GCPS Schools: A User's Guide

Andrew McLennan*

August 8, 2023

Abstract

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

1 Introduction

In the paper "An Efficient School Choice Mechanism Based on a Generalization of Hall's Marriage Theorem" (joint with Shino Takayama and Yuki Tamura) we describe a new algorithm for school choice, along with its theoretical foundations. This algorithm has been implemented (using the C programming language) in the software package *GCPS Schools* as an executable `gcps`, which passes from a school choice problem (as described below) to a matrix specifying, for each student-school pair, the probability that the student is assigned to the school. The software package also contains two other executables `purify` and `make_ex`. The first of these passes from a matrix of assignment probabilities to a random pure assignment whose probability distribution averages to the given matrix of probabilities. The second program generates example school choice problems of the sort that might occur in large school districts. These programs provide the basic computational resources required to apply our mechanism, and perhaps in some cases they will suffice. However, our primary hope is that the underlying code will be a useful starting point for further software development.

This document describes these programs, from the point of view of a user. It doesn't assume that the reader has already read our paper, but of course we are leaving out lots of relevant information. Instructions for downloading and setting up the software, and doing a test run, are

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

given in Appendix A. It would probably be a good idea to follow those instructions now, or before reading too far into this guide, but the main body of this guide does not assume that you have done so. The body also does not assume that the reader knows the C programming language, but some language features become relevant in Appendix B.

2 gcps

To begin with we describe a simple example of an input file, which the application `gcps` expects to find in a file called `schools.scp` in the current directory. (If there is no such file `gcps` simply complains and quits.)

```
/* This is a sample introductory comment.  */
There are 4 students and 3 schools
The vector of quotas is (1,2,1)
The priority matrix is
1 1 1
1 0 1
1 1 1
1 1 1
The students numbers of ranked schools are (3,2,3,3)
The preferences of the students are
1:  1 2 3
2:  1 3
3:  1 2 3
4:  1 2 3
The priority thresholds of the schools are
1 1 1
```

Our input files begin with a comment between `/*` and `*/`. This is purely for your convenience, and 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 will only start extracting information after it sees the `*/`. The computer divides the remainder into "generalized white space" (in addition to spaces, tabs, and new lines, '`(`', '`)`', and

`,` are treated as white space) and "tokens," which are sequences of characters without any of the generalized white space characters. Tokens are either prescribed words, numbers, or student tags (a student number followed by `:`) and 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`.

The next line gives the quotas (i.e., the capacities) of the schools, so school 2 has two seats, and the other two schools each have one seat. Here we see the convenience of making `(`, `)`, and `,` white space characters: otherwise we would have had to write `The vector of quotas is 1 2 1`.

Our treatment of priorities is somewhat different from what is typical in the school choice literature, where the priority is thought of as the "utility" the school gets from a student, and is often required to come from a strict ranking of the students. At this stage a student's priority at a school is either 1 if she is allowed to attend the school, and may be assigned a seat there, and otherwise it is 0. (We'll talk about more complicated priorities later.) A student's priority at a school may be 0 because she is not qualified (it is a single sex school for boys, or her test scores are too low) or it may be 0 because the student prefers a seat at her "safe school" (as we explain below) and can insist on receiving a seat at a school that she likes at least as much.

The next line provides information (for each student, the number of schools for which she has priority 1) 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. Finally, there are the schools' minimum priorities for admission, which in this context are all 1: a student is good enough to admit to a school if her priority for that school is 1 and not otherwise. (Again, we'll talk about more complex situations later.) The collection of information provided by an input file is a *school choice problem*.

What the software does (primarily) is compute a matrix of assignment probabilities. For our particular example `gcps` gives the following output. (Here and below we are leaving out the sample comment.)

```
The allocation is:
     1:   2:   3:
1:  0.25 0.67 0.08
2:  0.25 0.00 0.75
```

```
3:  0.25 0.67 0.08
4:  0.25 0.67 0.08
```

Note that the sum of the entries in each row is 1 and the sum of the entries in each school's column is that school's quota. In general the sum of the quotas may exceed the number of students, in which case we require that the sum of the entries in each school's column does not exceed that school's quota. An assignment of probabilities with these properties — each student's total assignment is 1 and no school is overassigned — is a *feasible allocation*.

Our mechanism is based on the "simultaneous eating" algorithm of Bogomolnaia and Moulin (2001) for probabilistic allocation of objects, as generalized by Balbuzanov (2022). In our example each student consumes probability of a seat in her favorite school (school 1) until that resource is exhausted at time 0.25, at which point each student switches to the next best thing. This continues until school 2 is also exhausted, after which all finish up by consuming probability of a seat in school 3.

This makes good sense if the schools simply fill up one by one, as in this example, but is that always what happens? Actually, no. To help understand this we first introduce a new concept, the "safe school." The idea is that each student has one school, say the closest school or the school that a sibling attends, to which she is guaranteed admission if she insists. Each student submits a ranking of the schools that she is eligible for and (weakly) prefers to her safe school, and her priority is 1 at those schools and 0 at all others.

