Fair Division of Cost via Matching of Goods

Somil Govani, Karthik Rajkumar, Justin Choi, Claudia Zhu

***Abstract*— Rental Harmony is a domain of algorithmic game theory problems motivated by the idea of dividing rent (cost) and rooms (goods) in a house shared by roommates. In this paper, we describe a more general form of this problem domain as well as various methods of assessing value and desiderata that are well-motivated. Then we go on to describe mechanisms that solve this problem under varying constraints and analyze the nature of the solutions that are produced. These solutions are adapted from a number of auction and fair division mechanisms that solve problems in this domain, but the analysis and methodology are expanded upon. Finally, this paper discusses experimental results from programmed implementations of a couple of these mechanisms.**

I. Introduction

A. Motivation

As this semester wraps up, many of us are anticipating moving into New York City apartments or back into Penn dorms. However along with this process, there is always the headache of who gets which room and how much to pay for that room. Luckily this dilemma is a common game theoretic problem that is a derivative of a fairness division problem, which are problems that concern dividing a set of objects between people. This housing predicament, known as Rental Harmony, arises when multiple tenants acquire a house with multiple rooms at a fixed cost of rent for the entire house. The tenants have preferences based on various features of the rooms and subsequently must collectively decide which tenant gets which room and how much each tenant must pay for each room such that the total cost of the house is accounted for.

This poses a matching problem that differs from those discussed in class. Although there is a fixed set of goods (i.e. the rooms), along with implicit valuation functions for each tenant, there is an important caveat that the payment vector cannot be assigned independently at each index. Namely, the sum of all payments must be equal to some fixed value. Moreover, the assignment itself is unique; each person needs to be assigned to exactly one room.

There are a number of mechanisms that address this issue, a number of which we will explore in this paper.

B. Paper Organization

The problem of Rental Harmony, which is a crucial part of fair allocation has been studied extensively through two main lenses: *ordinal utility* and *cardinal utility*. In this paper, we will first formally define a generalized version of this problem as well as general assumptions. Our analysis will cover four primary algorithms in both the cardinal and ordinal cases. We also include an experimental exploration of both Simplex Search and the Market-based methods.

II. Problem Statement

Consider a situation with n bidders and n objects. Denote the set of bidders as $\mathcal{B} = [1..n]$. Denote the set of objects as $\mathcal{O} = \{O_1, O_2, \ldots, O_n\}$. Let $C \in \mathbb{R}_{\geq 0}$ denote the total cost of $\mathcal{O}$ that needs to be allocated. The goal of this problem is to construct an assignment $a = (\mu, p)$ where $\mu : \mathcal{P} \to \mathcal{O}$ is a function assigning each bidder to an object and $p : \mathcal{P} \to \mathbb{R}_{\geq 0}$ is a payment function assigning a cost that each bidder must pay. The following criteria must be fulfilled for an assignment to be deemed feasible:

1) μ must be a bijection i.e. every object must be mapped to exactly one person and vice versa
2) The total cost C of $\mathcal{O}$ must be fully allocated i.e.

$$\sum_{i \in \mathcal{B}} p(i) = C$$

In some cases, it may be useful to define the prices on the objects rather than the people. In this case, we let $\phi : \mathcal{O} \to \mathbb{R}_{\geq 0}$ be a price vector on objects such that

$$\sum_{O \in \mathcal{O}} \phi(O) = C$$

We let our assignment be $a = (\mu, \phi \circ \mu)$. It is straightforward to verify that both problem formulations are equivalent.

A. Value Assessment

We present two potential ways to assess the utility derived by a particular bidder for a given assignment $a = (\mu, p)$. Both methods of assessment will be explored.

Ordinal Utility. Each bidder has a preference among accepting different objects at associated prices. Namely, each bidder i has a reflexive and transitive preference relation $\preccurlyeq_i$ over $\mathcal{O} \times \mathbb{R}_{\geq 0}$. Note that this implies prices are continuous. Given p, a bidder should be able to say which room(s) he weakly prefers to rent at that price.

Cardinal Utility. Each bidder has a real-valued valuation function $v_i : \mathcal{O} \to \mathbb{R}_{\geq 0}$. The utility of bidder i is defined as $u_i(a = (\mu, p)) \equiv v_i(\mu(i)) - p(i)$.

B. Desiderata

Here we will explore the different attributes that will make a solution interesting and feasible. Certain methods will lead to these desired criterion while others will not provably.

Non-Negativity. This condition states that all prices of rooms have to be greater than or equal to 0. This ensures that no tenant is being paid to take a room.

Envy-freeness. This desideratum is analogous to stability in previously studied problems. Namely, a given assignment $a = (\mu, p)$ is *envy-free* if for all bidders i we have that i

weakly prefers their object-price pairing to the object-price pairing of every bidder j. More concretely, using an ordinal evaluation, we have that

$$(\mu(j), p(j)) \preccurlyeq_i (\mu(i), p(i))$$

for all $i, j \in \mathcal{B}$. On the other hand, using a cardinal evaluation, we have that

$$u_i(a) \equiv v_i(\mu(i)) - p(i) \geq v_i(\mu(j)) - p(j)$$

i.e. bidder i's utility would not increasing by inheriting the bundle assigned to an alternative bidder.

Pareto Efficiency. In the context of this problem, we have that an assignment $a = (\mu, p)$ is *pareto efficient* if there does not exist another feasible assignment $a' = (\mu', p) \neq a$ (note that the price vector remains fixed) such that there exists some bidder i that strictly prefers a' while all bidders j weakly prefer a'. In the context of ordinal evaluation, we have that this means that for every alternate assignment a', there does not exist an i such that

$$(\mu(i), p(i)) \prec_i (\mu'(i), p(i))$$

while for every other $j \neq i$,

$$(\mu(j), p(j)) \preccurlyeq_i (\mu'(j), p(j))$$

In the context of cardinal evaluation, we have that this means that for every alternate assignment a', there does not exist an i such that

$$v_i(\mu(i)) - p(i) < v_i(\mu'(i)) - p(i)$$

while for every other $j \neq i$,

$$v_i(\mu(j)) - p(j) \leq v_i(\mu'(j)) - p(j)$$

Claim 1: Note that in both utility assessments, Envy-freeness implies Pareto efficiency.

Proof: We will prove the contrapositive that not Pareto efficient implies not envy free. Assume for some assignment that it is not Pareto Efficient but Envy Free. That is given some price vector and assignment a, let there be an alternate assignment a' such that it is strictly better for at least one tenant i. Then that tenant is envious since they have a strictly better assignment in a' vs a. ∎

However, note that in an alternate definition of pareto-efficiency in which $a' = (\mu', p')$ i.e. the price vector does not remain fixed, such a claim does not necessarily hold.

C. Manipulability of Mechanisms

We define *non-manipulability* or *truthfulness* or *strategyproofness* of a mechanism as follows. A cardinal mechanism $M : (\mathcal{B}, \mathcal{O}, C, V = (v_1, \ldots, v_n)) \mapsto a$, which outputs an assignment given an input problem, is *truthful* if no bidder $i \in \mathcal{B}$ can benefit from falsely reporting their valuation $v_i \in V$ as v_i'. Namely, we have that M is truthful if

$$u_i(M(\mathcal{B}, \mathcal{O}, C, V)) \geq u_i(M(\mathcal{B}, \mathcal{O}, C, (v_i', v_{-i}))$$

for all i. Note that a similar definition follows for the ordinal case. Namely, if we let $M' : (\mathcal{B}, \mathcal{O}, C, \preccurlyeq = (\preccurlyeq_1, \ldots, \preccurlyeq_n)) \mapsto a = (\mu, p)$ be some ordinal mechanism, we have that M' is truthful if for all $i \in \mathcal{B}$

$$(\mu'(i), p'(i)) \preccurlyeq (\mu(i), p(i))$$

where

$$a = (\mu, p) = M'(\mathcal{B}, \mathcal{O}, C, \preccurlyeq)$$

and

$$a' = (\mu', p') = M'(\mathcal{B}, \mathcal{O}, C, (\preccurlyeq_i', \preccurlyeq_{-i}))$$

for some non-truthful $\preccurlyeq_i'$.

Claim 2: There is no deterministic mechanism such that the fair allocation of goods and costs problem is not manipulable and also envy-free.

Proof: Consider the case when $\mathcal{B} = \{1, 2\}$ and $\mathcal{O} = \{O_1, O_2\}$ such that $n = 2$ with $C = 100$. Further, assume that we are in the cardinal case and the valuations are delineated by

TABLE I
COUNTEREXAMPLE VALUATION FUNCTION

	O_1	O_2
1	100	x
2	100	y

Without loss of generality assume that $0 < x < y < 100$. Let p_1 be the price given to O_1 and p_2 be the price given to O_2. Note that $p_1 = 100 - p_2$. For any price vector, since $x < y$, we have that an allocation is envy-free only if bidder 1 gets O_1 and bidder 2 gets O_2.

If the assignment is envy free, we have that

$$100 - p_1 = p_2 \geq x - p_2 \implies p_2 \geq x/2$$

Similarly, for an envy free assignment, we have that

$$100 - p_1 = p_2 \leq y - p_2 \implies p_2 \leq y/2$$

which means that $p_2 \in [x/2, y/2]$. If bidder 2 wishes to pay less, s/he can simply lower the value of y such that $y/2 < p_2$. Similarly, if bidder 1 wishes to pay less, s/he can simply increase the value of x such that $p_2 < x/2$. For any valid value of p_2, this must be possible for either bidder 1 or 2 and thus no mechanism can be truthful.

Note that any set of valuation functions can be translated to a valid preference relation since $\leq$ is both transitive and reflexive. This precise construction can be seen in Claim 5. As a result, this same proof holds for the ordinal case. ∎

1) Accounting for Manipulability via Price Ceilings: One method to make truthful mechanisms always possible is to incorporate price ceilings into the problem statement. Currently, our goal is to allocate a fixed cost C such that $\sum_{O \in \mathcal{O}} \phi(O) = C$. However, if instead we create a *price ceiling function* $PCF : \mathcal{O} \to \mathbb{R}_{\geq 0}$ such that

$$\phi(O) \leq PCF(O) \; \forall O \in \mathcal{O}$$

then replacing the fixed cost constraint with this one will lead to the existence of a truthful mechanism in all cases [1].

2) Accounting for Manipulability via Randomness: Note that Claim 2 proven above only held for deterministic mechanisms.

One way to easily make a truthful mechanism is to determine $p(i) = C/n$ for all $i \in \mathcal{B}$ and to randomly choose $\mu \sim \mathcal{B} \times \mathcal{O}$. In this case, clearly every person would be equally happy in expectation so no person would wish that they were in a different position. However, this would lead to allocations that were not envy-free in many cases.

