Class-Choice Problems and the Harvey Mudd Algorithm

Cameron Gray

June 11, 2021

1 Introduction

In recent years, Harvey Mudd College (HMC) has taken in an overwhelming number of CS majors, and has had a shortage of CS class spots available because of it. The normal process for registering for classes as soon as they become available, called placement or pre-registration, gives every student a registration time during which they can add classes that aren't yet filled. This is effectively a form of serial dictatorship, where a priority order (the same for all the classes) is chosen for the students, and they each take their turn in choosing classes. However, this isn't ideal, as it can end up with students near the start of the order obtaining up to four classes, while the ones at the end may obtain none. Department (2021)

In order to remedy this problem, the Harvey Mudd CS department has a process called pre-placement, where they use their own mechanism to decide before placement which of the CS classes that students can be placed into. One possible solution to this could again be serial dictatorship, or a modified version running serial dictatorship multiple times, where each student only chooses one class at a time. However, the CS department uses a more complicated mechanism involving Integer Linear Programming (ILP), where for every possible outcome (a student being pre-placed into a class), an integer cost is assigned based on the students' class preferences and their priority. The total cost is then minimized subject to constraints: that no student gets into more than two classes, that no student gets into the same class twice, and that no student gets placed into two classes with conflicting time (a constraint we'll ignore here).

2 Class-Choice Problems

As defined in Pomatto (a), a **school-choice problem** consists of the following entities:

- A finite set I of **students**;
- A finite set S of **schools**.

Each entity has a preference:

- The **students' preferences** consist of linear orders $\succ_I = (\succ_i)_{i \in I}$ over $S \cup \{\varnothing\}$;

- The **schools' priorities** consist of complete and transitive binary relations $\succcurlyeq_S = (\succcurlyeq_s)_{s \in S}$ over I.

There are also constraints:

- The **capacities** are a number $q_s \in \mathbb{N}$ for each school $s \in S$.

A **matching** is defined as a function $\mu : I \to S \cup \{\varnothing\}$ such that $|\mu^{-1}(s)| \leq q_s$ for all $s \in S$, so that no school has more students than its capacity. A number of useful properties are defined in Pomatto (a) for the school-choice problem: whether a matching μ eliminates justified envy, is non-wasteful, is Pareto efficient for the students, and for a mechanism, if it is strategy-proof.

However, because of the students being able to get into multiple classes (up to two in the HMC case), the model in the school-choice isn't sufficient. Here I define a **class-choice problem**. The entities again consist of

- A finite set I of **students**;
- A finite set S of **classes**;

and each entity has a preference:

- The **students' preferences** consist of linear orders $\succ_I = (\succ_i)_{i \in I}$ over $S \cup \{\varnothing\}$;
- The **classes' priorities** consist of linear orders $\succ_S = (\succ_s)_{s \in S}$ over I;

where for our case, the priority order is strict. The constraints are:

- The **course maximums** are a number $m_i \in \mathbb{N}$ for each student $m_i \in I$;
- The **capacities** are a number $q_s \in \mathbb{N}$ for each school $s \in S$.

A **matching** is defined as a function $\mu : I \to 2^S$ such that $|\mu^{-1}(s)| \leq q_s$ for all $s \in S$, so that no class has more students than its capacity, and such that $|\mu(i)| \leq m_i$, so that no student has more than their course maximum. Here 2^S is the power set of S, the collection of all subsets including the empty set.

For the HMC mechanism ϕ_{HMC}, this requires a **cost function**. This is defined for each student i as a function $c_i : S \cup \{\varnothing\} \to C \subset \mathbb{N}$ where C is a finite subset of the natural numbers consisting of the possible costs. This is required to satisfy two properties: if $s_1 \succ_i s_2$, then $c_i(s_1) \leq c_i(s_2)$; and if $\varnothing \succ_i s$, then $c_i(\varnothing) = c_i(s)$. The first property means that costs increase non-strictly (so that multiple classes can share the same cost) for less preferred classes, and the second ensures that getting a class that the student would prefer not getting a class to has the same cost as the class. Note that these cost functions can depend on the priority order of the students. The total cost is given by

$$c_{net} = \sum_{i \in I} \left(\sum_{s \in \mu(i)} c_i(s) + (m_i - |\mu(i)|) c_i(\varnothing) \right)$$

since the cost for an individual student consists of the costs of each of their classes, plus the cost for each class they did not get towards their maximum. If we let $x_{i,s,k} = 1$ when student

$i \in I$ gets class $s \in S \cup \{\varnothing\}$ (or no class) as the kth class in $\mu(i)$ (ordering that set), and zero else, this becomes an ILP contributors (2021) minimizing

$$c_{net} = \sum_{i \in I} \sum_{s \in S \cup \{\varnothing\}} \sum_{k=1}^{m_i} c_i(s) x_{i,s,k}$$

subject to our constraints.

