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

Andrew McLennan*

November 16, 2024

Abstract

This document provides a brief introduction to the software package GCPS MCC Schools.

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

A *school choice mechanism* is an algorithm that takes the preferences and the priorities as inputs and outputs an assignment of each student to one of the schools she ranked that is *feasible*, in the sense that the number of students assigned to each school is not greater than the school's

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

capacity. The most commonly used school choice mechanism, known as *deferred acceptance*, has some undesible features, which we describe in Subsection 1.5.

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

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

1.1 The GCPS Mechanism

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

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

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

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

We say that a bistochastic matrix of assignment probabilities is sd-*efficient* if there does not exist a second such matrix that gives each agent a distribution over O that stochastically dominates the distribution given by the first matrix, with strict domination for some agents. It turns out that random priority can produce a matrix of assignment probabilities that is not sd-efficient, but the matrix of assignment probabilities produced by the PS mechanism is always sd-efficient.

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

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

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

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

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

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

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

A school may become unavailable if its capacity is exhausted, but it can also happen that at a certain time, a set of schools has only enough capacity to meet the needs of the students who are not eligible to attend schools that are outside the set and are still available. Formally, a pair (J, P) of subsets $J \subset I$ and $P \subset O$ is *critical* at time t if, for every $i \in J$, the only schools that are still available to i are contained in P, and the remaining capacity of the schools in P is just enough to meet the remaining demand of the students in J. When this happens, students outside of J become ineligible to consume additional probability of schools in P. 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 *generalized constrained probabilistic serial* (GCPS) mechanism is implemented in the command `gcps`, which takes the schools' capacities and the students' eligibilities and declared preferences as inputs, and which outputs a feasible allocation if one exists. In order for `gcps` to produce a feasible allocation, a feasible allocation must exist, of course, and insuring that this is the case is a responsibility of the user. One way to do this is to choose an assignment of safe schools that is feasible, in the sense that no school is assigned more students than its capacity. If there is no feasible allocation, `gcps` will simply tell that and quit, so in some contexts it may be possible to apply trial and error.

There is a lot more to say about the GCPS mechanism, but for the time being we only make a couple comments. First, the feasible allocations produced by the GCPS mechanism are sd-efficient, although this is not obvious at this point. As we will describe in detail later, the allocations produced by the mechanism that is currently most commonly used for school choice are not sd-efficient.

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 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 who have the same second favorite object as you 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 you say that your second favorite object is actually your favorite, you will share it with the two others 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 two favorite objects, this can be beneficial.

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

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

1.2 The MCC Mechanism

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

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

A *coarse cutoff* for a school is a nonnegative integer, and a *coarse cutoff profile* is a specification of a coarse cutoff for each school. A feasible allocation *fulfills* a coarse cutoff profile if:

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

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

(c) when a school's capacity is not fully utilized (the sum of its assignment probabilities is less than its capacity) the school's coarse cutoff is zero.

A *fine cutoff* for a school is a nonnegative real number, and a *fine cutoff profile* is a specification of a fine cutoff for each school. For a school o, a fine cutoff c_o *refines* a coarse cutoff C_o if $C_o \leq c_o < C_o + 1$. Evidently a fine priority refines a unique coarse priority, and for a fine cutoff profile, the coarse cutoff profile refined by it is the coarse cutoff profile whose component, for each school, is the coarse cutoff refined by the school's fine cutoff. A feasible allocation *fulfills* a fine cutoff profile if:

(a) it fulfills the coarse cutoff profile that the fine cutoff profile refines;

(b) for each student i and school o, if i's priority at the school is equal to the school's coarse priority C_o, then the probability that i is assigned to o is not greater than $C_o + 1 - c_o$, where c_o is o's fine cutoff;

(c) when a school's capacity is not fully utilized, the school's fine cutoff is zero.

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

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

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

1.3 Enhanced MCC Mechanisms

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

Longer cycles of exchange are also possible. In fact a feasible allocation that fulfills C is not strictly sd-dominated by another feasible allocation that fulfills C if and only if no such trading cycle is possible. There are many algorithms that pass from a feasible allocation that fulfills C to such an allocation that is not strictly sd-dominated by another feasible allocation that fulfills C; the idea is to repeatedly identify and execute such cyclic trades until no further possibilities remain. An *enhanced MCC mechanism* is a mechanism that first computes the MCC allocation, and the profile of coarse cutoffs C that it fulfills, and then uses such an algorithm to compute a feasible allocation that fulfills C and is not strictly sd-dominated by any other feasible allocation that fulfills C. The command `emcc` implements one such mechanism.

An enhanced MCC mechanism may fail to be strategy proof in the large. A student may benefit, even when the population is large, in elevating a popular school in her reported preference, if she is confident that she will be able to trade probability of that school for probability of a school she really wants.

1.4 Generating a Random Deterministic Assignment

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

We begin with a feasible assignment m with entries m_{io}. The algorithm works by transitioning from m to a feasible assignment m^{α} with probability $\frac{\beta}{\alpha+\beta}$ and transitioning from m to a feasible assignment $m^{-\beta}$ with probability $\frac{\alpha}{\alpha+\beta}$, where $\frac{\beta}{\alpha+\beta}m^{\alpha} + \frac{\alpha}{\alpha+\beta}m^{-\beta} = m$. All the entries that are integral in m are integral in m^{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, 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 $m^{-\beta}$. 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 $m^{-\beta}$. We form an undirected graph $G = (V, E)$ whose set of *vertices* V consists of the students, the schools, and an artificial node called the *sink*. The set of *edges* E contains an edge between a student i and a school o if m_{io} is not an integer, so neither 0 nor 1, and E contains an edge between a school and the sink if the total probability $\sum_i m_{io}$ of assignment to that school is not an integer. If $E = \emptyset$, then every probability in the assignment is either 0 or 1, so it is a deterministic assignment, and we are done.

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

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

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

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

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

For $\alpha > 0$ consider the matrix m^α of numbers obtained by increasing $m_{w_i w_{i+1}}$ by α when the edge between w_i and w_{i+1} is a forward edge and decreasing $m_{w_{i+1} w_i}$ by α 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^α 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^α is the same as in m. Therefore m^α is a feasible assignment if α is sufficiently small.

Similarly, for $\beta > 0$ consider the matrix $m^{-\beta}$ of numbers obtained by decreasing $m_{w_i w_{i+1}}$ by β when the edge between w_i and w_{i+1} is a forward edge and increasing $m_{w_{i+1} w_i}$ by β when the edge between w_i and w_{i+1} is a backward edge. As above, $m^{-\beta}$ is a feasible assignment if β is sufficiently small. Clearly $\frac{\beta}{\alpha+\beta} m^\alpha + \frac{\alpha}{\alpha+\beta} m^{-\beta} = m$. If we choose α to 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, and we choose β similarly, then all the conditions described above are satisfied.

1.5 The Main Competitor: Deferred Acceptance

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

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

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

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

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

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

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

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

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

Let 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 $\mu'(R) = \{ \mu'(b) : b \in R\}$. Of course $\mu'(S) = R$.

