Deploying Fair and Efficient Course Allocation Mechanisms

Paula Navarrete Díaz, Cyrus Cousins, George Bissias, Yair Zick

Abstract

In large universities, the task of assigning classes to thousands of students while considering their preferences, along with course schedules and capacities, presents a significant challenge. Ensuring the effectiveness and fairness of course allocation mechanisms is crucial to guaranteeing student satisfaction and optimizing resource utilization. We address this problem from an economic perspective, using formal justice criteria to evaluate different algorithmic frameworks. We develop software for generating synthetic students with binary preferences over courses represented by linear inequality constraints, and implement four allocation algorithms: the SPIRE algorithm used by UMass Amherst; Round Robin; an Integer Linear Program; and the Yankee Swap algorithm, a flexible approach that offers significant fairness guarantees. We propose improvements to the Yankee Swap framework to handle scenarios with item multiplicities. Through experimentation with the Fall 2024 Computer Science course schedule at UMass Amherst, we evaluate each algorithm's performance relative to standard justice criteria, providing insights into fair course allocation in large university settings.

1 Introduction

Consider the problem of assigning classes to students in a large, public university. Due to the sheer scale of the problem (thousands of students, thousands of courses), universities often use automated systems that (a) collect student preferences and (b) assign students to classes based on their eligibility/preferences/priority. The challenge universities face is thus to ensure that their course allocation systems are fast, effective and satisfy certain design criteria. For example, a course allocation system should ensure allocative efficiency: ensuring that classes are assigned to students who actually want them; such a system should also be fair to students, ensuring that individual students are offered equal access to classes they want. This task is crucial as it guarantees student satisfaction, facilitates timely graduation by enabling students to fulfill their requirements, and optimizes resource utilization by maximizing the occupancy of class seats.

We approach this problem from an economic perspective. We use formal *justice criteria* [13, 15] to evaluate our algorithmic frameworks. We utilize the *Yankee Swap framework* [17, 16]: a simple and flexible framework that achieves many of these concepts under certain assumptions, namely that student utilities are given by binary submodular functions. Indeed, most fair allocation algorithms assume a certain structure on agent preferences (with the notable exception of Lipton et al. [12], which produces an approximately envy-free allocation for general utilities). Thus, there remains a gap between theory and practice: how do modern fair allocation algorithms perform in realistic settings? In the present work, we attempt to answer this question in the context of *course allocation.* We perform numerous experiments on both standard and state of the art allocation algorithms to compare performance in terms of different efficiency and justice criteria. We also introduce an improvement to one allocation algorithm that results in a significant improvement in its runtime.

1.1 Our Contributions

Our main contribution is the development of software for generating synthetic *students* based on a generic course schedule and simple statistics, and the implementation of four allocation algorithms (Section 2.5): Round Robin (a classical scheduling algorithm), SPIRE (the algorithm used at UMass Amherst), an Integer Linear Program that finds the optimal solution maximizing utilitarian social welfare, and Yankee Swap [16] (a state of the art fair allocation algorithm), to solve the course allocation problem. We also discuss improvements to the Yankee Swap framework, naturally arising from its implementation in a setting where items have multiplicities and agents can only desire a limited number of items (Section 3). Finally, we test each algorithm against the Fall 2024 UMass Amherst Computer Science course schedule, and measure their performance relative to a number of standard *justice criteria* (Section 4).

2 Fair Course Allocation

2.1 Related Work

Several works study the course allocation problem. Budish et al. [6] introduced the Course Match algorithm, which uses a massive parallel heuristic search to approximate a competitive equilibrium in course allocations. While demonstrating effectiveness in fairness and efficiency, Course Match requires students to articulate highly complex preferences. Diebold et al. [10] explored course allocation through stable matching mechanisms, particularly focusing on the Gale-Shapley student optimal stable mechanism (SOSM) and the efficiency-adjusted deferred acceptance mechanism (EADAM). While SOSM prioritizes stability and strategy-proofness, it may fall short in efficiency. EADAM mitigates this drawback by recuperating efficiency losses from SOSM at the expense of strategy-proofness. Biswas et al. [2] proposed an algorithmic approach to tackle course enrollment challenges, especially in scenarios where resources conflict, represented through conflict graphs. Furthermore, Biswas et al. [3] investigated fair allocation of indivisible items amidst conflicting pairs, represented by interval graphs, and provided a course allocation algorithm under identical utilities, implying that, for any course, all students have the same utility.

2.2 Preliminaries

We let $\mathbb{N}_0$ be the set of non-negative integers. Let $N = \{1, 2, ..., n\}$ be a set of *agents* and $G = \{g_1, g_2, ..., g_m\}$ be a set of *item types*. Each item type $g \in G$ has a limited number of identical copies q_g; we write $\vec{q} = (q_{g_1}, ..., q_{g_m})$. An *allocation* A is a partition of items. More formally, $A = (A_0, A_1, \ldots, A_n)$ is a set of non-overlapping bundles, where each agent i is assigned a bundle A_i, and A_0 denotes the set of unassigned items. It is easier to think of A as a matrix. Since an agent may own multiple copies of a single item, A_i is a vector in $\mathbb{N}_0^m$ that represents agent i's bundle under allocation A, where $A_{i,g}$ is the number of copies of item g owned by agent i. Thus, an allocation is valid if for any item $g \in G$, $A_{0,g} + \sum_{i=1}^{n} A_{i,g} = q_g$: the number of unassigned copies of g – $A_{0,g}$ – plus the number of copies assigned to all agents equals exactly q_g. For ease of readability, for an allocation A and an item type g, we say that $g \in A_i$ if $A_{i,g} > 0$.

Each agent has a *valuation function* $v_i : \mathbb{N}_0^m \to \mathbb{N}_0$ which depends only on the bundle A_i allocated to them. We define the *marginal utility* of agent i from receiving an additional copy of the item type g as

$$\Delta_i(A_i, g) \triangleq v_i(A_i + \mathbb{1}_g) - v_i(A_i).$$

Here, $\mathbb{1}_g \in \{0,1\}^m$ is the vector with 0s everywhere except for a 1 in the g-th coordinate.

