modify our augmentation process as follows

$$\bar{X}_{i,j} = X_{i,j} + k\tau_{i,j}, \quad k > 1.$$

Choosing $k = 2$ while letting $m_1 = m_2 = 1000$, we run the Monte-Carlo simulations to yield figure 2.5. Clearly, the probability paths appear to be less smooth, although this could simply be due to random errors in sampling.

Table 2.7: Z-test for proportions with a sample size of 1000, $k = 2$

	n	$\hat{p}_{n,o}$	$\hat{p}_{n,r}$	p-values
1	4	0.948	0.850	1
2	6	0.887	0.890	0.444
3	8	0.938	0.813	1
4	10	0.819	0.874	0.0004*
5	12	0.679	0.830	0*
6	14	0.710	0.810	0.00000*
7	16	0.712	0.779	0.0004*
8	18	0.684	0.705	0.166
9	20	0.713	0.734	0.159
10	22	0.730	0.653	1.000
11	24	0.754	0.682	1.000
12	26	0.736	0.707	0.919
13	28	0.803	0.648	1
14	30	0.750	0.718	0.942

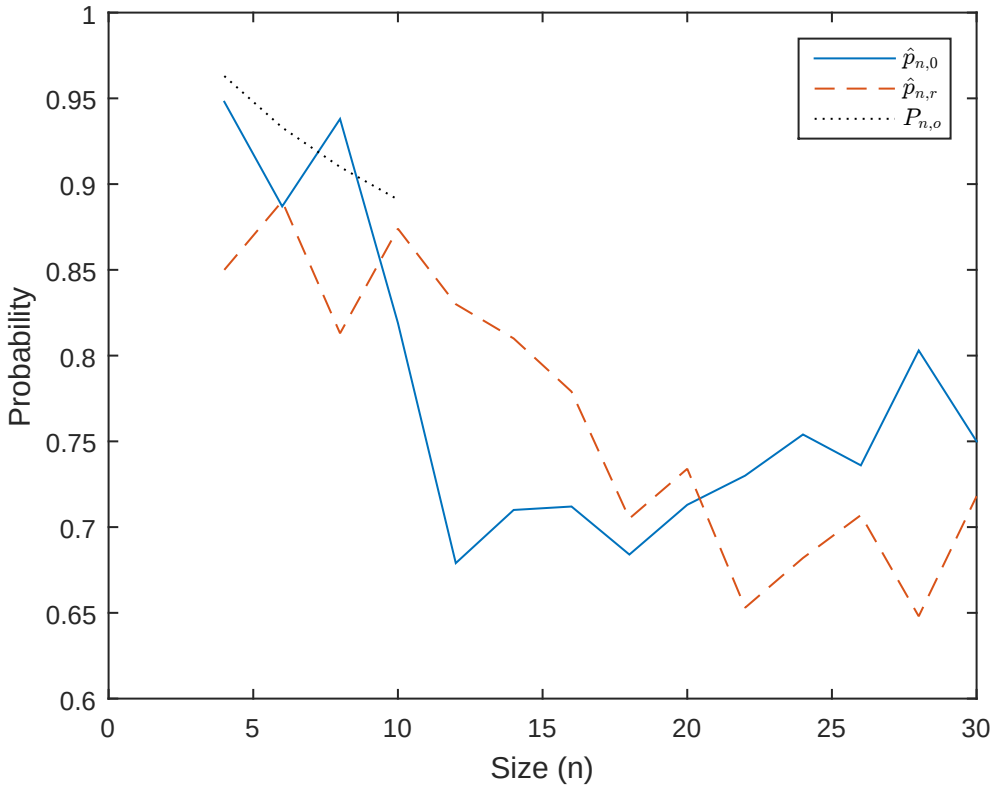


Figure 2.5: Graph of probabilities estimated using Monte-Carlo simulations with a sample size of 1000 and k = 2

In table 2.7, we see the results of the hypothesis tests which are essentially consistent with the results we obtained earlier. The proportion of significant outcomes is slightly larger and the significant (and near significant) values lie in the same range of n-values as seen earlier.

2.4.4 Trying a multiplicative model

As seen in sections 2.2.3 and 2.4.3, we have utilized an additive model in order to construct the augmented matrix $\bar{X}$. As a robustness exercise, we can relax the assumption of this specific functional form and see if the results still hold. In particular, we can try a multiplicative model and observe if the results we obtain are consistent with the ones obtained from the additive model. In the multiplicative model, the augmented matrix is constructed using the formula

$$\bar{X}_{i,j} = \tau_{i,j} X_{i,j}.$$

Using this framework and setting $m_1 = m_2 = 1000$ as usual, we arrive at the results encapsulated in figure 2.6 and table 2.8. Here the results are far more ambiguous than in other iterations of the model, with significant test outcomes being sparsely distributed between the values of $n = 6$ and $n = 20$.

The results from this model are hard to interpret; while they don't invalidate our earlier results, the also do not seem to suggest that our model is robust to changes in the functional form of the matrix augmentation process.

Overall, the robustness analysis seems to reveal that our initial hypothesis was ill-conceived. Future research should aim to uncover the underlying structure of stable roommates with random preferences where the preferences are not generated from a uniform distribution. This should lend insight into the results obtained here.

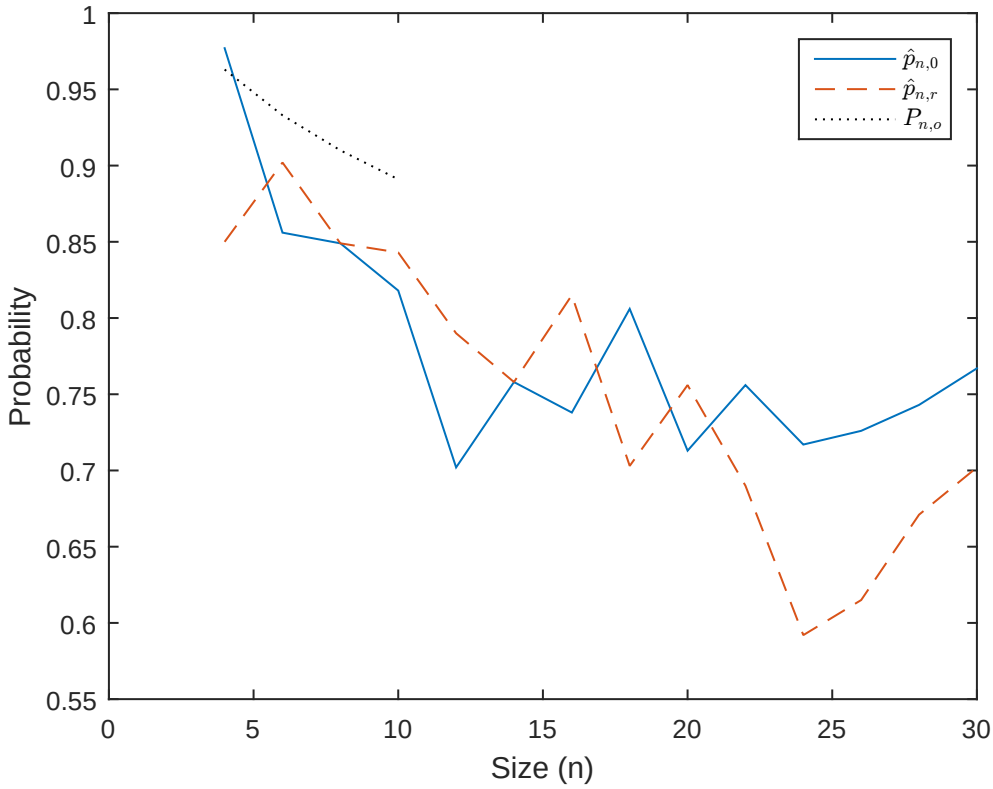


Figure 2.6: Graph of probabilities estimated using Monte-Carlo simulations in multiplicative mode with a sample size of 1000

