AI Assignment 1

Aaron Brennan (19420624)

Part A) *I will just include things to note for this part.*

1) Selecting the single best parent and then mutating (no crossover) evolves quicker than roulette wheel parent selection and crossover for simple problems?

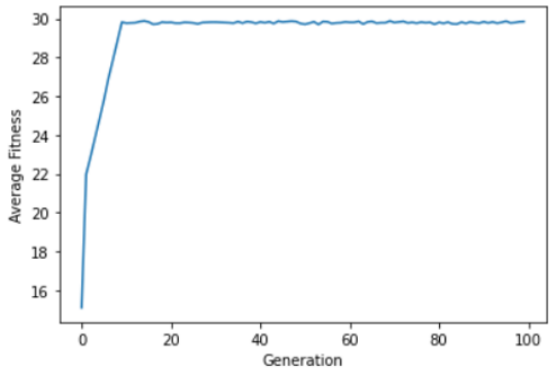
For part (i) and (ii) It is interesting to note that simply choosing the best performing ('fittest') member of the population and randomly mutating that specific sample to produce the children of the next generation very quickly produces a perfectly evolved population. The next generations mutations will randomly produce a sample slightly better then the best of the previous generation and the process continues.

In comparison to the roulette wheel parent selection and crossover approach which takes more generations to reach an optimal final population. However, this approach could be more useful when the problem is more challenging and requires more variation in the population to find a good solution.

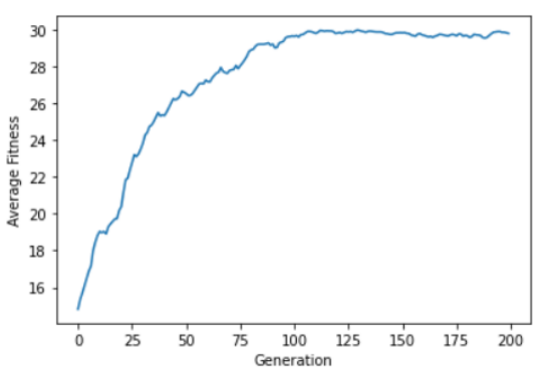
Here are graphs showing the fitness over time for the "just mutate" approach and the "roulette wheel and crossover" approach for part (i) and (ii):

Part A (i) "only mutation of the best sample" vs "roulette wheel selection & crossover"

Only mutating the best sample:

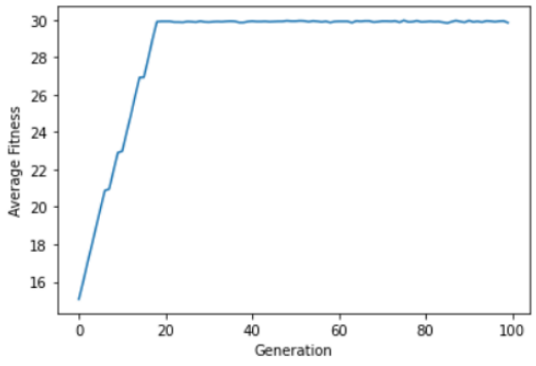


Roulette wheel selection and crossover:

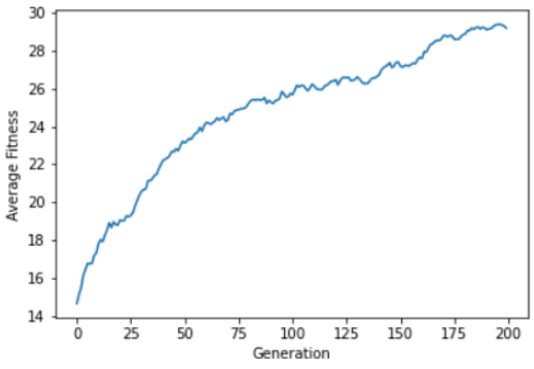


Part A (ii) "only mutation of the best sample" vs "roulette wheel selection & crossover"

Only mutating the best sample:



Roulette wheel selection and crossover:



2) For part (iii) it basically impossible to achieve a population with the optimal solution

Unless we get a “lucky” (1/2^30) random initialisation of a string with all 0’s we will never reach this optimal solution.

