College Course Scheduling using a Genetic Algorithm

Hunter Kalinoski

1. Introduction

Time is the most valuable resource that humans have. It is the one thing that you can never get more of. In order to prevent wasting time, schedules are often created. However, creating the most optimal schedule is not trivial in most scenarios. There is a near infinite number of ways to organize one. On top of this, there is another near infinite number of ways to determine how good a schedule is. The combination of these two things makes it infeasible for even a computer to find the best option in some cases. Therefore, a different approach must be taken.

2. Background

2.1 The Problem. The problem that I will be analyzing in this paper concerns fitting a number of courses into a limited number of rooms and time periods. Specifically, 26 courses must be divided amongst 7 rooms and 7 time periods. A few (14) of these classes require some kind of multimedia and only a subset (4) of the rooms can accommodate this. Some obvious logical restrictions are also in place, such as one room cannot be scheduled for multiple classes at the same time, and one professor cannot be scheduled to teach multiple classes at one time.

Creating a schedule that fulfills these requirements isn't too difficult. There are tons of possible combinations of rooms and times that would work. But not every schedule is equal. So how can the best one be found? As mentioned in the introduction, an exhaustive search is possible, but highly inefficient. Perhaps a better solution is some kind of artificial intelligence that can perform a search operation, but also utilizes some heuristics to make it more efficient.

2.2 Genetic Algorithms. Genetic Algorithms (GAs), as described by [1], with reference to [2] are:

"Stochastic search algorithms based on the mechanism of natural selection and natural genetics. GA, differing from conventional search techniques, start with an initial set of random solutions called population satisfying boundary and/or system constraints to the problem. Each individual in the population is called a chromosome (or individual), representing a solution to the problem at hand. Chromosome is a string of symbols usually, but not necessarily, a binary bit string. The chromosomes evolve through successive iterations called generations. During each generation, the chromosomes are evaluated, using some measures of fitness. To create the next generation, new chromosomes, called offspring, are formed by either merging two chromosomes from current generation using a crossover operator or modifying a chromosome using a mutation operator. A new generation is formed by selection, according to the fitness values, some of the parents and offspring, and rejecting others so as to keep the population size constant. Fitter chromosomes have higher probabilities of being selected. After several generations, the algorithms converge to the best chromosome, which hopefully represents the optimum or suboptimal solution to the problem."

To my understanding, this is a highly accurate and comprehensive definition of genetic algorithms. Any future mentions of GA will refer to this definition.

So, a genetic algorithm seems like a good candidate solution to the problem. It can be used like a search algorithm and can quickly converge to a good solution to the problem.

One major detail about genetic algorithms is that you need a way to measure fitness, or determine how good a possible solution is. This is not always possible and is often very flexible with complex problems. For the college courses problem, there are many different ways to decide if one schedule is better than another. For example, you can say a schedule is good if it only uses 6 of the 7 total rooms, and even better if it only uses 5. Another possibility is the same thing but with time periods. Instead of either of these or any other simple measurement methods, my algorithm uses one that optimizes professors' happiness (defined later), and also minimizes the number of unused seats in each class.

2.3 Hypothesis. I believe that use of a GA to solve this problem will be beneficial to my success. The GA should be able to find patterns that create better schedules and optimize them as far as possible. In the end, the GA should be able to find a schedule that has the lowest possible fitness, or at least something very close.

The alternative exhaustive search would provide the same or maybe even a better result, but is far more time intensive. For each course, there are 49 possible room and time combinations. Since each combination can only be used once, the resulting formula for the total number of possibilities is 49 * 48 * … * 24. This product comes out to be 235293776179298329884380649656156160000000, or 2.3e+41. This is an unfathomably large number. For reference, one trillion is 1e+12 and the number of atoms on the Earth is 1.3e+50. Of course, with the other restrictions in place, this number will be smaller, but is unreasonably large, nonetheless. It is massively improbable to perform an exhaustive search with these many possibilities. In comparison, the GA will only contain populationSize * numberGenerations schedules. With relatively normal values, this number comes out to be around 10,000, or 1e+4.

3. Design

3.1 The Algorithm. The overall algorithm goes as follows (with definitions for each step below): (1) Generate a random list of valid schedules, called the initial population. (2) Find and save the schedule with the best fitness value in the population. (3) Generate a new population using a process of selection, crossover, and mutation on the previous population. (4) Compare each schedule in the new population to the previous best. If any schedules are found with a lower fitness, replace the current best with that one. (5) Repeat 3-4 until max generation number is reached.

3.2 Parameters. Parameters for the algorithm include populationSize, numberGenerations, crossoverProbability, mutationProbability, and selectionMethod.

populationSize is the number of chromosomes (schedules) in each population. numberGenerations is the number of generations that will be iterated over and searched through by the GA. When generating a new schedule for the next population, crossover and mutation will happen only a percentage of the time, dictated by crossoverProbability and mutationProbability. selectionMethod determines which method of selection to use, with possible values of 1 and 2, meaning tournament and elitist respectively.

3.3 Valid Schedules. In order for a schedule to be valid it must fulfill a few different requirements. These being:

– The proposed room must contain multimedia if the course requires it.
– The proposed room must contain enough seats to accommodate the number of students taking the course.
– No room/time combination can be used twice within the schedule.
– No professor may be assigned to a specific time period more than once.

In order to create a random valid schedule, we first create a blank schedule without any proposed rooms or times. Then, for each course in the schedule, pick a random room and time combination and perform some checks on it. First, check that the seating and multimedia requirements are fulfilled, then check that the combination was not already used, and finally check that the course's professor does not already have another class assigned at the proposed time. If any of these checks fail, pick a new room/time combination for that course, and try again. Repeat this process 100 times, and if it fails every time, then an impossible schedule is trying to be created, so start over from the beginning. Eventually, valid room/time combinations will be found for every course in the list, so we know the overall schedule is valid.

3.4 Selection. As mentioned earlier, there are 2 selection methods, tournament selection and elitist selection. The GA will use only one of these, depending on the selectionMethod parameter. In general, the purpose of selection methods is to choose parents for the new generation. Most of the time, you want to pick parents that have higher fitness values.

The first selection method that I implemented was tournament selection. This method is extremely simple to create and allowed me to test my GA without worrying about issues with selection. This method of selection works by choosing two random chromosomes in the population, and simply returning the one with the higher fitness value. In my implementation, if the fitness values are the same, it just returns the second one.

The other selection method is called elitist. This selection method is also fairly simple but is quite different than tournament. It works by simply choosing a random chromosome from the top X%. In my implementation, I chose X to be 50, so only the top 50% of schedules may be chosen as parents. Unlike in tournament selection, some of the worse chromosomes have no chance to pass on their genetic material, which could lead to a reliance on mutation later.

3.5 Crossover. With two parents, crossover can be performed. Crossover is done by selecting a random point in the schedule, called the crossover point. A child chromosome is returned which contains the room/time combinations from above the crossover point from the first parent, and the below combinations from the second parent. For the scheduling problem, this can lead to the creation of invalid schedules, and it can sometimes even be impossible to combine two specific parents. Because of this, my implementation simply performs normal crossover and then checks if the resulting schedule is valid. If it is not valid, repeat up to 100 times. If it is valid, return the child. If 100 unsuccessful attempts were made, just return the first parent (crossover fails, nothing happens).