We say that v_i is a **binary** function if for any allocation A_i and any item type g,

$$\Delta_i(A_i, g) \in \{0, 1\}.$$

Given two bundles $A_i, B_i \in \mathbb{N}_0^m$, we say $A_i \preceq B_i$ if for all $g \in G$, $A_{i,g} \leq B_{i,g}$. We say that v_i is a *submodular* function if for any two bundles $A_i, B_i \in \mathbb{N}_0^m$ such that $A_i \preceq B_i$, and for any item type g,

$$\Delta_i(A_i, g) \geq \Delta_i(B_i, g).$$

Intuitively, the more items an agent i owns, the less marginal benefit they receive from additional items.

2.3 Justice Criteria

In order to compare two different allocations, we define the following metrics. The *Utilitarian Social Welfare* (USW) sums the total welfare of agents:

$$\texttt{USW}(A) = \frac{1}{n} \sum_{i \in N} v_i(A_i). \tag{1}$$

Let $N_{>0}(A)$ be the set of students that have a positive utility under allocation A, defined as $N_{>0}(A) = \{i \in N | v_i(A_i) > 0\}$. The Nash Social Welfare (NSW) [7] first minimizes the number of agents with zero utility (i.e., maximizes $|N_{>0}(A)|$), and then takes the product of utilities for all agents with positive utility under A:

$$\texttt{NSW}(A) = \frac{1}{n} \prod_{i \in N_{>0}(A)} v_i(A_i). \tag{2}$$

Given an allocation A, we say that agent i *envies* agent j [11] if $v_i(A_i) < v_i(A_j)$. An allocation is *envy-free* (EF) if no agent envies another. Envy-free allocations are not guaranteed to exist (consider a setting with two agents and one single item); this gave rise to a the canonical notion of *envy-freeness up to one item* (EF-1) [5, 12]. An allocation A is said to be EF-1 if for any two agents i and j, if i envies j, then there exists some item $g \in A_j$ such that $v_i(A_i) \geq v_i(A_j - \mathbb{1}_g)$.

2.4 Valuation Function

Thus far we have remained agnostic to the valuation function v_i that defines the preferences of agent $i \in N$. We introduce a valuation function defined by *linear inequality constraints*, which we employ in our implementation.

2.4.1 Linear inequality constraints

All the allocation algorithms implemented (later described in Section 2.5) rely on having access to the valuation function. Having a pre-computed oracle access to this valuation function means that we would need to store the utility value for each student $i \in N$ from any potential bundle A_i. However, keeping track of the utility each student derives from every possible combination of items becomes intractable when dealing with thousands of students and hundreds of classes. To address this challenge, we utilize linear inequality constraints to represent agent preferences. These constraints can be computed as needed at a relatively low cost, offering a more manageable approach to modeling student preferences. Such a constraint has the form

$$Z^i A_i \leq \vec{b^i}, \tag{3}$$

where Z^i is an $n \times m$ constraint matrix, A_i is an $m \times 1$ vector of item multiplicities, and $\vec{b^i}$ is an $n \times 1$ limit vector. For $\vec{x} \in \mathbb{N}_0^m$, we write $\vec{x} \preceq A_i$ whenever $x_g \leq A_{i,g}$ for all $g \in G$. We have

$$v_i(A_i) = \max\{|\vec{x}| : Z^i\vec{x} \leq \vec{b^i}, \vec{x} \preceq A_i\}, \tag{4}$$

where $|\vec{x}|$ denotes the 1-norm of $\vec{x}$.

2.4.2 Expressiveness of linear inequality constraints

Although linear inequality constraints do not always describe binary submodular valuation functions, every binary submodular value function can be represented by a set of linear inequality constraints. The following argument leverages the fact that every binary submodular valuation function is the rank function of some finite matroid [14]. We make the argument in the context of goods having multiplicity one, but the argument naturally extends to goods with multiplicities.

Let $M = (E, \mathcal{I})$ be any finite matroid with $|E| = m$. Let $\mathcal{C}$ be the set of *circuits* of M. A circuit is the smallest dependent set containing a set independent in M, i.e.,, for all $C \in \mathcal{C}$, there is some $c \in C$ such that $C \setminus \{c\}$ is an independent set. Being a finite matroid, we assume that it is possible to order both the elements $e_1, e_2, \ldots, e_m$ of E as well as the circuits $C_1, C_2, \ldots, C_n$ of $\mathcal{C}$. The *rank* of the circuit C_i is the size of the largest independent set $I \subseteq C_i$, i.e., $r_i = |C_i| - 1$.

We define an $n \times m$ constraint matrix Z as follows. The i-th row of Z, $\vec{z_i}$, is an indicator vector for circuit C_i: $z_{ij} = 0$ unless $e_j \in C_i$, in which case $z_{ij} = 1$. Let $X \subseteq E$ be an arbitrary subset of elements. We similarly define the length m indicator column vector for X by $\vec{x}$ such that $x_j = 0$ if $e_j \notin X$ and $x_j = 1$ otherwise. By construction, for every $i \in N$ and $X \subseteq E$, we have that $\vec{z_i}\vec{x} = |C_i \cap X|$. Finally, we define an n dimensional *constraint vector* $\vec{b}$ such that $b_i = r_i$.

Theorem 1. *For arbitrary $X \subseteq E$ and corresponding indicator vector $\vec{x}$, $Z\vec{x} \leq \vec{b}$ iff $X \in \mathcal{I}$.*

Proof. Suppose that $X \in \mathcal{I}$ and consider any row $\vec{z_i}$. By construction we have $\vec{z_i}\vec{x} = |C_i \cap X|$. On the other hand, since X is independent by assumption, and every subset of an independent set is itself independent, it must be the case that $C_i \cap X \in \mathcal{I}$. It follows then that $|C_i \cap X| \leq \max\{|I| : I \in \mathcal{I}, I \subseteq C_i\} = r_i$. Thus, $\vec{z_i}\vec{x} \leq b_i$.

Next, suppose that $X \notin \mathcal{I}$. Since Z possesses a row for every circuit in $\mathcal{C}$, and every dependent set contains a circuit, there must exist some row $\vec{z_i}$ such that $C_i \subseteq X$. Thus, $\vec{z_i}\vec{x} = |C_i \cap X| = |C_i| = r_i + 1$ and so $\vec{z_i}\vec{x} > b_i$. □

2.5 Allocation Algorithms

In the course allocation context, an allocation algorithm is an algorithm that takes as input a set or courses G and a set of students N with certain preferences over these courses. The preferences of student i are encoded by a valuation function $v_i : 2^G \to \mathbb{R}$, where for every bundle of classes $S \subseteq G$, $v_i(S)$ is the utility that student i derives from receiving the set of classes S. An allocation algorithm receives as input the students' preferences, and outputs a valid allocation A of the courses to the students. We consider four different allocation algorithms, SPIRE, Round Robin, an integer linear program, and Yankee Swap.

