Genetic Programming

Genetic School Allocation Algorithm

Philip W.B. Michgelsen

January 31, 2016

Abstract

In the city of Amsterdam students are allocated to high schools by a computer algorithm, based on a preference list, that students hand in, over high schools. The algorithm in use for this was the Gale and Shapley Deferred Acceptance (DA) algorithm with multiple tie-breaking. The outcomes of this algorithm were severely criticized. In this report a new genetic algorithmic approach for school allocation is being introduced. This genetic algorithmic approach is very adaptive, which allows it, to be adjusted to withstand the criticisms the DA algorithm faced. In this paper, a particular implementation of the new proposed genetic school matching algorithm, with a particular fitness function, is introduced and explained. Some promising test results are included.

1 Introduction

In Amsterdam students are allocated to high schools ('middelbare scholen' in Dutch)[1] by a computer algorithm. Students hand in a preference list over all available high schools, such that a computer subsequently allocates (or 'matches') them to some high school, thereby considering their preferences. Since some high schools are extremely popular, not all students placing a popular high school highest on their preference list, can be allocated to that school. To fairly solve this problem, the algorithm used, must contain some lottery element, which essentially determines if some student can and will be allocated to his highest preferred school or to some lower preferred school.

The algorithm opted for in 2015 to preform the school allocation is a Deferred Acceptance (DA) algorithm with multiple tie-breaking. The DA algorithm is a matching algorithm introduced in 1962 by Gale and Shapley. The algorithm is sometimes referred to as the *stable marriage algorithm*. This algorithm was chosen because it is *strategy-proof*, i.e. handing in a dishonest preference list can never lead to a better outcome than handing in an honest preference list. The final allocation obtained after running the DA algorithm was however severely criticized, because it created pairs of children, that could, by switching the schools assigned to them, both end up at a school higher on their preference list. Switching after running the DA algorithm is however not allowed, since it would break the *strategy-proofness* guaranteed by the algorithm. That is, if students know beforehand, that there is an option to switch, after running the algorithm, an honest school preference list, is no longer the dominant strategy for students. Students consider a school allocation where they want, with honest motives, to switch schools with one and another afterwards, but are not allowed to do so, as very unsatisfactory.

In this report we will introduce a genetic algorithm for preforming school allocation based on preference lists of the students. To do this, we will first give some formal definitions of the concepts we will use throughout this presentation and discussion; second, we will introduce the workings of the algorithm; third, we will discuss some test results; and finally we will discuss some limitations of the genetic school allocation algorithm and indicate directions for future research. Our hope is, that the genetic algorithm, introduced in this short report, will be *strategy-proof*, *pareto-efficient* and performs *utilitarianly better* than the DA algorithm. The test results look promising, but theoretical results have yet to show, that this genetic algorithm can be a more reasonable option for preforming school allocation in Amsterdam.

2 Genetic School Matching

A school allocation algorithm needs to have the following information in order to function: first, it needs to have a list of possible high schools, supplemented with their student capacity; second it needs to have a

[1] In The Netherlands there exist primary schools (4 - 12 years of age) and high schools (12 - 16/17/18 years of age)

list of students that want to be allocated; and finally it needs preference lists of each student over the high schools. What we exactly understand under these concepts, will become clearer in the following definitions.

Definition 1 (Preference List) *Let $\mathcal{H} = \{h_1, ..., h_n\}$ be the set of all available high schools. Let $\mathcal{S} = \{s_1, ..., s_m\}$ be the set of all students that need to be allocated to some high school. Let $\mathcal{P}$ be the set of all permutations of $\mathcal{H}$ s.t. a permutation is a finite sequence of high schools, wherein a high school h_i occurs only once. We then say that a* preference list *of some student s_i is some ordered n-tuple $(h_i, ..., h_j) \in \mathcal{P}$.*

Definition 2 (Preference Position) *Let p be some preference list in $\mathcal{P}$, let h_i be some high school in $\mathcal{H}$, where $\|\mathcal{H}\| = n$. Furthermore, let $\pi : \mathcal{P} \times \mathcal{H} \to \{1, ..., n\}$ be the* injective *preference position function, s.t. $\pi(p, h_i)$ is the position of school h_i in preference list p, where $\pi(p, h_i) = 1$ denotes that h_i is the most preferred school in p. If p_s is the preference list of some student s, we say that $\pi(p_s, h_i)$ is the preference position of h_i for s. With the preference position function π we can define a preference position ordering $>_{p_s}$, for every student s with preference list p_s on $\mathcal{H}$, s.t. $h_i >_{p_s} h_j$ iff $\pi(p_s, h_i) < \pi(p_s, h_j)$ (i.e. if s prefers h_i over h_j).*

Definition 3 (Allocation) *Let $\mathcal{H}$ again be a set of high schools and let $\mathcal{S}$ be a set of students. An* allocation *between $\mathcal{H}$ and $\mathcal{S}$ is a graph $A = (V, E)$ s.t.*

- *$V = \mathcal{S} \cup \mathcal{H}$, i.e. the vertices represent the high schools and the students;*
- *$E \subseteq \mathcal{H} \times \mathcal{S}$;*
- *$\forall s \in S \exists h \in H\, ((s, h) \in E)$, i.e. every student must be assigned to some school.*

We say some allocation is possible if we have, that the number of outgoing edges for some school $h \in V$, is at most the capacity of school h.

Definition 4 (Strategy-Proof) *Let p_{s_i} and p'_{s_i} be two distinct preference lists of some student s_i. Say that p_{s_i} is the real preference list of student s_i (i.e. if $\pi(p_{s_i}, h_i) = 1$, then s_i actually prefers h_i over all other high schools and thus hopes to be allocated to school h_i). Furthermore, let $\mathbb{P}[(p_{s_i}, h_i)]$ denote the probability that s_i will be allocated to school h_i when he hands in preference list p_{s_i}. We then say some allocation algorithm is* strategy proof *iff:*

$$\forall h_i \in \mathcal{H} \left(\mathbb{P}[(p_{s_i}, h_i)] \geq \mathbb{P}[(p'_{s_i}, h_i)] \right) \text{ iff } \pi(p_{s_i}, h_i) \leq \pi(p'_{s_i}, h_i).$$

In words: to hand in an honest preference list is the dominant strategy, assuming the students wants to be allocated to the school he actually prefers the most.