3.6 Mutation. Mutation can be performed on a chromosome to introduce missing or previously killed-off genetic material. This is done by randomly changing values at one position in the chromosome. In the schedule problem, this is not straightforward, and this can go wrong in the same ways that crossover could. To address these issues, my implementation first generates a random room/time combination. Then it applies it to a randomly selected gene and the entire schedule is validated in all the ways mentioned before. If the resulting schedule is good, return it, else, like in crossover, repeat 100 times, and upon failure simply return the input chromosome.

3.7 Population Creation. The initial generation is very easy to create. All that has to be done is create valid schedules in a loop and store them in a list until the population size parameter is reached.

Subsequent generations are created differently. First, you choose two parents via the chosen selection method. Next, if the crossover chance hits, you perform crossover on the two parents and end up with one child for the next population. If crossover chance doesn't hit, just clone the first parent, and continue with it. Then, if the mutation chance hits, you mutate the child. Finally, add this child to the new population and repeat until the population size is reached.

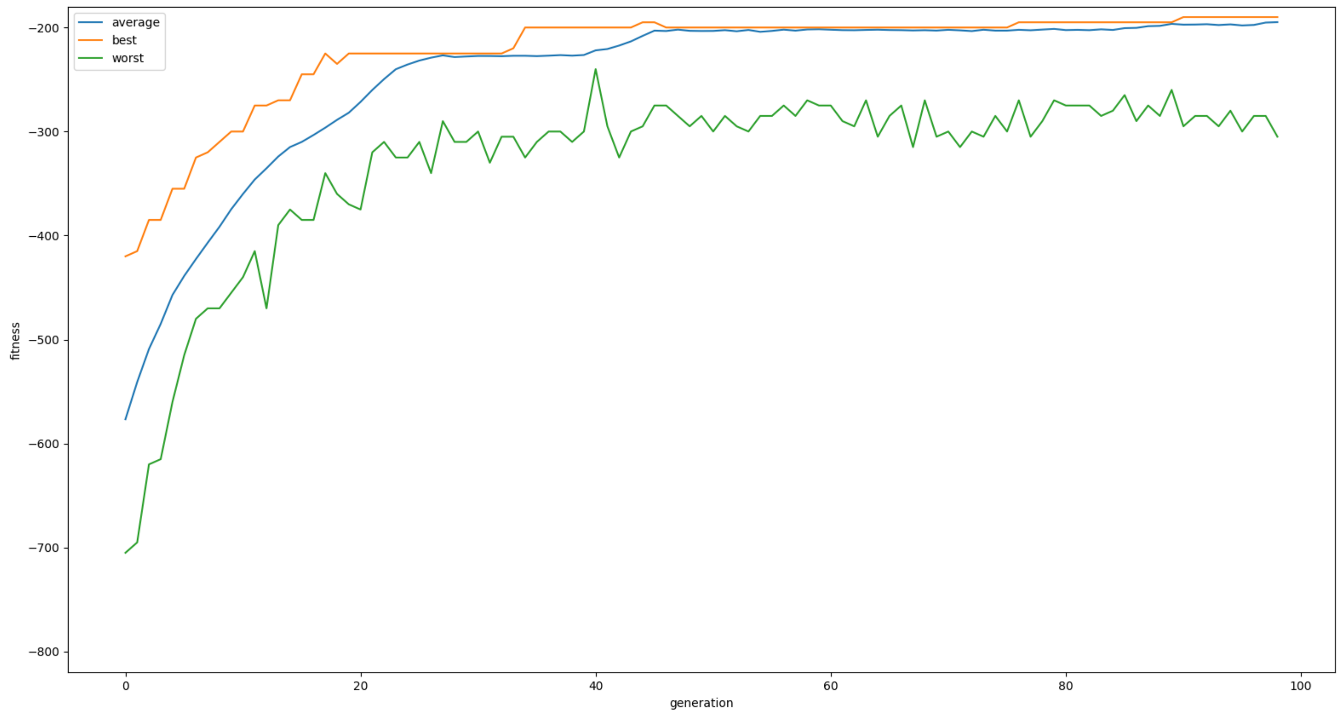
3.8 Fitness Calculation. Many parts of the algorithm depend on the fitness of a specific chromosome. As mentioned earlier, there are infinitely many ways to measure fitness, but the method I've used optimizes two things: professor happiness and unused seats. Getting a value for unused seats is easy, just subtract the room's number of seats from the course's number of students. Calculating professor happiness is a bit harder and up to interpretation. My interpretation includes three things. The first thing is a delay of three or more hours between classes. If a professor has a class at 9 and another class at 1, assuming all classes last for one hour, there is a period of three hours where they don't have anything to do. This is an extensive time period, and it is assumed that the professor does not want this, so we subtract 15 points from fitness for each occurrence in the schedule. 30 points are subtracted for a gap of 4 hours and so on. For similar reasons, we subtract 10 points for each occurrence where a professor has classes in a row (20 points for 4 classes in a row, etc.). Conversely, we provide a benefit of 5 points for each time a professor reuses a room past the first time (5 bonus points for using a room twice, 10 bonus points for using a room three times, etc.).