Now suppose that there are two schools, say 1 and 2, that are quite popular. Some students have school 1 as their safe school, but prefer 2, and some students have school 2 as their safe school, but prefer 1. There are also some students who have other safe schools, but prefer either 1 or 2, or both. As the students consume probability at their favorite schools, there can come a time at which schools 1 and 2 together only have enough remaining capacity to serve the students who can insist on going to one of these two schools, even though school 1 still has excess capacity if it can ignore the students who have 1 as their safe school but prefer 2 and the students who have 2 as their safe school but prefer 1, and similarly for school 2. When this happens we say that the set of schools $P = \{1, 2\}$ has become *critical*.

At this time further consumption of capacity at schools 1 and 2 is restricted to those students who cannot be assigned to other schools, so further consumption of these schools is denied to students who do not have 1 or 2 as their safe school, and also to students who have 1 or 2 as their safe school but prefer some third school that is still available. For each of the latter students the least preferred of the schools she is willing to attend that is still available becomes the new safe school.

More generally, let P be a set of schools, and let J_P be the set of students whose priorities for all schools outside of P are 0. For any $i \in J_P$, a feasible allocation must assign probability 1 to student i receiving a seat in P, so a necessary condition for the existence of a feasible allocation is that the total capacity of the schools in P is not less than the number of students in J_P. We call this condition the *GMC* (generalized marriage condition) *for* P. In fact this condition is sufficient for the existence of a feasible allocation: *if, for each set of schools* P*, the GMC inequality for* P *holds, then a feasible allocation exists.* This is not an obvious or trivial result, and a somewhat more general version of it is one of the main points of our paper. This result extends to situations where the resources have already been partially allocated: *if, for each set of schools* P*, the total remaining capacity of the schools in* P *is not less than the total remaining demand of students in* J_P (where this set is defined in relation to the students' current safe schools) *then there is an allocation of the remaining resources that gives a feasible allocation.*

We can now describe the algorithm in a bit more detail. At each time each student is consuming probability of a seat at the favorite school among those that are still available to her. This continues until the first time that there is a set of schools P such that the remaining capacity is just sufficient to meet the needs of the students in the set J_P of students who no longer have access to any schools outside of P. At this point the problem divides into two subproblems, one corresponding to the sets P and J_P and the other corresponding to the complements of these sets. These problems have the same form as the original problem, and can be treated algorithmically in the same way, so the algorithm can descend recursively to smaller and smaller subproblems until a feasible allocation has been fully computed.

3 `purify`

Leaving out the initial comment, the output of `gcps` is a matrix of assignment probabilities, as shown in the example below. (Now, to minimize confusion, the schools are A, B, and C.)

```
      A:   B:   C:
1:  0.25 0.67 0.08
2:  0.25 0.00 0.75
```

```
3:   0.25 0.67 0.08
4:   0.25 0.67 0.08
```

Generating a random deterministic assignment with a probability distribution that averages to this matrix is called *implementation* by Budish et al. (2013). The executable `purify` reads a feasible matrix m of assignment probabilities from a file `allocate.mat`, which must have the form of the output of `gcps`, and it produces a random deterministic allocation with a suitable distribution, using an algorithm of Budish et al. (2013), as it applies to our somewhat simpler framework.

We can illustrate the algorithm using the feasible allocation shown above. We consider a cyclic path alternating between students and schools, say $1 \to C \to 3 \to A \to 1$, such that the entries of the matrix for $(1, C)$, $(3, C)$, $(3, A)$, and $(1, A)$ are all strictly between 0 and 1. If we add 0.08 to the entries for $(1, C)$ and $(3, A)$ while subtracting 0.08 from the entries for $(3, C)$ and $(1, A)$, we obtain

```
      A:   B:   C:
1:   0.17 0.67 0.17
2:   0.25 0.00 0.75
3:   0.33 0.67 0.00
4:   0.25 0.67 0.08
```

(Recall that 0.08, 0.17, and 0.33 are really $\frac{1}{12}$, $\frac{1}{6}$ and $\frac{1}{3}$.) This is also a feasible allocation. We could also subtract 0.08 from the entries for $(1, C)$ and $(3, A)$ while adding 0.08 the entries for $(3, C)$ and $(1, A)$, thereby obtaining the feasible allocation

```
      A:   B:   C:
1:   0.33 0.67 0.00
2:   0.25 0.00 0.75
3:   0.17 0.67 0.17
4:   0.25 0.67 0.08
```

Note that each of these matrices has one more zero than the original matrix.

The computer chooses between these two matrices by flipping a coin. If heads, it then generates a random pure allocation that averages to the first matrix, and if tails, then it produces a random pure allocation that averages to the second matrix. The average of the overall distribution of pure allocations is the matrix we started with.

We now give a more formal explanation of the algorithm. There is a directed graph whose nodes are the students, the schools, and a *sink*. The graph has an arc from each student to each school, and an arc from each school to the sink. A *flow* is an assignment of a positive number to each arc such that for each student, the sum of the flows to all schools is 1, and for each school the sum of the flows from all students is equal to the flow from that school to the sink. A matrix of assignment probabilities m has an associated flow f in which the flow from each student to each school is the probability that the student receives a seat in the school, and the flow from the school to the sink is the sum of the school's assignment probabilities.