Definition 5 (Pareto-Efficiency) *Let $\mathcal{H}$ again be a set of high schools and let $\mathcal{S}$ be a set of students. Let $A = (V, E)$ be some allocation between $\mathcal{H}$ and $\mathcal{S}$. We then say that A is* pareto-efficient *iff:*

$$\forall (h, s) \in E\, \neg \exists (h', s') \in E \left(h' >_{p_s} h \wedge h >_{p_{s'}} h' \right).$$

In words: an allocation is pareto-efficient if there are no two students, that can be allocated to a school, that they prefer over the school they are allocated in under A, if they switched schools.

Before we introduce the genetic algorithm, we will first briefly explain what is meant be a genetic school matching algorithm. In the words of Goldberg, "Genetic algorithms are search algorithms based on the mechanics of natural selection and natural genetics."[2] Genetic algorithms are search methods that use ideas from genetics to perform a directed randomized efficient search to an optimum set of data points, in a given set of points, determined by some given *fitness function*. In our present setting, this means that we want to use the genetic algorithm, to search for the *fittest* school allocation, in the set of all allocations over some given set of students $\mathcal{S}$ and set of high schools $\mathcal{H}$.

[2] Goldberg 1989: p. 1.

Genetic algorithms use the idea of survival of the fittest, among data, combined with a structured yet randomized information exchange, given by the ideas of mutation and crossover, to provide a robust and very efficient optimum search algorithm. A genetic algorithm reproduces a sample of organisms, from a population, in such a way that, without heaving to go through the entire population, the fittest data points can be found.[3]

Genetic algorithms are functions that take as input a (i) sample of a particular population, in our case some set of possible allocations, (ii) a fitness function, in our case some measure to decide how 'good' the allocation is, (iii) a crossover parameter, (iv) a mutation parameter, (v) some bound on the duration of the algorithm, and an optional bound on the sizes of each generation. In our setting, given the foregoing inputs, our genetic algorithm, will return a set of allocations, the organisms in our simulated evolution, that are the fittest surviving, and hopefully optimal, school allocations.

Genetic algorithms essentially make use of three operations: reproduction, mutation and crossover. Together, these operations create new generations from the sample of the population, in such a way, that they tend to create the fittest set of organisms. The operation of reproduction is usually carried out first, creating new organisms from old, by incorporating the idea of survival of the fittest; then the operation of crossover is carried out on some percentage of the population, given by the crossover parameter; finally, the operation of mutation is carried out changing the organism slightly based on the mutation parameter.

There is no fixed set of rules how to perform the three genetic operations. They can be changed according to the search problem. However, it is customary, that (i) reproduction gives every organism a probability of surviving in to next generation based on its share in the total fitness, (ii) that mutation randomly alters a tiny fraction in a single surviving organism, and (iii) that crossover exchanges some fundamental structural property between two organisms.

It is clear, that much of the working of a genetic algorithm, depends on the fitness function. Essentially the fitness function is given by the search itself. In our case, we want to find the school allocation, that overall tends to place students as close to their most preferred school as possible. Note that this is not the only way possible, how one can measure the fitness of a school allocation. Another option could be to look for the allocation that places most students in their top k. The latter would probably enforce us to use another fitness function. The genetic school allocation algorithm introduced in this essay, will try to find the school allocation with a *mean preference position* as close to 1.

Definition 6 (Mean Preference Position) *Let A be some allocation over some set of high schools $\mathcal{H} = \{h_1, h_2, ..., h_n\}$ and students $\mathcal{S} = \{s_1, s_2, ..., s_m\}$. Let $a : \mathcal{S} \times \mathcal{A} \to \mathcal{H}$ be some function, that given some student and allocation, gives the school to which to student is allocated under that allocation. We then say that the mean preference position μ_A of allocation A is:*

$$\mu_A = \frac{\sum_{i=1}^{n} \pi(s_i, a(A, s_i))}{n}.$$

We call some allocation A_1 utilitarianly better *than some other allocation A_2 iff $\mu_{A_1} < \mu_{A_2}$.*

3 The Genetic Matching Algorithm

3.1 Running the Genetic Algorithm

The genetic matching algorithm presented here was written in Haskell. The code can be found in Section 7. In the following, we will mean by the genetic matching algorithm (or something similar) the particular algorithm written in Haskell. In the discussion of this algorithm, we will try to abstract from the particular algorithm, as implemented in Haskell, as much as possible, to elucidate the general principles of the genetic algorithm. Nonetheless, in explaining the general principles we will point to particular functions in the Haskell algorithm, such that the reader can find the actual computational implementations, of the principle under discussion. If we refer to some function in the Haskell algorithm, we will always use the following font: `Haskell`.

[3] Goldberg 1989: p. 1.

We use the function `evolve` to run our particular version of the genetic school allocation algorithm. This function takes the following arguments:

- `seed`: this number determines the randomness in the algorithm.
- `highschools`: this is a list of pairs *(school, capacity)*, which the algorithm needs to determine where it can allocate students.
- `gensize`: this is a fixed bound on the size of the initial sample and each generation, needed to perform reproduction.
- `nrgen`: this gives the number of reproductions the algorithm needs to perform, s.t. 0 returns the initial sample and n returns the n-th generation.
- `cpar`: this must be a number in [0,1] corresponding to the percentage of organisms upon which the operation of crossover will be performed.
- `mpar`: this must be a number in [0,1] corresponding to the percentage of students in a single allocation the will be mutated.

The list of high schools and their capacity (the argument `highschools`) must be such that there is at least one school, and that the sum of the capacities of all school is at least the number of students.

The function `evolve` runs the genetic matching algorithm, by first generating the student preferences. A general version of the algorithm, would of course not produce this, but would receive this list as an argument. In the particular example algorithm, however, the number of students is fixed at 100. This can be changed, but then the function `generateStudentpreferences`, used to generate the preferences of the students, should be altered accordingly. The function `generateStudentpreferences` is created in such a way that some schools are very popular (schools 1, 2 and 3), some very unpopular (schools 9 and 10) and the others randomly popular. This is done, in order to represented the actual situation in Amsterdam.

3.2 Initial Population

After creating the student preferences, the function `evolve` generates an initial population of allocations, i.e. the sample in the search space. These allocations are not randomly generated allocations. To optimise the genetic algorithm, we start with a set of pareto-efficient allocations, which are created using the following procedure:

Permuation Allocation Generation:

(i) randomly pick a student s from a given set of students $\mathcal{S}$ (N.B. this is the set of all students upon first execution of this step);

(ii) allocate student s to the school that he prefers most and that is still available.