Table 2.8: Z-test for proportions with a sample size of 1000 in a multiplicative model

	n	$\hat{p}_{n,o}$	$\hat{p}_{n,r}$	p-values
1	4	0.977	0.850	1
2	6	0.856	0.902	0.001*
3	8	0.849	0.849	0.500
4	10	0.818	0.843	0.076
5	12	0.702	0.790	0.00000*
6	14	0.758	0.758	0.500
7	16	0.738	0.815	0.00002*
8	18	0.806	0.703	1.000
9	20	0.713	0.756	0.017*
10	22	0.756	0.690	0.999
11	24	0.717	0.592	1
12	26	0.726	0.615	1.000
13	28	0.743	0.671	1.000
14	30	0.767	0.702	0.999

2.5 Conclusion

Our results, although ambiguous, provide evidence that our hypothesis does not universally hold true. That is to say, partially endogenous preferences do not necessarily lead to a higher probability of finding a stable solution to an instance of the roommates game. This is surprising, given the fact that the process outlined in section 2.2.3 should lead to greater reciprocity in roommate rankings across students. In particular, we would expect students with similar room choices to rank each other highly and students with dissimilar room choices to rank each other poorly. Consistent rankings should, in turn, lead to more stable matchings, as asserted by Pittel (1993).

The counter-intuitive results suggest two interpretations. One could be that the bias observed in our estimates has simply rendered our results meaningless. If order to test this case, replication studies should be conducted to ensure that the results obtained here are robust.

The second, and perhaps more important interpretation, is that the mathematical structure of this game yields counter-intuitive results. In order to investigate this second line of thought, we suggest an approach akin to the one followed in section 2.2.4 where we construct a formula for the exact value of $P_{n,o}$. The principal hindrance in constructing a similar formula for the value of $P_{n,r}$ is due to the difficulty in deriving a sampling distribution for Kendall's τ. If an approximate distribution could be found, then we could use the method of distributions to derive a distribution for the the sum of a uniform and τ random variable and consequently deduce a formula for $P_{n,r}$. We hope to continue research into this line of inquiry in the future.

Bibliography

Hervé Abdi. The kendall rank correlation coefficient. *Encyclopedia of Measurement and Statistics. Sage, Thousand Oaks, CA*, pages 508–510, 2007.

Atila Abdulkadiroğlu and Tayfun Sönmez. House allocation with existing tenants. *Journal of Economic Theory*, 88(2):233–260, 1999.

Atila Abdulkadiroğlu and Tayfun Sönmez. School choice: A mechanism design approach. *American Economic Review*, 93(3):729–747, 2003.

David J Abraham and David F Manlove. Pareto optimality in the roommates problem. In *Technical Report No. TR-2004-182*. The Computing Science Department of Glasgow University, 2004.

Kim-Sau Chung. On the existence of stable roommate matchings. *Games and Economic Behavior*, 33(2):206 – 230, 2000. ISSN 0899-8256. doi: https://doi.org/10.1006/game.1999.0779. URL http://www.sciencedirect.com/science/article/pii/S0899825699907790.

David Gale and Lloyd S Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.

Dan Gusfield and Robert W Irving. *The stable marriage problem: structure and algorithms*. MIT press, 1989.

Robert W Irving. An efficient algorithm for the "stable roommates" problem. *Journal of Algorithms*, 6(4):577–595, 1985.

Donald Ervin Knuth. *Mariages Stables: et leurs relations avec d'autres problèmes combinatoires*. Les Presses de l'Université de Montréal), 1976.

Stephan Mertens. Random stable matchings. *Journal of Statistical Mechanics: Theory and Experiment*, 2005(10):P10008, 2005.

Coe College of Computer Science. stableroomate python solution. https://github.com/CoeCS/tacklebox/tree/master/stableroomate, 2012.

Boris Pittel. The "stable roommates" problem with random preferences. *The Annals of Probability*, pages 1441–1477, 1993.

Boris G. Pittel and Robert W. Irving. An upper bound for the solvability probability of a random stable roommates instance. *Random Structures Algorithms*, 5(3):465–486, 1994. ISSN 1098-2418. doi: 10.1002/rsa.3240050307. URL http://dx.doi.org/10.1002/rsa.3240050307.

Alvin E Roth. The economics of matching: Stability and incentives. *Mathematics of operations research*, 7(4):617–628, 1982.

Alvin E Roth. The evolution of the labor market for medical interns and residents: a case study in game theory. *Journal of political Economy*, 92(6):991–1016, 1984.

Alvin E Roth and Elliott Peranson. The redesign of the matching market for american physicians: Some engineering aspects of economic design. Technical report, National bureau of economic research, 1999.

Alvin E Roth and Marilda Sotomayor. Two-sided matching. *Handbook of game theory with economic applications*, 1:485–541, 1992.

Lloyd Shapley and Herbert Scarf. On cores and indivisibility. *Journal of mathematical economics*, 1(1):23–37, 1974.

Jimmy JM Tan. A necessary and sufficient condition for the existence of a complete stable matching. *Journal of Algorithms*, 12(1):154–178, 1991.

Dennis Wackerly, William Mendenhall, and Richard Scheaffer. *Mathematical statistics with applications*. Nelson Education, 2007.

Chapter 3

Appendices

A Kendall's Tau

The measure of ordinal association used in this project is Kendall's tau (τ). A correlation, τ is bounded in [-1,1] where 1 represents perfect positive correlation and -1 represents perfect negative correlation. In order to understand how this correlation works, we first give a formal definition of τ and then provide an example to motivate the intuition behind the metric.

Let $X = \{x_1, \ldots, x_n\}$ be a collection of n unspecified objects. Then let R be the space of all possible rankings of the objects in X. Every $r_i \in R$ represents a permutation of the set $\{1, 2, \ldots, n\}$. Clearly $|R| = n!$. Now for any $r_i \in R$, let r_{ia} denote the rank of x_a in the ranking r_i. Thus if $r_{i5} = 6$, we have that in the ith ranking in R, object x_5 is ranked in the 6th position.