There is a subgraph consisting of all arcs whose flows are not integers. A key point is that for any node that is an endpoint of one of the arcs in the subgraph, there is another arc in the subgraph that also has that node as an endpoint. For each student, this is obvious because the sum of the student's assignment probabilities is one. If the sum of the flows into a school is an integer, and one of these flows is not an integer, then there must be another flow into the school that is not an integer. If the sum of the flows into a school is not an integer, then one of the flows into the school is not an integer, and the flow from the school to the sink is not an integer. The sum of the flows into the sink is the sum of the flows out of the students, which is the number of students, hence an integer, so if the flow from one of the schools to the sink is not an integer, there must be another such school.

Consequently the subgraph has a *cycle*, which is a sequence of distinct nodes $n_1, \ldots, n_k$ such that $k > 2$ and, for each $i = 1, \ldots, k$, n_i and n_{i+1} are the endpoints of an arc in the subgraph. (We are treating the indices as integers mod k, so $k+1 = 1$.) The algorithm for finding a cycle (whose correctness is the proof of the existence of a cycle) works in an obvious manner. Beginning with n_1 and n_2 that are the endpoints of an arc in the subgraph, it finds $n_3 \neq n_1$ such that n_2 and n_3 are the endpoints of an arc in the subgraph. In general, after finding n_i such that n_{i-1} and n_i are the endpoints of an arc in the subgraph, the algorithm asks whether there is $j = 1, \ldots, i-2$ such that $n_i = n_j$, in which case $n_j, \ldots, n_{i-1}$ is the desired cycle, and otherwise it finds $n_{i+1} \neq n_{i-1}$ such that n_i and n_{i+1} are the endpoints of an arc in the subgraph. Since there are finitely many nodes, the process must eventually halt.

Given a cycle $n_1, \ldots, n_k$, for each $i = 1, \ldots, k$ we say that $n_i n_{i+1}$ is a *forward arc* if n_i is a student and n_{i+1} is a school, or if n_i is a school and n_{i+1} is the sink, and otherwise we say that $n_{i+1} n_i$ is a *backward arc*. For any real number δ, if we modify f by adding a constant δ to the flow of each forward arc while subtracting δ from the flow of each backward arc, the result f^δ is a new flow, because for each student the sum of outward flows is 1, and for each school the sum of flows from students to the school is the flow from the school to the sink.

Let α be the smallest positive number such that f^α has at least one more integer entry than f, and let β be the smallest positive number such that $f^{-\beta}$ has at least one more integer entry than f. Then $f = \frac{\beta}{\alpha+\beta} f^\alpha + \frac{\alpha}{\alpha+\beta} f^{-\beta}$. Let m^α and $m^{-\beta}$ be the restrictions of f^α and $f^{-\beta}$ to the arcs from students to schools. It is easy to see that m^α and $m^{-\beta}$ are feasible allocations: their entries lie in $[0, 1]$, and the sums of the entries for each student and each school are the corresponding sums for m.

The algorithm passes from m and f to m^α and f^α with probability $\frac{\beta}{\alpha+\beta}$, and to $m^{-\beta}$ and $f^{-\beta}$ with probability $\frac{\alpha}{\alpha+\beta}$. Whichever of m^α and $m^{-\beta}$ is chosen, if it is not a deterministic assignment, then the process is repeated. We claim that the the average of the resulting distribution of pure allocations is m. Since $m = \frac{\beta}{\alpha+\beta} m^\alpha + \frac{\alpha}{\alpha+\beta} m^{-\beta}$, this is clear if m^α and $m^{-\beta}$ are pure allocations, and in general it follows from induction on the number of nonintegral entries of m.

The code for the algorithm described above is in `implement.c`, which has the associated header file `implement.h`. The file `purify.c` contains a high level sequence of commands that execute the algorithm.

4 make_ex

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

One of the files produced by `make_ex` begins as follows:

```
/* This file was generated by make_ex with 20 schools,
4 students per school, capacity 5 for all schools,
school valence standard deviation 1.00,
and idiosyncratic standard deviation 1.00.  */
```

In this example there are 20 schools that are spaced evenly around a circle of circumference 20. Since there are 4 students per school, there are 80 students. Their homes are also spaced evenly around the circle. Each student's safe school is the school closest to her home. A student's utility for a school is the sum of the school's valence and an idiosyncratic shock, minus the distance from the student's home to the school. Each school's valence is a normally distributed random variable with mean 0.0 and standard deviation 1.0, and for each student-school pair the idiosyncratic shock is a normally distributed random variable with mean 0.0 and standard deviation 1.0. All of these random variables are independent. The program passes from the utilities to an input for `gcps` by finding the ranking, for each student, of the schools for which the student's utility is at least as large as the utility of the safe school.

Near the beginning of the file `example.c` there are the following lines:

```
int no_schools = 20;
int no_students_per_school = 4;
int school_capacity = 5;
double school_valence_std_dev = 1.0;
double idiosyncratic_std_dev = 1.0;
```

Even for someone who knows nothing about the C programming language, this is pretty easy to understand. The keywords `int` and `double` are data types for integers and floating point numbers. Thus `no_schools`, `no_students_per_school`, and `school_capacity` are integers, while `school_valence_std_dev` and `idiosyncratic_std_dev` are floating point numbers. Each line assigns a value to some variable. If you would like to generate examples with different parameters, the way to do that is to first change the parameters by editing `example.c`, then run `make` to compile `make_ex` with the new parameters, and finally issue a command like `make_ex > my_file.scp` which runs `make_ex` and redirects the output to the file `my_file.scp`. For example, to diminish the relative importance of travel costs one can increase `school_valence_std_dev` and `idiosyncratic_std_dev`.