(iii) Set $\mathcal{S}$ to $\mathcal{S} \setminus \{s\}$;

(iv) repeat steps (i)-(iii) until $\mathcal{S} = \varnothing$.

The above procedure creates a single pareto-efficient allocation. To create more allocations, the procedure can simply be repeated. Due to the uniform random pick of students in step (i), the allocations obtained by the procedure are likely to be all distinct. The implementation of this procedure can be found in the functions `createSample` and `createAllocation`. In the function `createAllocation` the (ordered) set of all students is first randomly permuted, s.t. the new order over the students obtained, can be used to allocate the students successively in the school that they prefer most and that is still available.

Theorem 1 *The Permutation Allocation Generation procedure only creates pareto-efficient allocations.*

Proof. Let $A = (V, E)$ be some allocation created by the Permutation Allocation Generation procedure. Let s_1 and s_2 be two distinct students in V and let h_1 and h_2 be two distinct schools in V. Suppose w.l.o.g. that s_1 is picked before s_2 and that $(s_1, h_1), (s_2, h_2) \in E$. Now suppose for reductio that A is not pareto-efficient and that we have $h_2 >_{s_1} h_1$ and $h_1 >_{s_2} h_2$. Since s_1 is picked before s_2 and that we have $h_2 >_{s_1} h_1$ and $(s_1, h_1) \in E$, it must have been the case that school h_2 was full when s_1 was allocated to a school by the procedure. For suppose not, then s_1 was allocated to school h_2 and not h_1, because he prefers h_2 over h_1. Now note that the Permutation Allocation Generation procedure never removes students from schools. This means that if h_2 was full, when s_1 was allocated, it must have been still full, when s_2 was allocated. This means that s_2 could not have been allocated to h_2, but we have $(s_2, h_2) \in E$: contradiction. □

3.3 Reproduction & the Fitness Function

When the student preferences and the initial population are created, and the maximum number of generations specified is greater than 0, the function `evolve`, creates a new generation from the initial population. The first step in this procedure is to reproduce the initial population. Reproduction is carried out by performing the function `reproduction`, inside the larger function `createGen`.

The function `reproduction` uses a combination of a *roulette wheel reproduction technique* and *elitism*. In roulette wheel reproduction, each organism x_i is reproduced with a probability corresponding to their share of the total fitness.[4] Before we can introduce the reproduction procedure, we need to clarify our fitness function.

Definition 7 (Scaled Preference Position Fitness) *Let $A = (V, E)$ be some allocation over the set of high schools $\mathcal{H} = \{h_1, h_2, ..., h_n\}$ and students $\mathcal{S} = \{s_1, s_2, ..., s_m\}$. Let $a : \mathcal{S} \times \mathcal{A} \to \mathcal{H}$ be some function, that given some student and allocation, gives the school to which to student is allocated. We then say that the fitness of some allocation A by a scaled preference position fitness function $f : \mathcal{A} \to \mathbb{N}$ is:*

$$f(a) = \begin{cases} \left(\sum_{i=1}^{n} (11 - \pi(s_1, a(A, s_i))) \right)^2 & \text{if the allocation is possible (cf. Definition 3)} \\ 0 & \text{otherwise} \end{cases}$$

We call the fitness function from Definition 7 a *scaled* preference position fitness function, because we square the sum, to create a greater variance among the fitness values, such that the fitter organisms become, relatively, even fitter. It should be clear that possible allocations that have a mean preference position closer to 1, get assigned higher fitness values.

Using the above scaled preference position fitness function f, we can now perform reproduction, using a roulette wheel reproduction technique, by carrying out the following procedure:

Roulette Wheel Reproduction Procedure

1. Calculate $f(A_i)$ for all allocations A_i in our population;
2. Associated $p_i = \dfrac{f(A_i)}{\sum_{j=1}^{n} f(A_j)}$ with each A_i;
3. Create intervals $[0, p_1), ..., [\sum_{j<i}^{n} p_j, \sum_{j\leq i}^{n} p_j), ..., [\sum_{j<n}^{n} p_j, \sum_{j\leq n}^{n} p_j]$, s.t. to each $\langle A_i, p_i \rangle$ there is one interval associated;
4. Uniformly pick $c-1$ random real numbers $r_1, ..., r_c$ from $[0, 1]$, where c is the bound on the generation size;
5. For each r_j: if r_j falls in interval $\langle A_i, p_i \rangle$, then reproduce A_i.

Note that in the above procedure we have, that the probability that allocation A_i is reproduced $\mathbb{P}[A_i]$, is exactly its share in the total fitness of the population, that is:

[4] cf. Goldberg 1989: p. 30.; Mitchell 1999: 124-126.; Tuyt & Michgelsen 2016.

$$\mathbb{P}[A_i] = \frac{f(A_i)}{\sum_{j=1}^{n} f(A_j)}$$

After reproduction by the above procedure, we however still need to carry out the *elitist* part of the reproduction procedure. Since it is possible that the fittest organism, will not be reproduced, by a roulette wheel reproduction technique, we want to ensure, nonetheless, that it will be reproduced. This means we first need to calculate what the the fittest organism is, in the population being reproduced (cf. `bestentity` in `reproduction`); we then add this fittest organism to our new generation. In this way, we ensure that 1 copy of the fittest organism reproduces with probability 1. This is also the reason why we only pick $c-1$ random numbers in the Roulette Wheel Reproduction Procedure.

3.4 Crossover

When the organisms are reproduced, some percentage of these organisms, determined by the crossover parameter, will get crossed over. As explained earlier, crossover is carried out in genetic algorithms, to guide the search for the optimal datum, by exchanging fundamental structural properties between organisms. In our setting, such a suitable structural property is the set of students allocated to some specific school. Let $\mathcal{S}$ again be our set of students and $\mathcal{H}$ our set of high schools. Furthermore, let again $a : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ be some function, that given some student and allocation, gives the school to which to student is allocated. Moreover, let A_1, A_1 be two reproduced allocations that are selected for crossover (cf. `crossoverpool` in `createGen`). Consider the following crossover procedure:

School Students-Block Crossover

(i) randomly pick some school h_k from the list of schools;

(ii) create $b_i = \{s \mid s \in \mathcal{S}, a(s, A_i) = h_k\}$, i.e. the set of students allocated to h_k in allocation A_i, for both A_1 and A_2;

(iii) replace b_1 in A_1 by b_2 and replace b_2 in A_2 by b_1;

(iv) remove one copy of students that now occur twice in A_i, s.t. the no student allocated to h_k in A_i is removed;

(v) add students not occurring in A_i, s.t. they are allocated to the school that is preferred most by them and is still available.