Definition A.1. Given two rankings $r_i, r_j \in R$, the pair (x_a, x_b) is called *concordant* if and only if

$$\operatorname{sgn}(r_{ia} - r_{ib})(r_{ja} - r_{jb}) = 1$$

where

$$\operatorname{sgn}(x) = \begin{cases} 1 & x > 0; \\ 0 & x = 0; \\ -1 & x < 0. \end{cases}$$

Similarly, (x_a, x_b) is called *discordant* if and only if

$$\operatorname{sgn}(r_{ia} - r_{ib})(r_{ja} - r_{jb}) = -1.$$

Informally, this tells us that a concordant pair is a pair of objects x_a, x_b in X which are ranked consistently by two rankings r_i and r_j. That is to say, either $x_a \succ x_b$ in both r_i, r_j or $x_b \succ x_a$ in both r_i, r_j. A discordant pair is ranked inconsistently.

Definition A.2. Given two rankings $r_i, r_j \in R$, let C equal the number of concordant pairs in X and let D equal the number of discordant pairs in X. Then the Kendall's τ coefficient can be defined as

$$\tau_{ij} = \frac{C - D}{C + D}.$$

Now we can explore the intuition behind this further with an example.

Example A.1. Suppose two Colby students, code named 1 and 2, are asked to rank four dorms in their order of preference. The dorms are named Dana, Foss, Woodman and Coburn.[1] The students rank the dorms in table 3.1. There are a total of $\binom{4}{2} = 6$ pairs

Table 3.1: Dorm rankings by students 1 and 2

Dorm	r_1	r_2
Dana	2	4
Foss	3	2
Woodman	1	1
Coburn	4	3

[1] These are actual dorms at Colby College.

we can examine here. Since Woodman is ranked as the favorite dorm by both students, we already have three concordant pairs. Now (Dana,Foss) is a discordant pair since student 1 prefers Dana to Foss but student 2 prefers Foss to Dana. Similarly, (Dana, Coburn) is a discordant pair. However, (Foss, Coburn) is a concordant pair.

Thus, $C = 4$, $D = 2$ and so

$$\tau_{12} = \frac{4-2}{4+2} = \frac{1}{3} = 0.33$$

B Irving's Algorithm

The workhorse behind this project is Irving's algorithm as we need it in order to determine if a given instance of the stable roommates game admits a stable solution. Given the pivotal role it plays in this research, it is imperative that we give an overview of the algorithm. Note that this appendix does not provide a discussion on the correctness or the performance of this algorithm; an exhaustive deliberation on these topics can be found in Irving (1985) or Gusfield and Irving (1989). Before we actually discuss the details of the algorithm, we must set up the roommates game formally.

Let A be the space of agents such that $|A| = n$ where n is even. For every $a \in A$, let P_a denote the ordered preference list consisting of agents from $A \setminus \{a\}$. Treating P_a as a column vector, we can generate the *preference matrix* T by horizontally concatenating P_a for every $a \in A$. In this case, $T_{i,j}$ denotes the agent which occupies the ith position in agent j's preference list P_j.

A *preference table*, on the other hand, describes a reduced preference matrix in which some entries have been deleted. This reduction is always done symmetrically, in that if agent a is deleted from agent b's preference list, then b is also removed from a's preference list. Formally, we say that the pair (a, b) is removed from the preference matrix. New preference tables can be derived from reducing old ones in the manner specified here.

Phase 1

The algorithm can be conceptually (and computationally) divided into two phases, the first of which is reminiscent of the deferred acceptance algorithm provided by Gale and Shapley (1962) for the stable marriage problem. In order to understand the steps of this phase, we must define the notions of a *free* and *semiengaged* agent.

Definition B.1. The semiengagement relation $\overset{\text{S}}{\sim}$ is a non symmetric relation such that for any $x, y \in A$, $x \overset{\text{S}}{\sim} y$ denotes that y holds a proposal from x. In this case, we say x is

semiengaged to y. If there is no $y \in A$ such that $x \overset{S}{\sim} y$ then x is said to be *free*.

Note that the non symmetric nature of this relation is important. If $a \overset{S}{\sim} b$, it could be that b is free or that $b \overset{S}{\sim} c$ for some $c \in A \setminus \{a\}$.

Now we can describe the algorithm recursively/inductively. In the initial state (base case) of the game, all agents are free. Then, a random agent a proposes to the person at the top of his preference list $P_a(1) = b$. Thus $a \overset{S}{\sim} b$. When this happens, all agents c such that $b \succ_a c$ are removed symmetrically from the preference matrix by the reduction process described earlier, yielding a preference table.

Proceeding inductively, at any arbitrary stage in the algorithm, a free person x makes a proposal to the person at the top of their preference list (say y), if the list is not empty. Note that that y would not reject x immediately, since if y had held a better offer x would already have been rejected due to the reduction process described above. This algorithm continues to iterate unless there are no free persons (everyone is semiengaged) or if there is a person who runs out of people to propose to (their preference list is empty). If it is the latter case, the algorithm tells us that no stable matching exists (see Gusfield and Irving, 1989, Chapter 4). In case of the former, we can have two situations. In the first, the preference table is reduced to the form where each preference list contains a single agent. This represents a stable matching (Irving 1985; Gusfield and Irving 1989). If we have more than one agent in any preference list, we need to move to Phase 2 of the algorithm. The pseudocode shown below in Algorithm 1 provides the steps to Phase 1 of Irving's algorithm. Remember that the reduction referred to in the pseudocode follows the process outlined in paragraph 3 of this section. In order to illustrate Phase 1 concretely, we provide an example.

Data: Preference Matrix
Result: Reduced Preference Table

$A \leftarrow [a_1, a_2 \dots a_n]$ `/* Generate agents */`
$T \leftarrow [P_1, P_2, \dots, P_n]$ `/* generate preference matrix */`
while $\exists a \in A_{free}$ *s.t.* $P_a \neq \emptyset$ **do**
  $b \leftarrow P_a(1)$
  if $c \overset{S}{\sim} b$ **then**
    c.status $\leftarrow$ free
  end
  assign $a \overset{S}{\sim} b$.
  while $\exists a' \in A$ *s.t.* $a \succ_b a'$ **do**
    $T \leftarrow T \setminus (b, a')$
  end
end
if *T has only one entry per column* **then**
  return T
end
if *T has an empty column* **then**
  return *No solution exists*
end
if *T has multiple entries per column* **then**
  Go to phase 2
end

Algorithm 1: Pseudocode for Phase 1 of Irving's algorithm

Example B.1. Let A,B,C,D,E and F be agents with the following preference table.

A	B	C	D	E	F
B	D	D	F	F	A
D	E	E	C	C	B
F	F	F	A	D	D
C	A	A	E	B	C
E	C	B	B	A	E

We can execute phase 1 as follows. It would be helpful to the reader to follow along with a paper and pencil, and cross out entries in the preference table as they are removed.

- A proposes to B; A $\overset{S}{\sim}$ B. (B,C) is removed from the preference table.

- B proposes to D; B $\overset{S}{\sim}$ D.
- C proposes to D; D rejects B and so B is free and C $\overset{S}{\sim}$ D. (D,A),(D,E),(D,B) are removed from the table.
- B proposes to E; B $\overset{S}{\sim}$ E. (E,A) is removed from the table.
- D proposes to F; D $\overset{S}{\sim}$ F. (F,C),(F,E) are removed from the table.

