Genetic Algorithms Design Decisions and Implementation – Neil Krupa

Introduction

Within AI a genetic algorithm is a highly complex algorithm inspired by biological evolution, used to optimize a specific problem to gain the highest efficiency. Within this report I will cover the idea of genetic algorithms as well as their origins and future. I will also explain how my own genetic algorithm works, which is used to teach a population of ants to gather as much food as possible from a predetermined map. I will cover my design decisions and results as well as how changing hyper parameters effected the results of the optimization. The task I faced was to create a genetic algorithm that taught a population of ants to run the best possible route on a map simulation to collect as much food as they could from generation to generation. Each ant would run the simulation once and receive a fitness score according to how much food they collected. By filling each next generation with a combination of the best ants and ant children from previous generations, the goal was to get the best fitness score possible for the simulation out of a possible 89 food.

Literature Review of State of the Art

Based on the evolutionary process, genetic algorithms mimic the biological model of chromosomes, with each chromosome defining a set of particular traits. In nature this could consist of humans hair colour, being defined by the combination of their parents’ chromosomes, creating the chromosomes within a person’s body. Genetic algorithms also generate new populations from parent chromosomes, using crossover and mutation, in the same way that human generations are populated by parents reproducing (Crossover) and DNA can mutate in each offspring (Mutation). With each new population the goal is to produce the best preforming offspring possible (fittest) for a specific task. The final generation of offspring within the algorithm contain the optimized chromosomes for the task and therefore have the highest fitness values.

Genetic algorithms originated in the early 1950s with work from Alex Fraser, a UK born quantitative geneticist working with CISRO in Australia. Fraser saw the potential in computers at the time and came to the realisation that he could build computational models to mimic natural selection in bacteria to aid him in his research. Fraser formalised his work on modelling natural selection in a book called “Computer Models in Genetics” which covered the fundamentals of genetic algorithms, but was aimed at geneticists rather than computer scientists (published in 1970). Following this, five years later, an electrical engineering professor at the University of Michigan named John Holland wrote a book called “Adaptation in Natural and Artificial Systems” (published in 1975) which outlines specific concepts and possibilities for genetic algorithms. In around 1998 it because commercially available to purchase personal computers with enough processing power to warrant the creation of commercial genetic algorithm systems, with one of the first being Evolver. Evolver is a spreadsheet program that allows for optimization of results by having a genetic algorithm abstracted within the background. This led to genetic algorithms being more widely used across the differing fields gaining access into biomedical sciences, architecture and many more industry’s as it becomes more readily available.

An example of genetic algorithms in use is the Knapsack problem. Within this problem there is a bag that can hold a certain weight, *w*, and a number of items, *n*, each composed of a weight and value. The outcome to be determined is what is the optimum value of items you can fit into the bag, which is less than or equal to weight w? Using genetic algorithms to solve this example would first require a set of chromosomes to be randomly generated, each with a differing combination of weights and values that would fit into the bag. Each chromosomes fitness score is then calculated based on a fitness function, which is a key aspect of genetic algorithms as it determines the optimization value for each chromosome. These fitness values are then sorted into order of fitness and natural selection begins. Unlike the real world, natural selection in genetic algorithms is extremely fast and is only limited by the hardware and software of the machine being used. There are many differing design options for natural selection, such as killing off n lowest values, as well as the type of crossover and selection used to fill the rest of the new generation, which will be covered later in the report. The chromosomes are then bred to produce offspring for the new generation, with the more optimized chromosomes (higher fitness scores) having a greater chance of breeding. Each new chromosome has a chance to be mutated and is added into the new population to generate a new generation. The natural selection process is then repeated until one of two conditions are met. One, that a certain number of generations has been run through and it is deemed that the optimization is the maximum it will reach, or that the best chromosomes are maximised until an alpha chromosome is created. The outcome of this algorithm will therefore give the best optimization for the number of items that can fit in the bag, with the combined highest value. Although, results may differ during each runtime of the algorithm as the final optimization of the genetic algorithm is based on particular parameters entered. These include parameters such as mutation and crossover probabilities as well as fitness function coefficient, randomized initial generation and population size which will be covered later within this report.

The two main factors in generating offspring in a genetic algorithm are Mutation and Crossover. Crossover is the process of “breeding” chromosomes together and comes in three different forms, single point, two point and uniform. This form of offspring creation takes two parent chromosomes from the population and selects either a one, two or multiple points to split the parent chromosomes. The parent chromosomes exchange parts and are reassembled as two new offspring chromosomes with differing parts from each parent in each child. Each type of crossover defines the number of parts each parent is split into. Mutation is the process of changing part of the genome, where as in the real world it comes with negative connotations, in GA it is very useful to ensure that there is diversity within the population and prevents problems with local minima.

