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

Andrew McLennan*

September 7, 2025

Abstract

This document describes software implementing the GCPS 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 a motivated 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. For example, it may be a single sex school, and a selective school may have a minimum test score or GPA. In addition, a student may be able to veto assignment to some schools. In the schemes we focus on each student has a *safe school* that is guaranteed to admit her if she wishes, and she is not eligible to be assigned to any school that she likes less than her safe school.

Each school may have a ranking of eligible students called a *priority* that might, for example, express a preference for high test scores, minority status, or students who live nearby. We say that the priority is *dichotomous* if there is no such ranking, so each student is either eligible or inelligible to attend the school, and the school gives equal consideration to all of its eligible students. Otherwise the priority is *nondichotomous*.

A *school choice mechanism* is a scheme or mapping whose inputs are the preferences of the students and the priorities of the schools, and whose output is an assignment of each student to one of the schools. This assignment must be *feasible*, in the sense that the number of students assigned to each school is not greater than the school's capacity. Generally speaking, we want students to attend schools they like, and we want the schools' priorities to be respected, to the extent that they express society's values concerning who can take greatest advantage of the available educational resources.

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 mechanism* (GCPS) is a mechanism for dichotomous priorities. (The cumbersome name is derived from history. In a seminal paper Bogomolnaia and Moulin (2001) proposed the *probabilistic serial mechanism* for random assignment of a certain number of objects (e.g., jobs) to the same number of agents. The *generalized probabilistic serial mechanism* was proposed by Budish et al. (2013), and a further generalization, *generalized constrained probabilistic serial mechanism*, was proposed by Balbuzanov (2022).) The two other mechanisms, *GCPS-a* and *GCPS-b*, are designed to be applied to school choice settings with nondichotomous priorities.

GCPS Schools is a software package that implements algorithms that compute these mechanisms, and related algorithms that are required to apply these algorithms, and to test the software. The primary purpose of this document is provide an informal, but rather detailed, guide to *GCPS Schools*. For those who might wish to use our mechanisms, possibly without "looking under the hood," Section 2 gives detailed step-by-step instructions concerning downloading, installation,

preparation of inputs, and the mechanics of how the software is invoked. Sections 3–6 describe the code for programmers who might be interested in modifying or extending it in some way. We give a general overview of the structure of the code, and describe different modules briefly, but hopefully in enough detail that the reader can easily figure out where to find what they are interested in. 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 that mechanism that our mechanisms solve.

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.

A suprising finding of Bogolmonaia and Moulin is that random priority is inefficient in a rather strong sense. For a given preference ordering of the objects, one probability distribution over the objects *stochastically dominates* a second distribution if, for each object, say o, the total probability assigned to objects that are at least as good as o under the first distribution is at least as large as the total probability assigned to such objects under the second distribution. (Equivalently, the first distribution is obtained from the second by reassigning a certain amount of probability from worse objects to better objects.) If, in addition, the two distributions are different, then the first distribution *strictly stochastically dominates* the second distribution.

We say that a bistochastic matrix[1] of probabilities of assignments of objects to agents is *sd-efficient* if there does not exist a second such matrix that gives each agent a distribution over O that stochastically dominates (for that agent's preference ordering) the distribution given by

[1] A square matrix is *bistochastic* if its entries are nonnegative, the sum of the entries in each row is one, and the sum of the entries in each column is one.

the first matrix, with strict domination for some agents. Bogolmolnaia and Moulin give examples showing that random priority can produce a matrix of assignment probabilities that is not sd-efficient. They propose a *probabilistic serial* (PS) mechanism that produces matrices of assignment probabilities that are always sd-efficient.

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 consuming (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 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. As we will describe in detail later, for any such "bistochastic" matrix of assignment probabilities it is possible to compute a probability distribution over deterministic assignments that realizes all of the assignment probabilities.

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* mechanism by Balbuzanov (2022). Balbuzanov's framework is very general. One of our main contributions is to show that, when specialized to school choice, the GCPS mechanism is computationally tractable.

In the context of school choice, the elements of I are *students* and the elements of O are *schools*. For the time being we consider only dichotomous priorities, so each student has a preference ordering of the schools she is eligible to attend. Each school has a *quota*, which is the number of students it can accomodate. Thus the input data of the GCPS mechanism consists of a *school choice problem* (SCP) which specifies the sets of schools and students, the quotas of the various schools and, for each student, the set of schools she is eligible for, ranked from best to worst.

A *feasible allocation* for an SCP 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 quota.

(Later we will see that for any feasible allocation there is a probability distribution over deterministic feasible assignments that realizes the assignment probabilities.)

We only consider SCP's for which feasible allocations exist. Suppose that each student has a safe school, so that the schools she is eligible for in the SCP are those that satisfy any other criteria of eligibility, and that she weakly prefers to her safe school. A feasible allocation certainly exists if the assignment of each student to her safe school is feasible. Many school systems (e.g., the state of Victoria in Australia) have *neighborhood priority* in which each student's safe school is the school whose district contains her residence.

The GCPS mechanism may also be thought of as 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. As in the PS mechanism, a school may become unavailable if its capacity is exhausted, but now there is an additional possibility. 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. 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 i can attend 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 our algorithm is recursive, calling itself again and again as it descends through a tree of subproblems of the same type.

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.

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, misreporting can be beneficial.

Since the GCPS mechanism has PS as a special case, it is not strategy-proof. In "Efficient Computationally Tractable School Choice Mechanisms" we argue in some detail that the failures of strategy proofness are mild and unimportant practically. Here we briefly summarize that discussion.

We consider three different types of manipulation attempt by a student. An *augmentation* reports that some school that is worse than the student's safe school is actually better. A *truncation* reports that some school that is better than the safe school is actually worse. For a given set of schools that are at least as good as the safe school, a *reordering* reports a preference ordering of these schools that is different from the true ordering.

An augmentation, by itself, always gives the student a distribution of schools that is sd-dominated by the distribution given by reporting the true preference. Technically speaking, this does not quite imply that there are no profitable manipulations that combine augmentation with other types of manipulation, and are not improved by undoing the augmentations, but this possibility seems quite far-fetched, so we can safely ignore manipulation attempts involving augmentations.

Truncation manipulation attempts can succeed. To see how this might happen, suppose that the students favorite school is a, her safe school is b, and during the allocation process the set

$P = \{a, b\}$ becomes critical. If she reports that a is the only school she prefers to her safe school, she will be in J_P, and if enough other elements of J_P prefer b to a, she can end up with a with probability one. On the other hand, if she reports that there are other schools, besides a, that she prefers to b, then she will not be in J_P, and may end up receiving positive probabilities of some of these schools. Such manipulations may be annoying, but only a few students can have such opportunities, and (depending on the information the student has) such attempts can be risky. Perhaps more important, such manipulation attempts do little to change the character of the mechanism for other students, in contrast with what we will see later in connection with the Boston mechanism.

To study reordering manipulation attempts we look at a *restricted GCPS mechanism* in which each student's set of schools that are weakly preferred to the safe school are known, and that student's allowed reports are the orderings of that set that have the safe school at the bottom. We consider a concept that is weaker than strategy-proofness, but 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 schools. 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. 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 restricted 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. The probability distribution over these times is determined by the empirical distribution over reported types in the population, not who has which type. The reason that the restricted GCPS mechanism is strategy proof in the large is that when the population is large, the agent's beliefs about the probability distribution over such empirical distributions is only slightly affected by the agent's own report.

On the whole opportunities to manipulate are rare, and do significantly undermine the advantages of the GCPS mechanism.

1.2 The MCC Computation

We now consider priorities for the schools that are not dichotomous. That is, within the set of students a school finds acceptable, it prefers some over others. A common example is that students who live in the school district and have a sibling at the school have highest priority, students with a sibling at the school who live outside the district have second priority, students in the district, but without a sibling, have third priority, and all others have lowest priority. Now (in addition to the sets of students and schools, the quotas of the schools, and the eligibilities and preferences of the students) a *school choice problem* (SCP) specifies the priority of each student at each school. We assume that these priorities are nonnegative integers. Note that a student may have priority zero at a school and still be eligible to attend that school.

Any mechanism for such SCP's embeds choices concerning how much, and how, to bend the allocation in the direction of the priorities. To a certain extent there is inherent conflict between the students' preferences and the social values expressed by the priorities, and also between the students who are favored by the priorities and those who are disfavored. Although some mechanisms might be "better" than others for purely theoretical reasons, to a large extent the choice between theoretically acceptable mechanisms can be significantly affected by the types of problems to which they will be applied. In addition, once a mechanism is given, the priorities are policy choices that can be adjusted to more accurately attain the values they are intended to express. We can argue that what we propose below seems reasonable and has some desirable theoretical properties, but we cannot claim more than that.

A fundamental concept for our approach to nondichotomous priorities is a *coarse priority* for a school o, which is a nonnegative integer C_o. The idea is that a students with a priority at the school that is below C_o has no chance of being admitted to the school, and a student whose priority at the school is above C_o has unlimited access to o, and consequently has no chance of being assigned to any school they like less than o.

A *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 cutoff c_o refines a unique coarse cutoff, namely the floor[2] of c_o. The idea is that if a student's priority at the school is C_o, then the student's maximum allowed consumption of the school is $1 - (c_o - C_o)$, which decreases from 1 to 0 as c_o increases from C_o to $C_o + 1$.

A *coarse cutoff profile* $C = (C_o)_{o \in O}$ is a specification of a coarse cutoff for each school, and a *cutoff profile* $c = (c_o)_{o \in O}$ is a specification of a cutoff for each school. Of course for such a

[2]The *floor* of a real number t is the largest integer that is not greater than t.

c there is a unique coarse cutoff profile $C(c)$ such that for each o, c_o refines $C_o(c)$. We define the demands $D_{io}(c)$ of student i for the various schools o by specifying that i consumes as much of her favorite school as c allows, then devotes as much of her remaining probability as c allows to her second favorite school, and so forth until she either has one unit of probability or she has as much as she is allowed to consume of every school. Thus she is consuming the allowed amount of each school with positive consumption except possibly the least preferred one, where her consumption may be the amount needed to complete a probability distribution.

Let the total demand for school o be $D_o(c) = \sum_i D_{io}(c)$. We say that c is a *market clearing cutoff* (MCC) if $D_o(c) \leq q_o$ for all o, and $c_o = 0$ whenever $D_o(c) < q_o$. Some rather advanced mathematics implies that there is a minimum market clearing cutoff MCC-a that is (componentwise) less than or equal to any other market clearing cutoff, and there is a maximum market clearing cutoff MCC-b that is greater than or equal to any other market clearing cutoff. In particular, market clearing cutoffs exist.

Especially if you are trained in economics, you might expect that the allocation given by the demands at a market clearing cutoff is a good one, but in fact this is not the case, precisely because the concept prevents certain mutually beneficial trades. Suppose that you and your friend both have a positive probability of being admitted to Princeton, and you also both have a positive probability of being admitted to Yale. (Possibly you both have the maximum probability allowed to marginal applicants at both schools.) If you prefer Yale to Princeton, while you friend prefers Princeton, you can both be better off (without hurting anyone else) if you trade some of your probability of Princeton in exchange for an equal amount of your friend's probability of Yale.

What to do? In some sense the market clearing cutoff concept does a reasonable job of blending the priorities and the students' preferences, but the resulting allocation is not satisfactory. Kesten and Ünver (2015) propose passing from the allocation of MCC-a to an allocation that is constrained efficient (in a certain sense) by repeatedly implementing certain kinds of Pareto improving trades.