The implementation of this procedure can be found in the functions `crossover` and `highestLeftover`. Note that this crossover operation, upon given a *possible* allocation will only produce new *possible* allocations.

3.5 Mutation

After performing crossover on some of the organisms in our population, we perform the mutation operation on all organisms. Although the operation is performed on all organisms, this does not mean that all organisms will be mutated. The probability that some allocation gets altered due to mutation is:

$$\text{mutation parameter} \times \text{number of students.}$$

The operation of mutation is defined on organisms in the following way. Let ρ be the mutation parameter. Let A be some allocation consisting of high schools $h_1, ..., h_n$ and students $s_1, ..., s_n$. For every student s_i in allocation A we then randomly pick some $p_i \in [0, 1]$, if $p_i < \rho$ then we randomly pick some $h_j \in \{h_1, ..., h_n\} \setminus \{a(s_i, A)\}$ and consequently delete $(a(A, s_i), s_i)$ in A and add (h_j, s_i) to A (where a is again the function that, given some allocation and student gives the school to which the student is allocated under that allocation).

This mutation function widens the search space of the genetic algorithm, but not without a cost. The operation of mutation can create *impossible* allocations from *possible* allocations, since there is no check if the new school assigned, by mutation, actually has the capacity to be assigned another student.

4 Results

4.1 Representation in Haskell

To present the test results of the genetic school matching algorithm, first the representations used need to be introduced.

- *List of Schools:* The list of schools is represented by a list of pairs *(school, capacity)*, where both the schools and the capacities are integers. In our test case the following list `highschools` is used:

  `[(1,10),(2,10),(3,12),(4,12),(5,10),(6,15),(7,12),(8,12),(9,12),(10,15)]`

- *List of Students:* The list of schools is represented by a list of integers 1 to 100, such that each student is simply represented by an integer:

  `[1,2,3,...100]`

- *Student Preferences:* The student preferences are represented by a list of paris *(student,preferences)*, where the first element is an integer from the *list of students* and the second element is an integer from the *list of schools*, s.t. the first element in preferences is the school preferred most by the student. The student preferences used when running evolve can be obtained by running `generateStudentpreferences (generateSeeds 0 3 !! 0) highschools`:

  `[(1,[1,2,3,7,6,5,4,8,9,10]),...,(100,[8,7,1,10,6,9,4,2,3,5])]`

- *Allocation:* a single allocation is represented as a list of pairs *(school, students allocated)*, where the first element is an integer from from the *list of schools*, and the second element is a list of students allocated to that school, which is some subset of integers from the *list of students*, which is not specifically ordered. An example allocation can be obtained by running `generateAllocation 0 highschools (generateStudentpreferences 0 highschools)`:

  `[(1,[17,15,21,26,8,10,2,30,9,22]),...,(10,[83,96,74,86,85,87,89,73,62,71,79,68,61,75])]`

4.2 Running and Reading the Algorithm

As indicated earlier the genetic school allocation algorithm can be ran by running the function `evolve`. This function will produce a *population*, which is just a list of pairs *(allocations, frequency)*, where the second element indicates how often the allocation occurs in the population.

The functions `readPopulation` and `giveBestAllocation` can be used, to interpret a population produced by `evolve`. The function `readPopulation` returns the mean preference position of all the allocations in the population; the function `giveBestAllocation` gives the lowest mean preference position occurring in the population.

Throughout the tests the list of schools `highschools` is used; as the list of students, 100 students, represented by integers 1 to 100 are used. Furthermore, seed 0 is used in running evolve. This means that the student preferences used are the ones generated by running `generateStudentpreferences (generateSeeds 0 3 !! 0) highschools`. As bound on the generation size 40 is chosen and, with exception of the first test case, 100 generations are produced. The crossover and mutation parameter used, differ from test case to test case.

4.3 Test Results

4.3.1 Only Pareto Efficient Allocations

- Evolution: `evolve 0 highschools 40 0 0.0 0.0`
- Result: `readPopulation 0 it highschools`:
  1.8999996, 1.9499998, 1.9200001, 2.0600004, 2.0100002, 1.9899998, 2.04, 2.08, 2.0200005, 2.0100002, 1.9499998, 1.9700003, 2.0200005, 1.9399996, 1.8999996, 1.8999996, 2.0699997, 2.13, 1.9899998, 2.0699997, 1.9399996, 1.9099998, 2.0699997, 1.9300003, 2.0, 1.9399996, 2.0600004, 1.8500004, 1.9700003, 1.9899998, 2.0, 1.9499998, 2.0, 1.9200001, 1.9799995, 1.9799995, 2.0900002, 1.9300003, 2.08, 2.04
- Best mean preference position:: 1.8500004

4.3.2 Evolution with only Reproduction

- Evolution: `evolve 0 highschools 40 100 0.0 0.0`
- Result: `readPopulation 0 it highschools`:
  1.8500004
- Best mean preference position: 1.8500004

Comparing this run and the run above, it is clear, that, because reproduction with elitism is used, the 1.8500004 allocation survives, since it was the best mean preference position in the initial population.

4.3.3 Evolution with only Mutation

- Evolution: `evolve 0 highschools 40 100 0.0 0.005`
- Result: `readPopulation 0 it highschools`:
  1.8500004, -1.0, -1.0, 1.9799995, -1.0, 2.0200005, 1.8699999, 1.9799995, -1.0, -1.0, 1.8900003, 1.9399996, -1.0, 1.8999996, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0
- Best mean preference position: 1.8500004

A mutation parameter of 0.005 means that in every two allocations one student is expected to mutate. This means that in a population of 40, every generation 20 allocations are expected to have one student to be assigned to another randomly selected school, than to the school the student was assigned.

Note that again the best allocation from the initial population survives. This happens, because in a elitist approach to reproduction, it is also ensured that this fittest organism cannot mutate, nor crossover. Furthermore, note that in the result -1.0 mean preference positions are found. These are used to indicate that these allocations are not *possible* allocations. As noted in the explanation of mutation, it is likely that operation of mutation creates *impossible* allocations, because the new randomly assigned school to the student could very well not have the capacity to be assigned another student.

