Coursework B: Genetic Algorithms

EXERCISE 1 – Simple GA optimiser to find a number specified in least number of iterations:

A GA was made by modifying the provided code, the function of the GA was to act as an optimiser to find the specified number in the shortest number of iterations. The code was modified such that it could find any unsigned 32-bit integer (0 to 4.3 billion) and as such all-integer target numbers went through a binary encoding function to convert the integer to a 32-bit binary number. The fitness function was chosen to be the RMS difference between the target binary number and the individual chromosome binary number. Full code can be found in APPENDIX A or GitHub.

The default parameters for the GA were: target = 550, p_count = 100, retain = 0.2, random_select = 0.05, mutate = 0.01. figure 1 shows the fitness history for the population when the GA was used with these defaults; the correct solution was eventually found after a total of 502 generations.

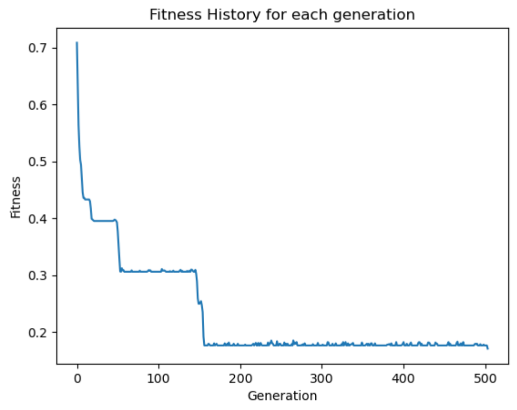


Figure 1 - Fitness History for the population over 1000 generations with the GA using defaults parameters p_count = 100, retain = 0.2, random_select = 0.05, mutate = 0.01

The average number of iterations needed to converge on a solution over the 10 runs was 303 with the GA converging on a solution within 1000 generations 100% of the time. The average final population fitness over 10 runs when a solution was found was 0.17%.

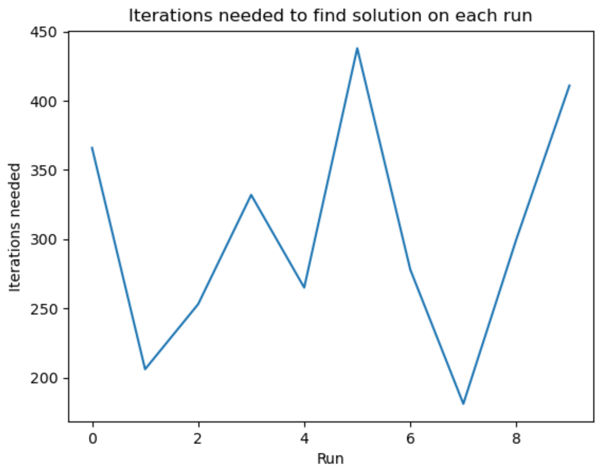


Figure 2 – number of iterations needed to find a solution for the 10 GA runs using default parameters

Figure 2 shows the variance in iterations needed to find the solution. This variation was caused by the random generation of the starting population, as the population was initialised with each chromosome containing 32 genes of random value 0 or 1, and so on different runs, the initial populations contained a varying number of chromosomes that were close to the solution in the problem space.

The GA then had the default population size, mutation, and crossover parameters optimised such that the solution may be found in the shortest number of iterations. The full process and reasoning behind these final parameters, and the effect of each parameter is explained in Exercise 2 below. The optimised parameters were p_count = 1250, retain = 0.15, random_select = 0.05, mutate = 0.075. Figure 3 shows the fitness history for the population when the GA used these optimised parameters; a solution was found in 5 generations. This was repeated a total of 10 times to find the average number of iterations to find the solution, whilst on optimised parameters.

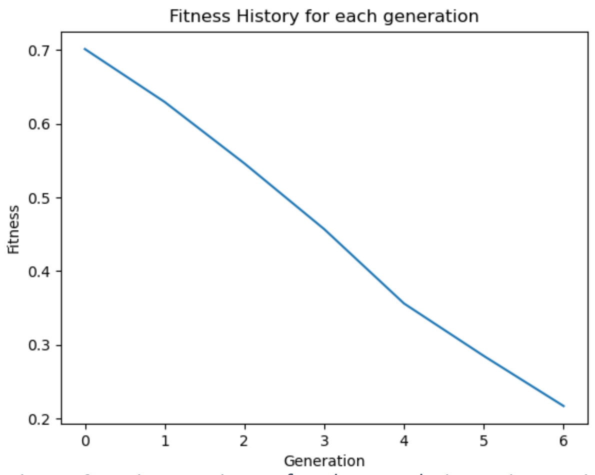


Figure 3 - Fitness History for the population using optimised parameters p_count = 1250, retain = 0.125, random_select = 0.05, mutate = 0.075

The average number of iterations needed to converge on a solution over the 10 runs was 8.4, an average reduction of 311.4 iterations when finding the solution. The GA also converged on a solution within 500 generations 100% of the time. The average final population fitness over 10 runs when a solution was found was 0.23 - slightly worse than the initial parameters. The number of iterations needed to find a solution being reduced by such a large degree, whilst the population fitness rose slightly, points to the default parameters causing stagnation where there was not enough diversity in the chromosomes. This is often caused by lack of or inefficient mutation and crossover, causing the GA to get stuck at a local minimum, not ever finding the solution. Figures 4 and 5 show the variance in iterations needed to find the solution, and final population fitness for the 10 GA runs on optimised parameters.

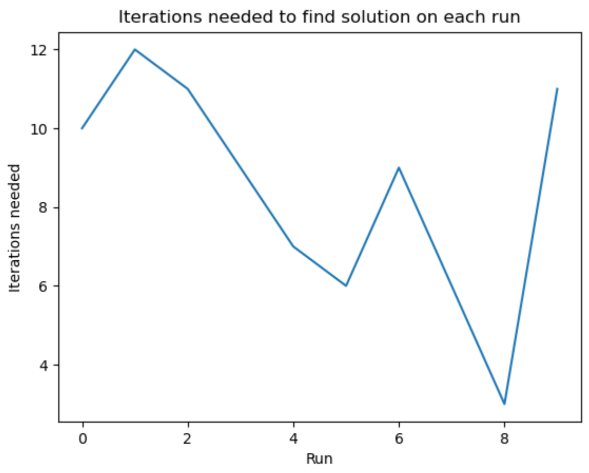


Figure 4 - number of iterations needed to find a solution for the 10 GA runs using optimised parameters

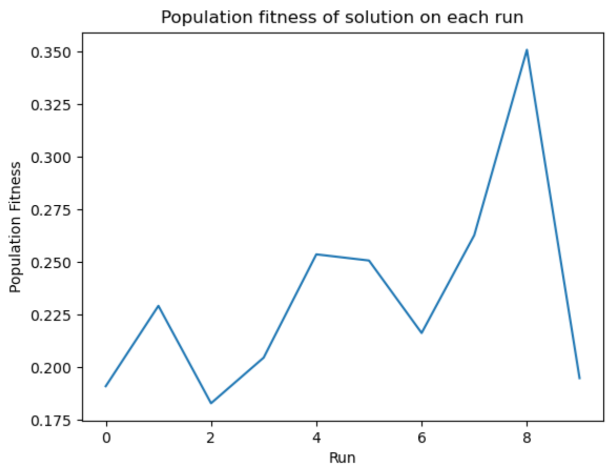


Figure 5 - population fitness when a solution was found the 10 GA runs using optimised parameters

The increase in crossover and mutation probabilities in the optimised parameters promoted more genetic diversity to ensure exploration of different parts of the optimisation landscape and prevent stagnation. As a result, a more dynamic population fitness which took longer to converge was present, even if the exact solution was found faster.

EXERCISE 2 – How do population size, mutation, and crossover parameters affect how quickly a solution is found:

To see the effect of population size, mutation and crossover parameters, the GA in APPENDIX A or GitHub was coded to include sweep functions such that the parameters could all be swept over a range to determine what impact different values had on how quickly a solution was found:

1. Population Size:

As the initial population contains randomly generated genes, the probability of containing the solution or a chromosome closer to the solution increases with population size. A larger population allows for more points in the problem space to be searched simultaneously, combined with the increased diversity a larger initial population possesses, the likelihood of stagnation is reduced. It would be expected then that a larger population size would result in the solution being found in fewer iterations. During the optimisation process in Exercise 1, a population sweep was completed, where the population size was swept from 10 to 2500 in steps of 200, whilst all other parameters were fixed, this resulted in figure 6.

A clear relationship between the population size and number of iterations needed to find the solution is present. The red best fit curve shows an exponential relationship where larger population sizes require fewer iterations to find the solution. As compute time also goes up dramatically with larger populations, the exponential curve suggests there is an optimal population size trading-off between iterations needed and compute power/time. A population size of 1250 was chosen for the optimised population size in Exercise 1. Located at the start of the plateau, it benefits from the greatly reduced number of iterations, but does not waste excess compute power for marginal improvement offered by larger population sizes.