SPIRE: The SPIRE algorithm is the course allocation algorithm currently employed by UMass Amherst to enroll students in courses. In this system, student enrollment operates on a first-come, first-served basis, following a sequential pattern: the first student to access the platform enrolls in all their desired courses. Subsequently, the following students enroll in all desired courses that have available seats left. More formally, the algorithm works as follows. Initially, students start with an empty bundle and all course seats are available. The first student picks a bundle of classes that maximizes their utility and course capacities are updated accordingly. The second student does the same, considering these updated capacities. The algorithm terminates once all students picked classes. We assume that students pick *clean bundles*: a bundle $A_i \subseteq G$ of classes is clean [1] with respect to agent $i \in N$ if for any item $g \in A_i$, $v_i(A_i \setminus g) < v_i(A_i)$, i.e., removing any item from i's bundle will strictly reduce their utility. In other words, we assume that on their turn, students do not enroll in classes that they do not want to enroll to, or that conflict with one another. The SPIRE algorithm significantly favors students who access the platform early, while those who log in later, face a considerable disadvantage in securing desired classes. In reality, the algorithm allows students to "hoard" courses, i.e., exacerbating the scarcity of available spots for subsequent students.

Round Robin: The well known *Round Robin algorithm* [4] bears resemblance to the SPIRE algorithm in its sequential nature. However, unlike SPIRE, Round Robin allows students to select only one course at a time, employing multiple rounds instead of a single one. In this algorithm, agents begin with an empty bundle and proceed through rounds, selecting one item from the pool of unallocated goods in each round. The first student enrolls in one of their desired courses; course capacities are updated accordingly. The second student does the same, considering these updated capacities. Students keep enrolling in one course at the time in rounds, until no student can strictly increase their utility by enrolling in any available class.

ILP: we encode agent preferences using linear inequality constraints given by Equation 4. Allocation by integer linear program (ILP) finds an allocation A that maximizes the utilitarian social welfare, subject to the constraints. Since both the objective and constraints are linear, the ILP is guaranteed to return the optimal integer allocation. In particular we solve the following ILP.

Maximize:

$$\texttt{USW}(A) = \frac{1}{n} \sum_{i \in N} v_i(A_i). \tag{5}$$

Subject to:

$$\begin{bmatrix} Z^1 & & & \\ & Z^2 & & \\ & & \ddots & \\ & & & Z^n \end{bmatrix} \begin{bmatrix} A_1 \\ \vdots \\ A_n \end{bmatrix} \leq \begin{bmatrix} \vec{b}^1 \\ \vdots \\ \vec{b}^n \end{bmatrix}. \tag{6}$$

Yankee Swap: the *Yankee Swap algorithm* [17] is similar to the well known Round Robin algorithm: agents sequentially pick items in rounds. However, unlike the Round Robin algorithm, agents are allowed to steal items from other agents if they do not like any of the available unassigned items. In each round, a student can either enroll in a course with available seats or steal a seat from another student; however, they can only do this if the student they are stealing from can recover their utility by either enrolling in another desirable course with available spots, or stealing a seat from a third agent, and so forth. The algorithm terminates when no student can further benefit from enrolling in courses with available seats, or when students are solely interested in stealing seats from students who cannot recover their utility. When agents have binary submodular valuations, Yankee Swap offers several theoretical guarantees: it outputs a leximin allocation, which also maximizes `USW` and returns an EF-1 allocation (see Section 2.3). In addition, Yankee Swap is *truthful*: no agent can increase their utility by misreporting their preferences, e.g. say that they want to enroll in an undesirable class, or say that they do not want to enroll in a desirable class.

The main computational hurdle in implementing the Yankee Swap algorithm is computing the *item exchange graph* (see Section 3.1 for more details): this is a graph over item types, where there is a directed edge from item g to item g' if the agent who owns item g can retain their current utility by exchanging g for g'. This graph is used to compute the transfer paths utilized in the Yankee Swap algorithm, and is the most computationally intensive aspect of its runtime.

We next describe how we exploit the underlying problem structure in order to derive additional improvements to the Yankee Swap algorithm.

3 Yankee Swap with Multiplicity of Items

The original Yankee Swap algorithm [17] assumes that there is no multiplicity of items: each item type g has only one copy, i.e., $q_g = 1$ for all $g \in G$. When items have multiple identical copies, such as seats in a class in the course allocation problem, considering each copy of an item as an individual good becomes highly inefficient: the item exchange graph becomes a data structure whose size scales by the number of course *seats* rather than the number of courses (which is typically significantly smaller). We propose a modified version of the Yankee Swap algorithm that allows multiplicity of items.

3.1 Key Concepts

Given an allocation A, the *exchange graph* $\mathcal{G}(A)$ is a directed graph over the set of items G. There is an edge from an item $g \in A_i$ to another item $g' \in G$ if $v_i(A_i) = v_i(A_i - \mathbb{1}_g + \mathbb{1}_{g'})$. In other words, there is an edge from g to g' if the agent who owns g under A can replace it with g' without reducing their utility. Note that if no agent owns item $g \in G$ (i.e., $g \in A_0$), then there are no outgoing edges from g.

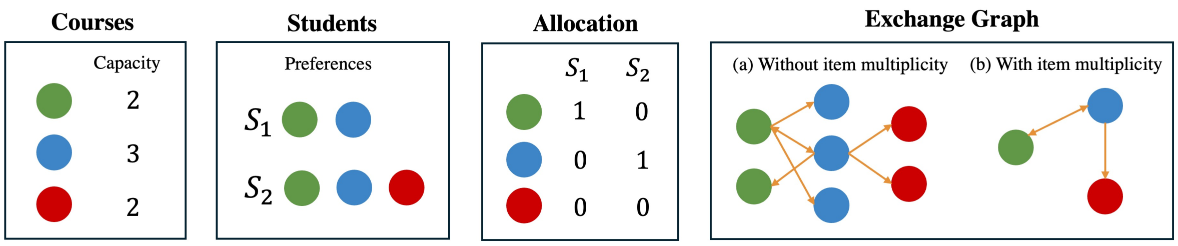

Figure 1: Consider a small instance with 3 courses — *green*, *blue* and *red* — and two students — S_1 and S_2 — who have preferences over the courses. Consider the allocation in which S_1 is enrolled in *green* and S_2 is enrolled in *blue*. The exchange graph (a) shows the exchange graph representation of this allocation without multiplicity of items, while exchange graph (b) shows the same representation considering item multiplicity.