Let's take a break and look at the state of the preference table.

A	B	C	D	E	F
(B)	~~D~~	(D)	(F)	~~F~~	A
~~D~~	(E)	E	C	C	B
F	F	~~F~~	~~A~~	~~D~~	D
C	A	A	~~E~~	B	~~C~~
~~E~~	~~C~~	~~B~~	~~B~~	~~A~~	~~E~~

In this reduced table, slashes denote deletion whereas circles denote semiengagement. Since agents E and F are still free, the algorithm continues to iterate.

- E proposes to C; E $\overset{S}{\sim}$ C. (C,A) is removed from the table.
- F proposes to A. F $\overset{S}{\sim}$ A.

Let's examine the preference table again.

A	B	C	D	E	F
(B)	~~D~~	(D)	(F)	~~F~~	(A)
~~D~~	(E)	E	C	(C)	B
F	F	~~F~~	~~A~~	~~D~~	D
~~C~~	A	~~A~~	~~E~~	B	~~C~~
~~E~~	~~C~~	~~B~~	~~B~~	~~A~~	~~E~~

Here, we can see that everyone is semiengaged but that the reduced table contains preferences with more than a single entry for every agent. This means we need to use the second phase of Irving's algorithm to find a stable matching (or, conversely, verify that no stable matching exists).

Phase 2

The second phase of the algorithm leads to further reductions in the preference table, proceeding iteratively until we reach one of two terminating conditions. In the first condition, we have that each preference table contains only one person. In this case, we have a stable matching. In the other condition, we have one person whose preference list is empty. In this case no stable matching exists.

The procedure of Phase 2 involves identifying what Irving (1985) calls an *all-or-nothing cycle*. This cycle consists of a sequence of pairs of agents

$$(a_0, b_0), (a_1, b_1) \dots (a_{r-1}, b_{r-1})$$

such that b_i is the first person in a_i's reduced preference list and the second person in a_{i-1}'s preference list. This sequence is called a *cycle* because it repeats after some fixed length r. Thus all addition (subtraction) operations in the subscripts are taken modulo r. Irving proves that such a cycle always exists in a Phase 1 reduced preference table which requires a Phase 2 reduction. An example of such a table is the final table shown in Example B.1.

In order to find an all-or-nothing cycle, we implement the following steps (see Irving's paper for an explanation as to why these steps yield the correct sequence). Let x_0 be an arbitrary agent in the preference table who has more than one entry in her preference list. Then we can generate the following sequences

- y_i = The second person in x_i's preference list.
- x_{i+1} = The last person in y_i's preference list.

The sequence x_i will eventually cycle and thus we can define

$$a_i = x_{s+i} \quad (i = 0, 1, 2, \ldots, r-1)$$

where x_s denotes the first element of the sequence x_i to repeat. Now that we have sequence a_i, we can quickly determine the sequence b_i. Phase 2 reduction involves removing the all-or-nothing cycle from the preference table. In other words, we remove all the pairs $(a_i, b_i), \quad i \in \mathbb{Z} \bmod r$. We summarize this process in the following pseudocode.

Function `FindCycle(`T,A`):`

```
/* A represents set of agents.                                    */
/* T represents reduced preference table.                         */
```

X $\leftarrow$ Empty vector
Y $\leftarrow$ Empty vector
A $\leftarrow$ Empty vector
B $\leftarrow$ Empty vector
X[0] $\leftarrow$ $a \in A$ s.t len$(P_a) > 1$.
$j \leftarrow 0$
while *X has no repetitions* **do**
  Y[j] $\leftarrow P_{X[j]}(2)$
  X[$j+1$] $\leftarrow P_{Y[j]}(-1)$
  $j \leftarrow j + 1$
end
$s \leftarrow$ index where X[j] first occurs.
A $\leftarrow$ X with everything from X[1] to X[$s-1$] removed.
B $\leftarrow P_A(2)$
return *(A,B)*

Algorithm 2: Find an all-or-nothing cycle from a preference table.

Finally, actually removing the all-or-nothing cycle is summarized in the pseudocode below.

Data: Preference Matrix
Result: Reduced Preference Table

$A \leftarrow [a_1, a_2 \dots a_n]$ `/* Generate agents */`
$T \leftarrow [P_1, P_2, \dots, P_n]$ `/* generate preference matrix */`
$T \leftarrow \text{Phase1}(T)$`/* Apply Phase 1 */`
while *T has mupltiple entries in each column* **do**
  $T \leftarrow T \backslash$`FindCycle`$(T, A)$
end
if *T has only one entry per column* **then**
  return *T*
end
if *T has an empty column* **then**
  return *No solution exists*
end

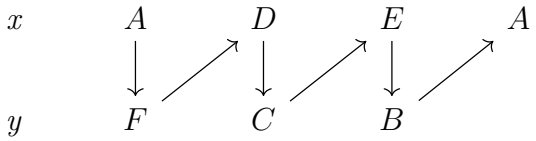
Algorithm 3: Phase 2 algorithm including Phase 1 as a function.

We show this in action by continuing the example we laid out in Phase 1.

Example B.2. Let us first set out the preference table found at the end of Example B.1.

A	B	C	D	E	F
B	.	D	F	.	A
.	E	E	C	C	B
F	F	.	.	.	D
.	A	.	.	B	.
.	.	.	.	.	.

Let $x_0 = \text{A}$. We can make the x, y sequences using the algorithm discussed earlier.

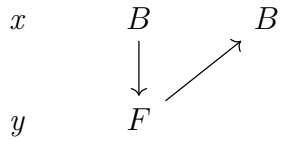


Since we have found the repeating pattern in the sequence x, we can now construct the sequence $a_i = x_{s+i}$. Since $s = 0$ here, we simply have that $a_i = x_i$. The corresponding b_i sequence is found by simply finding the first choice of every agent a_i. Thus in this case, the

all-or-nothing cycle is (a_i, b_i) = (A,B),(D,F),(E,C). Removing these from preference table yields the following:

A	B	C	D	E	F
~~B~~	.	D	~~F~~	.	A
.	E	~~E~~	C	~~C~~	B
F	F	.	.	.	~~D~~
.	~~A~~	.	.	B	.
.	.	.	.	.	.

Since we still have a preference table which is not completely reduced i.e. there are still columns with multiple entries, we do another round of Phase 2 reduction. Let x_0 =B this time.



That was quick! Again $s = 0$ so $a_i = x_i$. Thus our all-or-nothing cycle is (B,F). Removing this symmetrically, we have the completely reduced table

A	B	C	D	E	F
~~B~~	.	D	~~F~~	.	A
.	E	~~E~~	C	~~C~~	~~B~~
F	~~F~~	.	.	.	~~D~~
.	~~A~~	.	.	B	.
.	.	.	.	.	.

Looking at the table, we can infer the stable matching (A,F),(B,E),(C,D).