4.3.4 Evolution with only Crossover

- Evolution: `evolve 0 highschools 40 100 0.5 0.0`
- Result: `readPopulation 0 it highschools`:
  1.8100004, 1.9899998, 1.96, 1.8999996, 1.9200001, 1.8699999, 1.96, 1.8599997, 1.8900003, 1.8999996, 1.9399996, 1.79, 1.9200001, 1.9300003, 1.8800001, 1.8999996, 1.8999996, 1.8999996, 1.8800001, 1.9099998, 1.8500004, 1.9399996, 1.9399996, 1.8599997, 1.9799995, 1.9200001, 1.8800001, 1.96, 1.9099998, 1.9700003, 1.9300003, 1.8699999, 1.8999996
- Best mean preference position: 1.79

In the above run, half of the allocations of a generation crosses over. The best allocation now clearly has a lower mean preference position, than the best allocation in the initial population. This means that the genetic algorithm has optimised the initial set of pareto-efficient allocations. This might even be the optimum of the set of all possible allocations.

4.3.5 Evolution with both Crossover and Mutation

- Evolution: `evolve 0 highschools 40 100 0.5 0.005`
- Result: `readPopulation 0 it highschools`:
  1.8400002, 2.3000002, -1.0, 2.13, 2.46, 2.21, 1.9300003, -1.0, -1.0, 2.0900002, 2.17, 1.8999996, 2.12, 2.38, 2.1499996, 2.13, 1.96, 2.12, 2.29, 2.1899996, 2.12, 2.1400003, 2.0900002, -1.0, -1.0, 2.1000004, 1.8599997, 2.3199997, 2.17, -1.0, 2.1999998, 2.3500004, 2.1899996, 2.1800003, 2.1800003, 1.9399996, 2.3500004, 2.1999998
- Best mean preference position: 1.8400002

The best allocation is slightly better than the best allocation in the initial population, but the difference is very small and could therefore be due to the numerical imprecision of Haskell.

4.3.6 Evolution with high Crossover and low Mutation

- Evolution: `evolve 0 highschools 40 100 0.8 0.00025`
- Best mean preference position: 1.79
- Evolution: `evolve 0 highschools 40 100 0.8 0.001`
- Best mean preference position: 1.8100004

In the first of two runs above, one student in the generation is expected to mutate; in the second, four students are expected to mutate. These examples demonstrate that crossover seems to be the operation responsible for the optimisation and that mutation is not very effective. Higher mutation parameters prevent the algorithm from finding the allocation with a mean preference position equal to, or lower than, 1.79.

5 Discussion

5.1 Limitations

The genetic algorithm proposed for the student allocation problem in Amsterdam looks very promising. Especially the crossover operation really seems to improve allocations significantly. The test results are however very premature, because the student population, is only vaguely based on the real student population of Amsterdam, but is by no means statistically equal to the real student population. Furthermore, the algorithm that was in use to allocate students in the city of Amsterdam, could also cope with some students having a priority for some school. In this preliminary version of a genetic school allocation algorithm, this cannot be implemented (yet). There seem to be several possibilities for incorporating these priorities in the genetic algorithm, by either adjusting the fitness function, or by taking another initial population.

5.2 Future Research

From our test results it seems that the effect of mutation is not very large. It seems reasonable therefore to consider other mutation techniques. One possible option could be, to not only alter the school of one student, but also to interchange the schools, that two students are assigned to, maybe even considering the preferences lists, when doing so.

Furthermore, the fitness function could be made to account for more desired properties of the allocations. One obvious property could be pareto-efficiency, other candidates could be the number of students that are allocated in their top-k, or not to look only at the mean preference position, but also at the frequency of each preference position occurring in some allocation.

Finally, without any comparison with the outcomes of the DA algorithm, on the same student population, it is impossible to say if the genetic matching algorithm performs utilitarianly better than the DA algorithm. It is therefore necessary to also implement this algorithm in Haskell in a suitable way, as to compare the outcomes between the two. This implementation could also open the door, for another initial population, namely one created by the DA algorithm.

6 Bibliography

1. Goldberg, D.E., (1989), *"Genetic Algorithms in Search, Optimization, and Machine Learning"*, Addison-Wesley Publishing Company.
2. Mitchell, M. (1999), *"An Introduction to Genetic Algorithms"*, MIT press.
3. Tuyt, O. & Michgelsen, P. (2016), *"Reproduction Techniques"*, URL: https://github.com/janvaneijck/gpme/blob/master/GPME%20Reproduction%20Techniques.pdf.