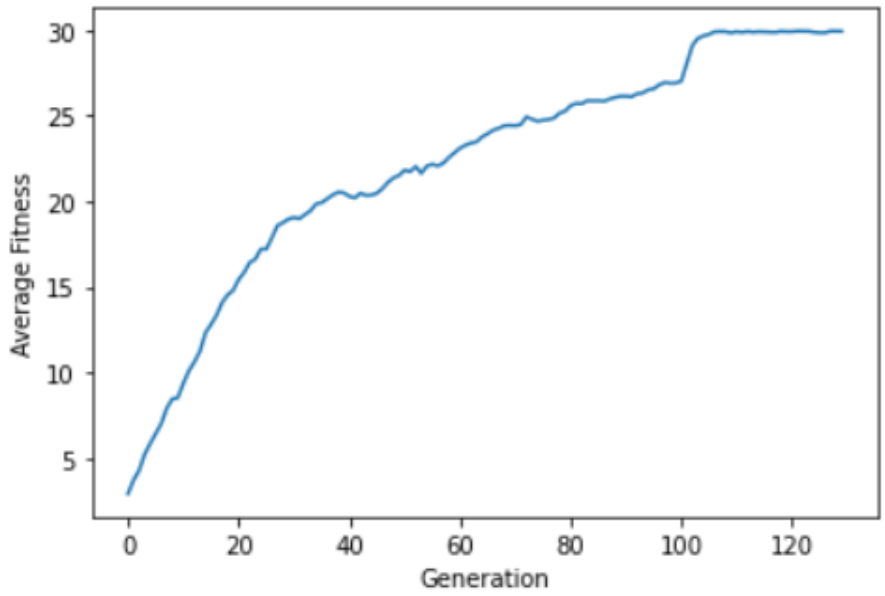
3) Algorithm explained for part (iv)

This problem is more difficult to evolve then the previous parts so I have used a few tricks to help the population to evolve to an optimal solution.

Step 1: For the first 100 generations the population involves in a typical way that involves roulette wheel parent selection, crossover, and random mutations. This typically achieves a decent result of 25/30 (25 out of the 30 positions have the correct number corresponding to the target string)

Step 2: After 100 generations (100-130) I change the parent selection such that one of the parents is always the “fittest” and the other parent is selected by roulette wheel. This reduces the variability in the population but helps it to hone in on an optimal solution. Usually this causes a sudden spike in fitness after 100 generations which helps it typically achieve a result of 28/30. Crossover and random mutation are still used for these 30 generations.

Step 3: By combining steps 1 and 2 we usually get a good but sub-optimal score, however roughly 10-20% of the time this method achieves an optimal score. Therefore to guarantee an optimal score on one iteration, I evolve 20 different populations on separate “islands” and when they are all finished evolving I choose the island that had the best average fitness in the end. This always achieves 30/30.



^^ Notice the spike in fitness after switching the parent selection method after 100 generations

Part B)

What is a sample in the population?

A list of length n (n students looking for supervisors, n=46 for all students) where each element in the list describes what student has which supervisor e.g. [[1, 14], [2, 12], [3, 5] …] says student 1 has supervisor 14, student 2 has supervisor 12, student 3 has supervisor 5 and so on. Measures are made in the code to ensure that no supervisor is taking on too many students for each sample.

How to make sure supervisors don't go beyond their capacity:

A list is created of all the supervisors, the supervisor is added to the list as many times are their capacity for students. E.g. if a supervisor has a capacity of 2 they are added twice to the supervisor list.

When performing different operation such as crossover and mutation a supervisors_remaining list is kept which keeps track of the supervisors in the supervisors list who have not yet been given a student. Once a supervisor has been given a student they are removed from the supervisors_remaining list.

Note however that just because a supervisor has been giving a student that doesn't necessarily mean that he is no longer not in the supervisors_remaining list. E.g. Supervisor 1 has a capacity of 3, even when they are given a student there are still 2 more instances of supervisor 1 in the supervisors_remaining list.

Operators used:

Mutate: The mutate operator is slightly different depending on whether or not the full 46 students are used or not

When all students are used: Swap two of the student's supervisors e.g. [[1, 14], [2, 12], [3, 5] …] now becomes [[1, 5], [2, 12], [3, 14] …] students 1 and 3 have swapped supervisors.

When a subset of the students are used: In this case there are supervisors who have the capacity for more students (supervisors_remaining list above) but aren't being used. Randomly one of the students loses their current supervisor and is replaced by one of the supervisors who previously wasn't being used. The supervisor that was removed is then added to the supervisors_remaining list

Roulette parent selection: The samples in the population with higher fitness scores (low preference score) have more likelihood of being picked to be a parent. Each fitness value is converted to a probability, and a parent is picked randomly based on those probabilities.

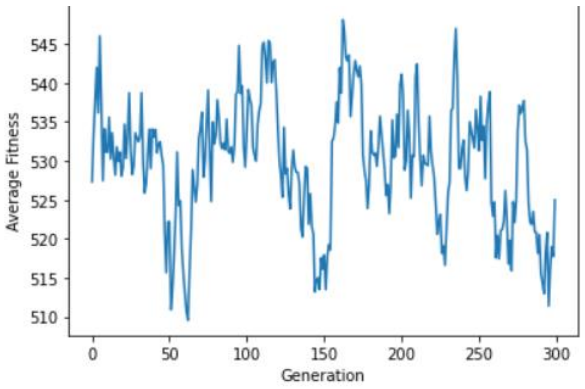
Best parent selection: The sample with the best fitness (lowest preference score) in the population is chosen.