This illustrates an important point concerning the relationship between this software and its users. Most softwares 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, the main purpose 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.

5 Finer Priorities

To appreciate the issue discussed in this section one needs to understand some of the history of other school choice mechanisms. Instead of matching students to seats in schools, it is perhaps more intuitive to consider matching a finite set of boys with a finite set of girls, who each have strict preferences over potential partners and remaining single.

The boy-proposes version of the famous deferred acceptance algorithm begins with each boy proposing to his favorite girl, if there is one he prefers to being alone. Each girl rejects all proposals that are less attractive than being alone, and if she has received more than one acceptable proposal, she holds on to her favorite and rejects all the others. In each subsequent round, each boy who was rejected in the previous round proposes to his favorite among the girls who have not yet rejected him, if one of these is acceptable. Each girl now has a number of new proposals, and possibly the proposal she brought forward from the previous round. She retains her favorite of these, if it is acceptable, rejecting all others. This procedure is repeated until there is a round with no rejections, at which point each girl holding a proposal pairs up with the boy whose proposal she is holding. This mechanism was first proposed in the academic literature by Gale and Shapley (1962), but it turned out that it had already been used for several years to match new graduates of medical schools with residencies. For almost twenty years it has been used in school matching, with the students proposing and the seats in the various schools rejecting, and it is now in widespread use around the world.

The key point for us is that this mechanism is not well defined unless both sides have strict preferences. In the context of school matching, the schools' preferences are called *priorities*. If these priorities are not actual reflections of society's values, this can result in inefficiency. For example, if Carol School's priorities rank Bob above Ted while Alice School's priorities rank Ted above Bob, then we could have an assignment in which Bob envies Ted's seat at Alice School while Ted envies Bob's seat at Carol School. This sort of inefficiency can be quantitatively important, and a major advantage of our mechanism is that it is efficient, in an even stronger sense than not allowing outcomes in which improving trades are possible.

However, there are cases in which the schools' priorities do reflect actual values. In China, for example, each student's priority at all schools is the score on a standardized test. A consequence of this, under deferred acceptance, is that, in effect, each school has an exam score cutoff, accepting all students above the cutoff, rejecting all students below the cutoff, and randomizing (roughly speaking) over students right at the cutoff. Our main concern in this section is to explain how our mechanism can achieve similar outcomes.

The first point is that our input files can have a richer structure than our original example suggests, as illustrated by the input below. The priorities can be arbitrary nonnegative integers. A student having a priority of 0 at a particular school is understood as indicating that the student cannot be assigned there, either because she is not qualified or because she prefers her safe school. A student's safe school can be indicated by giving the student the highest possible priority at that school. The computer passes from this input to a school choice problem in which the priority of a student at a school is 1 if her priority in the input is not less than the school's priority threshold, and it is 0 otherwise, each school's priority threshold is set to 1, and each student's preference is truncated by eliminating schools she is not eligible for. Applying this procedure to the input below gives our original example.

```
/* This is a sample introductory comment.  */
There are 4 students and 3 schools
The vector of quotas is (1,2,1)
The priority matrix is
5 6 9
2 2 9
5 4 9
3 4 9
The students numbers of ranked schools are (3,3,3,3)
The preferences of the students are
1:  1 2 3
2:  1 2 3
3:  1 2 3
4:  1 2 3
The priority thresholds of the schools are
1 3 5
```

It is possible to repeatedly adjust the schools' priority thresholds to achieve a desired effect. For example, suppose there are two selective schools, and the school district would like it to be the case that a well qualified student is almost certain to receive a seat in one of them if that is what she wants, and at the same time these schools do not have more than a small amount of unused capacity. One may raise the priority threshold of one of these schools if many students are receiving some probability of admission and lower the threshold if its seats are not being filled. Of course changing the priority threshold at one of the schools will effect demand for the other school, so repeated adjustment of the priority thresholds of all the schools may be required to achieve a desirable result. (Automating this iterative adjustment process may require the development of a version of `gcps` that can accept parameter inputs, without editing the source code. This should be a simple task for an experienced C programmer.)

Whether it is a good idea to use priorities as the Chinese do is an extremely complex question. On the one hand there is an obvious sense in which it is desirable to provide the best resources to those who can extract the greatest benefit. On the other side, the Chinese system intensifies the intergenerational transmission of advantage, and there is some education research suggesting that average students benefit from having talented peers while talented students are not disadvantaged by having some peers who are ordinary. One could easily list numerous additional issues. Balancing various concerns in practice requires information concerning what would actually happen under various policy alternatives. Our mechanism provides a wide range of alternatives, for which outcomes from existing data can be easily computed.

6 What If There Are Many Schools?

As the algorithm was described above, it looked ahead, for each nonempty set of schools P, to determine the time at which it would become necessary to restrict further consumption of schools in P to students in J_P. This is not unduly burdensome if there are a moderate number of schools. (For a "toy" example with 20 schools, hence over one million sets of schools, this form of the algorithm finishes in about 10 seconds.) But some school choice problems have several dozen or even hundreds of schools, and will overwhelm the naive version of the algorithm described above. There are several things that can be done about this.