7 Code

```
module PWBM_GeneticSchoolMatching where
import System.Random

-- Needed for the Student Generator and Permutation Function --
import Control.Monad
import Data.List
import Data.Array.ST
import Control.Monad.ST
import Data.STRef
import Data.Ord (comparing)



-- Types --
-------------------------------------------------------------------------------------------
type Seed         = Int
type Student      = Int
type School       = Int
type Frequency    = Int
type Allocation   = [(School,[Student])]
type Highschools  = [(School,Frequency)]
type Preferences  = [(Student,[School])]
type Population   = [(Allocation, Frequency)]



-- Help Functions --
-------------------------------------------------------------------------------------------

-- Generalisation of 'div' to any instance of Real (from Data.Fixed)
div' :: (Real a,Integral b) => a -> a -> b
div' n d = floor ((toRational n) / (toRational d))

-- Generalisation of 'mod' to any instance of Real (from Data.Fixed)
mod' :: (Real a) => a -> a -> a
mod' n d = n - (fromInteger f) * d where
    f = div' n d

-- The function generateseeds generates randomnumbers based on some seed
generateSeeds :: Seed -> Int -> [Int]
generateSeeds seed k = map (`mod` 10000) (take k (randoms $ mkStdGen seed :: [Int]))

-- randomRange takes a seed, an interval and a quantity n and returns n value
--  uniformly picked from the interval using the seed.
randomRange :: Random a => Seed -> (a, a) -> Int -> [a]
randomRange seed (a,b) k = take k $ randomRs (a,b) (mkStdGen seed)

-- The function shuffle generates a permutation of some given list based on the seed
shuffle :: Seed -> [b] -> [b]
shuffle seed xs = let
    gen = mkStdGen seed
    in runST (do
           g <- newSTRef gen
           let randomRST lohi = do
               (a,s') <- liftM (randomR lohi) (readSTRef g)
               writeSTRef g s'
               return a
           ar <- newArray n xs
           xs' <- forM [1..n] $ \i -> do
                  j <- randomRST (i,n)
                  vi <- readArray ar i
```

```
                vj <- readArray ar j
                writeArray ar j vi
                return vj
          gen' <- readSTRef g
          return xs')
    where
       n = length xs
       newArray :: Int -> [a] -> ST s (STArray s Int a)
       newArray n xs = newListArray (1,n) xs

-- pairUp takes a list and pairs up every two elements.
pairUp :: [a] -> [(a,a)]
pairUp [] = []
pairUp [_] = error "pairUp ERROR: not an even number in population/generation!"
pairUp (x:y:xs) = (x,y) : pairUp xs

-- tripleUp takes a list and pairs up every three elements
tripleUp :: [a] -> [(a,a,a)]
tripleUp [] = []
tripleUp [_] = error "tripleUp ERROR: not a multiple of three in population"
tripleUp [_,_] = error "tripleUp ERROR: not a multiple of three in population"
tripleUp (x:y:z:xs) = (x,y,z) : tripleUp xs

-- cumList converts a list of probabilities (summing up to 1) into a list of
--  intervals corresponding to the probabilities
cumList :: Fractional t => [t] -> [(t, t)]
cumList [] = []
cumList (p:rest) = (0.0,p) : cumList' p rest where
    cumList' _ [] = []
    cumList' p (q:rest) = (p,p+q) : cumList' (p+q) rest

-- The function freqRep represents a population of individual organisms as pairs
-- (organism type, frequency) similar to the type Population.
freqRep :: (Eq a) => [a] -> [(a, Int)]
freqRep [] = []
freqRep (org:rest) = (org, (count org rest) + 1) :
                     freqRep (filter (\x -> x /= org) rest)
                       where
                           count _ [] = 0
                           count x (y:ys)
                                     | x == y = 1 + (count x ys)
                                     | otherwise = count x ys

-- The function studentRepresentation creates a list of schools, s.t. at index i the
--  school to which student i+1 is allocated is represented.
studentRepresentation :: Allocation -> [School]
studentRepresentation allocation = map fst $ sortBy (comparing snd) $
                                listofpairs allocation
    where
       listofpairs [] = []
       listofpairs ((sch,stds):rest) = listofpairs' sch stds ++ listofpairs rest
           where
               listofpairs' sch stds = map (\x -> (sch,x)) stds

-- The function allocationRepresentation creates an allocation from a list of schools
--  created in the syle of the function studentRepresentation.
allocationRepresentation :: [School] -> Highschools -> Allocation
allocationRepresentation studentrep highschools = let
    schools = map fst highschools
   in allocationRepresentation' studentrep schools where
    allocationRepresentation' _ [] = []
    allocationRepresentation' studentrep (hsch:hschs) =
       (hsch, map (\x -> x+1) (elemIndices hsch studentrep)) :
           allocationRepresentation' studentrep hschs
```

```
-- Student, School and Allocation Generator --
-------------------------------------------------------------------------------

-- The list highschools is a list of pairs (school, capacity).
highschools :: Highschools
highschools = [(1,10),(2,10),(3,12),(4,12),(5,10),(6,15),(7,12),(8,12),(9,12),(10,15)]

-- The function studentgroup generates a group of students with specific preferences
studentgroup :: Seed -> Highschools -> [Student] -> [Int] -> [Int] -> Preferences
studentgroup seed schools students toppref lowpref = let
    schoolslist    = map fst schools
    leftoverschools = (schoolslist \\ toppref) \\ lowpref
    seeds          = generateSeeds seed (length students)
    in studentgroup' seeds students leftoverschools toppref lowpref where
        studentgroup' [] _ _ _ _ = []
        studentgroup' _ [] _ _ _ = []
        studentgroup' (seed:seedrest) (student:studentrest) leftoverschools top low =
            [(student,(toppref ++ (shuffle seed leftoverschools) ++ lowpref))] ++
            studentgroup' seedrest studentrest leftoverschools top low

-- The function generateStudentpreferences generates a list of students, given
--  some seed according to the semi-random specified structure below.
generateStudentpreferences :: Seed -> Highschools -> Preferences
generateStudentpreferences seed highschools = let
        s       = highschools
        seeds   = generateSeeds seed 6
        genericstudents1 = studentgroup (seeds !! 0) s [1..15]
                           [fst (s !! 0), fst (s !! 1), fst (s !! 2)]
                           [fst (s !! 7), fst (s !! 8), fst (s !! 9)]
        genericstudents2 = studentgroup (seeds !! 1) s [16..30]
                           [fst (s !! 0), fst (s !! 2), fst (s !! 1)]
                           [fst (s !! 7), fst (s !! 8), fst (s !! 9)]
        genericstudents3 = studentgroup (seeds !! 2) s [31..38]
                           [fst (s !! 1), fst (s !! 0), fst (s !! 2)]
                           []
        genericstudents4 = studentgroup (seeds !! 3) s [39..45]
                           [fst (s !! 1), fst (s !! 2), fst (s !! 0)]
                           []
        genericstudents5 = studentgroup (seeds !! 4) s [46..60]
                           []
                           [fst (s !! 8),fst (s !! 9)]
        genericstudents6 = studentgroup (seeds !! 5) s [61..100]
                           []
                           []
        in genericstudents1 ++ genericstudents2 ++ genericstudents3 ++
           genericstudents4 ++ genericstudents5 ++ genericstudents6

-- The function generateAllocation creates an allocation based on a permutation of the
--  list of student, then creating a list, based on this order, s.t. for each student
--  the highest still available school is picked.
generateAllocation :: Seed -> Highschools -> Preferences -> Allocation
generateAllocation seed highschools prefs = let
    studs          = map fst prefs
    permut         = shuffle seed studs
    startalloc     = map (\(s,c) -> (s,[])) highschools
   in sortBy (comparing fst) $
        generateAllocation' permut prefs (highschools,startalloc)
