EVCO Open Assessment

Y6375144

January 8, 2014

Chapter 1

Introduction

1.1 Genetic Algorithms

For some problems it can be difficult to find an optimal solution. This is because the search space grows at a much faster rate (exponentially, for example) than the input space. The Travelling Salesperson problem is a perfect example of this - for a search space with n cities, there are $n!$ paths that the salesperson could take. [2]

Evolutionary algorithms do not present a solution to this problem. However, by taking inspiration from biological evolution, they do offer an alternative approach that will often lead to a 'good-enough' solution in a much more reasonable number of calculations.

Genetic algorithms are a type of evolutionary algorithm that typically represent population members (i.e. candidate solutions) in bitstrings [3]. The population of strings is repeatedly updated through the following process:

1. Select - A number of the existing population members are selected to continue through to the next round. Typically these would be the fittest members in the current population.
2. Crossover - To generate new population members, existing ones are created by crossing two (or more) population members over. See figure 1.1 for a visual representation.
3. Mutate - The population is mutated at random. This typically involves passing over every bit in a string, and with a random probability flipping the bit from a 0 to a 1 (or vice versa).
4. Evaluate - A fitness function is run on each member of the population, giving an indicator of how 'good' a solution is.

Genetic algorithms hold much potential, but in order to get the most from them you must carefully choose your representation and parameters. The representation is the bit string typically, but it can also take other forms. There are many parameters that can be fine tuned, such as the number of offspring to breed (at the crossover stage), or the type of crossover to perform.

1.2 The Problem

Morpion Solitare is a one player game that is typically played on grid paper. To start, a cross shape is drawn in the centre of the page (as shown in figure 1.2). On each turn the player must add one cross that creates a line of five crosses either vertically, horizontally or diagonally. The lines may intersect, and in this particular variant the lines may also overlap (it could be argued that this makes the game easier). Figure 1.2 shows

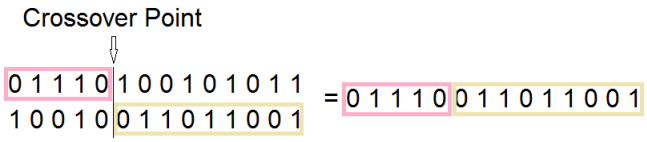


Figure 1.1: The child is made up of the top row (which represents the first parent) until the crossover point, where it then takes from the bottom row (the second parent).

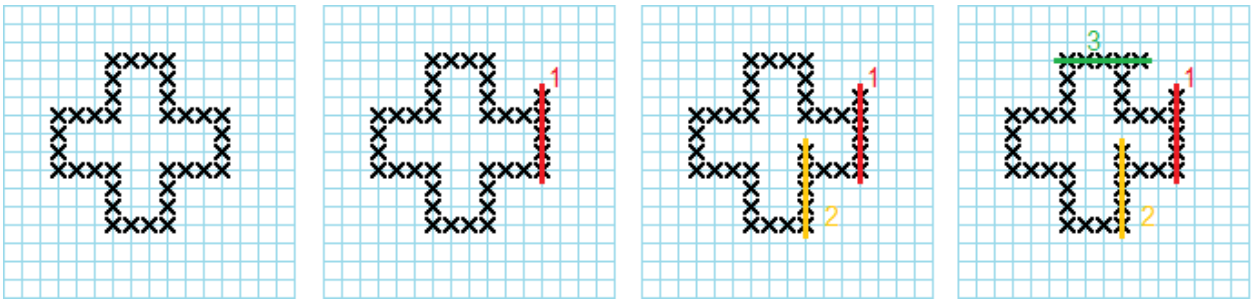


Figure 1.2: The initial state of the game on the left, and from left to right the first three moves are shown. The lines don't have to be numbered, but for clarity they are here.

a possible first three moves. Finishing at this stage would give an overall score of 3, as 3 lines of 5 crosses have been formed [1].

In this report I will detail my investigation into obtaining a high score through the use of a genetic algorithm. Working out the fitness of a population would be a tedious process if I have to take each population member and draw out its game from scratch. Thankfully that is not necessary as I have been provided with a computerised version of the game that is slightly faster.

When designing my genetic algorithm I will have to consider the format that the digital version takes its input by. The program represents positions on the grid by their (x, y) co-ordinate positions, where $(0, 0)$ is the bottom left of the grid. For each move, four parameters must be passed into the program in the format $x_1y_1x_2y_2$, where (x_1, y_1) is the position of one end of the line to draw, and (x_2, y_2) is the other end of that line. For the game shown in figure 1.2, the input would be:

```
linuxMsolitare.exe 12 11 12 7 9 9 9 4 6 13 10 13
```

The program accepts a maximum of 2000 inputs, or 500 moves. Given that the current best known solution is 178 [**?**], this is unlikely going to be a problem. One potential problem is that to evaluate a population of size n, the application will need to be called n times. Even with a decent processor, this could be a slow task for any sizeable population. Also, if a representation is chosen that doesn't directly map to the program's inputs is chosen, then each population member's representation will have to be converted to the correct format before the program can even be called. Again, this could be time consuming, but solutions to these problems will be considered in the next section.