3 Desirable Properties

Just like the properties for school-choice problems, we can define similar ones for class-choice problems. However, we run into a difficulty: when comparing two matchings $\mu(i)$ and $\nu(i)$, how does i choose which one is better, since they are both sets rather than single classes? An answer is needed in order to define Pareto efficiency, which asks if at least one student is made better off while none are made worse. In generality, given some linear order $\succ$ over a set S, this consists of defining a binary relation we call a **dominance relation** over 2^S, which may not be transitive, complete, or anti-symmetric.

Given two subsets $A, B \in 2^S$, we say that A **strongly dominates** B, $A \succcurlyeq^s B$, if for all $a \in A$ and for all $b \in B$, we have $a \succcurlyeq b$ (unless A is empty, in which we say it never strongly dominates unless B is also empty). We say that A strictly strongly dominates B, $A \succ^s B$, if we have $A \succcurlyeq^s B$ and if for at least one $a' \in A$ and at least one $b' \in B$, we have $a' \succ b'$ (unless B is empty in which we say that strict dominance holds unless A is also empty). This means that *everything* in A is preferred over B, and in the strict case, at least one element is strictly preferred.

Proposition 1. *Both strong dominance and strict strong dominance are transitive, but are not complete.*

Proof. Suppose that $A \succcurlyeq^s B$ and $B \succcurlyeq^s C$. Then for all $a \in A$, $b \in B$, and $c \in C$, we have $a \succcurlyeq b$ and $b \succcurlyeq c$. But linear relations are transitive, so $a \succcurlyeq c$ and so $A \succcurlyeq^s C$. If this is strict strong dominance, then we have some $a' \succ b'$ and $b' \succ c'$, so that $a' \succ c'$.

However, it is not complete: take $S = \{1, 2, 3\}$ with preference $1 \succ 2 \succ 3$, and let $A = \{1, 3\}$, $B = \{2\}$. We have $1 \succ 2$, so cannot have $B \succcurlyeq^s A$, but also have $2 \succ 3$, so cannot have $A \succcurlyeq^s B$. □

There are other possible transitive binary relations. We say A **weakly dominates** B, $A \succcurlyeq^w B$, if there exists some $a \in A$ and if there exists some $b \in B$ such that $a \succcurlyeq b$, unless B is empty, in which we say that A always weakly dominates B. This is strict weak dominance if there are some $a' \in A$ and some $b' \in B$ such that $a' \succ b'$.

Proposition 2. *Weak dominance is complete, but is not transitive.*

Proof. Consider two sets A and B, and any $a \in A$ and $b \in B$. Then either $a \succcurlyeq b$ or $b \succcurlyeq a$ is true since linear relations are complete, and so at least one of $A \succcurlyeq^w B$ or $B \succcurlyeq^w A$ is true, so it is complete.

However, it is not transitive: take $S = \{1, 2, 3, 4\}$ with preference $1 \succ 2 \succ 3 \succ 4$, and let $A = \{3\}$, $B = \{1, 4\}$, and $C = \{2\}$. We have $3 \succ 4$ and so $A \succcurlyeq^w B$, and $1 \succ 2$ and so $B \succcurlyeq^w C$. However, $2 \succ 3$, and so it is not true that $A \succcurlyeq^w C$, and so this is not transitive. □

The intuition behind strong dominance is that if everything in A is preferred to B, A will always be chosen over B. This is because if $A \succcurlyeq^s B$ and $B \succcurlyeq^s A$, then $A = B$, as $a \succcurlyeq b$ and $b \succcurlyeq a$ for all a and b, so $a = b$ for all, and $A = B$ (though this situation can only occur if each has one or zero elements). However, with weak dominance which is chosen might depend on other attributes, for two sets can weakly dominate each other: given $S = \{1, 2, 3\}$ with preference $1 \succ 2 \succ 3$, and let $A = \{1, 3\}$, $B = \{2\}$, we have both $A \succcurlyeq^w B$ and $B \succcurlyeq^w A$. One question is if there's an intuitive binary relation over 2^S that while isn't as strong as strong dominance, still satisfies this property. For example, given $S = \{1, 2, 3, 4\}$, with preference $1 \succ 2 \succ 3 \succ 4$, let $A = \{1, 3\}$, $B = \{2, 4\}$. It seems like A should dominate B, yet it is not by strong domination.

