# Better School Choice:
# A User's Guide to GCPS MCC Schools


Andrew McLennan*


February 10, 2025


**Abstract**

This document describes software implementing the GCPS, MCC, and EMCC mechanisms for school choice, which give probabilistic assignments of students to schools that are efficient, relative to the schools' priorities, and highly resistant to manipulation. In addition there is software for generating sample school choice problems, and for passing from a probabilistic assignment to a random deterministic assignment with the same assignment probabilities. These softwares are described informally, but in detail. There are precise instructions for users which do not depend on understanding the code. For those who might wish to work with the code, its large scale structure is described in a way that should allow the reader to penetrate its details.


## 1 Introduction

From time immemorial until about 25 or 30 years ago, each school had a district, and each student was required to attend the school whose district contained her residence, unless she enrolled in a private school. This had various problems, e.g., de facto school segregation echoing residential segregation, but the main issue for us is that it forbids students from trading their assignments in ways that are mutually beneficial.

School choice schemes allow students to attend public schools that they express a preference for. In the schemes discussed here, each student submits a rank ordered list of schools that she


*School of Economics, University of Queensland, a.mclennan@economics.uq.edu.au

would like to attend. Each school has an eligibility rule (it may be a single sex school, and a selective school may have a minimum test score or GPA) and in addition it may have a ranking of eligible students called a *priority* that might, for example, give preference for high test scores, minority status, or students who live nearby.

A *school choice mechanism* is an algorithm that takes the preferences and the priorities as inputs and outputs an assignment of each student to one of the schools she ranked that is *feasible*, in the sense that the number of students assigned to each school is not greater than the school's capacity. The most commonly used school choice mechanism, known as *deferred acceptance*, has some undesible features, which we describe in Subsection 1.5.

The academic paper "Efficient Computationally Tractable School Choice Mechanisms" (joint with Shino Takayama and Yuki Tamura) describes three new school choice mechanisms, in formal mathematical detail: the *generalized constrained probabilistic serial* (GCPS) *mechanism*, the *market clearing cutoffs* (MCC) *mechanism*, and a variant, an *enhanced market clearing cutoffs* (EMCC) *mechanism*. *GCPS MCC Schools* is a software package that implements the algorithms that define these mechanisms, and related algorithms that are required to apply these algorithms, and that allow the algorithms to be tested.

The primary purpose of this document is provide a guide to users of *GCPS MCC Schools*. In the remainder of this section we first describe our mechanisms informally, but in some detail. We then explain a bit more about the history of school choice, the mechanism that is currently most popular, and the problems with this mechanism that our mechanisms solve. The next section explains how to use the software, with step-by-step instructions. The final section describes the code for programmers who would like to have a general overview of its structure, and who might be interested in modifying or extending it in some way.

## 1.1 The GCPS Mechanism

The *probabilistic serial* (PS) *mechanism*, due to Bogomolnaia and Moulin (2001), is a mechanism for probabilistic allocation of objects. In the simplest instance there is a set $I$ containing $n$ agents and a set $O$ containing $n$ objects, each agent must receive exactly one of the objects, each object can be assigned to only one agent, and each agent has a strict preference ordering of the objects. The goal is to come up with a probability distribution over possible assignments that respects the agents' preferences and is fair.

*Random priority* is a common method of dealing with such problems. An ordering of the agents is chosen randomly, with each of the $n!$ orderings having probability $1/n!$. The first agent claims her favorite object, the second agent claims her favorite of those that remain after the first agent's choice, and so forth.

Bogolmonaia and Moulin describe the PS mechanism in terms of "simultaneous eating." Each object is thought of as a cake of unit size. At time zero each agent starts eating (that is, accumulating probability) from the cake corresponding to her favorite object, at unit speed. Whenever a cake is exhausted, the agents who were eating that cake switch to their favorite cakes from among those that have not yet been exhausted. At time one all cakes have been exactly allocated, and each agent has a probability distribution over the objects. A matrix of assignment probabilities with these properties is said to be *bistochastic*. As we will describe in detail later, for any bistochastic matrix of assignment probabilities it is possible to compute a probability distribution over deterministic assignments that realizes all of the assignment probabilities.

The main advantage of the PS mechanism, in comparison with random priority, is that it is more efficient. For a given agent, one probability distribution over $O$ *stochastically dominates* a second distribution if, for each object $o$, the probability of receiving an object that is better than $o$ under the first distribution is at least as large as the probability under the second distribution. The domination is *strict* if the two distributions are different. When one distribution stochastically dominates a second distribution, we can think of the difference as a matter of moving a certain amount of probability to better objects, so it is a clearcut improvement no matter how much or how little one cares about the differences between particular objects.

We say that a bistochastic matrix of assignment probabilities is $sd$*-efficient* if there does not exist a second such matrix that gives each agent a distribution over $O$ that stochastically dominates the distribution given by the first matrix, with strict domination for some agents. A surprising finding of Bogolmolnaia and Moulin is that random priority can produce a matrix of assignment probabilities that is not $sd$-efficient. In contrast, the matrix of assignment probabilities produced by the PS mechanism is always $sd$-efficient.

The PS mechanism was generalized by Budish et al. (2013) to the *generalized probabilistic serial* mechanism, and further generalized to the *generalized constrained probabilistic serial* (GCPS) mechanism by Balbuzanov (2022). In the context of school choice, the elements of $I$ are *students* and the elements of $O$ are *schools*. In school choice the schools have *priorities*, which are preferences over the students. The GCPS mechanism is appropriate when the schools' priorities are *dichotomous*: for each school and student, the student is either eligible to attend the school or she is not, and each school gives equal consideration to all of its eligible students.

Eligibility may be affected by gender, residential location, or test scores in the case of selective schools. We also assume that each student has a *safe school* which will certainly admit her if she is not admitted to a school she prefers, so she is not eligible to be assigned to any school that is worse for her than her safe school. Thus the given data of the GCPS mechanism consists of the capacities of the various schools and, for each student, the set of schools she is eligible for, ranked from best to worst.

A *feasible allocation* is an assignment, to each student-school pair, of a probability that the student receives a seat in the school, such that:

(a) the probability is zero if the student is not eligible to attend the school;

(b) for each student, the sum of her probabilities is one;

(c) for each school the sum of its probabilities is not greater than its capacity.

(Later we will see that for any feasible assignment there is a probability distribution over deterministic feasible assignments that realizes the assignment probabilities.) The GCPS mechanism is also a matter of simultaneous eating: at each time between zero and one, each student consumes probability of the best school that is available to her at that time, at unit speed.

A school may become unavailable if its capacity is exhausted, but it can also happen that at a certain time, a set of schools has only enough capacity to meet the needs of the students who are not eligible to attend schools that are outside the set and are still available. Formally, a pair $(J, P)$ of subsets $J \subset I$ and $P \subset O$ is *critical* at time $t$ if, for every $i \in J$, the only schools that are still available to $i$ are contained in $P$, and the remaining capacity of the schools in $P$ is just enough to meet the remaining demand of the students in $J$. When this happens, students outside of $J$ become ineligible to consume additional probability of schools in $P$. At this point the allocation process splits into two subprocesses of the same form, one for the students in $J$ and the schools in $P$, and the other for the students outside of $J$ and the schools outside of $P$. Thus the algorithm is recursive, calling itself again and again as it descends through a tree of subproblems.

Since the GCPS mechanism detects each critical pair and revises the students' sets of available schools in response, if there is a feasible allocation, the allocation it computes at time one is feasible. (This is a consequence of a significant theorem, so it should not be obvious at this point.) Since the number of subsets of $O$ is $2^{|O|}$, where $|O|$ is the number of elements of $O$, one might expect that the complexity of the computation is exponential, but it turns out that there is a bit of algorithmic magic that gets around this problem. The GCPS allocation is also $sd$-efficient; again, this is a significant theorem, and not something you should expect to see right away. As we will describe in detail later, the allocations produced by the mechanism that is currently most commonly used for school choice are not $sd$-efficient.

The *generalized constrained probabilistic serial* (GCPS) mechanism is implemented in the command `gcps`, which has the schools' capacities and the students' eligibilities and declared preferences as inputs. If there is no feasible allocation, it will output an error message to that effect and quit. If there is a feasible allocation, it computes the particular allocation described above.

Insuring that there is a feasible allocation is the responsibility of the user. One way to do this is to choose an assignment of safe schools that is feasible, in the sense that no school is assigned more students than its capacity. In particular, a common method called *neighborhood schools* is to have each student's safe school be the one whose district contains her residence. Of course many variations on this basic idea are possible.

There is a lot more to say about the GCPS mechanism, but for the time being we only introduce one new concept. A mechanism is *strategy-proof* if reporting a preference that is different from your true preference is never beneficial. Strategy-proof mechanisms are seemingly straightforward, and don't punish lack of strategic sophistication. A mechanism that is not strategy-proof is perhaps not what it seems to be on its surface, at best simply because it allows some students to get away with things, and at worst because it creates a tricky game in which each student has to anticipate how others will try to manipulate it.

It is easy to see that random priority is strategy-proof: when your time to choose comes, there is nothing better than to choose your favorite from the objects that remain. A simple example shows that the PS mechanism is not strategy-proof. Suppose that there is one other person for whom your favorite object is their favorite object, and there are two other people for whom your second favorite object is their favorite. Suppose also that no one else has any interest in either of these objects, and the two people whose favorite is your second favorite object have no interest in your favorite object. If you report your true preference you will divide your favorite object with the other person who likes it until time $\frac{1}{2}$, at which point your second favorite object will also have been fully allocated. If your reported preference flips the top two objects, you will share your second favorite object with the two others who like it until time $\frac{1}{3}$, after which two thirds of your favorite object will remain, and you will get half of it. In short, if you tell the truth your total probability of your two favorite objects will be $\frac{1}{2}$, and if you misreport your total probability of your two favorite objects will be $\frac{2}{3}$. If your main concern is to maximize the probability of receiving one of your top two objects, this can be beneficial.

Although the GCPS mechanism is not strategy-proof, it does satisfy a weaker condition that is good enough for practical purposes. This concept considers applying a mechanism to a sequence of problems with an increasing number of agents. The *type* of an agent is her preference ordering of the objects. We assume that each agent's beliefs about the types of the other agents are that they are independent draws from a distribution over a finite set of types that assigns positive probability to the agent's own type. The mechanism is *strategy-proof in the large* (Azevedo and Budish, 2019) if the maximum expected benefit of manipulation decreases asymptotically to zero as the population size goes to infinity. Among other things, strategy-proofness in the large assures us that we have not failed to notice a "one weird trick" manipulation that provides significant gains even when the population is large. Azevedo and Budish give several examples of mechanisms that are not strategy-proof, but are strategy-proof in the large, and which work well in practice.

In the GCPS mechanism, if the times at which certain schools become unavailable to certain agents are fixed, you do best by simply consuming your favorite available school at each moment. Therefore any benefit from manipulation is the result of changing the times at which certain schools become unavailable to certain agents. These times are determined by the distribution over types in the population, and when the population is large, the agent's beliefs about the probability distribution over such distributions is only slightly affected by the agent's own preference declaration.

## 1.2 The MCC Mechanism

The MCC and EMCC mechanisms are appropriate when the schools' priorities are not dichotomous. The inputs for the these mechanisms are *school choice problems*. A school choice problem has given sets of students and schools. For each student there is a strict preference ordering of some of the schools, with her safe school at the bottom of the list. Her *eligible schools* are those she ranks. For each student and each of her eligible schools, the student's *priority* at the school is a nonnegative integer.

We may imagine that the school "prefers" to admit students with higher priority. The pathways by which honoring the priorities results in a better utilization of society's educational resources are often not obvious and straightforward, but we will simply take the desirability of doing so as a given.

A *coarse cutoff* for a school $o$ is a nonnegative integer $C_o$, a *fine cutoff* for $o$ is a nonnegative real number $c_o$, and $c_o$ *refines* $C_o$ if $C_o \leq c_o < C_o + 1$. Evidently a fine cutoff refines a unique coarse cutoff. A *coarse cutoff profile* $C = (C_o)_{o \in O}$ is a specification of a coarse cutoff for each school, and a *fine cutoff profile* $c = (c_o)_{o \in O}$ is a specification of a fine cutoff for each school. For such a $c$ there is a unique $C$ such that for each $o$, $c_o$ refines $C_o$.

If $c$ is a profile of fine cutoffs and $C$ is the profile of coarse cutoffs that it refines, a feasible allocation *fulfills* $c$ if:

(a) when a student's priority at school $o$ is less than $C_o$, the probability that the student is assigned a seat in the school is zero;

(b) when a student's priority at school $o$ is equal to $C_o$, the probability that the student is assigned to $o$ is not greater than $1 - (c_o - C_o)$;

(c) when a student's priority at school $o$ is greater than $C_o$, the student's consumption of the school is not rationed, in the sense that the probability that the student is assigned a seat in a school she likes less is zero;

(d) when school $o$'s capacity is not fully utilized, $c_o = 0$.