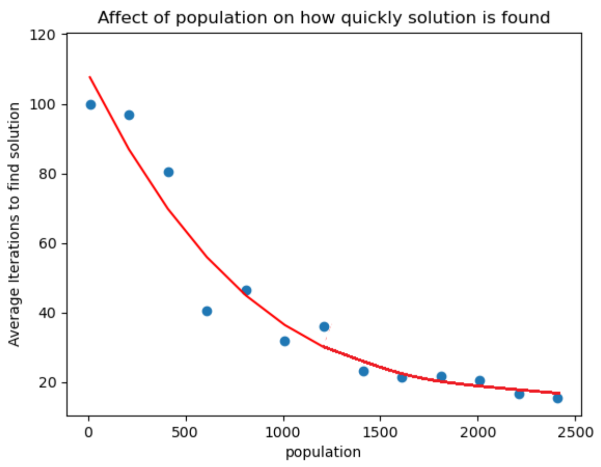


Figure 6 – Effect of population size on number of iterations needed to find the solution in a GA

2. Mutation probability:

Mutation probability is the likeliness that a gene in the chromosome will be flipped, hence producing a mutated individual. Mutation helps to promote diversity in the population by creating new genetic sequences, ensuring the entire problem space is explored, which may not have been possible with crossover alone. Mutation also helps prevent clones appearing in the population resulting in premature convergence. The mutation probability needs to be just high enough to obtain the benefits, but not so high that it mutates too many fit chromosomes destroying promising genetic sequences. During the optimisation in Exercise 1, a mutation probability sweep was completed, where the mutation probability was swept from 1 to 20% in 2.5% steps, figure 7 shows the result of this sweep with a red best fit curve applied. The expected behaviour can be seen in this graph, where an initial increase in mutation probability from 1 to 7.5% caused a reduction in iterations needed to find solution, but a further increase from 7.5-20% caused little to no improvement as the benefits of mutation were already being seen. As a result, the mutation probability of 7.5% was used in the optimised parameters for Exercise 1.

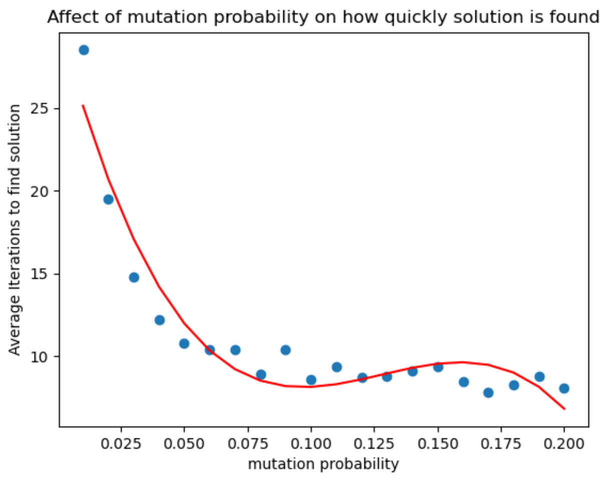


Figure 7 – Effect of mutation probability on number of iterations needed to find the solution in a GA

3. Crossover probability:

Crossover probability is the proportion of offspring made by crossover – where two parent chromosomes have their genes 'spliced' resulting in a child chromosome which possesses genes from both parents – much like reproduction in nature. Like mutation, crossover helps to combat the problem of stagnation by driving up population diversity and finding the best solution from pre-existing genes. Unlike mutation as it is the exchange of genetic material pre-existing in parents, it may only create new gene patterns around the crossover locus, so it is important to use a combination of mutation and crossover to explore the entire problem space. Too high values of crossover will greatly reduce the number of fit gene patterns in chromosomes being passed onto the next generation, a balance must be struck. During the optimisation in

Exercise 1, a crossover probability sweep was completed, where the crossover probability was swept from 5 to 30% in 2.5% steps, figure 8 shows the results with a red best fit curve applied.

Crossover clearly shows a slight reduction in the number of iterations needed to find a solution, up to a crossover probability of 15%. Beyond this the increasing probability starts to apply crossover to too many chromosomes that are close to the solution, reducing in the number of fit gene patterns passed to the subsequent generation worsening performance. As a result, a crossover probability of 15% was deemed optimal and used for the optimised parameters in Exercise 1.

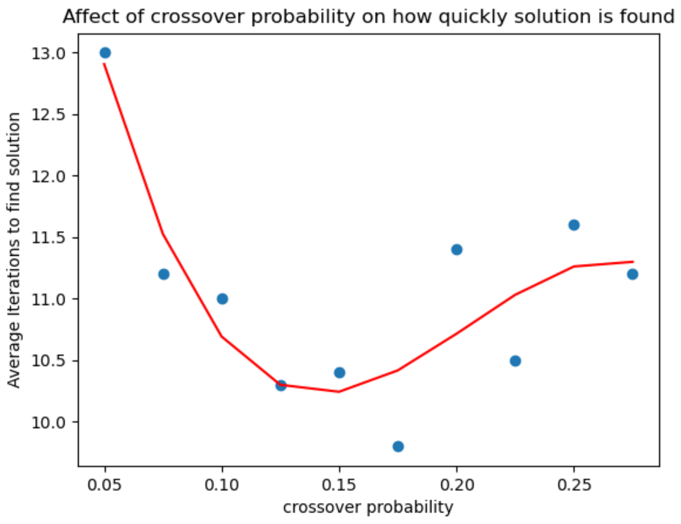


Figure 8 – Effect of crossover probability on number of iterations needed to find the solution in a GA

EXERCISE 3 – How to stop the algorithm when a suitable solution has been found to avoid excessive computation time:

There are many possible termination conditions for a GA to avoid excessive computation time when a suitable solution has been found. The GA used in Exercise one inherently stops when the maximum number of defined iterations 'generations' has been met. This termination condition however bares no relation to the solution being found, and only moves to avoid excessive computational time. To stop the algorithm when a suitable solution has been found, different termination conditions must be used. Stopping the GA based on population fitness does not guarantee the exact solution has been found as it is an average. Meaning it is possible for the solution to be found with a bad population fitness, just as the solution may not be found with a good population fitness. An example case of this is shown in *Exercise 4 – 3. Termination methods*, where the population fitness achieved the termination threshold, without the correct individual being found. To avoid this the GA must be stopped when the cost function of an individual chromosome is below a pre-defined threshold. In Exercise 1 and 5 this meant the same function that was used to find the fitness of an individual, was used as a termination criterion for the GAs. When the fitness function returned a fitness of 0 for any individual in the population, the algorithm was terminated, this was the fastest way to know the exact solution is present in the current population. The termination condition must come just after a new generation has been created, to avoid a solution being missed, mutated, or crossed over into the next generation. Once met, it breaks out of the algorithm, presents the most fit individual as the solution, and avoids any unnecessary computational power being used.

EXERCISE 4 – GA for optimisation of 5th-order polynomial $y = 25x^5 + 18x^4 + 31x^3 – 14x^2 + 7x -19$:

A GA was made to act as an optimiser to find the coefficients of the specified 5th-order polynomial in the shortest number of iterations. The program was written so user input was required to select the parameters used meaning their effect on how quickly the GA found a solution could be compared, these included different: crossover functions, fitness functions, selection methods, termination criteria, and more. Full code can be found in APPENDIX B or GitHub. The initial GA tested had the 5 coefficients and constant set as targets for the GA such that target = [25, 18, 31, -14, 7, -19].

1. Key Parameter settings: population size, mutation probability, crossover probability:

1. Population size:

As expected from the explanation in Exercise 2, an increase in population size caused the GA to perform better, requiring fewer iterations to find the solution. A population size sweep was performed from 1000 to 7000 in 1000 increments, yielding figure 9.

A plateau was seen at around 5000, where further increase in population size saw little in the way of improvement to how quickly the solution was found, and drastically increased compute time. Therefore 5000 was set as the GA population size, minimising iterations needed whilst not taking up too much computational resource.

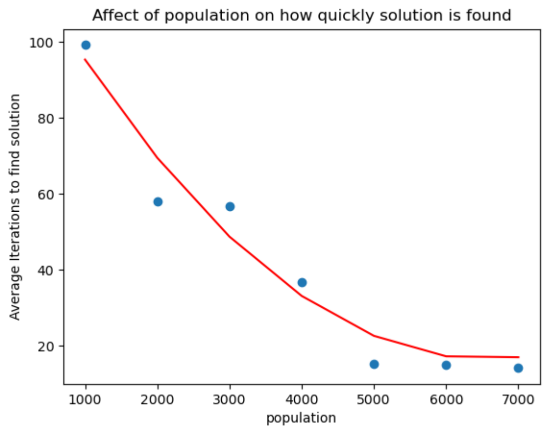


Figure 9 – Effect of population on the number of iterations needed to find the solution

2. Mutation Probability:

A mutation probability sweep was carried out for mutation probabilities 1-10% creating figure 10. 1-3% appeared to be optimal and result in the lowest number of iterations required. Higher mutation probabilities lead to an insufficient number of fit gene patterns passing on to the next generation, requiring more iterations to find the solution. A mutation probability of 0.03 was set as the optimal for this GA, to maximise diversity without the negative effects.

