GCPS Schools: A User's Guide

Andrew McLennan*

June 25, 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 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.

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

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 below, 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 is no worse for her than that.

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:

```
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. (It is a major advantage of our mechanism that we can provide such a guarantee.) Each student submits a ranking of schools that she is eligible for and (weakly) prefers to the 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. In fact this condition is sufficient for the existence of a feasible allocation: if, for each set of schools P, the total capacity of the schools in P is not less than the number of students in J_P, 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 current partial allocation) 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`

The output of `gcps` is a matrix of assignment probabilities, as shown in the example below.

	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

(Now, to avoid confusion, the schools are A, B, and C.)

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
```

Supposing that we decide which of these to do by flipping a coin, we could then look for another cycle and repeat the process. Each step reduces the number of entries that are strictly between 0 and 1, so we would eventually arrive at a deterministic assignment.

We now give a more formal explanation of the algorithm, as implemented in `implement.h`, `implement.c`, and `purify.c`. 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 node 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.

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 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 whose homes are 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 change the parameters by editing `example.c`, run `make`, and then issue a command like `make_ex > 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 on the next page. 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.

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 a school 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.)

```
/* 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
```

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.

To help things along a bit, the computer looks for schools whose capacity will not be exceeded even if every student who ranks it ends up receiving a seat. 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. The computer computes a square matrix `related` whose rows and columns are indexed by the schools, such that `related[j][k]` is one if either $j = k$ or j and k are both popular and $\alpha_j \cap \alpha_k \neq \emptyset$, and otherwise `related[j][k]` is zero. We think of `related` as encoding an undirected graph whose nodes are the schools, with an edge connecting j and k if and only if `related[j][k]` and `related[k][j]` are both one. 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. If P is critical, then both P_1 and P_2 are critical, so P cannot be a minimal critical set.

At this point we have seen that it is only necessary to consider subsets P of the set of popular schools such that the subgraph of `related` induced by P is connected. For large school choice problems the number of such sets can still be overwhelming.

The trick is to let the computational process itself tell us which sets are relevant. We maintain an assignment `subset_sizes` of a nonnegative integer to each school. At the beginning of the computation `subset_sizes[j]` is zero if j is unpopular and one if j is popular. This corresponds to simply allocating each school until it is exhausted, and (as in our initial example) it may be the case that this computes a feasible allocation. If it does not, there will be a time prior to 1 at which some student i no longer has any schools she can consume. For each school j in α_i we increase `subset_sizes[j]` from 1 to 2. We now repeat the allocation processs, this time looking for criticality of sets of schools whose elements are a school in α_j and another school that is related to it. In general, the allocation process looks for criticality of each connected set of schools P such that there is a school j such that $j \in P$ and the number of elements of P is not greater than `subset_sizes[j]`. If the allocation process does not succeed there is a student i who runs out of schools before time 1, and we increase `subset_sizes[j]` by 1 if $j \in \alpha_i$ and `subset_sizes[j]` is minimal for $j \in \alpha_i$.

In this way there arises the need to efficiently enumerate all of the relevant P, which is provided by the function `next_subset` in `subset.c`. For any `related`-connected r-element set P and any $j \in P$ there is a unique ordering $j_1, \ldots, j_r$ of the elements of P such that $j_1 = j$ and, for each $h = 2, \ldots, r$, j_h is the smallest number among $j_h, \ldots, j_r$ that is `related` to one of the schools $j_1, \ldots, j_{h-1}$. We endow these sets with a particular kind of lexicograpical order, namely that $\{j_1, \ldots, j_r\} < \{j'_1, \ldots, j'_{r'}\}$ if any of the following hold:

(a) $\{j_1, \ldots, j_r\} = \emptyset$.

(b) $j_1 < j'_1$.

(c) $j_1 = j'_1$ and $r < r'$.

(d) $j_1 = j'_1$, $r = r'$, and there is h such that $2 \leq h \leq r$, $j_1 = j'_1, \ldots, j_{h-1} = j'_{h-1}$, and $j_h < j'_h$.

For a given set $\{j_1, \ldots, j_r\}$ with the elements ordered as above, the function `next_subset` finds the next set in the ordering. If $\{j_1, \ldots, j_{r-1}, j'_r\}$ is the next set, then $j'_r > j_r$, and if h is the smallest index such that j_h and j'_r are `related`, then $j_{h+1}, \ldots, j_{r-1}$ are all smaller than j'_r, because if $h+1 \leq g \leq r$ and $j_g \geq j'_r$, then $j_g > j_r$ and j_g is not the smallest element of $j_g, \dots, j_r$ that is related to some element of $j_1, \dots j_h$. Conversely, if j'_r is the smallest number such that j'_r is related to one of $j_1, \dots, j_{r-1}$ and $j'_r > \max\{j_{h+1}, \dots, j_r\}$, where h is the smallest index such that j_h and j'_r `related`, then $\{j_1, \dots, j_{r-1}, j'_r\}$ is the next set in the ordering, and `next_subset` tests the numbers greater than j_r, in order, for these conditions.