Genetic algorithms have been developed to now contain many state of the art variations, such as Neuroevolution, Cellular Evolutionary Algorithms, Differential Evolution and Particle Swarm. In this paragraph I will cover Neuroevolution and Cellular Evolutionary Algorithms and the state of the art research that is currently underway on their development referencing multiple sources. Firstly a Cellular Evolutionary Algorithm is a variation of genetic algorithms designed for machines working in parallel composed of multiple processors executing simultaneously, which was proposed by Robertson in 1987. It contains nodes called ‘Individuals’, rather than chromosomes, which each have a neighborhood (cluster of individuals on a grid). Each neighborhood overlaps adjacent neighborhoods also composed of individuals; between two close neighborhoods an individual picks two of its closest individuals, based on criteria, and selects them as parents. These parents then go through variation operations and replace the individual that selected them as parents in the first place producing an offspring. This GA variation is still under development and optimization and state of the art improvements can be found in Enrique Alba and Bernabe Dorronsoros’ ‘Cellular Genetic Algorithms’ (2008) chapter two, where they suggest that the main use for cGA is “evolving cellular automata in order to make computational tasks”. Another variation of genetic algorithms is neuroevolution. Neuroevolution involves phenotypes instead of genotypes, which are normally used in genetic algorithms, which is a genotyped mapped to allow use in neural networks. State of the art work can be seen in Hidehiko Okada’s ‘Genetic Algorithms with Fuzzy Genotype Values and its Application to Neuroevolution’ (2014) which explains how neuroevolution is an extension of GA and outlines how fuzzy valued genotypes allows for the use of “Fuzzy variables in a fuzzy optimization problem”. This work would therefore allow neural networks to take in imprecise values and variables (fuzzy values and variables) to produce a useful result. This state of the art proposal would help with values taken from one of neuroevolution main uses, evolutionary robotics. This GA variation is also used within gaming.

Description of Algorithm

The genetic algorithm outlined in Appendix A is my solution to the task proposed in the introduction, this algorithm randomly populates a generation of *n* ants (chromosomes), consisting of 10 states, where each state is represented by 3 digits and a fitness value for a total of 31 digits per ant. Firstly the hyper parameters are defined, determining the crossover and mutation probabilities as well as population size per generation and number of generations to run through (Lines 1-6 Appendix A). Then population size is defined and the first population is generated (Lines 8-19 Appendix A), with each state being randomly assigned with corresponding valid values, and each ants fitness score initialized to zero in the following format, containing 31 cells:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1-4 | 0-9 | 0-9 | … | 1-4 | 0-9 | 0-9 | 0 |

The world is then loaded in and each ant simulates it using a function called simulate\_ant(), which runs the ants movements stored inside its personal genetic material. The ant’s fitness is then returned from the function and the algorithm updates that ant’s fitness value (Lines 25-30 Appendix A). The population is then sorted in order of fitness with most fit ants at the bottom and the least fit ants at the top. After this the finesses are normalised in a separate matrix and the cumulative sum is calculated being stored in the ‘SumNormalizedFitnesses‘ matrix for use in the roulette selection later in the algorithm (Lines 32-42 Appendix A). I then chose to include elitism, where the two best ants are saved and stored into the next generation to ensure that the best chromosomes were being carried forward from generation to generation. If I had not included this elitism the next generation has the potential to not mate these best chromosomes with anything and therefore lose the current alpha chromosomes. I therefore stored the best two ants into a new matrix and added them straight into the population (Lines 44-53 Appendix A). After the two best chromosomes were added the rest of the population needed to be filled using roulette selection. Roulette selection allocates parents space on the ‘roulette wheel’ according to their fitness, meaning that the chromosomes with the higher fitness are selected more times. This was chosen in contrast to rank selection, where each chromosome is allocated a fitness from its ranking where the worst have fitness 1 and the best have fitness n. The reason I chose roulette selection over this method is due to the slow convergence rate of rank selection due to the fact that the best chromosomes do not have a great deal of difference relative to the other chromosomes. Although this increases diversity, the fact that roulette wheel has a much greater probability of selecting a high fitness chromosome with a greater convergence rate, roulette selection is more likely to have a higher final optimization. The roulette wheel generates two random numbers (R1 and R2) and loops through the population summing the finesses, when the sum is greater than the random number then it stops and returns the chromosome that is stopped on. This is done to produce two random chromosomes from the population stored as ‘TempChromosome\_1’ and ‘TempChromsome\_2’ (Lines 55-62 Appendix A).