Chapter 2

The Solution

2.1 Representation

The standard genotype in a genetic algorithm is a bit-string [2]. Another common format is to use integers. Floating point numbers are also used, although they are not really appropriate for my solution. I will now discuss possible representations, and then explain my choice of representation.

2.1.1 Gray Code Bit-string

Binary bit-strings are not ideal for a genetic algorithm as a single bit flip during the mutation stage can cause the number to change a lot. Consider the bit string 10010110 that represents the 8 bit unsigned integer 150. During mutation, if just the first bit is flipped, the string becomes 00010110 and now represents 22. If the search space is a 256x256 grid, and that integer represents one co-ordinate, then 150 to 22 is a large jump. It would be more useful if the co-ordinate changed by just a small amount. If we wanted to change to 149 (10010101), two bit flips are required.

One solution to this is to use Gray code. This is a bit-string where each consecutive value differs by only one bit. In Gray code, 11011101 represents 150, and 11011111 represents 149. That's just a single bit flip, making Gray code ideal for genetic algorithms [2].

There are now a few possible options for how Gray code could be used to represent an individual move, and then a series of moves. The first option is to represent the co-ordinates as they are passed into the program in Gray code (remember that a move is passed in to the program as $x_1y_1x_2y_2$). The grid 40x40, so ($\log_2 40 = 5.32$) 6 bits would be required to represent each co-ordinate. Each move would use 24 bits, and if the maximum number of moves is played each time (500), then 12000 bits or 1500 bytes will be required for each population member. If we have a population size of 5000, then the total required space will be 7.15MB.

The second option is to store each move as a combination of (co-ordinate, direction), where co-ordinate is the new cross to draw on the board, and direction is the direction of the line. So ((20, 20), left) would mean draw a cross at position (20, 20), and draw left. This would then be translated into the program's input format $x_1y_1x_2y_2$ to be 20 20 15 20. Again, 6 bits would be required to represent each co-ordinate. As there are 8 possible directions, ($log_2 8$) 3 bits are required to represent the direction that the line should be drawn in. Therefore each move would use 15 bits, giving a maximum of 7500 bits or 938 bytes for each population member. With a population size of 5000, the total required space will be 4.47MB. This is a good saving on the last representation described. However, computers nowadays have a lot of memory, and the difference between 4.5MB and 7.2MB is tiny relative to the standard 4GB of memory that a low-end computer has. There is also the overhead to consider when the representation has to be converted into program input. The

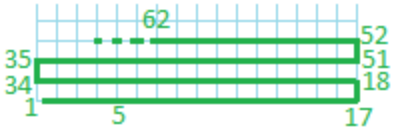


Figure 2.1: An alternative co-ordinate representation that starts at the bottom left of the grid numbering each position as shown.

first described representation would be the fairly simple task of converting Gray code to an integer. With this representation, the overhead would be slightly higher, as the start position and direction would need to be considered in order to calculate the program input.

The third and final possible representation that uses Gray code does not use the co-ordinate system as it has been previously described. Starting at the bottom left and moving right, each position will be assigned a single integer. When the end of the row is reached, the above position will be numbered, and then on that row moving left the positions will be numbered. See Figure 2.1 for a visual explanation. As the grid is 40x40, 1600 positions will be numbered. Describing a move will be done as above, with (postion, line direction). The amount of storage space required will be ($log_2$1600) 11 bits for the position, and 3 again for the direction. This uses 14 bits per move in total, and sticking with the already used example of 500 moves per population member and 5000 population members will use a total of 4.2MB. A slight saving on the previous representation, but again the overhead increases as we now also have to convert between the new position representation as well as the direction representation. Another disadvantage that comes with this representation is that it is more difficult during the mutation stage to move a move in a particular direction. If you wanted to mutate the population member by changing one of its moves to start two rows above, you have to perform a relatively complex calculation in order to work out the number assigned to the position two rows above. With the above two representations it is easy to convert the Gray code to an integer, add 2 to the y co-ordinate, and then convert back to Gray code. Even better (although not necessarily easier to code) would be to perform the arithmetic in Gray code.

2.1.2 Integer Representation

The natural input to the program is a set of integers. Using any other representation internally requires that a conversion takes places before the fitness of the population can be calculated. Because of this, it's worth weighing up the pro's and con's of using an integer representation.

To help understand a reason why an integer representation may be worth considering, I go back to looking at the use binary or Gray code. Consider the move 20 20 24 24 that draws a diagonal line from the centre of the grid to the position (24, 24). If we were using a binary representation with 6 bits for each co-ordinate, the move would look like:

```
010100 010100 01100 011000
```

We then perform a mutation operator on the representation that has a 10% probability of flipping each bit that it comes across. It outputs the following:

```
110100 010101 01100 011100
```

This is the same as 56 21 24 28, an invalid move because the board is limited to 40x40, and also because no the co-ordinates don't represent a line that is 5 positions long.

Using an integer representation then the mutation operator would know what changes could be made to a group of integers that would still be valid, and only perform these. For example, with 20 20 24 24, it might decide to shift down by 2 rows, and so it would subtract 2 from the y co-ordinates. Of course there is no reason why this can't be done with a binary representation - using binary arithmetic 000010 could be subtracted from each x co-ordinate, but using binary provides no advantage over using an integer representation, but does include the disadvantage that when used in the 'pure' genetic algorithm sense (i.e. flipping bits at random), then there is a high probability of creating an invalid move. Of course this is counteracted by using Gray code, but now we consider crossover.