Our approach is different. We regard the coarse cutoffs of a market clearing cutoff as a reasonable measure of the extent to which the priorities can be imposed on the allocation, but we discard the cutoffs themselves. Given a profile of coarse cutoffs, there is a *restricted school choice problem* obtained from the given SCP by eliminating each student's eligibility at each school where her priority is below that school's coarse cutoff, and also at each school that she likes less than some other school where her priority is above that school's coarse cutoff. We

define GCPS-a to be the allocation obtained by applying GCPS to the restricted school choice problem obtained in this way from the coarse cutoff profile of MCC-a, and we define GCPS-b similarly in relation to MCC-b.

In order for this to be workable, it must be the case that the restricted school choice problems have feasible allocations. This will certainly be the case if the allocations associated with MCC-a and MCC-b are feasible allocations for the given SCP (i.e., if each student is able to consume a full unit of probability) because these allocations also satisfy the additional restrictions of the respective restricted school choice problems. In turn the allocations associated with MCC-a and MCC-b will be feasible allocations for the given SCP if each student has a safe school, which in this context is a school such that the number of students with equal or greater priority is not greater than the school's quota.

Assuming that it is well defined, GCPS-a is an sd-efficient allocation within the set of feasible allocations for the restricted school choice problem derived from the coarse cutoffs of MCC-a. That is, there is no sd-dominating allocation *if we regard the additional restrictions coming from the coarse cutoffs of MCC-a as hard constraints*. Symmetrically, GCPS-b is, in the relevant sense, constrained sd-efficient.

The possibilities for manipulation of the GCPS-a and GCPS-b mechanisms are similar to those described earlier for GCPS. Although it has not been proven formally that a single augmentation can never lead to improvement, this seems likely, and in any event augmentations are practically irrelevant. Successful truncation manipulations are possible, as we described earlier, but uncommon. We define restricted GCPS-a and GCPS-b mechanisms by specifying that, for each student, the set of schools weakly preferred to the safe school are known, so that the only allowed reports are orderings of this set that have the safe school at the bottom. Holding the coarse cutoffs and the times at which sets of schools become unavailable fixed, truthful revelation is weakly dominant, so any benefit from manipulation results from changing these variables. These variables are determined by the empirical distribution over types in the population. Under the informational assumptions described earlier, when the population is large the agent's beliefs about the probability distribution over such empirical distributions is only slightly affected by the agent's own report, so the restricted GCPS-a and GCPS-b mechanisms are strategy-proof in the large.

We now describe a method for computing MCC-a. (The framework is entirely symmetric, so, with suitable modifications, this method can also be used to compute MCC-b.) For a profile

of cutoffs c, $J(c)$ is the profile of cutoffs whose component $J_o(c)$ for school o is either 0, if $D_o(c) < q_o$, or is the cutoff such that $D_o(J_o(c), c_{-o}) = q_o$. Note that c is an MCC if and only if it is a fixed point of J, i.e., $J(c) = c$.

We define a sequence $c^0, c^1, c^2, \ldots$ as follows. Let $c^0 = 0$ be the profile of cutoffs whose components are all 0, and for $t \geq 1$ let $c^t = J(c^{t-1})$. Intuitively, each school is trying to adjust its cutoff to make its total demand equal to its quota, but since all schools are doing this, some of their demand may be displaced to other schools, in which case we have to do it again. Thus $c_o^{t+1} \geq c_o^t$. Since the sequence $\{c^t\}$ is increasing, it converges to some limit, say c^∞. A few mathematical details are involved in showing that $c^\infty =$ MCC-a, but this is indeed the case. For reasons that will become clearer in Subsection 1.4, Kesten and Ünver (2015) call this computational procedure *fractional deferred acceptance*.

From a computational point of view, a blemish is that c^t need not converge to c^∞ in finitely many steps. (Nevertheless, continuing to iterate until each of the numbers $D_o(c^t) - D_o(c^{t+1})$ is below a suitable threshhold is in fact a perfectly viable computational method.) Kesten and Ünver present an algorithm that does achieve finite convergence, and in Subsection 5.2 we describe the implementation of another such algorithm.

1.3 Generating a Random Deterministic Assignment

Having computed a feasible allocation, 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). We now briefly explain the key idea of the special case, for our context, of the algorithm they propose for this.

We begin with a feasible allocation m with entries m_{io}. Suppose that $(1-t)m^\alpha + tm^{-\beta} = m$ where $0 < t < 1$, m^α and $m^{-\beta}$ are feasible allocations such that all the entries that are integral in m are integral in m^α 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^α and $m^{-\beta}$. In addition, suppose that either m^α has an integral entry that is not integral in m, or there is a school that has integral total demand in m^α but not in m, and similarly for $m^{-\beta}$. The algorithm works by transitioning from m to m^α with probability $1-t$ and to $m^{-\beta}$ with probability t. 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^α 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. Two nodes $v, w \in V$ are *neighbors* if $\{v, w\} \in E$, and the *degree* of v is the number of neighbors it has. A *cycle* is a sequence of distinct nodes $v_1, \ldots, v_k$ such that, for all $i = 1, \ldots, k$ (where the indices are integers mod k), v_i and v_{i+1} are neighbors. If $E \neq \emptyset$ and there are no nodes of degree one, then a cycle exists: starting with any $w_1 \in V$ of nonzero degree, we construct $w_2, w_3, \ldots$ inductively, for each j choosing a neighbor w_{j+1} that is different from w_{j-1}, and since V is finite, eventually the sequence repeats some node.

In the undirected graph $G = (V, E)$ of interest, the elements of the set of nodes are the students and 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 and only if m_{io} is not an integer (that is, neither 0 nor 1). There is an edge between a school and the sink if and only 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.