The Yankee Swap algorithm works as follows. Initially, all agents are assigned empty bundles and all items are unassigned, i.e., the initial allocation A is such that $A_{0,g} = 1$ and $A_{i,g} = 0$ for all item types $g \in G$ and agents $i \in N$. At each round, we pick an agent i according to some criterion (say, the agent with the lowest utility). Let $F_i(A) = \{g \in G : \Delta_i(A_i, g) = 1\}$ be the set of items that offer agent i a marginal benefit of 1 given their current bundle. We introduce a dummy node i and a dummy node t to the exchange graph. We add a directed edge from i to every item in $F_i(A)$, and a directed edge from each item $g \in A_0$ to t. Then, we search for a shortest path from i to t in the resulting graph. Note that such a path $(i, g_0, g_1, \ldots, g_k, t)$ corresponds to agent i taking item g_0, and the agent who owns item g_h replaces it with item g_{h+1}, until we reach the unassigned item g_k, which is now assigned to the last agent in the path. If there is no such path, that means that we cannot increase agent i's utility without reducing another agent's utility. At this stage, agent i is kicked out of the game. We repeat this process until there are no agents left playing.

3.2 The Algorithm

Consider the example shown in Figure 1. We have three courses with limited seats (Capacity) and two students with preferences over these courses. We assume that there are no course conflicts. Consider the allocation in which student S_1 is enrolled in the first course, and student S_2 is enrolled in the second one. Since Yankee Swap considers a unique copy per item, to represent the corresponding exchange graph we need to create one node per seat for each course. Each student will own a single seat in a course — a single node in the graph; however, since they are indifferent to which seat they get in a course they would like to add to their bundle, these nodes will have outgoing edges to all the copies of those courses, as shown in version (a) of the exchange graph in Figure 1. As the size of the instance grows, considering thousands of students with complex preferences and courses hundreds of available seats, this representation becomes intractable. We simplify this representation by considering a single node per course (item type), as shown in version (b) of the exchange graph in Figure 1.

We propose a modified version of the Yankee Swap algorithm that allows items to have multiple identical copies, i.e., $q_g \geq 1$ for all $g \in G$. The algorithm is largely the same, but some key aspects are redefined. Since items have multiple copies, they have multiple owners, which requires a redefinition of the exchange graph. The exchange graph $\mathcal{G}$ is now a directed graph over the set of items G, in which there is an edge of the form (g, g') if there *exists* an agent i

such that $g \in A_i$ and $v_i(A_i) = v_i(A_i - \mathbb{1}_g + \mathbb{1}_{g'})$, i.e., if there exists any agent in possession of g willing to exchange it for g'.

The definition of the transfer path remains the same, however its execution is trickier. Since item type g might have multiple owners, the existence of an edge (g, g') might be because of a single or possibly multiple owners willing to exchange a copy of g for a copy of g'. Thus, once we find a transfer path, it is not trivial to identify the agents involved in the transfer path, which is crucial to execute the path and update the allocation. To tackle this, we define a *responsible agents tuple* R of dimension $m \times m$ to keep track of the set of agents responsible for each of the edges. Here, $R_{g,g'}$ is the set of agents that are in possession of item of type $g \in G$ and are willing to exchange it for an item of type $g' \in G$:

$$R_{g,g'} = \{i \in N : A_{i,g} > 0 \wedge (v_i(A_i) = v_i(A_i - \mathbb{1}_g + \mathbb{1}_{g'}))\}.$$

In other words, $R_{g,g'}$ is the set of agents responsible for an existing edge between nodes g and g' in the exchange graph. Note that $R_{g,g'} = \emptyset$ if and only if there is no edge from g to g' in $\mathcal{G}$: there is no agent in possession of a g type item willing to exchange it for a g' type item. Initially, $R_{g,g'} = \emptyset$ for every $g, g' \in G$ since no items have been allocated.

We define $D_i = \{g \in G | v_i(\mathbb{1}_g) = 1\}$ as the set of agent i's desired items. Since agents have binary submodular valuations over the items, this set includes all items that could possibly contribute a positive marginal contribution to agent i under any allocation A. This observation implies that $F_i(A) \subseteq D_i$ under any allocation A. We use the set D_i instead of the originally defined set $F_i(A)$ (see 3) despite the possibility of $F_i(A)$ being smaller. This preference arises because maintaining D_i is more cost-effective. Given that an agent's desired set of items is typically small and can be pre-computed, D_i offers efficiency compared to $F_i(A)$, which requires calls to the valuation function in each iteration.

Algorithm 1 Yankee Swap with multiplicity of items

```
Require: A set of agents $N$, a set of item types $G$ with $\vec{q}$ copies of each type, and access to
         valuation functions $\{v_i\}_{i \in N}$
Ensure: A valid allocation $A$
 1: $A = (A_0, A_1, ..., A_n) \leftarrow (\vec{q}, \emptyset, ...\emptyset)$
 2: $R = \left(\left(R_{g,g'}\right)_{g \in G}\right)_{g' \in G} \leftarrow (\emptyset, ..., \emptyset)$
 3: Construct graph $\mathcal{G}$ with one node for each item type $g \in G$
 4: $U \leftarrow N$
 5: while $U \neq \emptyset$ do
 6:     Let $i \in \arg\min_{j \in N} v_j(A_j)$
 7:     Add source node $s$ to $\mathcal{G}$ with an edge to all nodes $g \in G$ such that $\Delta_i(A_i, g) > 0$
 8:     Check if there is a path $P = (s, g_0, ..., g_k)$ where $g_k \in A_0$
 9:     if a path exists then
10:         $A, I \leftarrow$ update_allocation$(A, P, R)$
11:         $\mathcal{G}, R \leftarrow$ update_exchange_graph$(A, \mathcal{G}, P, I, R)$
12:     else
13:         $U \leftarrow U \setminus \{i\}$
14: return $A$
```

Algorithm 1 is a modified version of the Yankee Swap algorithm [17] that allows item multiplicity. Note that the algorithm keeps the same structure as the original algorithm.