Fix a profile of fine cutoffs $c$, and let $C$ be the profile of coarse priorities that $c$ refines. We compute a student's *demand* by having her consume as much of her favorite school as she is allowed to, then as much of her second favorite school as allowed, and so on until she has one unit of probability. That is, the student's consumption of her favorite school $o_1$ is $0$ if her coarse priority at $o_1$ is less than $C_{o_1}$, $1 - (c_{o_1} - C_{o_1})$ if her coarse priority at $o_1$ is $C_{o_1}$, and $1$ if her coarse priority at $o_1$ is greater than $C_{o_1}$. Her consumption of her second favorite school $o_2$, her third favorite school $o_3$, and so forth, are defined similarly, except that she stops consuming when her total consumption reaches one unit of probability. Thus she is consuming the allowed amount of each school except for the last one with positive consumption, where her consumption is the amount needed to complete a probability distribution.

The *market clearing cutoffs* (MCC) mechanism is implemented in the command `mcc`, which takes the students' preferences and the schools' priorities as inputs and outputs a feasible allocation that fulfills a profile of fine cutoffs. To begin with we compute the students' demands for the profile of fine cutoffs in which each school's fine cutoff is $0$. If no school has excess demand then we are done, but otherwise we iterate as follows. For each school with demand greater than its capacity we compute the fine cutoff for that school that would reduce the demand, as previously computed, enough to equate supply and demand for seats in that school. This gives a second profile of fine cutoffs, for which we compute the students' demands. For each school $o$, the increase in the other schools' cutoffs increases demand for $o$, so again there may be schools with excess demand, and again we compute the fine cutoff for each school that would equate supply and demand for seats in that school if all other schools' fine cutoffs stayed the same, which gives a third profile of fine cutoffs. Iterating in this way need not converge in finitely many steps, but it does converge geometrically (i.e., with exponential decay of excess demands) and `mcc` outputs the allocation of demands when the total excess demand is zero to within some tolerable bound.

The MCC mechanism is strategy-proof in the large. If the fine cutoffs of the schools are given, utility is maximized by the demand of the true preferences. Therefore any benefit of manipulation is the result of its effect on the fine cutoffs. The fine cutoffs are determined by the distribution of types in the population, and when the population is large, and the agent's beliefs about the distribution of types are as described earlier, the agent's beliefs about the probability distribution of the distribution of types is insensitive to the agent's own declaration.

## 1.3 Enhanced MCC Mechanisms

Suppose that $c$ is the profile of fine cutoffs that the MCC allocation fulfills, and $C$ is the profile of coarse cutoffs that $c$ refines. It is possible that the MCC allocation is strictly $sd$-dominated by another feasible allocation that fulfills $c$. For example, suppose the priorities of Anne and Bob at school $A$ are both $C_A$, their priorities at school $B$ are both $C_B$, and Anne and Bob both receive $1 - (c_A - C_A)$ units of school $A$ and $1 - (c_B - C_B)$ units of school $B$. If Anne prefers $A$ to $B$ while Bob prefers $B$ to $A$, then they can achieve $sd$-dominating probability distributions by having Anne give Bob some of her probability of $B$ in exchange for an equal amount of Bob's probability of $A$. In conjunction with the probability distributions for the other students given by the MCC allocation, this gives an allocation that strictly $sd$-dominates the MCC allocation and also fulfills $c$.

Longer cycles of exchange are also possible. In fact a feasible allocation that fulfills $c$ is not strictly $sd$-dominated by another feasible allocation that fulfills $c$ if and only if no such trading cycle is possible. There are many algorithms that pass from a feasible allocation that fulfills $c$ to such an allocation that is not strictly $sd$-dominated by another feasible allocation that fulfills

$c$. The idea is to repeatedly identify and execute such cyclic trades until no further possibilities remain. An *enhanced MCC mechanism* is a mechanism that first computes the MCC allocation, and the profile of coarse cutoffs $c$ that it fulfills, and then uses such an algorithm to compute a feasible allocation that fulfills $c$ and is not strictly $sd$-dominated by any other feasible allocation that fulfills $c$. The command `emcc` implements one such mechanism.

It seems likely that an enhanced MCC mechanism may fail to be strategy-proof in the large. A student may benefit, even when the population is large, from elevating a popular school in her reported preference, if she is confident that she will be able to trade probability of that school for probability of a school she really wants. Since an enhanced MCC mechanism is only slightly different from the MCC mechanism, which is strategy-proof in the large, we do not regard this as a serious problem.

## 1.4 Generating a Random Deterministic Assignment

Having computed a feasible assignment, which is a matrix of assignment probabilities, the next problem is to generate a random assignment of students to schools that realizes these probabilities. Doing this is called *implementation* by Budish et al. (2013). The command `purify` implements the special case, for our context, of the algorithm they propose for this. We now briefly explain the key idea.

We begin with a feasible assignment $m$ with entries $m_{io}$. The algorithm works by transitioning from $m$ to a feasible assignment $m^\alpha$ with probability $\frac{\beta}{\alpha+\beta}$ and transitioning from $m$ to a feasible assignment $m^{-\beta}$ with probability $\frac{\alpha}{\alpha+\beta}$, where $\frac{\beta}{\alpha+\beta}m^\alpha + \frac{\alpha}{\alpha+\beta}m^{-\beta} = m$. All the entries that are integral in $m$ are integral in $m^\alpha$ and $m^{-\beta}$, and all the schools $o$ that have integral total demand $\sum_i m_{io}$ in $m$ also have integral total demand in $m^\alpha$ and $m^{-\beta}$. In addition, either $m^\alpha$ has an integral entry that is not integral in $m$, or there is a school that has integral total demand in $m^\alpha$ but not in $m$, and similarly for $m^{-\beta}$. Evidently repeatedly transitioning in this way leads eventually to a random deterministic assignment with a distribution that averages to $m$.

We now need to explain the construction of $m^\alpha$ and $m^{-\beta}$. An *undirected graph* is a pair $G = (V, E)$ where $V$ is a finite set of *nodes* and $E$ is a set of *edges*, where each edge is an unordered pair of distinct nodes. We form the undirected graph $G = (V, E)$ whose set of nodes consists of the students, the schools, and an artificial node called the *sink*. The set of edges contains an edge between a student $i$ and a school $o$ if $m_{io}$ is not an integer (that is, neither $0$ nor $1$) and $E$ contains an edge between a school and the sink if the total probability $\sum_i m_{io}$ of assignment to that school is not an integer. If $E = \emptyset$, then every probability in the assignment is either $0$ or $1$, so it is a deterministic assignment, and we are done.

Suppose that $v_1$ and $v_2$ are elements of $V$ such that there is an edge between $v_1$ and $v_2$. We will show, by enumeration of cases, that there is a vertex $v_3$ that is not $v_1$ such that $E$ contains an edge between $v_2$ and $v_3$. If $v_2$ is a student, then $v_1$ is a school such that $m_{v_2 v_1}$ is not an integer, and since the total $\sum_o m_{v_2 o}$ of $v_2$'s assignment probabilities is $1$, there must be another school $v_3$ such that $m_{v_2 v_3}$ is not an integer. If $v_2$ is a school and $v_1$ is a student, then either there is another student $v_3$ such that $m_{v_3 v_2}$ is not an integer or the sum $\sum_i m_{i v_2}$ of assignment probabilities for $v_2$ is not an integer, in which case there is an edge between $v_2$ and the sink. If $v_2$ is a school and $v_1$ is the sink, then the sum of the assignment probabilities to $v_2$ is not an integer, so there is a student $v_3$ such that $m_{v_3 v_2}$ is not an integer. Finally, if $v_2$ is the sink and $v_1$ is a school such that the sum of the assignment probabilities to $v_1$ is not an integer, then, since the sum of all assignment probabilities is the number of students, there is another school $v_3$ such that the sum of assignment probabilities to $v_3$ is not an integer.

Having found a $v_3 \neq v_1$ such that there is an edge between $v_2$ and $v_3$, we can repeat this step to find a $v_4 \neq v_2$ such that there is an edge between $v_3$ and $v_4$. We can continue in this manner, and since $V$ is finite, we will eventually revisit an element of $V$ we have already seen, so we have shown how to construct a path $v_1, \ldots, v_l$ such that $v_l = v_h$ for some $h < l - 2$. Let $k = l - h + 1$, and for $i = 1, \ldots, k$ let $w_i = v_{i+h-1}$. We have constructed $w_1, \ldots, w_k$ such that:

(a) for each $i = 1, \ldots, k-1$, $E$ contains an edge between $w_i$ and $w_{i+1}$, and $E$ contains an edge between $w_k$ and $w_1$.

(b) $w_k \neq w_2$, $w_{i-1} \neq w_{i+1}$ for all $i = 2, \ldots, k-1$, and $w_{k-1} \neq w_1$.

For each $i = 1, \ldots, k$, if $w_i$ is a student and $w_{i+1}$ is a school, then the edge between $w_i$ and $w_{i+1}$ is a *forward edge*, and if $w_i$ is a school and $w_{i+1}$ is a student, then the edge between $w_i$ and $w_{i+1}$ is a *backward edge*. (If $w_i$ or $w_{i+1}$ is the sink, then the edge between $w_i$ and $w_{i+1}$ is neither forward nor backward.) If $w_k$ is a student and $w_1$ is a school, then the edge between $w_k$ and $w_1$ is a forward edge, and if $w_k$ is a school and $w_1$ is a student, then the edge between $w_k$ and $w_1$ is a backward edge.

For $\alpha > 0$ consider the matrix $m^\alpha$ of numbers obtained by increasing $m_{w_i w_{i+1}}$ by $\alpha$ when the edge between $w_i$ and $w_{i+1}$ is a forward edge and decreasing $m_{w_{i+1} w_i}$ by $\alpha$ when the edge between $w_i$ and $w_{i+1}$ is a backward edge. For each student, the edges involving a student can be grouped in pairs, with each pair having one forward edge and one backward edge, so the total assignment probability for the student in $m^\alpha$ continues to be one. If the total probability assigned to some school is an integer, then the edges involving it can also be group in such pairs, so its total assignment probability in $m^\alpha$ is the same as in $m$. Therefore $m^\alpha$ is a feasible assignment if $\alpha$ is sufficiently small.

Similarly, for $\beta > 0$ consider the matrix $m^{-\beta}$ of numbers obtained by decreasing $m_{w_i w_{i+1}}$ by $\beta$ when the edge between $w_i$ and $w_{i+1}$ is a forward edge and increasing $m_{w_{i+1} w_i}$ by $\beta$ when the edge between $w_i$ and $w_{i+1}$ is a backward edge. As above, $m^{-\beta}$ is a feasible assignment if $\beta$ is sufficiently small. Clearly $\frac{\beta}{\alpha+\beta} m^\alpha + \frac{\alpha}{\alpha+\beta} m^{-\beta} = m$. If we choose $\alpha$ to be the smallest positive number such that $m^\alpha$ has an integral assignment probability that is not integral in $m$ or there is a school whose total assignment probability in $m$ is not integral and is integral in $m^\alpha$, and we choose $\beta$ similarly, then all the conditions described above are satisfied.

## 1.5 The Main Competitor: Deferred Acceptance

We now briefly describe the history of school choice, the mechanism that is currently most popular, and the reasons that our mechanisms are better. One of the first school choice mechanisms to be used in practice, called the *Boston mechanism* or *immediate acceptance*, requires each student to submit a ranking of the schools. The mechanism first assigns as many students as possible to their top ranked schools, then assigns as many of the remaining students to the schools they ranked second, and so forth.

A big problem with the Boston mechanism is that it is not strategy-proof for the students. For example, suppose there are three high schools, called Harvard High, Yale High, and Cornell High. Harvard High and Yale High each have 200 seats in their entering class, and Cornell High has 600 seats. Almost everyone prefers Harvard High to Yale High, and almost everyone strongly prefers Yale High to Cornell High. If all students state their preferences truthfully, then each has a 20% chance of going to Harvard High, a 20% chance of going to Yale High, and a 60% chance of going to Cornell High. If everyone else is truthful, and you list Yale High as your top choice, then you go there for sure. But everyone can see this, and many people will not be truthful, so before you can figure out what to do, you have to try to guess what others are doing.

The *student proposes deferred acceptance* (DA) mechanism was first proposed in the academic literature by Gale and Shapley (1962), but later people realized that it had already been used, very successfully, for several years to match medical school graduates with residencies. In a seminal article Abdulkadiroğlu and Sönmez (2003) recommend applying it to school choice, and it is now the dominant mechanism for school choice, and is used around the world. DA requires that each school has a priority that strictly ranks all of its eligible students. We will say more later about where these priorities might come from, but for the time being we simply assume they are given.

In the first round of deferred acceptance each student applies to her favorite school. Each school with more applicants than seats tentatively accepts its favorite applicants, up to its capacity, and rejects all the others. In the second round each student who was rejected in the first round applies to her second favorite school, and each school tentatively retains its favorite applicants from those who applied in both rounds, up to its capacity, and rejects the others. In each subsequent round each student who was rejected in the preceeding round applies to her favorite school among those that have not yet rejected her, and each school hangs on to its favorite applicants, from all rounds, up to its capacity, and rejects all others. This continues until there is a round with no rejections, at which point the existing tentative acceptances become the final assignment.