Taking inspiration from how stochastic domination is defined in Pomatto (b), we define **stochastic dominance**, where here we require a preference $\succ$ on $S \cup \{\varnothing\}$. Given A and B, order A's elements by their preferences, so that a_i is the ith favorite element of A. Similarly do the same for B. If the two are of different sizes, insert empty set elements $\{\varnothing\}$ within the smaller one corresponding to the place that the empty set is within the preferences until the two are the same size. Then we say that A stochastically dominates B, $A \succ^d B$, if for all i, $a_i \succcurlyeq b_i$. This is strict stochastic dominance if $a_j \succ b_j$ for some j.

The intuition behind this definition is that if the ith ranked element of A is better than the ith ranked element of B for all i, A will always be chosen since each individual element is made better off. As an example, given $S = \{1, 2, 3, 4\}$, with preference $1 \succ 2 \succ 3 \succ 4 \succ \varnothing$, let $A = \{1, 3\}$, $B = \{2, 4\}$. We see that since $1 \succ 2$ and $3 \succ 4$, $A \succ^d B$. For another example, given $S = \{1, 2, 3\}$, with preference $1 \succ 2 \succ \varnothing \succ 3$, let $A = \{1\}$, $B = \{2, 4\}$. Since $1 \succ 2$ and $\varnothing \succ 4$, we have $A \succ^d B$. We can see that if $A \succcurlyeq^d B$ and $B \succcurlyeq^d A$, then we have $a_i \succcurlyeq b_i$ and $b_i \succcurlyeq a_i$, so $a_i = b_i$ and $A = B$.

Proposition 3. *If both sets are of equal size, strong dominance implies stochastic dominance, and stochastic dominance implies weak dominance.*

Proof. Suppose that $A \succcurlyeq^s B$. Then for all $a \in A$ and $b \in B$, $a \succcurlyeq b$. Thus after ordering, $a_i \succcurlyeq b_i$ for all i since this is true for all elements. So $A \succcurlyeq^d B$. Suppose that $A \succcurlyeq^d B$. Then after ordering, $a_i \succcurlyeq b_i$ for all i. For $i = 1$ we have $a_1 \succcurlyeq b_1$, so this is true for at least one pair, and thus $A \succcurlyeq^w B$. □

It turns out the stochastic dominance is a more general version of the leximax ordering defined in Walter Bossert (1994), what we call here **leximax dominance**. This requires that the empty set be the least preferred element of S. First we define ordered sets $A' = (a_1, \ldots, a_n, \varnothing, \ldots, \varnothing)$ and $B' = (b_1, \ldots, b_m, \varnothing, \ldots, \varnothing)$ so that the non-empty set elements are ordered in decreasing preference, and so that both A' and B' have total size $|S|$ when the empty sets are added. We say that A leximax dominates B, $A \succcurlyeq^l B$, if for all $i = 1, \ldots, |S|$, $a'_i \succcurlyeq b'_i$. We say that two sets are leximax equivalent, $A \sim^l B$, if both $A \succcurlyeq^l B$ and $B \succcurlyeq^l A$, and define strict leximax dominance as $A \succ^l B$ if $A \succcurlyeq^l B$ and it is not true that $A \succ^l B$.

Stochastic dominance reduces to leximax dominance in the case where the empty set is the least preferred element of S, and all the empty sets are added at the end. In our specific class-choice problem, this is a reasonable assumption: no one who gets pre-placed is forced to take a class that they are pre-placed into, and are free to drop it, so there is no reason why they should prefer not getting pre-placed.

Proposition 4. *$A \sim^l B$ if and only if $A = B$.*

Proof. Suppose that $A = B$. Then $A' = B'$ and thus $a'_i \succcurlyeq b'_i$ and $b'_i \succcurlyeq a'_i$. So $A \succcurlyeq^l B$ and $B \succcurlyeq^l A$, and thus $A \sim^l B$. Now suppose that $A \sim^l B$. Then $a'_i \succcurlyeq b'_i$ and $b'_i \succcurlyeq a'_i$, so that $a'_i = b'_i$ and $A' = B'$. Since there are then an equal number of empty sets, this means that $A = B$. □

Proposition 5. *Leximax dominance is transitive.*

Proof. Suppose that $A \succcurlyeq^l B$ and $B \succcurlyeq^l C$. Then we have that $a'_i \succcurlyeq b'_i$ and $b'_i \succcurlyeq c'_i$. Since B' is the same in each comparison (unlike in the general case for stochastic dominance), we have $a'_i \succcurlyeq c'_i$, and so $A \succcurlyeq^l C$. □

There are a number of important properties that leximax dominance satisfies. These are proved in Walter Bossert (1994), and we do not prove them here.