Crossover:

1. For each (students picked in random order) student a supervisor is given by either parent 1 or 2 (randomly picked).
2. If that supervisor is already overloaded, we see if the other parent's supervisor is available instead. If not then we skip that student for the time being and note that they are missing a supervisor.
3. When this is done there will be a few students remaining without supervisors, the remaining supervisors with capacity available are then randomly assigned to these students.

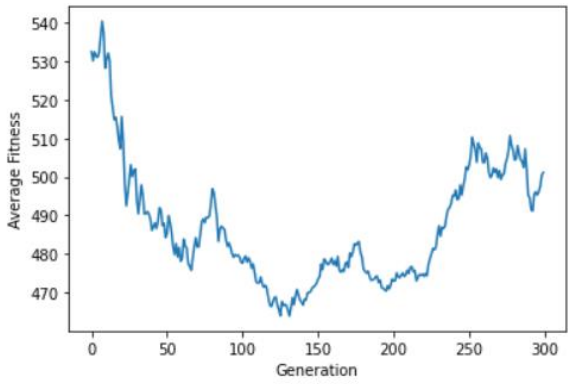
Parent selection method/results:

The crossover method I have created results in many mutations which could cause the population to not evolve correctly. Between the roulette wheel parent selection, the crossover method and the random mutation operator too much information is lost from one generation to the next. Here is graph showing the evolution of the population using the roulette wheel parent selection and crossover:



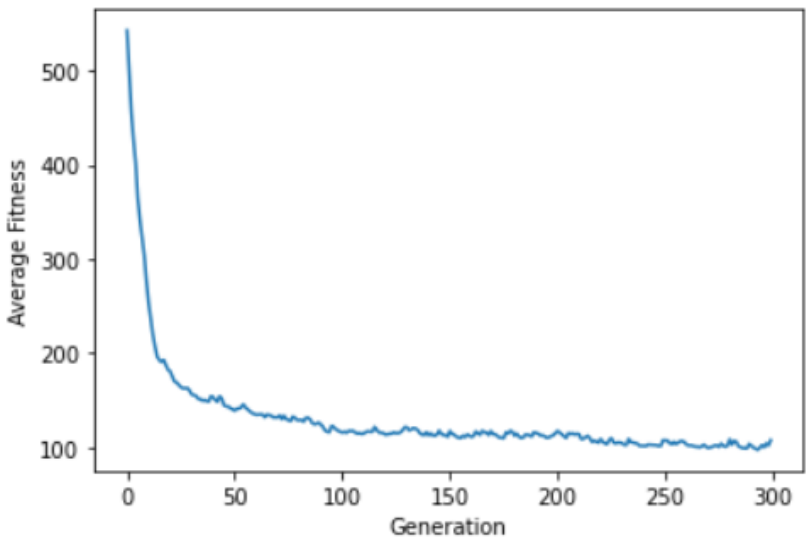
choosing both parents by roulette is not optimal

Then I tested how the population would evolve through the "mutating only the best sample" method from part A (sample with best fitness is chosen and all children are mutations from that single sample). By only picking the best sample each time there is not enough variation within the population to achieve good results. Here is a graph showing the evolution of the population using the "mutating only the best sample approach":



Not enough variation to achieve good results

Finally I combined both ideas by using the roulette wheel selection for one of the parents and the "best sample" in the population as the other parent. The crossover operator was then used on both parents along with random mutations. By picking the best parent after each generation the best student-supervisor mapping's could be persevered through the generations while the roulette wheel selection and crossover helped to add variations into the population. The final population after 300 generations got an average score of roughly 100 which is equivalent to each student (on average) getting their 2.1st preference of supervisor. Here is the graph showing the evolution of the population using this combination method to select parents:



Fitness function:

A dictionary was created which stored the preferences that each student had for each supervisor. For each sample in the population the dictionary was called for each student in that sample to get the preference/score. This was then added to a total score variable to get the total score of that sample.

To make it easy for myself I used 2 separate fitness functions. The first fitness function was called “actual fitness function” which I have described above. This was used for graphing and interpreting the results. The lower the value of this function the better (lower preferences)

Then to keep the genetic algorithm consist with part A, I created another fitness function which has the inverse of the “actual fitness function”. For each preference I got in the dictionary I took that value away from 23. E.g. preference 1 had a score of 22, preference 22 had a score of 1. This way the higher the score the fitter the sample and this function could then be used as part of the genetic algorithm.