    where
        generateAllocation' [] _ (_,alloc) = alloc
        generateAllocation' (s:rest) prefs (highschools,alloc) =
            generateAllocation' rest prefs (pickTopSchool s prefs highschools alloc)

-- The function pickTopSchool returns the highest-preferred school that is still
```

```
--  available for some given student, given some allocation.
pickTopSchool ::  Student
               -> Preferences
               -> Highschools
               -> Allocation
               -> (Highschools,Allocation)
pickTopSchool student prefs highschools allocation = let
   Just prefstud  = lookup student prefs
  in pickTopSchool' prefstud highschools allocation where
       pickTopSchool' [] _ _ = error "student cannot be placed"
       pickTopSchool' (s:rest) highschools allocation = let
                              Just capacity = lookup s highschools
                              Just allocated = lookup s allocation
                             in if capacity > 0
                                  then ((s, capacity-1):
                                      delete (s,capacity) highschools,
                                             (s,student:allocated):
                                               delete (s,allocated) allocation)
                                  else pickTopSchool' rest highschools allocation



-- Genetic Algorithm --
-------------------------------------------------------------------------------

-- CreateSample
createSample :: Seed -> Int -> Preferences -> Highschools -> [Allocation]
createSample seed size prefs highschools = let
    seeds = generateSeeds seed size
   in createSample' seeds highschools prefs where
       createSample' [] _ _ = []
       createSample' (s:rest) highschools prefs =
           generateAllocation s highschools prefs :
           createSample' rest highschools prefs

-- Fitness Function
-- The function allocationfitness is the fitness function that determines the fitness
--  of some allocation of students to highschools based, s.t.
--     - higher values of allocationfitness mean fitter allocations
--     - a fitness of zero is given to allocations that do not take the capacity of
--       school into account
fitnessalloc :: Allocation -> Preferences -> Highschools -> Float
fitnessalloc xs prefs highschools = if capacityCheck xs
                                  then fromIntegral $
                                      (fitnessalloc' xs prefs)^2
                                  else 0.0
    where
    capacityCheck [] = True
    capacityCheck ((s,xs):rest) = let
        Just capacity  = lookup s highschools
        nrallocated    = length xs
       in nrallocated <= capacity && capacityCheck rest
    fitnessalloc' [] _ = 0
    fitnessalloc' ((sch,stds):rest) prefs = (fitnessschool sch stds prefs) +
                                         fitnessalloc' rest prefs
        where
            fitnessschool _ [] prefs = 0
            fitnessschool sch (std:rest) prefs =
                (10 - (head $ elemIndices sch $ snd (prefs !! (std-1)))) +
                fitnessschool sch rest prefs

-- Function Evolve
-- The function evolve preforms the genetic algorithm.
evolve ::  Seed
          -- seed
```

```
        -> Highschools
          -- list of schools and their capacity
        -> Int
          -- bound on generation size
        -> Int
          -- maximum number of generations
        -> Float
          -- crossover parameter
        -> Float
          -- mutation parameter
        -> Population
          -- Population after evolution
evolve seed highschools gensize nrgen cpar mpar = let
       primeseeds = generateSeeds seed 3
       seedpref   = primeseeds !! 0
       seedpop    = primeseeds !! 1
       seedgen    = primeseeds !! 2
       prefs      = generateStudentpreferences seedpref highschools
       pop        = freqRep $ createSample seedpop gensize prefs highschools
       seedsgen   = generateSeeds seedgen nrgen
   in evolve' seedsgen prefs pop nrgen where
       evolve' _ _ pop 0 = pop
       evolve' _ _ [(perfectentity,0)] k = [(perfectentity,0)]
       evolve' (s:seeds) prefs pop k =
         evolve' seeds prefs (createGen s highschools gensize cpar mpar prefs pop)
                (k-1)

-- The function createGen creates a new generation from a given population, by
--  performing reproduction, mutation and crossover. The actual ellitism takes
--  place here, in the sense that the best reproduced organism cannot crossover
--  nor mutate.
createGen ::  Seed
             -- seed
           -> Highschools
             -- list of schools and their capacity
           -> Int
             -- bound in generation size
           -> Float
             -- crossover parameter
           -> Float
             -- mutation parameter
           -> Preferences
             -- preferencelists for all students
           -> Population
             -- old generation
           -> Population
             -- new generation
createGen seed highschools gensize cpar mpar prefs pop = let
       [(seedPool, seedCross, seedMut)] = tripleUp $ generateSeeds seed 3
       reproduced                   = reproduction seedPool pop gensize
                                        highschools prefs
       bestentity                   = head reproduced
       pool                         = tail reproduced
       nrstudents                   = fromIntegral $ length $ map fst prefs
   in if length reproduced == 1 &&
         (fitnessalloc (head reproduced) prefs highschools == nrstudents * 10)
          then [(head reproduced,0)]
          else let
                 sizecrossoverpool = round $ (fromIntegral gensize)*cpar -
                                     (mod' ((fromIntegral gensize)*cpar) 2)
                 crossoverpool   = pairUp $ take sizecrossoverpool pool
                 clonepool       = drop sizecrossoverpool pool
                 seedscross      = take gensize (randoms $ mkStdGen seedCross)
                 seedsmut        = generateSeeds seedMut gensize
               in freqRep $ bestentity : (mutate seedsmut highschools prefs
```

```
                    ((crossover' seedscross highschools prefs crossoverpool) ++
                       clonepool))
                    where
                     crossover' _ _ _ []                              = []
                     crossover' (s:srest) highschools prefs ((a,b):prest) =
                        (crossover s highschools prefs a b) ++
                            (crossover' srest highschools prefs prest)
                     mutate _ _ _ []                          = []
                     mutate (s:srest) highschools prefs (o:orest) =
                        (mutation s (mpar :: Float) highschools o) :
                            mutate srest highschools prefs orest

-- Function Reproduction
-- The function reproduciton performs reproduction using a roulette wheel
--  reproduction technique in combination with elitism.
reproduction ::  Seed
                  -- seed
                -> Population
                  -- population
                -> Int
                  -- bound on generation size
                -> Highschools
                  -- list of schools and their capacity
                -> Preferences
                  -- preferencelists for all students
                -> [Allocation]
                  -- allocations that are reproduced s.t. the first allocation is the
                  --  best entity of the population
reproduction seed pop size highschools prefs = let
       nrstudents = fromIntegral $ length $ map fst prefs
       pop'       = map fst pop
       fitnesspop = zip pop (map (\x -> fitnessalloc x prefs highschools) pop')
       bestentity = fst $ fst $ maximumBy (comparing snd) fitnesspop
       perfectlist = (filter (\((x,y),z) -> z == (nrstudents * 10.0)) fitnesspop)
       in if perfectlist /= []
           then [fst $ fst $ head perfectlist]
           else let