An assignment of each student to some school is *feasible* if the number of students assigned to each school is not greater than the school's capacity. A *blocking pair* for a feasible assignment is a student-school pair $(i, o)$ such that the $i$ prefers $o$ to the school she has been assigned to and $o$ either has an empty seat or has a higher priority for $i$ than some other student that has been assigned to $o$. A feasible assignment is *stable* if there are no blocking pairs. The assignment produced by DA is stable: if $i$ prefers $o$ to the school she has been assigned to, $o$ must have rejected $i$ at some stage, and $o$'s pool of applicants only expanded after that, so $o$ never came to regret this rejection.

In fact DA produces an assignment that is at least as good, for each student $i$, as any other stable assignment. If $i$ is rejected by her favorite school in the first round of DA, then $i$ is not matched with that school in any stable assignment, because there are enough students with higher priority at that school who would certainly block such an assignment if they were not already matched to that school. Now suppose that $i$ is rejected in the second round, either by her favorite school or by her second favorite school. For all the students that the school retains after the second round, the school is either their favorite, or it is their second favorite and they are not matched to their favorite in any stable assignment, so each of them would block an assignment of $i$ to that school if they were not already matched to that school. In general, whenever $i$ is rejected by a school, each student the school retains has higher priority than $i$ and is not matched to a school she prefers in any stable assignment.

It turns out that DA is strategy-proof for the students. The proof of this is rather hard. For the curious, we will go through it anyway, but nothing later on depends on you understanding it, and you should feel free to skip it if you don't like this sort of stuff.

It will be somewhat easier to work with Gale and Shapley's original, more romantic, setting of one-to-one matching of boys and girls, with the boys proposing. (You can think of each student as a boy and each seat in each school as a girl.) Let $B$ and $G$ be the sets of boys and girls. A *matching* is a function $\mu\colon B \cup G \to B \cup G$ such that $\mu(b) \in G \cup \{b\}$ for each boy $b$, $\mu(g) \in B \cup \{g\}$ for each girl $g$, and $\mu \circ \mu$ is the identity function. (You have exactly one partner, and your partner's partner is yourself.)

A matching is *stable* if there is no pair consisting of a boy and a girl who prefer each other to their partners in the matching, and also no one is matched to a partner that is worse for them than being matched to themself. For the sake of simplicity we assume that there are at least as many girls as boys, and that everyone prefers any partner of the opposite sex to being alone, so every boy is matched with a girl in any stable matching.

Let $\mu$ denote the DA matching when everyone reports their true preference. The proof is by contradiction: we assume the desired conclusion is false and show that that assumption, in conjunction with the given conditions, implies something that is impossible. So, we suppose that there is a boy Albert who, when he reports some false preference, induces a DA matching $\mu'$ that is better for him. Let $R$ be the set of boys who prefer their partner in $\mu'$ to their partner in $\mu$. Since Albert is an element of $R$, $R$ is nonempty. Let $S = \mu'(R)$ where $\mu'(R) = \{ \mu'(b) : b \in R\}$ be the set of partners, in $\mu'$, of boys in $R$.. Of course $\mu'(S) = R$.

We claim that $\mu(S) = R$. Let Beth be an element of $S$. Then there is an element of $R$, say Abe, such that $\mathrm{Beth} = \mu'$(Abe). Let Carl $=$ $\mu$(Beth), and let Doris $=$ $\mu'$(Carl). Since Abe has different partners in $\mu$ and $\mu'$, Beth has different partners in $\mu$ and $\mu'$, so Abe and Carl are different. Since Abe prefers Beth to his partner in $\mu$ and $\mu$ is stable, Beth must prefer Carl to Abe. Among other things, this implies that Carl is a boy and not Beth herself. Since Beth's partners in $\mu$ and $\mu'$ are different, Carl has different partners in $\mu$ and $\mu'$, so Beth and Doris are different. Since Beth prefers Carl to Abe, if Carl and Albert are different, then the stability of $\mu'$ (with respect to the preferences modified by Albert's manipulation) implies that Carl prefers Doris to Beth, so Carl is an element of $R$. Of course if Carl is Albert, then Carl is an element of $R$, so we have shown that $\mu(S)$ is a subset of $R$. Since the matching is one-to-one, it follows that

$\mu(S) = R$.

Under DA for the true preferences, leading to $\mu$, there is a last round in which an element of $R$ makes a proposal. Since every boy in $R$ prefers his partner in $\mu'$ to his partner in $\mu$, every girl in $S$ has already rejected her partner in $\mu'$ prior to this round. Let Don be one of the elements of $R$ who proposes in this round, and let Ella be the girl he proposes to. When Don proposes to Ella, she is holding a proposal from a boy Fred who she rejects in favor of Don. Since Fred has more proposing to do, Fred is not an element of $R$ and thus Fred is not Ella's partner in $\mu'$. Since Ella rejected her partner in $\mu'$ on her way to holding a proposal from Fred, she prefers Fred to her partner in $\mu'$. Since Fred was rejected by Ella, Fred prefers Ella to his partner in $\mu$, and since Fred is not in $R$, Fred weakly prefers his partner in $\mu$ to his partner in $\mu'$, so Fred prefers Ella to his partner in $\mu'$. Thus Ella and Fred are a blocking pair for $\mu'$. This contradiction of the stability of $\mu'$ completes the proof.

It turns out that DA is not strategy-proof for the girls. It can happen that when Alice is holding a proposal from Harry and receives a proposal from Bob, who she prefers to Harry, she might nevertheless do better by rejecting Bob if the result is that Bob proposes to Carol, who then dumps David, after which David proposes to Alice, which is what Alice really wanted all along.

If you understood all of these arguments, congratulations! The main reason for presenting all this theory, about a mechanism that isn't even one of the ones the software implements, is to explain why students, parents, and school administrators find DA extremely confusing. The strategy-proofness of DA for students was discovered two decades after Gale and Shapley's paper, so it should come as no surprise that students and parents do not understand it. Experimental studies find that misreporting of preferences is quite common. Largely for these reasons, the Boston mechanism continued to be used around the world for many years, in spite of its clear cut theoretical inferiority. An important practical advantage of our mechanisms is that they are at least somewhat easier to explain than DA, and in particular the strategy-proofness in the large of GCPS and MCC are much easier to understand than the strategy-proofness of DA.

Where do the schools' priorities come from? We will distinguish between a school's *given priority*, which is a weak ordering of the students that embodies social values, and the school's *final priority*, which is the strict ordering that is an input to the DA algorithm.

At one extreme the given priorities may be *dichotomous*: a student is either eligible or ineligible to attend a school, and each school gives equal consideration to all of its eligible students. In this case each school's final priority is a random strict ordering of its eligible students. (In order to be fair, the possible orderings should each have equal probability.) When DA is applied to such priorities, inefficiencies can result. For example, if Bob likes Carol School and Ted likes Alice School, the mechanism may still match Bob with Alice School and Ted with Carol School if Carol School "prefers" Ted and Alice School "prefers" Bob. Longer cycles of potentially improving trades are also possible. Such inefficiencies have been found to be quantitatively important in practice. In a study of New York City data (Abdulkadiroğlu et al., 2009) it was found that if all schools used the same ordering of students, out of roughly 90,000 students, 1500 students' placements in the DA assignment could be improved without harming anyone, and if different schools used different orderings, 4500 placements could be improved. Our GCPS mechanism avoids such inefficiencies.

A common example of given priorities that express actual social values is that, among eligible students, those with a sibling at the school who live in the school's walk zone have highest priority, those with a sibling at the school who live outside the walk zone have second priority, those without a sibling at the school who live in the walk zone have third priority, and other eligible students have lowest priority. It is common for given priorities to value a residential location near the school, a high test score or grade point average, or minority status. In order to apply DA there must be (usually randomly generated) strict priorities that refine the given priorities, and again the DA allocation may be inefficient, insofar as there can be mutually beneficial trades. Applying our EMCC mechanism avoids such inefficiencies while honoring the given priorities to the extent possible.

Almost all school choice mechanisms limit the number of schools that a student is allowed to rank. With this limitation DA is no longer strategy-proof, and can be quite tricky. In the 2006 New York City High School Match students were allowed to rank 12 schools. Of the roughly 100,000 participants, over 8000 were unmatched after the first round, having not received an offer from any school they ranked. These students participated in a supplementary round, in which they submitted ranked lists of schools that had remaining capacity after the first round. Students who did not receive an offer in the supplementary round were assigned administratively. The overall mechanism is clearly not strategy-proof because it may be best in the first round to rank schools that are "realistic" rather than most preferred, in order to avoid the supplementary round.

A way around these difficulties is to arrange for each student $i$ to have a *safe school* which is guaranteed to not have more students ranked above $i$ than the school's capacity, so that $i$ will not be rejected by the school if she applies to it. Assigning safe schools that the students can be expected to like is consistent with the main goal of school choice, which is to place students in schools they are happy to attend.

Safe schools also make sense for the GCPS, MCC, and enhanced MCC mechanisms, and in fact the structure of the GCPS mechanism solves an algorithmic problem created by safe schools. In abstract theory these mechanisms can be applied in multiround systems by having the safe school in the first round be participation in the second round, having the safe school in the second round be participation in the third round, and so forth. However, our software presumes that there are safe schools, and applying it more generally will probably require at least some modifications by the user.

We suppose that each student knows which school is her safe school, so she only needs to submit a ranked list of the schools she prefers to it. If the number of such schools is not greater than the number she is allowed to rank, the strategy-proofness of DA is restored, and the strategy-proofness in the large of GCPS and MCC are restored, because it is as if she submits a ranking of all schools. We expect safe schools to be popular with students and parents because they simplify the application process, and because the lower bound on the outcome that they provide is intuitively appealing.

# 2 For the User

In the remainder of the main body of this document we look at the software from the point of view of a school administrator (or perhaps the administrator's tech support person) who wants to know how to use the software to come up with an assignment of students to schools. We'll talk only about what you need to do, not how it works or why it works. There will be much more information about those aspects in the Appendices, where we describe the code.

## 2.1 Downloading and Setting Up

Here we give step-by-step instructions for downloading the code and compiling the executables. We will assume a Unix command line environment, which could be a terminal in Linux, the terminal application in MacOS, or some third flavor of Unix. (There are probably easy enough ways to do these things in Windows, but a Windows user can also just get Cygwin.)

First, in a web browser, open the url

```
https://github.com/Coup3z-pixel/SchoolOfChoice/
```

You will see a list of directories and files. Clicking on the filename `gcps_mcc_schools.tar` will take you to a page for that file. On the line beginning with `Code` you will see a button marked `Raw`. Clicking on that button will download the file to your browser. Move it to a suitable directory.

We use the `tar` command to extract its contents, then go into the directory `GCPS` that this action creates and display its contents:

```
$ tar xvf gcps_mcc_schools.tar
$ cd gcps_mcc_schools
$ ls
GCPS_MCC_User_Guide.pdf segment.c segment.h efficient.c
efficient.h emcc.c emcccode.c emcccode.h endpoint.c
endpoint.h gcps.c gcpscode.c gcpscode.h makefile makex.c
makexcode.c makexcode.h mcc.c mcccode.c mcccode.h my.scp
normal.c normal.h parser.c parser.h partalloc.c partalloc.h
pivot.c pivot.h purify.c purifycode.c purifycode.h
pushrelabel.c pushrelabel.h schchprob.c schchprob.h
script.sh sprsmtrx.c sprsmtrx.h subset.c subset.h
```

In addition to a copy of this document, there are number of files ending in `.h`, and for each such file there is a corresponding file ending in `.c`. There are also a number of `.c` files with no corresponding `.h` file. There is a file called `makefile`, and there is a file `my.scp`, which is an input file that we describe below. Finally, there is a file `script.sh` which (as we will see) illustrates how the executables can be combined.

To compile the executables we need the tools `make` and `gcc`, and we can check for their presence using the command `which`:

```
$ which make
/usr/bin/make
$ which gcc
```

```
/usr/bin/gcc
```

If you don't have them, you will need to get them. Assuming all is well, we issue the command `make`, which tells the computer to execute the commands specified in the `makefile`, and we see the text that the command directs to the screen:

```
$ make
gcc -I. -Wall -Wextra -g -c normal.c
gcc -I. -Wall -Wextra -g -c parser.c
gcc -I. -Wall -Wextra -g -c subset.c
gcc -I. -Wall -Wextra -g -c schchprob.c
gcc -I. -Wall -Wextra -g -c partalloc.c
gcc -I. -Wall -Wextra -g -c pushrelabel.c
gcc -I. -Wall -Wextra -g -c pivot.c
gcc -I. -Wall -Wextra -g -c endpoint.c
gcc -I. -Wall -Wextra -g -c segment.c
gcc -I. -Wall -Wextra -g -c efficient.c
gcc -I. -Wall -Wextra -g -c purifycode.c
gcc -I. -Wall -Wextra -g -c sprsmtrx.c
gcc -I. -Wall -Wextra -g -c gcpscode.c
gcc -o gcps gcps.c normal.o parser.o subset.o schchprob.o partalloc.o
pushrelabel.o pivot.o endpoint.o segment.o efficient.o purifycode.o
sprsmtrx.o gcpscode.o -static-libasan -lm
gcc -I. -Wall -Wextra -g -c mcccode.c
gcc -o mcc mcc.c mcccode.o efficient.o partalloc.o subset.o normal.o
parser.o schchprob.o pushrelabel.o sprsmtrx.o -static-libasan -lm
gcc -I. -Wall -Wextra -g -c emcccode.c
gcc -o emcc emcc.c emcccode.o mcccode.o efficient.o partalloc.o
subset.o normal.o parser.o schchprob.o pushrelabel.o sprsmtrx.o -static-li
-lm
gcc -o purify purify.c normal.o parser.o subset.o partalloc.o purifycode
sprsmtrx.o schchprob.o -static-libasan -lm
gcc -I. -Wall -Wextra -g -c makexcode.c
```

```
gcc -o makex makex.c normal.o linprog.o subset.o sprsmtrx.o schchprob.o
makexcode.o -static-libasan -lm
```

If you do `ls` again you will see that, in addition to the files we started with, for each `.h` file there is now a `.o` file (these are called *object* files) and there are the executable files `gcps`, `mcc`, `emcc`, `purify`, and `makex`.