For each student i, $\sum_o m_{io} = 1$, so if there is any o such that m_{io} is not an integer, then there is an least two such nodes, and thus the degree of i is not one. For a school o, if $\sum_i m_{io}$ is not an integer, then the sink is a neighbor and there is at least one i such that m_{io} is not an integer, so the degree of o is not one. On the other hand, if $\sum_i m_{io}$ is an integer, then there cannot be exactly one i such that m_{io} is not an integer, and again the degree of o is not one. Since $|I| = \sum_o (\sum_i m_{io})$ is an integer, there cannot be exactly one o such that $\sum_i m_{io}$ is not an integer, and consequently the sink cannot have degree one.

Let $w_1, \ldots, w_k$ be a cycle. For each $i = 1, \ldots, k$, the edge from w_i to w_{i+1} is a *forward edge* if w_i is a student and w_{i+1} is a school, or if w_i is a school and w_{i+1} is the sink, and otherwise the edge between w_i and w_{i+1} is a *backward edge*.

For a real number δ consider the matrix m^δ of numbers obtained by increasing $m_{w_i w_{i+1}}$ by δ when the edge between w_i and w_{i+1} is a forward edge and decreasing $m_{w_{i+1} w_i}$ by δ 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^δ 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^δ is the same as in m. Therefore m^δ is a feasible allocation if δ is sufficiently small.

Let α be the smallest positive number such that m^α 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^{α}. Let $\beta > 0$ be the smallest positive number such that $m^{-\beta}$ 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 $m^{-\beta}$. Then m^{α} and $m^{-\beta}$ satisfy all the conditions specified at the beginning of this subsection.

1.4 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-proposing 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.

As it is usually described, DA takes place over multiple rounds. In the first round 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.)

In this setting 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 some girl in any stable matching.

Let μ 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 μ' that is better for him. Let R be the set of boys who prefer their partner in μ' to their partner in μ. 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 μ', of boys in R. Of course $\mu'(S) = R$.

We first prove that $\mu(S) = R$. Let Beth be an element of S. Then there is an element of R, say Abe, such that $\mathrm{Beth} = \mu'(\mathrm{Abe})$. Let $\mathrm{Carl} = \mu(\mathrm{Beth})$. Since Abe has different partners in μ and μ', Beth has different partners in μ and μ', so Abe and Carl are different. Since Abe prefers Beth to his partner in μ and μ is stable, Beth must prefer Carl to Abe. Among other things, this implies that Carl is a boy and not Beth herself. Let $\mathrm{Doris} = \mu'(\mathrm{Carl})$. Since Beth's partners in μ and μ' are different, Carl has different partners in μ and μ', so Beth and Doris are different. Since Beth prefers Carl to Abe, the stability of μ' (with respect to the preference profile modified by Albert's manipulation) implies that Carl prefers Doris to Beth, so Carl is indeed an element of R. This completes the proof that $\mu(S) = R$.

Under DA for the true preferences, leading to μ, there is a last round in which an element of R makes a proposal. Let Don be one of the elements of R who proposes in this round, and let Ella be the girl he proposes to. Since every boy in R prefers his partner in μ' to his partner in μ, every girl in S has already been proposed to by her partner in μ' prior to this round, has rejected him, and is holding a proposal from some other boy. In particular, 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 μ'. Since Ella rejected her partner in μ' on her way to holding a proposal from Fred, she prefers Fred to her

partner in μ'. Since Fred was rejected by Ella, Fred prefers Ella to his partner in μ, and since Fred is not in R, Fred weakly prefers his partner in μ to his partner in μ', so Fred prefers Ella to his partner in μ'. Thus Ella and Fred are a blocking pair for μ'. This contradiction of the stability of μ' 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 *realized 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 realized 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.

It can happen that society's values give rise to priorities that not dichotomous, but are coarse insofar as a school may be indifferent between two students. The example described earlier in which the student's priority depends on whether she has a sibling at the school and whether the student lives in the school's district has this property. 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 the GCPS-a or GCPS-b 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 for DA 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.

As we have stressed, safe schools also make sense for the GCPS, GCPS-a, and GCPS-b 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 the GCPS, GCPS-a, and GCPS-b mechanisms is restored, because it is as if she submits a ranking of all schools. Although not very well studied in the academic literature, safe schools seem to fairly common in practice. One would expect them 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 Sections 3–6, 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_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_schools` that this action creates and display its contents:

```
$ tar xvf gcps_schools.tar
$ cd gcps_schools
$ ls
GCPS_User_Guide.pdf defaccep.c defaccep.h efficient.c
efficient.h endpoint.c endpoint.h fdamcc.c fdamcc.h gcpsa.c
gcpsacode.c gcpsacode.h gcpsb.c gcpsbcode.c gcpsbcode.h gcps.c
gcpscode.c gcpscode.h makefile makex.c makexcode.c makexcode.h
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 schchprob.c schchprob.h script.sh segment.c
segment.h sprsmtrx.c sprsmtrx.h subset.c subset.h vecmatrx.c
vecmatrx.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 -c -o normal.o normal.c
```

```
gcc -I. -Wall -Wextra -c -o subset.o subset.c
gcc -I. -Wall -Wextra -c -o sprsmtrx.o sprsmtrx.c
gcc -I. -Wall -Wextra -c -o schchprob.o schchprob.c
gcc -I. -Wall -Wextra -c -o makexcode.o makexcode.c
gcc -o makex makex.c normal.o subset.o sprsmtrx.o schchprob.o
     makexcode.o -lm
gcc -I. -Wall -Wextra -c -o parser.o parser.c
gcc -I. -Wall -Wextra -c -o partalloc.o partalloc.c
gcc -I. -Wall -Wextra -c -o pivot.o pivot.c
gcc -I. -Wall -Wextra -c -o endpoint.o endpoint.c
gcc -I. -Wall -Wextra -c -o segment.o segment.c
gcc -I. -Wall -Wextra -c -o efficient.o efficient.c
gcc -I. -Wall -Wextra -c -o purifycode.o purifycode.c
gcc -I. -Wall -Wextra -c -o defaccep.o defaccep.c
gcc -I. -Wall -Wextra -c -o gcpscode.o gcpscode.c
gcc -o gcps gcps.c normal.o parser.o subset.o schchprob.o
     partalloc.o pivot.o endpoint.o segment.o efficient.o
     purifycode.o sprsmtrx.o defaccep.o gcpscode.o -lm
gcc -I. -Wall -Wextra -c -o gcpsacode.o gcpsacode.c
gcc -I. -Wall -Wextra -c -o mcccode.o mcccode.c
gcc -I. -Wall -Wextra -c -o fdamcc.o fdamcc.c
gcc -I. -Wall -Wextra -c -o vecmatrx.o vecmatrx.c
gcc -o gcpsa gcpsa.c gcpsacode.o mcccode.o fdamcc.o vecmatrx.o
     gcpscode.o defaccep.o segment.o endpoint.o pivot.o
     efficient.o partalloc.o subset.o normal.o parser.o
     schchprob.o sprsmtrx.o -lm
gcc -I. -Wall -Wextra -c -o gcpsbcode.o gcpsbcode.c
gcc -o gcpsb gcpsb.c gcpsbcode.o mcccode.o fdamcc.o vecmatrx.o
     gcpscode.o defaccep.o segment.o endpoint.o pivot.o
     efficient.o partalloc.o subset.o normal.o parser.o
     schchprob.o sprsmtrx.o -lm
gcc -o purify purify.c normal.o parser.o subset.o partalloc.o
```

```
purifycode.o sprsmtrx.o schchprob.o -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 `makex`, `gcps`, `gcpsa`, `gcpsb`, and `purify`.

Readers who have some experience with `C` programming will understand the compilation process 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. Let's look at the lines in the `makefile` that govern the construction of the executable `makex`.

```
makex:  makex.c normal.o subset.o sprsmtrx.o schchprob.o
        makexcode.o
    $(CC) -o makex makex.c normal.o subset.o sprsmtrx.o
        schchprob.o makexcode.o $(LDFLAGS)
```

The first line above specifies the resources that are required to build `makex`, which are `makex.c` and a number of *objects*, which are files that end in `.o`. If you type `make makex` on the terminal's command line, for each object, if the object does not exist or the corresponding `.c` file has been edited since the last time it was built, the object is built, as we see above in the first lines of the output of make. After all the objects have been built, `makex` itself is constructed by executing the second line above.

The command for building `normal.o` could have been written as

```
$(CC) -o normal.o normal.c $(LDFLAGS)
```

Here `$(CC)` is the "value" (in the Unix operating system the character `$` often has this meaning) of `CC`, which is set in the first line of the `makefile` to be `gcc`. Whenever the `C` compiler is invoked, it looks for the `CFLAGS` and `LDFLAGS`, which have been set to `-I. -Wall -Wextra` and `-lm` respectively. (Until one is quite experienced, it is unnecessary to worry about what the `CFLAGS` and `LDFLAGS` mean.) The symbol `-o` tells the compiler to use the next word in the command line (here "`normal.o`" as the name of the output. Similarly, in the line for constructing `makex`, `makex` is specified as the name of the output.

A target of `make` can be a list of other targets, and in this `makefile` 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.

2.2 School Choice Problems

We now 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. 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 Schools* is that one of the executables `gcps`, `gcpsa`, or `gcpsb` 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 SCP file with the relevant information. We now explain the format of such files. We recommend file names for SCP files that end with `.scp`, but the software does not enforce this.

GCPS 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`, `gcpsa`, and `gcpsb`.

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 newlines), 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, gcpsa, and gcpsb

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`, `./gcpsa`, `./gcpsb`, 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, so the matrix of assignment probabilities 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`, `gcpsmin`, and `gcpsmax` 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.

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`, `gcpsa`, and `gcpsb` 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`, `gcpsa`, and `gcpsb` 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.

```
$ ./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`, `gcpsmin`, and `gcpsmax`, 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 for someone who simply wants to use the software as is, there is 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`, `gcpsmin`, `gcpsmax`, 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.

3 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`, `gcpsmin`, `gcpsmax`, and `purify`, and for other objects the printing functions are primarily 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 $0, \ldots, n-1$. We always think of it as indexed by the integers $1, \ldots, n$, so the j^{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. There are some user defined functions that take variable such as `i` and `j` as arguments, and these are called with the actual values of these variables, so, for example, `get_entry(myalloc, i, j)` is the probability of school `j` assigned to student `i` by the (pointer to) allocation `myalloc`.

4 Basic Code

In this section we give brief descriptions 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.

4.1 `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`, `gcpsa`, and `gcpsb` 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.

4.2 `subset.h` and `subset.c`

One may represent a subset of $\{1, \ldots,$ `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.

4.3 `vecmatrx.h` and `vecmatrx.c`

This module provides the linear algebra that we need. Many of the functions have self-explanatory titles, and need no further description. Several of the functions provide the usual row operations that are used in the computation of `inverse`.

We will need some rather specialized concepts in the computation of the MCC allocations. Row i of an $k \times k$ matrix $M = (m_{ij})$ is *weakly diagonally dominant* (WDD) if $|m_{ii}| > 0$ and $|m_{ii}| \geq \sum_{j \neq i} |m_{ij}|$, and it is *strictly diagonally dominant* (SDD) if the latter inequality holds strictly. We say that M is *weakly diagonally dominant* (WDD) if each of its rows is WDD. The function `is_WDD` returns 1 or 0 according to whether this is the case.

The *directed graph* of M has vertex set $\{1, \ldots, k\}$ and an edge from i to j if and only if $m_{ij} \neq 0$. The function `directed_graph` returns an $n \times n$ array of integers whose (i, j) entry is 1 if $m_{ij} \neq 0$ and is 0 if $m_{ij} = 0$. The function `T_subset` returns the set T of $i \in \{1, \ldots, k\}$ such that there is no walk $i = i_1 \to i_2 \to \cdots \to i_p$ in the directed graph of M such that row i_p is SDD. We say that M is *weakly chained diagonal dominant* (WCDD) if it is WDD and $T = \emptyset$. If M is WCDD, then it is invertible (Shivakumar and Chew, 1974), and if, in addition, the diagonal entries are positive and the off-diagonal entries are nonpositive, then the entries of M^{-1} are nonnegative. (For the proof of this see "Efficient Computationally Tractable School Choice Mechanisms".)