There are more guarantees that can be achieved to alleviate this concern such as a minimum probability that the allocation is envy-free along with an expected number of envy-free partners; however, we deemed this to be out of the scope of this paper.

III. Ordinal Mechanisms

A. Optional Assumptions

There are some assumptions that are not implied by the problem statement; however some mechanisms will need to make these assumptions in order to work as desired and fulfill certain desiderata. These assumptions are mostly useful for the ordinal case.

Miserliness. For every bidder i, all $O_j, O_k \in \mathcal{O}$, and $x \in \mathbb{R}_{\geq 0}$ we have that

$$(O_j, x) \preccurlyeq_i (O_k, 0)$$

i.e. every bidder weakly prefers receiving an object for free. Even more strongly, if $x \in \mathbb{R}^+$, we have that

$$(O_j, x) \prec_i (O_k, 0)$$

i.e. every bidder strictly prefers receiving an object for free to receiving an object in exchange for a positive payment. This is somewhat analogous to the free-disposal assumption, but it is stronger in the sense that the bidder would accept even the worst good for free over the best good for any price $\epsilon > 0$.

Note that this assumption is slightly controversial as it is not always realistic. For example, a bidder might not want an object that is very bad even if it is free [2]. Francis Su, who made major developments in the Simplex Search via Sperners's Triangle mechanism, weakened this assumption by modifying it such that all bidder would never choose the most expensive object given that there is a free object available instead of enforcing the choice to be the free object. This modification will hold if a bidder's preferences dictate that they prefer a free object to an object costing at least $1/(n-1)$ of the total rent. However the modification might still fail especially in Cardinal utility cases.

Quality Goods. For any valid payment function p, each bidder finds at least one object-price pairing acceptable. This is implied by the fact that a maximum element must exist via the preference ordering; however, this further states that this maximum is always an acceptable outcome for the bidder.

Normalized Payment Vector. It may be a simplifying assumption to assume that the cost $C = 1$ and the payment function is refined to be $\hat{p} : \mathcal{P} \to [0, 1]$ such that we still have that

$$\sum_{i \in \mathcal{B}} \hat{p}(i) = C = 1$$

If this is the case, preference functions in the ordinal case do not change; however, valuation functions in the cardinal case may also need to be scaled by a factor of $1/C$ accordingly as well. This assumption will let us treat payment functions as vectors that are L_1-normalized.

B. Simplex Search via Sperner's Lemma

1) Describing the Space as a Simplex: Parts of this approach was first popularized by Francis Su as an adaptation of the fair division problem for cake-cutting [3]. Later in this paper, we discuss an implementation of this procedure in Python V-A. For this approach we will make the normalized payment vector assumption and fix $C = 1$. We can model our problem as an $(n-1)$-simplex $\mathcal{S}_{n-1}$ made up of the union of feasible payment vectors:

$$\mathcal{S}_{n-1} = \{\hat{p} : \sum_{i \in \mathcal{B}} \hat{p}(i) = 1, p_i \geq 0 \ \forall i\}$$

Geometrically, a 0-simplex is a point, a 1-simplex is a line, a 2-simplex is a triangle, and a 3-simplex is a tetrahedron. Note that the endpoints $\mathbf{E} = \{e_1, e_2, \ldots, e_n\}$ of an $(n-1)$ simplex are of the form

$$e_1 = (1, 0, \ldots, 0)$$

$$e_2 = (0, 1, \ldots, 0)$$

$$\ldots$$

$$e_n = (0, 0, \ldots, 1)$$

Each of these endpoints describes a payment vector in which all of the cost is allocated to one object. Our goal is to search this space to find an allocation that is envy-free and maximizes social welfare. Due to run time constraints of a feasible mechanism, it would not be plausible to search the entire space. Thus, it is in our best interest to first discretize the space represented by the simplex to reach an approximate solution.

2) Discretizing the Space via Barycentric Triangulation: This can be accomplished via barycentric triangulation. Namely, we define a *triangulation* $\mathcal{T}$ of an $(n-1)$-simplex $\mathcal{S}^*$ to be

$$\mathcal{T} = \{S : S \text{ is an } (n-1)\text{-simplex}\}$$

forming a partition such that

$$\bigcap_{S \in T} S = \emptyset \text{ and } \bigcup_{S \in T} S = \mathcal{S}^*$$

A triangulation can be found algorithmically by recursively computing the barycenters of all subsets of vertices. In particular we let

$$\mathbb{P} = \{\pi : [1..n] \to \mathbf{E}\}$$

be the set of all permutations on the endpoints of the simplex. Then for each permutation π_i we define an $(n-1)$-simplex made up of the endpoints

$$\mathbf{E}_i = \left\{ \frac{1}{m} \sum_{k=1}^{m} \pi_i(k) : \forall m, 1 \leq m \leq n \right\}$$

Although it is not readily apparent, the set of simplices S_i induced by endpoint sets $\mathbf{E}_i$ for each permutation $\pi_i \in \mathbb{P}$ form a valid triangulation $\mathcal{T}$ of an $(n-1)$-simplex. Furthermore, since there is a bijection between permutations of endpoints and subdivided simplices, we have that $|\mathcal{T}| = |\mathbb{P}| = n!$.

Moreover, because each element of $\mathcal{T}$ is itself an $(n-1)$-simplex, we can further apply barycentric subdivision to each of these recursively. As a result, after each iteration of subdivision, we have that the size of the triangulation increases by a factor of $n!$. Namely,

$$|\mathcal{T}_i| = (n!)^i$$

where $i \geq 0$ is the number of iterations of subdivision undergone. Due to how quickly this function grows with respect to n, we will likely not require many iterations of barycentric subdivision to be successful in discretizing the space with acceptable granularity for an ϵ-approximate solution.

3) Searching the Triangulation and Sperner's Lemma: Next arises the question of searching this discretization efficiently and more importantly whether or not a solution does exist in this approximation. We first label each of the endpoints in this triangulation with some $i \in \mathcal{B}$ such that no simplex has any two endpoints with the same label; we will refer to such a labeling as a *disparate* labeling. Call this labeling $\mathcal{L} : \mathsf{E} \to \mathcal{B}$ where

$$\mathsf{E} = \bigcup_{i}^{n!} \mathbf{E}_i$$

describes the set of all endpoints delineating this triangulation. Note that such a labeling must exist for a barycentric subdivision consisting of $k > 1$ iterations. After the kth iteration we can simply label each point with the cardinality of points of which the new endpoint is a function of. Namely, we let $\mathcal{L}(e) = x \in \mathcal{B}$ if and only if e is defined as

$$e = \frac{1}{x} \sum_{k=1}^{x} \pi_i(k)$$

For all permutations π_i that it is computed by. Further note that this means that $\mathcal{L}(e) = 1$ iff e was an endpoint of the simplex after the $(k-1)$th iteration of subdivision. Note that since each simplex is made up of exactly one point for each value of x, we quickly conclude that this is a valid disparate labeling.

Practically speaking this labeling will define the ownership of a vertex by a bidder. Namely, consider an allocation-independent ordering of objects such that the bidder that receives object O_i will pay $\hat{p}(i)$. Moreover, let $f : \mathsf{E} \to \mathcal{O}$; recall that each point in the simplex describes some payment function i.e. $e \in \mathsf{E}$ can be interpreted as a function. Define $f(e) = O_i$ if and only if for all $O_j \in \mathcal{O}$ such that $j \neq i$

$$(O_j, e(j)) \preccurlyeq_{\mathcal{L}(e)} (O_i, e(i))$$

If there are multiple such objects O_i, ties are broken arbitrarily. This is with the exception of the endpoints of the original pre-subdivision simplex $\{e_1, e_2, \ldots, e_n\} \equiv \mathbf{E} \subseteq \mathsf{E}$ where

$$f(e_i) = \begin{cases} O_n & i = 1 \\ O_{i-1} & i \neq 1 \end{cases}$$

i.e. the value of f is disparate for the original endpoints of the simplex (the standard vectors). We interpret f as the object that bidder $\mathcal{L}(e)$ (i.e. the owner of the endpoint) would prefer given the price allocation defined at e.

Recall that $\mathbf{E}_i$ describes the endpoints of one element of the triangulation. Assume that $f(e)$ is disparate for all $e \in \mathbf{E}_i$ for some i. By definition $\mathcal{L}(e)$ is also disparate. As a result, we interpret this as: for each price allocation defined by the endpoints of this sub-simplex, each bidder prefers a different object.

We let $\epsilon > 0$ define the largest distance between any $e, e' \in \mathbf{E}_j$ for any such j. In particular, we define $\mu(B \in \mathcal{B}) = O_B$ iff $\mathcal{L}(e) = B$ and $f(e) = O_B$ for some $e \in \mathbf{E}_i$. We arbitrarily choose any endpoint $e^* \in \mathbf{E}_i$ as our price allocation and output $a = (\mu, \mu \circ e^*)$ as our assignment. We have that each bidder is ϵ away from a price allocation in which the object assigned by μ is their (weakly) most preferred object i.e. if we are able to find such an $\mathbf{E}_i$ we are able to find an assignment that is *ϵ-approximate envy-free*.

***Claim** 3:* **Sperner's Lemma.** For any triangulation $\mathcal{T}$ of an $(n-1)$-simplex that fulfills the following constraints:

1) The endpoints of the original simplex are labeled disparately
2) The edges of the original simplex have the same label as one of their endpoints

we have that $\mathcal{T}$ must contain an odd number of subdvisions (i.e. sets of the form $\mathbf{E}_i$) such that the labels are disparate. In this case, the labeling in question is that produced by the function f. *Proof:* This will merely be a sketch of the proof since Sperner's Lemma is already well-regarded.

We describe a graph $G = (V, E)$ induced by the labeling of f. Namely, let there be a vertex associated with each subdivision $\mathbf{E}_i$ and a vertex associated with the area directly outside each $(n-2)$-simplex forming the border of the $(n-1)$-simplex. Further, let there be an edge between two vertices if and only if the $(n-2)$-simplex face that defines a border between the two vertices is labeled disparately with $\{1, \ldots, n-1\}$.

As a base case, note that any 1-simplex (i.e. a line) has disparate endpoints. As the labeling of the points along the line progresses there must be an odd number of alternations between the two labels of the endpoints (since the endpoint ends as a different value). By the induction hypothesis, we have that the $(n-2)$-simplices forming the border with endpoints $\{1, \ldots, n-1\}$ must also have an odd number of

such faces induced by the triangulation. As a result, there are an odd number of edges between the vertex outside the simplex and the vertices within the simplex (near the border).