Readers who have some experience with `C` programming will understand this quite well. For the rest, it is not necessary to know what is going on, but it may still be interesting, and provide some useful insight. Each line of the output above describes the execution of a command in the `makefile`. One type of line describes the execution of a command that passes from a `.h` file and a `.c` file to a corresponding `.o` file, which is called an *object*. A second type of line describes a command that passes from a `.c` file and a bunch of `.o` files to an *executable*. Each of these commands can be called by typing `make` and the file that is created, so `make normal.o` on the command line calls the command that creates that file. Commands such as `make gcps` not only invoke the construction of that executable, but also the construction of all the object files that are required by that executable. (These must be listed on a line below the line of the command.) A target of `make` can be a list of other targets, and the target `all` is in fact the list of all executables, so `make all` on the command line leads to the construction of the whole shebang. Finally, typing `make` on the command line without any target is interpreted as a request to construct the first target in the `makefile`, which in our case is `all`, so `make` is a shorthand for `make all`.

At the bottom of the `makefile` there is an object `clean` whose construction is a matter of removing all the files created by `make`, as well as any files ending in ~ that the `emacs` editor left behind when it was used to edit a file. Doing `make clean`, then doing `make` again, is fun, and recommended.

Below we describe how to use the executables, and for those who only want to do that, there is no need to know anything about what is going on under the hood.

## 2.2 School Choice Problems

We begin by looking at the file `my.scp`:

```
$ cat my.scp
/* This is a sample introductory comment.  */
```

```
There are 6 students and 2 schools
The vector of quotas is (4,4)
The priority matrix is
0 2
1 2
2 0
2 0
2 0
0 2
The students numbers of ranked schools are
(1,2,1,1,1,2)
The preferences of the students are
1:  2
2:  1 2
3:  1
4:  1
5:  1
6:  1 2
```

This is a *school choice problem* with 6 students and 2 schools. Each school has 4 seats. Each student has a preference ranking of the schools she is eligible to attend, ordered from best to worst as we go from left to right. In this case, and in general, the specification of each student's number of ranked schools is redundant, since it can be determined by looking at the preferences. Each student has a *priority* at each school she is eligible to attend. Looking at student 2, we see that a student's priority at a school she is eligible to attend can be zero.

The workflow of *GCPS MCC Schools* is that one of the executables `gcps`, `mcc`, or `emcc` is applied to a school choice problem, yielding a *feasible allocation*, which is a matrix specifying for each student and school the probability that the student is assigned to the school. Application of the executable `purify` to the allocation then yields a random assignment of students to schools whose probability distribution matches the probabilities given by the allocation.

For a user, say a school district administrator, the primary responsibility is to prepare the school choice problem file with the relevant information. We now explain the format of such files. We recommend file names for school choice problem files that end with `.scp`, but the software does not enforce this.

*GCPS MCC Schools* input files begin with a comment between `/*` and `*/`. This is purely for your convenience. The comment can be of any length, and provide whatever information is useful to you, but it is mandatory insofar as the computer will insist that the first two characters of the file are `/*`, and it will only start extracting information after it sees the `*/`. Removing or replacing the comment (while leaving the `/*` and `*/`) won't change how the file is processed by `gcps`, `lpgcps`, `mcc`, and `emcc`.

The computer divides the remainder of the input file, after the initial comment, into *generalized white space* and *tokens*. Generalized white space includes the usual white space characters (spaces, tabs, and new lines), and in addition '`(`', '`)`', and '`,`' are treated as white space. (If '`(`', '`)`', and '`,`' were not white space characters we would have to write `The vector of quotas is 4 4`.) Tokens are contiguous sequences of characters without any of the generalized white space characters. White space is discarded, so the first stage of parsing the input file reduces it to a sequence of tokens.

Tokens must be either prescribed words, nonnegative integers, positive integers, or student or school tags (a student or school number followed by '`:`'). The formatting rules are very simple and very rigid: everything must be more or less exactly as shown above, modulo white space and the numbers of rows and columns, so, for example, the first line must not be

```
There are 3 students and 1 school,
```

but it could be

```
There are 3 students and   1 schools.
```

If one of the GCPS executables tries to read an input file and finds a violation of the format requirements, it will print a short statement describing the problem and quit.

## 2.3 gcps, mcc, and emcc

We now assume that an input file `my.scp` is in the current directory. We assume that the executable `gcps` is also in this directory.

In the Unix OS the user has a `PATH`, which is a list of directories. When you issue a command from the command line, the first item on the command line is the name of the command, and the computer goes through the directories in the `PATH` looking for an executable with that name. On many Unix's the current directory (denoted by ` . `) is in the `PATH`, in which case the executables in the current directory can be invoked simply by typing the executable name on the command line. However, for security reasons some flavors of Unix do not put the current directory in the `PATH`, in which case you need to tell the computer that that is where you want it to look, and you will need to type `./makex`, `./gcps`, `./mcc`, `./emcc`, or `./purify`.

Therefore the next step could be to issue the command:

```
$ ./gcps my.scp
/* This is a sample introductory comment.  */
There are 6 students and 2 schools
        1:         2:
1:  0.00000000 1.00000000
2:  0.50000000 0.50000000
3:  1.00000000 0.00000000
4:  1.00000000 0.00000000
5:  1.00000000 0.00000000
6:  0.50000000 0.50000000
```

Note that the sum of the entries in each row is 1 and the sum of the entries in each school's column is not greater than that school's quota. An assignment of probabilities with these properties — each student has positive probability only in schools they are eligible for, each student's total assignment is 1, and no school is overassigned — is a *feasible allocation*.

It is aesthetically unfortunate that the results are printed with eight significant digits, but this is necessary because the output of `gcps`, `mcc`, and `emcc` are inputs to `purify`, as we will explain below. The software regards two numbers as "the same" if they differ by less than 0.000001, so 0.3333 and 0.33333333 are different numbers.

Mechanisms like GCPS are usually described in terms of simultaneous eating: each school is thought of as a cake whose size is its capacity, and at each moment during the unit interval of time each student "eats" probability of the favorite cake that is still available to her. In our example each student consumes probability of a seat in her favorite school until time 0.5. At that time the remaining 1.5 unallocated seats in school 1 are just sufficient to meet the needs of students 3, 4, and 5, who cannot consume any other school, so students 2 and 6 are required to switch to consumption of their second favorite schools.

We apply `mcc` to this input file:

```
$ ./mcc my.scp
```

```
/* This is a sample introductory comment.  */
There are 6 students and 2 schools
         1:          2:
1:  0.00000000 1.00000000
2:  0.50000000 0.50000000
3:  1.00000000 0.00000000
4:  1.00000000 0.00000000
5:  1.00000000 0.00000000
6:  0.50000000 0.50000000
```

We get the same output the came from `./gcps my.scp`, but the way this happens is different. The mechanism recognizes that without any restrictions, the total demand for school 1 would be 5, which exceeds school 1's capacity. Therefore it raises the fine cutoff of school 1 to 0.5. This displaces some demand from school 1 to school 2, but not so much that further increases in fine cutoffs are necessary.

If we ran the command `./emcc your.scp`, the software would compute the outcome coming from `./mcc your.scp` and then look around for mutually beneficial trading cycles. Since there aren't any such cycles, `./emcc your.scp` would also give the output shown above.

## 2.4 `purify`

By default the output of the `gcps` goes to the screen, which is not very useful, so

```
$ ./gcps my.scp > my.mat
```

is probably a preferable command because it *redirects* the output to a file `my.mat`, which is created (or overwritten if it already exists) in the current directory by this command. We recommend that files produced by `gcps`, `mcc`, and `emcc` have filenames ending in `.mat` (for *matrix*), but the software does not enforce this.

Having generated the file `my.mat`, which is a matrix of assignment probabilities, the next problem is to generate a random assignment of students to schools that realizes these probabilities. That is, we want to generate a random deterministic feasible assignment of students to schools such that for each student `i` and school `j`, the probability that `i` receives a seat in `j` is the corresponding entry in `my.mat`. Doing this is called *implementation* by Budish et al. (2013), and the algorithm for accomplishing this was described earlier.

Implementation can be accomplished by issuing the command:

```
$ ./purify my.mat
/* This is a sample introductory comment.  */
1 -> 2; 2 -> 1; 3 -> 1; 4 -> 1; 5 -> 1; 6 -> 2;
```

In effect, the computer flips a coin to decide which of students 2 and 6 will be allowed to attend school 1, while the other is required to attend school 2. As with `gcps`, `purify` directs its output to the screen. Thus

```
$ ./purify my.mat > my.pur
```

is probably a more useful command.

## 2.5 makex

Development of this sort of software requires testing on a wide range of inputs, under at least somewhat realistic conditions. The utility `makex` produces examples of input files for `gcps`, `lpgcps`, `mcc`, and `emcc` that reflect the geographical dispersion of schools within school districts with many schools, and the idiosyncratic nature of school quality and student preferences.

We consider a simple example, emphasizing the formatting requirements.

```
$ ./makex
/* This file was generated by makex with 2 schools,
3 students per school, capacity 4 for all schools,
school valence std dev 1.00, idiosyncratic std dev 1.00,
student test std dev 1.00, and 2 nontop priority grades.  */
There are 6 students and 2 schools
The vector of quotas is (4,4)
The priority matrix is
0 2
1 2
2 0
```

```
2 0
2 0
0 2
The students numbers of ranked schools are
(1,2,1,1,1,2)
The preferences of the students are
1:  2
2:  1 2
3:  1
4:  1
5:  1
6:  1 2
```

The output of `makex` is an input for `gcps`, `lpgcps`, `mcc`, and `emcc`, and the output goes to the screen, so it is more useful to redirect it:

```
$ ./makex > my.scp
```

We have already seen that `makex` has four integer parameters: the number of schools, the number of students per school, the common quota (capacity) of all schools, and the number of priority classes at each school. There are also three types of random variables that are independent and normally distributed, with mean zero. The default values of their standard deviations are all 1.0, but these can be reset.

Each school has a normally distributed *valence*. For each student-school pair there is a normally distributed *idiosyncratic match quality*. The student's utility for a school is the sum of the school's valence and the idiosyncratic match quality minus the distance from the student's house to the school. These numbers determine the sudent's ordinal preferences over the schools, and in particular they determine the ordinal ranking of the schools that are weakly preferred to the safe school.

Each student has a normally distributed *test score*. A student's *raw priority* at a school is her test score minus the distance from her house to the school. The raw priorities give an ordinal ranking of the students who have ranked the school. The students for whom the school is the safe school are assigned to the top priority class, and the remaining students are divided, as equally as possible, into the specified number of priority classes. To the extent that equal division is not possible, a lower priority class will have one fewer student than a higher priority class. For example, if there are seven students and three nontop priority classes, and this is the safe school for two students, the remaining five students are divided into two priority class with two students and one class with one student, which is the lowest priority class.

It is possible to run `makex` in several ways. If it is invoked without any other arguments on the command line (which corresponds to `argc` = 1) it is run with the default parameters in the code, specified by the following lines in `makex.c`.

```
nsc = 2;
no_students_per_school = 3;
school_capacity = 4;
school_valence_std_dev = 1.0;
idiosyncratic_std_dev = 1.0;
test_std_dev = 1.0;
no_priority_grades = 3;
```

One can change the values of these parameters by editing the code. For example, to diminish the relative importance of travel costs one can increase `school_valence_std_dev` and `idiosyncratic_std_dev`. As the code is currently configured, it is possible to invoke `makex` with seven other arguments on the command line, resetting all of these parameters, and it also possible to invoke it with four other arguments, resetting the integer parameters without changing the standard deviations. Without really knowing anything about the C programming language, it should be apparent how to create other customized versions of `makex` by editing the source code.

This illustrates an important point concerning the relationship between this software and its users. Most softwares you are familiar with have interfaces with the user that neither require nor allow the user to edit the source code, but to create such an interface here would be counterproductive. It would add complexity to the source code that had nothing to do with the underlying algorithms. More importantly, one of the main purposes of this software is to provide a starting point for further programming effort in adapting these resources to the requirements and idiosyncratic features of particular school choice setting. Our algorithms are not very complicated, and someone familiar with C should hopefully not have a great deal of difficulty figuring out what is going on and then bending it to her purposes. Starting to look at and edit the source code as soon as possible is a first step down that road.