The next step in the algorithm is crossover, which is only done if a randomly generated number is greater than the ‘Crossover Probability’ hyper parameter stored at the start of the algorithm. With this I had the option of single, two point or uniform crossover techniques (as seen in the Literature Review of State of the Art section). Single point crossover was placed within the algorithm (commented out) to compare and contrast the difference between single and two point crossovers, allowing me to analyse the results of each (which can be seen in the Presentation of Results and Analysis section). Due to the length of each ant’s genetic material I decided not to implement uniform crossover as it would be overly complex, whereas single point crossover would be similarly too simple for the length of the genetic material. Based on these deductions I chose to go for two point crossover. This crossover technique generates two random numbers, one from the first half of the genetic material and one from the second half (not including fitness). It then takes these numbers and splits each temporary chromosome (parent) at these points, creating two segments of chromosome per parent. Two of these corresponding segments are then exchanged with the other parent and reassembled as two new children (Lines 79-95 Appendix A).

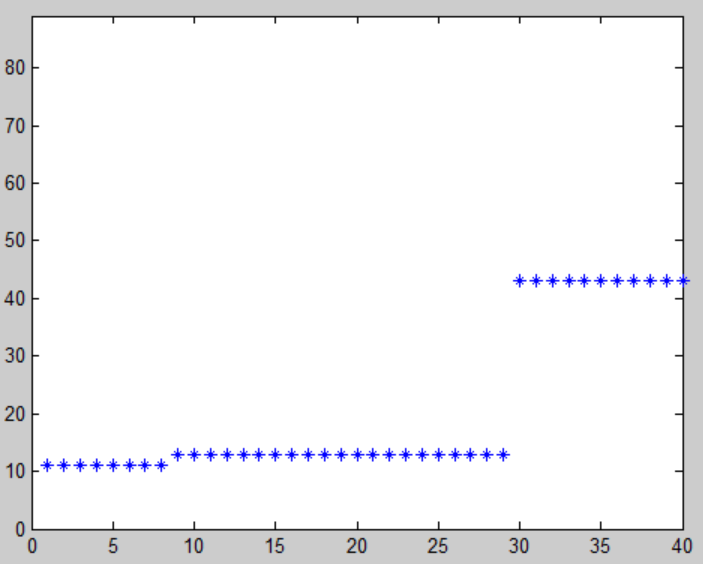
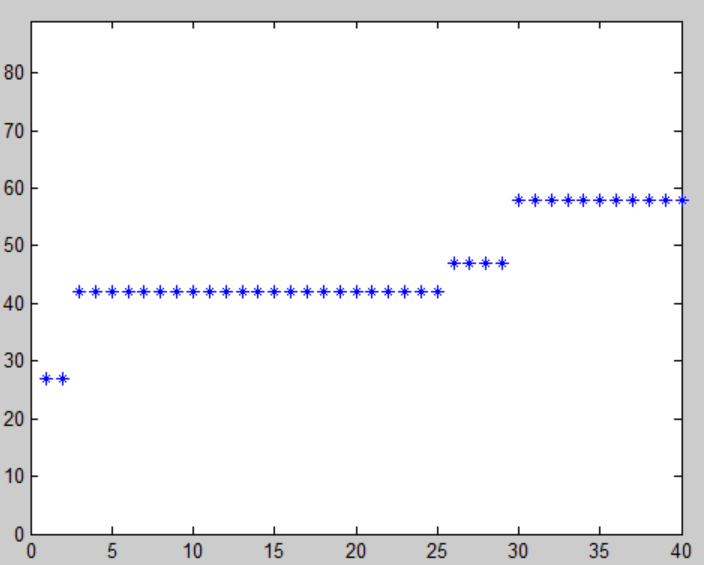
After this the new offspring both have a probability of being mutated individually. If a randomly generated number is less than the ‘MutationProbability’ then the chromosome will be mutated, similarly for both TempChromosome\_1 and TempChromosome\_2. If mutation occurs then a random cell within the genetic material is selected and replaced with a random corresponding value to that particular cell. This helps ensure the change in genome diversity throughout the algorithm as a potentially better or worse ant can be created depending on the value stored (Lines 99-119 Appendix A).

After mutation, the temporary ants are placed into the new population, ensuring that there is space for them, and plots the ant with the best fitness onto a graph. It then displays the highest fitness ant in the current population as well as the generation onto console. After this the new population overwrites the old population and the algorithm loops back up to each ant simulating the world (Lines 121-141 Appendix A). This is done for the number of generations specified in the hyper parameters, to finally get the optimized ant who collects the most food. Once this ant has been optimized the algorithm displays the route travelled by the alpha ant.