We claim that $\mu(S) = R$. Let Beth be an element of S. Then there is an element of R, say Abe, such that $\text{Beth} = \mu'(\text{Abe})$. Let Carl $=$ μ(Beth), and let Doris $=$ μ'(Carl). 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. 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, if Carl and Albert are different, then the stability of μ' (with respect to the preferences modified by Albert's manipulation) implies that Carl prefers Doris to Beth, so Carl is an element of R. Of course if Carl is Albert, then Carl is an element of R, so we have shown that $\mu(S)$ is a subset of R. Since the matching is one-to-one, it follows that $\mu(S) = R$.

Under DA for the true preferences, leading to μ, there is a last round in which an element of R makes a proposal. Since every boy in R prefers his partner in μ' to his partner in μ, every girl in S has already rejected her partner in μ' prior to this round. Let Don be one of the elements of R who proposes in this round, and let Ella be the girl he proposes to. When Don proposes to Ella, she is holding a proposal from a boy Fred who she rejects in favor of Don. Since Fred has more proposing to do, Fred is not an element of R and thus Fred is not Ella's partner in 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 *final priority*, which is the strict ordering that is an input to the DA algorithm.

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

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

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

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

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

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

2 For the User

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

2.1 Downloading and Setting Up

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

First, in a web browser, open the url

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

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

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

```
$ tar xvf gcps_mcc.tar
$ cd gcps_mcc
```

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` and see the text that the command directs to the screen:

```
$ make
gcc -I. -Wall -Wextra -fsanitize=address -g -c normal.c
gcc -I. -Wall -Wextra -fsanitize=address -g -c parser.c
gcc -I. -Wall -Wextra -fsanitize=address -g -c subset.c
gcc -I. -Wall -Wextra -fsanitize=address -g -c cee.c
gcc -I. -Wall -Wextra -fsanitize=address -g -c schchprob.c
gcc -I. -Wall -Wextra -fsanitize=address -g -c partalloc.c
gcc -I. -Wall -Wextra -fsanitize=address -g -c pushrelabel.c
gcc -I. -Wall -Wextra -fsanitize=address -g -c pivot.c
gcc -I. -Wall -Wextra -fsanitize=address -g -c endpoint.c
gcc -I. -Wall -Wextra -fsanitize=address -g -c gcpscode.c
gcc -o gcps gcps.c normal.o parser.o subset.o cee.o schchprob.o
partalloc.o pushrelabel.o pivot.o endpoint.o gcpscode.o -fsanitize=address
-static-libasan -lm
gcc -I. -Wall -Wextra -fsanitize=address -g -c mcccode.c
gcc -o mcc mcc.c mcccode.o partalloc.o subset.o normal.o parser.o
cee.o schchprob.o -fsanitize=address -static-libasan -lm
gcc -I. -Wall -Wextra -fsanitize=address -g -c emcccode.c
```

```
gcc -o emcc emcc.c emcccode.o mcccode.o partalloc.o subset.o normal.o
parser.o cee.o schchprob.o -fsanitize=address -static-libasan -lm
gcc -I. -Wall -Wextra -fsanitize=address -g -c purifycode.c
gcc -o purify purify.c normal.o parser.o subset.o partalloc.o purifycode
-fsanitize=address -static-libasan -lm
gcc -I. -Wall -Wextra -fsanitize=address -g -c makexcode.c
gcc -o makex makex.c normal.o makexcode.o -fsanitize=address -static-lib
-lm
```

We have now constructed the executable files `gcps`, `mcc`, `emcc`, `purify`, and `makex`. (The compilation process also constructed various "object" files with file names ending with `.o`, that are intermediated steps in the compilation process.) On many Unix's these can invoked simply by typing the executable name on the command line, but it may be the case that, for security reasons, the current directory is not in the `path` (the list of directories that the command line looks in when a command is invoked) in which case you will need to type `./gcps`, `./mcc`, etc.

2.2 Input Files

The executables `gcps`, `mcc`, and `emcc` act on *school choice problem* input files. The executable `purify` acts on *partial allocation* input files. We now describe the formats of these files. We will study an example of a school choice input file, but the rules specified below apply to both types of files.

Among the files you downloaded, there is a sample school choice problem file `my.scp`:

```
$ cat my.scp
/* 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 1 priority grades.  */
There are 6 students and 2 schools
The vector of quotas is (4,4)
The priority matrix is
0 0
```

```
0 0
0 0
0 0
0 0
0 0
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
```

GCPS MCC Schools input files begin with a comment between `/*` and `*/`. This is purely for your convenience. The comment can be of any length, and provide whatever information is useful to you, but it is mandatory insofar as the computer will insist that the first two characters of the file are `/*`, and it will only start extracting information after it sees the `*/`. As you can see, when `makex` makes an input file, it uses the comment to provide some information concerning how the file was made; removing or replacing this won't change how the file is processed by `gcps`, `mcc`, and `emcc`.

The computer divides the remainder of the file into "generalized white space" and "tokens." Generalized white space includes the usual white space characters (spaces, tabs, and new lines), and in addition '`(`', '`)`', and '`,`' are treated as white space. Tokens are contiguous sequences of characters without any of the generalized white space characters. Tokens are either prescribed words, nonnegative integers, positive integers, or student or school tags (a student or school number followed by '`:`'). Everything must be more or less exactly as shown above, modulo white space, 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 a file and finds a violation of the format requirements, it will print a short statement describing the problem and quit.

The second line after the comment gives the quotas (i.e., the capacities) of the schools, so both schools have four seats. Here we see the convenience of making '(', ')', and ',' white space characters: otherwise we would have to write `The vector of quotas is 4 4`.

In general the lowest priority level is 0, which means that the student may receive some probability of a seat at the school if she is eligible and not crowded out by students with higher priorities. The schools' priorities play no role in `gcps`, which is to say that in `gcps` the priorities are *dichotomous*: for each school and student, either the student is eligible to attend the school (and her preferences rank that school) or she is not, and the school gives equal consideration to all eligible students. A student may be ineligible to attend a school if she is not qualified (it is a single sex school for boys, or her test scores are too low) or she may be ineligible because she weakly prefers a seat at her safe school.

The next line provides information (for each student, the number of schools she ranks) that the computer could figure out for itself, but we prefer to confirm that whatever person or software prepared the input knew what they were doing. After that come the students' preferences: for each student, that student's tag followed by the schools she might attend, listed from best to worst. The collection of information provided by such an input file is a *school choice problem*. We recommend file names for input files for `gcps` that end with `.scp`, but the software does not enforce this.

2.3 gcps, mcc, and emcc

We now imagine that you've done the hard work of figuring out what each school's capacity is, which schools each student is eligible to attend, assigning a safe school to each student, and getting each student to provide the rank ordered list of schools she is eligible for and likes at least as much as her safe school. All this information has been encoded in an input file `my.scp` that is in the current directory. We assume that the executable `gcps` is also in this directory. The next step could be to issue the command:

```
$ ./gcps my.scp
```

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. For security reasons some flavors of Unix do not put the current directory (denoted by `.`) in the

PATH, so you need to tell the computer that that is where you want it to look. Thus `./gcps my.scp` is telling the computer to apply the version of `gcps` in the current directory to the file `my.scp`. (It is also possible to run `gcps` without specifying an input file, in which case it will look for a file `schools.scp` in the current directory.)