However, two subroutines, *update allocation* and *update exchange graph* are modified (see Algorithms 2 and 3 respectively).

Algorithm 2 `update_allocation`(A, P, R)

```
Require: A feasible allocation $A$, a transfer path $P$, an edge tuple $R$
Ensure: An updated allocation $A$, a list $I$ with involved agents in the path
1: $I_1 \leftarrow i$
2: $A_i \leftarrow A_i + \mathbb{1}_{g_0}$
3: for each pair $(g, g')$ in $P$ do
4:     Let $j \in R_{g,g'}$
5:     $I_{l+1} \leftarrow j$
6:     $A_j \leftarrow A_j - \mathbb{1}_g + \mathbb{1}_{g'}$
7: $A_0 \leftarrow A_0 - \mathbb{1}_{g_k}$
8: $I \leftarrow \left(I_j\right)_{j=1}^{n}$
9: return $A, I$
```

3.3 Analysis

Let $q = \sum_{g \in G} q_g$ be the total number of copies of all item types. A naive implementation of Yankee Swap, considering every copy of an item type to be a different item runs in $O((q+n)q^2(n+\tau))$ time [17] where τ the maximum time to compute any agent's valuation function. This implementation considers an oracle access to the valuation function, which is not necessarily realistic. Our implementation addresses this issue by separating the exchange graph representation from the agents through the responsible agents tuple, avoiding unnecessary calls to the valuation function.

Before going over the time complexity analysis of our implementation, we define the following parameters. First, each agent is limited to a maximum bundle size of $c_{\max}$, and can get a positive marginal contribution from at most γ different item types: $|D_i| \leq \gamma$ for all $i \in N$. Second, we define $q_{\max} = \max_{g \in G} q_g$ as the maximum number of copies of any item type. Finally, we define p as the maximum length of a transfer path. While transfer paths could potentially be of length m, they are far shorter in practice, often consisting of no more than two or three items swaps. Intuitively, this is because transfer paths can only be long when there are several courses that are booked to capacity, which did not occur naturally in our empirical evaluation.

The `update allocation` (Algorithm 2) and `update exchange graph` (Algorithm 3) subroutines run in $O(m)$ and $O(p\gamma c_{\max}(q_{\max}+\tau))$. Thus, our Yankee Swap implementation runs in $O((n+q)(\ln n + m^2 + p\gamma c_{\max}(q_{\max}+\tau))$ time. Assuming that the number of item copies is large, then $p \leq m \ll q$, resulting in a considerable reduction in the number of calls to the valuation function.

4 Experiments

We implement a general suite of tools for creating fair allocations among an arbitrary set of goods for a set of agents having arbitrary utility functions (full access to the code base will be available upon publication).

Algorithm 3 `update_exchange_graph`$(A, \mathcal{G}, P, I, R)$

```
Require: An updated allocation $A$, its exchange graph representation $\mathcal{G}$, a transfer path $P$,
         a sorted tuple of involved agents $I$, an edge tuple $R$
Ensure: An updated exchange graph $\mathcal{G}$ and edge tuple $R$
 1: Remove node $s$ from $\mathcal{G}$ and from path $P$
 2: for each agent $j \in I$ do
 3:     if $j$ exchanged $g$ for $g'$ in $P$, then
 4:         for each item type $h \in D_j$ do
 5:             if $j \in R_{g,h}$ and $A_{j,g} = 0$ then
 6:                 $R_{g,h} \leftarrow R_{g,h} \setminus \{j\}$
 7:                 if $R_{g,h} = \emptyset$ then
 8:                     Remove edge $g \to h$ from $\mathcal{G}$
 9:     for each item type $h \in A_j$ do
10:         for each item type $h' \in D_j$ do
11:             if $j \in R_{h,h'}$ then
12:                 if $v_j(A_j) > v_j(A_j - \mathbb{1}_h + \mathbb{1}_{h'})$ then
13:                     $R_{h,h'} \leftarrow R_{h,h} \setminus \{j\}$
14:                     if $R_{h,h'} = \emptyset$ then
15:                         Remove edge $h \to h'$ from $\mathcal{G}$
16:             else
17:                 if $v_i(A_i) \leq v_i(A_i - \mathbb{1}_h + \mathbb{1}_{h'})$ then
18:                     $R_{h,h'} \leftarrow R_{h,h'} \cup \{j\}$
19:                     if $|R_{h,h'}| = 1$ then
20:                         Add edge $h \to h'$ to $\mathcal{G}$
21: return $\mathcal{G}, R$
```

BA and BS CompSci	BS Informatics	MS	PhD	Total
1,539	154	613	297	2,603

Table 1: Distribution of students enrolled in the UMass Amherst Computer Science department.

	Undergraduate	MS	PhD
Number of students	1,693	613	148
Enrollment capacity	6	4	4
Preferred categories	{UGRAD, 500L}	{500L, 600L }	{500L, 600L}
Max number of liked classes	20	15	10
Min number of liked classes	1	1	1

Table 2: Parameters used to build the students' preferences and valuation functions.

For items, we encode their multiplicity and arbitrary *features*, the latter of which hold a particular value to agents. Agents are modeled as objects that implement a valuation function, which assigns a scalar value to every set of items. We evaluate allocations according to the justice criteria described in Section 2.3. We implement the allocation algorithms discussed in Section 2.5: Round Robin, SPIRE, Yankee Swap, and the ILP.

In our implementation items are courses whose features include course number, section, and meeting time. The set of all such courses, which we called the *schedule*, is populated from real course schedule data. Agents' (students) valuation functions are defined by linear constraint equations (see Section 2.4). Valuations are randomly generated as described in Section 4. The allocation algorithms were then run on the schedule and student list.

We conduct our experiments using the Fall 2024 UMass Amherst Computer Science course schedule, which comprises of 98 courses. Each course is specified by its time slot, day(s) of the week, and section number. These details are necessary for constructing the linear constraints in our model. Each course belongs to one of three categories: `UGRAD` for courses intended for undergraduate students, `500L` for advanced undergraduate and beginning graduate courses, and `600L` for advanced graduate courses.