If there is no such j'_r, then `next_subset` tests the numbers $j'_{r-1} > j_{r-1}$, in order, for being `related` to some j_h with $1 \leq h \leq r-2$ and satisfying $j'_{r-1} > j_{h+1}, \dots, j_{r-1}$, where h is the smallest such index. Having found such a j'_{r-1}, it looks for the smallest j'_r such that $\{j_1, \dots, j_{r-2}, j'_{r-1}, j'_r\}$ is satisfactory, either finding such a j'_r or moving on to the next j'_{r-1}. If there is no satisfactory j'_{r-1}, then `next_subset` considers possible $j'_{r-2} > j_{r-2}$, and so forth. If there is no successor of $\{j_1, \dots, j_r\}$ of the form $\{j_1, \dots, j_r\}$, then `next_subset` looks for the first successor of the form $\{j_1, j'_2, \dots, j'_{r'}\}$ where $r' > r$, and if there is no such successor, then it moves on to the enumeration of subsets containing $j_r + 1$.

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 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 less convenience for the user. In particular, the input and output formats are inflexible, and some users will probably want to develop a more sophisticated interface.

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.

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. With perhaps one or two exceptions, each object has a destroyer, which deallocates the memory that stores the object's data, and for many objects there is a way of printing the object, primarily 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 should ignore 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.

A.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. 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.

A.2 `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 characters without any 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.

A.3 `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 representation to the second.

A `square_matrix` is a `dimension`$\times$`dimension` matrix whose (i,j) entry is an integer `entries[i-1][j-1]`. The most important use of this notion is to represent an undirected graph with

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

if $i=j$ or the graph has an edge with endpoints i and j, and otherwise

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

The coding of `next_subset` has already been described in Section 6.

A.4 `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 `increase_subset_sizes` increases the components of `subset_sizes` for schools that `underallocated_student` is eligible for and for which the current value is minimal.

The function `minimum_gmc_inequality` takes a `double_cee` and a (pointer to) a set `school_subset` as arguments, computes the set of students that cannot attend any school outside `school_subset`, returns 1 (true) if the GMC inequality for `school_subset` and this set of students holds, and returns 0 (false) otherwise. The function `gmc_holds` decides whether the argument satisfies `minimum_gmc_inequality` for every subset of the set of schools. The running time is proportional to 2 raised to the power of the number of schools, which can be practical when the number of schools is moderate, say around 20, but of course it blows up rapidly after that.

A.5 `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`.

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`. This function has a new argument `underallocated_student`, which is a pointer to an integer. This interger is 0 until there is a subproblem for which some student has no eligible schools, at which point it becomes that student's number. The general idea is that when this happens, the current attempt at an allocation is abandoned as quickly as possible, and another attempt is made after increasing the minimal `subset_sizes` for the schools that the student is initially eligible for.

The function `time_remaining_of_gmc_eq` computes the time that remains 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. The function `time_remaining_after_first_gmc_eq` considers all the school subsets generated by `next_subset` with the parameters `related` and `subset_sizes`. 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.

A.6 `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 `increment_partial_alloc` increases a `base` partial allocation by adding an `increment` partial allocation to it. The function `allocate_until_new_time` creates a partial_alloc in which each student receives

$$\texttt{my\_scp->time\_remaining} - \texttt{new\_time\_remaining}$$

units of her favorite school and none of any other school. 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.

A.7 solver.h and solver.c

The function GCPS_schools_solver_top_level creates a copy of the input my_scp, the vector subset_sizes, which is initially just the indicator function of the set of popular schools, the matrix related, and the integer pointer underallocated_student. It then asks GCPS_schools_solver to try to find the GCPS allocation. If GCPS_schools_solver comes back with underallocated_student != 0, then subset_sizes is increased by applying increase_subset_sizes and the process is repeated, again and again, until underallocated_student == 0 and the computation is complete.

The function GCPS_schools_solver begins by computing the time that will remain after the allocation has proceeded to the first GMC equality, and stu_subset and sch_subset are set to be the student and school subsets of this equality. The complements stu_compl and sch_compl of these sets are computed, and first_alloc is the result of giving each student her favorite school, at unit speed, until end_time. At end_time the problem splits into two subproblems (possibly one of them is null because it has no students) and in the process of constructing them an underallocated_student may be found. If this happens, then GCPS_schools_solver aborts, sending first_alloc (which is of no consequence) and the underallocated_student back to the function that called GCPS_schools_solver, which may be either GCPS_schools_solver itself or GCPS_schools_solver_top_level.

If the construction of the subproblems does not turn up an underallocated_student, then GCPS_schools_solver is called on each of them. (In this sense the algorithm is recursive.) Of course these calls my find an underallocated_student, whose number in the smaller problem must be translated into a number in the current problem. The calls for the two problems produce left_alloc and right_alloc, which are partial_alloc's, and increment_partial_allocation is used to add these to first_alloc, which is now the return value of the current call to GCPS_schools_solver.

A.8 `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 algorithm 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 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 the `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.

A.9 `solve.c`, `purify.c`, and `example.c`

The files `solve.c`, `purify.c`, and `example.c` contain the `main` functions of the executables `gcps`, `purify`, and `make_ex`, respectively. The `main` functions in `solve.c` and `purify.c` are simple and straightforward, but the `main` function in `example.c` 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 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.