Presentation of Results and Analysis

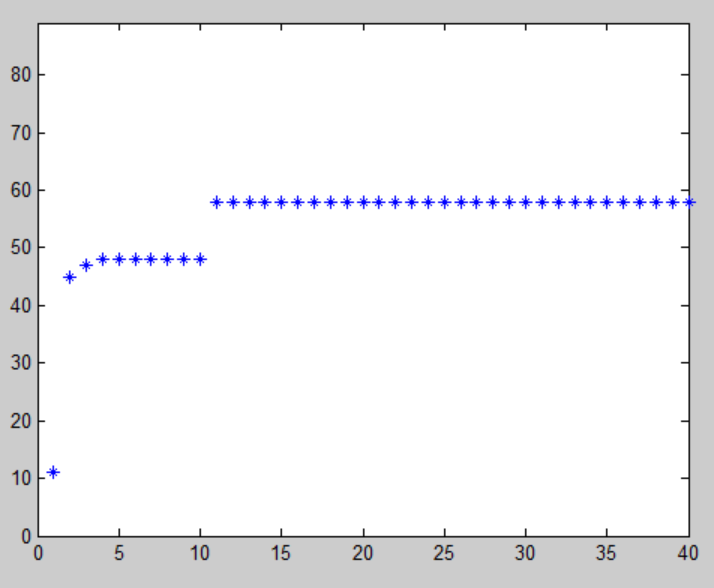
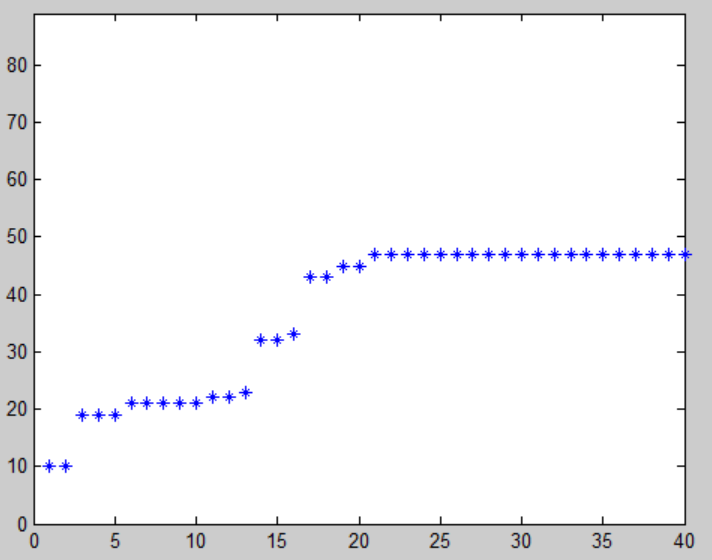
I wanted to find out the best combination of hyper parameters and crossover types to ensure that my algorithm was optimised for the task. I firstly compared single point crossover with multi-point crossover, with a population of 10 ants over 40 generations where the mutation probability and crossover probability were different. I first ran the algorithm with a high crossover probability (1.0) and low mutation probability (0.2) and gained the following results for single point crossover (left) two point crossover (right) where x-axis is the generation number and y-axis is fitness:

Single Point Crossover Two Point Crossover

From these results the single point crossover reached a maximum of 42 from a minimum of 10 with a range of 32 and the other hand the two point crossover reached a maximum of 59 from a minimum of 27 also with a range of 32. This was a fairly consistent result when run multiple times with both having fairly similar ranges. I then tested the difference between both single point and two point crossover with a high mutation probability (1.0) and low crossover probability (0.2), with single point crossover (left) and two point crossover (right) with the same x, y axis:

Single Point Crossover Two Point Crossover

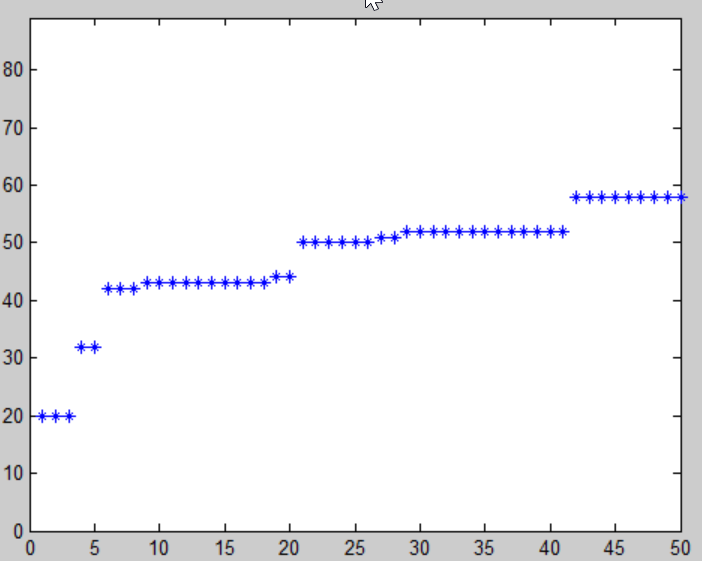
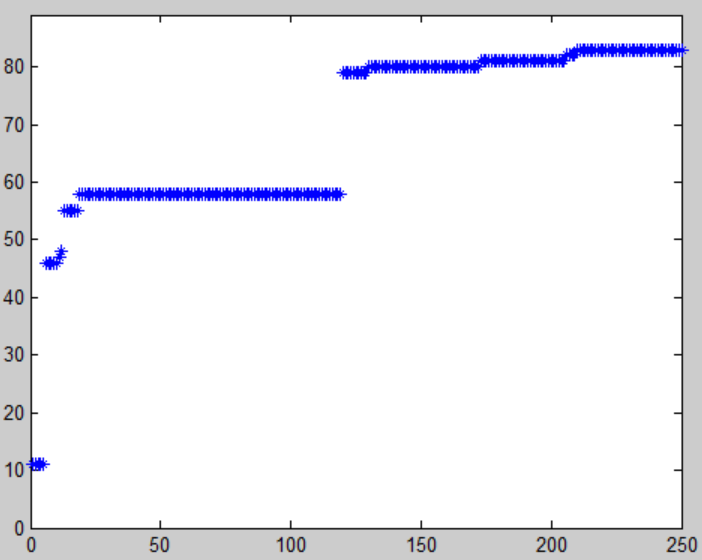
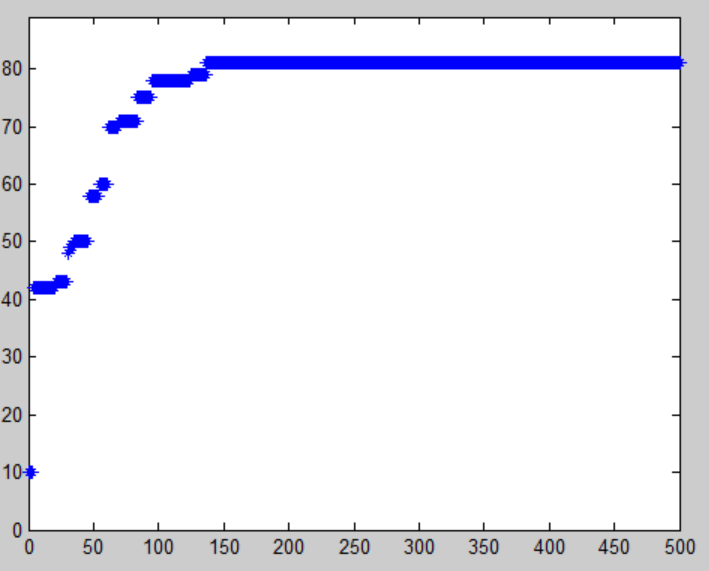
This set of results show how the single point crossover can get stuck on local maximum a lot faster than two point crossover due to the reduced population diversity, which is down to the fundamentals of each method. Although the range in the single point crossover is 48, with a low of 11 and a high of 59 compared to a range of 39 with a low of 10 and a high of 49 for two point crossover, when the algorithms were run multiple times the ranges varied, but still became stuck on local maxima. I therefore chose to use two point crossover within my algorithm, even though the single point crossover had a greater range on this occasion, the increased diversity it gives to the population and the prevention of having it stuck on local maxima is more beneficial to the algorithms optimization.

After selecting to use two point crossover I analysed the results on changing crossover and mutation probability from 0.2 – 1 for each to allow me to compare the results and determine the best strategy for the algorithm.

This graph shows the combinations of crossover probability and mutation probability (x-axis) as well as the highest fitness and lowest fitness for each combination (y-axis). I found that the greater the disparity between crossover and mutation probability the more optimized the results, as you can see the range increased with larger distance between crossover and mutation probability values. I ran the same values five times and received the same overall trend throughout testing, the closer the two probability values were together the lower the range and final fitness values were. This shows that for an optimal result the values of each crossover probability and mutation probability should be significantly different and will be taken on to further analysis when considering population size and number of generations to be run.

Considering the crossover and mutation probability I decided to test my algorithm over 50, 250 and 500 generations to determine the optimum number of generations to return the best fitness value possible. When the mutation probability was set at 1.0 and the crossover probability was set at 0.2 I achieved the following results, with 50, 250 and 500 generations respectively:

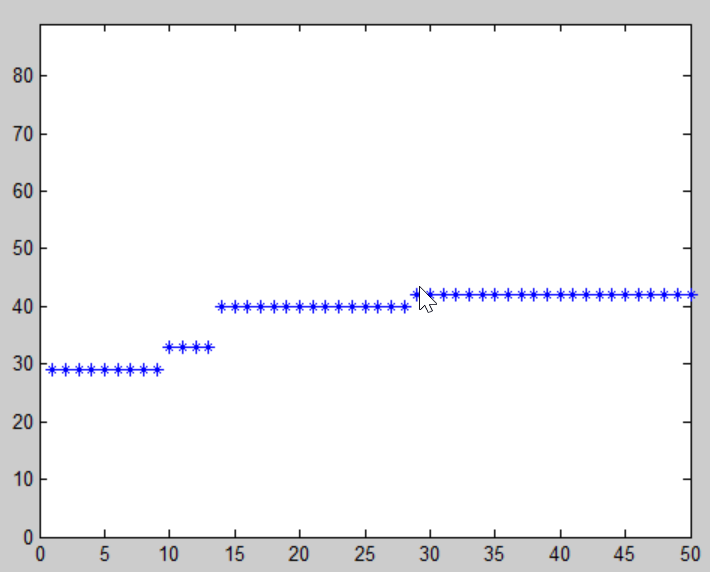
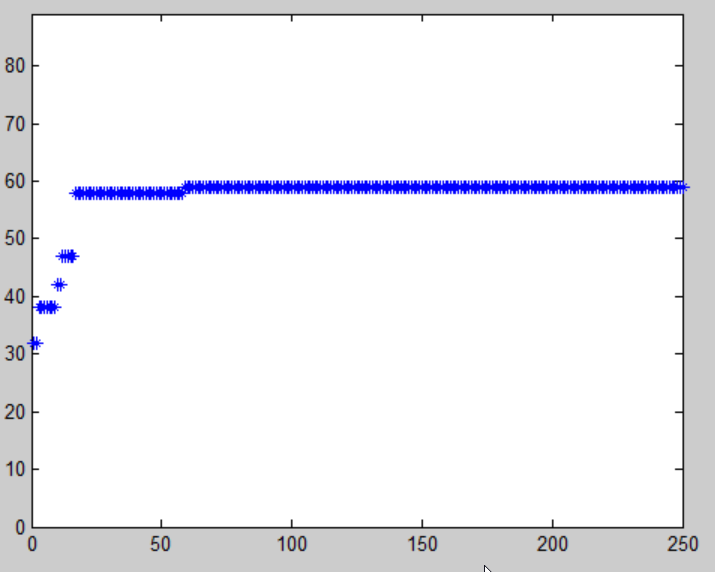
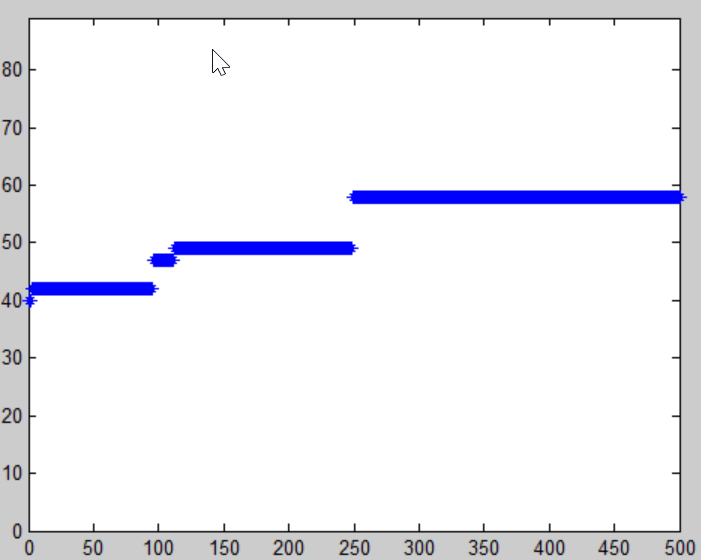
50 Generations 250 Generations 500 Generations

Using 50 generations the results showed that it reached a maximum fitness value of 68, whereas 250 generations and 500 generations reached 82 and 81 respectively. The results also show how during 500 generations the algorithm became stuck on a fitness value of 81 for 362 generations, meaning that each iteration was wasted computation time and power.

Next I set the mutation probability to 0.2 and the crossover to 1.0 and once again tested 50, 250 and 500 generations to see if there was any difference within the final fitness value. The results can be seen below:

50 Generations 250 Generations 500 Generations

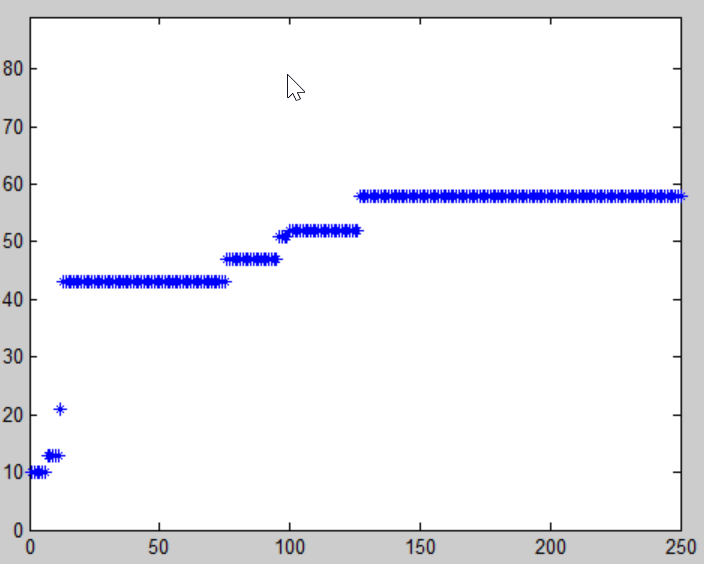
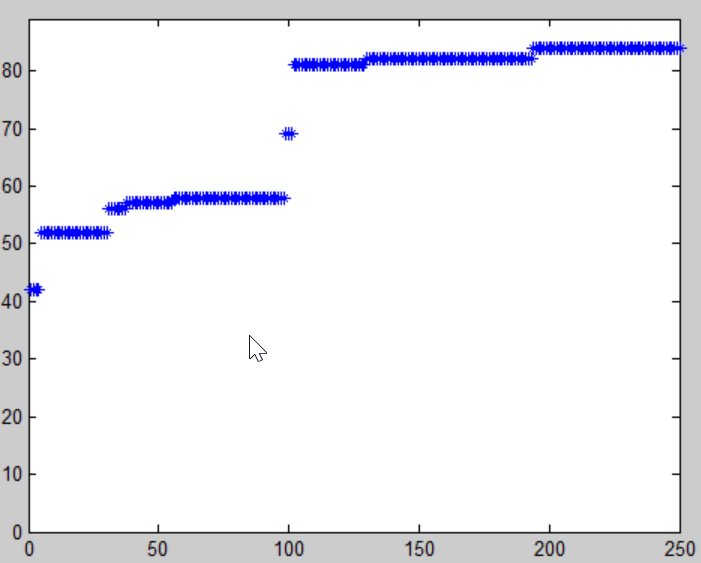
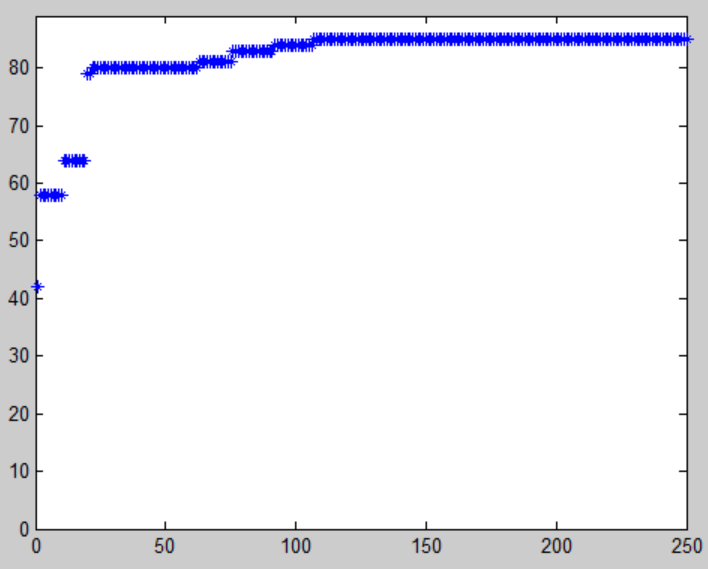
  

Similar to the previous 500 generation test each runtime of the high crossover probability because stuck on the local maximum due to similar chromosomes crossing over with each other. This is due to the fact that there is a lack of diversity within the population due to a low mutation probability value.

From these results the optimum number of generations to be run is around 250 as well as the optimum probabilities being high mutation and fairly low crossover to allow for diversity within the population. These hyper parameters are the values that will be carried on to the next stage of testing, which is population size.

After finding the optimum hyper parameters for my algorithm I decided to test whether population size would affect the results in any way. I therefore decided to run tests of population sizes 10, 100 and 1000 to determine the optimum population size for the algorithm, results can be seen below respectively:

Population 10 Population 100 Population 1000

With a population of 10 the computation time was extremely quick, getting up to a maximum fitness value of 59, although this is not particularly accurate compared to the other two population values, the computation time makes up for this methods lack in accuracy. Using a population of 100 increased the computation time significantly also greatly increasing the population’s maximum fitness value to 84, although this took longer to compute, the increase in accuracy is significant, therefore making it a better hyper parameter combination than the smaller population. Finally the computation time for a population of 1000 was extremely slow, and for a maximum fitness value of 85 it was not worth the extra computation cost. Analysing the results I am able to determine that there is a relationship between population size, computation speed and accuracy, of which the greater the population size the slower the computation time but the greater the accuracy. Therefore I concluded that having a population between 10 and 100 has the best optimization for my genetic algorithm, allowing for a combination of computation speed and accuracy I set my population size to 50.