As of 2023, the UMass Amherst Computer Science department had a total of 2,603 enrolled students, with the distribution shown in Table 1.

Consequently, our student population for the experiments is divided into three statuses: undergraduates, master's students (MS), and PhD students. We assume that only half of the PhD students enroll in classes each semester. The UMass Amherst enrollment system imposes constraints on the number of courses students can register for, which differ for undergraduate and graduate students. We also assume that students from different statuses have varying preferences and constraints regarding the courses they wish to enroll in: they like courses only from certain preferred categories, they like at least one class and have a maximum number of courses they might be interested in. These parameters are shown in Table 2.

For example, Undergraduate students prefer courses from the `UGRAD` and `500L` categories and are interested in a maximum of 20 courses, i.e.,, they obtain a utility of 1 from at most 20 courses.

To model student preferences, we employ a random sampling approach. For undergrad-

	USW(A)	NSW(A)	Zeroes(A)	Envy(A)	Envy1(A)
YS	3.01	3.00	0	488.5	0
RR	3.01	2.96	0	1202.57	69.76
ILP	3.01	2.79	121.01	1879.78	1306.26
SPIRE	2.82	4.06	861.16	1135.01	1057.42

Table 3: Average values for USW, NSW, number of zeros and envy metrics of the output allocations by the four allocation algorithms, obtained from 100 instances of randomly sampled students.

uates, we randomly determine the number of courses they are interested in (up to 20) and then randomly select this number of courses from the UGRAD and 500L categories. For MSc students, we randomly determine the number of courses they are interested in (up to 15) and then randomly select this number of courses from the 500L and 600L categories. Similarly, for PhD students, we randomly determine the number of courses they are interested in (up to 10) and then randomly select this number of courses from the 500L and 600L categories.

Based on data provided by UMass Amherst Computer Science department administrators, our preference model is plausible and serves primarily to illustrate the algorithm and its outcomes. We are currently in the process of a large-scale data collection exercise among UMass Amherst CS majors of all levels. Future experiments will be run on those real-world student preferences.

4.1 Results

We generate 100 instances of randomly sampled students, adhering to the characteristics outlined in Table 2. All four allocation algorithms — Yankee Swap, Round Robin, ILP, and SPIRE— were executed on these instances. In all three sequential algorithms — SPIRE, Round Robin, and Yankee Swap — the course selection order remains consistent, prioritizing PhD students for enrollment first, followed by MS students, and finally undergraduate students; indeed, we actually implemented a more advanced version of the Yankee Swap algorithm — General Yankee Swap [16] — that accommodates student priorities, as well as many other adaptations.

We assess and compare the performance of these algorithms based on metrics defined in Section 2.3. Given an allocation A, we evaluate Zeroes(A): the number of students that receive an empty bundle under A; Envy(A) is the number of students who are envious of another student under allocation A; and Envy1(A) is the number of students who are envious of another student under the EF-1 definition described in section 2.3.

Table 3 presents the average values for all five metrics across allocations obtained through the 100 simulations for each algorithm. These values, along with error bars, are depicted in Figure 2.

As anticipated, both ILP and Yankee Swap maximize USW, together with Round Robin. SPIRE lags slightly behind. Even though the difference in USW seems small, given the total of 2454 students, this disparity implies approximately 466 unallocated seats under the SPIRE algorithm. Yankee Swap and Round Robin ensure that no students receive an empty bundle in all 100 simulations, while ILP and particularly SPIRE leave numerous students without any

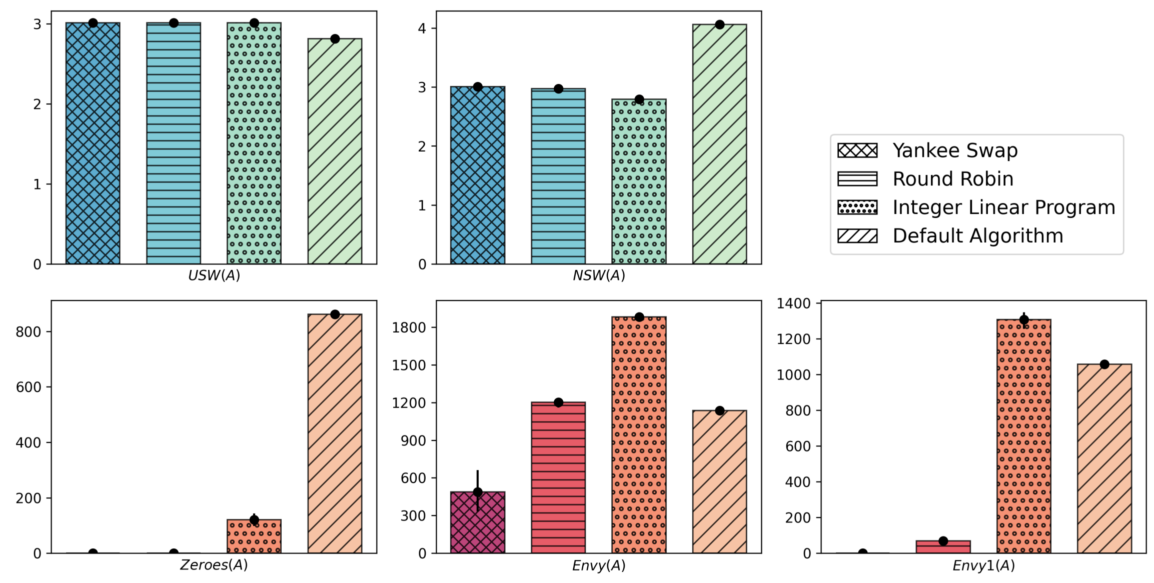

Figure 2: Average values and error bars for `USW`, `NSW`, number of zeros and envy metrics of the output allocations by the four allocation algorithms, obtained from 100 instances of randomly sampled students. The top row shows in cold colors the metrics we want to maximize, while the bottom row shows in warm colors metrics we want to minimize.

items[1]. SPIRE ranks high on the Nash welfare (`NSW`) metric. This is, however, misleading: `NSW` only considers students with positive utility, thus resulting in a biased assessment as there are several students with zero utility under SPIRE. Yankee Swap outperforms the other algorithms on the `NSW` metric, which indicates that its allocation balances well between social welfare and ensuring that all agents have as high a utility as possible; this is unsurprising as Yankee Swap outputs a leximin allocation. The ILP solution, followed by Round Robin, exhibits the most envy. It is worth noting that regardless of Round Robin outperforming SPIRE in other metrics, it exhibits more envy. While Yankee Swap only offers EF-1 guarantees, it yields allocations with the fewest envious agents.