First, to help things along a bit, at each stage of the allocation process the computer looks for schools whose capacity will not be exceeded even if every student who ranks it receives all of their remaining probability from the school. Such a school is said to be *unpopular*. An unpopular school will never be an element of a minimal critical set. A school is *popular* if it is not unpopular

Two schools are *related* if there is a student who can attend either one. For each student i let α_i be the set of schools at which i has priority one. Formally, two schools are related if there is an i such that both of the schools are elements of α_i. At each stage of the allocation process the computer computes a square matrix `related` whose rows and columns are indexed by the schools, such that `related[j][k]` $= 1$ if j and k are both popular at that stage and $\alpha_j \cap \alpha_k \neq \emptyset$, or if $j = k$ and j is popular, and otherwise `related[j][k]` $= 0$. We think of `related` as encoding an undirected graph whose nodes are the popular schools, with an edge connecting j and k if and only if `related[j][k]` $= 1 =$ `related[k][j]`. For any set of schools P there is an induced subgraph whose set of nodes is P and whose edges are the edges of the graph whose endpoints are both in P.

An undirected graph is *connected* if, for any pair of nodes j and k, there is a sequence of edges leading from j to k. Suppose that P is a set of schools such that the induced subgraph of `related` is not connected. Then $P = P_1 \cup P_2$ where P_1 and P_2 are nonempty, $P_1 \cup P_2 = \emptyset$, and there is no path of edges leading from a school in P_1 to a school in P_2. In particular, there is no school in P_1 that is `related` to a school in P_2. If P is critical, then both P_1 and P_2 are critical, because every student in J_P is either in J_{P_1} or in J_{P_2}, so there are just enough remaining seats in P_1 to meet the remaining needs of students in J_{P_1}, and similarly for P_2. Therefore P cannot be a *minimal* critical set.

At this point we have seen that in a search for a minimal critical set, it is only necessary to consider subsets P of the set of popular schools such that the graph obtained by restricting `related` to P is connected. For large school choice problems the number of such sets can still be overwhelming.

There is probably no algorithm that is guaranteed to compute the desired allocation, in a reasonable amount of time, for every possible problem, because it is certainly possible to concoct a problem for which there is a very large critical set. What I have tried to do is to design `gcps` so that it will perform reasonably well, with very high probability, when its input is the sort of problem that comes from `make_ex`, which we hope provides a satisfactory approximation of the sorts of problems that might arise in practice.

Concretely, `gcps` begins by letting the schools fill up one by one, until it notices that some school j has been overallocated, insofar as it does not have enough remaining capacity to serve those students who no longer have any feasible alternatives. Since the process has arrived at an infeasible allocation, there must have been a point in the process where, for some set of schools P, the total remaining capacity of the schools in P went below the total remaining demand of students who could no longer consume any school outside of P.

We now restart the allocation process, this time watching out for criticality of sets of popular schools that one or two elements. If such a set becomes critical, the process splits into two parts, one for the set and the students who can no longer consume schools outside this set, and the other for the other remaining schools and the other students.

The process might fail again because, even with more sets of schools being monitored, there still comes a time at which one of the sets we are monitoring does not have enough remaining capacity to meet the needs of the students who cannot consume schools outside this set. That is, looking at two element sets of schools did not resolve the issue. In this event the next iteration of the allocation process watches out for criticality of sets of schools with three or fewer elements that are `related`-connected. If such a set becomes critical during this iteration, the first such set to become critical is added to a list of sets whose criticality is monitored in all future iterations of the allocation process.

In general, we maintain a list of sets of schools that have proven relevant to the allocation process. In each iteration of the process we monitor the criticality of `related`-connected sets of schools that consist of a subset of one of these sets and at most one other element. In the event that the process fails because there comes a time at which some set P of schools has insufficient capacity for the students who have no options outside P, the first response is to add P to our list. If this does not resolve the issue because even with more schools being monitored, there still comes a time at which P has insufficient capacity, we run iterations of the process that monitor larger and larger supersets of subsets of P, until the issue is resolved. This procedure is described in more precise detail in Subsection B.8 of the Appendix, and in the code itself.

Several remarks are in order. First, this procedure is an algorithm, in the sense that it is guaranteed to halt in finite time, because the collection of sets being monitored increases as the process progresses. Since minimal critical sets can be arbitrarily large, it is almost certainly the case that its worst case running time is exponential. Nevertheless, extremely large minimal critical sets seem to be rare events, and the procedure seems to work very well in practice. For example, it finishes in a few minutes on an example produced by `make_ex` with 100 schools and 900 students. Finally, there is room for further refinement of the procedure, and it is possible that there are variants that are much faster still.

References

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

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

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

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

A Downloading and Setting Up

Here we give step-by-step instructions for downloading the code, compiling the executables, and starting to use them. I am going to 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 (I wouldn't know) but a Windows user can also just get Cygwin.)