By the Handshaking Lemma, we know that there are always an even number of vertices with odd degree in any undirected graph (since the total degree must always be even). As a result, we have that since the outer vertex has odd degree, there must be an odd number of vertices in the simplex with odd-degree.

Note that any sub-simplex in the triangulation must have at most 2 edges. If it had more, this means that the sub-simplex is made up of at least 3 disparately labeled faces which is a contradiction. As a result, the only odd-degree that an element of the triangulation can have is a degree of 1. However, if only one face is labeled with $\{1, \dots n-1\}$ then this means that the remaining endpoint must be n. As a result, this means that there must exist an odd number of elements in the triangulation such that the endpoints are labelled disparately. ■

Note that the labelling induced by f does fulfill the constraints of a Sperner labelling by definition and due to *miserliness* of bidders. As a result, we conclude that there are an odd number of $\mathbf{E}_i$ such that the labelling is disparate. Namely, there is at least 1 so an ϵ-approximate envy-free solution must exist.

4) Computing a Solution: To find such a solution, we can construct the graph G described in the proof of Sperner's Lemma. Then we simply conduct a depth first search starting from the outer vertex v. In one case, one branch of the depth first search will end at a vertex with degree 1 (i.e. a vertex representing a disparate sub-simplex). In another case, the branch will end in a vertex of degree 2 where the other edge leads back to v. Since every vertex has degree $0, 1$, or 2 inside the simplex, it cannot be the case that the search ends because the vertex still has additional neighbors (not v) that were already visited.

Since v has odd-degree, there must be at least one branch that does not terminate by leading back to v (as otherwise, each path contributes two edges and thus v has even degree). As a result, this search will always locate such a disparately labelled element of the triangulation.

Since any such triangulation can be made finer via further barycentric subdivision, we have a method to find an ϵ-approximate solution for any $\epsilon > 0$.

C. Azriely and Shmaya Generalization for Multi-Quantity Objects

In a generalization of this problem, μ need not be a bijection - namely, there could be multiple bidders that are assigned to the same object. The analogy for this is the idea that multiple roommates could live in the same room. Using the same assumptions as the Simplex Search protocol generalized from Su, there is a protocol developed by Azriely and Shmaya that always finds an envy-free assignment [4].

This generalization follows from the KKMS theorem (which follows from Sperner's Lemma) as well as Hall's Theorem.

D. Discussion of Ordinal Mechanisms

In both ordinal utility mechanisms, the preference relation of each bidder is allowed to depend on the entire price-vector. For instance, bidder 1 prefers object A if it costs 100 and then prefer object B to object C. However, if object A costs 700 then they prefer object C to object B [4].

These generalities are useful in the context of future planning, incomplete information, and irrationality effects. Consider the following:

Future Planning. Suppose the bidder thinks that object A is best, then B, then C. If A is expensive, the bidder settles on B. But if A is cheaper, the bidder might buy C (which is the cheapest), and then save some money and switch to A.

Incomplete information. The price-vector may give the bidder some indication on the quality of objects.

Similar Objects. The price-vector may allow the bidder to predict, to some extent, what kind of people are going to buy similar objects. More specifically, in the case of tenants, they might be able to predict what their neighbors are like.

Irrationality effects, e.g. framing effects. If object B and object C are of the same quality and have the same price, then the bidder may buy A. But, if object B becomes more expensive, then the bidder may switch to C, thinking that "it is the same as B but at a cheaper price".

IV. Cardinal Mechanisms

A. Accounting for Non-Negativity/Envy-Freeness Incompatibility

According to the problem statement, we have carefully defined the payment function p as part of a feasible allocation to have the co-domain $\mathbb{R}_{\geq 0}$ i.e. the payment for all bidders is non-negative. *Non-negativity* makes sense as an assumption since negative payments would imply that the mechanism would have to pay bidders (rather than vice-versa) which would not make sense in many scenarios.

***Claim** 4:* There may not exist a feasible assignment (which fulfills non-negativity by definition) that is envy-free.

Proof: Consider a cardinal scenario in which

$$\mathcal{B} = \{1, 2\}$$
$$\mathcal{O} = \{O_1, O_2\}$$

and the valuation functions are given by

TABLE II
Counterexample Valuation Function

	O_1	O_2
1	200	0
2	200	0

Assume that the total cost that needs to be allocated is $C = 100$. The maximum price for O_1 is 100. As a result, the minimum utility for O_1 for either bidder is $200 - 100 = 100$. The valuation for O_2 for both bidders is 0. As a result, the maximum utility for O_2 for either bidder is 0 (given

non-negativity). Regardless of the price allocation, both bidders will always strictly prefer O_1 rendering envy-freeness to be an impossible goal. ■

If we relax the constraint of non-negativity, it does become possible to reach an envy-free assignment. Namely, we choose an assignment in which the player assigned to O_1 pays 150 and the player assigned to O_2 pays -50. The utility for both players comes out to exactly 50 any element of $\mathcal{O}$ becomes weakly preferred. However, this only happens to be true for this case and scenarios in which $|\mathcal{B}| \leq 3$. In general, non-negativity and envy-freeness do not hold simultaneously when $|\mathcal{B}| \geq 4$ [2]. We can, however, make the following adjustment to the valuation function to get the desired outcome.

***Claim** 5:* Assume that the *miserliness* condition holds. This can be encapsulated in the cardinal utility framework with the following refinement of the utility function for each bidder $i \in \mathcal{B}$:

$$\hat{u_i}(a = (\mu, p)) = \begin{cases} \infty & p(i) = 0 \\ u_i(a) & p(i) \neq 0 \end{cases}$$

Note that $\hat{u_i}$ implicitly encapsulates the miserliness condition.

Proof: We proceed by reducing this problem to the original case. Note that the utility function in the cardinal context can be used to quickly define a valid preference ordering in the ordinal case. Namely, we say that

$$(O_k, x) \preccurlyeq_i (O_j, y) \iff \hat{u_i}(a) \leq \hat{u_i}(a')$$

where $a = (\mu, p)$ is any assignment such that $\mu(i) = O_k \in \mathcal{O}$ and $p(i) = x$ and $a' = (\mu', p')$ is any assignment such that $\mu(i) = O_j \in \mathcal{O}$ and $p(i) = y$. It is plain to see that envy-freeness holds in the ordinal case if and only if it holds in the cardinal case. As we saw earlier via Sperner's Lemma, since this preference relation is miserly, there must exist a feasible assignment that is envy-free, completing the reduction. ■

B. Market Simulation Based Mechanism

One approach that is both intuitive and highly performant for the cardinal case is to place the rooms on a market and adjust the prices depending on demand exhibited by the bidders. Thus, the notion of *demand* is important for this mechanism. This approach is adapted by a market-approach developed by Abdulkadiroglu and Sonmez and Unver [5]. Later in the paper we discuss an implementation in Python for this mechanism [V-B].

1) Idea Behind the Mechanism: First we define the notion of the *demand set* of a particular bidder. For simplicity, since this is a market-based approach, in this case, we define the price vector $\phi : \mathcal{O} \to \mathbb{R}_{\geq 0}$; note we can quickly get a price allocation over $\mathcal{B}$ given some allocation μ with the composition $p \equiv \phi \circ \mu$. The demand set for bidder $i \in \mathcal{B}$ for some price vector ϕ can be defined as

$$D_i(\phi) = \{O \in \mathcal{O} : u_i((\mu, \phi \circ \mu)) \geq u_i((\mu', \phi \circ \mu')) \\ \ni \mu(i) = O, \mu'(i) \neq O\}$$

for all feasible μ, μ'. We can visualize this market as a bipartite graph

$$G(\phi) = (\mathcal{B} \cup \mathcal{O}, \bigcup_{i \in \mathcal{B}} \{i\} \times D_i(\phi))$$

i.e. there is an edge between a bidder and all objects in his/her demand set. Our goal is to find a price vector ϕ^* such that the induced graph $G(\phi^*)$ contains a perfect matching. It is obvious that this solution would thus be envy-free. By Hall's Theorem, we know that there exists a perfect matching if and only if

$$\left| \bigcup_{i \in B} D_i(\phi) \right| \geq |B|$$

for all choices of $B \subset \mathcal{B}$. As a result, our goal quickly becomes to reduce the overlap of demand sets with respect to the number of bidders' demand sets being considered. This is intuitive - of course a more desirable scenario is one in which the fewest number of bidders have overlapping preferences; a similar paradigm was seen in the construction via Sperner's Lemma.

To solidify this intuition, we first define the notion of an *overdemanded* set. A set $S \subset \mathcal{O}$ is overdemanded if

$$|\{i \in \mathcal{B} : D_i(\phi) \subseteq S\}| > |S|$$

On its own, this definition is not very useful; note that an object $O \in \mathcal{O}$ that has no demand (i.e. $O \notin D_i(\phi)$ for all $i \in \mathcal{B}$) can be part of any overdemanded set S as long as

$$|\{i \in \mathcal{B} : D_i(\phi) \subseteq S\}| - 1 > |S|$$

To tighten this definition, we define a *minimal overdemanded set*. We say that a set $S \subset \mathcal{O}$ is a minimal overdemanded set if

1) S is overdemanded
2) S' is not overdemanded for all $S' \subset S$

Finally, we define a *full set of overdemanded rooms* F algorithmically as:

1) Let $F = \emptyset$
2) Compute the minimal overdemanded set S
3) Let $F = F \cup S$
4) Let the new demand sets of each bidder i be $D_i(\phi) - F$
5) Repeat steps $2 - 3$ until $S = \emptyset$

Note that in the case that $F = \emptyset \implies S = \emptyset$, we have that for all subsets $B \subseteq \mathcal{B}$ that $|B|$ is at most the union of their demand sets; otherwise, this set B would be a minimal overdemanded set. However, this is precisely the sufficient (and necessary) condition for a perfect matching to exist in Hall's Theorem. As a result, if we can find such a p^*, we can reach a feasible allocation.

In particular, given a perfect matching M^* induced by ϕ^*, the feasible envy-free assignment is given by $a = (M^*, \phi \circ M^*)$ where M^* can be interpreted naturally as an encoding of a bijective function.

2) Computing a Solution: To compute such a ϕ^*, we use a market based approach i.e. a descending and ascending auction - ascending for objects in the full overdemanded set and descending otherwise. The algorithm proceeds as follows:

1) Initialize $\phi(O) = C/n \quad \forall O \in \mathcal{O}$ (note that $\sum_{O\in\mathcal{O}} \phi(O) = C$)
2) Compute the full overdemanded set $F(\phi)$ of $\mathcal{B}$
3) If $F(\phi) = \emptyset$ we compute the maximum matching of $G(\phi)$ and return our assingment
4) Otherwise, we alter the prices to better represent the demand.