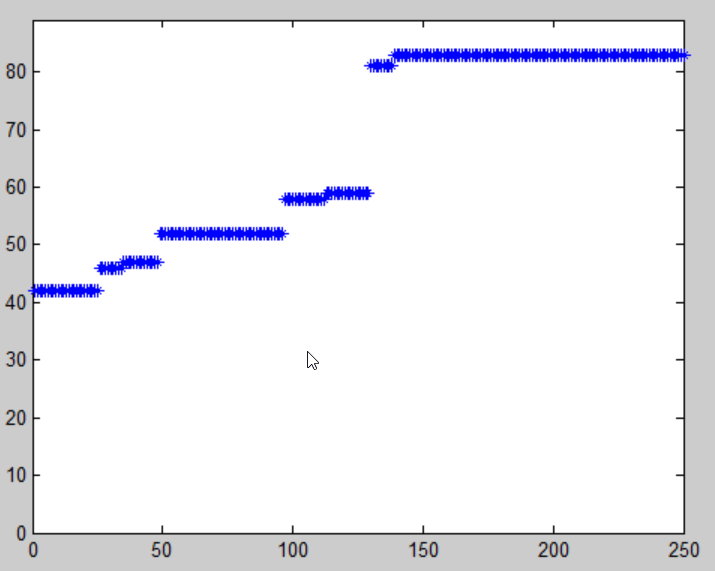
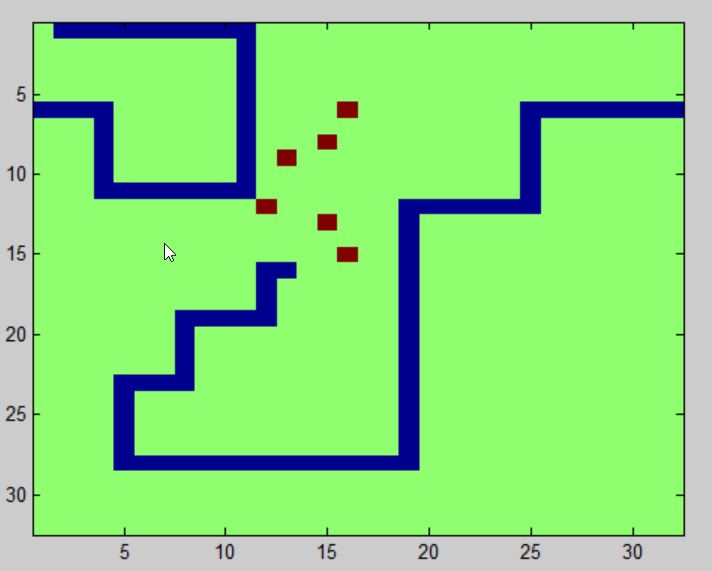
From these results I can determine that the optimized hyper parameters for my genetic algorithm are ensuring that there is a high mutation probability (1.0) and a low crossover probability (0.2) as well as having 250 generations of population 50. These hyper parameters produce the most efficient and accurate results for the optimization of an ant running the simulation.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 4 | 4 | 4 | 7 | 6 | 2 | 6 | 4 | 4 | 8 | 6 | 1 | 2 | 4 | 1 | 9 | 1 | 3 | 7 | 6 | 3 | 0 | 8 | 1 | 4 | 8 | 1 | 2 | 8 |

After running the simulation with the parameters outlined in the previous paragraph I achieved the following results. The fittest individual within the Muir world in the first generation achieved a fitness of 42, where as in the last generation the highest fitness was 83. This is a significant increase in fitness score with a range of 41, with the alpha ant’s genetic material having the following 30 digits (excluding column 31, fitness score):

The following two diagrams display the highest fitness level of each generation (left) and the final highest fitness ant’s trail it took around the Muir World map (right).

Generation Vs Fitness Score (Optimized) Optimized Alpha Ant Track

Conclusion

In conclusion genetic algorithms are now used for a wide range of modelling systems throughout many industries and are still being developed. More about general genetic algorithms. I believe that I have successfully accomplished the task outlined in the introduction, by creating and optimizing an ant to successfully, through generations, learn to find the most food it can during its life span. The optimization is extremely important as differing hyper parameters highly affect the outcome of the genetic algorithm, for example if you have extremely high crossover probability but low mutation probability the algorithm will have less diversity and is likely to get stuck on a local maximum fitness value. Also if your population size or number of generations to be run is too low you are not going to get the optimum end result, but with larger population and generation sizes affect computation speed directly and therefore the user has to take into account which is more important, computation speed or accuracy. As well as hyper parameters there are other factors that affect the outcome of the final result such as fitness function coefficient which if not correctly programmed may give inaccurate fitness results, as it must be determined what the most important factors are that affect the fitness value returned. Also the initial, randomly assigned, genetic material values, which can change the results each time the algorithm is run. If these values are completely different from a working result and all very similar, the outcome is less likely to be the optimum result. This can be overcome by running the algorithm multiple times and removing any anomalous results.

References

Appendix A – cw2\_krupa\_neil Genetic Algorithm