To confirm that using an integer representation instead of Gray code doesn't disadvantage the algorithm, we must also analyse the effect on crossover. Consider the below samples of two population members in binary / Gray code (for this section the point holds for both representations):

```
... 011111 011111 011111 0100011 001100 000001 001000 000001 ...
... 000000 000001 000000 0000101 101000 000110 100100 000010 ...
```

The top one describes the move 31 31 31 35 12 1 8 1, and the bottom describes 0 1 0 5 40 6 36 2. With a naive crossover algorithm, any point in the two binary strings could be chosen to crossover. The above samples each have 48 bits, so let's randomly choose to swap at bit 37. The two new population members are now:

```
... 011111 011111 011111 0100011 001100 000001 000100 000010 ...
... 000000 000001 000000 0000101 101000 000110 101000 000001 ...
```

The top member now represents 31 31 31 35 12 1 4 2, and the bottom 0 1 0 5 40 6 40 1. The second move in each member has been made invalid, reducing the fitness of the member. The chances of performing a crossover and increasing the fitness around the crossover point with a binary representation is therefore low. However, it could still be the case that ultimately the fitness of the member is increased as the moves before the crossover point combined with the moves after the crossover point could together improve the overall fitness.

It is not hard to imagine a situation where the crossover produced a number that is greater than 40, thus rendering the move invalid. To counter these problems we could improve the crossover algorithm to only crossover after whole numbers (i.e. where bit position MOD 6 = 0). If this approach is to taken though, we may as well be using integers because we will only be crossing over in the same position as if we were using integers.

2.1.3 Conclusion

Because I do not believe that using a binary/Gray code representation will provide an advantage, and because an integer representation will be easier to debug and quickly analyse, I have decided to use an integer representation. In the Gray Code Bit-string section I discussed the different representations that could be used, such as giving every position on the board a single number, and then following that by a direction. The same could be done with an integer representation, but I have dismissed this for the same reasons as before: the overhead associated with converting between two formats does not appear to outweigh the small advantage of using less memory.

2.2 Selection

Now that a representation has been decided on, each stage of the genetic algorithm must be decided on. The first of these is selection. With a population in whose members we know the individual fitness of, we must

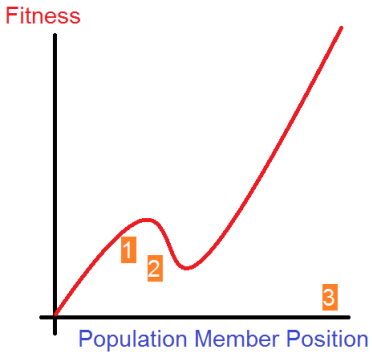


Figure 2.2: A simplified example of population members and their fitness

decide which of the members to take through to the next generation, and which should be discarded to be replaced by crossed over and mutated population members.

2.2.1 Highest Ranking

The Highest Ranking selection algorithm is very simple. The population members are sorted in order of fitness, and then a fraction of the top members are selected to continue to the next round [2, 3].

At first glance this sounds like an ideal solution. To understand why this is not necessarily the case, imagine if the population members were dogs. While choosing which dogs to breed to create a new generation of dogs, we only choose the dogs that are the best looking. One particular ugly dog is not chosen to be bred, but what we didn't know is that if it had been bred with some other dog then it would produce very attractive offspring. This is an extreme example, but the underlying point is that just because the fitness of a population member is not currently high does not necessarily mean that it won't be useful in the future.

Figure 2.2 gives a visual example of this. Each orange box is a population member, the x-axis is the current position of the population member in a search space, and the y-axis is the fitness measurement. Current population members 1 and 2 would be chosen over 3 in the highest ranking algorithm. What we don't know is that if 3 had been selected and crossed with another population member then it could easily have had a higher fitness that 1 or 2 currently have. By selecting 1 and 2 we have got stuck in a local maxima.

2.2.2 Roulette Wheel

In order to overcome the problems with highest ranking selection we must allow for low fitness population members to be included, but we also want to include mainly high fitness population members. A good way of doing this is to select population members with a probability that is proportional to their fitness [3]. So if the sum of fitness across the whole population is 360, and the fittest population member has a fitness of 90, then there is a $\frac{90}{360}$ or $\frac{1}{4}$ chance of the population member being included in the next generation.

Figure 2.3 gives a visual representation of this: the whole area of the wheel represents the summed fitness of the population, and each segment is the proportional fitness of a population member. In the left-hand example, the yellow population member has the highest fitness, and is therefore the most likely to be chosen to continue to the next generation. The green population member has the lowest fitness, and only has a slim chance of being carried through to the next generation. This does not completely fix Highest Ranking's problem that was demonstrated with Figure 2.2, but it is an improvement because it at least gives population member 3 a chance of roulette wheelbeing carried into the next generation.