Mathematically speaking, let dx describe an infinitesimal change. We compute the change on price vector ϕ via composition with f (i.e. $f \circ \phi$) where f is defined as:

$$f(\phi)(o) = \begin{cases} \phi(o) + \dfrac{n - |F(\phi)|}{n} dx & o \in F(\phi) \\ \phi(o) - |F(\phi)|/n \, dx & o \notin F(\phi) \end{cases}$$

Note that the total price after transformation is

$$\begin{aligned} \sum_{o\in\mathcal{O}} f(\phi)(o) = &\sum_{o\in F(\phi)} \phi(o) + \frac{n - |F(\phi)|}{n} dx \\ &+ \sum_{o\notin F(\phi)} \phi(o) - \frac{|F(\phi)|}{n} dx = \\ &\left(\sum_{o\in\mathcal{O}} \phi(o)\right) + |F(\phi)| \frac{n - |F(\phi)|}{n} dx \\ &- |n - F(\phi)| \frac{|F(\phi)|}{n} dx = \\ &\boxed{\sum_{o\in\mathcal{O}} \phi(o) = C} \end{aligned}$$

As a result, we see that this is a valid price vector that allocates the cost completely. Moreover, note that the price on the overdemanded objects are increased while the price on the other objects are decreased. Similar to the format of a stock market, eventually this will converge on a fair price; even more importantly, there is a discrete choice of dx such that this process will converge in a finite number of steps [5]:

We define $J(\phi)$ as the set of bidders whose demand set is fully contained in the full overdemanded set:

$$J(\phi) = \{i \in \mathcal{B} : D_i(\phi) \subseteq F(\phi)\}$$

and further we choose dx for a particular price vector ϕ to be $dx(\phi) =$

$$\min_{j\in J(\phi)} \left(\widetilde{u_j}(\phi) - \max_{s\in\mathcal{O}-F(\phi)} u_j((\mu \ni \mu(j) = s, \phi \circ \mu)) \right)$$

where we define $\widetilde{u_i} : (\mathcal{O} \to \mathbb{R}_{\geq 0}) \to \mathbb{R}_{\geq 0}$ as the *best-case utility* i.e.

$$\widetilde{u_i}(\phi) = \max_{O\in\mathcal{O}} u_i((\mu \ni \mu(i) = O, \phi \circ \mu))$$

The proof for finite convergence can be found in [5].

3) Solution Guarantees: Earlier we showed that there does not always exist a solution that is both envy-free and non-negative in the cardinal case. This can be accounted for with the miserliness assumption as we've seen. Clearly, as stated before, this mechanism always converges to a envy-free solution i.e. it results in a perfect matching over the demand-set bipartite graph. Furthermore, if a non-negative solution exists, it will always be found. This is due to the fact that all potential positive pricings are searched before the first element of the price vector becomes negative [5].

C. Brams/Kilgour Gaps

Another mechanism proposed by Steven Brams and Marc Kilgour to solve the cardinal case of this problem is called the "Gap Procedure", in which they use a approach similar to the *Vickrey auction* that creates an assignment $a = (\mu, p)$ that is both non-negative as well as Pareto-efficient, but will not necessarily guarantee envy-freeness [6]. As such, we will use the comparison that valuations are analgous to bids in this section.

1) Motivation/Background: First, for the analysis of this mechanism, we won't be assuming that miserliness holds, hence why we cannot guarantee both non-negativity and envy-freeness. However, in the mechanisms following this one, we'll find that others have built upon the foundations of the Gap Procedure and have indeed found ways to guarantee both envy-freeness and non-negativity. For now, however, let's analyze the motivation behind this mechanism.

The reason why the mechanism was compared to the *Vickrey auction* in the introduction was due to the similarity in how, for a Vickrey auction, the highest bidder obtains the object but pays the bid of the second highest bidder; in this way, each bidder pays the minimum amount without making the second highest bidder envious. The idea that our mechanism uses to calculate an assignment $a = (\mu, p)$ is similar such that:

1) $\sum_{i\in\mathcal{B}} v_i(\mu(i))$ i.e. the sum of all valuations from bidders $\mathcal{B}$ is maximized for some feasible object allocation μ. We will define such an assignment as the *maxsum* assignment.
2) $\forall i \in \mathcal{B}$, $p(i)$ is not calculated solely from the valuation v_i bidder i places on a given $O_k \in \mathcal{O}$, but is additionally a function of $\{v_{i'} : \forall i' \in \mathcal{B} \text{ s.t. } v_{i'} < v_i\}$, where the more competitive the lower $v_{i'}$'s are (i.e. the more desired the room is), the closer the winner's $p(i)$ is to his/her valuation v_i.

With these desiderata in mind, what we'll prove is that the Gap Procedure ultimately ends up finding assignments a such that $p(\mu(i)) \leq v_i$ i.e. the bidder will pay at most their valuation, if not less than v_i in most cases, for each $i \in \mathcal{B}$, and a will ultimately be Pareto-optimal.

2) Algorithm: In order to find a maxsum assignment $a = (\mu, p)$, we'll be utilizing the Hungarian algorithm, a dynamic programming algorithm that runs in $O(n^3)$ time and computes the maxsum from an input matrix by permuting all rows and columns to maximize the trace of said matrix. Although this pseudocode represents the original algorithm

that runs in $O(n^4)$ time, the Jonker-Volgenant algorithm is a popular variant of this same algorithm that runs in $O(n^3)$ and performs the same task. We then write out the algorithm as follows:

1) Run the Hungarian algorithm on our valuation function matrix to retrieve the maxsum, which we'll denote as $s_{\max}$
2) If $s_{\max} < C$, then we declare that the problem is infeasible, as this means that the total $\sum_{i \in \mathcal{B}} p(i) < C$, i.e. the bidders are unwilling to pay the total cost.
3) If $s_{\max} = C$, then simply allocate according to the assignment $a = (\mu, p)$ that led to the output of $s_{\max}$, and each bidder respectively pays $p(i)$.
4) If $s_{\max} > C$, then we lower the p_i for each bidder via the *Gap Procedure* as follows: for each $O \in \mathcal{O}$, find the next lowest valuation and find the sum of all "next-lowest" valuations across all rooms. When descending down the valuation ordering, stop on a given room when the lowest valuation is reached, and if the sum across the current set of valuations we're keeping track of is still not less than 100, continue to descend on other $O' \in \mathcal{O}$ until the sum is indeed ≤ 100.
    a) If we stop our descent and our sum turns out to exactly $= 100$, we can finish here, and these set of valuations can simply become the payment vector for each bidder $i \in \mathcal{B}$. *Otherwise*, in the case that these don't add up, return to the next-highest sum from our iterations, and since this sum must be > 100 (as otherwise, our algorithm would've stopped at that sum instead), reduce each valuation comprising it in proportion to the "gaps" between it and the "next-lowest" valuation (i.e. the one we just ascended from) such that the reduced valuations sum up to 100. Hence, we then fulfill the $= 100$ condition, and we can cease the algorithm here.

To begin, let us consider a potential edge case: what about the case where the sum of lowest valuations for each room across all bidders still does not dip below the value of C? In their paper, Brams and Kilgour address this very concern: "we judge this case to be rather unlikely: how often would it occur that the lowest bids for the rooms are sufficient to pay the entire rent? Nonetheless, should this be the case, we assume one further descent - to bids of 0 for each room. Prices could then be determined according to step 3 on the basis of the gap between 0 and the minimum bids" [6] In our case, step "3" is actually step 4 of our denoted algorithm, but the same ideas still hold.

Another thing to note about the final step (i.e. the "gap" part of the Gap Procedure) is that it avoids the pitfalls of both equally reducing $p(i)$ for all bidders $i \in \mathcal{B}$ and proportionally reducing $p(i)$, as both methods simply reduce the highest v_i for each $O \in \mathcal{O}$ as a method of determining payment. In the first method of equally reducing, we can use the following valuation function as an example (to guarantee feasibility, we'll be adding the extra invariant that, for a given bidder $i \in \mathcal{B}$, $\sum_{O \in \mathcal{O}} v_i = 100$, i.e. their valuations for each object O adds up to 100):

TABLE III
EQUAL REDUCING OF HIGHEST VALUATION

	O_1	O_2	O_3
1	**90**	5	5
2	5	**90**	5
3	33	33	**34**

As we can see here, our total maxsum adds up to 214, so we have a surplus of 114 to work with. If we do equal reductions, $114/3 = 38$ is subtracted from each highest v_i for each room, so bidder 1 would pay 52, bidder 2 would pay 52, but bidder 3 would pay -4 for their room; this illustrates how using equal reductions cannot obviate negative payoffs, and this would obviously create conflict amongst all bidders, since 1 and 2 would be dissatisfied that they must pay bidder 3 to get the room that he/she most prefers!

For the idea of proportionally reducing rent (which does indeed guarantee non-negativity, as the amounts subtracted from each v_i are always $< v_i$, let us use a different valuation matrix below to illustrate its own pitfalls:

TABLE IV
EQUAL REDUCING OF HIGHEST VALUATION

	O_1	O_2	O_3
1	**40**	19	**41**
2	30	**40**	30
3	31	38	31

What we note here is that now, each object O_i's max valuation is not unique, as bidder 1 has the max valuation for both O_1 and O_3. If we sum our highest valuations, we get 121, and hence a surplus of 21.

Now, for the equal reduction method, once again we subtract $21/3 = 8$ from each of the highest valuations, the payments for each $O_i \in \mathcal{O}$ would be 32, 32, and 33 respectively. However, in this case, notice that O_1 and O_3 could not be assigned to anyone other than bidder 1, since 1's valuations of these objects are at lower-bounded by the payments. Thus, for whichever object bidder 1 receives, neither of the other bidders would be willing to pay for the remaining object, as its price would be higher than what they value it as. Thus, there exists no assignment $a = (\mu, p)$ such that all bidders don't pay more than their valuations i.e. don't suffer a loss (we'll denote this as the "no-loss constraint", which is also defined in Brams and Kilgour's paper [6]).

If we then consider the case of proportional reduction method, its shortcomings become apparent: we would reduce p_1 by 6.94, p_2 by 6.94, and p_3 by 7.12, but again, we face the same issue; both O_1 and O_3 could only be assigned to bidder 1 to fulfill the no-loss constraint, and hence, for whichever object is remaining, neither bidder 2 nor 3's valuation is at

least as high as the price of the object, and hence there exists no assignment such that the no-loss constraint is satisfied.

3) Solution Guarantees: One of the main guarantees of the Brams-Kilgour Gap Procedure is that it is both *non-negative* and *pareto-efficient*.