                  totalfit = sum $ map (\((x, freq), fit) -> fromIntegral freq * fit)
                                   fitnesspop
                  fitProb = map (\ ((x, freq),fit) ->
                                  fromIntegral freq * fit / totalfit )
                                fitnesspop
                  popInterval = zip (map fst pop) (cumList fitProb)
                  coinflips  = randomRange seed (0.0,1.0) (size-1)
              in bestentity : findStrings coinflips popInterval where
                  findStrings [] _ = []
                  findStrings (flip:rest) popInterval =
                      findStrings' flip popInterval : findStrings rest popInterval
                      where
                          findStrings' flip ((string,(a,b)):rest) = if b == 1
                              then if a <= flip && flip <= b
                                     then string
                                     else findStrings' flip rest
                              else if a <= flip && flip < b
                                     then string
                                     else findStrings' flip rest

-- Function Mutation
-- The function mutation1 flips a coin to determine if some student will be placed at
--  some other randomly picked school.
mutation ::  Seed
              -- seed
            -> Float
              -- mutation parameter
            -> Highschools
```

```
            -- list of schools and their capacity
          -> Allocation
            -- school allocation (organism)
          -> Allocation
mutation seed parameter highschools allocation = let
    allocation' = studentRepresentation allocation
    seeds  = generateSeeds seed (length allocation')
   in allocationRepresentation (mutation' seeds highschools allocation') highschools
   where
    mutation' [] _ _ = []
    mutation' _ [] _ = []
    mutation' (s:rest) highschools (sch:schs) = let
       f = fst $ randomR (0,1) (mkStdGen s)
       in
        if f < parameter
           then let
               schools = delete sch (map fst highschools)
               randindex = fst $ randomR (0, (length schools)-1) (mkStdGen s)
               sch' = schools !! randindex
               in sch' : mutation' rest highschools schs
           else sch : mutation' rest highschools schs

-- Function Crossover
-- The function crossover pick a random school, then switches the students allocated to
--  this school between to organisms and corrects the allocations with the function
--  highestLeftover
crossover :: Seed -> Highschools -> Preferences -> Allocation -> Allocation -> [Allocation]
crossover seed highschools prefs aloc1 aloc2 = let
       schools       = map fst highschools
       randindex     = fst $ randomR (0, (length schools)-1) (mkStdGen seed)
       randomsch1    = schools !! randindex
       leftoverschools = delete randomsch1 schools
       Just studsaloc1 = lookup randomsch1 aloc1
       Just studsaloc2 = lookup randomsch1 aloc2
      in fillupmissing prefs
        [(randomsch1, studsaloc2) :
         (removeduplicates studsaloc2 (delete (randomsch1, studsaloc1) aloc1)),
         (randomsch1, studsaloc1) :
         (removeduplicates studsaloc1 (delete (randomsch1, studsaloc2) aloc2))]
       where
           removeduplicates [] alloc = alloc
           removeduplicates _ [] = []
           removeduplicates candidates alloc = removeduplicates' candidates alloc
               where
               removeduplicates' candidates [] = []
               removeduplicates' candidates ((sch,stds):rest) =
                   (sch, rmvdpl candidates stds) : removeduplicates' candidates rest
                   where
                       rmvdpl candidates [] = []
                       rmvdpl candidates (std:stds) =
                           if std `elem` candidates then rmvdpl candidates stds
                                                    else std : rmvdpl candidates stds
           fillupmissing prefs [] = []
           fillupmissing prefs (alloc:allocs) = [fillupmissing' prefs alloc] ++
                                        fillupmissing prefs allocs where
               fillupmissing' prefs alloc = let
                   allstuds   = map fst prefs
                   allstudsnow = concat $ map snd alloc
                   missing    = allstuds \\ allstudsnow
                  in generateAllocation' missing prefs highschools alloc where
                       generateAllocation' [] _ _ alloc = alloc
                       generateAllocation' (s:rest) prefs highschools alloc =
                              generateAllocation' rest prefs highschools
                                  (highestLeftover s prefs highschools alloc)
```

```
-- The funciton highestLeftover places a student in the highest-preffered school that
--  is still available in some given allocation.
highestLeftover :: Student -> Preferences -> Highschools -> Allocation -> Allocation
highestLeftover student prefs highschools allocation = let
   Just prefstud  = lookup student prefs
  in highestLeftover' prefstud highschools allocation where
       highestLeftover' [] _ _ = error "student cannot be placed"
       highestLeftover' (s:rest) highschools allocation = let
                              Just capacity = lookup s highschools
                              Just allocated = lookup s allocation
                             in if (capacity - (length allocated)) > 0
                                    then ((s,student:allocated):
                                        delete (s,allocated) allocation)
                                    else highestLeftover' rest highschools allocation

-- The function calculateMeanPos calculates the mean preference-place of the students
--  in some allocation s.t. the i-th preffered school gets preference place i and the
--  first preffered school is the school the students places number 1 on his preference
--  list
calculateMeanPos :: Allocation -> Preferences -> Highschools -> Float
calculateMeanPos xs prefs highschools = if capacityCheck xs
                                then 11 - (fromIntegral $
                                    (calculateMeanPos' xs prefs))/100
                                else -1
    where
    capacityCheck [] = True
    capacityCheck ((s,xs):rest) = let
        Just capacity  = lookup s highschools
        nrallocated    = length xs
       in nrallocated <= capacity && capacityCheck rest
    calculateMeanPos' [] _ = 0
    calculateMeanPos' ((sch,stds):rest) prefs = (fitnessschool sch stds prefs)
                                          + calculateMeanPos' rest prefs
        where
        fitnessschool _ [] prefs = 0
        fitnessschool sch (std:rest) prefs =
            (10 - (head $ elemIndices sch $ snd (prefs !! (std-1)))) +
            fitnessschool sch rest prefs

-- The function readPopulation gives a list of the mean preference-palce in some
--  population
readPopulation :: Seed -> Population -> Highschools -> [Float]
readPopulation seedused population highschools = let
    prefs = generateStudentpreferences ((generateSeeds seedused 3) !! 0) highschools
   in readPopulation' population prefs highschools where
    readPopulation' [] _ _ = []
    readPopulation' ((aloc,freq):rest) prefs highschools =
        calculateMeanPos aloc prefs highschools : readPopulation' rest prefs highschools

-- The function giveBestAllocation gives a pair consisting of the mean preference-place
--  and some allocation, that was the allocation with the lowest mean preference-place
--  in some population.
giveBestAllocation :: Seed -> Population -> Highschools -> (Float, Allocation)
giveBestAllocation seedused population highschools = let
    allocations = map fst population
    scores      = zip allocations $ readPopulation seedused population highschools
    cleanscores = filter (\(x,y) -> y >= 1.0) scores
    highscore   = minimumBy (comparing snd) cleanscores
   in (snd highscore, fst highscore)
```