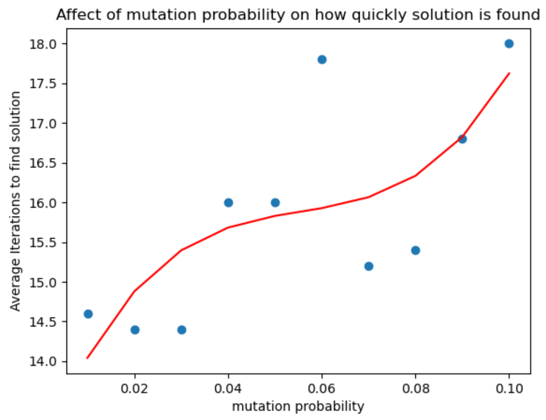


Figure 10 – Effect of mutation probability on the number of iterations needed to find the solution

3. Crossover Probability

The final key parameter set was the crossover probability. Again, a parameter sweep was done for 10-40% crossover probability and resulted in figure 11. This graph showed a minimum around 20% crossover probability, with less than that becoming volatile and producing some much longer times to find the solution. Values above 20% gradually increased the number of iterations required, so the optimised probability of crossover was set to 0.2.

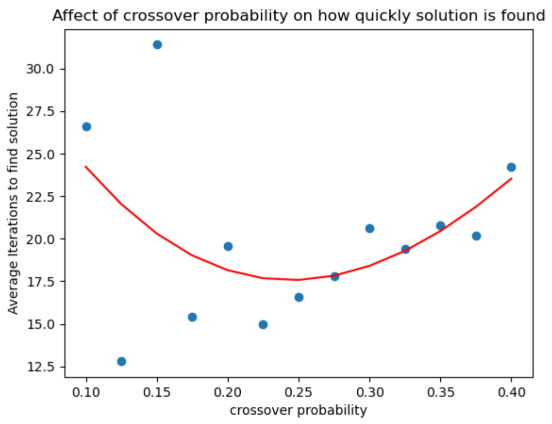


Figure 11 – Effect of crossover probability on the number of iterations needed to find the solution

With the key parameters set, analysis was carried out on the impact of different selection methods, termination methods, crossover methods, mutation methods, and fitness functions. The impact was averaged over 5 runs for each method being tested to consider the random initial generation.

2. Selection methods and Elitism:

Until this point the GA used a ranked selection method where individuals were ranked in order of fitness, and then the top proportion selected for the next generation. The performance of ranked selection was compared against Roulette wheel selection. Roulette wheel selection gives each individual a probability of being selected where $P = \frac{fitness\ of\ individual}{sum\ of\ fitnesses\ of\ population}$, resulting in the fittest individual having the largest probability of being selected.

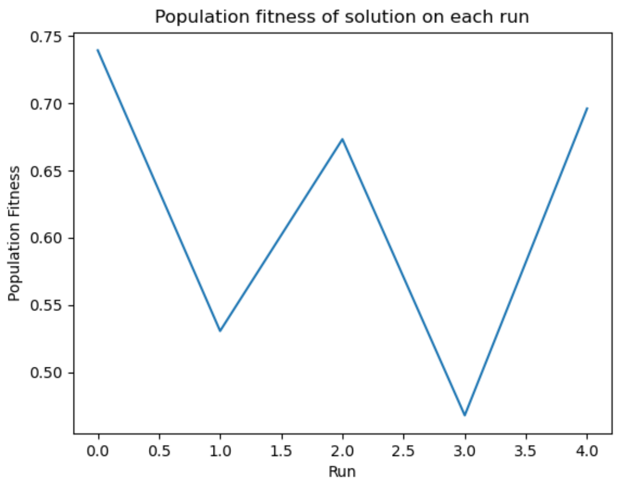


Figure 12 – Average population fitness for 5 runs using Ranked selection

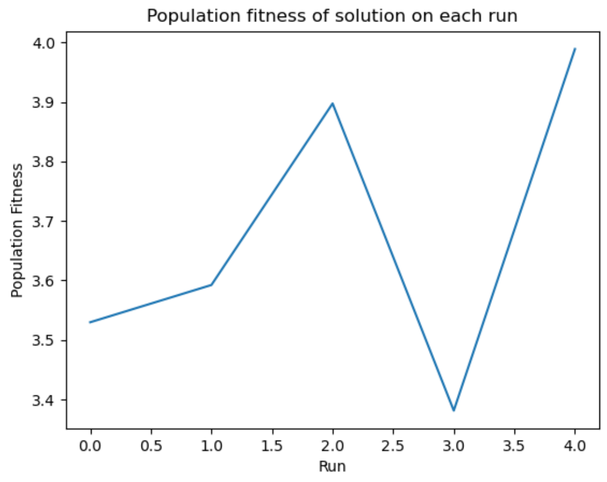


Figure 13 – Average population fitness for 5 runs using Roulette selection

Figures 12 and 13 show results for average population fitness using ranked (left) and roulette (right) selection methods. Ranked selection averaged a population fitness of 0.62 vastly superior to 3.67 average of roulette.

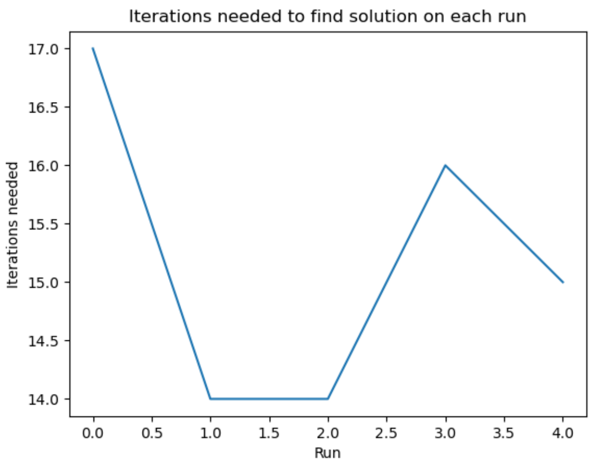


Figure 14 – Number of iterations needed to find solution (100 = no solution found within 100 generations) for ranked selection

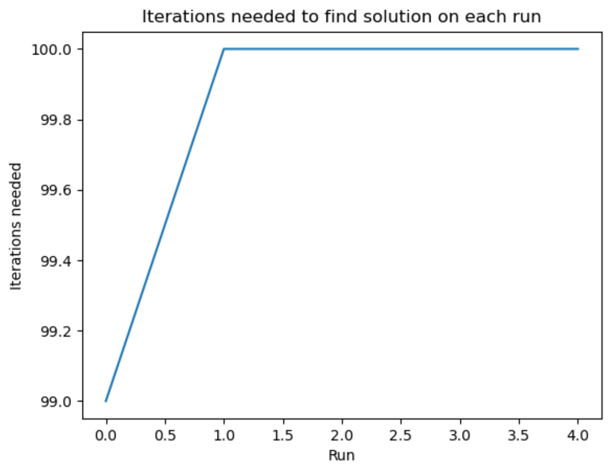
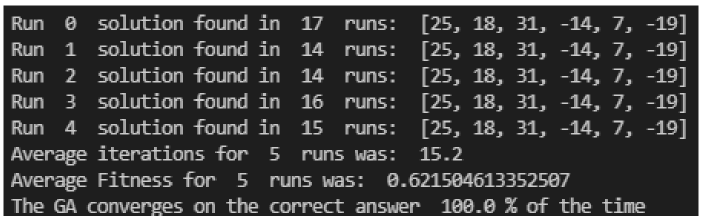


Figure 15 – Number of iterations needed to find solution (100 = no solution found within 100 generations) for roulette selection

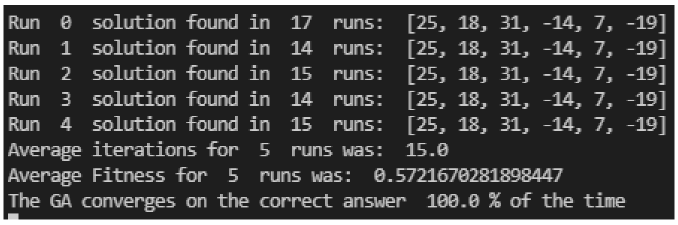
Roulette selection continued to perform worse than ranked selection when comparing the average number of iterations needed to find a solution in figure 14 (ranked), and figure 15 (roulette). Roulette selection was only able to find a solution within 100 generations in one of the five runs completed. It took 99 iterations to find this solution compared to the average of 15 iterations for ranked. There are a few reasons why roulette selection performed worse than ranked. Firstly, one very fit individual can dominate the probability of selection, and therefore greatly reduce the diversity of population causing stagnation. Secondly, when fitness of the individuals became similar, the selection puts almost no favour on the fittest chromosomes, as a result progress became very slow as the population converged close to a solution, so many more iterations were required.