A set $T' \subset T$ is *minimal* if it is nonempty, there is no walk $i_1 \to i_2 \to \cdots \to i_p$ in the directed graph of M such that $i_1 \in T'$ and $i_p \notin T'$, and there is no nonempty proper subset of T' with this property. The function `minimal_T_subset` computes such a T'. The function `sub_mat` computes the submatrix $M_{T'}$ of M whose rows and columns are indexed by the elements of T'.

We now specialize to matrices M whose diagonal elements m_{ii} are positive and whose off-diagonal elements m_{ij} for $i \neq j$ are nonpositive. Let $H_k = \{ x \in \mathbb{R}^k : \sum_i x_i = 0 \}$. For each $i \in T'$, row i of M is WDD but not SDD, so it is an element of H_k. Let $n = |T'|$, and let $H_n = \{ x \in H_k : x_i = 0 \text{ for all } i \notin T' \}$. For $i \in T'$, $m_{ij} = 0$ whenever $j \notin T'$, so the corresponding row of $M_{T'}$ is an element of H_n. In this circumstance the minimality of T' implies that the origin is in the interior (in H_n) of the convex hull of the rows of $M_{T'}$. (For the proof of this see "Efficient Computationally Tractable School Choice Mechanisms".)

The input of `convex_weights` is an $n \times n$ matrix whose rows $x_1, \ldots, x_n$ are elements of

H such that the origin is in the interior of their convex hull. The output is the n-vector c such that $\sum_i c_i x_i = 0$ and $\sum_i c_i = 1$. To find c we solve the system of equations $\sum_i c_i x_{ij} = 0$ for $j = 1, \ldots, n-1$ (which implies that $\sum_i c_i x_{in} = 0$ because $\sum_j x_{ij} = 0$ for all i) and $\sum_i c_i = 1$.

4.4 `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 `structs` `int_sparse_matrix` and `dbl_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 basic 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.

4.5 `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.

4.6 `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 seat 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.

4.7 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.

4.8 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.

4.9 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 (which is forbidden in C) 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 Subsection 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.

4.10 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 as it is quite unlikely that mistaken software could somehow produce an allocation that was incorrect but nevertheless sd-efficient.

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 Δ 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 which `i` is receiving a postive amount of. 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 the 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 either `all_so_far` or `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. Otherwise all elements of `accepting_students` will eventually be removed, revealing that the allocation is sd-efficient.

5 Computing the MCC Allocations

In this section we describe the computation of the MCC-a and MCC-b allocations, and the associated profiles of coarse cutoffs. In the overall scheme of the project, the coarse cutoffs are inputs to the computation of the GCPS-a and GCPS-b allocations, so there is no executable that outputs the results of these computations. (It would be extremely easy to create such an executable, if one wanted to do so.)

There are two methods of computation. The first, which was described in Subsection 1.2, is the method of iterating a function until one gets to a point that is close to being a fixed point of the function. This is simple and practical, in the sense that it has been tested on problems at the scale of the the world's largest school choice problems, with satisfactory running times. It is conceptually imperfect insofar as it need not converge in finitely many steps, and thus is defined only up to some tolerance for error.

The second method, which is a bit more complicated and is described in Subsection 5.2, uses linear algebra to achieve convergence in finitely many steps.

We have included both methods for two reasons. The first is that by checking to see that they give the same result, we are assured that the codes for both are correct. The second is that the two methods can be combined. Intuitively, the function iteration method quickly gets close to the final solution, but then takes many iterations to achieve the desired accuracy. If the linear algebra method is started far from the final solution, then there are a great many steps in which, in effect, the path of the method changes direction, but if it is started near to the final solution, the final solution will be reached after only a few iterations. As the code is configured, the function is iterated until the total error falls below 1.0 (this is arbitrary, and has not been optimized) after which the linear algebra method takes over. This seems to give running times that are at least a little better than either method by itself.

5.1 `fdamcc.h` and `fdamcc.c`

The function `mcca_alloc_plus_coarse_cutoffs` computes the MCC-a allocation. It first sets all of the `cutoffs` to zero, and then applies the function `naive_increase_of_cutoffs` again and again until the total `excess_sum` (that is, the sum, over schools, of the extent to which total demand for the school exceeds the school's quota) falls below 0.000000001. (The amount of acceptable error is set so small in order to avoid `MCC_alloc_plus_coarse_cutoffs` getting into an infinite loop in which it repeatedly computes the same `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 function `naive_increase_of_cutoffs` first computes the demands resulting from the cutoffs. Then, for each school with demand in excess of its quota, it revises the cutoff for that school to the cutoff that would result in that school's demand exactly equalling the quota if no other school changed its cutoff. Increasing the cutoff can be thought of as rejecting a certain mass of students who (through their demands) are "applying" for admission to the school, and for this reason Kesten and Ünver (2015) describe this computational procedure as *fractional deferred acceptance*.

The function `naive_eq_cutoff` computes the `cutoff[j-1]` that would reduce the total demand for school `j` to school `j`'s quota. For a candidate 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 at `new_cand` is close enough to the quota, we are done. Otherwise we let `midpoint` be the midpoint of the interval `[lower_cand, upper_cand]`, and we replace this interval with either `[lower_cand, midpoint]` or `[midpoint, upper_cand]`, according to whether the demand at `midpoint` is less than or greater than the quota. This subdivision process is repeated until a satisfactory approximation is obtained, at which point the function returns `new_cand`.

The function `mccb_alloc_plus_coarse_cutoffs` computes the MCC-b allocation by

first setting all of the `cutoffs` to a sufficiently high number, then repeaatedly applying the function `naive_decrease_of_cutoffs` until the total `deficit_sum` falls below 0.000000001. The function `naive_decrease_of_cutoffs` computes the demands resulting from the cutoffs, then, for each school `j` with demand below its quota, it decreasees `cutoffs[j-1]` to the cutoff that would result in that `j`'s demand exactly equalling its quota if no other school changed its cutoff, or to zero if no such cutoff exists. Thus the computation of the MCC-b allocation is a simple mirror of the computation of the MCC-a allocation.

5.2 `mcccode.h` and `mcccode.c`

We now describe an algorithm that computes MCC-a in finite time. The discussion below assumes familiarity the notation for demand introduced in Subsection 1.2 and with the linear algebra described in Subsection 4.3.

As with the algorithms described above, this algorithm computes an increasing sequence $c^0, c^1, c^2, \ldots$ where $c^s \leq c^{s+1}$ and $c^s \leq J(c^s)$ for all s. Suppose that the computation has reached a profile of cutoffs c^s, which we denote by c for the sake of brevity. We now describe the computation of c^{s+1} in the function `linear_adjustment_a`.

Let $O_c = \{ j \in O : D_j(c) \geq q_j \}$ be the set of *active schools*. (`active_schools[j-1]` is 1 if `j` is in this set and 0 otherwise.) Suppose that $O_c = \{j_1, \ldots, j_k\}$. For each school `j`, `lexicon[j-1]` is 0 if `j` is not an element of O_c, and it is `h` if `j` $= j_h$, in which case `rev_lex[h] = j`.

We replace each c_j with the largest number c'_j such that $D_j(c'_j, c_{-j}) = D_j(c)$ using the function `adjust_cutoffs_to_create_constrained_students_a`. For a school $j \in O_c$ let

$$E_j = \{ i : e_{ij} = C(c_j) \text{ and } D_{ij}(c) = 1 - (c_j - C(c_j)) \}$$

be the set of students who are not prohibited from consuming o, due to deficient priority, but whose consumption of o is constrained by c_j. Since increasing c_j slightly decreases demand for o, E_j is nonempty. For each $j \in O$, `constrained_students[j-1]` is `NULL` if $o \notin O_c$, and otherwise the pointee of `constrained_students[j-1]` is E_j in `index` form.

For each student i it may be the case that she is consuming only a school o to which her access is unlimited because $e_{ij} > C(c_j)$. (In this case `next_school[i-1]` is 0.) Otherwise there is a school $j_i =$ `next_school[i-1]` to which her demand would be displaced if her demand for some other school were reduced because that school increased its cutoff. Often this is the worst

school that i is actually consuming, but it could also be the next best school i is eligible for if consumption of the worst school that i is actually consuming would be constrained by a small increase in that school's cutoff. For distinct schools $j \in O_c$ and j' let

$$F_{jj'} = \{\, i \in E_j : j_i = j' \,\}.$$

The $k \times k$ matrix $\Gamma =$ `displacements` is the $k \times k$ matrix with diagonal entry $\gamma_{ll} = |E_{j_l}|$ and off-diagonal entry $\gamma_{lm} = -|F_{j_l j_m}|$ if $m \neq l$. For each $l = 1, \ldots, k$, row l of Γ is clearly WDD, and it is SDD if and only if there is an $i \in E_{j_l}$ such that $j_i \notin O_c$. Let $T =$ `T_sub` be the set of l such that row l of Γ is not SDD and there is no walk in the directed graph of Γ from l to the index of a row that is SDD. There are now two cases.

If $T = \emptyset$ we use `adjust_cutoffs_empty_T_sub_a` to adjust the cutoffs. In this case Γ is WCDD, hence invertible. Let $\varepsilon = (\varepsilon_1, \ldots, \varepsilon_k)$ be the vector with components $\varepsilon_l = D_{j_l}(c) - q_{j_l}$, and let $\delta = (\Gamma^t)^{-1}\varepsilon$. Since the diagonal entries of Γ are positive and the off-diagonal entries are nonpositive, the entries of $(\Gamma^t)^{-1} = (\Gamma^{-1})^t$ are all nonnegative, so $\delta \geq 0$. We define $c(t)$ for $t \geq 0$ by setting $c_{j_l}(t) = c_{j_l} + \delta_l t$ for $l = 1, \ldots, k$ and $c_j(t) = c_j$ for $j \notin O_c$. The idea is to let $c^{s+1} = c(\overline{t})$ where $\overline{t}$ is the largest $t \in [0, 1]$ such that $D_{j_l}(c(t)) = D_{j_l}(c) - \varepsilon_l t$ for all $l = 1, \ldots, k$ and $D_j(c(t)) \leq q_j$ for all $j \in O \setminus O_c$.

Let t^* be the smallest number at which, for some $l = 1, \ldots, k$, one of the following happens:

(a) for some $j \in O \setminus O_c$, $D_j(c(t)) = q_j$;

(b) $c_{j_l}(t^*) = C(c_{j_l}) + 1$;

(c) for some $i \notin E_{j_l}$, $e_{ij_l} = C(c_{j_l})$ and $D_{ij_l}(c(t^*)) = g_{ij_l}(e_{ij_l}, c_{j_l}(t^*))$.

That is, either $O_{c(t^*)}$ is a proper superset of O_C, of, for some l, either c_{j_l} increases to the point where $C(c_{j_l})$ increases, or, for some i, the difference between the amount of j_l that i is allowed to consume and the amount she is consuming decreases to 0. Usually $\overline{t} = \min\{t^*, 1\}$, and it is always the case that $\min\{t^*, 1\} \leq \overline{t}$, so we actually set $c^{s+1} = c(\min\{t^*, 1\})$.