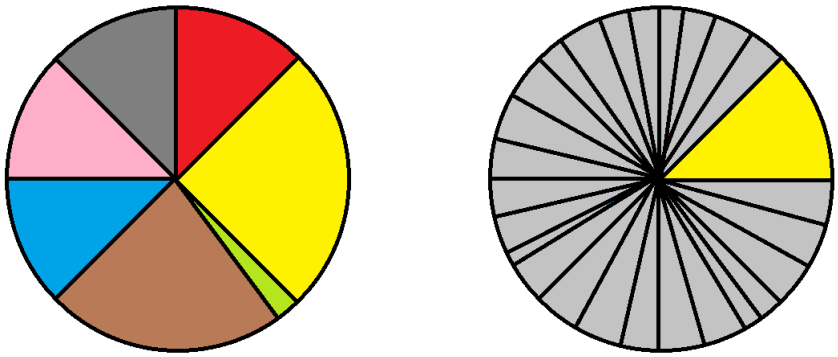


Figure 2.3: Two visual representations of roulette wheel selection

Looking again at Figure 2.3, but this time the right-hand wheel, we can see the problem with roulette wheel selection. If there is one population member that is significantly fitter than the group average, the chances of it being chosen aren't high enough. For example, if we want to choose a third of the population to take to the new generation, there is a good chance that strongest population member (yellow) will not be chosen.

Elitism

A fix for this problem is to introduce elitism. This is when the fittest population members are immediately chosen for the next generation, and then roulette selection is performed [2, 3]. This is not perfect because if you have one population member with fitness 90, and then twenty members with fitness 89, and six hundred members with an average fitness of 10, then you still only guarantee that the single 90 fitness population member will be carried to the next generation. We could of course say that the two top fitness ratings will be considered, but again the problem could be a 90 fitness member, an 89 fitness member, nineteen 88 fitness rated members and six hundred members with an average fitness of 10. Elitism is unfortunately more of a workaround than a fix.

2.2.3 Conclusion

I will implement all of the above selection methods, and test each to determine which results in the best final result.

2.3 Crossover

Crossover has fewer decisions to make than selection. To create a new population member we must select two population members (that were selected during the selection stage), look at their list of moves, and decide which moves from each population member to use for the new population member. The decision to make is in when to use the first parent's data, and when to use the second parent's data. (By data I mean the genotype.)

In the simplest solution, a random n would be picked that is less than the smallest parent's data. The first parent's data would be copied until position n, and then from there the second parent's data from n until the end would be copied. In figure 1.1, the value n is the crossover point.

The immediate problem that can arise from this with the chosen representation is that is if a crossover point is chosen that isn't between two moves (where a move is a set of 4 integers), then the new move that is created on that point has a high chance of being invalid. How much of a problem this is will be discussed in the results section after I have tried and compared the completely random approach, and the approach that

continues to randomly select numbers until one which is between two moves.

Another consideration is how many crossover points should be used. The simple approach uses a single crossover point, but when the data size of a population member can be up to 2000 integers, better results may be obtained by crossing in multiple places. These places could be completely random, or selected such that they are between moves. How many positions is optimal will be discussed in the results section.

The final consideration is how many parents should be used. Up to this point only two parents has been considered, but there is no reason why three, four, five or even more parents could be used. Of course this would require multiple crossover points, and each crossover point could Again this is another question that will be answered in the results section.

2.4 Mutation

After selection and crossover, we have a new generation. A small amount of mutation within the new generation allows new solutions to be found that are not currently within the population. Mutation ensures that convergence at a local maxima won't prevent the population from moving to a higher fitness MitchelGA. It may take many generations, but with a decent mutation operator eventually the local maxima will be escaped and fitter members can be generated.

The naive mutation operator will move across every population member, and every piece of data in the population member (in this case the list of integers), and with a fixed probability change the data item to some other random data. With my chosen representation each integer changed to some other integer between 0 and 40 with a small probability.

The problem with the naive approach has the same problem as the naive approach in the above sections - a completely random mutation has a low probability of creating a valid move, but a high probability of destroying a currently valid move.

As with the above sections, this can once again be counteracted by being a bit more careful with the mutations that can occur. The most simple approach is to treat the list of integers as groups of 4 (i.e. as moves), and take a group of 4 and shift it one position in a particular direction. For example, if we find the move 15 12 12 12 and decide to mutate it. A direction is chosen at random (from up, down, left, right), for this example we'll choose left. The move becomes 14 12 11 12, so the x co-ordinates have shifted once place to the left.

An extension to the above idea is where the number of positions shifted could be chosen randomly, with a normal probability distribution so that shifting 1 space is likely, but shifting 5 spaces is very unlikely.

The final enhancement that could be made is that the moves could 'rotate'. So the horizontal line that is described above could become a diagonal line. If the representation (co-ordinate, direction) was being used then this would be trivial. However with the chosen representation, the new co-ordinates would have to be calculated from the existing ones, which first involves deciding the current orientation of the line, and then determining which co-ordinates should change and by how much.

Chapter 3

The Results

3.1 Introduction

The highest score achieved by my genetic algorithm was 81. In this section I will discuss my solution and how it performed with different parameters.