Elitism can be used in combination with either selection method. This is the process of identifying the fittest chromosome and copying it over to the next generation verbatim, preventing crossover and mutation affecting the chromosome and causing destruction of the best gene pattens. Elitism was applied to the GA using ranked selection, such that the fittest 2 chromosomes were carried on to the next generation. Figure 16 and figure 17 show the results for the GA without and with Elitism respectively.



```
Run  0  solution found in  17  runs:  [25, 18, 31, -14, 7, -19]
Run  1  solution found in  14  runs:  [25, 18, 31, -14, 7, -19]
Run  2  solution found in  14  runs:  [25, 18, 31, -14, 7, -19]
Run  3  solution found in  16  runs:  [25, 18, 31, -14, 7, -19]
Run  4  solution found in  15  runs:  [25, 18, 31, -14, 7, -19]
Average iterations for  5  runs was:  15.2
Average Fitness for  5  runs was:  0.621504613352507
The GA converges on the correct answer  100.0 % of the time
```

Figure 16 – GA performance with no Elitism



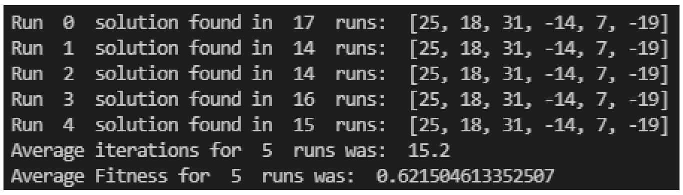
```
Run  0  solution found in  17  runs:  [25, 18, 31, -14, 7, -19]
Run  1  solution found in  14  runs:  [25, 18, 31, -14, 7, -19]
Run  2  solution found in  15  runs:  [25, 18, 31, -14, 7, -19]
Run  3  solution found in  14  runs:  [25, 18, 31, -14, 7, -19]
Run  4  solution found in  15  runs:  [25, 18, 31, -14, 7, -19]
Average iterations for  5  runs was:  15.0
Average Fitness for  5  runs was:  0.5721670281898447
The GA converges on the correct answer  100.0 % of the time
```

Figure 17 – GA performance with Elitism

Comparing the performance of the GA without elitism (left) and with elitism (right) the improvement of elitism averaged only 0.2 less iterations for the five funs performed. This is small improvement is within the margin of error for each set of runs, so does not conclusively show the advantages of elitism. The average fitness of the population did show a larger improvement. The Impact of elitism was most likely minimal, due to the large 5000 population size with only two individuals being carried over verbatim, a larger elitism pool would likely better highlight the benefits of elitism, however elitism of the two fittest individuals did slightly improve the number of runs needed to find a solution and yielded a larger improvement to average fitness of the next generation as expected.

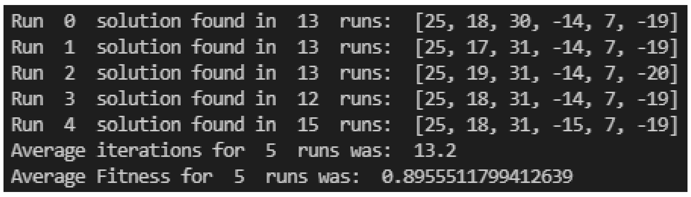
3. Termination methods:

A comparison between terminating the GA by population fitness and Individual fitness was made in exercise 3. The GA in exercise 4 had these two termination methods compared, first with an average population termination threshold < 1, and then an individual fitness termination threshold = 0. Figures 18 and 19 show the performance of the GA using individual and population fitness terminations respectively.



```
Run  0  solution found in  17  runs:  [25, 18, 31, -14, 7, -19]
Run  1  solution found in  14  runs:  [25, 18, 31, -14, 7, -19]
Run  2  solution found in  14  runs:  [25, 18, 31, -14, 7, -19]
Run  3  solution found in  16  runs:  [25, 18, 31, -14, 7, -19]
Run  4  solution found in  15  runs:  [25, 18, 31, -14, 7, -19]
Average iterations for  5  runs was:  15.2
Average Fitness for  5  runs was:  0.621504613352507
```

Figure 18 – GA performance with individual fitness termination condition



```
Run  0  solution found in  13  runs:  [25, 18, 30, -14, 7, -19]
Run  1  solution found in  13  runs:  [25, 17, 31, -14, 7, -19]
Run  2  solution found in  13  runs:  [25, 19, 31, -14, 7, -20]
Run  3  solution found in  12  runs:  [25, 18, 31, -14, 7, -19]
Run  4  solution found in  15  runs:  [25, 18, 31, -15, 7, -19]
Average iterations for  5  runs was:  13.2
Average Fitness for  5  runs was:  0.8955511799412639
```

Figure 19 – GA performance with population fitness termination condition

Initially comparing performance of the two termination methods, population fitness termination (right) appears to perform better, meeting the termination criteria after 13.2 runs average compared to 15.2 of the individual fitness termination (left). However, on closer inspection it was seen the solutions given in figure 19 only had every coefficient correct in one of the 5 runs. This was because despite the general population meeting the termination criteria, there is no way of knowing if the exact target solution has been found, as is demonstrated here where the GA terminated on population fitness without a suitable solution. Therefore, individual fitness works much better as a termination function where the role of the GA is to find an exact solution. For a GA where an exact solution is not required or possible, a terminating condition based on population fitness may save some computational power and time.

4. Crossover methods:

To this point the GA has been using single point crossover at a random point in the chromosome, such that the children were produced by adding the prefix of a male to the suffix of a female. The performance of 1-point, 2-point, and uniform crossover was analysed, and the results shown in Table 1. Two-point crossover is where both parent chromosomes are split at two points, and the children constructed from parts 1 and 3 of the first parent, parts 2 and 4 of the second or vice versa. Uniform crossover is where a random binary crossover mask of same length as the parent is produced, parity indicates which parent the gene will be taken from, such that every gene can be a potential crossover point.

Table 1 – Testing results for different crossover functions, and their impact on iterations needed to find solution, average population fitness, and success rate

Crossover operator	Crossover probability	Average iterations	Average population fitness	Success rate (%)
One-point	0.1	33.6	0.35	80
	0.2	**14.2**	**0.27**	**100**
	0.3	19.4	0.27	100
	0.4	23.1	0.32	100
	0.5	30.6	0.33	100
	0.6	39.6	0.35	100
	0.7	55.4	0.45	100
	0.8	82.4	0.56	100
Two-point	0.1	69.0	0.38	80
	0.2	**30.4**	**0.32**	**100**
	0.3	39.8	0.44	80
	0.4	35.2	0.26	100
	0.5	39.8	0.49	100
	0.6	61.4	0.38	100
	0.7	71.4	0.37	100
	0.8	89.6	0.54	60
Uniform	**0.1**	**11.4**	**0.36**	**100**
	0.2	13.4	0.32	100
	0.3	17.6	0.30	100
	0.4	21.8	0.26	100
	0.5	27.4	0.40	100
	0.6	37.2	0.44	100
	0.7	52.2	0.48	100
	0.8	83.8	0.46	100

Based on the results in table 1, solutions generated using the uniform crossover method outperformed both one-point and two-point crossover. Uniform crossover successfully obtained the solution 100% of the time within 100 generations and managed to find the solution in the least number of iterations - 11.4. Unlike single- and double-point crossover which define only one or two places where an individual can be split, uniform crossover makes every point a potential crossover point avoiding problems with gene loci. It creates more possibilities of recombination, so searches a much larger problem space helping avoid getting stuck in local minima. Two-point crossover surprisingly performed slightly worse than one-point. This may be because the building blocks are more likely to be disrupted due to double the number of crossover points, however, it would be expected that this would be outweighed by the possibility of good genetic information from the head and tail of the parents being preserved. Performance of two-point crossover was unexpected.

5. Mutation methods:

Mutation has only been performed on 1 gene in the chromosome thus far. The impact of mutating 1 (left), 2 (middle), and 3 (right) random genes on the number of iterations to find a suitable solution was analysed in figures 20, 21, and 22 respectively.

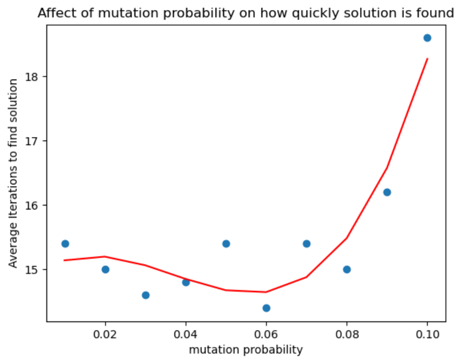


Figure 20 – 1 gene mutation, effect of mutation probability

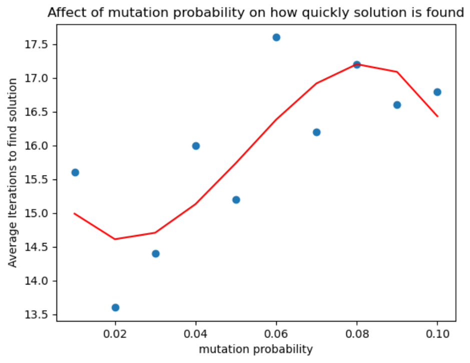


Figure 21 – 2 gene mutation, effect of mutation probability

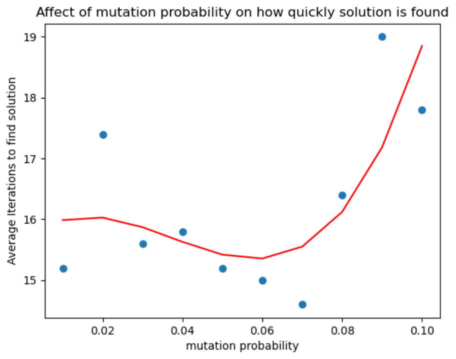
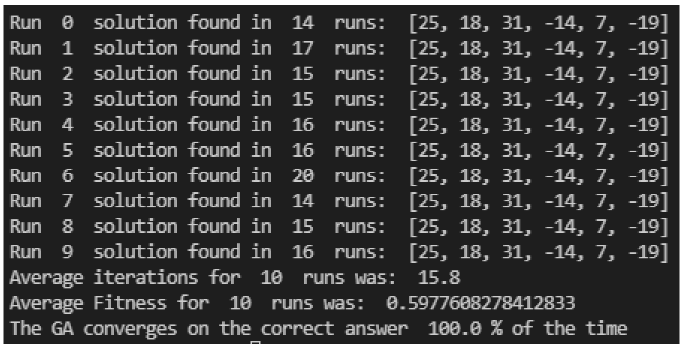


Figure 22 – 3 gene mutation, effect of mutation probability

Changing method of mutation such that the number of genes being mutated increased had only a slight impact on the number of iterations needed to find a solution. All methods tested yielded acceptable results, but mutation of 2 genes seemed the sweet spot for this GA, producing the lowest number of iterations needed to find a solution when mutation probability was 0.02; needing only 13.5 iterations on average. By mutating more genes in the chromosome, greater diversity is entered into the population, as new genetic structures are created. However, increasing the number of genes mutated also resulted in higher standard deviation as fit gene building blocks in the chromosomes are also more likely to be destroyed.