C Bias and the Validity of Results

In section 2.3 of chapter 2, we saw that the estimates of the probability of solving a random stable roommates game i.e. $\hat{p}_{n,o}$ appeared to be biased. In particular, we saw that our estimates consistently appeared to underestimate the value of $P_{n,o}$, which we derived using the multidimensional integral formula proved in section 2.2.4. Since we do not have a similar expression to compute the probability of solving a roommates instance with partially endogenous preferences i.e. $P_{n,r}$, we cannot say for certain that $\hat{p}_{n,r}$ is biased with respect to $P_{n,r}$ with certainty. However, since the data generating processes for both estimates are similar, it is reasonable to contend that the same kind of bias exists for the two estimates.

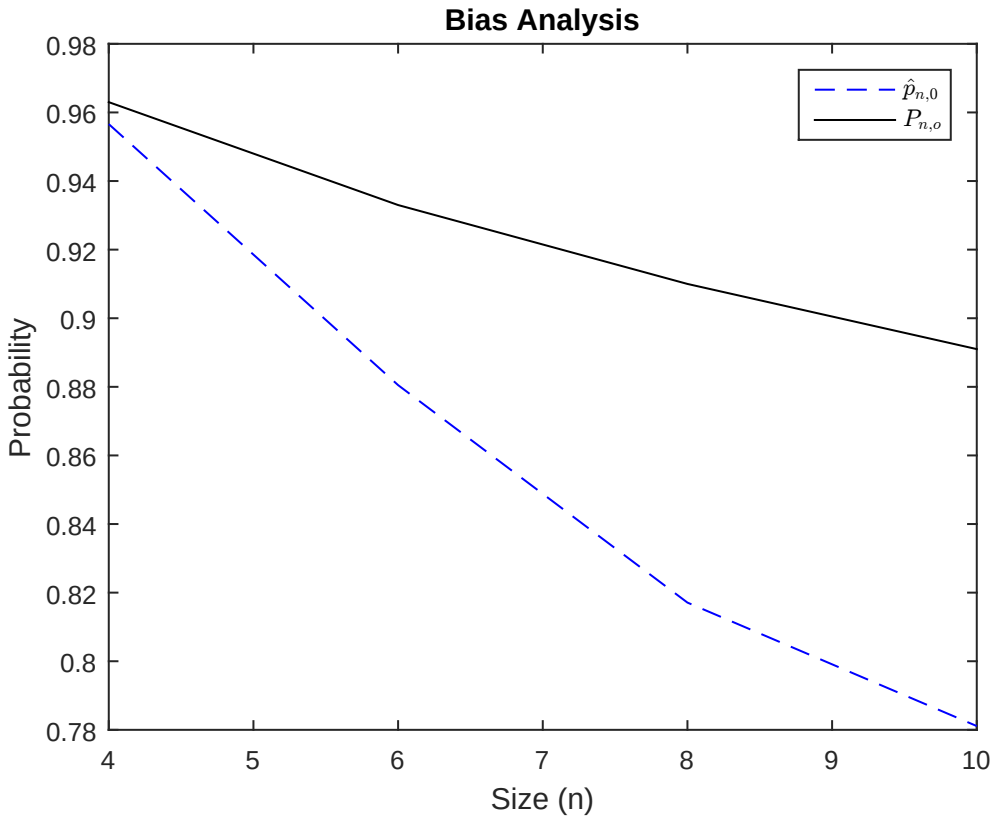


Figure 3.1: Graphs of $P_{n,o}$ and $\hat{p}_{n,o}$ for values of n up to 10.

In particular, we expect the bias to be of the form

$$\mathrm{E}[\hat{p}_{n,o}] = P_{n,o} + \underbrace{f(n)}_{\text{bias}}$$
$$\mathrm{E}[\hat{p}_{n,r}] = P_{n,r} + \underbrace{f(n)}_{\text{bias}}.$$

Our test statistic is unaffected because

$$\mathrm{E}[\hat{p}_{n,o} - \hat{p}_{n,r}] = P_{n,o} - P_{n,r}.$$

D Source Code

tests.r
This file conducts the hypothesis tests as outlined in section 2.2.5

```
    library(stargazer)

# STANDARD MODEL
df <- read.csv('data_32_1000_1_False_0.csv')
len <- length(df$P1)
pvals <- numeric(len)
ssize <- 1000
for (i in 1:len)
{
  result = prop.test(c(df$P1[i],df$P2[i]),c(ssize,ssize),alternative='less',conf.level =
  df$P1[i] = df$P1[i]/ssize
  df$P2[i] = df$P2[i]/ssize
  pvals[i] = result$p.value
}

df$pvals = pvals

stargazer(df,summary = FALSE)


#HIGH SAMPLE SIZE MODELs
df2 <- read.csv('data_32_10000_1_False_0.csv')
len <- length(df2$P1)
pvals <- numeric(len)
ssize <- 10000
for (i in 1:len)
{
  result = prop.test(c(df2$P1[i],df2$P2[i]),c(ssize,ssize),alternative='less',conf.level
  df2$P1[i] = df2$P1[i]/ssize
  df2$P2[i] = df2$P2[i]/ssize
  pvals[i] = result$p.value
}

df2$pvals = pvals

stargazer(df2,summary=FALSE)

#k-Factor Model
```

```
df3 <- read.csv('data_32_1000_2_False_0.csv')
len <- length(df3$P1)
pvals <- numeric(len)
ssize <- 1000
for (i in 1:len)
{
  result = prop.test(c(df3$P1[i],df3$P2[i]),c(ssize,ssize),alternative='less',conf.level
  df3$P1[i] = df3$P1[i]/ssize
  df3$P2[i] = df3$P2[i]/ssize
  pvals[i] = result$p.value
}

df3$pvals = pvals

stargazer(df3,summary=FALSE)

#Multiplicative model
df4 <- read.csv('data_32_1000_1_True_0.csv')
len <- length(df4$P1)
pvals <- numeric(len)
ssize <- 1000
for (i in 1:len)
{
  result = prop.test(c(df4$P1[i],df4$P2[i]),c(ssize,ssize),alternative='less',conf.level
  df4$P1[i] = df4$P1[i]/ssize
  df4$P2[i] = df4$P2[i]/ssize
  pvals[i] = result$p.value
}

df4$pvals = pvals

stargazer(df4,summary=FALSE)
```