***Claim** 6:* Let $a = (\mu, p)$ be the assignment decided by this mechanism. We have that p is necessarily valid i.e. it is non-negative.

Proof: If $s_{max} < C$ no assignment is returned. On the other hand if $s_{max} = C$, the price is exactly equal to $v_i(\mu(i)) \geq 0$. If $s_{max} > C$, the correctness follows from the Gap Procedure. Namely, by the Gap Procedure, we always reduce the higher bid in the gap by an amount that is a fraction of the gap to get our price; this is the case since the sum of the lower bids in the gaps dip below C, so the sum of the gaps is larger than the difference between C and the sum of the higher bids. Since the reduction from the higher bids is proportional to the gap size, we have that the resulting price is always higher than the lower bid of the gap which is at least 0 by definition. ■

This assignment certainly fulfills pareto efficiency. Namely, for a fixed price vector an alternative allocation of objects would lead to a decrease in valuation for at least one bidder since the current allocation is a maxsum allocation. As a result, any alternative allocation would leave a bidder worse off.

However, as was mentioned before, Brams-Kilgour does not fulfill envy-freeness; as an example, refer to Property 4, Example 5. on pg. 429 of Brams paper [6], in which we find that one of the bidders could increase their surplus by being assigned to a different object, and hence envy-freeness doesn't hold. Yet, an interesting property that results from the Pareto-optimality of our solution is that envy can never be "two-way" i.e. if one player envies another, then the other will not envy the latter, as otherwise they both would benefit from trading, and hence contradict the Pareto-optimality of the solution. In essence, mutually beneficial trades of any kind will never happen within our assignment. An interesting thing to note is that, although envy-freeness is desirable, Brams and Kilgour make the interesting point that an auction-like mechanism is still preferrable as, intuitively, a bidder should indeed pay more when the valuations for a given object are competitive, even if this causes envy [6].

Additionally, although envy-freeness does not hold, we find that Brams-Kilgour actually holds to a *stronger* version of pareto-efficiency:

***Claim** 7:* Given an alternate assignment $a' = (\mu', p')$ in which both the allocation of bidders to objects μ' and the price vector p' change (note that in the weaker definition of pareto efficiency only μ' changed), all bidders still weakly prefer the current assignment a output by the procedure

Proof: First, consider the total surplus from the sum of valuations. As a property of the algorithm, the Gap Procedure inherently maximizes the sum of of valuations $\sum_{i \in \mathcal{B}} v_i(\mu(i))$, and hence maximizes the total surplus as well, since surplus is equal to $\left(\sum_{i \in \mathcal{B}} v_i(\mu(i))\right) - C$. Thus, in order for one or more bidders to do better than they currently are (i.e. increasing their surplus), at least one other bidder $j \in \mathcal{B}$ must do worse. However, as we showed when we were discussing envy-freeness, our Gap Procedure obviates any mutually-beneficial trades within any subset of bidders, and hence there are no trades within our assignment $a = (\mu, p)$ that would benefit some players without harming others. Thus, our Gap Procedure fulfills the definition of Pareto-optimality. ■

D. Sung and Vlach

1) Background: Although Brams and Kilgour's Gap Procedure [6] pioneered a fundamental procedure for Cardinal Utility in $O(n^3)$ that guarantees non-negativity, it does not guarantee envy freeness. As such, various algorithms have been developed building on worr done by Brams and Kilgour, utilizing methods ranging from linear programming to graph traversal algorithms.

One we found to be particularly interesting is Sung and Vlach's procedure that is both envy free and non-negative and is based on the Bellman Ford Algorithm. Consequently, the Sung and Vlach algorithm runs in $O(n^3)$, which is the same as the Brams and Kilgour Gap Procedure.

In short, the algorithm outputs a feasible envy free solution exists and outputs a solution where every roommate pays no more than their initial preference for room costs such that every envious housemate is assigned a room whose rent is zero if an envy free solution does not exist.

Note that this section is based on [7].

2) Notation: First we describe some notation that is useful for the description of the algorithm as well as some additional terminology to be used in the description and proof of this algorithm.

Again we have n bidders and n objects as above. Then we define B to be an $n \times n$ matrix where $B_{i,j}$ is bidder i's bid for object j. Note that every bidder must submit the costs for each of the n objects such that for bidder i,

$B_{i,1} + B_{i,2} + \cdots + B_{i,n} = C$ where $C \in \mathbb{R}_{\geq 0}$ denotes the total cost of $\mathcal{O}$ that needs to be allocated as above.

Again, p represents the payment scheme which must add up to c. Let $\mathcal{O} = \{1, \cdots, n\}, \mathcal{P} = \{1, \cdots, n\}$. If $\mu(i) = j$ define a *discount* to be the amount $B_{i,j} - p_j$.

An assignment $a = (\mu, p)$ is envy free if no bidder is envious. Under the notion of a *discount*, a bidder is *envious*, which holds under assumption of miserliness, if

$$B_{i,\mu(i)} - p_{\mu(i)} < B_{i,j} - p_j \quad \text{for some } j \in \mathcal{P}$$

This means that bidder i can get some room for a higher discount. Since this solution solves for cardinal utility, we have our utility function as $u_i(a = (\mu, p)) \equiv v_i(\mu(i)) - p(i)$. Note that we will approach the algorithm and proofs in this section using the definition of discounts.

3) Motivation: We define an *optimal assignment* or *maxsum assignment* as an assignment μ that satisfies:

$$\sum_{i=1}^{n} B_{i,\mu(i)} \geq \sum_{i=1}^{n} B_{i,\sigma(i)}$$

For all other room assignments σ.

We will now prove that an assignment is envy free if and only if it is also an optimal assignment.

Proposition: If assignment $a = (\mu, p)$ is envy free then μ is an optimal assignment.

Proof: If $a = (\mu, p)$ is envy free then we know that for all other room assignment σ and for all bidders i,

$$B_{i,\mu(i)} - p_{\mu(i)} \geq B_{i,\sigma(i)} - p_{\sigma(i)}$$

Now we will show that μ is an optimal assignment. We have the following:

$$\begin{aligned}\sum_{i=1}^{n} B_{i,\mu(i)} - \sum_{j=1}^{n} p_j &= \sum_{i=1}^{n} \left(B_{i,\mu(i)} - p_{\mu(i)}\right) \\ &\geq \sum_{i=1}^{n} \left(B_{i,\sigma(i)} - p_{\sigma(i)}\right) \\ &= \sum_{i=1}^{n} B_{i\sigma(i)} - \sum_{j=1}^{n} p_j\end{aligned}$$

■

Proposition: If some assignment $a' = (\sigma, p)$ is envy free then $a = (\mu, p)$ is envy free for every optimal assignment.

Proof: Given that a' is envy free and both μ, σ are optimal assignments (σ optimal from above proposition), we have that

$$\begin{aligned}\sum_{i=1}^{n} B_{i,\mu(i)} &= \sum_{i=1}^{n} B_{i,\sigma(i)} \\ \sum_{i=1}^{n} B_{i,\mu(i)} - C &= \sum_{i=1}^{n} B_{i,\sigma(i)} - C \\ \sum_{i=1}^{n} \left(b_{i\pi(i)} - p_{\pi(i)}\right) &= \sum_{i=1}^{n} \left(b_{i\sigma(i)} - p_{\sigma(i)}\right)\end{aligned}$$

Since a' is envy free, we have that for all bidders i

$$B_{i,\sigma(i)} - p_{\sigma(i)} \geq B_{i,\mu(i)} - p_{\mu(i)}$$

The above two equivalences give the following constraint:

$$B_{i,\sigma(i)} - p_{\sigma(i)} = B_{i,\mu(i)} - p_{\mu(i)}$$

So we can conclude that a is envy free as for all $i \in \mathcal{B}, j \in \mathcal{O}$, we have

$$B_{i,\mu(i)} - p_{\mu(i)} \geq B_{i,j} - p_j$$

Note that non-negativity follows by setting a' to be both envy free and non-negative. ■

Therefore, we see that for an arbitrary optimal assignment μ, there exists a pricing mapping p, such that $a = (\mu, p)$ is non-negative and envy free if such an assignment exists. This reduces our problem to be equivalent tot solving the weighted bipartite matching problem described above. We describe the following algorithm with runtime $O(n^3)$.

4) Algorithm: The algorithm we will describe maintains that no object is sold at a negative price, every bidder has a non-negative discount, and the object assignment maximizes the total discounts (no bidder is envious and thus the solution is envy free). The algorithm proceeds as follows:

1) Find an optimal assignment of rooms, μ
2) Find price vector, p that maximizes discounts and is envy free and non-negative. Note that $\sum_{i=1}^{n} p_i$ may not necessarily sum up to 1.
3) From here we have the following cases:
   a) If $\sum_{i=1}^{n} p_i = C$ then return $a = (\mu, p)$
   b) If sum of $\sum_{i=1}^{n} p_i < C$ then increase all the prices of all p_i's by $C - \sum_{i=1}^{n} p_i$ until the sum is C
   c) If $\sum_{i=1}^{n} p_i < C$ then there is no solution and we decrease the positive prices evenly, until the sum equals the total cost.

For step 1, we find an *optimal assignment* of rooms in $O(n^3)$. This can be done by looking at the sum of all n possible room assignments for each of the n players and then comparing the sums to select the maxsum assignment.

For step 2, we find such a price vector p using the Bellman Ford algorithm described below:

Algorithm 1 Bellman-Ford Algorithm for Finding Pricing Scheme

Given B, μ
Output envy free, non-negative pricing scheme that maximizes discounts
Initialize $p = [0, 0, \cdots, 0]$
loop
The following n times
for $i, j = 1, 2, \cdots, n$ **do**
if $p_j < p_{\mu(i)} - B_{i,\mu(i)} + B_{ij}$ **then**
Then set $p_j \leftarrow p_{\mu(i)} - B_{i\mu(i)} + B_{ij}$
end if
end for
end loop
Return p

We use this algorithm, which initializes the payment vector as 0 and adds only positive amounts to the vector because every envy free solution of some B, C exists if and only if for every optimal assignment μ, there is a payment scheme, p^* satisfying $\sum_{j=1}^{n} p_j^* \leq C$ [7]. Further, doing so guarantees non-negativity.

Now onto step 3, we branch into 3 possibilities. If it is the case that the pricing scheme sums up to 1, then we know that it is feasible, non-negative, envy free and that the assignment is optimal. Therefore, we return $a = (\mu, p)$ and we are done.