3.2 The Code

The code was written from scratch in C#, and built using the Mono framework. Interfaces were defined for Mutation, Crossover, Selection and Random, which allowed me to try different approaches to each without making large changes to the code. Through the parameters passed into the program I could change which module was used for selection, mutation and crossover. Through these parameters I could also specify other useful parameters such as the population size, mutation rate and selection rate. The full list of parameters is found in the appendix.

3.3 The Environment

A fresh Ubuntu 13.10 installation was used to run the evolutionary algorithm. The computer that it was run on had an Intel quad core i5 2500k, 8GB RAM and a 120GB SSD.

The amount of time that the algorithm took to run was almost completely dependent on the population size. The vast majority of computation time was spent on evaluating the fitness of the whole population with the provided executable file. A population of size 1000 took 6 seconds to evaluate, whereas a population of size 5000 took around 45 seconds. I did consider distributing the population evaluation across multiple computers on a network, but due to time constraints I was unable to do this.

The use of an SSD improved time slightly. When the option of saving after the creation of a new generation was enabled, having an SSD sped the program up greatly over using a mechanical disk drive.

3.4 Results

Because genetic algorithms use randomness to perform mutations and generate populations, it is necessary to run each configuration a number of times and calculate the average and standard deviation. Each of my configurations was run 100 times. The algorithm was stopped after one hour. Ideally the algorithm would

be given longer to run, but because I have many possible configurations to test, this is not possible. In the following sections I detail the results of changing different parameters.

3.4.1 Selection

I tested Highest Ranking Selection, Roulette Selection and Roulette Selection with Elitism. The results showed that contrary to expectations, Highest Ranking Selection performed just as well as Roulette Selection with Elitism. Roulette Selection performed poorly - it gradually approached a score, typically around 20, but then got stuck around there. This happened regardless of selection ratio. Introducing elitism certainly helped, although I found that over as the scores went into the 50's, Roulette Selection with Elitism slowed down whereas Highest Ranking Selection continued steadily. Box plot figure 5.1 shows that Roulette Selection with Elitism has a much higher average population fitness than Highest Ranking Selection. I can't explain why that means that ultimately Highest Ranking Selection performs better.

3.4.2 Crossover

Single point and multi-point crossover was implemented. The former is only suitable for two parents, and so that's all it supported. The latter supported any number of parents.

Interestingly the results suggest that single point crossover doesn't perform much worse that multi-point crossover with two parents 5.1. The number of crossover points in multi-point crossover was for each crossover chosen randomly. The reasoning behind this is that it was difficult to measure any difference in performance when a fixed number of crossover points was chosen. The results were usually the same ± 5. I set the random generator to choose between 1 and $(genotypelength)/16$, as anecdotal evidence suggested that this worked well.

When the number of parents increased, the results weren't much difference, except that the average population fitness was a lot higher (1.58 for 2 parents, 8.56 for 10 parents), as shown in Figure 5.1.

3.4.3 Mutation

Three mutation operators were implemented: Shift One Space, Shift N Spaces, and Shift N Spaces and Rotate. The former moves the position of a move by one position in any direction. Shift N Spaces chooses a random value between 1 and 5, and moves it up, down, left or right by the random amount. Shift N and Rotate does this, and then changes the direction of the line (so where it may have been horizontally, it could change to vertical).

The operators performed best in the order described above. Shift One Space had a significantly higher median and maximum value, as well as a higher population fitness average. Shift N Rotate performed significantly worse, and Shift N worse still. This shows that smaller, more subtle mutations work better for increasing the fitness of the population. Figure **??** visualises this.

As an experiment, I added a function that with a certain probability completely overwrite the population member's genome with some completely random moves. I found that with the right probability of occurring (around $\frac{1}{20}$ of all mutations), this increased the population fitness after 10,000 iterations as it helped to prevent getting stuck at a local maxima. The box plot in figure **??** compares an extreme example of having a one in five chance of generating a whole new random population member with not doing this at all, and the results are surprising - the former performs better (median of 22 against 10, population fitness average of 2.16 over 2.77). It should be noted that it seems to create a few good population members, although the average fitness is lower.

Finally I investigated the effects of adjusting the mutation chance. The algorithm goes over each population member and decides to mutate it with a certain probability. The results seem to be in line with the mutation operator results, and suggest that fewer mutations is better than a lot of mutations. With a 1 in 100 probability of mutating a population member, a population fitness median of 34 was obtained, and an average fitness across the population of 8.5. With the 1 in 20 probability the median was just 25, although the average fitness across the population was 16.28 (see Figure **??**).

3.4.4 Population Size

I expected that a larger population would perform better than a small population. I started with a population size of 500, and results with other settings changed ranged from 16 to 38. With a population size of 500, each evaluation of the population took 4.1 seconds on average, with a standard deviation of 0.11 seconds. Such a low standard deviation suggests that varying other parameters makes much less of an impact on the runtime that the population size.

I then setup an unused computer with an identical specification running the algorithm with a population size of 20,000, and also running the algorithm with the same parameters, but with a population size of 30. According to [**?**], 30 is the ideal population size for the best result to runtime tradeoff.