First, in a browser, open the url

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

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

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

```
%% tar xvf gcps_schools.tar
%% cd gcps_schools
```

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. -c normal.c
gcc -o make_ex example.c normal.o -lm
gcc -I. -c parser.c
gcc -I. -c subset.c
gcc -I. -c cee.c
gcc -I. -c schchprob.c
gcc -I. -c partalloc.c
gcc -I. -c solver.c
gcc -o gcps solve.c parser.o subset.o cee.o schchprob.o
  partalloc.o solver.o normal.o -lm
gcc -I. -c implement.c
gcc -o purify purify.c normal.o parser.o subset.o partalloc.o
  implement.o -lm
```

Each line of output above corresponds to one of the commands in the `makefile`, and each of the commands in the makefile specifies an object to be constructed (either an object (that is, a `.o` file) or an executable) the resources that are required to construct it, and the command that constructs it. (There is also an object `all`, which requires the three executables. When make is asked to construct something (e.g., the command `make gcps`) it first makes all the resources that that thing requires, so `make all` will result in constructing all of the executables at once.

Because `make`'s default behavior is to construct the first object in the `makefile`, `make` has the same effect as `make all`. There is also an object `clean`. The command `make clean` removes the objects and executables, and also any of the `*~` files that the `emacs` editor leaves behind after a preexisting file has been edited. The command `make clean` takes us back to the situation before `make` was invoked, and after that you can issue the command `make` again to see it all happen again.

There are now the executables `make_ex`, `gcps`, and `purify`. On most 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 `./make_ex`, `./gcps`, and `./purify`. We begin with `make_ex`:

```
%% make_ex
/* This file was generated by make_ex with 2 schools, 3 students
per school, capacity 4 for all schools, school valence standard
deviation 1.00, and idiosyncratic standard deviation 1.00.  */
There are 6 students and 2 schools
The vector of quotas is (4,4)
The priority matrix is
   1 1
   1 1
   1 0
   1 0
   1 0
   0 1
The students numbers of ranked schools are
(2,2,1,1,1,1)
The preferences of the students are
1:  1 2
2:  1 2
3:  1
4:  1
```

```
5:  1
6:  2
The priority thresholds of the schools are
1 1
```

Warning: If what you get looks a bit different, it may be because your installation of C and mine have different random number generators.

Let's redirect the output to the file `schools.scp`, then invoke `gcps`:

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

Finally, let's redirect the output of `gcps` to the file `allocate.mat`, then invoke `purify`:

```
%% gcps > allocate.mat
%% purify
/* This is a sample introductory comment.  */
    1:  2:
1:  0   1
2:  1   0
3:  1   0
4:  1   0
5:  1   0
6:  0   1
```

That's all there is to it! We've now been through a complete cycle, and the rest is up to you. If you feel like it, you may want to experiment with different parameters for `make_ex` by editing the file `example.c`, as described in Section 4, then running `make` again and going through the `make_ex`-`gcps`- `purify` cycle. This will give you an initial feel for how fast `gcps` is. (When applied to small school choice problems, it's *very* fast.) But after reading the rest of this guide, you may well have your own ideas concerning what to do next.

B 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 applications to districts with particular features. For this reason we have kept things as simple as possible, even if that entails somewhat less convenience for the user. In particular, the input and output formats are inflexible, and some users will probably want to develop more sophisticated interfaces.

In this Appendix we provide an overview of the code, passing from the simpler and more basic files to increasingly higher levels, in each case describing those features that might not be so obvious. Our hope is to ease the process of learning about the code by providing a level of explanation in which the objects in the code are described in human terms, and in relation to the earlier descriptions of the algorithms. While reading the descriptions of the files below, the reader should also be looking at the files themselves, and especially the header (`*.h`) files.

Before diving into details, here are some general remarks. First, although we have used C rather than C++ (for a project as small as this, the various advantages of C++ seem not worth the additional complexity of that language) the code is object oriented in spirit, being organized as interactions of objects given by `struct`s. Most 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 a prerequisite to any detailed understanding of the code.

With perhaps one or two exceptions, each object has 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 `make_ex`, `gcps`, and `purify`, and for other objects the printing functions can be useful for debugging. In all cases the code for these functions is simple and straightforward, and printing and destroyer functions will not be mentioned below.

When studying the code, the reader can mostly ignore the many calls to destroyers, trusting 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 bulk to the code, once you get used to it, in a curious way it seems to enhance the readability of the code.

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

The function `min` computes the minimum 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` have many digits is that an output of `gcps` must be an accurate input 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.

B.2 `example.c`

The file `example.c` contains the `main` function of `make_ex`, which contains all of the code that is involved in generating an example. Although the code is somewhat lengthy, the process is a straight line:

(a) Locate the schools and students around the circle.

(b) Compute the matrix of distances between students and schools.

(c) Generate normally distributed random valences for the schools.

(d) The utility of student `i` for school `j` is the valence of `j` plus a normally distributed `(i,j)`-idiosyncratic shock minus the distance from `i` to `j`.

(e) Each student's safe school is (roughly) the one that is closest.

(f) Student `i`'s priority at school `j` is one if its utility for i is not less than the utility of `i`'s safe school, and otherwise it is zero.

(g) The preference of student `i` is the list of schools of priority one, in order of decreasing utility.

B.3 `parser.h` and `parser.c`

Two parsing functions `sch_ch_prob_from_file` and `allocation_from_file` are declared in `parser.h`. 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. There are numerous functions checking that the verbal tokens are the ones that are expected, and quitting with an error message if one of them isn't. This makes the code extremely verbose and thoroughly amateurish. If the reader kindly refrains from looking in `parser.c`, this author will be spared considerable embarrassment.

B.4 `subset.h` and `subset.c`

One may represent a subset of $\{1,\ldots,n\}$ as an n-tuple of 0's and 1's, or as a list of its elements. The first of these is given by `subset`, which, in addition to the n-tuple `indicator` of elements of $\{0,1\}$, keeps track of the number of elements of the subset and the number of elements of the set it is a subset of. 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 elements of $\{1,\ldots,$ `large_set_size`$\}$. The function `index_of_subset` passes from the first 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, if will complain and halt the program.

A `square_matrix` is a `dimension` $\times$ `dimension` matrix whose (i,j) entry is an integer `entries[i-1][j-1]`. In the most important use of this notion

$$\texttt{entries[i-1][j-1]} = \texttt{entries[j-1][i-1]} \in \{0,1\}$$

for all i and j, with `entries[i-1][j-1]` $= 0$ whenever `entries[i-1][i-1]` $= 0$ or `entries[j-1][j-1]` $= 0$. Such a matrix represents an undirected graph whose nodes are the i such that `entries[i-1][i-1]` $= 1$, with an edge between `i` and `j` if and only if `entries[i-1][j-1]` $= 1$.

A `subset_list` is a linked list of subsets in `index` form. A `subset_list` keeps the subsets in order from least to greatest. The ordering of subsets is lexicographic, with smaller subsets preceeding larger subsets and subsets of the same size ordered according the element of least index, the element of second smallest index if the two subsets have the same element of least index, etc. The function `reduced_subset_list` returns the list of subsets in `my_list` that are subsets of `my_subset`.

Mostly the functions in `subset.h` have self explanatory titles, with code that is not hard to understand. The most important and complex function is `next_subset`, which is described later.

B.5 `cee.h` and `cee.c`

A school choice *communal endowment economy* (CEE) consists of `no_students` students, `no_schools` schools, a specification of `quotas` (i.e., capacities) for the schools, and a matrix `priority` specifying a nonnegative integer `priority[i-1][j-1]` for each student `i` and each school `j`. When a CEE occurs as a part of an input, the `quotas` are usually integers, but partially allocated CEE's are used in the computations, when the remaining unallocated `quotas` are floating point numbers. For this reason there are `int_cee`'s and `double_cee`'s. Some of the more advanced functions in `cee.h` are specific to priorities that are either 0 or 1; in Section 5 we explained how to pass from more complicated priorities to binary priorities using priority thresholds.

The computations of `popular_schools` and `relatedness_matrix` are straightforward, and were explained in Section 6. The function `sub_double_cee` computes the `sub_cee` obtained from `given_cee` by restricting to the set of students given by `stu_index` and the set of schools given by `sch_index`.

B.6 `schchprob.h` and `schchprob.c`

A *school choice problem* combines a CEE, which may be thought of as describing the outcomes that are physically possible, with preferences for the students and priority thresholds for the schools. A student is *eligible* for a school if her priority at that school is at or above the school's priority threshold. A student's (strict) preference is the list of the schools she is eligible for, going from best to worst. For convenience we keep track of each student's number of eligible schools.

The underlying CEE may be either an `int_cee` or a `double_cee`. Typically the input school choice problem has an `int_cee`, and `double_cee`'s are used in computing an allocation, so there are `input_sch_ch_prob`'s, which have `int_cee`'s, and `sch_ch_prob`'s, which have `double_cee`'s. A `sch_ch_prob` is typically what remains to be allocated after a certain time, so it has a member `time_remaining`.

The function `sch_ch_prob_from_input` takes an `input_sch_ch_prob` as input and passes to a `sch_ch_prob` by converting the quotas from integers to floating point numbers, and by setting `time_remaining` to 1.0. The function `reduced_sch_ch_prob` passes from a `sch_ch_prob` with general priorities and priority thresholds to one in which the priorities of student `i` at school `j` is one if `i` is eligible to attend school `j` and zero otherwise, and the priority thresholds of all schools are one, as described in Section 5.