If the sum of the pricing scheme is less than the total cost, C, then we will divide the remaining amount evenly between the remaining players. The resulting payment scheme is feasible, non-negative, but we need to verify that it is envy free. We do so below:

Proof: Given that $a = (\mu, p)$ is envy free, then we have for all $i \in \mathcal{B}, j \in \mathcal{O}$, if we have some arbitrary number q (note that it could be negative), the following holds:

$$\begin{aligned} B_{i,\mu(i)} - p'_{\mu(i)} &= B_{i,\mu(i)} - p_{\mu(i)} - q \\ &\geq B_{i,j} - p_j - q \\ &= B_{i,j} - p'_j \end{aligned}$$

This shows that we can allocate some constant amount to all bidders and still maintain that the pricing scheme is envy free. ■

Otherwise, in the case of 3d, we find the smallest value of q satisfying:

$$\sum_{j=1}^{n} \max(0, p_j - q) \leq C$$

This will take upwards of $O(n^2)$ time since in the worst case, we reduce the payment cost of only one bidder, so we must eliminate $n-1$ other bidders. After each elimination, we then take the sum again, which takes n time. Giving us an $O(n^2)$ solution. Let m be the number of objects in $\mathcal{O}$ such that $p^*_j \geq q$. Let r be $C - \sum_{k=1}^{n} \max\{0, p_k - q\}$, or the amount that the bidders are short from C after deducting q from each bidder paying more than q. Then we define our new pricing scheme as

$$p^*_j = \begin{cases} p_j - q + \frac{r}{m} & \text{if } p^*_j \geq q \\ 0 & \text{otherwise} \end{cases}$$

We see that p^* is non-negative for all values because we only deduct from the payment of an object if the original payment is greater than the amount that we deduct. Further, this pricing scheme sums up to C. We see this as: $\sum_{j=1}^{n} p^*_j = \left(\sum_{j=1}^{n} \max\{0, p_j - q\}\right) + m\left(\frac{c - \sum_{j=1}^{n} \max\{0, p_j - q\}}{m}\right) = C$ Such a q must always exist because in the worst case, we deduct from the bidder paying the maximum payment. Since we make the assumption that payments are continuous, then such a q will always exist.

Finally, we see that no bidder who pays for their object is envious. We know that p^*, μ is envy free. Since if $p^*_{\mu(i)} \geq q$ then for all bidders $i \in \mathcal{B}$ and objects $j \in \mathcal{O}$, we have:

$$\begin{aligned} B_{i,\mu(i)} - p_{\mu(i)} &= B_{i,\mu(i)} - \left(p^*_{\mu(i)} - q + \frac{r}{m}\right) \\ &\geq B_{i,j} - \left(p^*_j - q + \frac{r}{m}\right) \\ &\geq B_{i,j} - p_j \end{aligned}$$

Since the above only applies to bidders whose original payments are greater than q, all other bidders that are not guaranteed to be envy free are paying 0 for their objects.

Sung and Vlach's solution are guaranteed to be non-negative and envy free when possible. If not, it will return a solution that is envy free for all paying bidders. Further, the algorithm runs in $O(n^3)$ time, a significant improvement on previous solutions.

V. Simulated Experiments

In order to prove the concepts behind these mechanisms and to show that processes exist to compute them efficiently, code implementing the aforementioned mechanisms was written. All relevant code is attached in the appendix; alternatively, the code is available at https://github.com/somilgo/rentalharmony

A. Simplex Search via Sperner's

1) Algorithm: We continue our discussion of this mechanism [III-B] which finds envy-free ϵ-approximate solutions (for arbitrarily small ϵ) in the ordinal utility case. As aforementioned, the steps of the simplex search algorithm are

1) Construction of the simplex space and delineation of bidder preferences
2) Barycentric subdivision of the simplex space
3) Construction of graph overlay in the subdivided simplex space
4) Assigning owner labels in a disparate fashion
5) Recursively computing the disparately labeled elements of the triangulation
6) Choosing a best solution via barycenter approximation

The first step is to indeed delineate the constraints of our problem. This happens by setting the value of n which will induce the endpoints of our $(n-1)$-simplex in n-dimensional space. Furthermore, given a parameter n, there needs to be n bidders that are delineated where each bidder has a preference function over all pairs of objects at a given price. Creating such a pairwise function may be space inefficient and non-transitive in most cases if not computed correctly. As a result, we simply gave the bidders a vector of values such that the sum of the values is equal to C. This way, the utility function given a price induces a transitive preference relation for the bidders that is easily computed.

The next step is to triangulate the simplex space via barycentric subdivision. This is done by generating all permutations of the parents simplex's vertices and computing the barycenter of the first m vertices as m increases. This is recursively repeated for k iterations, a value that is set at the start of the program.

Following triangulation, each subdivision is associated with a vertex in the graph overlay. Then edges are created between adjacent sub-simplices using a locality-sensitive hash map for the vertices that are shared by neighboring faces.

Owner labels are assigned via a breadth first search starting from a single sub-simplex. The first sub-simplex is arbitrarily (yet disparately) labeled. Each subsequent vertex (i.e. sub-simplex) in the BFS has all but one of its vertices' labels fixed by the previous neighbor; the final vertex can only take one value which is assigned appropriately.

Finally, the algorithm will recursively compute the valid edges along the border (via the induction hypothesis in Sperner's Lemma's proof). Then a DFS will be conducted along all valid edges to compute every sub-simplex that is disparately desired by the bidders (using their preference

relations laid out before). In one case, the goal may be to find any envy-free solution as quickly as possible. To do this, one may wait until the top level of the recursion is being processed and only further subdivide into simplices that are disparately labeled. If this is the case then the order of subdivision is no longer exponential. Mathematically, this will still always yield a solution since the sub-simplex might as well be considered the main simplex with disparately labeled corners.

At this point we have a set of envy-free approximate solutions to the problem so we are done. It may be useful to find the *best* solution from this set. First, we may hope that taking the barycenter of the n points of the solution simplices will lead a single solution that is envy-free. Then we can compute the social welfare (since we have valuation functions) as well as total utility to accept the solution with the highest social welfare.

2) Results: One example of a delineation of bidder preference relations is given by:

```
[
    Bidder(0,[0.34, 0.35, 0.33]),
    Bidder(1,[0.5, 0.20, 0.35]),
    Bidder(2,[0.75, 0.2, 0.05])
]
```

Listing 1. Delineation of Preferences for n=3 bidders

where the first argument is the label associated with the bidder (for owner ship) and the second argument is a list of preferences for each object (the element at the ith index is associated with the value for the ith room) which sums to $C = 1$. Clearly in this case, $n = 3$.

Computing this delineation for $k = 6$ subdivisions (i.e. $|\mathcal{T}| = 46656$), we get the following results:

```
Creating Graph...
Subdividing...
Done Subdividing!
Done creating graph!
Labelling the Sperner Triangle...
Done Labelling!
Solving for approximate solution...
Solved! Computed 1757 solutions...
Best Solution:
[0.4985282, 0.2015889, 0.2998828]
with (social welfare, total_utility):
(1.45, 0.4500001)
[Finished in 4.1s]
```

Listing 2. Results of 3 bidders with 6 subdivisions

In this case, the barycenter of one of the solutions was indeed a best solution as we had hoped and it was envy-free. It achieved a social welfare of 1.45.

B. Market Based Mechanism

1) Algorithm: We continue our discussion of this mechanism [IV-B] which find envy-free solutions to the cardinal utility case which are always non-negative if such a non-negative solution exists. The steps of this algorithm are more or less fully delineated in the section about computing a solution in the previous discussion [IV-B.2].

In this Python implementation, the challenges that remained were as follows:

1) Computing a maximum matching in the bipartite graph $G(\phi)$ induced by ϕ and the demand sets of the bidders
2) Efficiently computing the minimal overdemanded set (leading to an efficient computation of $F(p)$, the full overdemanded set)

i.e. the highest level of the algorithm is given by:

```
def conduct_discrete_price_auction(bidders, C):
    n = len(bidders)
    prices = [C/n] * n
    while True:
        for bidder in bidders:
                bidder.recompute_demand_set(prices)
        od = compute_ODS(bidders)
        for bidder in bidders:
                bidder.recompute_demand_set(prices)
        if len(od) == 0:
                return compute_maximum_matching(bidders, n), prices
        x = compute_price_differential(od, bidders, prices)
        for r in range(n):
                if r not in od:
                        prices[r] -= len(od) * x / n
                else:
                        prices[r] += (n - len(od)) * x / n
```

Listing 3. Python code for high-level market mechanism

where the functions `compute_maximum_matching` and `compute_ODS` will be discussed next.

2) Maximum Matching in a Bipartite Graph: One method to construct a maximum matching in a bipartite graph is via the constructive proof of Hall's Thereom. Namely, this process involves finding *Hall Violators* and further saturating M-augmenting paths of the incomplete matching.

The approach used in this implementation was via a representative flow network and computing the max flow via the Ford-Fulkerson algorithm. The flow network was produced on

$$G(p) = (\mathcal{V} \equiv \mathcal{B} \cup \mathcal{O}, \mathcal{E} \equiv \bigcup_{i \in \mathcal{B}} \{i\} \times D_i(\phi))$$

as follows. Namely we first compute

$$G' = \left(\mathcal{V} \cup \{s,t\}, \mathcal{E} \cup (\{s\} \times \mathcal{B}) \cup (\{t\} \times \mathcal{O}) \right)$$

i.e. we add a source vertex s with an edge to each bidder and a target vertex t with an edge to each object. We let the capacity of each edge be 1 and search only for integral solutions. It is well known that the min-cut of a maximum flow solution (computed by Ford-Fulkerson) is also a maximum matching M. If $|M| = |\mathcal{B}| = |\mathcal{O}|$ then we have that $M = M^*$ is also a perfect matching. As we showed before, a perfect matching also quickly defines an envy-free feasible assignment.

3) Computing the Minimal Overdemanded Set: The simplest solution in this case is to simply compute every possible subset of objects and check the overdemanded set property for each. This can be done easily on a per set basis. After doing so, the sets with minimum cardinality fulfilling the

property are returned. However, this is clearly an inefficient exponential algorithm.

A polynomial algorithm for computing the MODS is given in [8]. This method leverages the nature of maximum matchings to take an arbitrary over-demanded set and culls it down to a minimal one with the knowledge that no MODS if a perfect matching exists.