I expected that it would take a long time to run the 20,000 population algorithm, but that ultimately return a higher population fitness. Unfortunately this was not the case - after running for 4 days it had a score 68, and the score had been stuck on that for 22 hours. Each evaluation of the population fitness took around 5 minutes, although it was competing for resources with the 30 population instance. The 30 population instance had a score of 81, although it also had been stuck on that for around 15 hours.

3.5 Conclusion

The result of 81 is a long way from the world record of 178 [**?**]. I believe that with enough time and computational power that I could come close to the record, and possibly even surpass it.

This experiment has provided surprising results. I did not believe that a population size of 30 would be suitable, particularly when the genotype length is 2000 integers. What I discovered though is that the number of generations is more important than the population size, as it allows for many mutations to occur that ultimately increase the fitness of the population.

Chapter 4

Discussion

4.1 Critique

4.1.1 Overall Approach

From the very beginning of the project I chose to approach the problem with a genetic algorithm. There is another approach that I could have taken though, which is genetic programming. I have more experience with genetic algorithms, which is why I decided to use them. This is a poor way to decide on an approach, as with additional research I would likely have done just as well, if not better, by using genetic programming.

4.1.2 Choice of Programming Language

ECJ is a java based framework for evolutionary algorithms [**?**]. It would have been ideal for this problem, as it reduces some of the complexity of implementing a genetic algorithm by providing a framework. I decided that I wanted to start from scratch in C# as I thought it would be more interesting. While I believe that it probably was more interesting, it is not a proper reason for not using a tried and tested framework. By starting from scratch I likely wasted valuable time that could have been spent improving my algorithm or allowing instances to run for longer.

4.1.3 Statistical Analysis

While building the program I wasn't thinking about writing a formal report, and so not all of the instances of the program had their results recorded. I also suspect that had a lot of the instances had longer to run, that they would have performed better. Unfortunately time was limited as I spent so long writing the code. Early versions of the program also neglected to output the number of the current generation, which would have been very valuable for comparisons between other instances (because if one instance stops at 42 after 2000 generations, and one stops at 45 after 200,000 generations, we will conclude that the latter is better, despite taking so much longer).

4.2 Further Work

4.2.1 Automatic Statistical Analysis

Most instances had their results stored on a piece of paper until I had the opportunity to type them up into the table in the appendix. It would have been very useful if early on I had included a function that automatically logged the results to a spreadsheet. I think that this would also be fairly easy to retroactively implement.

4.2.2 Parameter Picking

Because there are so many parameters, it is too time consuming to thoroughly test every single combination. I tried to get a good range of results, although it is possible that some combinations of parameters that weren't tried would have given better results. A better approach may have been to implement a basic genetic algorithm for choosing the parameters. The only downside to this approach is that it would have taken a very, very long time to evaluate a population!

Chapter 5

Appendix

Parameter	Description	
CROSSOVER_OPERATOR	The crossover operator to use	
SELECTION_OPERATOR	The selection operator to use	
MUTATION_OPERATOR	The mutation operator to use	
RANDOM_GENERATOR	The random number generator to use	
SELECTION_RATIO	The fraction of population members to keep into the next generation	
POPULATION_SIZE	The size of the population	
MUTATION_CHANCE_INVERSE	The inverse of the fraction of population members to mutate	
NEW_DURING_MUTATION_INVERSE	The inverse of the fraction of population members to generate from scratch during mutation	
POPULATION_FILE	The file to load the population from	A population file,
SAVE_FILE	The file to save the population to at the end of each generation	A p
CROSSOVER_PARENT_COUNT	The number of parents to use during crossover	

[**?**]

test	test	test

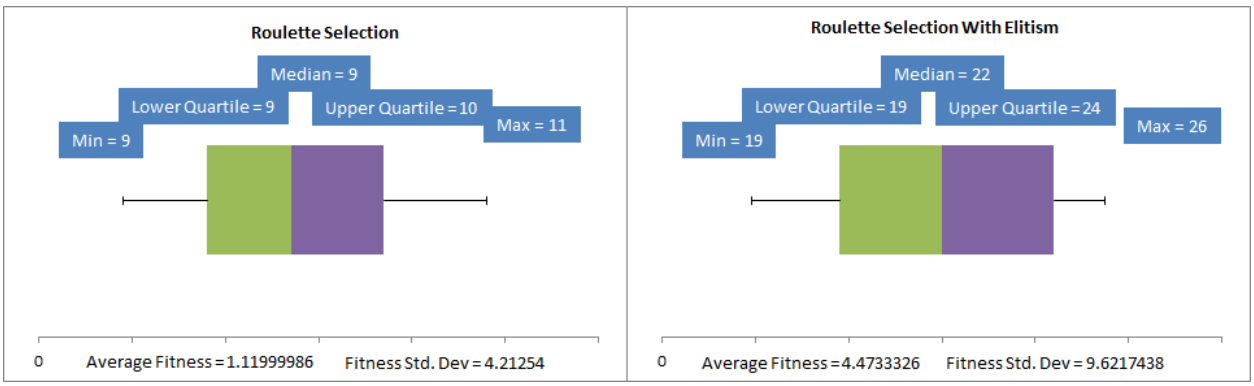
Bibliography

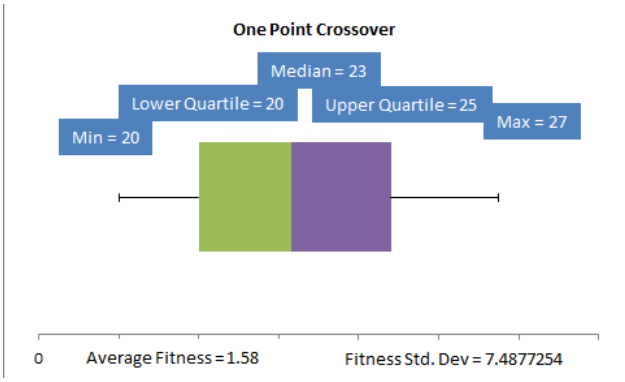
[1] Christian Boyer. Morpion solitare, 2013.

