GCPS Schools: A User's Guide

Andrew McLennan*

January 5, 2024

Abstract

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

1 Introduction

In the paper "An Efficient, Computationally Tractable School Choice Mechanism" (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, the 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
```

GCPS Schools input files begin with a comment between `/*` and `*/`. This is purely for your convenience. The comment can be of any length, and provide whatever information is useful to you, but it is mandatory insofar as the computer will insist that the first two characters of the file are `/*`, and it will only start extracting information after it sees the `*/`. 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 output shown below. (Here and below we are leaving out the sample comment.) 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*.

```
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
```

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 again below. (Now, to minimize 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.

	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

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 $n_1, \dots, n_k$ such that $k > 2$ and, for each $i = 1, \dots, k$, 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 $j = 1, \dots, i-2$ such that $n_i = n_j$, in which case $n_j, \dots, n_{i-1}$ 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 $n_1, \dots, n_k$, for each $i = 1, \dots, k$ 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 you are familiar with have interfaces with the user that neither require nor allow the user to edit the source code, but to create such an interface here would be counterproductive. It would add complexity to the source code that had nothing to do with the underlying algorithms. More importantly, 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.)

References

Ahuja, R. K., Magnanti, T. L. and Orlin, J. B. (1993), *Network Flows: Theory, Algorithms, and Applications*, Prentice Hall.

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

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

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

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

Cheriyan, J. and Maheshwari, S. N. (2005), 'Equivalence of efficiency notions for ordinal assignment problems', *Lecture Notes in Computer Schiece* **338**, 30–48.

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

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

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

McLennan, A., Tamura, Y. and Takayama, S. (2023), An efficient, computationally tractable school choice mechanism. University of Queensland Working Paper.

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. -Wall -Wextra -c normal.c
gcc -o make_ex example.c normal.o -lm
gcc -I. -Wall -Wextra -c parser.c
gcc -I. -Wall -Wextra -c subset.c
gcc -I. -Wall -Wextra -c cee.c
gcc -I. -Wall -Wextra -c schchprob.c
gcc -I. -Wall -Wextra -c partalloc.c
gcc -I. -Wall -Wextra -c push_relabel.c
gcc -I. -Wall -Wextra -c gcps_solver.c
gcc -o gcps solve.c normal.o parser.o subset.o cee.o
  schchprob.o partalloc.o push_relabel.o gcps_solver.o -lm
gcc -I. -Wall -Wextra -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 many Unix's these can invoked simply by typing the executable name on the command line, but it may be the case that, for security reasons, the current directory is not in the `path` (the list of directories that the command line looks in when a command is invoked) in which case you will need to type `./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. (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 index representation can be much more efficient when we are dealing with little subsets of big sets.

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

A index_list is a linked list of subsets in index form.

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

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

The most general notion of a *communal endowment economy* (CEE) has a finite set of *agents*, a finite set of *objects*, a vector of (nonegative) *requirements* for the agents, a vector of (nonnegative) *quotas* for the objects, and a matrix that specifies, for each agent and each object, the maximum amount of the object that the agent may consume. A *partial allocation* is matrix that specifies a nonnegative consumption of each object by each agent. A partial allocation is *feasible* if: a) no agent consumes more than the allowed quantity of any object; b) the sum of each agents allocation is her requirement; c) the sum of each object's allocations does not exceed the object's quota. In the context of school choice the agents are *students*, the objects are *schools*, each student's requirement is one, and each school has *quota*, which is the number of seats it can fill.

In the software, a CEE is part of the algorithm's input, and CEE's also occur as the allocation unfolds, so two CEE structs are defined in `cee.h`, `input_cee` and `process_cee`. An `input_cee` has a matrix `priority` that specifies a nonnegative integer for each student-school pair. At the most basic level the `priority` for a student-school pair would be 1 if the student was eligible to attend the school and weakly preferred it to her safe school. But this matrix presents an opportunity to encode the priority of each student at each school, which we can use to reduce to the more basic level, as we explain below.

In `process_cee` the quotas and `maximums` for each student-school pair are floating point numbers, and there is an additional number `time_remaining` that encodes the amount of time that remains for allocation for a `process_cee`. The `maximums` will often either agree with `time_remaining` or be zero, but the algorithm allows them to be in between.

With `maximums` that are strictly between 0 and `time_remaining` the notion of a *critical pair* becomes a bit more complex. A pair (J, P) consisting of a set of students J and a set of schools P is *critical* if the only way to meet the requirement of the agents in J is to give each student in J her maximum of each of the schools outside P, and to give all of the seats in schools in P only to students in J. When the pair (J, P) becomes critical for the CEE of as yet unallocated resources, the computation recursively descends to two subprocesses, one for the agents in J and all the resources that they might possibly consume (which is the minimum needed to meet their requirements) and the other for students in the complement of J and schools in the complement of P. The functions `critical_sub_process_cee` and `crit_compl_sub_process_cee` compute the CEE's of these subprocesses.

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.

There are two types of school choice problem, `input_scp` and `process_scp`, which differ according to whether the underlying CEE is an `inpu_cee` or a `process_cee`, and whether the schools have `priority_threshold`'s.

The function `process_scp_from_input` passes from an `input_sch_ch_prob` to a derived `sch_ch_prob`. It converts the quotas from integers to floating point numbers, and its sets `time_remaining` to 1.0. For each student and school, the student's `maximums` at the school is 1.0 if the input `priority` is not less than the school's `priority_threshold`, and positive (that is, the `priority` and the `priority_threshold` are not both zero) and otherwise it is 0.0. For each student the `no_eligible_schools` of the output `process_scp` is the `no_eligible_schools` of the input `input_scp` less the number of schools with positive input `priority` that was below the school's threshold, and the output `preferences` are the input `preferences` restricted to the remaining schools.

Using `critical_sub_process_cee` and `crit_compl_sub_process_cee`, the functions `critical_sub_process_scp` and `crit_compl_sub_process_scp` compute the derived `process_scp`'s when a pair (J, P) becomes critical. In the `process_scp` given by `crit_compl_sub_process_scp` the quotas of schools in the complement of P are reduced by the `maximums` for these schools of the students in J.

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

A `partial_alloc` is *feasible* if the total amount assigned to each student is 1 and the total assigned amount of each school is not more than the school's quota. A `partial_alloc` is *possible* if it is (component-wise) less than or equal to some feasible allocation. The algorithm we are implementing allocates, at unit speed, increments of each student's favorite school to her until it encounters some face of the polytope of possible allocations, at which point some pair (J, P) is critical and there is recursive descent to subproblems, as described above.

The detection of faces of the polytope of possible allocations is aided by keeping track of a feasible allocation, called the `feasible_guide`, that is (component-wise) greater than or equal to the allocation we are computing. The allocation we are computing, and `feasible_guide`, are piecewise linear functions of time, and the algorithm moves from one endpoint of a linear piece to another by computing a direction `theta` of motion for `feasible_guide`, then finding the maximum amount of time `delta` that the allocation can be increased by adding favorites to the allocation while moving `feasible_guide` according to `theta` before some constraint becomes binding. The functions `augment_partial_alloc` and `adjust_feasible_guide` adjust the computed allocation and `feasible_guide` accordingly.

When a pair (J, P) becomes critical the `feasible_guide`'s of the derived subproblems are obtained from `feasible_guide` by restriction, using the functions `left_feasible_guide` and `right_feasible_guide`. After the allocations of the subproblems have been computed, the function `increment_partial_alloc` combines them with the allocation for the larger problem that has already been computed.

B.8 `push_relabel.h` and `push_relabel.c`

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

B.9 `gcps_solver.h` and `gcps_solver.c`

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

B.10 `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.11 `solve.c` and `purify.c`

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

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

C The Principal Algorithms

In this appendix we provide brief descriptions of the two principal algorithms. These are intended mainly to help the reader figure out what is going on in the code. For more information the reader is referred to the papers that are the sources of these algorithms.

C.1 Push-Relabel

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

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

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

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

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

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

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

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

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

so we may define *value* of f to be

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