If `my.scp` is the input file above, `./gcps my.scp` gives the output shown below.

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

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

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

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

For the discussion of `mcc` and `emcc` we consider a slightly different input file.

```
$ cat your.scp
/* 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 3 priority grades.  */
There are 6 students and 2 schools
The vector of quotas is (4,4)
The priority matrix is
0 0
0 1
1 0
1 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 only difference in the parameters used to prepare `your.scp` is that now each school has three priority classes. Note that the students who rank only school 1 (students 3, 4, and 5) all have priorities among the top three for that school. This guarantees that they will be admitted to school 1, which is their safe school.

We apply `mcc` to this input file:

```
$ ./mcc your.scp
/* This is a sample introductory comment.  */
There are 6 students and 2 schools
          1:          2:
1:  0.00000000 1.00000000
2:  0.50000000 0.50000000
```

```
3:  1.00000000 0.00000000
4:  1.00000000 0.00000000
5:  1.00000000 0.00000000
6:  0.50000000 0.50000000
```

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

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

2.4 `purify`

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

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

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

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

Implementation can be accomplished by issuing the command:

```
$ ./purify my.mat
```

When we run the command above we get:

```
/* This is a sample introductory comment.  */
    1: 2:
1:   0 1
2:   0 1
3:   1 0
4:   1 0
5:   1 0
6:   1 0
```

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`, `mcc`, and `emcc` that reflect the geographical dispersion of schools within school districts with many schools, and the idiosyncratic nature of school quality and student preferences.

We 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. Students for whom the school is the safe school are reassigned numerical priorities that are high enough to insure that they will in the top group of students that is certain to be admitted if they apply. After this, the adjusted numerical priorities give an ordinal ranking of the students who have ranked the school, and these 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 more student than a higher priority class. For example, if there are five students and three priority classes, the bottom two priority classes will each have two students and the top priority class will have one student.

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 the user's own programming effort in adapting it to the particular requirements and idiosyncratic features of the user's 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.

A About the Code

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

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

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

In comparison with languages such as `Python`, C certainly has some disadvantages. Organizing the hierarchical structure of the code using curly brackets rather than indentation makes the code bulky, but this is primarily an aesthetic concern. More serious is the fact that in C the programmer is responsible for explicitly allocating and deallocating memory. Historically this gave rise to major headaches, because it was easy to accidently write to or access a part of memory that had not actually been allocated, or to accidently overwrite some previously allocated memory, without the computer complaining at all. Such bugs were notoriously hard to track down. If you look in the `makefile` you will see that for Linux there are the additional `CFLAGS` `-fsanitize=address` and `-g` and the additional `LDFLAGS` `-fsanitize=address` and `-static-libasan`, while for Mac OS the last flag becomes `-static-libsan`. These flags invoke `addresssanitizer`, which results in compilation of executables that check for memory errors such as those described above. On Linux, but not on Mac OS, the executables also check for memory leaks, which are allocated memory that is not deallocated at the end of run time. Of course this checking is a burden that slows execution and eats up additional memory, so you may want to use `addresssanitizer` only while debugging.

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

In the C programming language, an n element array is indexed by the integers $0, \ldots, n-1$. 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.

A.1 Overview of the Code

Compiling (by typing `make` on the command line) produces the executables `makex`, `gcps`, `mcc`, `emcc`, and `purify`, which can be invoked from the command line, or by using shell scripts or scripting languages such as `perl`. For `makex` there are associated files `makex.c`, `makexcode.h`, and `makexcode.c`. For each of the other executables there are similarly titled associated files, and for each executable the division of labor between them is similar. In all cases the relationship between the header `file.h` and the source `file.c` is as usual: `file.h` includes other resources, defines `struct`'s, and states the signatures of the functions defined in `file.c`. We will not mention the header files in our subsequent discussion. The code in `makex.c` only manages the interaction with the command line, and, once it is has sorted out the relevant parameters, it invokes the function `make_example` defined in `makexcode.c`.

The code in the `code.c` files is organized, roughly, in a top down manner, with the "big picture" functions at the beginning, and functions providing greater detail organized in a roughly hierarchical manner lower down. (This is vivid in the code for `make_example`, but true to some greater or lesser extent for all the executables.

The code in `normal.c` provides simple numerical functions, and the code in `subset.c` gives operations on subsets of the set of integers from 1 to some finite upper bound. The code in `cee.c` defines and provides support for collective endowment economies, the code in `schchprob.c` defines and provides support for school choice problems, and the code in `partalloc.c` defines and provides support for partial allocations. The functions defined in `parser.c` read input files, which are either school choice problems or partial allocations.

Perhaps the most important thing to understand is that the code for `gcps` is *much* more complex than the code for the other executables. In addition to `gcps.c` and `gcpscode.c`, `gcps` uses the code in `endpoint.c`, `pivot.c`, and `pushrelabel.c`. We will have much more to say about it below.

A.2 `normal.c`

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

A.3 subset.c

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

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

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.

A.4 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 there are situations in which the schools have been partially allocated and the remaining amounts to be allocated are no longer integers. The computations of `gcps` use 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. It also has a member `eligible` which is a matrix whose entry `eligible[i-1][j-1]` is 1 if `j` is one of the schools ranked by `i` and 0 otherwise. This entry does not provide independent information, and is computed from the `preferences` by the function `compute_eligibility_matrix`.

A pair (J, P) consisting of a set of students J and a set of schools P is *critical* if the only way to meet the requirement of the agents in J is to give each student in J her maximum of each of the schools outside P, and to give all of the seats in schools in P only to students in J. During the `gcps` computation, when the pair (J, P) becomes critical for the as yet unallocated resources, the computation recursively descends to two subprocesses, one for the agents in J and all the resources that they might possibly consume (which is the minimum needed to meet their requirements) and the other for students in the complement of J and schools in the complement of P. The functions `critical_sub_process_scp` and `crit_compl_sub_process_scp` compute the `process_scp`'s of these subprocesses.

A.5 `makex.c` 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.

A.6 `partalloc.h` and `partalloc.c`

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

A `partial_alloc` is *feasible* if the total amount 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. When the process encounters a critical pair (as described earlier and in more detail later) the process 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.

A.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.

A.8 `purify.c` and `purifycode.c`

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

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

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

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

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

B Market Clearing Cutoffs, Enhanced

We now describe the computation of the market clearing cutoffs and enhanced market clearing cutoffs allocations.

B.1 `mcc.c` and `mcccode.c`

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

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

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

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

B.2 **emcc.c and emcccode.c**

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

The function EMCC_allocation first applies MCC_alloc_plus_coarse_cutoffs to get an allocation alloc_to_adjust and a profile coarse of coarse cutoffs. It then repeatedly finds cycles $(i_1, j_1), \ldots, (i_k, j_k)$ such that for each $h = 1, \ldots, k-1$, i_h is consuming a positive quantity of j_h in alloc_to_adjust, i_h prefers j_{h+1} to j_h, and i_h's priority at j_{h+1} is at least as large as `coarse[`j_{h+1}`]`. (Also i_k is consuming a positive quantity of j_k in `alloc_to_adjust`, i_k prefers j_1 to j_k, and i_k's priority at j_1 is at least as large as `coarse[`j_1`]`.) Each time it finds such a cycle it adjusts `alloc_to_adjust` by increasing each allocation of j_{h+1} to i_h (and of j_1 to i_k) by `Delta` while decreasing each allocation of j_h to i_h by `Delta`, where `Delta` is the largest amount allowed by the various constraints. When there are no more cycles it returns `alloc_to_adust`.

The struct `stu_sch_node` has three members: a student number, a school number, and a pointer to a `stu_sch_node`. A cycle, such as was described above, is a linked list of such nodes. The `cmatrix` in `EMCC_allocation` is a matrix such that for each student `i` and school `j`, `cmatrix[i-1][j-1]` is a pointer to a `stu_sch_node` that either points to the first node of a cycle containing `(i,j)` or is `NULL` if no such cycle exists. In order to contribute as little as possible to the many advantages enjoyed by people with names like Aaron Aardvark, `EMCC_allocation` chooses a `random_cycle` from `cmatrix`.

The function `get_cycle_matrix` first uses `get_envy_graph` to get `egraph`, which is also a matrix of pointers to `stu_sch_node`'s, but in this case `egraph[i-1][j-1]` is simply a list of pairs `(k,l)` such that `i` would like to get some `l` from `k` in exchange for some `j`. For each `i` and `j`, the function `get_cycle` is applied in order to construct a cycle containing `(i,j)` or show that no such cycle exists.

The function `get_envy_graph` first creates a matrix `active` where `active[i-1][j-1]` is 1 if `alloc_to_adjust` assigns a positive amount of `j` to `i` and `j` is not `i`'s favorite school, and 0 otherwise. For each `i` and `j` such that `active[i-1][j-1]` is 1, the function looks at all the schools `l` that `i` prefers to `j` at which `i` has high enough priority, and for each such `l` and each student `h` such that `active[h-1][l-1]` is 1 it appends `(h,l)` to `egraph[i-1][j-1]`.

The main operation in `get_cycle` is the construction of `previously_found`, which is a list of lists of pairs. The first list `previously_found[0]` is a copy of `egraph[i-1][j-1]`. Having constructed the lists for levels 1 up to `level-2`, `previously_found[level-1]` is constructed by looking at each of the pairs `(m,n)` in `egraph[k-1][l-1]` for each pair `(k,l)` in `previously_found[level-2]`. If `(m,n) = (i,j)`, then we have shown that a cycle exists. If `(m,n)` has not already been seen, then it is added to `previously_found[level]`. This process continues until a cycle has been shown to exist or no pairs are added to `previously_found[lev` which shows that there is no cycle that includes `(i,j)`. If a cycle has been shown to exist it is obtained from `extract_cycle`.

Other functions in `emcccode.c` do what their names suggest, in ways that are not too difficult to figure out.

C Computing the GCPS Allocation

C.1 Theoretical Background

It is now time to 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 $o \in O$ let $\omega_o = \{\, i : o \in \alpha_i \,\}$ be the set of students who might attend o. For each $o \in O$ let $q_o > 0$ be the *quota* of school o. Usually the given q_o will be an integer, but this is not necessary.

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

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

$$\sum_{i \in J_P^c} \sum_{o \in P} p_{io} \leq \sum_{o \in P} q_o - |J_P|$$

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

Recall that the GCPS allocation is $p(1)$ where $p\colon [0,1] \to R$ is the function such that $p(0)$ is the origin and at each time, each student is increasing, at unit speed, her consumption of her favorite school among those that are still available to her, with her other allocations fixed. It may happen that this process simply assigns each student to her favorite school, but the more important

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

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. Implementing this recursion requires a fair amount of code, but it is conceptually straightforward.

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. 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.