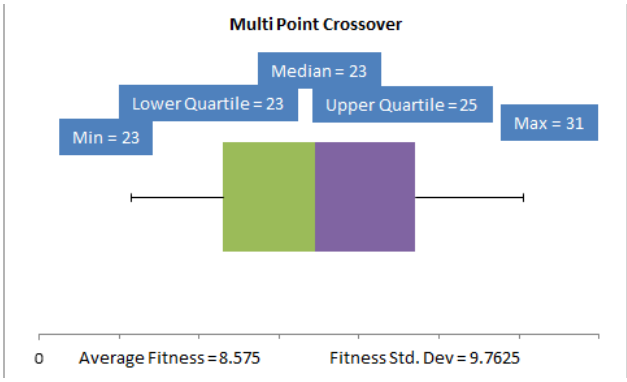
[2] Smith Eiben. *Introduction to Evolutionary Computing*. Spring, 2007.

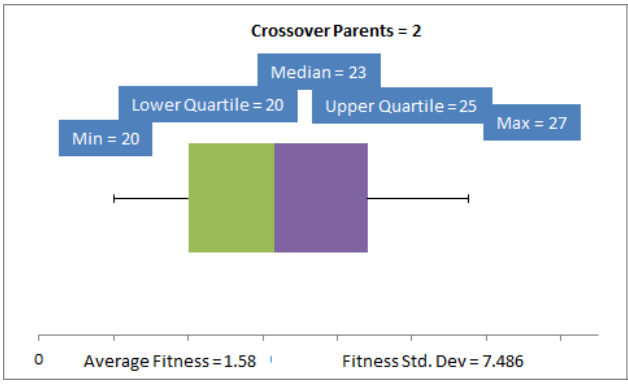
[3] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1998.

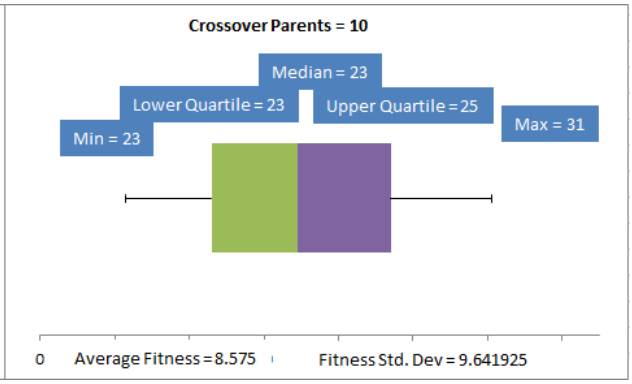
5.1 Box Plots of Results

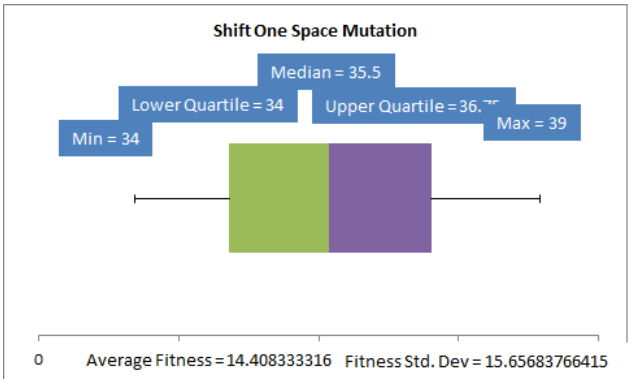


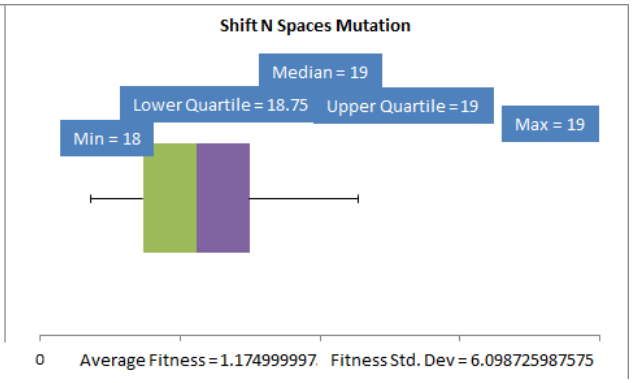

One Point Crossover
Median = 23
Lower Quartile = 20
Upper Quartile = 25
Max = 27
Min = 20
0
Average Fitness = 1.58
Fitness Std. Dev = 7.4877254