4) Results: One possible delineation of the valuation function for each bidder (where $\mathcal{B} = [0..5]$ and $\mathcal{O} = \{O_0, \ldots, O_5\}$) is

TABLE V
VALUATION MATRIX FOR N=6

	O_0	O_1	O_2	O_3	O_4	O_5
0	15	18	10	15	24	28
1	18	25	3	18	25	15
2	6	25	15	18	18	25
3	15	5	18	12	9	25
4	6	22	5	5	10	12
5	6	9	2	21	25	9

The output of the algorithm is as follows. Note that at each iteration we output both the current price vector along with the current full overdemanded set.

```
Price on iteration 1:
[10.0, 10.0, 10.0, 10.0, 10.0, 10.0]
FODS on iteration 1:
{1, 4, 5}
Price on iteration 2:
[8.0, 12.0, 8.0, 8.0, 12.0, 12.0]
FODS on iteration 2:
{1, 5}
Price on iteration 3:
[7.0, 14.0, 7.0, 7.0, 11.0, 14.0]
FODS on iteration 3:
{1, 3, 4, 5}
Price on iteration 4:
[5.0, 15.0, 5.0, 8.0, 12.0, 15.0]
FODS on iteration 4:
set()
The envy-free assignment is given by:
mu = (3,2),(5,4),(0,5),(2,3),(1,0),(4,1)
p = [5.0, 15.0, 5.0, 8.0, 12.0, 15.0]
[Finished in 0.1s]
```

Listing 4. Output for cardinal example of market mechanism

As it is seen in this example, a perfect matching is only achieved when the full overdemanded set is empty.

VI. CONCLUSION

In conclusion, the problem presented in this paper is well motivated by the Rental Harmony problem of splitting up rooms and rent to roommates living in a house. As we saw, there are a number of varying assumptions that can be made that change the nature of the mechanisms as well as the nature of the solutions. It is interesting to see how many of the assumptions can be motivated in various ways. For example, this paper explored the trade-offs between using an ordinal versus cardinal approach to value assessment.

Moreover, we were able to explore the relationship between different desiderata. Namely, we expressed paramount interest in the notion of *envy-freeness* which turned out to be strictly stronger that pareto-efficiency, a common measure of efficiency. Furthermore, we found an incompatibility between envy-freeness with truthfulness and non-negativity of payment allocation which we were able to account for in various ways through our solution.

Finally, it is astounding to see the number of lenses through which such a simple problem can be viewed. It was interesting to see this problem solved through search across a discretized simplex as well as an adaption to market economics of a rising and falling auction. This was indeed manifested through the implementation of these algorithms which proved the true constructivity and feasibility of the aforementioned mechanisms. As we discussed in class, from a mathematical point of view it is important that a mechanism has guarantees such as existence and optimality; however it was even more so interesting to see solutions that could be implemented efficiently and constructively in polynomial time.

If anything this exploration left us with a taste of what this very specific problem domain has to offer. While conducting this research we also realized that there exist a number of modifications that can be made to this problem (beyond the ones we studied) that can lead to even more interesting desiderata.

REFERENCES

[1] Z. Sun, Ning; Yang, "Ca general strategy proof fair allocation mechanism," *Economics Letters*, vol. 81, 2003.

[2] M. Kaminski, "Mathematics and democracy: Designing better voting and fair-division procedures by steven j. brams," *Political Science Quarterly*, vol. 124, 03 2009.

[3] F. E. Su, "Rental harmony: Sperner's lemma in fair division," *The American Mathematical Monthly*, vol. 106, no. 10, pp. 930–942, 1999.

[4] E. Azrieli, Yaron; Shmaya, "Rental harmony with roomates," *Journal of Economic Theory*, vol. 153, 2014.

[5] A. Abdulkadiroglu, T. O. Sonmez, and U. Unver, "Room assignment-rent division: A market approach," 2002.

[6] S. J. Brams and D. M. Kilgour, "Competitive fair division," *Journal of Political Economy*, vol. 109, no. 2, pp. 418–443, 2001.

[7] M. Sung, Shao Chin; Vlach, "Competitive envy-free division," *Social Choice and Welfare*, vol. 23, 2004.

[8] W. Huang, J. Lou, and Z. Wen, "Allocating indivisible resources under price rigidities in polynomial time," *CoRR*, vol. abs/1405.6573, 2014.

Appendix

Python interface for creating and interacting with bidders

```python
import random


class Bidder:

    def initialize_valuations(self, valuations):
        self.n = len(valuations)
        self.valuations = valuations

    def initialize_simplex_valuations(self, n):
        self.n = n
        random_list = []
        for i in range(n-1):
            random_list.append(random.random())
        random_list.append(0.)
        random_list.append(1.)
        list.sort(random_list)
        valuations = []
        for i in range(len(random_list)-1):
            valuations.append(random_list[i+1] - random_list[i])
        self.valuations = valuations

    def initialize_random_valuations(self, n, max_price_per_room = 1.0):
        self.n = n
        valuations = []
        for i in range(n):
            valuations.append(max_price_per_room * random.random())
        self.valuations = valuations

    def initialize_random_int_valuations(self, n, max_price_per_room = 100):
        assert(int(max_price_per_room) == max_price_per_room)
        self.n = n
        valuations = []
        for i in range(n):
            valuations.append(random.randint(0,max_price_per_room))
        self.valuations = valuations

    def recompute_demand_set(self, prices):
        max_utility = -1e9
        demand_set = []
        n = self.n
        self.prices = prices
        for i in range(n):
            curr_utility = self.valuations[i] - self.prices[i]
            if curr_utility > max_utility:
                max_utility = curr_utility
                demand_set = [i]
            elif curr_utility == max_utility:
                demand_set.append(i)
        self.demand_set = set(demand_set)

```

```
    def indirect_utility(self, prices):
        max_utility = -1e9
        n = self.n
        for i in range(n):
            curr_utility = self.valuations[i] - self.prices[i]
            if curr_utility > max_utility:
                max_utility = curr_utility
        return max_utility

    def utility(self, i, prices):
        return self.valuations[i] - self.prices[i]


class BidderGroup:
    def __init__(self, n, valuation_scheme="simplex", max_val=0):
        bidders = [Bidder() for i in range(n)]
        for b in bidders:
            if valuation_scheme == "simplex":
                b.initialize_simplex_valuations(n)
            elif valuation_scheme == "random":
                b.initialize_random_valuations(n, max_price_per_room=max_val)
            elif valuation_scheme == "random_int":
                b.initialize_random_int_valuations(
                    n, max_price_per_room=max_val)
        self.bidders = bidders
```

Appendix

Python implementation for Simplex Search via Sperner's Lemma

```
import numpy as np
import itertools
import copy
from functools import partial

class Edge:

    def __init__(self, neighbor, data):
        self.neighbor = neighbor
        self.data = data

    def __str__(self):
        return str(self.data)
    def __unicode__(self):
        return str(self.data)
    def __repr__(self):
        return "Edge: " + str(self.data)

class Vertex:

    def __init__(self, coords):
        self.coords = coords
        self.edges = []

    def add_edge(self, neighbor, data):
        self.edges.append(Edge(neighbor, data))
        neighbor.edges.append(Edge(self, data))

    def __hash__(self):
        if self.coords == None:
            return 0
        return hash(frozenset(self.coords))

    def __eq__(self, other):
        return hash(self) == hash(other)

    def __str__(self):
        return str(self.coords)
    def __unicode__(self):
        return str(self.coords)
    def __repr__(self):
        return "Vertex: " + str(self.coords)

class Bidder:

    def __init__(self, name, valuation):
        self.name = name
        self.valuation = valuation

    def utility(self, prices, i):
        return self.valuation[i] - prices[i]
```

```
52
53      def pref(self, prices):
54          show=False
55          for p in range(len(prices)):
56              if prices[p] == 1:
57                  out = len(prices)-1 if p==0 else p-1
58                  return out
59              if prices[p] != 0:
60                  break
61          for i in range(len(prices)):
62              if prices[i] == 0:
63                  return i
64          utility = [value - price for value, price in zip(self.valuation, prices)]
65          return utility.index(max(utility))
66
67  def is_coplanar(v, vertices):
68      vectors = [v - vertex for vertex in vertices]
69      rank = np.linalg.matrix_rank(np.array(vectors))
70      return rank <= 2
71
72  def generate_graph(sub_divs):
73      verts = {}
74      vertstest = {}
75      n = len(sub_divs[0])
76      count = 0
77      for sub_div in sub_divs:
78          sub_div = [tuple(round(x, 7) for x in y) for y in sub_div]
79          count += 1
80          new_vert = Vertex(sub_div)
81          for x in range(n):
82              hashed_edge = frozenset([sub_div[a] for a in range(n) if a != x])
83              for vert in verts.get(hashed_edge, []):
84                  new_vert.add_edge(vert, hashed_edge)
85              if verts.get(hashed_edge):
86                  verts[hashed_edge].append(new_vert)
87              else:
88                  verts[hashed_edge] = [new_vert]
89      return verts
90
91
92  def subdivide_wrapper(simplex_points, iterations):
93      def subdivide(p, iterations):
94          if iterations == 0:
95              return [p]
96          n = len(p)
97          dim = len(p[0])
98          perms = itertools.permutations(p)
99          sub_divs = []
100         for perm in perms:
101             sub_verts = []
102             for i in range(n):
103                 v = [0] * dim
104                 for j in range(i+1):
105                     v = [v[k] + perm[j][k] for k in range(dim)]
106                 v = tuple([x / (i+1) for x in v])
107                 sub_verts.append(v)
```

```
            sub_divs.append(sub_verts)

        all_sub_divs = []
        for sub_div in sub_divs:
            all_sub_divs += subdivide(sub_div, iterations-1)
        return all_sub_divs

    print("Subdividing...")
    sub_divs = subdivide(simplex_points, iterations)

    print("Done Subdividing!")
    return generate_graph(sub_divs)

def assign_owners(start_vert, gg):

    n = len(start_vert.coords)
    owner_labels = {}
    for label, coord in zip(range(n),start_vert.coords):
        if owner_labels.get(coord):
            raise BaseException
        owner_labels[coord] = label

    labeled = set()
    labeled.add(start_vert)
    to_label = []
    for e in start_vert.edges:
        to_label.append(e.neighbor)
    while len(to_label) > 0:
        vert = to_label.pop(0)
        if vert in labeled:
            continue
        unlabeled_coord = None
        found_unlabeled = False
        unused_labels = set(range(n))
        for coord in vert.coords:
            label = owner_labels.get(coord)
            if label == None:
                if found_unlabeled:
                    raise BaseException
                unlabeled_coord = coord
                found_unlabeled = True
            else:
                unused_labels.remove(label)

        if found_unlabeled and (len(unused_labels) == 1):
            owner_labels[unlabeled_coord] = unused_labels.pop()
        labeled.add(vert)
        for e in vert.edges:
            if e.neighbor not in labeled:
                to_label.append(e.neighbor)

    return owner_labels
def traverse_trap_doors(graph, owner_labels, bidders, brute_force = False):
    n = len(next(iter(graph.values()))[0].coords)

    def get_trap_doors():
```

```
        trans_owner_labels = {}
        trap_doors = []
        for i in range(n):
            border_simplex = []
            trunc_bidders = []
            for edge_set in graph:
                if all([c[i]==0 for c in edge_set]):
                    border_simplex.append(edge_set)
            simplex_graph = generate_graph(border_simplex)
            output = traverse_trap_doors(simplex_graph, owner_labels, bidders)
            for o in output:
                trap_doors.append(frozenset(o.coords))
        return trap_doors
    def check_if_trap_door(edge):
        unused_prefs = set(range(n))
        for coord in edge.data:
            pref = bidders[owner_labels[coord]].pref(coord)
            if pref in unused_prefs:
                unused_prefs.remove(pref)
        return len(unused_prefs) == 1

    def check_if_disparate(vertex):
        used_prefs = set()
        used_owners = set()
        for coord in vertex.coords:
            pref = bidders[owner_labels[coord]].pref(coord)
            used_owners.add(owner_labels[coord])
            used_prefs.add(pref)
        assert(len(used_owners) == n)
        return len(used_prefs) == n

    disparates = set()
    full_disparates = set()

    if n == 2:
        for vertices in graph.values():
            for vertex in vertices:
                if check_if_disparate(vertex):
                    disparates.add(vertex)
    else:
        if brute_force:
            for vertices in graph.values():
                for vertex in vertices:
                    if check_if_disparate(vertex):
                        full_disparates.add(vertex)
            return full_disparates
        else:
            starting_edges = get_trap_doors()
            traversed = set()
            to_traverse = []
            to_traverse += starting_edges
            while len(to_traverse) != 0:
                edge = to_traverse.pop()
                vertices = graph[edge]
                for vertex in vertices:
                    if vertex in traversed:
```

```
                        continue
                    if check_if_disparate(vertex):
                        disparates.add(vertex)
                    for e in vertex.edges:
                        if check_if_trap_door(e):
                            to_traverse.append(e.data)
                    traversed.add(vertex)
    return disparates

def compute_average_solution(v):
    n = len(v.coords)
    average_sol = [0.] * n
    for coord in v.coords:
        average_sol = [average_sol[i] + coord[i] for i in range(n)]
    average_sol = [x / n for x in average_sol]

    return average_sol

def check_solution_quality(v, bidders = []):
    n = len(v.coords)
    #Compute average solution from approximate
    average_sol = compute_average_solution(v)

    #Make sure everyone prefers a different room
    used_prefs = set()
    social_welfare = 0.
    total_utility = 0.
    for bidder in bidders:
        pref = bidder.pref(average_sol)
        social_welfare += bidder.valuation[pref]
        total_utility += bidder.valuation[pref] - average_sol[pref]
        used_prefs.add(pref)

    if len(used_prefs) != n:
        return -1e9, -1e9 #This solution is not envy free
    return social_welfare, total_utility

def create_n_simplex(n):
    simplex = []
    for i in range(n):
        x = [0.] * n
        x[i] = 1
        simplex.append(np.array(x))
    return simplex


bidders = {
    3 : [Bidder(0,[0.34, 0.35, 0.33]), Bidder(1,[0.5, 0.20, 0.35]), Bidder(2,[0.75, 0.2, 0.05])]
    4 : [Bidder(0,[0.25, 0.25, 0.25, 0.25]),
        Bidder(1,[0.5, 0.5/3, 0.5/3, 0.5/3]),
        Bidder(2,[.2, 0.3, 0.1, 0.4]),
        Bidder(3,[0.8, .1, .07, .03])]
}


n = 3
```

```
number_of_subdivisions = 5
simplex = create_n_simplex(n)

print("Creating Graph via Sperner Triangle...")
graph = subdivide_wrapper(simplex, number_of_subdivisions)
print("Done creating graph!")

print("Labelling the Sperner Triangle with owners...")
owner_labels = assign_owners(next(iter(graph.values()))[0], graph)
print("Done Labelling!")

print("Solving for approximate solution...")
solutions = traverse_trap_doors(graph, owner_labels,bidders[n])
print("Solved! Computed", len(solutions), "solutions.")
best_solution = max(solutions, key=partial(check_solution_quality, bidders=bidders[n]))
print("Best Solution:", compute_average_solution(best_solution),
        "with (social welfare, total_utility)", check_solution_quality(best_solution, bidders=b
```