It turns out that the computation can be sped up by also computing a piecewise linear path $\overline{p}\colon [0, t^*] \to Q$ such that $p(t) \le \overline{p}(t)$ for all t. 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 $\theta_{io} = 0$ if $o \neq e_i$. For $t \le t^*$ we have $p(t) = \theta t$.

Suppose that for some time t_0 we have computed a $\overline{p}(t_0) \in Q$ such that $p(t_0) \le \overline{p}(t_0)$. We attempt to find a $\overline{\theta} \in \mathbb{Z}^{I \times O}$ such that for sufficiently small $\varepsilon > 0$, $\overline{p}(t_0) + \overline{\theta}\varepsilon \in Q$ and $p(t_0) + \theta\varepsilon \le \overline{p}(t_0) + \overline{\theta}\varepsilon$. If we can find such a $\overline{\theta}$, we let t_1 be the largest t such that $\overline{p}(t_0) + \overline{\theta}(t_1 - t_0) \in Q$ and $p(t_0) + \theta\varepsilon \le \overline{p}(t_0) + \overline{\theta}(t_1 - t_0)$, we replace t_0 with t_1 and $\overline{p}(t_0)$ with $\overline{p}(t_0) + \overline{\theta}(t_1 - t_0)$, and we repeat the calculation. If our attempt to find a suitable $\overline{\theta}$ fails, then $t_0 = t^*$, and it will turn oot that the failure of the attempt will produce a P that is critical at t^*.

For $\overline{\theta} \in \mathbb{Z}^{I \times O}$ it is the case that $p(t_0) + \theta\varepsilon \le \overline{p}(t_0) + \overline{\theta}\varepsilon$ for sufficiently small $\varepsilon > 0$ if and only if:

(a) For each i and o, if $\overline{p}_{io}(t_k) = p_{io}(t_k)$, then $\overline{\theta}_{io} \ge 1$ if $o = e_i$, and otherwise $\overline{\theta}_{io} \ge 0$.

For a $\overline{\theta}$ satisfying this condition, if $\overline{p}_{io}(t_0 = 0$, then $\theta_{io} \ge 0$, so $\overline{p}(t_0) + \overline{\theta}\varepsilon \in Q$ for sufficiently small $\varepsilon > 0$ if and only if, in addition:

(b) For each i and o, if $o \notin \alpha_i$, then $\overline{\theta}_{io} = 0$.

(c) For each i, $\sum_o \overline{\theta}_{io} = 0$.

(d) For each o, if $\sum_i \overline{p}_{io}(t_0) = q_o$, then $\sum_i \overline{\theta}_{io} \le 0$.

Our search for a suitable $\overline{\theta}$ begins 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 $\overline{\theta}^0_{io} = 0$ for all o. If $\overline{p}_{ie_i} = p_{ie_i}(t_0)$, then we set $\overline{\theta}^0_{ie_i} = 1$, we set $\overline{\theta}^0_{io_i} = -1$ for some $o_i \in \alpha_i \setminus \{e_i\}$ such that $\overline{p}_{io_i}(t_0) > p_{io_i}(t_0)$, and we set $\overline{\theta}^0_{io} = 0$ for all other o. Evidently $\overline{\theta}^0$ satisfies (a), (b), and (c).

Now suppose that $\overline{\theta}$ satisfies (a), (b), and (c), but not (d), and o_0 is an element of O such that $\sum_i \overline{p}_{io}(t_0) = q_o$ and $\sum_i \overline{\theta}_{io} > 0$. For $o \in O$ let

$$J(o) = \{\, i \in \omega_o : \text{if } \overline{p}_{io}(t_0) = p_{io}(t_0), \text{ then } \overline{\theta}_{io} > 1 \text{ if } o = e_i, \text{ and otherwise } \overline{\theta}_{io} > 0 \,\}$$

be the set of i such that decreasing $\overline{\theta}_{io}$ by one does not result in a violation of (a) or (b). For $i \in I$ let $P(i) = \alpha_i$. We define sets $P_0, J_1, P_1, J_2, \ldots$ inductively, beginning with $P_0 = \{o_0\}$ and continuing inductively with

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

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

If there is such an o_h we construct $i_1, \ldots, i_h$ and $o_1, \ldots, o_h$ by choosing $i_h \in J_h$ such that $o_h \in P(i_h)$, choosing $o_{h-1} \in P_{h-1}$ such that $i_h \in J(o_{h-1})$, choosing $i_{h-1} \in J_{h-1}$ such that $o_{h-1} \in P(i_{h-1})$, and so forth. Clearly $o_0, o_1, \ldots, o_h$ and $i_1, \ldots, i_h$ are distinct. We define $\overline{\theta}'$ by setting

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

for $g = 1, \ldots, h$ and $\overline{\theta}'_{io} = \overline{\theta}_{io}$ for all other (i, o). Since $\overline{\theta}$ satisfies (a) and $i_g \in J(o_{g-1})$ for all g, $\overline{\theta}'$ satisfies (a). Since $\overline{\theta}$ satisfies (b) and $i_g \in J(o_{g-1})$ and $o_g \in P(i_g)$ for all g, $\overline{\theta}'$ satisfies (b). Since $\overline{\theta}$ satisfies (c), $\overline{\theta}'$ satisfies (c). We have $\sum_i \overline{\theta}'_{io_0} = \sum_i \overline{\theta}_{io_0} - 1$ and $\max\{0, \sum_i \overline{\theta}'_{io}\} = \max\{0, \sum_i \overline{\theta}_{io}\}$ for all $o \ne o_0$, so repeating this maneuver will eventually produce a $\overline{\theta}$ satisfying (a)–(d) unless at some point it becomes impossible to find a satisfactory h and $i_1, \ldots, i_h$ and $o_1, \ldots, o_h$.

Now suppose that the construction terminates with $P_h = \emptyset$. Let $J = \bigcup_h J_h$ and $P = \bigcup_h P_h$. We have $\sum_i \overline{p}_{io}(t_0) = q_o$ for all $o \in P$. If $o \in P$ and $i \notin J$, then $i \notin J(o)$, so $\overline{p}_{io}(t_0) = p_{io}(t_0)$. If $i \in J$ and $o \notin P$, then $o \notin P(i) = \alpha_i$. 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 $1 - p_{io}(t_0)$ to $i \in J$ whenever $o \in O \setminus P$. Clearly any feasible allocation also has these properties, so (J, P) is a critical pair for $E - p(t_0)$.

C.2 Linear Programming

A different approach to computing the GCPS allocation is to solve the linear program

$$\max t \quad \text{subject to} \quad (t, m) \in [0, 1] \times Q \text{ and } \theta t \leq m.$$

I suspect that this is not very efficient, and probably infeasible for large school choice problems, because the matrices in the canonical formulation of linear programming for this problem have very large numbers of entries for large school choice problems. Nevertheless the proof of the pudding is in the eating, and others may wish to experiment, so a linear programming based algorithm has been included in the code.

We should emphasize that the software is not designed to handle more general problems than the ones that arise is the school choice application. For many more general applications, simple modifications should suffice, but that is up to the user. We should also emphasize that we have not applied any sophisticated optimization techniques.

The most general form of a linear programming problem is $\max \mathbf{c}x + c_0$ subject to $\mathbf{A}_= x = \mathbf{b}_=$, $\mathbf{A}_\leq x \leq \mathbf{b}_\leq$, and $x \geq 0$, where $\mathbf{A}_=$ and $\mathbf{A}_\leq$ are $m_= \times n$ and $m_\leq \times n$ matrices, $\mathbf{b}_= \in \mathbb{R}^{m_=}$, and $b_\leq \in \mathbb{R}^{m_\leq}$. A linear programming problem in *standard form* is a problem of the form $\max \mathbf{c}x + c_0$ subject to $\mathbf{A}x = \mathbf{b}$ and $x \geq 0$ where $\mathbf{A}$ is an $m \times n$ matrix and $\mathbf{b} \in \mathbb{R}^m$. The general problem is equivalent to the problem $\max \mathbf{c}x + c_0$ subject to $\mathbf{A}_= x = \mathbf{b}_=$, $\mathbf{A}_\leq x + s_\leq = \mathbf{b}_\leq$, $x \geq 0$, and $s_\leq \geq 0$, where $s_\leq$ is a vector of slack variables, so any problem can be reexpressed in standard form. (This is trivial, but of course we will need routines that do the conversion, and that convert a solution of the converted problem to a solution of the original problem.)