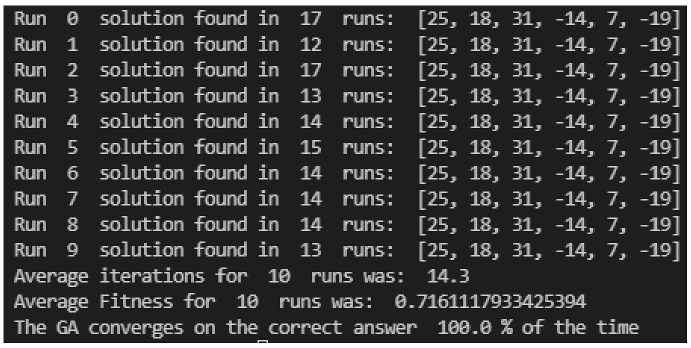
6. Fitness functions:

The fitness function is the function used to determine the fitness of an individual (how close it is to the solution) this information is fed to the selection process and termination condition to create the next generation or determine if a suitable solution has been found. Two fitness functions were compared. RMS between the individual chromosome genes and the target polynomial coefficients, and RMS between 100 points generated by the individual genes and the target polynomial coefficients. Figure 23 (left) shows the GA performance over 10 runs using fitness function = RMS of target and individual, Figure 24 (right) shows GA performance over 10 runs when the fitness function = RMS of 100 generated target and individual points.



```
Run  0  solution found in  14  runs:  [25, 18, 31, -14, 7, -19]
Run  1  solution found in  17  runs:  [25, 18, 31, -14, 7, -19]
Run  2  solution found in  15  runs:  [25, 18, 31, -14, 7, -19]
Run  3  solution found in  15  runs:  [25, 18, 31, -14, 7, -19]
Run  4  solution found in  16  runs:  [25, 18, 31, -14, 7, -19]
Run  5  solution found in  16  runs:  [25, 18, 31, -14, 7, -19]
Run  6  solution found in  20  runs:  [25, 18, 31, -14, 7, -19]
Run  7  solution found in  14  runs:  [25, 18, 31, -14, 7, -19]
Run  8  solution found in  15  runs:  [25, 18, 31, -14, 7, -19]
Run  9  solution found in  16  runs:  [25, 18, 31, -14, 7, -19]
Average iterations for  10  runs was:  15.8
Average Fitness for  10  runs was:  0.5977608278412833
The GA converges on the correct answer  100.0 % of the time
```

Figure 23 – Performance of GA with the fitness function as RMS of target and individual coefficients



```
Run  0  solution found in  17  runs:  [25, 18, 31, -14, 7, -19]
Run  1  solution found in  12  runs:  [25, 18, 31, -14, 7, -19]
Run  2  solution found in  17  runs:  [25, 18, 31, -14, 7, -19]
Run  3  solution found in  13  runs:  [25, 18, 31, -14, 7, -19]
Run  4  solution found in  14  runs:  [25, 18, 31, -14, 7, -19]
Run  5  solution found in  15  runs:  [25, 18, 31, -14, 7, -19]
Run  6  solution found in  14  runs:  [25, 18, 31, -14, 7, -19]
Run  7  solution found in  14  runs:  [25, 18, 31, -14, 7, -19]
Run  8  solution found in  14  runs:  [25, 18, 31, -14, 7, -19]
Run  9  solution found in  13  runs:  [25, 18, 31, -14, 7, -19]
Average iterations for  10  runs was:  14.3
Average Fitness for  10  runs was:  0.7161117933425394
The GA converges on the correct answer  100.0 % of the time
```

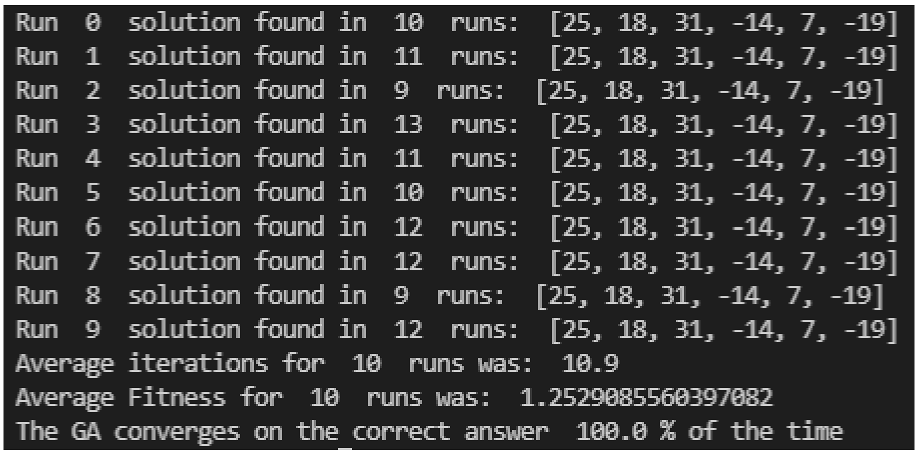
Figure 24 - Performance of GA with the fitness function as RMS of 100 generated points from target and individual polynomial coefficients

Results showed the fitness function performing an RMS of 100 generated target and individual points performed best and found the solution in 1.5 less iterations on average. This is down to the increased resolution in fitness gained by comparing 100 points as opposed to just the 6 gene coefficients, as a result, individuals of similar fitness were better distinguished and ranked. A downside was significantly more compute power and time needed to produce these 100 points every time the fitness function was called, the improvement in iterations was outweighed by the excessive computational time increase. Therefore, RMS between the target and solution coefficients should be used, despite its slightly poorer iteration average.

From the tests carried out, the GA which found the solution in fastest used:

- Parameters: population = 5000, crossover probability = 10%, mutate probability = 3%
- Elitism
- Ranked selection
- 2-gene mutation
- Uniform mask crossover
- RMS of target and individual fitness function
- Individual fitness termination condition

Performance of the fastest GA over 10 runs is shown in figure 25 below. It was able to find the solution – all 5 polynomial coefficients and one constant - with 100% success rate in an average of 10.9 generations.



```
Run  0  solution found in  10  runs:  [25, 18, 31, -14, 7, -19]
Run  1  solution found in  11  runs:  [25, 18, 31, -14, 7, -19]
Run  2  solution found in  9  runs:  [25, 18, 31, -14, 7, -19]
Run  3  solution found in  13  runs:  [25, 18, 31, -14, 7, -19]
Run  4  solution found in  11  runs:  [25, 18, 31, -14, 7, -19]
Run  5  solution found in  10  runs:  [25, 18, 31, -14, 7, -19]
Run  6  solution found in  12  runs:  [25, 18, 31, -14, 7, -19]
Run  7  solution found in  12  runs:  [25, 18, 31, -14, 7, -19]
Run  8  solution found in  9  runs:  [25, 18, 31, -14, 7, -19]
Run  9  solution found in  12  runs:  [25, 18, 31, -14, 7, -19]
Average iterations for  10  runs was:  10.9
Average Fitness for  10  runs was:  1.2529085560397082
The GA converges on the correct answer  100.0 % of the time
```

Figure 25 – results of best performing GA to find the 5 coefficients and 1 constant of a polynomial, after optimising parameters and mutation, crossover, selection, fitness, and termination functions.

EXERCISE 5 - Holland's schema theorem with regards to results obtained [1]:

Much of Holland's schema theorem has already been explained indirectly during exercise 1 & 4 results explanation, however the results seen according to Holland's Schema Theorem will be explained explicitly here.

A Schema is a gene pattern that can exist within a chromosome, for example in exercise 1 where a chromosome is 32 bit binary encoded such as C = [01010101110101010101010101110001]. A schema could be the pattern H = [010************101**********0001] where * indicates a do not care gene. A chromosome is said to match the schema if it contains the bits specified in the same positions. Chromosome C would match and be an instance of Schema H, but a copy of chromosome C with the first bit flipped, Chromosome D = [11010101110101010101010101110001] would not match Schema H as it does not contain every bit in the same position and parity as schema H. The order of a schema is the number of bits that are not '*' don't care parity; this is important as the order of a schema represents the number of bits that can be changed by mutation. The defining length is the distance between the first and last defined bits of the schema and represents the number of positions in which the schema may be divided during crossover. The example schema H above has an order o(H) = 10 and defining length δ(H) = 31.