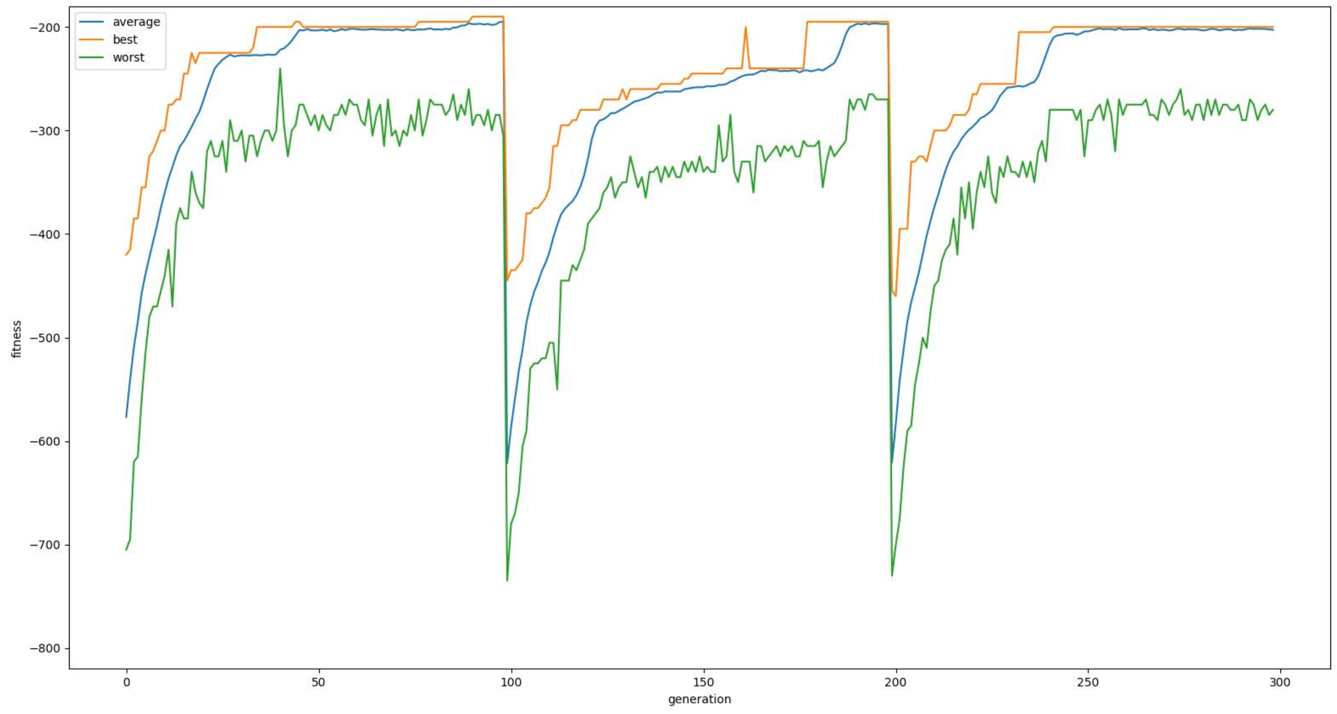
4. Results

The GA was run multiple times with the intention of finding good parameter values. The following is a summarization of the resulting data.

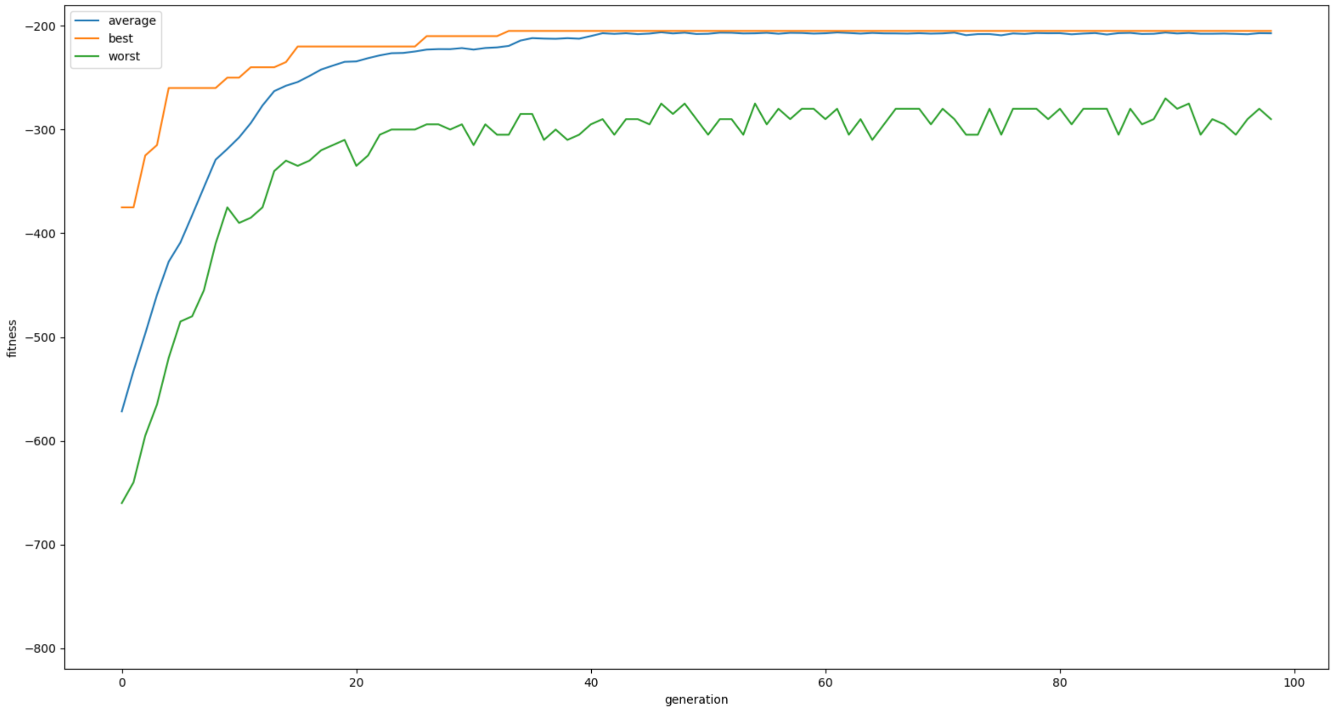
Note, for each run, populationSize was set to 250 and numberGenerations 100. The lower graph contains data from 3 runs of the GA, generating a new completely random population after 100 generations, and the upper graph is an enlarged version of the most average-looking run from the 3 below.

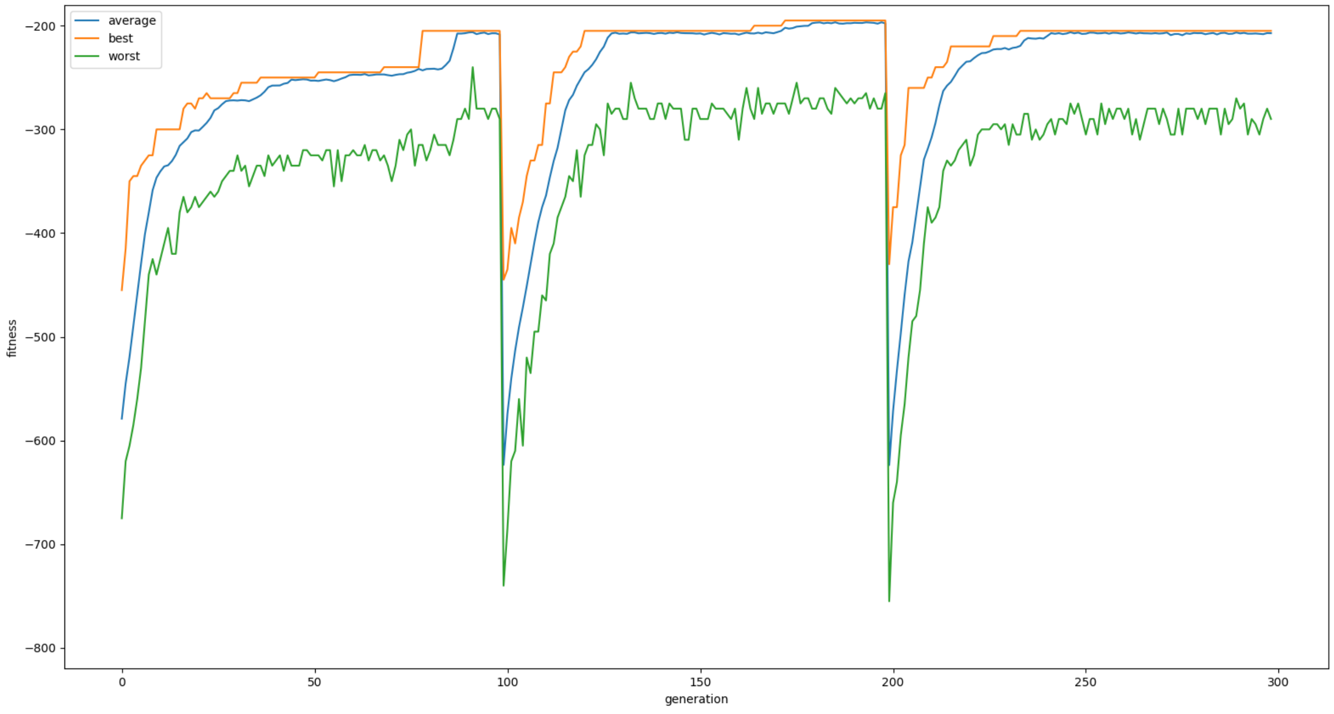
(1) Control, Tournament
50% crossover chance, 5% mutation chance, tournament selection



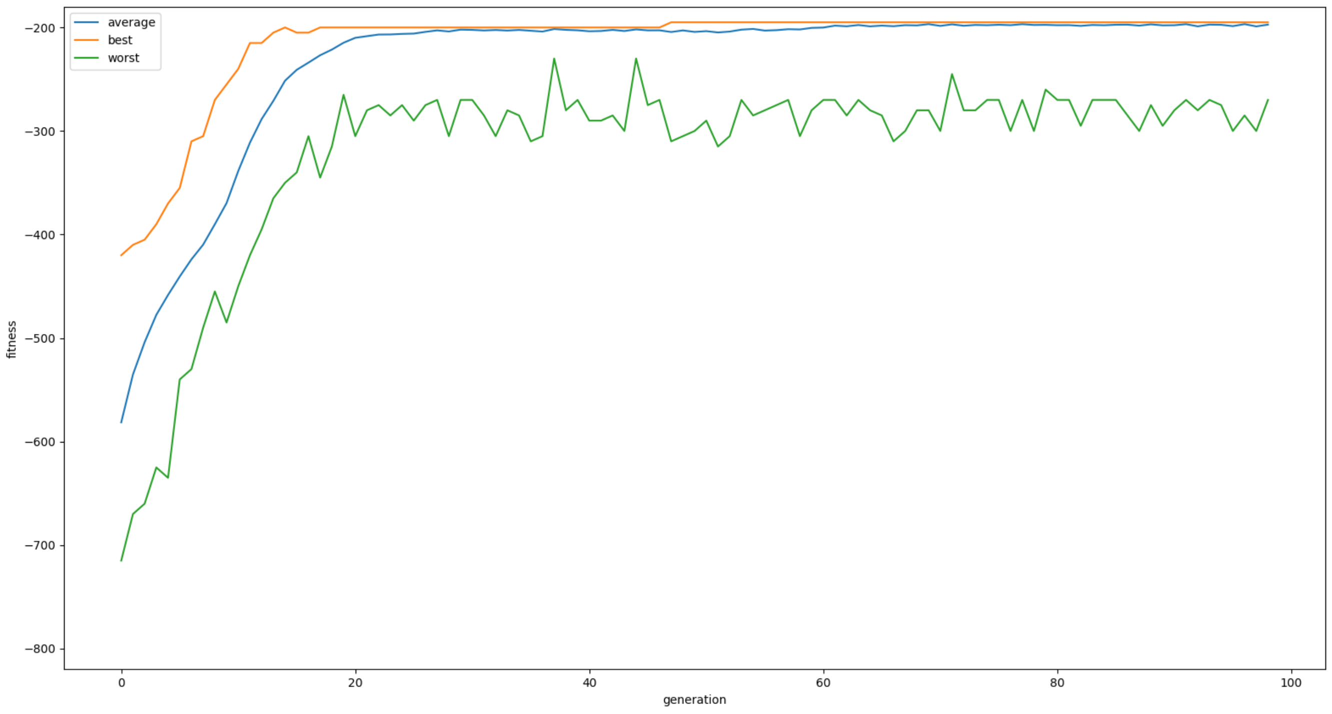


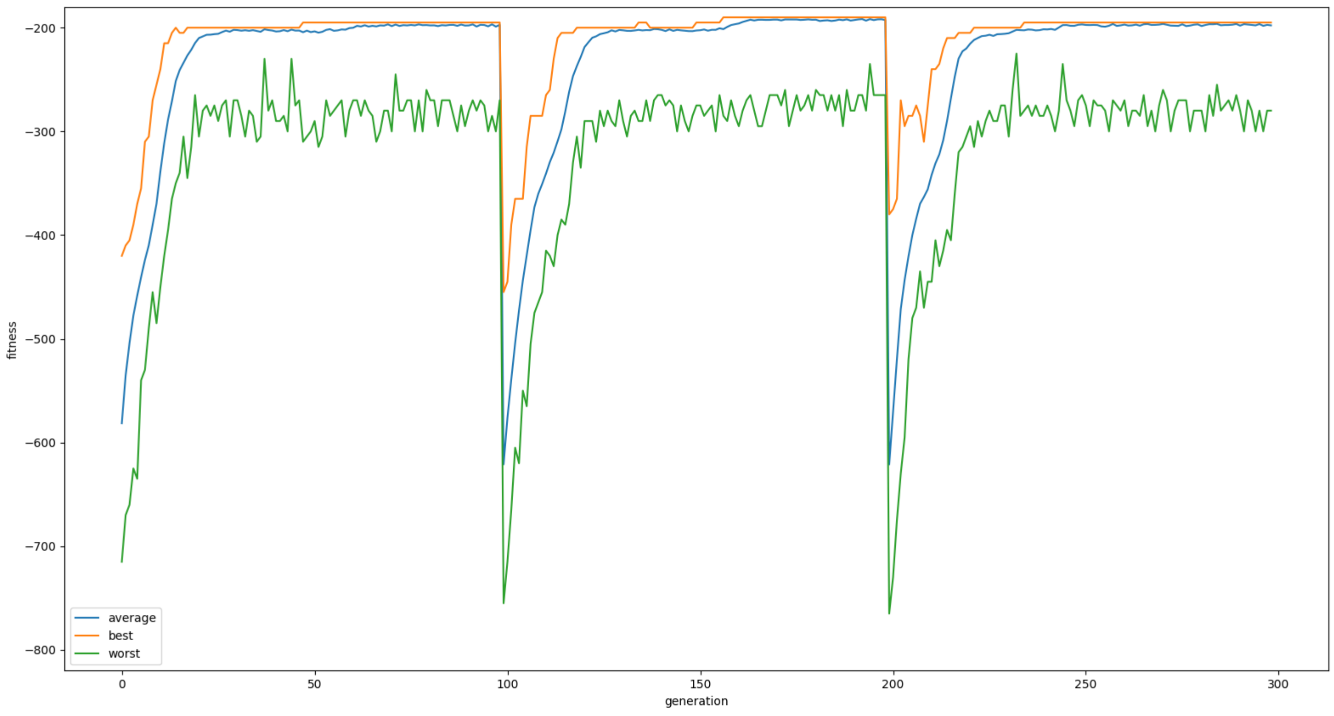
(2) Control, Elitist
50% crossover chance, 5% mutation chance, elitist selection