If $c^{s+1} = c(1)$, then $D_{j_l}(c^{s+1}) = q_{j_l}$ for all l and $D_j(c^{s+1}) \leq q_j$ for all $j \in O \setminus O_c$, so c^{s+1} is an MCC. Furthermore, $c(t) \leq J(c(t))$ for all $t < 1$, with strict inequality in at least one component, so $c^{s+1} =$ MCC-a. Thus the algorithm can halt, and we return `done = 1`.

The first times that (a) and (b) occur are computed by `time_until_new_active_school` and `time_until_new_coarse_cutoff`. There may not be a time $t \geq 0$ such that (c) holds, in

which case `time_until_new_constrained_student` returns -1, and otherwise it returns the first time such that (c) occurs.

When $T \neq \emptyset$ we use `adjust_cutoffs_nonempty_T_sub_a` to adjust the cutoffs. Possibly after replacing T with a nonempty proper subset, it will be the case that T is minimal in the sense that there is no nonempty proper subset T' such that $\{ m : \gamma_{lm} \neq 0 \} \subset T'$ for all $l \in T'$. Without loss we may assume that $T = \{1, \ldots, n\}$.

It turns out that in this situation the origin is in the interior (in $\{ x \in \mathbb{R}^n : \sum_l x_l = 0 \}$) of the convex hull of the rows of Γ'. (For the proof of this see "Efficient Computationally Tractable School Choice Mechanisms".) That is, there is a vector $\delta = (\delta_1, \ldots, \delta_n)$ with positive components such that for each $l = 1, \ldots, n$, $\sum_{m=1}^{n} \delta_m \gamma_{lm} = 0$. The computation of δ by means of linear algebra is carried out by `get_delta_nonempty_T_sub`. We define $c(t)$ for $t \geq 0$ by setting $c_{j_m}(t) = c_{j_m} + \delta_m t$ for $m = 1, \ldots, n$ and $c_j(t) = c_j$ for $o \notin \{j_1, \ldots, j_n\}$. We let $c^{s+1} = c(t^*)$ where t^* is the smallest number at which, for some $l = 1, \ldots, n$, (b) or (c) holds. (Since $D_j(c(t)) = D_j(c)$ for all j and $t \in [0, t^*]$, (a) is irrelevant.)

The overall computation is governed by `mcca_alloc_plus_coarse_cutoffs`, which applies `compute_new_cutoffs_a` until this function signals (by `done` = 1) that that computation is complete, which is to say that `cutoffs` is MCC-a. In `compute_new_cutoffs_a` the function `naive_increase_of_cutoffs` is applied repeatedly until the total excess of demand above quotas falls below one, after which it applies `linear_adjustment_a` repeatedly. The intuition is that when `cutoffs` is far from MCC-a, `naive_increase_of_cutoffs` can move relatively rapidly, whereas `linear_adjustment_a` has to navigate a large number of events of types (a), (b), and (c). On the other hand, when `cutoffs` is close to MCC-a, `naive_increase_of_cutoffs` moves very slowly, so that many iterations are required to reach the desired accuracy, but one can hope that a few iterations of `linear_adjustment_a` will suffice. Some computational experience supports this intuition, but there has been no serious attempt to optimize performance.

The computation of MCC-b is symmetric. However, for a student `i`, `next_school[i-1]` is replaced by `last_school[i-1]`. These are the same except when `i` is consuming exactly the amount allowed by the cutoff of the last school with positive consumption, in which case `last_school[i-1]` is this school (that is, the school whose consumption by `i` decreases as other cutoffs are reduced) and `next_school[i-1]` is next best school that `i` is eligible to consume (which is the school whose consumption by `i` would increase if other schools increased

their cutoffs). This creates various specific differences in the code, but does not affect its conceptual structure.

6 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 be the *quota* of school j. Initially this is a positive integer, but our algorithm is recursive, and when it calls itself on subproblems, the q_j of the subproblem may not be integral.

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[3].

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

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

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 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 α_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 $\sum_i \overline{p}_{ij_h}(t_h) < q_{j_h}$ 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 $E - p(t^*)$.*

6.1 `defaccep.h` and `defaccep.c`

As we explained above, `gcps` follows a path in the set of feasible allocations that lies above the path of the allocation, and its first task is to compute a feasible allocation at which this path can begin. The algorithm we use to solve this computational problem is none other than our old friend DA (student-proposing deferred acceptance)!

The key point to understand is that the outcome of DA does not depend on the order in which the students are rejected. That is, the DA outcome is the result if we simply keep rejecting, one by one, in any order, those students who are known to be rejectable, because the school can already fill its quota with higher priority students. One can argue for this inductively. If one does this, then every student who is rejected in the first round of DA (as it is usually described) will certainly be rejected eventually. Once those rejections have taken place, it is clear that that every second round rejection will also transpire, and so forth.

The function `deferred_acceptance` implements DA by updating `applicant_lists`, which is a list of lists of applicants. This lists of lists is initialized by having each student apply to her most preferred school. After that the function cycles repeatedly through the schools, asking each if it has a student it can reject because its quota is exceeded. (Implicitly a student has higher priority at all schools than another if her number is lower.) If it has such a student, the function `reject_student` removes it from the school's applicant list and has it apply to the next best school in the student's ranking. This continues until there is a round in which no school has a student to reject, at which point `applicant_lists` becomes the final allocation.

6.2 `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_exhausted` 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_of_segment` bundles these together.

6.3 `pivot.h` and `pivot.c`

Recall that, in the process as we described it above, a sequence $j_0, i_1, j_1, \ldots, j_{h-1}, i_h, j_h$ that is used to adjust θ 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`.

6.4 `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.

6.5 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 computational 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 `deferred_acceptance`, 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.

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

Kesten, O. and Ünver, M. U. (2015), 'A theory of school-choice lotteries', *Theor. Econ.* **10**, 543–595.

Shivakumar, P. N. and Chew, K. H. (1974), 'A sufficient condition for nonvanishing of determinants', *Proc. Am. Math. Soc.* **43**, 63–66.