- Hollands Schema theorem on Selection:

The expected number of instances of H in the next generation is defined by the following equation:

$$Equation\ 1. \qquad E[m(H,t+1)] = M\ \cdot P(h \in H) = \frac{m(H,t)f(H,t)}{\bar{f}\ (t)} \qquad ,$$

$$where, M = poulation, \bar{f}\ (t) = average\ fitness\ of\ population, m(H,t) = instances\ of\ H,$$

$$f(H,t) = average\ fitness\ of\ instances\ of\ H, P(h \in H) = probability\ of\ instance\ H\ in\ mating\ pool$$

It's worth noting this equation ignores the small probability a new instance is created by mutation or crossover which was not in the last generation. Schemas with a better fitness than the population average are more likely to appear in the next generation, so the number of schema H will increase between generations if the average fitness of instances of H is above population fitness as per equation 1. The effect of this was seen in Exercise 5 where the GA used both ranked and roulette selection methods. Roulette selection provides poor bias to fitter solutions when the population is similar in fitness, as a result the GA becomes poorer at selecting the fittest individuals tending $P(h \in H)$ towards 0. So fewer than expected fit schemas were present in the next generation, slowing the rate in which the solution was found. This had a much smaller impact on Ranked selection, as the fittest individuals are always selected even if the fitness is very similar, hence its better performance.

- Hollands Schema theorem on Crossover

A Schema survives crossover if one of the parents is an instance of the schema H, and at least one of the children is also an instance of schema H. A schema is destroyed if crossover occurs in the defining length of the schema, unless repaired by the genes provided by the other parent. The probability crossover occurs in this defining length is given by:

$$Equation\ 2. \qquad p_{c1} = \frac{\delta(H)}{\iota - 1}$$

$$Where\ \delta(H) = defining\ length, \iota = string\ length$$

The upper bound of the probability schema H is destroyed, is found by multiplying Equation 2 by the crossover probability and the lower bound found by subtracting this from 1 as seen in equations 3 and 4 respectively.

$$Equation\ 3. \qquad D_c(H) \leq\ p_c \cdot \frac{\delta(H)}{\iota - 1} \qquad , \qquad Equation\ 4. \qquad S_c(H)\ \geq\ 1 -\ p_c \cdot \frac{\delta(H)}{\iota - 1}$$

$$Where\ D_c(H) = upper\ bound\ probability\ schema\ H\ survives, S_c(H) = lower\ bound\ probability\ schema\ H\ survives,$$

$$p_c = crossover\ probability$$

Therefore, an increase in probability of crossover in the GA will cause a decrease in the probability a schema survives as the second term in equation 4 increases linearly with p_c. This explains the results seen in exercise 1 and 4, when the crossover probability was swept, an initial increase in crossover probability yields the benefits of increased diversity preventing stagnation. But as crossover probability increased further, it caused the second term of Equation 4 to dominate, resulting in too small of a survival probability and many fit schemas being destroyed between generations hindering performance. Hollands Schema theorem also explains why 1- position crossover performed worse than uniform mask crossover. The longer fit schemas in the population that are subject to 1-position crossover rarely survive evolution, as probability of crossover occurring in the defining length, given in Equation 2 is high. However, with multi-point crossover such as the uniform mask and 2-point crossover the probability for a second crossover point to occur within the length of the schema is also high, and so if both points of crossover happen between the head and the tail of the long schema, provided no genes of the schema lie between these crossover points, the long schema survives.

- Hollands Schema theorem on Mutation:

For a schema to survive mutation, all the fixed bits must maintain parity, the do not care '*' bits can be mutated without affecting the schema. In Exercise 5 when 1, 2, and 3 gene mutation was compared, the probability a gene is mutated changes. For 1-bit mutation, the probability a gene is mutated is given by Equation 5 as there are 6 genes, each with an equal probability of mutation where p_m is the user set mutation probability,

$$Equation\ 5. \qquad Mutation\ probability\ of\ a\ gene\ p_{m1} = p_m \cdot \frac{1}{6}$$

For 2-bit mutation, the probability a gene is mutated is given by Equation 6:

$$Equation\ 6. \qquad Mutation\ probability\ of\ a\ gene\ p_{m2} = p_m \cdot \frac{2}{6}$$

For 3-bit mutation, the probability a gene is mutated is given by Equation 7:

$$Equation\ 7. \qquad Mutation\ probability\ of\ a\ gene\ p_{m3} = p_m \cdot \frac{3}{6}$$

The probability a gene is not mutated is given by Equation 8, where p_{mx} is the mutation probability of a gene:

$$Equation\ 8\ . \qquad probability\ gene\ not\ mutated = 1 - p_{mx}$$

The probability a schema survives mutation is given by equation 9:

$$Equation\ 9. \qquad S_m(H) = (1 - p_{mx})^{o(H)}\ , where\ o(H) is\ scheme\ order$$

Equation 9 shows that as the number of bits mutated by the function increases, p_{mx} increases in size proportionally, so the base of Equation 9 reduces in size. As a result, the probability a schema survives mutation is exponentially reduced. This explains why in Exercise 4, when the mutation function was changed between mutating either 1, 2, or 3 bits, the 3-bit mutation function performed worst, as it had the lowest probability of survival for a fit schema going through mutation.

- Hollands Schema Theorem Conclusion

The full Hollands Schema Theorem is defined by Equation 10:

$$Equation\ 10. \qquad E[m(H, t+1)] \geq \frac{m(H,k)f(H,k)}{\bar{f}} \cdot \left(1 - p_c \cdot \frac{\delta(H)}{\iota - 1}\right) \cdot (1 - p_{mx})^{o(H)}$$

Where the first term represents the selection effect, the second term the crossover effect, and the third term the mutation effect. As the terms are multiplied together, it explains how the slight improvements made in each term during Exercise 4 optimisation, can be compounded into a large impact on the likelihood of fit schema survival. Hollands Schema effect can therefore be confirmed as a large contributor to the reason why the reduction in generations needed to find the solution in exercise 1 and 4 was so large.

References:

[1]Dingguo Zhang.(2021). *Lecture 10 GA Part 2.* [Online]. Available: https://moodle.bath.ac.uk/mod/resource/view.php?id=926793

APPENDIX:

Appendix A:

```python
#import required libraries

from random import randint, random
from operator import add
import matplotlib.pyplot as plt
import functools
import numpy as np
import math

#Create a member of the population
def individual(length, min, max):
    return[randint(min,max) for x in range(length)]

#create a number of idividuals
def population(count, length, min, max):
    return[ individual(length, min, max) for x in range(count)]

#determine fitness of individual
def fitness(individual, target):
    #RMS error between individual and target
    delta = np.subtract(target, individual)
    delta2 = np.square(delta)
    ave = np.average(delta2)
    dfitness = math.sqrt(ave)
    return dfitness

#find average fitness of population
def grade(pop, target):
    summed = functools.reduce(add, (fitness(x, target) for x in pop))
    return summed/(len(pop)*1.0)

#convert int to binary
def int_to_binary(integer, length):
    binary = format(integer, "b")
    individual = [int(x) for x in str(binary)]
    size_diff = length - len(individual)
    final_individual = []
    if size_diff != 0:
        for x in range(size_diff):
            final_individual.append(0)
        final_individual.extend(individual)
    else:
        final_individual = individual
    return final_individual

#evolve the population
def evolve(pop, target, retain, random_select, mutate):
    graded = [(fitness(x, target), x) for x in pop]
    graded = [x[1] for x in sorted(graded)]
    retain_length = int(len(graded)*retain)
    parents = graded[:retain_length]
    #randomly add other individuals to promote genetic diversity
    for individual in graded[retain_length:]:
        if random_select > random():
            parents.append(individual)
    #mutate some individuals
    for individual in parents:
        if mutate > random():
            pos_to_mutate = randint(0, len(individual)-1)
            #flip bit
            if individual[pos_to_mutate] == 0:
                individual[pos_to_mutate] = 1
            else:
                individual[pos_to_mutate] = 0
    #crossover parents to create children
    parents_length = len(parents)
    desired_length = len(pop)-parents_length
    children = []
    while len(children) < desired_length:
        male = randint(0, parents_length-1)
        female = randint(0, parents_length-1)
        if male != female:
            male = parents[male]
            female = parents[female]
            half = len(male)//2
            child = male[:half]+female[half:]
            children.append(child)
    parents.extend(children)
    return parents

#re-usable function to sweep parameters to see their effect on how quickly a solution is found
def sweep_parameter(p_count, retain, random_select, mutate):

    runs_fitness_history = []
    iterations_needed_history = []
    sum_of_runs = 0
    performance = 0
    sum_of_iterations = 0

    #loop for the number of runs to be averaged as smooths random starting variance
    for x in range(runs_to_average):
        iterations_needed = generations
        #main usage
        p = population(p_count, i_length, i_min, i_max)
        fitness_history = [grade(p, target),]
        for i in range(generations):
            p = evolve(p, target, retain, random_select, mutate)
            fitness_history.append(grade(p, target))
            #stop the algorithm if suitable solution has been found
            suitable_solution = 0
            for indv in p:
                if fitness(indv,target) == suitable_solution:
                    iterations_needed = i
                    solution_found = 1
```

```python
                solution = int("".join(str(i) for i in indv),2)
                break
            else:
                solution_found = 0

        if solution_found == 1:
            performance = performance + 1

            break

    #was solution found
    if solution_found == 1:
        print("Run ", x, " solution found in ",iterations_needed," generations: ", solution)
    if solution_found == 0:
        print("Run ", x, " solution NOT found in ", generations," generations")

    #get sum of iterations for average, store in array
    sum_of_iterations = sum_of_iterations + iterations_needed
    iterations_needed_history.append(iterations_needed)

    #for datum in fitness_history:
        #print(datum)

    #get how many runs were completed total, store fitness for runs
    sum_of_runs = sum_of_runs + fitness_history[-1]
    runs_fitness_history.append(fitness_history[-1])

    #display fitness history for individual run
    if show_generation_fitness_graph == "Y":
        plt.title("Fitness History for each generation")
        plt.xlabel("Generation")
        plt.ylabel('Fitness')
        plt.plot(fitness_history)
        plt.show()

#display average iterations needed to get solution
average_iterations = sum_of_iterations/runs_to_average
print("Average iterations for ", runs_to_average, " runs was: ", average_iterations)
average_iteration_history.append(average_iterations)

#display the average fitness of all the runs
average_fitness = sum_of_runs/runs_to_average
print("Average Fitness for ", runs_to_average, " runs was: ", average_fitness)

#display performance of all runs as percentage
final_performance = (performance/runs_to_average)*100
print("The GA converges on the correct answer ", final_performance, "% of the time")

#plot graph of each runs final fitness to see variance
if show_average_fitness_variance_graph == "Y":
    plt.title("Population fitness of solution on each run")
    plt.xlabel("Run")
    plt.ylabel('Population Fitness')
    plt.plot(runs_fitness_history)
```

```python
        plt.show()

    #plot graph of iterations needed to see variance
    if show_average_iterations_needed_graph == "Y":
        plt.title("Iterations needed to find solution on each run")
        plt.xlabel("Run")
        plt.ylabel('Iterations needed')
        plt.plot(iterations_needed_history)
        plt.show()
    return

#function to plot affects of sweep on number of itterations to find solution
def plot_swept_param(swept, swept_history):
    x = np.array(swept_history)
    y = np.array(average_iteration_history)
    theta = np.polyfit(x, y, 3)
    print(f'The Parameters of the curve: {theta}')
    y_line = theta[3] + theta[2] * pow(x, 1) + theta[1]  * pow(x, 2) +theta[0] * pow(x,3)

    title = "Affect of " + swept + " on how quickly solution is found"
    plt.title(title)
    plt.xlabel(swept)
    plt.ylabel('Average Iterations to find solution')
    plt.scatter(x,y)
    plt.plot(x, y_line, 'r')
    plt.show()
    return

#default values
    #target = 550
    #p_count = 100
    #i_length = 6
    #i_min = 0
    #i_max = 100
    #generations = 100
    #retain=0.2
    #random_select=0.05
    #mutate=0.01

#initialise global variable and get user input
i_length = 32
i_min = 0
i_max = 1
target = 550
generations = 500
runs_to_average = int(input("Enter number of runs to find average fitness from: "))
show_generation_fitness_graph = input("Do you want Generational Fitness Graphs for every run? Y/N: ")
show_average_fitness_variance_graph = input("Do you want to show fitness variance graphs? Y/N: ")
show_average_iterations_needed_graph = input("Do you want to show iterations variance graphs? Y/N: ")

target = int_to_binary(target,i_length)

#run code for optimal solution (least number of iterations to converge on solution)
if input("Least Iterations Solution? Y/N: ") == "Y":
    retain=0.15
    random_select=0.05
    mutate=0.075
    p_count = 1250
    average_iteration_history = []

    sweep_parameter(p_count, retain, random_select, mutate)

#run code for population sweep to see effect on how quickly solution is found
if input("Sweep population? Y/N: ") == "Y":

    #set values for GA
    p_count_history = []
    average_iteration_history = []
    retain=0.15
    random_select=0.05
    mutate=0.01

    #sweep population
    for p_count in range(10, 2500, 200):
        p_count_history.append(p_count)

        sweep_parameter(p_count, retain, random_select, mutate)

    #plot affect
    plot_swept_param("population", p_count_history)

#run code for mutation sweep to see effect on how quickly solution is found
if input("Sweep Mutation probability? Y/N: ") == "Y":

    #set values for GA
    mutate_history = []
    average_iteration_history = []
    p_count = 1250
    retain=0.15
    random_select=0.05

    #sweep mutation probability
    for mutate in np.arange(0.01, 0.2, 0.025):
        mutate_history.append(mutate)

        sweep_parameter(p_count, retain, random_select, mutate)

    #plot affect
    plot_swept_param("mutation probability", mutate_history)

#run code for crossover sweep to see effect on how quickly solution is found
if input("Sweep crossover probability? Y/N: ") == "Y":

    #set values for GA
    crossover_history = []
    average_iteration_history = []
    p_count = 1250
```

```
random_select= 0.05
mutate=0.075

#sweep crossover probability
for retain in np.arange(0.05, 0.3, 0.025):
    crossover_history.append(retain)

    sweep_parameter(p_count, retain, random_select, mutate)

#plot affect
plot_swept_param("crossover probability", crossover_history)
```