Theorem 1. *Leximax dominance satisfies **simple dominance**: for all $a, b \in S$, if $a \succ b$, then $\{a\} \succ^l \{b\}$.*

This property means that if a student gets a choice between two single classes, they will choose the result with the class that they prefer.

Theorem 2. *Leximax dominance satisfies **simple monotonicity**: for all $a, b \in S$ such that $a \neq b$, $\{a, b\} \succ^l \{a\}$.*

This property means that if a student has an option to take one class, or that one class plus another class, they will choose the latter. This is only realistic in our case because the students aren't forced to take classes and pre-placement is an option: students often times wouldn't prefer six classes over five!

Theorem 3. *Leximax dominance satisfies **independence**: for all $A, B \in 2^S$, for all $s \in S - (A \cup B)$ we have that $A \cup \{s\} \succcurlyeq^l B \cup \{s\}$ if and only if $A \succcurlyeq^l B$.*

This property means that adding or subtracting the same single class from two pre-placement results won't change which result the student chooses. This may not be satisfied in our case, as often times two classes may pair well together.

Theorem 4. *Leximax dominance satisfies **robustness of strict preference**: for all $A, B \in 2^S$, for all $s \in S - (A \cup B)$ we have that if $A \succ^l B$, $a \succ s$ for all $a \in A$, and $b \succ s$ for all $b \in B$, then $A \succ^l B \cup \{s\}$.*

This property means that if a student prefers one class schedule to another, adding a class that the student likes less than any class in either of those schedules to the schedule the student prefers less does not make them prefer it more.

Theorem 5. *Suppose that $\succcurlyeq^r$ is a dominance relation on 2^S that is also transitive and reflexive. Then $\succcurlyeq^r$ satisfies simple dominance, simple monotonicity, independence, and robustness of strict preference if and only if it is leximax dominance, $\succcurlyeq^r = \succcurlyeq^l$. (Theorem 4.1 of Walter Bossert (1994))*

This result says that leximax ordering is unique in that it is the only such ordering with these properties.

Returning to our class-choice problem, suppose that we are given a dominance relation $\succcurlyeq_i^r$ and a strict dominance relation $\succ_i^r$ for each i. In our particular problem, each class has the same priority relation over students, so we assume that $\succ^* = \succ_s$ for all $s \in S$. We also assume that each student can get at most the same number of classes (2 in the HMC case), so that $m^* = m_i$ for all $i \in I$. We can now define some desired properties that require comparisons between sets.

We say that a matching μ **eliminates justified envy** if there is $i \in I$ such that, for some $j \in I$ both $i \succ^* j$ and $\mu(j) \succ_i^r \mu(i)$ hold.

We say that a matching μ is **non-wasteful** if there is no pair $i \in I$ and non-empty set of classes $S' \subseteq S$, $|S'| \leq m^*$ such that both $S' \succ_i^r \mu(i)$ and all classes in S' have an empty seat.

We say that a matching μ is **Pareto efficient for students** if there does not exists some matching ν such that for all $i \in I$, $\nu(i) \succcurlyeq_i^r \mu(i)$, and for some $j \in I$, $\nu(j) \succ_j^r \mu(j)$.

We say that a mechanism ϕ is **strategy-proof** if for every profile of preferences $(\succ_i)_{i \in I}$, there is no such preference relation $\succ_i'$ for agent i in which

$$\phi[\succ_i', \succ_{-i}](i) \succ_i^r \phi[\succ_i, \succ_{-i}](i).$$

4 HMC Algorithm

In order to run the HMC algorithm, coding it into an ILP solver is needed. Code for this is given in the appendix, along with code testing desirable properties under different dominance relations. For the remainder of this paper, we assume that $m^* = 2$, and as noted for the reasons to do with leximax dominance, that getting no class is the least preferred class of each student. For our cost function, we take a linear one of the form

$$c_i^{\text{linear}}(s) = c_i^{\text{preference}}(s) + c_i^{\text{priority}}(s)$$

so that there is cost from both the priority number of the student, and the preferences of the student. For these we assume

$$c_i^{\text{preference}}(s) = C_1(\text{preference}_i(s) - 1)$$
$$c_i^{\text{priority}}(s) = C_2(\text{priority}(i) - 1)$$

where $\text{preference}_i(s)$ is the place in which student i ranks class s, and $\text{priority}(i)$ is rank of the student in priority. The subtraction by one allows it so that the top-priority student getting their top course has no cost. C_1 and C_2 are positive integer constants, and their ratio determines how much preferences cost in comparison to priority. Since priority is used primarily as a tie-breaker, in real-world implementation we likely have $C_1 \gg C_2$, but we do not assume that here.