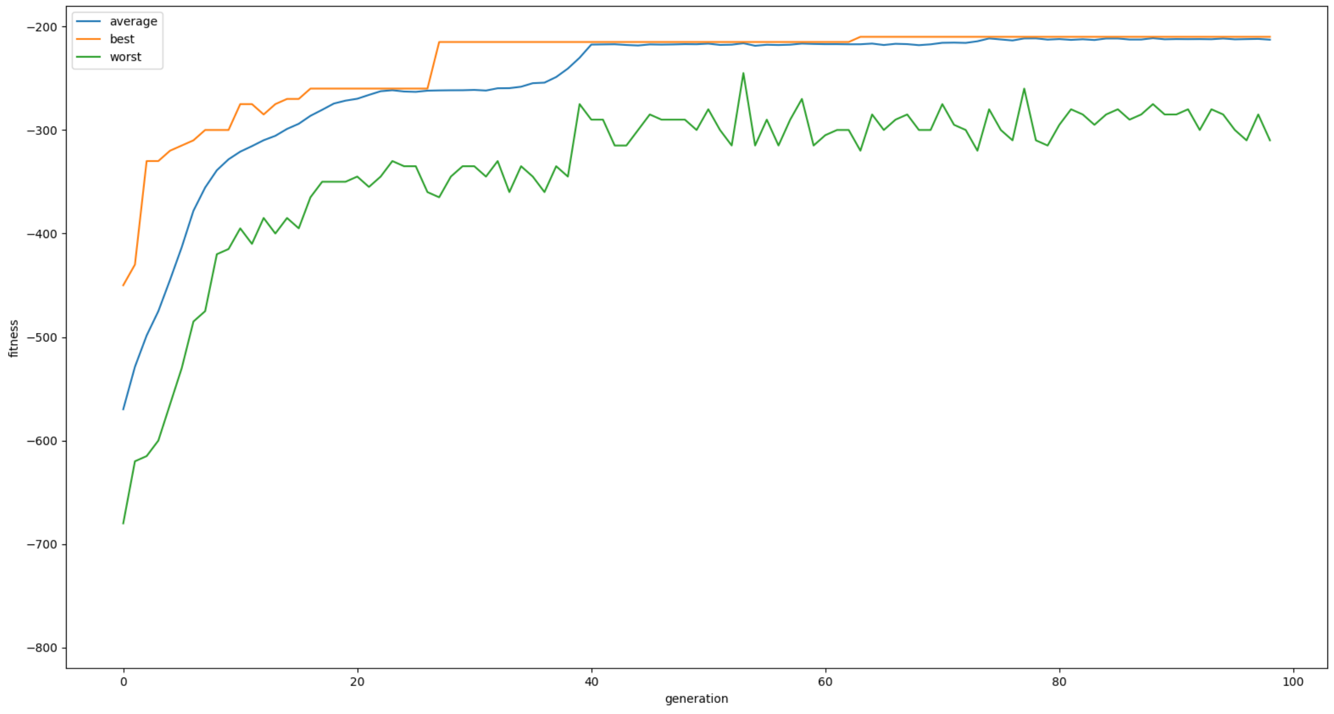


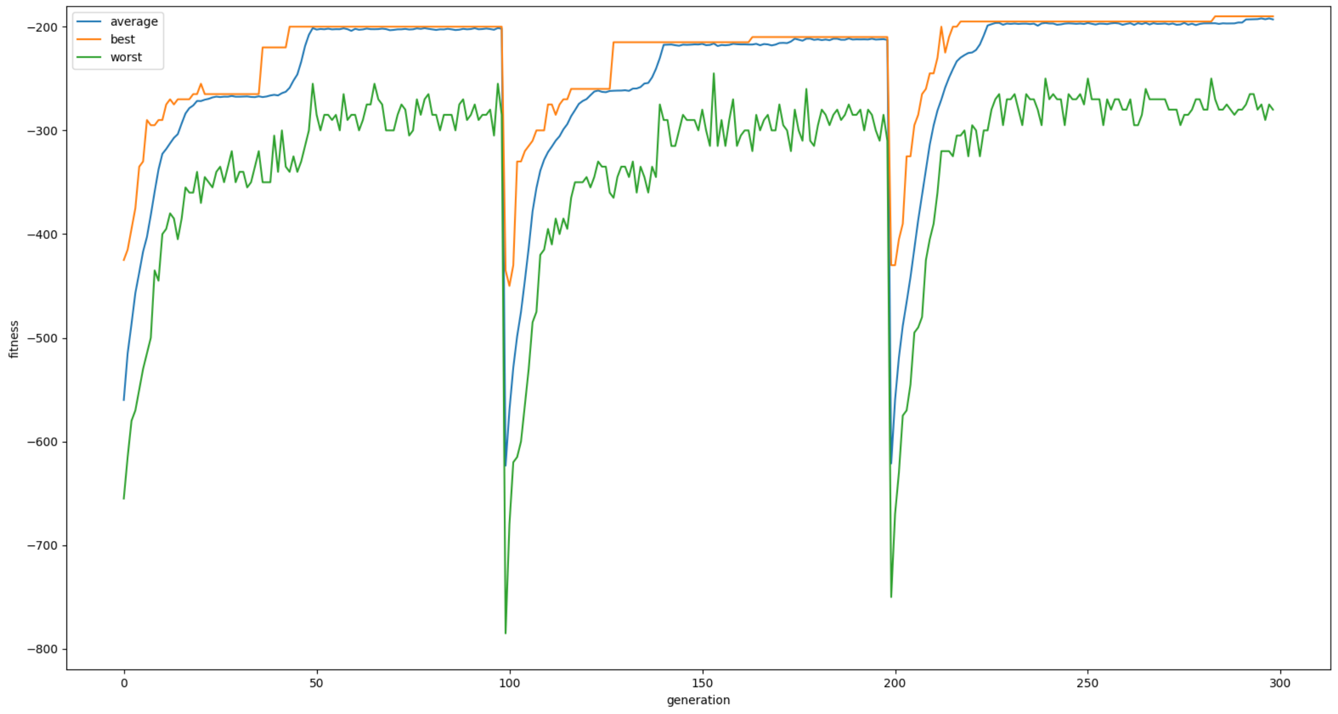
(3) High Crossover, Tournament
100% crossover chance, 5% mutation chance, tournament selection