During the allocation process, when a GMC inequality for a set P of schools is encountered, there is a smaller allocation problem for P and the set J_P of students who, at that point in the process, are not eligible for any schools outside of P. There is a similar allocation problem for the complements P^c and J_P^c of P and J_P, and the continuation of the allocation process is the sum of the allocation processes for these subproblems. The function `sub_sch_ch_prob` constructs the subproblem for $J_P =$ `stu_subset` and $P =$ `sch_subset` and the subproblem for $J_P^c =$ `stu_compl` and $P^c =$ `sch_compl`.

The function `time_re_after_first_gmc_eq` computes the time remaining if the allocation process continues until the GMC inequality for `school_subset` and `captive_students` holds with equality or the unit interval of time is exhausted, ignoring all other constraints. It may happen that the GMC inequality for these sets has already been violated. In this case the argument `overallocated_schools` is set equal to `school_subset`, and the current attempt at computing an allocation is abandoned as quickly as possible, then restarted after the list of schools being monitored has been adjusted by adding `overallocated_schools` to it.

The function `time_remaining_after_first_gmc_eq` considers all the school subsets in known_facets and observed_overallocated_sets, and all related connected subsets with depth or few elements. For each such set of schools time_remaining_of_gmc_eq is applied to that set and the set of students who cannot be assigned further probability in schools outside that set. It returns the maximum of these quantities while setting the pointees of crit_stu_subset and crit_sch_subset to subsets that attain the maximum.

B.7 partalloc.h and partalloc.c

In a partial_alloc for no_students students and no_schools schools, allocations is a matrix that specifies an allocation allocations[i-1][j-1] of school j to student i 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.

The function allocate_until_new_time creates a partial_alloc in which each student receives

my_scp->time_remaining − new_time_remaining

units of her favorite school and none of any other school. Increasing a base partial allocation by adding an increment partial allocation to it is what increment_partial_alloc does. The function school_sums returns an array of double that specifies, for each school, the total amount of it that has been allocated in my_alloc.

B.8 solver.h and solver.c

The files solver.h and solver.c contain the top level of code that defines the executable gcps. The function GCPS_schools_solver_top_level repeatedly attempts to find the desired allocation by initiating a computation in which GCPS_schools_solver calls itself recursively.

The subset list known_facets is a list of sets of schools that have been critical during some attempt. An attempt can fail in two ways. First, it can encounter a critical set new_critical_set of schools that has not been observed before, in which case the index of new_critical_set is added to known_facets and the next attempt begins. The other possibility is that at some point the computation discovers that a set overallocated_schools of schools has allocated its seats to an extent such that it can no longer meet the needs of the students who cannot attend schools outside the set. The index of overallocated_schools is added to the list observed_overallocated_sets, and another attempt is initiated.

Observing an overallocated_schools means that the allocation process went past some GMC inequality, and it is necessary to find this inequality before the process can succeed. The integer parameter depth is the maximal size of sets (in addition to those in known_facets and observed_overallocated_sets) that are examined when looking for either the next new_critical_set or an overallocated_set, which may or may not be one that has already been seen. Each time an attempt returns an overallocated_set that has been seen before, depth is increased by one. When an attempt returns a new_critical_set, depth is reset to 1.

Searching over sets of size depth throughout the entire search process can be extremely burdensome, and it is natural to expect that the missing new_critical_set can be found near the end of the allocation process as it currently stands. The variable level is the depth of recursion, i.e., the number of times GCPS_schools_solver has called itself. The maximum set size searched over is set to depth only when

$$\texttt{max\_level} - \texttt{depth} \leq \texttt{level} \leq \texttt{max\_level},$$

where max_level is the level at which the overallocated_set is found.

We now turn to the description of GCPS_schools_solver. The key function applied here is time_rem_after_first_gmc_eq, which is coded in schchprob.h and schchprob.h. The main purpose of this function is to compute the amount of time that will remain if the allocation continues allocating the favorite school to each student until there is a critical set of schools or an overallocated set of schools, among the sets of schools that are searched over. A side effect of this function is to compute the critical or overallocated set. If the critical set has not been observed before, or if there is an overallocated set, the process reports this back to the instance of GCPS_schools_solver that called it, or to GCPS_schools_solver_top_level, by returning a partial allocation that will be discarded.

Otherwise GCPS_schools_solver allocates up until the computed time, and it then splits the remaining allocation problem into two subproblems, and calls itself on each of these. The first “left” subproblem relates to the critical set sch_subset and the set stu_subset of students who are no longer able to consume any school outside of this set. The second “right” subproblem is obtained by restricting to the complements stu_compl and sch_compl of these sets. For each of these subproblems the result may be either a new_critical_set or an overallocated_set, or it may be a complete allocation for the subproblem.

When we look at the coding of time_rem_after_first_gmc_eq in schchprob.c, we see that it scans over a collection of sets of schools. For each set it asks how long the allocation process could allocate the favorite school to each student before that set becomes critical. If the amount of time is negative, then the set is overallocated. If it finds an overallocated set of schools, it reports that set and sets exit_status to 2. If there is no overalllocated school, then it reports the maximum time remaining after some set becomes critical, and a set of schools that attains this maximum, setting exit_status to 1 if that set's index is not already in known_facets.

The collection of sets that time_rem_after_first_gmc_eq scans over consists of the sets in known_facets and observed_overallocated_sets, and the sets generated by the function next_subset, which is coded in subset.h and subset.c. This function depends of the matrix related, which encodes the undirected graph whose nodes are the schools that still might exhaust their capacities, with an edge between schools i and j if there is a student who might attend either one. As we explained in Section 6, a minimal critical set necessarily induces a connected subgraph of related. One could enumerate all the sets of schools of size subset_size with this properties by first enumerating all the sets of schools that might still fill up of size subset_size, then asking for each one whether it induces a connected subgraph of related, but when the set of schools is large, this is very laborious. Instead next_subset takes advantage of the fact that each set of schools that induces a connected subgraph has a unique ordering in which the first school is the least (according to the numbering of the schools) school in the set, the second school is the least among the remainder that is connected to the first, the third is the least in the remainder that is connected to either the first or the second, and so forth. These orderings of schools can be ordered lexicographically, and next_subset passes from a set to the next set in this ordering. The code that does this is intricate, and perhaps the most difficult part of the code to understand.

B.9 implement.h and implement.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 3. 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 3. 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.10 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 simple and straightforward.