We reiterate that someone who simply wants to use the software as is has no need to worry about any of this.

## 2.6 `script.sh`

To be practically useful, it must be possible to make the commands `gcps`, `mcc`, `emcc`, and `purify` components of larger processes that might also, for example, govern the collection of the schools' and students' data and the dissemination of results. There are many ways to do this, with the programming language `perl` being one option that springs to mind. Shell scripts are an older tool, which is still popular and in widespread use. The file `script.sh` is an example. We can endow it with the power to execute, then call it as an executable:

```
$ chmod +x script.sh
$ ./script.sh
The script began at
Fri 22 Nov 2024 08:12:44 UTC
The script ended at
Fri 22 Nov 2024 08:12:45 UTC
```

This is not the place to describe the intricacies of shell programming, but if you look inside `script.sh` it should be pretty obvious that:

(a) It prints the time.

(b) It then creates directories `TextSCPs`. `TextMATs`, and `TextDETs`;

(c) For each of eight different values of `schno` it:

   (i) invokes `makex` to create a file that is stored in `TextSCPs`;

   (ii) applies `gcps` to create a file that is stored in `TextMATs`;

   (iii) applies `purify` to create a file that is stored in `TextDETs`.

(d) For each of the three directories, it deletes the contents of the directory, then deletes the directory itself.

(e) It prints the time again.

The only visible consequence of all this is that we see the starting and finishing times, so it seems that the real point here is get some sense of the speed of the algorithms.

Depending on your prior knowledge of shell programming, or your willingness to learn about it, perhaps `script.sh` will be a useful starting point for your own explorations.

# A About the Code

As we have mentioned earlier, we hope that our code provides a useful starting point for others, either contributing to the repository at Github, or for people specializing it for applications to districts with idiosyncratic features. We don't expect anyone to try to understand every detail, but we have tried to write and organize things in a way that makes it possible for someone else to figure out the things they need to understand in order to do whatever they want to do.

Before diving into details, here are some general remarks. The code is written in C, which some regard as an archaic language, but it is still often taught as a first language, and it is a prerequisite to C++, which is still in widespread use, so it is about as close to a lingua franca as currently exists in the world of programming. C is also still at the front of the industry pack for execution speed, which is a critical consideration for application of `gcps` to very large school districts. Even though we are not using C++, the code is largely object oriented in spirit, being organized as an interaction of objects that are given by `struct`'s.

Practically speaking, we will assume that the reader knows at least the basics of C, but those languages that give the computer step-by-step instructions are all pretty similar, so even without knowing much, it should mostly be possible to have a good sense of what is going on, and a monkey-see-monkey-do approach to writing your own modifications can go quite far. Much of the time objects are "passed by reference" to functions, which means that instead of passing the object itself, what is passed is a pointer to the object. Understanding the pointer concept of C is certainly a prerequisite to any detailed understanding of the code.

In comparison with languages such as `Python`, C certainly has some disadvantages. Organizing the hierarchical structure of the code using curly brackets rather than indentation makes the code bulky, but this is primarily an aesthetic concern. More serious is the fact that in C the programmer is responsible for explicitly allocating and deallocating memory. As it happens, in the development of the code this turned out to be a major advantage because being able to control when memory is deallocated allowed for the development of versions of the algorithms that did not exhaust the computer's memory.

Historically explicit control of memory has given rise to major headaches, because it was easy to accidently write to or access a part of memory that had not actually been allocated, or to accidently overwrite some previously allocated memory, without the computer complaining at all. Such bugs were notoriously hard to track down. If you look in the `makefile` you will see that for Linux there are the additional `CFLAGS` `-fsanitize=address` and `-g` and the additional `LDFLAGS` `-fsanitize=address` and `-static-libasan`, while for Mac OS the last flag becomes `-static-libsan`. These flags invoke `addresssanitizer`, which results in compilation of executables that check for memory errors such as those described above. On Linux, but not on Mac OS, the executables also check for memory leaks, which are allocated memory that is not deallocated at the end of run time. Of course this checking is a burden that slows execution and eats up additional memory, so you may want to use `addresssanitizer` only while debugging.

Many objects have a destroyer, which frees the memory that stores the object's data, and for many objects there is a way of printing the object. These printing functions provide the format of the output of `makex`, `gcps`, `mcc`, `emcc`, and `purify`, and for other objects the printing functions can be useful for debugging. In all cases the code for these functions is simple, straightforward, and located at the end of the source code files, and printing and destroyer functions will not be mentioned below. The many calls to destroyers, and to `free`, add unfortunate bulk to the code, but when studying the code, the reader can safely ignore these calls, trusting that they are conceptually insignificant, and that the allocation and freeing of memory is being handled correctly.

In the C programming language, an $n$ element array is indexed by the integers $0, \dots, n-1$. We always think of it as indexed by the integers $1, \dots, n$, so the $j^{\text{th}}$ component of `vec` is `vec[j-1]`. Similarly, the $(i, j)$ component of a matrix `mat` is `mat[i-1][j-1]`. While this is perhaps not one of the most appealing features of C, and it certainly adds some bulk to the code, once you get used to it, in a curious way it seems to enhance the readability of the code.

# B The Code

In this Appendix we give brief description of the various parts of the code. These are not systematic, first of all because the situations being described are quite diverse, but also because the intent is merely to give a casual introduction to what is going on, and a bit of information that might help someone decide whether studying the code more deeply might be beneficial, and how one might begin to do so. The ordering of the subsections is, roughly, from basic to advanced, and from simple to complex.

## B.1 `normal.h` and `normal.c`

The functions `min` and `max` compute the minimum and maximum of two doubles.

The function `is_integer` returns 1 (true) if the given double is within one one millionth of an integer and 0 (false) otherwise. In general, throughout the code, two floating point numbers are regarded as equal if they differ by less that one millionth. This prevents rounding error from creating a spurious impression that two numbers differ. Incidently, the reason that the numbers in the output of `gcps`, `mcc`, and `emcc` have many digits is that outputs of these executables must be accurate inputs for `purify`, so `gcps` shouldn't (for example) print 0.99 instead of 0.99999999.

The functions `uniform` and `normal` provide pseudorandom numbers that are uniformly distributed (in $[0,1]$) and normally distributed (for mean 0 and standard deviation 1) respectively.

## B.2 `subset.h` and `subset.c`

One may represent a subset of $\{1,\ldots,\texttt{large\_set\_size}\}$ as an $n$-tuple of 0's and 1's, or as a list of its elements. The first representation is given by `subset`, which, in addition to the $n$-tuple `indicator` of elements of $\{0,1\}$, keeps track of the number `subset_size` of elements of the subset. The second representation is given by `index`, in which `no_elements` is the number of elements of the subset (not the containing set) and `indices` is a strictly increasing `no_elements`-tuple of positive integers. The `index` representation can be much more efficient when we are dealing with little subsets of big sets.

The function `index_of_subset` passes from the first representation to the second, and `subset_of_index` goes in the other direction. (Since an `index` does not know the size of the set it is a subset of, that piece of data is a required argument.) There is no index representation of the empty set, and if `subset_of_index` receives the empty set as an argument, it will complain and halt execution..

An `index_list` is a linked list of subsets in `index` form.

Mostly the functions in `subset.c` have self explanatory titles, with code that is not hard to understand. There may now be some functions that are not used elsewhere, as we have not made an effort to eliminate such functions when they may prove useful later, and are illustrative of what is possible.

## B.3 `sprsmtrx.h` and `sprsmtrx.c`

As larger and larger examples were considered during the development of this software, it turned out that the amount of memory allocated by the compiler to the program could be exhausted unless some steps were taken to avoid this. For example, an allocation is an assignment to each student-school pair of a probability that the student is enrolled in the school, so when there are 100,000 students and 500 schools, this is a matrix with 50,000,000 entries Each student has a relatively small number of schools that she is eligible for, so if we represent an allocation simply as a matrix, it would have many entries that are known a priori to be zero.

In the `struct`s `int_sparse_matrix` and `int_sparse_matrix`, for $i = 1, \ldots,$ `no_rows`, `nos_active_cols[i-1]` is the number of entries in row `i` that are potentially nonzero and `index_of_active_cols[i-1]` is an array of `nos_active_cols[i-1]` integers specifying which columns contain potentially nonzero entries. If $k \leq$ `index_of_active_cols[i-1]`, then `entries[i-1][k-1]` is the entry in column `index_of_active_cols[i-1][k-1]`.

The three things you can do with a matrix are ask what the value of an entry is, set an entry, and add something to an entry. These three functions for each of the two types of sparse matrices take values of `row_no` between `1` and `no_rows` and values of `col_no` between `1` and `no_cols`. (Thus we are *not* following the `C` style letting the `(i,j)` entry of `mat` be `mat[i-1][j-1]`.) These functions make the inner workings of `int_sparse_matrix` and `dbl_sparse_matrix` invisible to the user, who can imagine that these are just normal matrices. (If you try to assign or increment a nonzero value of an entry that must be zero, there will be an error message, and execution will halt.) For very large problems there is possible some scope for speed up by iterating over `k` from `1` to `nos_of_active_cols[i-1]` instead of iterating over `j` from `1` to `no_cols`, but this has not been explored.

## B.4 `schchprob.h` and `schchprob.c`

A *school choice problem* consists of a set of students and a set of schools. For each school `j`, `quotas[j-1]` is the school's capacity. Each student `i` has `no_eligible_schools[i-1]` they can be assigned to, and preferences[i-1] is the list of such schools, ordered from most preferred to least preferred. For student i and school j, priorities[i-1][j-1] (a nonnegative integer) is student i's *priority* at school j. Even if priorities[i-1][j-1] is zero, student i can be assigned to school j if it is one of the schools she ranked.

When the school choice problem is given as an input, in the struct input_sch_ch_prob, the schools' quotas are integers. In the computation of the gcps allocation there are situations in which the schools have been partially allocated and the remaining amounts to be allocated are no longer integers. The computation of the gcps allocation uses the struct process_scp which has floating point quotas and the member time_remaining that keeps track of how much longer the allocation process will continue.

When the gcps computation encounters a critical pair (as we describe in detail later) the computation descends recursively to two subprocesses. The functions left_sub_process_scp and right_sub_process_scp create the process_scp's of the subprocesses.

## B.5 makex.c, makexcode.h, and makexcode.c

The file makex.c contains the main function of makex, which sets the parameters of makex and then calls the function make_example. This function, which is defined in makexcode.c, implements the description of makex given in Subsection 2.5 in a straightforward manner that is easy to understand.

## B.6 partalloc.h and partalloc.c

In a partial_alloc for no_students students and no_schools schools, there is a matrix allocations that specifies an amount allocations[i-1][j-1] of school j to student i, i.e., a probability that $i$ receives a set in $j$, for each i and j. A pure_alloc has the same structure, but now allocations[i-1][j-1] is an integer that should be zero or one, and for each student i there should be exactly one school j such that allocations[i-1][j-1] is one.

A partial_alloc is *feasible* if the total amount of probability assigned to each student is 1 and the total assigned amount of each school is not more than the school's quota. In the gcps computation, in addition to computing the path of the allocation itself, the process computes a path in the set of feasible allocations that is above the path of the allocation. As we mentioned earlier, when computation encounters a critical pair it descends recursively to two subprocesses. The function `reduced_feasible_guide` computes the initial point of the path of feasible allocations for such a subprocess, and the functions `left_feasible_guide` and `right_feasible_guide` call this function to compute the two specific initial points.

The functions in `partalloc.h` are mostly straightforward. We should probably mention that `partial_allocs_are_the_same` tests whether, for all student-schools pairs, the entries of the two inputs differ by less than 0.000001. This avoids a false negative resulting from the two inputs having been computed with different round off errors.

## B.7 parser.c

Two parsing functions `sch_ch_prob_from_file` and `allocation_from_file` are defined in `parser.c`. As their names suggest, these functions read data from files, constructing, respectively, a school choice problem (`sch_ch_prob`) and an allocation (`partial_alloc`). A valid input file has an opening comment, which begins with `/*` and ends with `*/`, and a body. In the body, in addition to the usual white space characters (space, tab, and newline) the characters '`(`', '`)`', and '`,`' are treated as white space. The body is divided into whitespace and tokens, which are sequences of adjacent characters without any white space that are preceeded and followed by white space.

Everything in `parser.c` is easy to understand. The bulk of the actual code is devoted to functions checking that the verbal tokens are the ones that are expected, and quitting with an error message if one of them isn't.

## B.8 purify.c, purifycode.h, and purifycode.c

The code of the algorithm going from a fractional allocation to a random pure allocation whose distribution has the given allocation as its average follows the description in Section 2.4. The `nonintegral_graph` derived from the given allocation is an undirected graph with an edge between a student and a school if the student's allocation of the school is strictly between zero and one, and an edge between a school and the sink if the total allocation of the school is not an integer. The function `graph_from_alloc` has the given allocation as its input, and its output is the derived `nonintegral_graph`.

Especially for large school choice problems, we expect the `nonintegral_graph` to be quite sparse, so it can be represented more compactly, and be easier to work with, if we encode it by listing the neighbors of each node. The `stu_sch_nbrs` member of `neighbor_lists` is a list of `no_students` lists, where the `stu_sch_nbrs[i-1]` are arrays of varying dimension. We set `stu_sch_nbrs[i-1][0] = 0` in order to have a place holder that allows us to not have an array with no entries when `i` has no neighbors. The actual neighbors of `i` are

`stu_sch_nbrs[i-1][1],...,stu_sch_nbrs[i-1][stu_no_nbrs[i-1]].`

The members `sch_no_nbrs` and `sink_sch_nbrs` follow this pattern, except that in the latter case there is just a single list. The member `sch_sink_nbrs` is a `no_schools`-dimensional array of integers with `sch_sink_nbrs[j-1] = 1` if there is an edge connecting `j` and the `sink` and `sch_sink_nbrs[j-1] = 0` otherwise. To pass from a `nonintegral_graph` to its representation as a `neighbor_lists` we apply `neighbor_lists_from_graph`.

A cycle in the `nonintegral_graph` is a linked list of `path_node`'s. The function `find_cyclic_path` implements the algorithm for finding a cycle that we described in Section 2.4. Given a cycle, `bound_of_cycle` computes the smallest "alternating perturbation," in one direction or the other, of the entries of (the pointee of) `my_alloc` that turns some component of the allocation, or some total allocation of a school, into an integer. For such an `adjustment` the function `cyclic_adjustment` updates the allocation, and it calls the functions `student_edge_removal` and `sink_edge_removal` to update `neighbor_lists`. When `graph_is_nonempty(my_lists) = 0` (false) the entries of `my_alloc` are doubles that are all very close to integers, and the function `pure_allocation_from_partial` passes to the associated `pure_alloc`. The function `random_pure_allocation` is the master function that supervises the whole process.

## B.9 `mcc.c`, `mcccode.h`, and `mcccode.c`

The function `MCC_alloc_plus_coarse_cutoffs` computes the MCC allocation. (It also sets the coarse cutoffs, which are an input to the EMCC computation.) It first sets all of the `fine_cutoffs` to zero. It then repeatedly goes through the cycle of computing the demands of the students given the `fine_cutoffs`, the differences `excesses[j-1]` between total demand for school `j` and school `j`'s quota, and setting each `fine_cutoffs[j-1]` to the number that would reduce the computed demand for school `j` to its quota. This continues until the sum `excess_sum` of demands beyond quotas is close enough to zero.

The function `naive_eq_cutoff` computes the `fine_cutoff[j-1]` that would reduce the total demand for school `j` to school `j`'s quota. For a candidate fine cutoff `cand` the demand of student `i` for school `j` is the minimum of the amount given by `i`'s component of `demands` and the maximum demand allowed by `cand` given `i`'s priority at `j`. The total of the students' demands is a nonincreasing piecewise linear function of `cand`. Repeated subdivision is used to compute the point in its domain where the value of this function is school `j`'s quota. We begin with two points `(lower_cand, lower_dmd)` and `(upper_cand,upper_dmd)` in the graph of this function with `lower_cand` less than `upper_cand`, and `lower_dmd` greater than school `j`'s quota, which is in turn greater than `upper_dmd`. The number `new_cand` is the horizontal coordinate of the point on the line segment between these points whose vertical coordinate is `j`'s quota. If the demand `new_dmd` at `new_cand` is less than school `j`'s quota, then we replace `(upper_cand,upper_dmd)` with `(new_cand,new_dmd)`, and if `new_dmd` is greater that school `j`'s quota, then we replace `(lower_cand,lower_dmd)` with `(new_cand,new_dmd)`. This subdivision process is repeated until `new_dmd` is approximately equal to school `j`'s quota, at which point the function returns `new_cand`.

A point of interest is that in this context the acceptable error of approximation is one billionth rather than one millionth. This is in order to avoid `MCC_alloc_plus_coarse_cutoffs` getting into an infinite loop in which it repeatedly computes the same `fine_cutoffs` such that for each `j` the excess is less than a millionth, but the sum of the excesses is greater than a millionth.

The remaining functions in `mcccode.c` (`demand_at_new_cutoff`, `excess_demands`, and `compute_demands`) are defined by straightforward code that computes what the function names lead us to expect.

## B.10 `efficient.h` and `efficient.c`

The code in `efficient.h` and `efficient.c` provides a test of whether an allocation is $sd$-efficient. Strictly speaking, this is not required in order to attain the main goals of the project, but it is helpful in various ways. For example, the `gcps` allocation is $sd$-efficient, so asking whether the allocation that the `gcps` computation produces is $sd$-efficient provides a simple test of whether the computation is correct, insofar it is quite unlikely that mistaken software could somehow produce an allocation that was incorrect but nevertheless $sd$-efficient. In addition, the code required to test $sd$-efficiency is closely related to, and contributes to, the code that passes from the `mcc` allocation to the `emcc` allocation.

An allocation is *nonwasteful* if there is no student-school pair such that the school's quota is not exhausted and the student has a positive probability of going to some school she like less. Obviously an $sd$-efficient allocation is nonwasteful.

An allocation is not $sd$-efficient if there is mutually beneficial trading cycle in which each student gives an amount $\Delta > 0$ of some school in exchange for $\Delta$ of some school they like better, because executing the trades in the cycle gives a dominating allocation. The converse is also true: for a nonwasteful allocation the nonexistence of such a cycle implies that the allocation is $sd$-efficient. To see this, suppose that a nonwasteful allocation is dominated by a second allocation. There must be some student who is receiving less of some school, say $o_1$, in the second allocation than in the first, and they must be receiving more of some school, say $o_2$, that they like better. Since the first allocation is nonwasteful, $o_2$ must be exhausting its quota in the first allocation, so there is a second student who is receiving less $o_2$ in the second allocation than in the first, and who must be receiving more of some school $o_3$ that they like better. Since there are finitely many student-school pairs, continuing in this fashion eventually yields a cycle.

At an allocation, a student `i` could be part of a trading cycle in which she accepts an additional amount of school `j` if she is eligible to attend `j` and she is being allocated a positive amount of some school that she likes less than `j`. The search for a cycle begins by building `accepting_students`, which is a `list_of_students` in which, for each `j`, the list of those students who are able to accept additional `j` is `accepting_students.lists[j-1]`. Beginning with a pair `(i,j)` such that `i` is able to accept additional `j`, we look at each school that is worse than `j` for `i`, and for each such school `accepting_students` gives the students who might accept that school. In this way we obtain a list of student-school pairs that might be next element of a cycle after `(i,j)`. We might call the list the `first_layer`.

This process continues recursively. The main recursive step in `get_new_layer` has two lists of student-school pairs, namely `all_so_far` and `last_layer`, that include all the pairs that have been found to be potential links in a cycle beginning at `(i,j)`. The list `answer` consists of all the pairs that could follow elements of `last_layer` in a cycle, and that have not already occurred in `all_so_far` and `last_layer`. The list `answer` is computed by the function `simple_new_layer`. If `answer` includes the pair `(i,j)`, there is a cycle including the pair `(i,j)`, so the allocation is not $sd$-efficient. If `answer` is the null list, then we can conclude that there is no cycle including `(i,j)`. In this case we remove `(i,j)` from `accepting_students` and repeat the calculation beginning at a different pair. Finally, if answer is not the null list, and does not include (i, j), then the calculation is repeated with all_so_far replaced by the union of all_so_far and last_layer, and last_layer replaced by answer.

If the allocation is not $sd$-efficient, then a cycle will eventually be found, and otherwise all elements of accepting_students will eventually be removed, revealing that the allocation is $sd$-efficient.

### B.11 emcc.c and emcccode.c

The main function in emcc.c reads an input_sch_ch_prob from a file, passes from this to a process_scp, and applies EMCC_allocation to this to get an allocation, which it returns.

The function EMCC_allocation first applies MCC_alloc_plus_coarse_cutoffs to get an allocation alloc_to_adjust and a profile coarse of coarse cutoffs. Eliminating the eligibility of each student at each school whose priority for the student is below the school's cutoff, as given by coarse, gives red_scp, which is a new process_scp. (In this context the standard of efficiency is $sd$-efficiency within the set of feasible allocations for red_scp.)

The procedure then looks for a cycle $(i_1, j_1), ..., (i_k, j_k)$ such that for each $h = 1, \ldots, k-1$, $i_h$ is consuming a positive quantity of $j_h$ in alloc_to_adjust, $i_h$ prefers $j_{h+1}$ to $j_h$, and $i_h$ is eligible for $j_{h+1}$ in red_scp. If a cycle is found, then the maximal mutually beneficial trade allowed by the cycle is executed. This process is repeated until it arrives at an allocation that is $sd$-efficient relative to red_scp.

The process of finding cycles is very similar to what we saw above in efficient.c. We cycle over all pairs (i, j), looking for mutually improving cycles beginning at (i, j). The key recursive function is cycle_or_not which is called in find_cycle_or_show_none_exist and then calls itself recursively. The main difference is that instead of quitting as soon as a cycle is found, the maximal beneficial trades allowed by the cycles are executed.

## C Computing the GCPS Allocation

It should be emphasized that the code for gcps is *by far* the most complex part of the software. To begin with we develop a more detailed theoretical understanding of the GCPS mechanism, as applied to school choice. We consider a fixed school choice problem with set of students $I$ and set of schools $O$. For each $i \in I$ let $\alpha_i \subset O$ be the set of schools that $i$ ranks. For each $j \in O$ let

$\omega_j = \{\, i : j \in \alpha_i \,\}$ be the set of students who might attend $j$. For each $j \in O$ let $q_j > 0$ be the *quota* of school $j$. Usually the given $q_j$ will be an integer, but this is not necessary.

A *feasible allocation* is a point $m \in \mathbb{R}_+^{I \times O}$ such that $m_{ij} = 0$ for all $i$ and $j$ such that $j \notin \alpha_i$, $\sum_j m_{ij} = 1$ for all $i$, and $\sum_i m_{ij} \leq q_j$ for all $O$. Let $Q$ be the set of feasible allocations. Throughout the following discussion we assume that $Q$ is nonempty. As a bounded set of points satisfying a finite system of weak linear inequalities, $Q$ is a polytope[1].

A *possible allocation* is a point $p \in \mathbb{R}_+^{I \times O}$ such that $p \leq m$ for some $m \in Q$. Let $R$ be the set of possible allocations. It is visually obvious that $R$ is also a polytope, and this is not particularly difficult to prove. A much more subtle result is that $R$ is the set of points $p \in \mathbb{R}_+^{I \times O}$ satisfying the inequality

$$\sum_{i \in J_P^c} \sum_{j \in P} p_{ij} \leq \sum_{j \in P} q_j - |J_P|$$

for each $P \subset O$. Here $J_P = \{\, i : \alpha_i \subset P \,\}$ is the set of students who have not ranked any school outside of $P$, and must receive a seat in a school in $P$, and $J_P^c$ is the complement of this set. The inequality says that the total allocation of seats in schools in $P$ to students outside of $J_P$ cannot exceed the number of seats that remain after every student in $J_P$ has been assigned to a seat in a school in $P$. Clearly every point in $R$ satisfies each such inequality. Much more subtle, and difficult to prove, is the fact that these inequalities completely characterize $R$, in the sense that a $p \in \mathbb{R}_+^{I \times O}$ that satisfies all of them is, in fact, an element of $R$.

Recall that the GCPS allocation is $p(1)$ where $p\colon [0,1] \to R$ is the function such that $p(0)$ is the origin and at each time, each student is increasing, at unit speed, her consumption of her favorite school among those that are still available to her, with her other allocations fixed. It may happen that this process simply assigns each student to her favorite school, but the more important possibility is that at some time before 1, say $t^*$, there is a $P \subset O$ such that the inequality above holds with equality at $t^*$ and does not hold at time $t > t^*$. We say that $P$ becomes *critical* at $t^*$.

At this point the process splits into two parts:

(a) assignment of the remaining probability of receiving a school in $P$ to the students in $J_P$;

(b) assignment of additional probability of seats in schools in $P^c$ to the students in $J_P^c$.

These problems are independent of each other, in the sense that each is determined by data that does not affect the other, and each has the form of the original problem, except that now the

[1] A *polytope* is a bounded set defined by some system of finitely many weak linear inequalities.

time remaining $1 - t^*$ may be less that 1. Thus our algorithm is recursive, applying itself to the subproblems that arise in this way. The functions `descend_to_left_subproblem` and `descend_to_right_subproblem` in `gcpscode.c` implement this recursive descent.

The remaining algorithmic problem is the computation of $t^*$ and a set $P$ that becomes critical at that time. One possibility is to simply compute the time at which the inequality above holds with equality for every $P$, then take the minimum time and some $P$ for which the inequality holds with equality at that time. This has been implemented, and works reasonably well if the number of schools is not too large, say 25 or less. But for the largest school choice problems (e.g., NYC with over 500 schools) this approach is completely infeasible.

We now describe a different approach. For each $i$ let $e_i$ be $i$'s favorite element of $\alpha_i$. Let $\theta \in \mathbb{Z}^{I \times O}$ be the matrix such that $\theta_{ie_i} = 1$ and $\theta_{ij} = 0$ if $o \neq e_i$, so for $t \leq t^*$ we have $p(t) = \theta t$. Suppose that for some time $t_0 \in [0, t^*)$ we have computed a piecewise linear $\overline{p}\colon [0, t_0] \to Q$ such that $p(t) \leq \overline{p}(t)$ for all $t \in [0, t_0]$, and in particular we have a $\overline{p}(t_0) \in Q$ such that $p(t_0) \leq \overline{p}(t_0)$.

We will search for an integer matrix $\overline{\theta} \in \mathbb{Z}^{I \times O}$ such that

$$p(t_0) + \theta(t - t_0) \leq \overline{p}(t_0) + \overline{\theta}(t - t_0) \in Q$$

for $t > t_0$ sufficiently close to $t_0$. In this circumstance we let $t_1$ be the largest $t$ satisfying this condition, and we set $\overline{p}(t) = \overline{p}(t_0) + \overline{\theta}t$ for $t_0 \leq t \leq t_1$. After replacing $t_0$ with $t_1$ and $\overline{p}(t_0)$ with $\overline{p}(t_1)$, we can attempt to repeat the calculation. The mechanics of computing $t_1$ and setting up the new version of the problem are encoded in `endpoint.h` and `endpoint.c`.

We now describe the search for a suitable $\overline{\theta}$. Fix $\overline{\theta} \in \mathbb{Z}^{I \times O}$. If $\overline{p}(t_0) + \overline{\theta}\varepsilon \in Q$ for sufficiently small $\varepsilon > 0$, then:

(a) For each $i$ and $j$, if $j \notin \alpha_i$, then $\overline{\theta}_{ij} = 0$.

(b) For each $i$, $\sum_j \overline{\theta}_{ij} = 0$.

In addition, $p_{ij}(t_0) + \overline{\theta}_{ij}\varepsilon \leq \overline{p}_{ij}(t_0) + \overline{\theta}_{ij}\varepsilon \leq 1$ for all $i$ and $j$ and sufficiently small $\varepsilon > 0$ if and only if, for each $i$ and $j$:

(c) If $\overline{p}_{ij}(t_0) = p_{ij}(t_0)$, then $\overline{\theta}_{ij} \geq 0$, and if $o = e_i$, then $\overline{\theta}_{ij} \geq 1$.

(d) If $\overline{p}_{ij}(t_0) = 1$, then $\overline{\theta}_{ij} \leq 0$.

If $\overline{\theta}$ satisfies (a)–(d), then $\overline{p}(t_0) + \overline{\theta}\varepsilon \in Q$ for sufficiently small $\varepsilon > 0$ if and only if, for each $j$:

(e) If $\sum_i \overline{p}_{ij}(t_0) = q_j$, then $\sum_i \overline{\theta}_{ij} \leq 0$.

We begin by defining an initial $\overline{\theta}^0 \in \mathbb{Z}^{I \times O}$ as follows. For each $i$, if $\overline{p}_{ie_i}(t_0) > p_{ie_i}(t_0)$, then we set $\overline{\theta}^0_{ij} = 0$ for all $j$. If $\overline{p}_{ie_i}(t_0) = p_{ie_i}(t_0)$, then we set $\overline{\theta}^0_{ie_i} = 1$, we set $\overline{\theta}^0_{ij_i} = -1$ for some $j_i \in \alpha_i \setminus \{e_i\}$ such that $\overline{p}_{ij_i}(t_0) > p_{ij_i}(t_0)$, and we set $\overline{\theta}^0_{ij} = 0$ for all other $j$. By construction $\overline{\theta}^0$ satisfies (a)–(d).

Assume that $\overline{\theta}$ satisfies (a)–(d), but not (e). We repeatedly adjust this matrix, bringing it closer to satisfying (e) while continuing to satisfy (a)–(d). For $j \in O$ let

$$J(j) = \{\, i \in \omega_j : \text{if } \overline{p}_{ij}(t_0) = p_{ij}(t_0), \text{ then } \overline{\theta}_{ij} > 0, \text{ and if } j = e_i, \text{ then } \overline{\theta}_{ij} > 1 \,\}$$

be the set of $i$ such that decreasing $\overline{\theta}_{ij}$ by one does not result in a violation of (a) or (c). For $i \in I$ let

$$P(i) = \{\, j \in \alpha_i : \text{either } \overline{\theta}_{ij} < 0 \text{ or } \overline{p}_{ij}(t_0) < 1 \,\}$$

be the set of $j$ such that increasing $\overline{\theta}_{ij}$ by one does not result in a violation of (a) or (d).

A *pivot* for $\overline{\theta}$ is a sequence $j_0, i_1, j_1, \ldots, i_h, j_h$ such that $i_1, \ldots, i_h$ are distinct elements of $I$, $j_0, \ldots, j_h$ are distinct elements of $O$, and:

(a') $\sum_i \overline{p}_{ij_0}(t_0) = q_{j_0}$ and $\sum_i \overline{\theta}_{ij_0} > 0$;

(b') $i_g \in J(j_{g-1})$ and $j_g \in P(i_g)$ for all $g = 1, \ldots, h$;

(c') either $\sum_i \overline{p}_{ij_0}(t_h) < q_{j_h}$ or $\sum_i \overline{\theta}_{ij_h} < 0$.

Given such a pivot, we define $\overline{\theta}'$ by setting

$$\overline{\theta}'_{i_g j_{g-1}} = \overline{\theta}_{i_g j_{g-1}} - 1 \quad \text{and} \quad \overline{\theta}'_{i_g j_g} = \overline{\theta}_{i_g j_g} + 1$$

for $g = 1, \ldots, h$ and $\overline{\theta}'_{ij} = \overline{\theta}_{ij}$ for all other $(i, j)$. Since $\overline{\theta}$ satisfies (a) and $i_g \in \omega_{j_{g-1}}$ and $j_g \in \alpha_{i_g}$ for all $g$, $\overline{\theta}'$ satisfies (a). Since $\overline{\theta}$ satisfies (b) and $\sum_j \overline{\theta}'_{i_g o} = \sum_j \overline{\theta}_{i_g o}$ for all $g$, $\overline{\theta}'$ satisfies (b). Since $\overline{\theta}$ satisfies (c) and (d), the definitions of $J(o)$ and $P(i)$ imply that $\overline{\theta}'$ satisfies (c) and (d).

We have $\sum_i \overline{\theta}'_{ij_0} = \sum_i \overline{\theta}_{ij_0} - 1$ and $\max\{0, \sum_i \overline{\theta}'_{ij}\} = \max\{0, \sum_i \overline{\theta}_{ij}\}$ for all $j \neq j_0$, so $\overline{\theta}'$ is closer to satisfying (e) than $\overline{\theta}$. Repeating this maneuver will eventually produce a $\overline{\theta}$ satisfying (a)–(e) unless at some point it becomes impossible to find a pivot.

Our search for a pivot begins by choosing $j_0 \in O$ such that $\sum_i \overline{p}_{ij}(t_0) = q_j$ and $\sum_i \overline{\theta}_{ij} > 0$. We define sets $P_0, J_1, P_1, J_2, \ldots$ inductively, beginning with $P_0 = \{j_0\}$ and continuing inductively with

$$J_g = \bigcup_{j \in P_{g-1}} J(j) \setminus \bigcup_{f<g} J_f \quad \text{and} \quad P_g = \bigcup_{i \in J_g} P(i) \setminus \bigcup_{f<g} P_f.$$

We continue this construction until we arrive at an $h$ such that either $P_h = \emptyset$ or there is a $j_h \in P_h$ such that either $\sum_i \overline{p}_{ij_h}(t_0) < q_{j_h}$ or $\sum_i \theta_{ij_h} < 0$.

If there is such an $j_h$ we construct $i_1, \ldots, i_h$ and $j_1, \ldots, j_h$ by choosing $i_h \in J_h$ such that $j_h \in P(i_h)$, choosing $j_{h-1} \in P_{h-1}$ such that $i_h \in J(j_{h-1})$, choosing $i_{h-1} \in J_{h-1}$ such that $j_{h-1} \in P(i_{h-1})$, and so forth. Clearly $j_0, i_1, j_1, \ldots, i_h, j_h$ is a pivot.

Now suppose that the construction terminates with $P_h = \emptyset$. Let $J = \bigcup_h J_h$ and $P = \bigcup_h P_h$. We have $\sum_i \overline{p}_{ij}(t_0) = q_j$ for all $j \in P$. If $j \in P$ and $i \notin J$, then $i \notin J(o)$, so $\overline{p}_{ij}(t_0) = p_{ij}(t_0)$. If $i \in J$ and $j \notin P$, then $j \notin P(i) = \alpha_i$. Thus $\overline{p}(t_0) - p(t_0)$ is a feasible allocation for $E - p(t_0)$ that gives all of the resources in $P$ to students in $J$, and it gives $1 - p_{ij}(t_0)$ to $i \in J$ whenever $j \in O \setminus P$. Clearly any feasible allocation also has these properties, so $(J, P)$ is a critical pair for $E - p(t_0)$. *Either our procedure finds a satisfactory $\overline{\theta}$, and we can move to $t_1$, $p(t_1)$, and $\overline{p}(t_1)$ as described above, or $t_0 = t*$ and our search finds a critical pair for $E - p(t^*)$.*

## C.1 `pushrelabel.h` and `pushrelabel.c`

As we explained above, `gcps` follows a path in the set of feasible allocations that lies above the path of the allocation, and the first task is to compute a feasible allocation at which this path can begin. This is an example of the problem of finding a maximal flow in a network, which has received a great deal of attention in the computer science literature.

The files `pushrelabel.h` and `pushrelabel.c` implement the push-relabel algorithm of Goldberg and Tarjan (1988). We first describe the general setting of networks and flows, then we describe the algorithm. Finally we describe the specialized setting in which it is applied in our software.

Let $(N, A)$ be a directed graph (that is,$N$ is a finite set of *nodes*, and $A \subset N \times N$ is a set of *arcs*) with distinct distinguished nodes $s$ and $t$, called the *source* and *sink* respectively. We assume that $(n, s), (t, n) \notin A$ for all $n \in N$.

A *preflow* is a function $f \colon N \times N \to \mathbb{R}$ such that:

(a) for all $n$ and $n'$, if $(n, n') \notin A$, then $f(n, n') \leq 0$.

(b) for all $n$ and $n'$, $f(n, n') = -f(n', n)$ (antisymmetry);

(c) $\sum_{n' \in N} f(n', n) \geq 0$ for all $n \in N \setminus \{s, t\}$.

If neither $(n, n')$ nor $(n', n)$ is in $A$, then (a) and (b) imply that $f(n, n') = 0$. Note that $f(s, n), f(n, t) \geq 0$ for all $n \in N$. In conjunction with the other requirements, (c) can be understood as saying that for each $n$ other than $s$ and $t$, the total flow into $n$ is greater than or equal to the total flow out.

A preflow $f$ is a *flow* if $\sum_{n' \in N} f(n,n') = 0$ for all $n \in N \setminus \{s,t\}$. In this case antisymmetry and this condition imply that

$$0 = \sum_{n' \in N} \sum_{n \in N} f(n,n') = \sum_{n \in N} f(n,s) + \sum_{n' \in N} f(n,t),$$

so we may define *value* of $f$ to be

$$|f| = \sum_{n \in N} f(s,n) = \sum_{n \in N} f(n,t).$$

A *capacity* is a function $c \colon N \times N \to [0,\infty]$ such that $c(n,n') = 0$ whenever $(n,n') \notin A$. A *cut* is a set $S \subset N$ such that $s \in S$ and $t \in S^c$ where $S^c = N \setminus S$ is the complement. For a capacity $c$, the *capacity* of $S$ is

$$c(S) = \sum_{(n,n') \in S \times S^c} c(n,n').$$

A preflow $f$ is *bounded* by a capacity $c$ if $f(n,n') \leq c(n,n')$ for all $(n,n')$. It is intuitive, and not hard to prove formally, that if $f$ is a flow bounded by $c$ and $S$ is a cut for $c$, then $|f| \leq c(S)$, so the maximum value of any flow is not greater than the minimum capacity of a cut. The max-flow min-cut theorem (Ford and Fulkerson, 1956) asserts that these two quantities are equal.

The computational problems of finding a flow of maximal value or a minimal cut for a network $(N,A)$ and a capacity $c$ are very well studied, and many algorithms have been developed. The push-relabel algorithm is relatively simple, and certainly fast enough for our purposes. (The literature continues to advance, and algorithms (e.g., Chen et al. (2022)) with better asymptotic worst case bounds have been developed.)