For the problem $\max \mathbf{c}x + c_0$ subject to $\mathbf{A}x = \mathbf{b}$ and $x \geq 0$ in standard form, we can attain the additional condition $\mathbf{b} \geq 0$ by replacing each constraint $\sum_j \mathbf{a}_{ij} x_j = \mathbf{b}_i$ such that $\mathbf{b}_i < 0$ with the equivalent condition $-\sum_j \mathbf{a}_{ij} x_j = -\mathbf{b}_i$. A key point of the simplex algorithm is that the condition $\mathbf{b} \geq 0$ is preserved at each step.

Suppose that we can find a collection of m variables such that (after rearranging the columns of A to put these variables at the beginning) $\mathbf{A} = [\mathbf{B}\ \mathbf{N}]$, $\mathbf{B}$ is invertible, and $\mathbf{b}' = \mathbf{B}^{-1}\mathbf{b} \geq 0$. We say that the collection of m variables is a *feasible basis*. Let $\mathbf{A}' = \mathbf{B}^{-1}\mathbf{A} = [\mathbf{I}_m\ \mathbf{N}']$ where

$\mathbf{N}' = \mathbf{B}^{-1}\mathbf{N}$. If $x' = [x'_{\mathbf{B}}\ x'_{\mathbf{N}}]$, then the condition $\mathbf{A}x' = \mathbf{b}$ becomes $\mathbf{b}' = \mathbf{A}'x' = x'_{\mathbf{B}} + \mathbf{N}'x'_{\mathbf{N}}$, so $x'_{\mathbf{B}} = \mathbf{b}' - \mathbf{N}'x'_{\mathbf{N}}$ and thus the basis variables $x'_{\mathbf{B}}$ can be expressed as a function of $x'_{\mathbf{N}}$. If $\mathbf{c} = [\mathbf{c}_{\mathbf{B}}\ \mathbf{c}_{\mathbf{N}}]$, then

$$\mathbf{c}x' = \mathbf{c}_{\mathbf{B}}x'_{\mathbf{B}} + \mathbf{c}_{\mathbf{N}}x'_{\mathbf{N}} = \mathbf{c}_{\mathbf{B}}(\mathbf{b}' - \mathbf{N}'x'_{\mathbf{N}}) + \mathbf{c}_{\mathbf{N}}x'_{\mathbf{N}} = \mathbf{c}_{\mathbf{B}}\mathbf{b}' + (\mathbf{c}_{\mathbf{N}} - \mathbf{c}_{\mathbf{B}}\mathbf{N}')x'_{\mathbf{N}}.$$

Consider a nonbasis variable, say x_i, such that the i-component of $\mathbf{c}_{\mathbf{N}} - \mathbf{c}_{\mathbf{B}}\mathbf{N}'$ is positive. Beginning at $[\mathbf{b}\ 0]$, if we increase x'_i while the other nonbasis variables are fixed at 0, the objective function increases. We can do this until further increase would cause some component of $x'_{\mathbf{B}} = \mathbf{b}' - \mathbf{N}'x'_{\mathbf{N}}$ to become negative. (This may happen even when $x'_i = 0$.) The component that would become negative can become a nonbasis variable while x_i becomes a basis variable. The matrix $\mathbf{A}''$, the value $\mathbf{b}''$ of the new basis variables, and the coefficient vector and constant term of the transformed objective function $\mathbf{c}'' = \mathbf{c}''_{\mathbf{N}}x''_{\mathbf{N}} + c''_0$, can be obtained from $\mathbf{A}'$, $\mathbf{b}'$, $\mathbf{c}'_{\mathbf{N}}$, and c'_0 by conceptually simple (but tedious to write out) algebraic manipulations. This process is called *pivotting*.

The remaining problem is to find an initial feasible basis. For this purpose we consider the artificial problem $\max -\sum_i w_i$ subject to $w + \mathbf{A}x = \mathbf{b}$, $w \geq 0$, and ≥ 0. The artificial variables w with value $\mathbf{b}$ may be taken as the initial feasible basis for this problem. In the case expressing the artificial objective function in terms of the nonbasis variables is somewhat simpler: $w' = \mathbf{b} - \mathbf{A}x'$, so

$$\sum_i w'_i = \sum_i (\mathbf{b}_i - \sum_j \mathbf{a}_{ij}x'_j) = \sum_i \mathbf{b}_i - \sum_j (\sum_i \mathbf{a}_{ij})x'_j.$$

Since we only consider feasible given problems, repeated pivotting leads eventually to the objective function being zero, i.e., $y = 0$. When this happens, the value of x is a basic feasible solution for the given problem. It may happen that some of the components of w are in the basis at this point, but all of the components of x that are not in the basis vanish. To obtain a tableau that is a suitable starting point of the simplex algorithm it is desirable, for any component of w that is in the basis, to pivot to put it outside the basis.

C.3 `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. An important point is that although the the input to simple_GCPS_alloc is a process_scp, as is the case for the mcc and emcc mechanisms, the priorities play no role in the computation. Insofar as this information is copied at various times during the computation, this entails a certain amount of inefficiency, but probably not enough to make a significant difference to running times. The additional infrastructure of a different input type (reading from a file, printing, copying, and otherwise managing) would be a burden, and does not seem worthwhile at this point.

C.4 Computing the GCPS Allocation

C.5 gcps.c

The first function defined in gcpscode.c is simple_GCPS_alloc. This function obtains a partial_alloc from GCPS_allocation, which is a function that, in addition to the the input argument, has integer pointers no_segments, no_splits, no_new_pivots, no_old_pivots, and h_sum. These pointers count the number of times certain events (described below) occur during the computation, and thus provide interesting information concerning the performance of the algorithm. There is commented out code for printing these numbers. There is also commented out code for comparing the results of the computation with the contents of the file allocate.mat. By running this code frequently during (for example) a large scale reorganization of the code, one can quickly determine if a bug was introduced by some step in the process.

No Revisions Below

We now describe how the GCPS allocation is computed. We work with a fixed school choice CEE $E = (I, O, q, g)$ and a profile $\succ = (\succ_i)_{i \in I}$ of strict preferences over O. Let Q be the set of feasible allocations, and let R be the set of possible allocations. The allocation procedure is a piecewise linear function $p\colon [0, 1] \to R$ with $p(0) = 0$, $p(t) \in R \setminus Q$ for all $t < 1$, and $p(1) \in Q$. The *GCPS allocation* is

$$GCPS(E, \succ) = p(1).$$

At each moment the trajectory of p increases, at unit speed, each student's assignment of her favorite school, among those that are still available to her, while leaving other allocations fixed. This direction is adjusted when an student i's assignment of a school j reaches g_{ij}, and when p arrives at one of the facets of R. Suppose that t^* is the first time such that $p(t^*)$ is in a facet of R, so that there is a minimal critical pair (J, P) for the residual CEE, which we denote by $E - p(t^*)$. The GCPS allocation has a recursive definition: for $t \in [t^*, 1]$, $p(t)$ is, by definition, the sum of $p(t^*)$ and the results of applying the allocation procedure to the derived CEE's obtained by restricting $E - p(t^*)$ to J and P and to the complements of J and P, as described earlier.

The main computational challenge is to detect when p arrives at one of the facets of R. During the computation our algorithm computes an auxilary piecewise linear function $\overline{p} \colon [0, 1] \to Q$ such that $p(t) \leq \overline{p}(t)$ for all t. We use the push-relabel algorithm to compute $\overline{p}(0)$.

The combined function $(p, \overline{p})$ is piecewise linear, and $[0, 1]$ is a finite union of intervals $[t_0, t_1], [t_1, t_2], \ldots, [t_{K-1}, t_K]$, where $t_0 = 0$ and $t_K = 1$, such that on each interval $[t_k, t_{k+1}]$ the derivative of $(p, \overline{p})$ is constant. Suppose that we have already computed $p(t_k)$ and $\overline{p}(t_k)$. For each student i we compute the set $\alpha_i(t_k)$ of schools that are still possible for i, and we determine her $\succ_i$-favorite element e_i^k. Let $\theta^k \in \mathbb{Z}^{I \times O}$ be the matrix such that $\theta_{ij}^k = 1$ if $j = e_i^k$, and otherwise $\theta_{ij}^k = 0$.