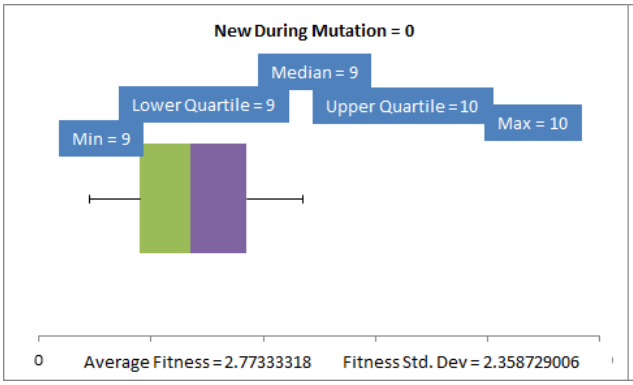

Multi Point Crossover
Median = 23
Lower Quartile = 23
Upper Quartile = 25
Max = 31
Min = 23
0
Average Fitness = 8.575
Fitness Std. Dev = 9.7625

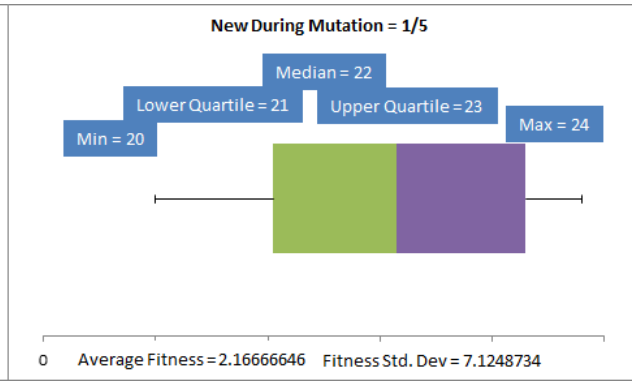

Crossover Parents = 2
Median = 23
Lower Quartile = 20
Upper Quartile = 25
Max = 27
Min = 20
0
Average Fitness = 1.58
Fitness Std. Dev = 7.486

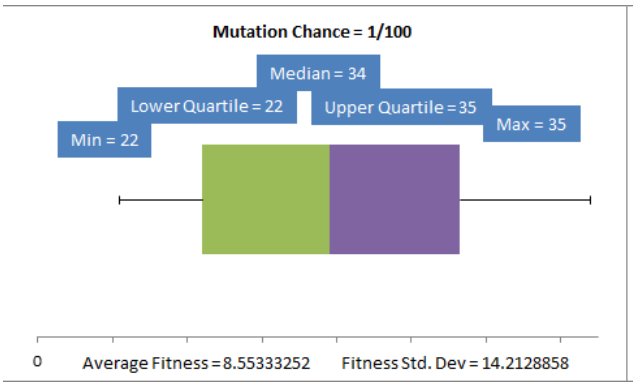

Crossover Parents = 10
Median = 23
Lower Quartile = 23
Upper Quartile = 25
Max = 31
Min = 23
0
Average Fitness = 8.575
Fitness Std. Dev = 9.641925

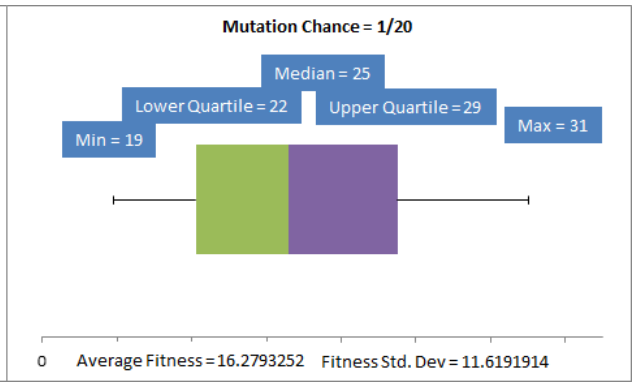

Shift One Space Mutation
Median = 35.5
Lower Quartile = 34
Upper Quartile = 36.
Max = 39
Min = 34
0
Average Fitness = 14.408333316
Fitness Std. Dev = 15.65683766415


Shift N Spaces Mutation
Median = 19
Lower Quartile = 18.75
Upper Quartile = 19
Max = 19
Min = 18
0
Average Fitness = 1.174999997
Fitness Std. Dev = 6.098725987575


New During Mutation = 0
Median = 9
Lower Quartile = 9
Upper Quartile = 10
Max = 10
Min = 9
0
Average Fitness = 2.77333318
Fitness Std. Dev = 2.358729006


New During Mutation = 1/5
Median = 22
Lower Quartile = 21
Upper Quartile = 23
Max = 24
Min = 20
0
Average Fitness = 2.16666646
Fitness Std. Dev = 7.1248734


Mutation Chance = 1/100
Median = 34
Lower Quartile = 22
Upper Quartile = 35
Max = 35
Min = 22
0
Average Fitness = 8.55333252
Fitness Std. Dev = 14.2128858


Mutation Chance = 1/20
Median = 25
Lower Quartile = 22
Upper Quartile = 29
Max = 31
Min = 19
0
Average Fitness = 16.2793252
Fitness Std. Dev = 11.6191914