C.2 School Choice Communal Endowment Economies

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

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

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

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

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

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

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

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

C.3 Computing the GCPS Allocation

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

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

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

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

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

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

$$p(t_k) + \theta^k(t - t_k) \le \overline{p}(t_k) + \theta(t - t_k). \tag{$*$}$$

Now t_{k+1} is the first time after t_k such that one of the following holds: a) $p_{ie_i^k}(t_{k+1}) = g_{ie_i^k}$ for some i; b) $\overline{p}(t_k) + \theta(t - t_k) \notin Q$ for $t > t_{k+1}$; c) $(*)$ does not hold for $t > t_{k+1}$. For $t \in [t_k, t_{k+1}]$ we have determined that $p(t) = p(t_k) + \theta^k(t - t_k)$, and we set $\overline{p}(t) = \overline{p}(t_k) + \theta(t - t_k)$. Having determined $p(t_{k+1})$ and $\overline{p}(t_{k+1})$, we can repeat the process.

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

We now describe an algorithm that determines which of these possibilities holds, In the first case it finds a satisfactory θ, and in the second case it finds a suitable (J, P).

Suppose that there is a $\theta \in \mathbb{Z}^{I \times O}$ such that for there exists a $t' > t_k$ such that for all $t \in [t_k, t']$, $(*)$ holds and $\overline{p}(t_k) + \theta(t - t_k) \in Q$. Together these conditions imply that $p(t_k) + \theta^k(t - t_k) \in R$, so the first possibility above holds, and we can use θ to define the continuation of $\overline{p}$. The algorithm may be thought of as a search for such a θ.

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

(a) For each i and j:

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

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

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

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

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

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

Let
$$\tilde{P} = \{\, o : \sum_i \overline{p}_{ij}(t_k) = q_o \text{ and } \sum_i \theta_{ij} > 0 \,\}.$$
If $\sum_{j \in \tilde{P}} \sum_i \theta_{ij} \leq 0$, then (c) holds. Suppose that this is not the case. We now describe a construction that may or may not be possible. When it is possible, it passes from θ to a $\theta' \in \mathbb{Z}^{I \times O}$ satisfying (a) and (b) such that if $\tilde{P}' = \{\, o : \sum_i \overline{p}_{ij}(t_k) = q_{j_h} \text{ and } \sum_i \theta'_{ij} > 0 \,\}$, then
$$\sum_{j \in \tilde{P}'} \sum_i \theta'_{ij} = \sum_{j \in \tilde{P}} \sum_i \theta_{ij} - 1.$$

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

Choose $j_0 \in \tilde{P}$, and let $P_0 = \{j_0\}$. We define sets $J_1, P_1, J_2, P_2, \ldots$ inductively. If P_{h-1} is given, let $J_h = \bigcup_{j \in P_{h-1}} J_h(j)$ where

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

If J_h is given, let $P_h = \bigcup_{i \in J_h} P_h(i)$ where

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

Suppose that for some h there is an $j_h \in P_h \setminus P_{h-1}$ such that either $\sum_i \overline{p}_{ij_h} < q_{j_h}$ or $\sum_i \theta_{ij_h} < 0$. We can find a $i_h \in J_h$ such that $j_h \in J_h(i_h)$, then find an $j_{h-1} \in P_{h-1}$ such that $i_h \in J_h(j_{h-1})$, and so forth. Define θ' by setting

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

for $g = 1, \ldots, h$ and $\theta'_{ij} = \theta_{ij}$ for all other (i, j).

It is easy to see that θ' satisfies (a), because θ'_{ij} differs from θ_{ij} only when the difference is permitted, according to (i)–(iii). For each i, $\sum_j \theta'_{ij} = \sum_j \theta_{ij}$, either by construction if $i = i_g$ for some g, or because $\theta'_{ij} = \theta_{ij}$ for all j. Since θ satisfies (b), θ' also satisfies (b).

For $o \notin \{j_0, \ldots, j_h\}$ we have $\theta'_{ij} = \theta_{ij}$ for all i and thus $\sum_i \theta'_{ij} = \sum_i \theta_{ij}$. For each $g = 1, \ldots, h-1$ we have $\sum_i \theta'_{ij_g} = \sum_i \theta_{ij_g}$ by construction. Clearly $\sum_i \theta'_{ij_0} = \sum_i \theta_{ij_0} - 1$ and $\sum_i \theta'_{ij_h} = \sum_i \theta_{ij_h} + 1$. Since either $\sum_i \overline{p}_{ij_h} < q_{j_h}$ or $\sum_i \theta'_{ij_h} \leq 0$, θ' has all desired properties.

It is impossible to construct θ' in this manner if there is no h and $j_h \in P_h \setminus P_{h-1}$ such that $\sum_i \overline{p}_{ij_h} < q_{j_h}$ or $\sum_i \theta_{ij_h} < 0$. Supposing that this is the case, let $J = \bigcup_h J_h$ and $P = \bigcup_h P_h$. We have $\sum_i \overline{p}_{ij}(t_k) = q_j$ for all $j \in P$. If $j \in P$ and $i \notin J$, then $\overline{p}_{ij}(t_k) = p_{ij}(t_k)$. If $i \in J$ and $o \notin P$, then $\overline{p}_{ij}(t_k) = g_{ij}$ if $j \in \alpha_i \setminus P$, and $\overline{p}_{ij}(t_k) = g_{ij} = 0$ if $o \notin \alpha_i$. Thus $\overline{p}(t_k) - p(t_k)$ is a feasible allocation for $E - p(t_k)$ that gives all of the resources in P to students in J, and it gives $g_{ij} - p_{ij}(t_k)$ to $i \in J$ whenever $j \in O \setminus P$. Clearly any feasible allocation also has these properties, so (J, P) is a critical pair for $E - p(t_k)$.

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