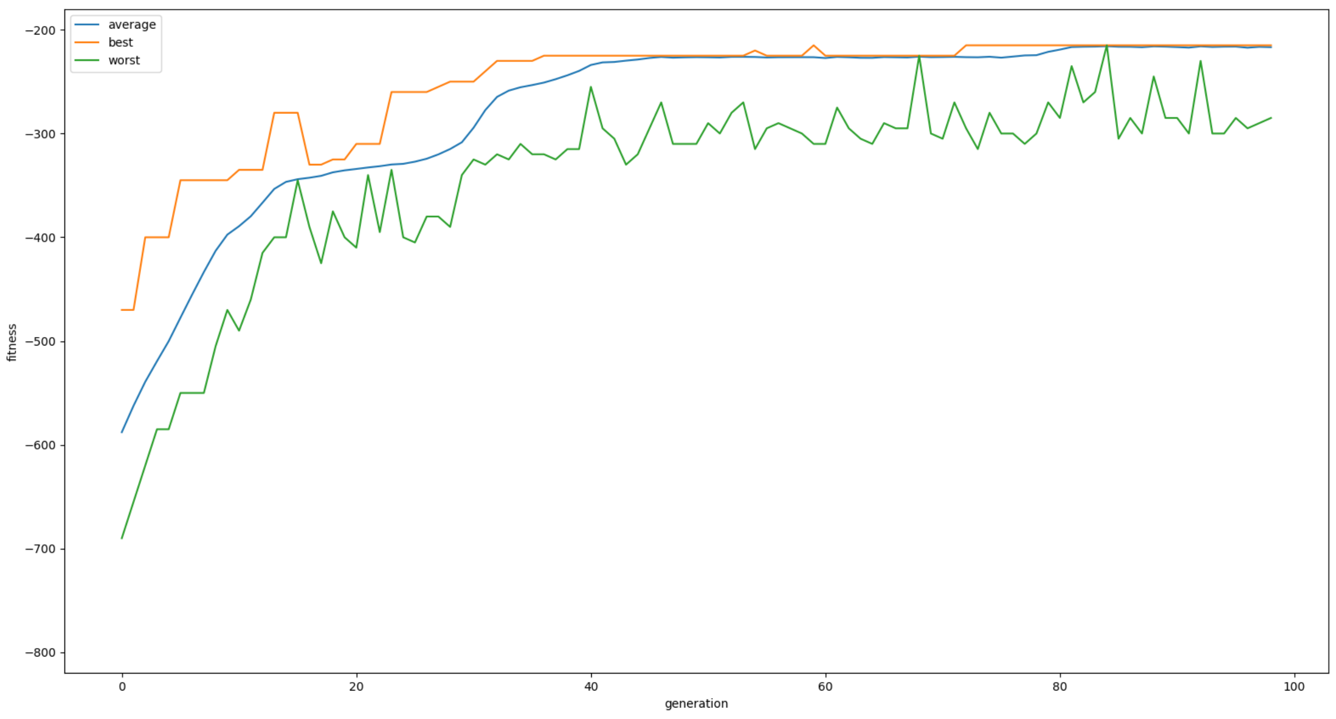


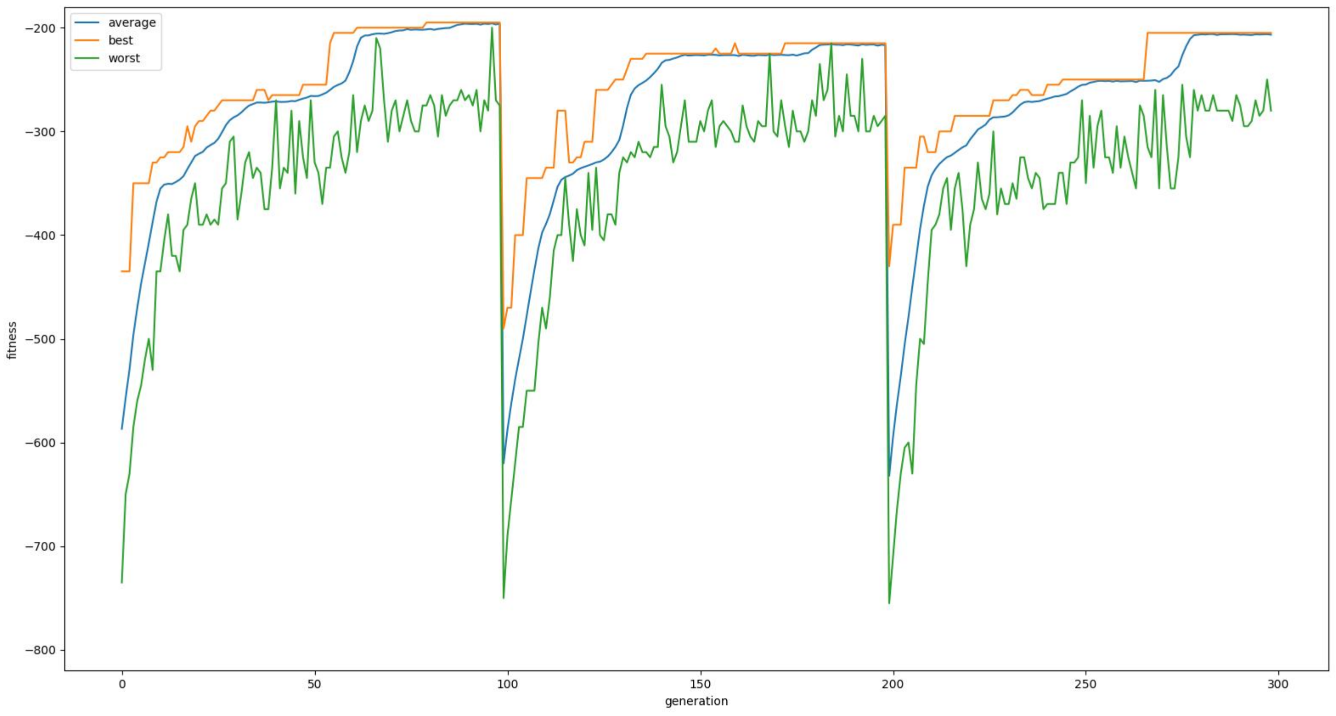
(4) High Crossover, Elitist
100% crossover chance, 5% mutation chance, elitist selection