game.py
This file implements in Python the procedures described in sections 2.2.1 and 2.2.3

```
# author: Yashaswi Mohanty
# last modified: 4/26/2018

import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import numpy as np
from stableroomate import *
import scipy.stats as stats
from multiprocessing import Pool
```

```
import sys
import os.path
import pandas as pd


class Player:
    def __init__(self, id, rooms=False):
        """
        Constructor for player instance

        :param id: player id
        :param rooms: boolean value representing if the game is played with
            information about rooms
        """
        self.id = id
        if rooms:
            self.room_prefs = []
        self.roommate_prefs = []

    def get_id(self):
        """
        Accessor for player id
        :rtype: int
        :return: player id
        """
        return self.id

    def get_room_prefs(self):
        """
        Accessor for player room prefereneces
        :rtype: list
        :return: player room preferences
        """
        return self.room_prefs


class Game:

    def __init__(self, size, room_mode=False, k=1, mult=False):
        """
        Initializes the game state.

        :param size: The number of agents in the game. Has to be a multiple of 2
        :param room_mode: Boolean values representing whether the game is being
            played with room information
        :param k: k-factor modified (see paper)
        :param mult: Boolean value which controls whether we use the
```

```
        multiplicative or additive model
    """
    if size % 2 != 0:
        print "The game size must be even."
    self.size = size
    self.room_mode = room_mode
    self.players = []
    self.k = k
    self.mult = mult
    if room_mode:
        self.rooms = range(self.size / 2)
    for i in range(self.size):
        self.players.append(Player(i, room_mode))

    self.setup()

def setup(self):
    """
    Sets up the preferences of the players
    """
    # Construct room preferences randomly if in room mode
    if self.room_mode:
        for player in self.players:
            player.room_prefs = np.random.permutation(self.rooms)

    # Generate a matrix of random uniform (0,1) variables
    pref_matrix = np.random.rand(self.size, self.size)

    # This is a way of effectively preventing people from ranking themselves
        as roommates.
    np.fill_diagonal(pref_matrix, -9999999)

    order_matrix = np.zeros((self.size, self.size - 1))

    # generate the kendall tau values if in room mode
    if self.room_mode:
        taus = {}
        for player1 in self.players:
            p1_taus = {}
            for player2 in self.players:
                if player1 != player2:
                    p1_taus[player2.get_id()] = \
                        stats.kendalltau(player1.get_room_prefs(),
                            player2.get_room_prefs())[0]
            taus[player1.get_id()] = p1_taus

        # add or multiply kendall tau values depending on the model being
```

```
                used.
            for i in range(self.size):
                for j in range(self.size):
                    if i != j:
                        if self.mult:
                            pref_matrix[i][j] *= self.k * taus[i][j]
                        else:
                            pref_matrix[i][j] += self.k * taus[i][j]

        # just in case
        np.fill_diagonal(pref_matrix, -999999)

        # generate preferences from the random matrix
        for i in range(self.size):
            order_matrix[i] = pref_matrix[i].argsort()[::-1][:self.size - 1]

        # format the preference lists as a Python dictionary which can be read by
            our implementation of Irving's algorithm.
        self.prefs = self.generate_dict(order_matrix)


    @staticmethod
    def generate_dict(order_matrix):
        """
        Generates a dictionary of preferences from the matrix of preferences
        :param order_matrix: matrix of preferences
        :return: prefs: a dictionary of preferences
        """
        prefs = {}

        for i in range(order_matrix.shape[0]):
            prefs[str(i)] = [str(int(j)) for j in order_matrix[i]]

        return prefs

    def play(self):
        """
        play a round of the stableroommates game with the given preferences
        :return: a stable matching or None, depending on whether a matching
            exists.
        """
        return stableroomate(self.prefs, parse=False)

# non parallel version of the game
def game(size, iter, k=1, mult=False):
    norm_solns = []
    for i, x in zip(range(4, size, 2), range((size - 4) / 2)):
```

```
            game = Game(i, False, k, mult)
            norm_solns.append(0)
            for j in range(iter):
                game.setup()
                value = game.play()
                if value is not None:
                    norm_solns[x] += 1

            norm_solns[x] = float(norm_solns[x]) / iter

        room_solns = []

        for i, x in zip(range(4, size, 2), range((size - 4) / 2)):
            game2 = Game(i, True, k, mult)
            room_solns.append(0)
            for j in range(iter):
                game2.setup()
                value = game2.play()
                if value is not None:
                    room_solns[x] += 1

            room_solns[x] = float(room_solns[x]) / iter

        plt.plot(range(4, size, 2), norm_solns)
        plt.plot(range(4, size, 2), room_solns)
        plt.savefig('fig1')


    def game_parallel(size, iter, k=1, mult=False):
        """
        The main function of our simulation.

        :param size: The number of agents in the simulation
        :param iter: The sample size of the simulation
        :param k: The k-factor
        :param mult: Boolean value for determining whether we are using a
            multiplicative model or an additive model
        """
        # init lists for containing the values of p_{n,o}
        norm_solns = []
        norm_props = []
        for i, x in zip(range(4, size, 2), range((size - 4) / 2)):
            game = Game(i, False, k, mult)
            norm_solns.append(0)

            # run game in Parallel with 5 processes
            values = Pool(5).map(parallel, [game] * iter)
```

```
    norm_solns[x] = sum(values)
    norm_props.append(float(norm_solns[x]) / iter)

# init lists for containing the values of p_{n,r}
room_solns = []
room_props = []

for i, x in zip(range(4, size, 2), range((size - 4) / 2)):
    game2 = Game(i, True, k, mult)
    room_solns.append(0)

    # run game in parallel with 5 processes
    values = Pool(5).map(parallel, [game2] * iter)
    room_solns[x] = sum(values)
    room_props.append(float(room_solns[x]) / iter)

#generate figure
l1, = plt.plot(range(4, size, 2), norm_props, label='Normal Simulated
    Probabilities')
l2, = plt.plot(range(4, size, 2), room_props, label='Augmented Simulated
    Probabilities')
l3, = plt.plot([4, 6, 8, 10], [0.963, 0.933, 0.91, 0.89], label='Theoretical
    Probabilities')
plt.xlabel('Size (n)')
plt.ylabel('Probability')
plt.legend([l1, l2, l3])

#generate csv for hypothesis testing
out = [(i, j) for i, j in zip(norm_solns, room_solns)]
labels = ['P1', 'P2']
df = pd.DataFrame.from_records(out, columns=labels,index=range(4,size,2))

# make sure we are not overwriting other files while saving csv and figures
m1 = 0
while os.path.exists('data_%d_%d_%d_%s_%d.csv' % (size,iter,k,mult,m1)):
    print "T1"
    m1 = m1 + 1
df.to_csv('data_%d_%d_%d_%s_%d.csv' % (size,iter,k,mult,m1))

m2 = 0
#print "fig%d" % m2
while os.path.exists('fig_%d_%d_%d_%s_%d.png' % (size,iter,k,mult,m2)):
    print "T2"
    m2 = m2 + 1
plt.savefig('fig_%d_%d_%d_%s_%d.png' % (size,iter,k,mult,m2))
```

```
# wrapper function
def parallel(game):
    game.setup()
    value = game.play()
    if value is not None:
        return 1
    else:
        return 0


if __name__ == '__main__':
    game_parallel(int(sys.argv[1]), int(sys.argv[2]), float(sys.argv[3]),
        sys.argv[4])
```