Theorem 6. *The HMC algorithm does not eliminate justified envy under strong, weak, stochastic, or leximax dominance.*

Proof. Considering a problem with students $I = \{1, 2, 3\}$ and classes $\{s_1, s_2, s_3\}$, each with maximum one student allowed. Suppose that we have the preferences

$$\succcurlyeq_1 : 1, 2, 3, \varnothing$$
$$\succcurlyeq_2 : 1, 2, 3, \varnothing$$
$$\succcurlyeq_3 : 2, 1, 3, \varnothing$$

and the priority

$$\succcurlyeq^* 1, 2, 3.$$

We choose the cost constants $C_1 = 100$ and $C_2 = 1$. The HMC algorithms returns the matching

$$\mu_{HMC}(1) = \{1\}$$
$$\mu_{HMC}(2) = \varnothing$$
$$\mu_{HMC}(3) = \{2, 3\}$$

with cost 1106.

We can see here that Student 2 has justified envy towards Student 3. We have $\mu_{HMC}(3) \succ_2^s \mu_{HMC}(2)$ and $\mu_{HMC}(3) \succ_2^w \mu_{HMC}(2)$ vacuously. Under stochastic/leximax dominance, we compare $2 \succ_2 \varnothing$ and $3 \succ_2 \varnothing$, and so $\mu_{HMC}(3) \succ_2^d \mu_{HMC}(2)$ and $\mu_{HMC}(3) \succ_2^l \mu_{HMC}(2)$. □

Theorem 7. *The HMC algorithm is not strategy-proof under strong, weak, stochastic, or leximax dominance.*

Proof. Consider the problem in the previous proof, except that agent 2 reports the preferences

$$\succcurlyeq_2 : 2, 3, 1, \varnothing$$

The HMC algorithms returns the matching

$$\mu'_{HMC}(1) = \{1\}$$
$$\mu'_{HMC}(2) = \{3\}$$
$$\mu'_{HMC}(3) = \{2\}$$

with cost 1006.

Vacuously again, we have $\mu'_{HMC}(2) \succ_2^s \mu_{HMC}(2)$ and $\mu'_{HMC}(2) \succ_2^w \mu_{HMC}(2)$. Under stochastic/leximax dominance, we have the comparison $3 \succcurlyeq_2 \varnothing$, and both $\mu'_{HMC}(2) \succ_2^d \mu_{HMC}(2)$ and $\mu'_{HMC}(2) \succ_2^l \mu_{HMC}(2)$. □

We next prove a proposition about lexicon domination and cost.

Proposition 6. *If for lexicon dominance, $A \succcurlyeq_i^l B$, and the cost function $c_i(s)$ is strictly increasing with* $\text{preference}_i(s)$*, then the (individual) cost for any $i \in I$ of A is less than B,*

$$\sum_{s \in A} c_i(s) + (m^* - |A|)c_i(\varnothing) \leq \sum_{s \in B} c_i(s) + (m^* - |B|)c_i(\varnothing).$$

If the lexicon dominance is strict, so is this inequality.

Proof. We can put this in the form

$$\begin{aligned}\sum_{s\in A} c_i(s) + (|S| - |A|)c_i(\varnothing) &\leq \sum_{s\in B} c_i(s) + (|S| - |B|)c_i(\varnothing)\\ \sum_{s\in A'} c_i(s) &\leq \sum_{s\in B'} c_i(s)\\ \sum_{i=1}^{|S|} c_i(a'_i) &\leq \sum_{i=1}^{|S|} c_i(b'_i)\\ 0 &\geq \sum_{i=1}^{|S|} (c_i(a'_i) - c_i(b'_i))\end{aligned}$$

now since $a'_i \succcurlyeq_i b'_i$ by definition of lexicon dominance, we have $c_i(a'_i) \leq c_i(b'_i)$ as the cost function strictly increases. If the lexicon dominance is strict then $a'_i \succ_i b'_i$ for (at least) one of these, and so one of the summands must be positive. □

Note that our specific cost function c_i^{linear} is strictly increasing with $\text{preference}_i(s)$. We have the following results:

Theorem 8. *If the cost function $c_i(s)$ is strictly increasing under* $\text{preference}_i(s)$*, the HMC algorithm is non-wasteful under leximax dominance.*

Proof. Suppose that we have a non-empty set of classes $S' \subset S$ such that $S' \succ_l^s \mu(i)$, $|S'| \leq m^*$, and all classes in S' have an empty seat. Define a matching ν so that it assigns i to S' while keeping all other agents in the same matching as μ. The cost for each student other than i, and by the previous proposition, the cost for i decreases, so the net cost decreases. However, this means that the cost was not minimized subject to the constraints, and so μ could not have been given by the HMC algorithm, a contradiction. □

Theorem 9. *If the cost function $c_i(s)$ is strictly increasing under* $\text{preference}_i(s)$*, the HMC algorithm is Pareto efficient for students under leximax dominance.*

Proof. Suppose that is not Pareto efficient under leximax dominance, then there is some matching ν such that for all $i \in I$, $\nu(i) \succcurlyeq_i^l \mu(i)$, and for some $j \in I$, $\nu(j) \succ_j^l \mu(j)$. Now, for each i this means that the cost for them must decrease or stay the same, and for j the cost must decrease, and so the cost decreases overall. This implies that the cost decreases for the algorithm, and so we have a contradiction. □

Note however, the one used in practice is not of the form $c^{\text{linear}}(i)$, as there are only four different costs given for each preference. Consider a more general form of the cost function, where

$$\begin{aligned}c_{net} &= \sum_{i\in I} c_{net,i}(\mu(i))\\ c_{net,i}(\mu(i)) &= \sum_{s\in S\cup\{\varnothing\}} \sum_{k=1}^{m_i} c_i(s)x_{i,s,k}\\ &= \sum_{s\in\mu(i)} c_i(s) + (m_i - |\mu(i)|)c_i(\varnothing)\end{aligned}$$

and we suppose that $c_i(s)$ is a monotonically increasing function of preference(s) and a strictly increasing function of priority(i). Unfortunately, the proposition proving lexicon dominance gives an decrease in cost does not hold, as the inequality may not be strict in the case of strict lexicon dominance. In this case we can have multiple preferences give the same cost, and the algorithm can choose the less preferred one. Note that our linear cost function with $C_1 = 0$ (which we didn't allow earlier) satisfies monotonically increasing still, so we use that in the following results.

Theorem 10. *The HMC algorithm is not non-wasteful nor Pareto efficient for the students under leximax dominance using a cost function $c_i(s)$ only monotonically increasing under* preference$_i(s)$.

Proof. Considering a problem with students $I = \{1, 2, 3\}$ and classes $\{s_1, s_2, s_3\}$, each with maximum one student allowed. Suppose that we have the preferences

$$\succcurlyeq_1 : 1, 2, 3, \varnothing$$
$$\succcurlyeq_2 : 1, 2, 3, \varnothing$$
$$\succcurlyeq_3 : 2, 1, 3, \varnothing$$

and the priority

$$\succcurlyeq^* 1, 2, 3.$$

We choose the cost constants $C_1 = 0$ and $C_2 = 1$, which is monotonically increasing. The HMC algorithms returns the matching

$$\mu_{HMC}(1) = \varnothing$$
$$\mu_{HMC}(2) = \varnothing$$
$$\mu_{HMC}(3) = \{3\}$$

with cost 6.

We can see here that while Student 1 would prefer to get $\{1\}$ than $\varnothing$ under lexicon dominance and class 1 has available seats. So the algorithm is not non-wasteful. Similarly, a matching where Student 1 is assigned to $\{1\}$ would improve Student 1's situation without changing either Student 2 or Student 3, and so this is not a Pareto efficient matching for the students. □

5 Conclusion

In conclusion, the HMC algorithm under lexicon domination satisfies none of the desired properties of eliminating justified envy, being non-wasteful, being Pareto efficient for students, and being strategy-proof. However, this is only because the forces preferences to be grouped into the categories "Excited about taking it", "Interested in taking it", "Open to taking it", and "Can't or won't take it" Department (2021), and if instead the algorithm took in full strict preference rankings for every of the classes, it would become Pareto efficient

for students and non-wasteful. While ranking like this may take considerably more time on the students' parts, ensuring Pareto efficiency for students might be worth that time.

However, we should note that lexicon domination likely is not the greatest model for domination. Likely, dominance relations on subsets of courses depend on the individual student and not just on their single-course preferences. For example, we could have two students with the same single-course preferences where one wants to take two similar courses together, whereas another only wants to take one, and their dominance relations will be different. One remedy to this could be by asking the students about their domination relations, however, this will take $1 + S + \frac{S(S-1)}{2}$ entries as opposed to just S (more generally, $O(S^{m^*})$, which for a large number of courses is infeasible.