We analyze the student bundle size frequencies averaged across 100 simulations for all four algorithms. Since bundles are clean in all implementations, a student's bundle size is simply their utility. The ILP algorithm prioritizes finding a feasible point that maximizes `USW` with no consideration for welfare distribution. SPIRE exhibits a trend where a significant number of students enroll in six courses, likely due to the advantageous ordering of class selection, while also resulting in a notable proportion of students enrolling in zero courses, as indicated by the `Zeroes` metric in Figure 2. In contrast, Round Robin tends to cluster students in the middle range, enrolling them in bundles of two, three, or four courses, while Yankee Swap allocates three courses to most students.

It is reasonable to argue that the effectiveness of the algorithms is closely tied to the sequence in which students choose their courses, particularly for sequential algorithms, or to the order in which they are defined, in the case of the ILP. To investigate this further, we rerun

[1]Through discussions with university administrators, what happens in reality is that students who sign up late simply sign up for classes that they do not want to take.

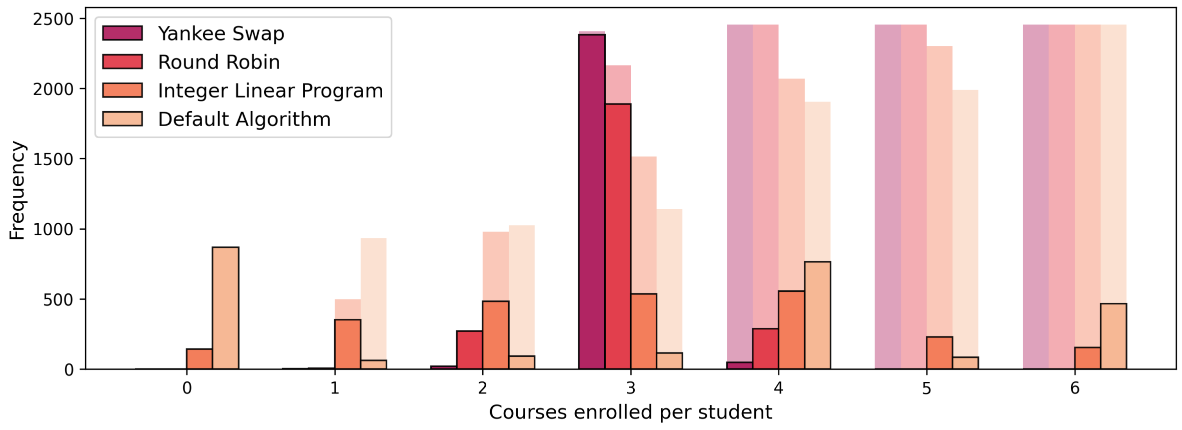

Figure 3: Histogram shows student bundle size frequencies (solid) and cumulative distribution (faint) averaged across 100 simulations for all four allocation algorithms.

the allocation algorithms, varying the orders in which students select courses while maintaining the hierarchy of PhD, MS, and undergraduate students. This analysis is conducted on a single instance of randomly generated students, considering ten distinct orders of students.

Once again, we compute all metrics on the allocations obtained through all four algorithms for the ten sequences. These values are depicted in Figure 4.

The outcomes remain consistent regardless of the sequence in which students enroll in classes. SPIRE consistently under-utilizes resources by not maximizing `USW`, while both SPIRE and ILP consistently leave numerous students with empty bundles. Finally, Yankee Swap consistently produces allocations with lower envy compared to allocations obtained by other algorithms, not only on the same sequence but also across all other sequences.

5 Discussion

In this work, we present several important steps towards a practical implementation of a state-of-the-art allocation mechanism in a realistic domain. We discuss some theoretical improvements, as well as practical workarounds we had utilized in order to maximize the efficiency of our implementation. We compare our candidate mechanism to several natural course allocation mechanisms, as well as to the current mechanism (SPIRE) used by UMass Amherst. The Yankee Swap mechanism is scalable, and outperforms the benchmarks on several standard justice criteria. In addition, the Yankee Swap mechanism is truthful, ensuring that students do not try and game the mechanism, a valuable property in markets without money. Thus, we believe that it is a natural candidate to replace the current mechanism utilized by UMass Amherst, and indeed, other large academic institutions.

We are currently in the process of collecting data on student preferences among the CS majors in UMass Amherst. The survey is ongoing, and our methods will be reexamined on this student data. The data will also be made public for the research community.

Our approach suffers from several natural limitations. First of all, while the Yankee Swap algorithm can run when agents do not have binary submodular utilities, it does not offer the same strong guarantees. Analyzing the theoretical and practical guarantees offered by the algorithm for more general classes of valuations is an important next step (see analysis in [8, 9]