---

stableroomate.py

For a given set of preferences, returns a stable solution if one exists.
This code was based on the free to use source code provided by the Coe College of Computer Science (2012) on their Github. This code was modified by the author and Kyle McDonnell to correct errors and make it usable for the application the author was considering.

```
#!/usr/bin/env python

import csv
import logging
import optparse
import random
import sys

log = logging.getLogger("stableroommate")


# log.setLevel(logging.INFO)


def readprefs(prefsfn):
    """
    read the preferences from "prefs.csv"
    """
    inner = csv.reader(open(prefsfn))

    prefs = {}

    for line in inner:
        if len(line) == 0: continue
        line = [s.strip() for s in line]
        prefs[line[0]] = line[1:]
```

```
    return prefs


def fillin(prefs):
    """
    some choices may not include everyone.

    fill in the rest by rearranging the others available at random.
    """

    names = set(prefs.keys())

    for name, choices in prefs.iteritems():

        left = set(names).difference([name]).difference(choices)

        if len(left) > 0:
            print("Filling in for", left)
            left = list(left)
            random.shuffle(left)
            choices.extend(left)


def checkprefs(prefs):
    """
    - `prefs`: preferences dict, name key, list of names as choices in order
    """

    names = set(prefs.keys())

    for name, choices in prefs.iteritems():
        try:
            assert len(names.difference(choices)) == 1
        except AssertionError, e:
            log.warning("len(names.difference(choices)) = {0} != 0".format(
                len(names.difference(choices))))
            log.warning(name, choices)
            log.warning(names.difference(choices))

            raise AssertionError(e)

        try:
            assert len(choices) == len(names) - 1
        except AssertionError, e:
            log.critical("len(choices) != len(names) - 1")
            log.critical("{0} != {1}".format(len(choices), len(names) - 1))
```

```
            log.critical("{0} {1}".format(name, choices))
            raise AssertionError(e)


def verify_ranks(ranks, prefs):
    """
    check that ranks and prefs correspond

    Arguments:
    - `ranks`: dict mapping name to rank index
    - `prefs`: preferences dict, name key, list of names as choices in order
    """

    for n in ranks:
        for m in ranks[n]:
            idx = ranks[n][m]
            try:
                assert m == prefs[n][idx]
            except AssertionError, e:
                log.critical("m != prefs[n][idx]")
                log.critical("{0} != {1}".format(m, prefs[n][idx]))
                raise AssertionError(e)


def reject(prefs, ranks, holds):
    """
    This does a reduction of the ranks if either of the following conditions
    holds.

    (i)

    (ii)
    """

    for y in holds:

        # n holds holds[n]
        i = 0
        x = holds[y]
        while i < len(prefs[y]):
            yi = prefs[y][i]

            try:
                if yi == x:
                    prefs[y] = prefs[y][:i + 1]

                # lower rank is better
```

```
                elif ranks[yi][holds[yi]] < ranks[yi][y]:
                    prefs[y].pop(i)
                    continue
                i += 1
            except KeyError:
                #print "bro"
                return -1


def find_all_or_nothing(prefs, ranks, holds):
    """
    Find an all or nothing cycle.

    Arguments:
    - `prefs`:
    - `ranks`:
    - `holds`:
    """
    p = []
    q = []

    # first find a key that has more than one pref left
    for x in sorted(prefs):
        if len(prefs[x]) > 1:
            cur = x
            break
    else:
        return None

    # trace through
    while cur not in p:
        # q_i = second person in p_i's list
        q.append(prefs[cur][1])

        # p_{i+1} = q_i' last person
        p.append(cur)
        cur = prefs[q[-1]][-1]

    a = p[p.index(cur):]
    b = [prefs[n][0] for n in a]

    return a


def kylephase1(prefs):
```

```
# Attempted proposals for each roommate
attempted_proposals = dict((name, 0) for name in prefs.keys())
# Set of currently promised pairs
promises = dict((name, None) for name in prefs.keys())

# Give each person a chance to propose to others
proposer_list = prefs.keys()
proposer = proposer_list[0]

# Propose to others in order of preference
while proposer is not None:
    # Remove this proposer from the list to propose if they are there
    if proposer in proposer_list:
        proposer_list.remove(proposer)

    # If this proposer has tried to propose to all preferences already, no
        solution exists!
    if attempted_proposals[proposer] == len(prefs[proposer]):
        # print("MATCHING FAILED (oh no!)")
        return promises # TODO or null?

    # Otherwise get the next proposal target and increment the number of
        attempted proposals
    target = prefs[proposer][attempted_proposals[proposer]]
    attempted_proposals[proposer] += 1

    # If the target is not promised, or they prefer the proposer, its a match
        made in heaven!
    existing_promise = promises[target]
    if existing_promise is None or prefs[target].index(proposer) <=
        prefs[target].index(existing_promise):

        # Give them a promise ring
        promises[target] = proposer

        # Let the rejected make new proposals.
        if existing_promise is not None:
            proposer = existing_promise
        # If there is no proposer, get the next one or simply stop
        elif len(proposer_list) > 0:
            proposer = proposer_list[0]
        else:
            proposer = None

#print promises

# If the loop finished, everyone made promise -- We have weddings to plan
```

```
    return promises


def phase1(prefs, ranks, curpref=None, debug=False):
    """
    perform phase 1 of the stable roomates problem.

    Arguments:
    - `prefs`: preferences dict, name key, list of names as choices in order
    - `ranks`: dict mapping name to rank index
    """

    # holds

    print prefs

    print ranks

    print "..................\n\n\n"

    holds = dict((name, None) for name in prefs.keys())

    print holds

    if curpref is None:
        curpref = dict((name, 0) for name in prefs.keys())

    print curpref

    people = prefs.keys()
    random.shuffle(people)

    proposed_to = set()

    # log.info("-- phase 1 -----------")
    # log.debug("{0}".format(people))

    # print "Entering loop 1"

    for person in people:
        poser = person

        print person
        while (1):

            # find poser someone
            # print "Entering loop 3"
```

```
        # print curpref

        while curpref[poser] < len(prefs[poser]):

            # person poser is proposing to
            nchoice = prefs[poser][curpref[poser]]
            curpref[poser] += 1

            # person poser is holding
            cchoice = holds[nchoice]
            # print ("{0} proposes to {1};".format(poser, nchoice))

            # lower ranking is better
            if cchoice is None or \
                    ranks[nchoice][poser] < ranks[nchoice][cchoice]:
                # print "Exit loop 3"
                break
            # print("{0} rejects {1};".format(nchoice, poser))

        # print("{0} holds {1}".format(nchoice, poser))
        holds[nchoice] = poser

        if nchoice not in proposed_to:
            # print "Exit loop 2"
            # log.warning("done")
            assert cchoice is None
            break

        # log.warning("and rejects {0}".format(cchoice))
        poser = cchoice

        # print poser
        #
        # print nchoice

    proposed_to.add(nchoice)

    # print holds
    #
    # print proposed_to
    #
    # print poser
    #
    # print cchoice

return holds
```

```
def newphase1(prefs, ranks, curpref=None, debug=False):
    """
    perform phase 1 of the stable roommates problem.

    Arguments:
    - `prefs`: preferences dict, name key, list of names as choices in order
    - `ranks`: dict mapping name to rank index
    """

    # holds
    holds = dict((name, None) for name in prefs.keys())
    # print holds

    if curpref is None:
        curpref = dict((name, 0) for name in prefs.keys())
    # print curpref

    people = prefs.keys()
    random.shuffle(people) # TODO Kyle says: no need to shuffle people

    proposed_to = set()

    # log.info("-- phase 1 -----------")
    # log.debug("{0}".format(people))

    # print "Entering loop 1"
    for person in people:
        poser = person

        while poser is not None:
            # find poser someone

            # While there are still people to consider proposing to for
            # this proposer, try to propose
            while True:

                # TODO Kyle added this. This is the correct condition
                # If the person has run out of people to propose to,
                # they are left alone and there is no stable solution!
                if curpref[poser] == len(prefs[poser]):
                    # print("MATCHING FAILED (oh no!)")
                    return holds

                # person poser is proposing to
                nchoice = prefs[poser][curpref[poser]]
                curpref[poser] += 1
```

```
            # person poser is holding
            held = holds[nchoice]
            # print ("{0} proposes to {1};".format(poser, nchoice))

            # If the proposer is better than the current choice's hold
            # of they don't have one, tentatively accept proposal
            # lower ranking is better
            if held is None or ranks[nchoice][poser] < ranks[nchoice][held]:
                # print("{0} rejects {1};".format(nchoice, held))
                break
            # print("{0} rejects {1};".format(nchoice, poser))

        # When the above loop ends, someone has accepted our proposal
        # Add this to the list of held proposals, and if someone became
            rejected,
        # allow them to make new proposals next
        # print("{0} holds {1}".format(nchoice, poser))

        previously_held = holds[nchoice] # TODO Kyle added this line.
        # TODO You didn't have access to this variable before and used it
            below

        holds[nchoice] = poser

        # TODO Kyle commented this out
        # if nchoice not in proposed_to:
        #     #print "Exit loop 2"
        #     # log.warning("done")
        #     assert held is None
        #     break

        # log.warning("and rejects {0}".format(cchoice))

        poser = previously_held
        proposed_to.add(nchoice) # TODO Kyle this line was indented
            incorrectly

    # print holds
    #
    # print proposed_to
    #
    # print poser
    #
    # print cchoice
return holds
```

```
def stableroomate(prefsfn, debug=False, parse=True):
    """
    find a stable roommate matching
    """

    #print "**********NEW***********"

    # read prefs from file
    # print "Reading preferences..."
    if parse:
        prefs = readprefs(prefsfn)
    else:
        prefs = prefsfn

    # print prefs

    # make sure everyone has the same number of choices
    # print "Filling in preferences..."
    fillin(prefs)

    # validate that names are correct
    # print "Validating preferences..."
    checkprefs(prefs)

    # generate a dictionary of rank values for each name
    # print "Generating dictionaries..."
    ranks = dict((idx, dict(zip(val, range(len(val)))))
                 for idx, val in prefs.iteritems())
    # print prefs
    # print(ranks)

    # validate the ranks correspond to the proper indices
    # print "Verifying ranks..."
    verify_ranks(ranks, prefs)

    # phase1
    # print "Phase 1..."
    holds = kylephase1(prefs)
    # print "Holds done"
    #print "HOLDS: ", holds

    log_holds(holds)

    error = reject(prefs, ranks, holds)
    #print "ERROR: ",error
    # print "Rejections done"
```

```
    if error == -1:
        #print("Returning none")
        return None

    cycle = find_all_or_nothing(prefs, ranks, holds)
    # print "Cycles found"

    if cycle is not None and len(cycle) == 3:
        # print "no solution exists"
        return

    ## phase 2
    # print "Phase 2"
    while cycle is not None:
        log.debug("-- cycle detected -----------")
        log.debug("{0}".format(cycle))

        curpref = {}
        for x in prefs:
            if x in cycle:
                curpref[x] = 1
            else:
                curpref[x] = 0

        holds = newphase1(prefs, ranks, curpref)

        log_holds(holds)

        error = reject(prefs, ranks, holds)

        if error == -1:
            return None

        cycle = find_all_or_nothing(prefs, ranks, holds)

    # print "***********RESULTS*********"

    # print prefs
    # print ranks
    return holds


def log_holds(holds):
    """

    Arguments:
```

```
    - `holds`:
    """

    log.info("-- holds -----------")
    for h in holds:
        log.info("{0} {1}".format(h, holds[h]))


def log_prefs(prefs):
    """

    Arguments:
    - `prefs`:
    """

    log.info("-- prefs -----------")
    for x in sorted(prefs):
        log.info("{0}\t{1}".format(x, " ".join(prefs[x])))


def swap_better(set1, set2, ranks):
    """
    """

    x1, y1 = set1

    x2, y2 = set2

    x1y1 = ranks[x1][y1]
    y1x1 = ranks[y1][x1]
    x2y2 = ranks[x2][y2]
    y2x2 = ranks[y2][x2]

    x1x2 = ranks[x1][x2]
    x2x1 = ranks[x2][x1]
    y1y2 = ranks[y1][y2]
    y2y1 = ranks[y2][y1]

    x2y1 = ranks[x2][y1]
    y1x2 = ranks[y1][x2]
    x1y2 = ranks[x1][y2]
    y2x1 = ranks[y2][x1]

    if x1x2 < x1y1 and x2x1 < x2y2 and y1y2 < y1x1 and y2y1 < y2x2:
        log.error("({0},{1}) ({2},{3}) -> ({4},{5}) ({6},{7})".format(
            x1, y1, x2, y2, x1, x2, y1, y2))
        log.error("({0},{1}) ({2},{3}) -> ({4},{5}) ({6},{7})".format(
```

```
            x1y1, y1x1, x2y2, y2x2, x1x2, x2x1, y1y2, y2y1))

    if x2y1 < x2y2 and y1x2 < y1x1 and x1y2 < x1y1 and y2x1 < y2x2:
        log.error("({0},{1}) ({2},{3}) -> ({4},{5}) ({6},{7})".format(
            x1, y1, x2, y2, x1, y2, x2, y1))
        log.error("({0},{1}) ({2},{3}) -> ({4},{5}) ({6},{7})".format(
            x1y1, y1x1, x2y2, y2x2, x1y2, y2x1, x2y1, y1x2))


def verify_match(matches):
    """
    """

    prefsfn = sys.argv[1]

    # read prefs from file
    prefs = readprefs(prefsfn)
    fillin(prefs)

    # generate a dictionary of rank values for each name
    ranks = dict((idx, dict(zip(val, range(len(val)))))
                 for idx, val in prefs.iteritems())

    for x in matches:
        for y in matches:
            if y == x or y == matches[x]:
                continue

            set1 = (x, matches[x])
            set2 = (y, matches[y])

            swap_better(set1, set2, ranks)


def main():
    """
    main function
    """

    # random.seed(1000)

    parser = optparse.OptionParser(usage="usage: %prog [options] prefsfn")
    parser.add_option("-v", dest="validate", action="store_true",
                      default=False, help="Validate the Algorithm")
    parser.add_option("-d", dest="debug", action="store_true",
                      default=False, help="Print Debuggin Code")
    (options, args) = parser.parse_args()
```

```
    if options.debug:
        logging.basicConfig(level=logging.DEBUG)

    if len(args) < 1:
        parser.print_help()
        return

    print "Finding matches..."

    # print args[0]

    matches = stableroomate(args[0], options.debug)

    if matches is not None:
        print("-- matches -----------")

        for m in matches:
            print "{0} {1}".format(m, matches[m])

        if options.validate:
            log.info("verifying matches...")
            verify_match(matches)


if __name__ == '__main__':
    main()
```