There are now two possibilities. The first is that for some $t' > t_k$, $p(t_k) + \theta^k(t - t_k) \in R$ for all $t \in [t_k, t']$. In this case we will find a $\theta \in \mathbb{Z}^{I \times O}$ such that for some $t' > t_k$,

$$\overline{p}(t_k) + \theta(t - t_k) \in Q \quad \text{and} \quad p(t_k) + \theta^k(t - t_k) \leq \overline{p}(t_k) + \theta(t - t_k). \tag{$*$}$$

and all $t \in [t_k, t']$. Now t_{k+1} is the first time after t_k such that one or more of the following holds:

a) $p_{ie_i^k}(t_{k+1}) = g_{ie_i^k}$ for some i;

b) $\overline{p}(t_k) + \theta(t - t_k) \notin Q$ for $t > t_{k+1}$;

c) for $t > t_{k+1}$ it is not the case that $p(t_k) + \theta^k(t - t_k) \leq \overline{p}(t_k) + \theta(t - t_k)$.

For $t \in [t_k, t_{k+1}]$ we have determined that $p(t) = p(t_k) + \theta^k(t - t_k)$, and we set $\overline{p}(t) = \overline{p}(t_k) + \theta(t - t_k)$. Having determined $p(t_{k+1})$ and $\overline{p}(t_{k+1})$, we can repeat the process.

The second possibility is that it is not possible to continue p, as described above, without leaving R, because there is a critical pair (J, P) for the residual economy at time t_k. In this case we find such a pair, then descend recursively to the computation of the GCPS allocations of derived subeconomies.

We now describe an algorithm that determines which of these possibilities holds, In the first case it finds a satisfactory θ, and in the second case it finds a critical pair (J, P). Suppose that there is a $\theta \in \mathbb{Z}^{I \times O}$ such that for some $t' > t_k$, $(*)$ holds for all $t \in [t_k, t']$. The two conditions in $(*)$ together imply that $p(t_k) + \theta^k(t - t_k) \in R$ for all $t \in [t_k, t']$, so the first possibility above holds, and we can use θ to define the continuation of $\overline{p}$. The algorithm may be thought of as a search for such a θ.

For a given θ, a $t' > 0$ as above exists if and only if θ satisfies the following conditions:

(a) For each i and j:

(i) If $o \notin \alpha_i$, then $\theta_{ij} = 0$.

(ii) If $\overline{p}_{ij}(t_k) = p_{ij}(t_k)$, then $\theta_{ij} \geq 0$, and if, in addition, $o = e_i^k$, then $\theta_{ij} \geq 1$.

(iii) If $\overline{p}_{ij}(t_k) = g_{ij}$, then $\theta_{ij} \leq 0$.

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

(c) For each j, if $\sum_i \overline{p}_{ij}(t_k) = q_j$, then $\sum_i \theta_{ij} \leq 0$.

Our search for a suitable θ begins by defining an initial $\theta \in \mathbb{Z}^{I \times O}$ as follows. For each i, if $\overline{p}_{ie_i^k}(t_k) > p_{ie_i^k}(t_k)$, then we set $\theta_{ij} = 0$ for all j. If $\overline{p}_{ie_i^k} = p_{ie_i^k}(t_k)$, then we set $\theta_{ie_i^k} = 1$, we set $\theta_{ij_i} = -1$ for some $j_i \neq e_i^k$ such that $\overline{p}_{ij_i}(t_k) > p_{ij_i}(t_k)$, and we set $\theta_{ij} = 0$ for all other j. Evidently θ satisfies (a) and (b).

Let

$$\tilde{P} = \{\, j : \sum_i \overline{p}_{ij}(t_k) = q_j \text{ and } \sum_i \theta_{ij} > 0 \,\}.$$

If $\sum_{j \in \tilde{P}} \sum_i \theta_{ij} \leq 0$, then (c) holds. Suppose that this is not the case. We now describe a construction that may or may not be possible. When it is possible, it passes from θ to a $\theta' \in \mathbb{Z}^{I \times O}$ satisfying (a) and (b) such that if $\tilde{P}' = \{\, j : \sum_i \overline{p}_{ij}(t_k) = q_j \text{ and } \sum_i \theta'_{ij} > 0 \,\}$, then

$$\sum_{j \in \tilde{P}'} \sum_i \theta'_{ij} = \sum_{j \in \tilde{P}} \sum_i \theta_{ij} - 1. \tag{$**$}$$

Repeating this construction will eventually produce an element of $\mathbb{Z}^{I \times O}$ satisfying (a)–(c) unless, at some point, the construction becomes impossible.

Choose $j_0 \in \tilde{P}$. We wish to find an integer $h \geq 1$, distinct $i_1, \ldots, i_h$, and $j_1, \ldots, j_h$ such that $j_0, j_1, \ldots, j_h$ are distinct, such that if we define θ' by setting

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

for $g = 1, \ldots, h$ and $\theta'_{ij} = \theta_{ij}$ for all other (i, j), then θ' satisfies (a), (b), and ($**$). Evidently θ' satisfies (i) if $j_{g-1}, j_g \in \alpha_{i_g}$ for all $g = 1, \ldots, h$, it satisfies (ii) if, for each $g = 1, \ldots, h$, if

$\overline{p}_{i_g j_{g-1}}(t_k) = p_{i_g j_{g-1}}(t_k)$, then $\theta_{i_g j_{g-1}} > 0$ and $\theta_{i_g j_{g-1}} > 1$ if $j_{g-1} = e^{\succ}_{i_g}$, and it satisfies (iii) if, for each $g = 1, \ldots, h$, either $g_{i_g j_g} < \overline{p}_{i_g j_g}(t_k)$ or $\theta_{i_g j_g} < 0$. Clearly θ' satisfies (b) if θ satisfies (b). We have $\sum_i \theta'_{ij_0} = \sum_i \theta_{ij_0} - 1$, $\sum_i \theta'_{ij_g} = \sum_i \theta_{ij_g}$ for $g = 1, \ldots, h-1$, and $\sum_i \theta'_{ij_h} = \sum_i \theta_{ij_h} + 1$, so θ' satisfies $(**)$ if $\sum_i \overline{p}_{ij_h}(t_k) < q_{j_h}$ or $\sum_i \theta_{ij_h} < 0$.

We now describe the search for $j_0, j_1, \ldots, j_h$ and $i_1, \ldots, i_h$. For $i \in I$ let

$$P(i) = \{\, j \in \alpha_i : \theta_{ij} < 0 \text{ if } \overline{p}_{ij}(t_k) = g_{ij} \,\}.$$

For $j \in O$ let

$$J(j) = \{\, i : j \in \alpha_i \text{ and if } \overline{p}_{ij}(t_k) = p_{ij}(t_k), \text{ then } \theta_{ij} > 0 \text{ and } \theta_{ij} > 1 \text{ if } j = e_i \,\}.$$

We define sets $P_0, J_1, P_1, J_2, \ldots$ inductively, beginning with $P_0 = \{j_0\}$. If P_{g-1} has already been computed, we set $J_g = \bigcup_{j \in P_{g-1}} J(j)$. If J_g has already been computed, we set $P_g = \bigcup_{i \in J_g} P(i)$. We continue this construction until we arrive at an h such that either $P_h = P_{h-1}$ or there is a $j_h \in P_h$ such that $\sum_i \overline{p}_{ij_h}(t_k) < q_{j_h}$ or $\sum_i \theta_{ij_h} < 0$.

If there is such a j_h, we can find an $i_h \in J_h$ such that $j_h \in P(i_h)$, then find an $j_{h-1} \in P_{h-1}$ such that $i_h \in J(j_{h-1})$, then find an $i_{h-1} \in J_{h-1}$ such that $j_{h-1} \in P(i_{h-1})$, and so forth. Continuing in this fashion produces $j_0, j_1, \ldots, j_h$ and $i_1, \ldots, i_h$ as above. Note that j_h cannot be an element of P_{h-1} because the process would have terminated sooner. Similarly i_h is not an element of J_{h-1}, $j_{h-1} \notin P_{h-1}$, and so forth. Therefore $j_0, j_1, \ldots, j_h$ are distinct and $i_1, \ldots, i_h$ are distinct.

Now suppose that the construction terminates with $P_h = P_{h-1}$. Let $J = \bigcup_h J_h$ and $P = \bigcup_h P_h$. We have $\sum_i \overline{p}_{ij}(t_k) = q_j$ for all $j \in P$. If $j \in P$ and $i \notin J$, then $i \notin J(j)$, so $\overline{p}_{ij}(t_k) = p_{ij}(t_k)$. If $i \in J$ and $j \notin P$, then $j \notin P(i)$, so $\overline{p}_{ij}(t_k) = g_{ij}$ if $j \in \alpha_i \setminus P$, and $\overline{p}_{ij}(t_k) = g_{ij} = 0$ if $j \notin \alpha_i$. Thus $\overline{p}(t_k) - p(t_k)$ is a feasible allocation for $E - p(t_k)$ that gives all of the resources in P to students in J, and it gives $g_{ij} - p_{ij}(t_k)$ 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 $E - p(t_k)$.