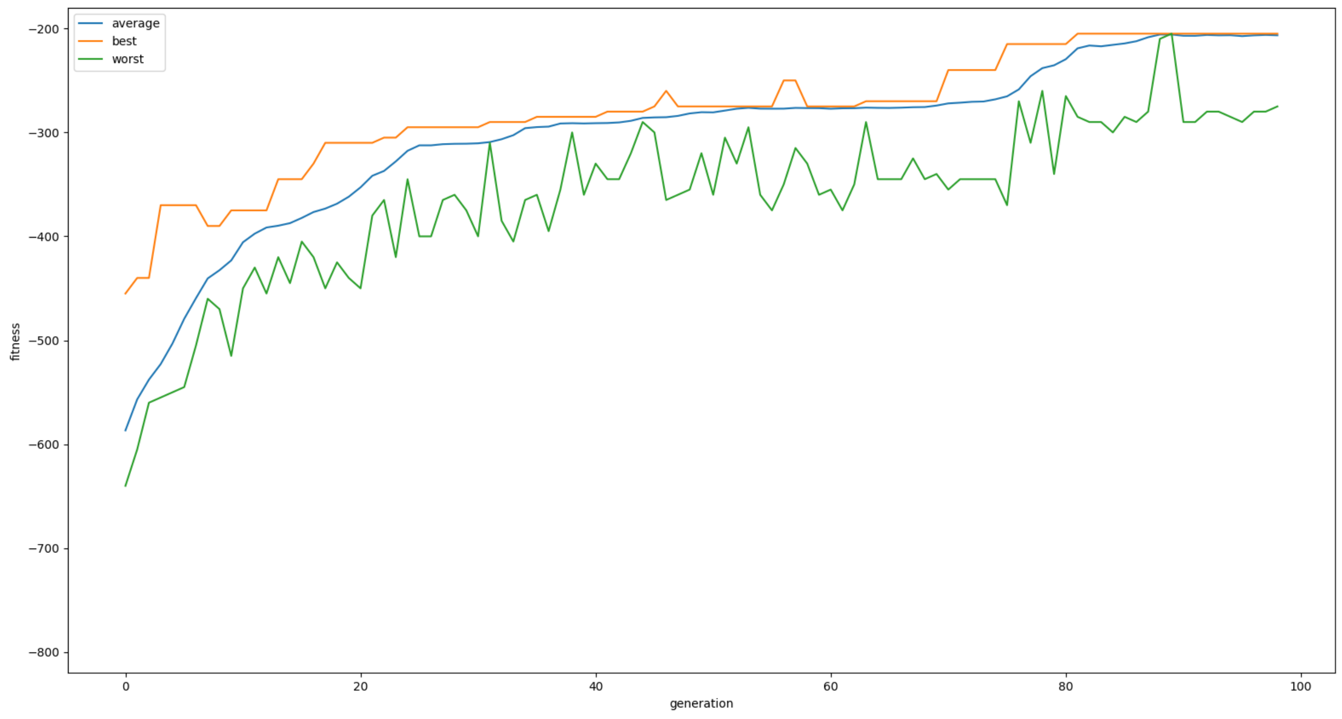


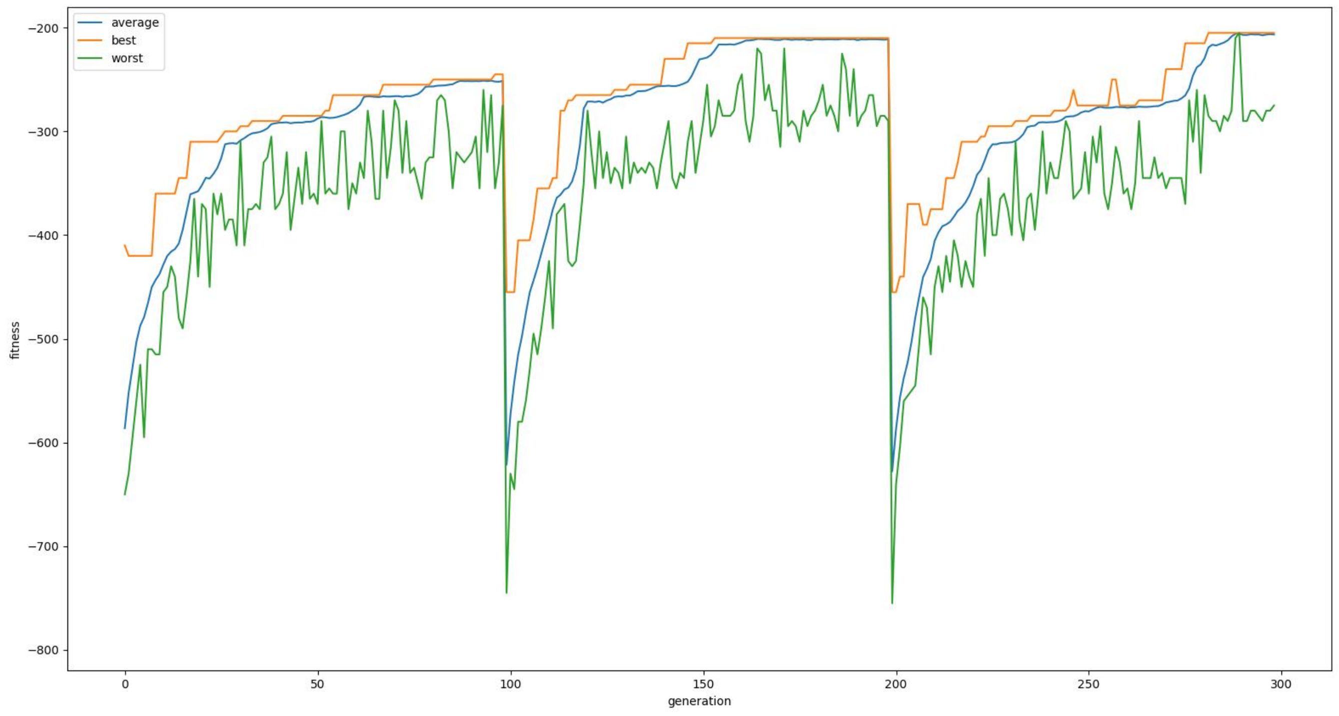
(5) Low Crossover Low Mutation, Tournament
20% crossover chance, 2% mutation chance, tournament selection



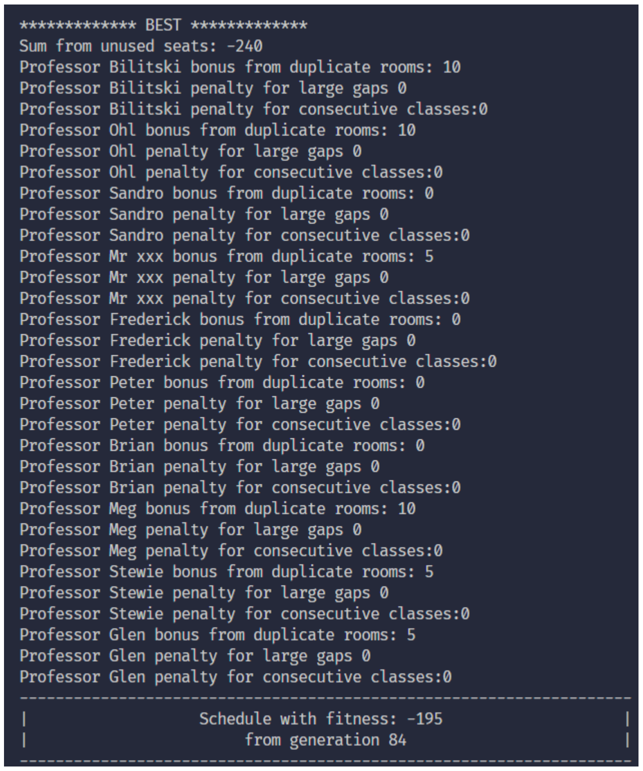


(6) Low Crossover Low Mutation, Elitist
20% crossover chance, 2% mutation chance, elitist selection