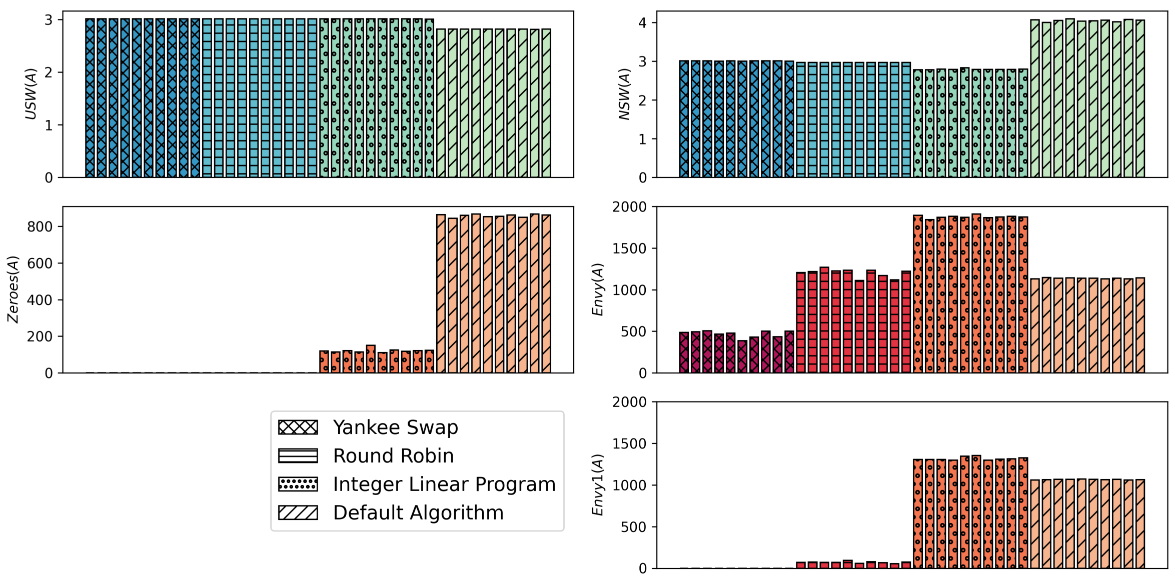

Figure 4: Values for `USW`, `NSW`, number of zeros and envy metrics of the output allocations by the four allocation algorithms, obtained from a single instance of randomly sampled students, considering ten distinct sequences of students. The top row shows in cold colors the metrics we want to maximize, while the bottom rows shows in warm colors metrics we want to minimize.

for some preliminary steps in this direction). More generally, we believe that having students express simpler preferences is the more practical approach. Methods that elicit complex student preferences (e.g. the CourseMatch [6] mechanism) place a significant cognitive burden on students, and do not scale to the magnitude of a large academic institution with tens of thousands of students. That being said, it would be valuable to analyze the *distortion* of simple allocation mechanisms: if we assume that each student has a true utility given by a valuation function v_i, but we approximate it with a simpler (say, binary submodular) function $\hat{v}_i$, what guarantees can we offer with respect to the true valuation v_i?.

The course allocation framework offers many other interesting challenges. First, students often have required classes in early semesters, which they must take to fulfill their degree requirements. How does one account for these classes when computing metrics such as envy and welfare? Secondly, the valuation functions induced by course conflict and capacity constraints may be binary, but are not guaranteed to be submodular. Indeed, as pointed out by Biswas et al. [3], the problem of computing fair allocations under general course conflict graphs is computationally intractable (even when the number of agents is constant). While we recognize that such challenges occur for arbitrary course conflict graphs, real-world course conflict graphs are far more structured (in UMass Amherst, they are by and large a union of disjoint cliques, which induces submodular preferences). It would be interesting to characterize course conflict graph classes which admit efficient algorithms for computing fair course allocations. Finally, discussions with UMass Amherst administrators indicate that students often over-enroll to classes. This is largely due to the fact that students are not certain whether they actually like the classes they enrolled in. Thus, they enroll to more classes than they will actually intend to take, and simply drop them within the Add/Drop period. Analyzing this phenomenon theoretically and empirically is an important step for future work.

References

[1] Nawal Benabbou, Mithun Chakraborty, Ayumi Igarashi, and Yair Zick. Finding fair and efficient allocations for matroid rank valuations. *ACM Transactions on Economics and Computation*, 9(4), oct 2021.

[2] Arpita Biswas, Yiduo Ke, Samir Khuller, and Quanquan C Liu. An algorithmic approach to address course enrollment challenges. *arXiv preprint arXiv:2304.07982*, 2023.

[3] Arpita Biswas, Yiduo Ke, Samir Khuller, and Quanquan C Liu. Fair allocation of conflicting courses under additive utilities. In *Proceedings of the 22nd International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 2162–2164, 2024.

[4] Sylvain Bouveret, Yann Chevaleyre, and Nicolas Maudet. Fair allocation of indivisible goods. In Felix Brandt, Vincent Conitzer, Ulle Endriss, Jérôme Lang, and Ariel D Procaccia, editors, *Handbook of computational social choice*, chapter 12, pages 284–309. Cambridge University Press, 2016.

[5] Eric Budish. The combinatorial assignment problem: Approximate competitive equilibrium from equal incomes. *Journal of Political Economy*, 119:1061–1061, 12 2011.

[6] Eric Budish, Gérard P. Cachon, Judd B. Kessler, and Abraham Othman. Course match: A large-scale implementation of approximate competitive equilibrium from equal incomes for combinatorial allocation. *Operations Research*, 65(2):314–336, 2017.

[7] Ioannis Caragiannis, David Kurokawa, Hervé Moulin, Ariel D Procaccia, Nisarg Shah, and Junxing Wang. The unreasonable fairness of maximum Nash welfare. *ACM Transactions on Economics and Computation (TEAC)*, 7(3):1–32, 2019.

[8] Cyrus Cousins, Vignesh Viswanathan, and Yair Zick. Dividing good and great items among agents with submodular valuations. In *Proceedings of the 19th International Conference on Web and Internet Economics (WINE)*, pages 225–241, 2023.

[9] Cyrus Cousins, Vignesh Viswanathan, and Yair Zick. The good, the bad and the submodular: Fairly allocating mixed manna under order-neutral submodular preferences. In *Proceedings of the 19th International Conference on Web and Internet Economics (WINE)*, pages 207–224, 2023.

[10] Franz Diebold, Haris Aziz, Martin Bichler, Florian Matthes, and Alexander Schneider. Course allocation via stable matching. *Business & Information Systems Engineering*, 6: 97–110, 2014.

[11] Duncan K. Foley. Resource allocation and the public sector. *Yale Economics Essays*, 7: 45–98, 1967.

[12] Richard J. Lipton, Evangelos Markakis, Elchanan Mossel, and Amin Saberi. On approximately fair allocations of indivisible goods. In *Proceedings of the 5th ACM Conference on Electronic Commerce (EC)*, pages 125–131, 2004.

[13] Hervé Moulin. *Fair Division and Collective Welfare.* MIT Press, 2003.

[14] James Oxley. *Matroid Theory.* Oxford Graduate Texts in Mathematics. Oxford Univerity Press, 2nd edition, 2011.

[15] Amartya Sen. *The Idea of Justice.* Harvard University Press, 2009.

[16] Vignesh Viswanathan and Yair Zick. A general framework for fair allocation with matroid rank valuations. In *Proceedings of the 24th ACM Conference on Economics and Computation (EC)*, pages 1129–1152, 2023.

[17] Vignesh Viswanathan and Yair Zick. Yankee swap: a fast and simple fair allocation mechanism for matroid rank valuations. In *Proceedings of the 22nd International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 179–187, 2023.