Let $f \colon N \times N \to \mathbb{R}$ be a preflow that is bounded by $c$. The *residual capacity* of $(n,n')$ is

$$r_f(n,n') = c(n,n') - f(n,n').$$

We say that $(n,n')$ is a *residual edge* if $r_f(n,n') > 0$. This can happen either because $c(n,n') > f(n,n') \geq 0$ or because $f(n,n') < 0$. The *excess* of $f$ at $n$ is

$$e_f(n) = \sum_{n' \in N} f(n',n).$$

Of course $e_f(n) \geq 0$, and $f$ is a flow if and only if $e_f(n) = 0$ for all $n \in N \setminus \{s,t\}$. We say that $n \in N \setminus \{s,t\}$ is *active* for $f$ if $e_f(n) > 0$. A *valid labelling* for $f$ and $c$ is a function $d\colon N \to \{0,1,2,\ldots\}$ such that $d(t) = 0$ and $d(n) \leq d(n') + 1$ whenever $(n,n')$ is a residual edge.

The initial preflow of the algorithm is given by setting $f(s,n) = c(s,n)$ for all $n$ such that $(s,n) \in A$, and setting $f(n,n') = 0$ for all other $n$ and $n'$. The initial labelling of the algorithm is given by setting $d(s) = |N|$ and $d(n) = 0$ for all other $n$. The only $(n,n')$ such that $d(n) > d(n')$ are those with $n = s$, and none of these are residual, so $d$ is a valid labelling for $f$.

The algorithm then consists of repeatedly applying the following two *elementary operations*, in any order, until there is no longer any valid application of them:

(a) $\mathrm{Push}(n,n')$ is valid if $n$ is active, $r_f(n,n') > 0$ and $d(n') = d(n) - 1$. The operation resets $f(n,n')$ to $f(n,n') + \delta$ and $f(n',n)$ to $f(n',n) - \delta$ where $\delta = \min\{e_f(n), r_f(n,n')\}$.

(b) $\mathrm{Relabel}(n)$ is valid if $n$ is active and $d(n) \leq d(n')$ for all $n'$ such that $r_f(n,n') > 0$. The operation resets $d(n)$ to $1 + \min_{n':r_f(n,n')>0} d(n')$. (It turns out that there is always at least one $n'$ such that $r_f(n,n') > 0$.)

One intuitive understanding of the algorithm is that we imagine excess as water flowing downhill, so that $d(n)$ can be thought of as a height, (Goldberg and Tarjan offer a somewhat different intuition in which $d$ is a measure of distance.) We think of $\mathrm{Push}(n,n')$ as moving $\delta$ units of excess from a node $n$ to an adjacent node $n'$ that is one step lower. The operation $\mathrm{Relabel}(n)$ is valid when there is excess "trapped" at $n$, and this operation increases $d(n)$ to the largest value allowed by the definition of a valid labelling, which is the smallest value such that there is a neighboring node the excess can flow to.

Based on the description above, it is not obvious that the push-relabel algorithm is, in fact, an algorithm in the sense of always terminating, nor is it obvious that it can only terminate at a maximum flow. Goldberg and Tarjan's proofs of these facts are subtle and interesting, and their paper is recommended to the curious reader.

## C.2 `endpoint.h` and `endpoint.c`

The situation that these files deal with is that we have found a $\overline{\theta}$ (denoted by `theta` in the code) satisfying (a)–(e). We need to compute $t_1$ and the values of the school choice problem, and the feasible guide, at this time. The function `time_until_some_school_exhaustion` gives the amount of time before some school exhausts its quota as each student consumes her favorite. The function `time_until_feasible_guide_not_above_alloc` gives the amount of time until the feasible guide, on its trajectory given by `theta`, ceases to be above the allocation given by consumption of `favorites`. The amount of time until the feasible guide, on its trajectory given by `theta`, would ceases to be a feasible allocation, is computed by the function `time_until_feasible_guide_not_feasible`. The minimum of the functions above is given by the function `time_until_trajectory_change`.

As their names suggest, functions computing the allocation, feasible guide, and scp, at the new endpoint, are given, and the function `move_to_endpoint_jf_segment` bundles these together.

## C.3 `pivot.h` and `pivot.c`

Recall that, in the process as we described it above, a sequence $j_0, i_1, j_1, \ldots, j_{h-1}, i_h, j_h$ used to adjust $\theta$ is called a *pivot*. Finding $j_0, i_1, j_1, \ldots, j_{h-1}, i_h, j_h$ involves set operations that are not optimized at the hardware level (unlike numerical computations) so this can be rather time consuming. On the other hand, checking whether $j_0, i_1, j_1, \ldots, j_{h-1}, i_h, j_h$ is a valid pivot is easy. Furthermore, if we compute $p$ and $\overline{p}$ on $[t_0, t_1]$ and then on $[t_1, t_2]$, it is intuitively plausible (and born out by computational experience) that many of the pivots in the first calculation will also be valid pivots in the second computation.

All of this suggests that we keep a list of the pivots that occurred in the first calculation, and we begin the second calculation by going through this list, for each of its pivots checking whether it is valid, and applying it if it is. The code for constructing and managing such lists (which is fairly simple and self-explanatory) is in `pivot.h` and `pivot.c`.

## C.4 `segment.h` and `segment.c`

The code in these files can be thought of as managing the construction of a single segment of the piecewise linear path of $p$ and $\overline{p}$. We can imagine that the process has been computed up to time $t_0$, so we have a `working_scp`, a `feasible_guide`, and the list `old_list` of pivots that were valid in the preceeding segment.

There is some preliminary information gathering. For student `i`, `alpha[i-1]` is the set of schools that $i$ is eligible for and that have some remaining capacity. The set of schools that have remaining capacity and students who are eligible has the index `index_of_active_schools`, so this is the index of the union of the sets of schools given by `alpha`. For `k` from `1` to `index_of_active_schools.no_elements`, `omega[k-1]` is the index of the set of `i` such that `index_of_active_schools[k-1]` is one of the elements of `alpha[i-1]`.

In the code the matrix that was denoted by $\overline{\theta}$ in the first part of this section is now `theta`. The function `initialize_theta` sets the initial value of the matrix `theta` as we described earlier. For each `j`, `theta_sums[j-1]` is the sum over `i` of `theta[i-1][j-1]`, and the function `initialize_theta_sums` computes the initial value of this array, which is updated dynamically, instead of being recomputed again and again.

For student `i` and school `j` recall the set `J(j)` of students and the set `P(i)` of schools defined in the first part of this section. The function `student_qualified_for_school` returns `1` if `i` is an element of `J(j)` and `0` otherwise. Similarly, if `j` is an element of the set `P(i)`, then the function `school_qualified_for_student` returns `1`, and otherwise it returns `0`. These functions are the basis of the test provided by the function `pivot_is_valid`, and the function `reuse_prior_pivots` goes through the list `old_pivots`, for each element either discarding it if it is not valid or executing it, and adding it to the list `new_pivots`, if it is valid.

The functions `next_J_g` and `next_P_g` use `student_qualified_for_school` and `school_qualified_for_student` to compute next terms in the sequences `P_0,...,P_h` and `J_1,...,J_h` where `P_0 =` $\{$`j`$\}$. The arguments `J_subset` and `P_subset` are the unions of the preceeding `J_f` and `P_f` respectively. It should be noted that `next_J_g` and `next_P_g` update these sets by appending the elements of the computed sets. The function `compute_increments_and_j_h` compute the sequences `P_0,...,P_h` and `J_1,...,J_h`, ending either when some `P_h` is empty or there is a `j_h` in `P_h` such that either `j_h` is not fully allocated in `feasible_guide` or `theta_sums[j_h]` is negative. For given `P_0,...,P_h` and `J_1,...,J_h` the function `extract_pivot` finds an `i_h` in `J_h` that is a suitable predecessor of `j_h`, then finds a suitable predecessor of `i_h`, and so forth. These functions are combined in the function `mas_theta_of_find_crit_pair_for_sch`, which, for a given `j`, computes `P_0,...,P_h` and `J_1,...,J_h`, and, if a critical pair has not been found, extracts a pivot, executes it, and adds it to `new_list`.

The function `massage_theta_or_find_critical_pair` proceeds through all schools `j` with positive `theta_sums[j-1]`, applying `mas_theta_of_find_crit_pair_for_sch` until either `theta_sums[j-1]` is zero or a critical pair has been found. The end result is either that theta is suitable for the construction of another segment of the path of $p$ and $\overline{p}$, or a critical pair was found. The function compute_next_path_segment_or_find_critical_pair puts everything together. It first initializes alpha, active_school_list, omega, and other objects. It then calls massage_theta_or_find_critical_pair, and if a critical pair was not found it moves the entire process to the endpoint of the next line segment.

## C.5 gcps.c, gcpscode.h, and gcpscode.c

The main function of gcps is contained in gcps.c. As with the other executables, the main function in gcps.c reads an input_sch_ch_prob from a file. It derives a process_scp from it, uses the function simple_GCPS_alloc to obtain a partial_alloc, prints this, and then cleans up memory.

It is interesting to keep track of the numbers of various types of events that occur during the computation. A *split* occurs when we reach $t^*$ and the computation descends recursively to the two subproblems. The pointer to an integer no_splits records the number of times this happens. The computation of $p$ and $\overline{p}$ on an interval such as $[t_0, t_1]$ is a *segment*, and the pointer to an integer no_segments records the number of such computations. The pointers to integers no_old_pivots and no_new_pivots record the number of times that a pivot brought forward from the previous segment was applied in the computation of the current segment, and the number of times that new pivots were generated. One interesting finding is that the ratio of no_old_pivots to no_new_pivots is on the order of 10 to 1, so keeping the lists of pivots may speed things up to some extent. Whenever a pivot $j_0, i_1, j_1, \ldots, j_{h-1}, i_h, j_h$ is implemented, the pointee of the pointer to an integer h_sum is incremented by $h$. In cmputational experience the ratio of h_sum to the total number of pivots is about 1.5, so most pivots finish after one step, and only a few pivots are long.

These variables add a certain amount of clutter to the code, but they are simple and easy to understand, and can easily be ignored when studying other aspects. The GCPS allocation, along with these other variables, is computed by the function simple_GCPS_alloc, after which the extra variables are thrown away, as the code is currently configured. It computes the GCPS allocation by calling GCPS_alloc, which initializes the feasible guide using push_relabel, and probe_list(), which is the list of pivots, and then calls GCPS_allocation_with_guide, which is the recursive version of the algorithm. (In the recursive descent the feasible_guide of each subproblem is derived from the feasible_guide at the time.)

The function `compute_until_next_critical_pair` computes an allocation and, possibly, a critical pair, by repeatedly calling `massage_theta_or_find_critical_pair` until either the allocation process ends or a critical pair is found. This function is called at the beginning of `GCPS_allocation_with_guide`. This may end the computation, but if it does not then the computation splits into a "left" and a "right" subproblem. The left subproblem is the one associated with the critical pair of `J` and `P`, and it is typically quite small, with `P` containing only a small number of schools.

Since the computation is recursive, when there are many schools there can be a large stack of calls to `GCPS_allocation_with_guide` coming from `descend_to_right_subproblem`. Each call has certain data associated with it, and if this data is not minimized, the cumulative burden can overwhelm available memory. For this reason we compute the left subproblem first and then destroy the `working_scp`, the `feasible_guide`, and the `probe_list` before calling `GCPS_allocation_with_guide` within `descend_to_right_subproblem`. The main data associated with each call to `GCPS_allocation_with_guide` is the version of `final_alloc` brought in as an argument of `descend_to_right_subproblem`. The version of `final_alloc` coming in is the result of the call to `descend_to_left_subproblem`, which is small, plus the result of `compute_until_next_critical_pair`, which is the allocation of each agent's favorite over a certain amount of time, so the incoming `final_alloc` can be recovered from smaller objects, but this would require substantial reorganization of the code, so it has not yet been done.

# References


Abdulkadiroğlu, A., Pathak, P. A. and Roth, A. E. (2009), 'Strategy-proofness versus efficiency with indifferences: Redesigning the NYC high school match', *Am. Econ. Rev.* **99**, 1954–1978.

Abdulkadiroğlu, A. and Sönmez, T. (2003), 'School choice: a mechanism design approach', *Am. Econ. Rev.* **93**, 729–747.

Azevedo, E. M. and Budish, E. (2019), 'Strategy-proofness in the large', *Rev. Econ. Stud.* **86**, 81–116.

Balbuzanov, I. (2022), 'Constrained random matching', *J. Econ. Theory* **203**. Article 105472.

Bogomolnaia, A. and Moulin, H. (2001), 'A new solution to the random assignment problem', *J. Econ. Theory* **100**(2), 295–328.

Budish, E., Che, Y.-K., Kojima, F. and Milgrom, P. (2013), 'Designing random allocation mechanisms: Theory and practice', *Amer. Econ. Rev.* **103**, 585–623.

Chen, L., Kyng, R., Liu, Y., Gutenberg, M. and Sachdeva, S. (2022), 'Maximum flow and minimum-cost flow in almost-linear time', *arXiv preprint* .

Ford, L. and Fulkerson, D. (1956), 'Maximal flow through a network', *Canadian J. Math.* **8**, 399–404.

Gale, D. and Shapley, L. (1962), 'College admissions and the stability of marriage', *Am. Math. Mon.* **65**, 9–15.

Goldberg, A. V. and Tarjan, R. E. (1988), 'A new approach to the maximum flow problem', *J. ACM* **35**, 921–940.