Example of fitness calculation results



```
************* BEST *************
Sum from unused seats: -240
Professor Bilitski bonus from duplicate rooms: 10
Professor Bilitski penalty for large gaps 0
Professor Bilitski penalty for consecutive classes:0
Professor Ohl bonus from duplicate rooms: 10
Professor Ohl penalty for large gaps 0
Professor Ohl penalty for consecutive classes:0
Professor Sandro bonus from duplicate rooms: 0
Professor Sandro penalty for large gaps 0
Professor Sandro penalty for consecutive classes:0
Professor Mr xxx bonus from duplicate rooms: 5
Professor Mr xxx penalty for large gaps 0
Professor Mr xxx penalty for consecutive classes:0
Professor Frederick bonus from duplicate rooms: 0
Professor Frederick penalty for large gaps 0
Professor Frederick penalty for consecutive classes:0
Professor Peter bonus from duplicate rooms: 0
Professor Peter penalty for large gaps 0
Professor Peter penalty for consecutive classes:0
Professor Brian bonus from duplicate rooms: 0
Professor Brian penalty for large gaps 0
Professor Brian penalty for consecutive classes:0
Professor Meg bonus from duplicate rooms: 10
Professor Meg penalty for large gaps 0
Professor Meg penalty for consecutive classes:0
Professor Stewie bonus from duplicate rooms: 5
Professor Stewie penalty for large gaps 0
Professor Stewie penalty for consecutive classes:0
Professor Glen bonus from duplicate rooms: 5
Professor Glen penalty for large gaps 0
Professor Glen penalty for consecutive classes:0
--------------------------------------------------------------------
|                 Schedule with fitness: -195                      |
|                      from generation 84                          |
--------------------------------------------------------------------
```

5. Analysis

Parameter set number (3); "High Crossover Tournament" is the winner. 100% Crossover chance, 5% mutation chance, and tournament selection seems to be the best combination of parameters to solve the problem.

In general, it seemed like tournament selection outperformed elitist selection. The control test had similar results for both methods, but the other two tests were clearly won by tournament selection. I think this could be due to some good genes being discarded early by the elitist selection. Schedules could have some perfect genes but also a lot of bad genes and the whole thing would have no chance of becoming a parent. If this was the only chromosome with those perfect genes, then it becomes extinct in the population, and will only come back later through mutation.

Another generalization is that high crossover outperformed the control tests, and the control tests outperformed the low mutation/low crossover tests. From my implementation, both crossover and mutation, even when landing their chance, will not always actually happen. I think this impacted the third set of tests the most because they already have a lower chance to perform these operations. Together, this leads to an outcome where relatively little amounts of change are happening between generations, and therefore progress towards the optimal solution is slower. I suspect that the high crossover tests outperformed the control tests for the same reason.

There are some patterns that are common to all of the tests. One, all of the lines loosely resemble a logarithmic function, that is, they begin incrementing steeply and flatten out over time. This is typical behavior of GAs, so it is evidence that my implementations were correct. Also, there are a lot of small steep jumps. This can most easily be seen in the worst line but is present in all of them. This is because even one single mutation can completely change the fitness of the schedule, for example, changing a room from one with 40 seats to one with 100 seats not only decreases fitness by 60 for unused seats, but could also make it so a professor does not receive the bonus for consecutive room usage anymore, another 5-point decrement. With other similar situations, large jumps in fitness are actually quite common, and small jumps rarer.

6. Conclusion

6.1 The Algorithm. A genetic algorithm I think was a perfect solution to this problem. It was able to consistently produce a good schedule, even with suboptimal parameters. It even runs very quickly, only a second or two on average. I think genetic algorithms would be very capable to solve many other kinds of problems too. The issue is that they cannot always be used. When there is no clear way to measure fitness of a solution, genetic algorithms do not work at all. Even in this problem, we had to craft some kind of measurement method that we think might give us good results. There are many other methods out there, some of which could result in even better schedules being created. For this reason, GAs are not useful in every situation, but where fitness is easily calculated, I think they are very efficient. Again, from my results, I think it is good to use a selection method that allows some of the worse chromosomes to create children, and also to have a high crossover rate and medium mutation rate, meaning around 100% and 5% respectively. Population size and number of generations are inversely related, and dependent on the specific problem, so optimal values for these don't really exist. It's best to just try things and see what performs well. In general, higher values for both of these will produce a better solution, but will also take longer to run, and might even cause most of your resource use to be fruitless. Good values should be located through trial and error.

6.2 Problems and Improvements. My algorithm I don't think was perfect at all. For one, crossover and mutation work kind of 'hopefully'. They make a random change and hope that it results in a valid schedule, and if not, they repeat the process 100 times. This leads to longer run-times than necessary, and more magic numbers that could be parameters and overall introduces more complexity. There is probably some way to implement both of these methods without ‘hoping’ and is more efficient, that is something I would improve if continuing on this project.

Another issue with my algorithm was fitness-proportional selection. I initially was going to use this instead of the elitist selection method, but was getting terrible results. It ended with a peak of -300 fitness sometimes, even after 10x more generations and all the other parameters the same. My implementation attempt was similar to the weighted roulette wheel, but instead of using fitness for tickets, I used the chromosomes index in a sorted list. This means, the worst schedule gets 1 ticket, next worse gets 2 tickets, etc.… On paper this seemed liked a very good selection method. Bad schedules still have a chance to produce children, but it is very low (~10). Medium schedules have a medium chance (~30%) to produce children and great schedules have a high chance (~60%). But, in practice, this method did not work at all. This may have been a coding fault of mine but either way, this is definitely something that could have been improved on. A third selection method would provide a lot more value in comparing the tests.

I actually think the fitness calculation method that was used is pretty good. It is able to optimize what I called ‘professor happiness’ very well. If that was the only thing we were optimizing for, many great solutions would be found, all equal and different in some ways. By introducing penalty for unused seats, it kind of just narrowed down that set of great solutions even further. All of the best solutions that my algorithm found resulted in situations where nearly no penalties aside from unused seats were issued, and a high number of bonuses existed. So, it was still finding schedules very good for professor happiness (as can be seen in the fitness calculation example under 4. Results), but with an additional restriction to improve the schedule in some other way. Depending on what you want to achieve with the schedule, fitness calculation can be done in a ton of ways. Of course, one way to improve the GA, if you believe that good schedules minimize the number of unique rooms used, is to base fitness calculation on that. Improving the fitness calculation method is almost always determined by some opinion, and therefore it is hard to say whether the one I used is actually good or not.

In summary, I found that a genetic algorithm provided a very good solution to my problem. It found a good solution very quickly and is easily adjustable to meet different requirements.

References

[1]. Gen, Mitsuo., Runwei. Cheng, and Lin. Lin. Network Models and Optimization Multiobjective Genetic Algorithm Approach. 1st ed. 2008. London: Springer London, 2008. Web. https://pitt.primo.exlibrisgroup.com/permalink/01PITT_INST/e8h8hp/alma9998537992506236

[2]. Goldberg, D. (1989). Genetic Algorithms in Search, Optimization and Machine Learning, Reading, MA: Addison-Wesley.