Another possibility could be to use a version of deferred acceptance, as at least for the school-choice problem, this does eliminate justified envy and is strategyproof. While I do not explore this in the paper, because the classes all have the same priority over students, it seems reasonable to suspect that this class of mechanisms may end up with the same results as one of the versions of serial dictatorship briefly mentioned earlier. If so, for Harvey Mudd this would mostly mean just giving CS students priority on CS courses and having the pre-placement mechanism be the same as placement. If the same results hold as in the school-choice problem, this won't lead to a mechanism that is Pareto efficient for students, though it may not have other mechanisms Pareto dominating it.

References

contributors, W. (2021). Linear programming — Wikipedia, the free encyclopedia.

Department, H. M. C. (2021). Hmc cs course preplacement (for fall '21 courses). Document for Harvey Mudd CS Students.

Pomatto, L. Ec 117 week5-b: School choice. Course materials for Ec 117, Spring 2021.

Pomatto, L. Ec 117 week6: Random allocation. Course materials for Ec 117, Spring 2021.

Walter Bossert, Prasanta K. Pattanaik, Y. X. (1994). Ranking opportunity sets: An axiomatic approach. *Journal of Economic Theory*, 63:326–345.

6 Appendix: Python Implementation

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import numpy as np
import cvxpy as cp
```

```
import cvxopt as cpt


# In[2]:


NUM_OF_STUDENTS = 3
NUM_OF_CLASSES = 3
MAX_CLASSES = 2 # must make lists manually
x1 = cp.Variable((NUM_OF_STUDENTS,(NUM_OF_CLASSES+1)),integer=True)
x2 = cp.Variable((NUM_OF_STUDENTS,(NUM_OF_CLASSES+1)),integer=True)
preferences = np.array([[1,2,3,0],[1,2,3,0],[2,1,3,0]])
# Each row lists the preference for a student
priority = np.array([1,2,3])
max_students = np.array([MAX_CLASSES*NUM_OF_STUDENTS,1,1,1])
# Maximum students for each class

PREFERENCE_MULTIPLIER = 100 # C_1
PRIORITY_MULTIPLIER = 1 # C_2

def cost_func(i,s,pref,prior):
    """Takes in the student, class and their
    preferences and preferences, returns cost"""
    this_pref = pref[i-1]
    school_loc = np.argwhere(this_pref==s).flatten()[0]
    empty_loc = np.argwhere(this_pref==0).flatten()[0]
    c_pref = school_loc*PREFERENCE_MULTIPLIER
    c_prior = (np.argwhere(prior==i).flatten()[0])*PRIORITY_MULTIPLIER
    return c_pref+c_prior

cost = np.fromfunction(np.vectorize(
    lambda a, b: cost_func(a+1,b,preferences,priority)),
                           (NUM_OF_STUDENTS,NUM_OF_CLASSES+1) ,dtype=int)
print("Cost Matrix:")
print(cost)
diag_wzero = np.diagflat(np.ones(NUM_OF_CLASSES+1,dtype=int))
diag_wzero[0][0] = 0

constraints = [x1>=0,x1<=1,x2>=0,x2<=1,
               cp.sum(x1,axis=1)==1,cp.sum(x2,axis=1)==1,
               # Must be in at least one class or no class
               (x1+x2)@diag_wzero<=1,
               # Cannot be in the same class twice
               # but can be in no class twice
              cp.sum(x1,axis=0) + cp.sum(x2,axis=0)<=max_students]
```

```
# Must obey class size limits
obj = cp.Minimize(cp.vec(cost)@cp.vec(x1)+cp.vec(cost)@cp.vec(x2))

# Solving the problem
prob = cp.Problem(obj, constraints)
prob.solve(verbose=False)
print("Status:", prob.status)
print("Total Cost:", prob.value)

def return_text(values):
    classnum = np.argwhere(values==1)[0][0]
    if classnum == 0:
        return "No Class"
    else:
        return "Class " + str(classnum)

for i in range(NUM_OF_STUDENTS):
    print("Student",i+1,"is in",return_text(x1.value[i]),
          "and",return_text(x2.value[i]))

values = np.array(x1.value+x2.value,dtype=int)
print("Values Matrix:")
print(values)