APPENDIX B:

```
#import required libraries

from random import randint, random
from operator import add, pos
import matplotlib.pyplot as plt
import functools
import numpy as np
import math

#Create a member of the population
def individual(length, min, max):
    return[randint(min,max) for x in range(length)]

#create a number of idividuals
def population(count, length, min, max):
    return[ individual(length, min, max) for x in range(count)]

#determine fitness of individual
def fitness(individual, target):
    if fitness_function == 1:
        #RMS error between individual and target
        delta = np.subtract(target, individual)
        delta2 = np.square(delta)
        ave = np.average(delta2)
        dfitness = math.sqrt(ave)

    if fitness_function == 2:
        #difference between 100 points of polynomial
        target_y = []
        individual_y = []
        for x in range(1, 100, 1):
            target_y.append(target[5] + target[4] * pow(x, 1) + target[3]  * pow(x, 2) + target[2] *
pow(x,3) + target[1] * pow(x,4) + target[0] * pow(x,5))
            individual_y.append(individual[5] + individual[4] * pow(x, 1) + individual[3]  * pow(x, 2) +
individual[2] * pow(x,3) + individual[1] * pow(x,4) + individual[0] * pow(x,5))
        #RMS error between 100 individual and target X,Y coordinates
        delta = np.subtract(target, individual)
        delta2 = np.square(delta)
        ave = np.average(delta2)
        dfitness = math.sqrt(ave)

    return dfitness


#find average fitness of population
def grade(pop, target):
    summed = functools.reduce(add, (fitness(x, target) for x in pop))
    return summed/(len(pop)*1.0)

#evolve the population
def evolve(pop, target, retain, random_select, mutate):
    elite = []
```

```python
#ranked selection
if selection_method == 1:
    graded = [(fitness(x, target), x) for x in pop]
    graded = [x[1] for x in sorted(graded)]
    retain_length = int(len(graded)*retain)
    parents = graded[:(retain_length)]
    #decide if using elitism or not
    if eletism_status == "Y":
        # 2 top kept out of mutations and crossover elitism for addition to next generation
        elite.append(graded[0])
        elite.append(graded[1])

#roulette selection
if selection_method == 2:
    #population fitness
    population_fitness = sum([fitness(x, target) for x in pop])
    #chromosone probability
    chromosone_prob = [fitness(x, target)/population_fitness for x in pop]
   #normalise
    norm_chromosone_prob = []
    max_prob = max(chromosone_prob)
    min_prob = min(chromosone_prob)
    for x in range(len(chromosone_prob)):
        norm_chromosone_prob.append((chromosone_prob[x]-min_prob)/(max_prob - min_prob))
    #convert prob for minaturization
    norm_chromosone_prob = 1 - np.array(norm_chromosone_prob)
    #make probabilities sum to 1
    final_chromosone_probs = []
    for x in range(len(norm_chromosone_prob)):
        final_chromosone_probs.append(norm_chromosone_prob[x]/sum(norm_chromosone_prob))
    #create parents population
    retain_length = int(len(final_chromosone_probs)*retain)
    graded = []
    for x in range(retain_length):
        temp = np.arange(0,len(pop),1) # 1D size of pop as random choice numpy only takes 1D
        roulette_position_selected = np.random.choice(temp, p=final_chromosone_probs)
        graded.append(pop[roulette_position_selected])
    #decide if using elitism or not
    if eletism_status == "Y":
        parents = graded[2:(retain_length)]   # 2 top kept out of mutations and crossover elitism
    else:
        parents = graded[:(retain_length)]      # no eletism

#randomly add other individuals to promote genetic diversity
for individual in graded[retain_length:]:
    if random_select > random():
        parents.append(individual)

#mutate some individuals
for individual in parents:
    if mutate > random():
        #repete 1 - 3 times to mutate 1 - 3 genes depending on method chosen by user
        mutation_bit_hist = []
        for x in range(mutation_method):
                pos_to_mutate = randint(0, len(individual)-1)
                #check not same gene being mutated multiple times
                while pos_to_mutate in mutation_bit_hist:
                   pos_to_mutate = randint(0, len(individual)-1)
                mutation_bit_hist.append(pos_to_mutate)
                #mutation is non-ideal as restricts range of possible values
                individual[pos_to_mutate] = randint(i_min, i_max) #in range

    #crossover parents to create children
    parents_length = len(parents)
    #children produced depends on if elite are kept
    if eletism_status == "Y":
        desired_length = len(pop)-(parents_length+2)
    else:
        desired_length = len(pop)-(parents_length)
    children = []
    while len(children) < desired_length:
        male = randint(0, parents_length-1)
        female = randint(0, parents_length-1)
        if male != female:
            male = parents[male]
            female = parents[female]
            #1-point crossover at random crossover point
            if crossover_method == 1:
                #half = len(male)//2
                crossover_point = randint(1,(len(male)-1))
                child = male[:crossover_point]+female[crossover_point:]
                children.append(child)
            #2-point crossover at random crossover points
            if crossover_method == 2:
                #half = len(male)//2
                crossover_point_1 = randint(1,(len(male)-1))
                crossover_point_2 = randint(1,(len(male)-1))
                #determine which crossover point comes first and add genes to child accordingly
                if (crossover_point_1 < crossover_point_2) & (crossover_point_1 != crossover_point_2):
                    child =
male[:crossover_point_1]+female[crossover_point_1:crossover_point_2]+male[crossover_point_2:]
                    child2 =
female[:crossover_point_1]+male[crossover_point_1:crossover_point_2]+female[crossover_point_2:]
                    children.append(child)
                    children.append(child2)
                if (crossover_point_1 > crossover_point_2) & (crossover_point_1 != crossover_point_2):
                    child =
male[:crossover_point_2]+female[crossover_point_2:crossover_point_1]+male[crossover_point_1:]
                    child2 =
female[:crossover_point_2]+male[crossover_point_2:crossover_point_1]+female[crossover_point_1:]
                    children.append(child)
                    children.append(child2)
            #uniform crossover
            if crossover_method == 3:
                #generate mask
                unifrom_crossover_mask = np.random.choice([0,1], size=len(male))
                #make child gene from male or female parent depending on mask bit
                child = []
                for x in range(len(unifrom_crossover_mask)):
                    if unifrom_crossover_mask[x] == 0:
                        child.append(male[x])
                    if unifrom_crossover_mask[x] == 1:
                        child.append(female[x])
                children.append(child)

    #if eleitism used add back in most elite
    if eletism_status == "Y":
        parents.append(elite[0])#add back in most elite
        parents.append(elite[1])#add back in 2nd most elite
    parents.extend(children)
    return parents

#re-usable function to sweep parameters to see their effect on how quickly a solution is found
def sweep_parameter(p_count, retain, random_select, mutate):

    runs_fitness_history = []
    iterations_needed_history = []
    sum_of_runs = 0
    performance = 0
    sum_of_iterations = 0

    #loop for the number of runs to be averaged as smooths random starting variance
    for x in range(runs_to_average):
        iterations_needed = generations
        #main usage
        p = population(p_count, i_length, i_min, i_max)
        fitness_history = [grade(p, target),]
        for i in range(generations):
            p = evolve(p, target, retain, random_select, mutate)
            fitness_history.append(grade(p, target))
            #stop the algorithm if suitable solution has been found
            #individual solution fitness function termination
            if termination_function == 1:
                suitable_solution = 0
                for indv in p:
                    if fitness(indv,target) == suitable_solution:
                        iterations_needed = i
                        solution_found = 1
                        solution = indv
                    else:
                        solution_found = 0
            #population average fitness function termination
            if termination_function == 2:
                suitable_solution = 1
                if grade(p, target) < suitable_solution:
                    iterations_needed = i
                    solution_found = 1
                    performance = performance + 1
                    lowest_indv_fitness = 1000000 #arbitrary high
                    for indv in p:
                        if fitness(indv,target) < lowest_indv_fitness:
                            solution = indv
```

```python
                break
            else:
                solution_found = 0

        if solution_found == 1:
            performance = performance + 1
            break

    #was solution found
    if solution_found == 1:
        print("Run ", x, " solution found in ",iterations_needed," runs: ", solution)
    if solution_found == 0:
        print("Run ", x, " solution NOT found in ", generations," generations")

    #get sum of iterations for average, store in array
    sum_of_iterations = sum_of_iterations + iterations_needed
    iterations_needed_history.append(iterations_needed)

    #for datum in fitness_history:
        #print(datum)

    #get how many runs were completed total, store fitness for runs
    sum_of_runs = sum_of_runs + fitness_history[-1]
    runs_fitness_history.append(fitness_history[-1])

    #display fitness history for individual run
    if show_generation_fitness_graph == "Y":
        plt.title("Fitness History for each generation")
        plt.xlabel("Generation")
        plt.ylabel('Fitness')
        plt.plot(fitness_history)
        plt.show()

#display average iterations needed to get solution
average_iterations = sum_of_iterations/runs_to_average
print("Average iterations for ", runs_to_average, " runs was: ", average_iterations)
average_iteration_history.append(average_iterations)

#display the average fitness of all the runs
average_fitness = sum_of_runs/runs_to_average
average_fitness_history.append(average_fitness)
print("Average Fitness for ", runs_to_average, " runs was: ", average_fitness)

#display performance of all runs as percentage
final_performance = (performance/runs_to_average)*100
print("The GA converges on the correct answer ", final_performance, "% of the time")

#plot graph of each runs final fitness to see variance
if show_average_fitness_variance_graph == "Y":
    plt.title("Population fitness of solution on each run")
    plt.xlabel("Run")
    plt.ylabel('Population Fitness')
    plt.plot(runs_fitness_history)
    plt.show()
    #plot graph of iterations needed to see variance
    if show_average_iterations_needed_graph == "Y":
        plt.title("Iterations needed to find solution on each run")
        plt.xlabel("Run")
        plt.ylabel('Iterations needed')
        plt.plot(iterations_needed_history)
        plt.show()
    return

#function to plot affects of sweep on number of itterations to find solution
def plot_swept_param(swept, swept_history):
    x = np.array(swept_history)
    y = np.array(average_iteration_history)
    theta = np.polyfit(x, y, 3)
    print(f'The Parameters of the curve: {theta}')
    y_line = theta[3] + theta[2] * pow(x, 1) + theta[1]  * pow(x, 2) +theta[0] * pow(x,3)

    title = "Affect of " + swept + " on how quickly solution is found"
    plt.title(title)
    plt.xlabel(swept)
    plt.ylabel('Average Iterations to find solution')
    plt.scatter(x,y)
    plt.plot(x, y_line, 'r')
    plt.show()
    return

#default values
    #target = 550
    #p_count = 100
    #i_length = 6
    #i_min = 0
    #i_max = 100
    #generations = 100
    #retain=0.2
    #random_select=0.05
    #mutate=0.01

#get user input to configure the GA for testing
#initialise global variable and get user input
target = [25, 18, 31, -14, 7, -19]
i_length = 6
i_min = -50
i_max = 50
generations = 100
runs_to_average = int(input("Enter number of runs to find average fitness from: "))
eletism_status = input("Do you want to use Elitism? Y/N: ")
selection_method = int(input("Select selection function: Ranked (1) or Roulette (2): "))
crossover_method = int(input("Select crossover function: 1-point (1) or 2-point (2) or uniform mask (3):
"))
mutation_method = int(input("Select mutation function: 1-bit (1) or 2-bit (2) or 3-bit (3): "))
fitness_function = int(input("Select fitness function: RMS between chromosone and target (1) or RMS
between 100 points of individual polynomial and target (2): "))
termination_function = int(input("Select termination function: Individual (1) or Population (2): "))
```

```python
show_generation_fitness_graph = input("Do you want Generational Fitness Graphs for every run? Y/N: ")
show_average_fitness_variance_graph = input("Do you want to show fitness variance graphs? Y/N: ")
show_average_iterations_needed_graph = input("Do you want to show iterations variance graphs? Y/N: ")



#run code for optimal solution (least number of iterations to converge on solution)
if input("Least Iterations Solution? Y/N: ") == "Y":

    retain=0.1 #0.2
    random_select=0.05
    mutate=0.03
    p_count = 5000
    average_iteration_history = []
    average_fitness_history = []

    sweep_parameter(p_count, retain, random_select, mutate)

#run code for population sweep to see effect on how quickly solution is found
if input("Sweep population? Y/N: ") == "Y":

    #set values for GA
    p_count_history = []
    average_iteration_history = []
    average_fitness_history = []
    retain=0.2
    random_select=0.05
    mutate=0.01

    #sweep population
    for p_count in range(1000, 8000, 1000):
        p_count_history.append(p_count)

        sweep_parameter(p_count, retain, random_select, mutate)

    #plot affect
    plot_swept_param("population", p_count_history)

#run code for mutation sweep to see effect on how quickly solution is found
if input("Sweep Mutation probability? Y/N: ") == "Y":

    #set values for GA
    mutate_history = []
    average_iteration_history = []
    average_fitness_history = []
    retain=0.2
    random_select=0.05
    p_count = 5000

    #sweep mutation probability
    for mutate in np.arange(0.01, 0.11, 0.01):
        mutate_history.append(mutate)

        sweep_parameter(p_count, retain, random_select, mutate)
    #plot affect
    plot_swept_param("mutation probability", mutate_history)

#run code for crossover sweep to see effect on how quickly solution is found
if input("Sweep crossover probability? Y/N: ") == "Y":

    #set values for GA
    crossover_history = []
    average_iteration_history = []
    average_fitness_history = []
    random_select=0.05
    mutate=0.01
    p_count = 5000

    #sweep crossover probability
    for retain in np.arange(0.1, 0.9, 0.1):
        crossover_history.append(retain)

        sweep_parameter(p_count, retain, random_select, mutate)

    #plot affect
    plot_swept_param("crossover probability", crossover_history)
```