Appendix

Python implementation for the market-based approach in the cardinal case

```python
from bidders import Bidder

def compute_maximum_matching(bidders, n):
    room_edges = {}
    bidder_edges = {}
    for i in range(n):
        room_edges[i] = set([])
    for i in range(len(bidders)):
        bidder_edges[i] = bidders[i].demand_set
        for room in bidders[i].demand_set:
            room_edges[room].add(i)

    flow = {(i,j) : 0 for i in range(len(bidders)) for j in range(n)}

    unsaturated_bidders = set(range(len(bidders)))
    unsaturated_rooms = set(range(n))

    while True:
        bidder_q = list(unsaturated_bidders)
        room_q = []
        room_paths = {}
        bidder_paths = {}
        while len(bidder_q) != 0 or len(room_q) != 0:
            if len(room_q) > 0:
                v = room_q.pop()
                if v in unsaturated_rooms:
                    room_paths[-1] = v
                    unsaturated_rooms.remove(v)
                    break
                for bidder in room_edges[v]:
                    if bidder not in bidder_paths and flow[(bidder, v)] > 0:
                        bidder_q.append(bidder)
                        bidder_paths[bidder] = v
            elif len(bidder_q) > 0:
                v = bidder_q.pop()
                if v not in bidder_paths:
                    bidder_paths[v] = -1
                for room in bidder_edges[v]:
                    if room not in room_paths and flow[(v,room)] <= 0:
                        room_q.append(room)
                        room_paths[room] = v
            else:
                raise BaseException
        if room_paths.get(-1) == None:
            break
        else:
            curr_room = True
            curr = room_paths[-1]
            while True:
                if curr_room:
                    new_curr = room_paths[curr]
```

```
                else:
                    new_curr = bidder_paths.get(curr)
                    if new_curr == -1:
                        unsaturated_bidders.remove(curr)
                        break

                if curr_room:
                    flow[(new_curr, curr)] += 1
                else:
                    flow[(curr, new_curr)] -= 1
                curr_room = not curr_room
                curr = new_curr
    matching = set([])
    for edge in flow:
        if flow[edge] == 1:
            matching.add(edge)
    return matching

def compute_mods(bidders):
    n = len(bidders)
    new_bidders = []
    for b in bidders:
        if len(b.demand_set) != 0:
            new_bidders.append(b)
    bidders = new_bidders
    if len(bidders) == 0:
        return set([])
    matching = compute_maximum_matching(bidders, n)
    N = set(range(len(bidders)))
    matched_bidders = set([m[0] for m in matching])
    unmatched_bidders = N - matched_bidders
    if len(unmatched_bidders) == 0:
        return set([])
    i = next(iter(unmatched_bidders))
    mod_set = bidders[i].demand_set
    od_set = set([])

    while len(mod_set) != 0:
        mod_set_matched_bidders = [m[0] for m in matching if m[1] in mod_set]
        od_set = od_set | mod_set
        mod_set = set([])
        for j in mod_set_matched_bidders:
            mod_set = mod_set | bidders[j].demand_set
        mod_set = mod_set - od_set

    min_set = set([])
    mod_set = od_set

    for a in od_set:
        mod_set = mod_set - set([a])
        mod_set_matched_bidders = [i for i in bidders if i.demand_set <= (min_set | mod_set)]
        trunc_bidders = [i for i in mod_set_matched_bidders]
        trunc_matching = compute_maximum_matching(trunc_bidders, n)
        k = len(trunc_matching)
        if k == len(mod_set_matched_bidders):
            min_set = min_set | set([a])
```

```
108
109     return min_set
110
111 def compute_ODS(bidders):
112     mod = compute_mods(bidders)
113     od = mod
114     while len(mod) > 0:
115         for b in bidders:
116             b.demand_set = b.demand_set - mod
117         mod = compute_mods(bidders)
118         od = od | mod
119     return od
120
121 def compute_price_differential(od, bidders, prices):
122     if len(od) == 0:
123         return 0
124     overly_demanding_bidders = [b for b in bidders if b.demand_set <= od]
125     potential_differentials = []
126     for bidder in overly_demanding_bidders:
127         indirect_utility = bidder.indirect_utility(prices)
128         under_demanded_utilities = []
129         for s in range(len(bidders)):
130             if s not in od:
131                 under_demanded_utilities.append(bidder.utility(s, prices))
132         potential_differentials.append(indirect_utility - max(under_demanded_utilities))
133
134     return min(potential_differentials)
135
136
137 def conduct_discrete_price_auction(bidders, C):
138     n = len(bidders)
139     prices = [C/n] * n
140     count = 0
141     while True:
142         count+=1
143         print("Price on iteration", str(count)+ ":")
144         print(prices)
145         for bidder in bidders:
146             bidder.recompute_demand_set(prices)
147         od = compute_ODS(bidders)
148         print("FODS on iteration", str(count)+ ":")
149         print(od)
150         for bidder in bidders:
151             bidder.recompute_demand_set(prices)
152         if len(od) == 0:
153             return compute_maximum_matching(bidders, n), prices
154         x = compute_price_differential(od, bidders, prices)
155         for r in range(n):
156             if r not in od:
157                 prices[r] -= len(od) * x / n
158             else:
159                 prices[r] += (n - len(od)) * x / n
160
161 bidders = [Bidder() for b in range(6)]
162 bidders[0].initialize_valuations([15, 18, 10, 15, 24, 28])
163 bidders[1].initialize_valuations([18, 25,  3, 18, 25, 15])
```

```
bidders[2].initialize_valuations([ 6, 25, 15, 18, 18, 25])
bidders[3].initialize_valuations([15,  5, 18, 12,  9, 25])
bidders[4].initialize_valuations([ 6, 22,  5,  5, 10, 12])
bidders[5].initialize_valuations([ 6,  9,  2, 21, 25,  9])
C = 60
res = conduct_discrete_price_auction(bidders, C)
print("The envy-free assignment is given by:", "\nmu = " + str(res[0]) + "\np = " + str(res[1]))

# bidders = []
# n = 100
# for i in range(n):
#   bidders.append(Bidder())
#   bidders[i].initialize_random_int_valuations(n)
# C = n * 50
# res = conduct_discrete_price_auction(bidders, C)
# print("The envy-free assignment is given by:", "\nmu = " + str(res[0]) + "\np = " + str(res[1
```