# In[3]:


def compare(a,b,pref):
    """Compares two classes"""
    if np.argwhere(pref==a)[0][0] <= np.argwhere(pref==b)[0][0]:
        return True
    return False

def compare_strict(a,b,pref):
    """Strictly compares two classes"""
    if np.argwhere(pref==a)[0][0] < np.argwhere(pref==b)[0][0]:
        return True
    return False

def strong_domination(A,B,pref):
    """Check if for class lists A and B,
    A strongly dominates B"""
    for i in range(len(A)):
        if A[i] != 0:
```

```
            for j in range(len(B)):
                if B[j] != 0:
                    if compare(i,j,pref)==False:
                        return False
    return True

def weak_domination(A,B,pref):
    """Check if for class lists A and B,
    A weakly dominates B"""
    for i in range(len(A)):
        if A[i] != 0:
            for j in range(len(B)):
                if B[j] != 0:
                    if compare(i,j,pref)==True:
                        return True
    return False

def ordering(A,pref):
    """From a class list, returns a list of the classes
    ordered by preferences"""
    element_number = np.sum(A)
    A2 = np.copy(A)
    order_A = np.zeros(element_number,dtype=int)
    for i in range(element_number):
        for j in pref:
            if A2[j] != 0:
                order_A[i] = j
                A2[j] -= 1
                break
    return order_A

def stochastic_domination(A,B,pref):
    """Check if for class lists A and B, A stochastically
    dominates B. Requires each set to be the same size"""
    if np.sum(A) != np.sum(B):
        raise NotImplementedError(
            "Error: not same size functionality not implemented")
    element_number = np.sum(A)
    order_A = ordering(A,pref)
    order_B = ordering(B,pref)
    for i in range(element_number):
        if compare(order_A[i],order_B[i],pref) == False:
            return False
    return True
```

```
def strong_domination_strict(A,B,pref):
    """Check if for class lists A and B,
    A strictly strongly dominates B"""
    if strong_domination(A,B,pref) == False:
        return False
    for i in range(len(A)):
        if A[i] != 0:
            for j in range(len(B)):
                if B[j] != 0:
                    if compare_strict(i,j,pref)==True:
                        return True
    return False

def weak_domination_strict(A,B,pref):
    """Check if for class lists A and B,
    A strictly weakly dominates B"""
    if weak_domination(A,B,pref) == False:
        return False
    for i in range(len(A)):
        if A[i] != 0:
            for j in range(len(B)):
                if B[j] != 0:
                    if compare_strict(i,j,pref)==True:
                        return True
    return False

def stochastic_domination_strict(A,B,pref):
    """Check if for class lists A and B,
    A strictly stochastically dominates B"""
    if stochastic_domination(A,B,pref) == False:
        return False
    element_number = np.sum(A)
    order_A = ordering(A,pref)
    order_B = ordering(B,pref)
    for i in range(element_number):
        if compare_strict(order_A[i],order_B[i],pref) == True:
            return True
    return False

def lexicon_domination(A,B,pref):
    """Check if for class lists A and B,
    A lexicon dominates B"""
    n = len(np.delete(pref,np.where(pref==0)))
    order_A = ordering(A,pref)
    order_B = ordering(B,pref)
```

```
    for i in range(n-len(order_A)):
        order_A = np.append(order_A,0)
    for i in range(n-len(order_B)):
        order_B = np.append(order_B,0)
    for i in range(n):
        if compare(order_A[i],order_B[i],pref) == False:
            return False
    return True

def lexicon_domination_strict(A,B,pref):
    """Check if for class lists A and B,
    A strictly lexicon dominates B"""
    n = len(np.delete(pref,np.where(pref==0)))
    order_A = ordering(A,pref)
    order_B = ordering(B,pref)
    for i in range(n-len(order_A)):
        order_A = np.append(order_A,0)
    for i in range(n-len(order_B)):
        order_B = np.append(order_B,0)
    if np.array_equal(order_A,order_B):
        return False
    for i in range(n):
        if compare(order_A[i],order_B[i],pref) == False:
            return False
    return True


# In[9]:


def eliminates_justified_envy(vals,prefs,prios,comparison_strict):
    """Checks if justified envy is eliminated"""
    for i in range(NUM_OF_STUDENTS):
        for j in range(NUM_OF_STUDENTS):
            if (j != i and np.argwhere(prios==i+1)[0][0]
                < np.argwhere(prios==j+1)[0][0] and
                comparison_strict(vals[j],vals[i],prefs[i]) == True ):
                print("Student",i+1,"envies_Student",j+1)
                return False
    print("Eliminates_justified_envy")
    return True


# In[ ]:
```