Summarizing, the algorithm repeatedly extends p and $\overline{p}$ to intervals such as $[t_k, t_{k+1}]$ until $p(t_k)$ satisfies the GMC equality for a pair (J, P), at which point it descends recursively. If $p(t_k)$ does not satisfy such a GMC inequality, it finds a θ satisfying (a)–(c) by beginning with a θ that satisfies (a) and (b) and repeatedly adjusting it until it also satisfies (c).

C.6 `push_relabel.h` and `push_relabel.c`

The push-relabel algorithm of Goldberg and Tarjan (1988) is implemented in `push_relabel.h` and `push_relabel.c`. The code straightforwardly follows the description of the algorithm in Appendix C.8, and the best way to approach it is to read that description, then examine the code.

C.7 Push-Relabel

This subsection describes the push-relabel algorithm of Goldberg and Tarjan (1988). We first describe the general setting of networks and flows, then we describe the algorithm, and in the next subsection we describe the specialized setting in which it is applied in our software.

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

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

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

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

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

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

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

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

so we may define *value* of f to be

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

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

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

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

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

Let $f \colon N \times N \to \mathbb{R}$ be a preflow that is bounded by c. The *excess* of f at n is

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

Of course $e_f(n) \ge 0$, and f is a flow if and only if $e_f(n) = 0$ for all $n \in N \setminus \{s, t\}$. The *residual capacity* of (n, n') is

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

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

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

The algorithm is initialized by setting $d(s) = |N|$, $d(n) = 0$ for all other n, $f(s, n) = c(s, n)$ for all n such that $(s, n) \in A$, and setting $f(n, n') = 0$ for all other n and n'. The algorithm then consists of repeatedly applying the following two *elementary operations*, in any order, until there is no longer any valid application of them:

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

(b) $\mathrm{Relabel}(n)$ is valid if n is active and $d(n) \le d(n')$ for all n' such that $(n, n') \in A_f$. The operation resets $d(n)$ to ∞ if there is no n' such that $(n, n') \in A_f$ (this never actually happens) and otherwise it resets $d(n)$ to $1 + \min_{n' : (n,n') \in A_f} d(n')$.

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

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

C.8 School Choice Communal Endowment Economies

A *school choice communal endowment economy* (CEE) is a quadruple $E = (I, O, q, g)$ in which I is a nonempty finite set of *students*, O is a nonempty finite set of *schools*, $q \in \mathbb{R}^O_+$, and $g \in \mathbb{R}^{I \times O}_+$. For $i \in I$ and $j \in O$ we say that q_j is the *quota* of j, and that g_{ij} is *i's j-max*.

We apply the push-relabel algorithm to a particular directed graph (N_E, A_E) in which the set of nodes is

$$N_E = \{s\} \cup I \cup O \cup \{t\}.$$

For $i \in I$ and $j \in O$ let $a_i = (s, i)$, $a_{ij} = (i, j)$, and $a_j = (j, t)$, and let

$$A_E = \{\, a_i : i \in I \,\} \cup \{\, a_{ij} : i \in I, j \in O \,\} \cup \{\, a_j : j \in O \,\}.$$

Let c_E be the capacity in which $c_E(a_i) = 1$ for all i, $c_E(a_{ij}) = g_{ij}$ for all i and j, and $c_E(a_j) = q_j$ for all j. It turns out that when the push-relabel algorithm is applied to a graph of this form, it is possible to speed it up by initializing d by setting $d(s) = 2|O| + 2$ and $d(n) = 0$ for all other n. Roughly (this is not the place to explain the details) this works because $2|O| + 2$ is an upper bound on the number of nodes on a simple (no repeating nodes) path from s to t when $|O| \leq |I|$.

An *allocation* for E is a matrix $p \in \mathbb{R}^{I \times O}_+$. A *partial allocation* for E is an allocation p such that $\sum_j p_{ij} \leq 1$ for all i, $\sum_i p_{ij} \leq q_j$ for all j, and $p_{ij} \leq g_{ij}$ for all i and j. A *feasible allocation* is a partial allocation m such that $\sum_j m_{ij} = 1$ for all i. A *possible allocation* is an allocation p such that there is a feasible allocation m such that $p \leq m$.

If p is an allocation, there is a unique flow f_p such that $f_p(a_{ij}) = p_{ij}$ for all i and j that has $f_p(a_i) = \sum_j p_{ij}$ for all i and $f_p(a_j) = \sum_i p_{ij}$ for all j. Evidently p is a feasible allocation if and only if f_p is bounded by c_E and $f_p(a_i) = 1$ for all i, which is the case if and only if $|f_p| = |I|$. Conversely, if f is a flow bounded by c_E with $|f| = |I|$ and thus $f(a_i) = 1$ for all i, then setting $p_{ij} = f(a_{ij})$ gives a feasible allocation p. Thus there is a feasible allocation if and only if the maximum value of a flow bounded by c_E is $|I|$. Although our primary use of the push-relabel algorithm is to compute a feasible allocation, it also provides an efficient method of determining whether a feasible allocation exists.

D Junk

D.1 `gcps_solver.h` and `gcps_solver.c`

The files `gcps_solver.h` and `gcps_solver.c` implement the algorithm for computing the GCPS allocation. This algorithm is also described in Appendix C.8, and again the best way to approach it is examine the code only after the description has been read and understood. In this case it is best to work backwards from the ends of `gcps_solver.h` and `gcps_solver.c`, because that follows a top-down understanding of how the algorithm is implemented.

D.2 `solve.c` and `purify.c`

The files `solve.c` and `purify.c` contain the `main` functions of the executables `gcps` and `purify` respectively. These `main` functions are mostly simple and straightforward.

In `solve.c` there are `int*` variables `segments`, `splits`, `pivots`, and `h_sum`. These are, respectively, the number of linear segments of the piecewise linear function $(p, \overline{p})$ described in Appendix C.8, the number of time the algorithm descends recursively to two derived subproblems, the number of times that the algorithm modifies θ, as described iin Appendix C.8, and the sum of the indices h that arise in paths $i_0, j_1, i_1, \ldots, i_h, j_h$ that are used to pivot. These provide interesting information about the algorithm's performance, and there is a sample print statement below for them.

References

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

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

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

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

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

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

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

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

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

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