COS 314 Assignment 3 Report

Author: JS Potgieter - u18068911

Introduction:

The aim of this report is to provide the implementation specifications and some test results for a soduko solver program, using a genetic algorithm. No external libraries were used for the implementation of the genetic algorithm, all parts are directly constructed an implemented - these parts are indicated with comments in the program code.

The program can be found in the uploaded folder as 'soduko.py' and requires the indicated parameters on execution, as given by the assignment specification.

Most ideas for the construction of the chromosome and the genetic operators were acquired from the journals included, I do not take any credit for these ideas - only the implementation thereof.

Representation of the chromosome and genes:

Gene: A single digit from 1-9, present as a single block in the sodoku.

Chromosome: Has $9 \times 9 = 81$ total genes, broken up into the 9 subgrids of the soduko from left->right and top->bottom. Within a subgrid, genes are also in the same order of left->right and top->bottom.

Constraints and initial population generation:

For the constraints of the soduko problem at hand, we chose to fix subgrids to always have all the digits 1-9 uniquely - which serves as motivation for the choice of chromosome construction. The other two constraints (row and colums containing all digits 1-9 uniquely) are then optimised through the choice of fitness function and mutations.

Taking this into account, initial population individuals were constructed by starting with the given sodoku problem and then filling in subgrids with the required missing digits (since this was chosen to be fixed), in a random order.

The fixed digits given by the starting problem are thus immediately fixed in individuals and are also never changed by any genetic operations.

Fitness function:

The fitness of an individual was evaluated by iterating through all rows and columns, for each iteration the fitness score (starting at 0) is increased by 1 for every digit missing in the row/column and increased by the number of times every digit 1-9 appears more than once in the row/column - this choice is motivated by the explanations of the contraints above.

It should also be clear that the motive is thus to minimize the fitness function, where the full solution will have a fitness score of zero.

Mating selection:

The elitism selection algorithm given by the supplied journals was directly used - meaning that the fittest individuals had the highest chance of mating, but there is still a non-zero chance for all individuals in the current population to also reproduce.

Genetic operators and parameters:

Children were first constructed through crossover on the chosen parents and then secondly mutation was applied to each child.

Crossover operator: Firstly 4 indexes of subgrids in the chromosome are chosen at random, allowing for the same index to be chosen more than once. For each unique chosen index, the subgrids at the indexes are swpped between prents to form the children. Thus, 1-4 subgrids can be swapped during this operation.

Mutation operator: The following mutation steps were applied to every subgrid of the individual. Firstly a random number of mutations were chosen between 1-5 (both included), for each mutation step 2 digits within the subgrid were randomly chosen to be swapped. If these chosen digits were not fixed and would appear a maximum of 3 times in the appropriate rows/columns after the swap collectively, the swap was performed.

Other parameters:

- Popultion size: 11
- n_gen: 2000 (stop condition also implemented to halt when solution is found, thus this is only maximum)

Testing and results:

Three tests were conducted for the following sodoku problems: s02a, s04a, s06a, s12a (as indicated on the given data website).

Below we present graphs that show the change in average fitness of the population and minimum obtained fitness as a function of increasing generations.

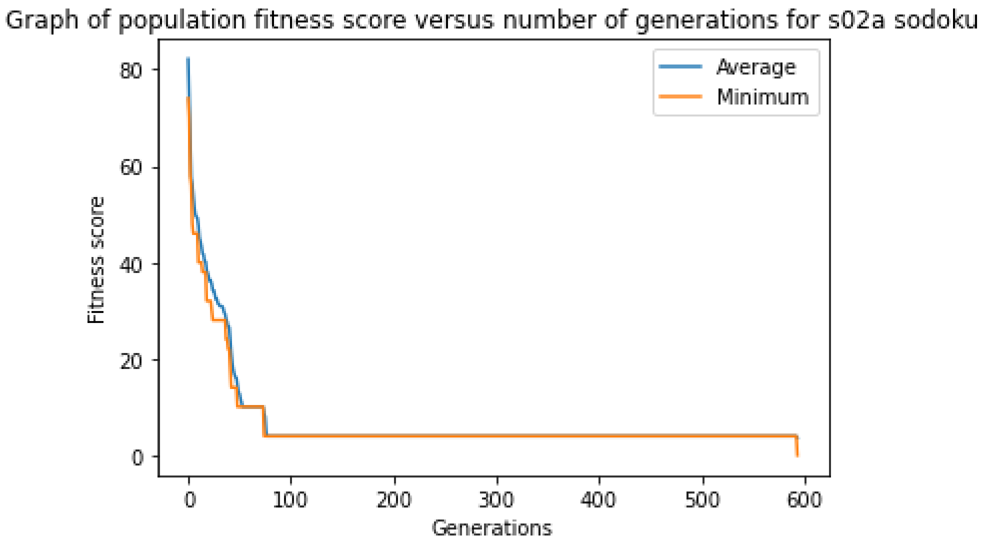
s02a:

In [25]:

```
import numpy as np
import matplotlib.pyplot as plt

s02a_average = np.loadtxt("./s02a/Average_fitness.txt")
s02a_min = np.loadtxt("./s02a/Min_fitness.txt")
generations = np.arange(0, len(s02a_average))

plt.plot(generations, s02a_average, label="Average")
plt.plot(generations, s02a_min, label="Minimum")
plt.xlabel('Generations')
plt.ylabel('Fitness score')
plt.title('Graph of population fitness score versus number of generations for s02a sodoku')
plt.legend()
plt.show()
print("Final average: ", s02a_average[-1])
print("Final minimum: ", s02a_min[-1])
```



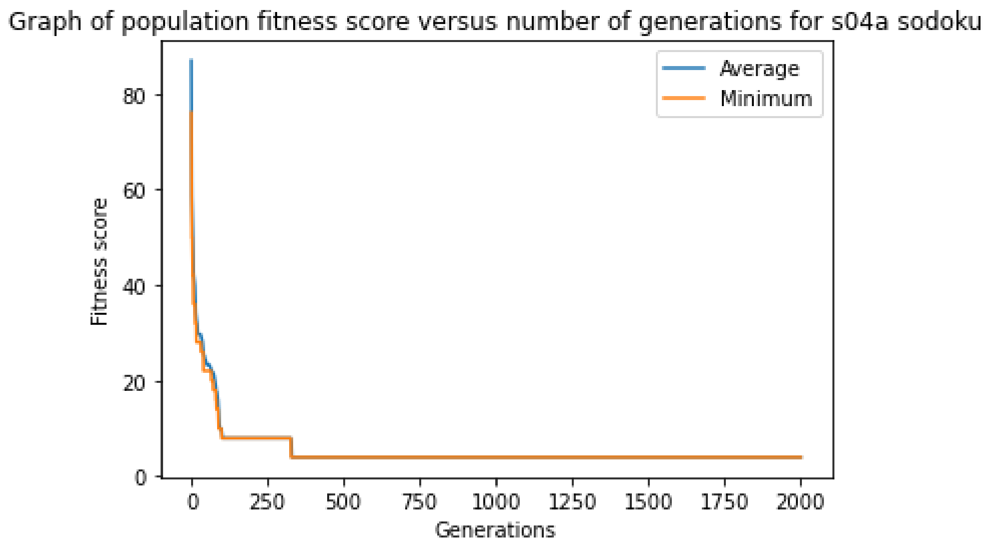
```
Final average:  3.6363636363636362
Final minimum:  0.0
```

s04a:

In [26]:

```
s04a_average = np.loadtxt("./s04a/Average_fitness.txt")
s04a_min = np.loadtxt("./s04a/Min_fitness.txt")
generations = np.arange(0, len(s04a_average))

plt.plot(generations, s04a_average, label="Average")
plt.plot(generations, s04a_min, label="Minimum")
plt.xlabel('Generations')
plt.ylabel('Fitness score')
plt.title('Graph of population fitness score versus number of generations for s04a sodoku')
plt.legend()
plt.show()
print("Final average: ", s04a_average[-1])
print("Final minimum: ", s04a_min[-1])
```



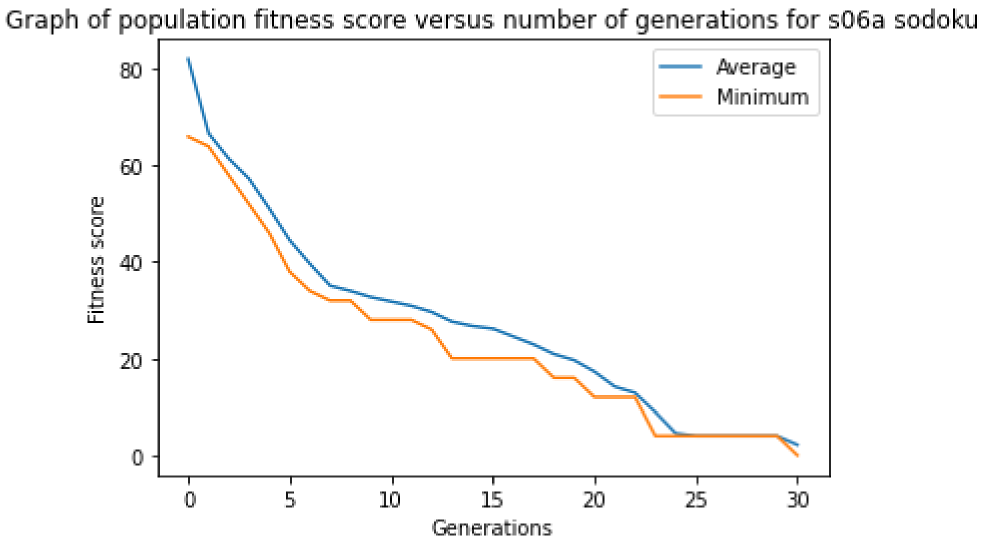
```
Final average:  4.0
Final minimum:  4.0
```

s06a:

In [27]:

```
s06a_average = np.loadtxt("./s06a/Average_fitness.txt")
s06a_min = np.loadtxt("./s06a/Min_fitness.txt")
generations = np.arange(0, len(s06a_average))

plt.plot(generations, s06a_average, label="Average")
plt.plot(generations, s06a_min, label="Minimum")
plt.xlabel('Generations')
plt.ylabel('Fitness score')
plt.title('Graph of population fitness score versus number of generations for s06a sodoku')
plt.legend()
plt.show()
print("Final average: ", s06a_average[-1])
print("Final minimum: ", s06a_min[-1])
```



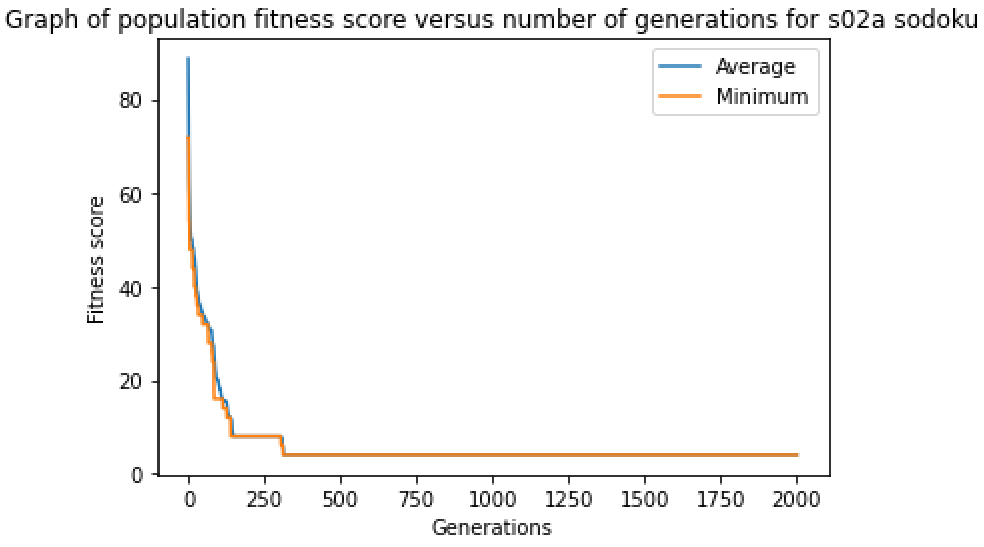
```
Final average:  2.1818181818181817
Final minimum:  0.0
```

s12a:

In [28]:

```
s12a_average = np.loadtxt("./s12a/Average_fitness.txt")
s12a_min = np.loadtxt("./s12a/Min_fitness.txt")
generations = np.arange(0, len(s12a_average))

plt.plot(generations, s12a_average, label="Average")
plt.plot(generations, s12a_min, label="Minimum")
plt.xlabel('Generations')
plt.ylabel('Fitness score')
plt.title('Graph of population fitness score versus number of generations for s02a sodoku')
plt.legend()
plt.show()
print("Final average: ", s12a_average[-1])
print("Final minimum: ", s12a_min[-1])
```



```
Final average:  4.0
Final minimum:  4.0
```

Observations and conclusion:

Only two of the sodoku problems we tested our genetic algorithm on were able to find a complete solution - at around generations 600 for s02a and generation 30 for s06a (most likely this quickly due to some well chanced mutations). The other two tests got stuck in a local minima with a fitness score of 4. A reason for this could be due to our genetic algorithm lacking aging, where the fitness score of a recurring best solution gets incremented. Our algorithm also lacks the ability to reinitialize the population at a certain point of being stuck on a single local minima. It is not presented in the results, but through testing I realised that the program can finish on different outcomes for different identical runs on a problem - sometimes it finds the solution quickly with a well initialized population and mutations or it gets stuck on a local minima. This supports the idea that both aging and the ability to reinitialize the population could potentially help the algorithm in getting out of a local minima.

In [ ]: