ECO - Report

Christopher Jones – 40274924

November 2019

1. Introduction

I first decided to analyse the code for the women's team pursuit and try to understand what a good way could be to optimise the code knowing my current knowledge of evolutionary algorithms. When you ran the code, it would display a fitness score depending on how well that team did in the team pursuit. I looked specifically at the Selection, Crossover and Mutation methods to start off with and found that the program was quite slow to run. This was due to selection method that was currently being used to find the parents that would start the evolutionary process off. The selection method in question was tournament which is a fight for survival-esk type of selection. To optimise the code, I would need to find another selection method that would be faster and hopefully better. The Crossover method in the beginning only used a 1-point crossover which gives crosses over at one point from both parents. However, this can result in a bad gene being passed to the child or it might not. By optimising the crossover to a uniform crossover, the child gets a 50/50 chance to inherit a gene from either parent or hopefully get a higher chance to get a good gene. The mutation method used was only mutating one part of the team’s pursuit which dictated when the front rider should switch on a turn to become the back rider. This would give very bad results as there are two parts to the team pursuit, the first being the transition and the second being the pacing. By also having the algorithm evolve and optimise the pacing strategy, it would give a much lower fitness.

I also decided that the population required some more diversity when it came to optimise the program. I was able to do this by using a Saw Tooth algorithm. This allowed the algorithm to get rid of the worst of the population until we were left with half and then re-add the population back to the cap. This adds diversity to the environment in the hope we can get a very good gene to evolve. I was also able to optimise the code by adding a hill climber into the algorithm to see if it was possible to find good optimums within the population it was in. By doing all this optimisation it was possible to drastically decrease the overall fitness score of the team pursuit so that they were using their energy and transitions in the most optimal way.

1. Method
   1. Solution to the Problem

I represented the problem by first understanding how the transition strategy and the pacing strategy affected the race fitness score. I investigated which of the two gave a more dramatic change in terms of fitness by randomly flipping the transition strategy and setting the pacing strategy to a random number. I found that the pacing strategy did this because it had a bigger range of possible values than either being true or false. By doing this representation I was able to find out that evolving one and then switching to evolving the other can help decrease the fitness score if you are say stuck at a false optimum. However, it was only effective if the parameters of the program had a high max iteration because otherwise it wasn't worth doing. I also represented the problem by looking at the default parameters for both the transition and the pacing strategy to give man an idea of what I could try to evolve. I also created my default values just to test what they would do the fitness score. After getting my head around the problem I next looked into the methods of which I could, and potentiality would use to evolve team pursuit and give me a lower fitness.

* 1. Fitness Function

I didn't particularly do very much with the fitness function apart from make it so that a solution that didn't finish the race wasn't just straight up given a fitness of 1000 (the worst fitness). This means that a solution that almost made it to the finish line was given a better fitness than one who was nowhere near because in theory it wasn't a bad solution it just needed a bit more evolving, to make it cross the line. This is by using the getProportionCompleted() function and multiplying it 100 and then taking that away from 1000 and therefore giving a more reasonable fitness for closer solutions.

1. Algorithms
   1. Selection: Roulette Wheel

I used the roulette wheel selection to pick the parents from the population of 20. I used this over tournament because it was much quicker even though it might not have found the most optimal parents as it uses a proportionate selection, it is able to get to the higher iterations faster and perhaps get an equivalent or better solution than tournament for the same amount of time. Roulette works based on adding all the fitness's from population together and picking a random number between 0 and that total fitness. From there we loop around the population again and check while the random number is greater or equal to the sum of the current population’s fitness, we add the fitness of the current population’s fitness to the sum. Once the sum is greater than the random number, we take the position of where the sum went greater in the population as the parent. Two parents are selected every cycle by this algorithm to hopefully make the solution better than the previous. The roulette wheel from my testing (in experiments) trades blows with tournament depending on the other algorithms used to optimise the other parts of the evolutionary algorithm. Because of this I stand by my reasoning of using the roulette wheel for time proposes as it can find parents much faster, which may carry a good gene, which would be good to exploit.

* 1. Crossover: Uniform

I used the uniform crossover to optimise the crossover in the code as much as I could. I did this because uniform has a 50/50 chance of picking a gene from each parent at a given cut and having the child inherit that gene. This could potentially mean that child gets a good gene from the parent which we can exploit more to hopefully give better results in the solution. However, in saying that it could also give the child a very bad gene which we don't to have in the child's chromosome but this is the risk you have to take when using this type of crossover and hope in the next iteration that gene doesn't make it into the next child. Uniform Crossover is a good optimisation because for this solution compared to the one point crossover because it has more chances to hopefully take the best part of the parents where one point has a much higher chance of taking best and worst parts of the parents due to it only cutting the genotype once. With other crossover methods the children tend to resemble the parents genotype but with uniform it can become a mash of both parents. Overall uniform crossover is a good crossover for getting a higher probability of having good genes in the genotype of the child.

* 1. Mutation: Creep

I used the creep mutation to optimise the mutation of the transition and pacing strategies. Creep works by adding or subtracting a value from a random value in the strategy array. However, in the case of the transition strategy there is only true or false so this type of mutation cannot be used in conjunction with this, so instead I just bit flip that flips the transitions current Boolean. In the case of the pacing strategy I started off by changing the value to a random value between the cyclist’s power limit which is 200-1200 watts. This gave a drastic drop in the fitness score but, because it is so random it is more brute forcing the problem and the chance of it always getting a power that changes the fitness score in the higher iterations was not very likely. This meant I decided to change it add a value of 50 + or - to the current power which saw less of a drop to start off with because it was less random but in the longer runs vastly improved the best fitness as it was much easier to find a power which dropped the fitness. I further improved and optimised this by changing the value from 50 to 20 after so many iterations. This was because it became much harder to find a power that dropped the fitness as the margins to decrease the fitness were so small that 50 would jump over them or by sheer luck it might drop the value. This would in the end lead to a sort of brute force approach because the algorithm was just randomly picking which value to add or subtract 50 to/from. By making it switch to 20 after so many iterations it gave a much higher probability of finding a new solution for the child and therefore getting a lower fitness. If I were to run it much longer for testing purposes than I may have even added another if statement to say after 5000 iterations, we changed the value to 10 because 20 might even be too big at that point.

* 1. Saw Tooth

The saw tooth algorithm was used to create diversity within the population that was being evolved. By introducing diversity into the population, it is possible to introduce a chromosome with good genes. This is because I have set the initialise function to randomise what pacing strategy values are chosen rather than using the default. The saw tooth algorithm works by removing the worst fitness in the population every x number of iterations where x is a value chosen depending on how long you plan to run the program for. Once the population hits less or equal to 10 then we want to reinitialise the population back to 20 and add in our new chromosomes. By doing this we are exploring other areas of the population to try and find a genotype that could be exploited to get a better fitness. I found saw tooth only really works in longer runs because it takes time to spawn a good chromosome sometimes and trying to test it within 1000 iterations means we are adding and removing chromosomes out the of population too quickly for evolution too really take effect. Overall the saw tooth diversity was used to optimise the algorithm through its ability to have good exploration and allow for good exploitation of different areas of the search space but requires longer runs for it to really become useful to the evolution.

* 1. Hill Climber

The hill climber algorithm is used to find "hills" which are known local optima in the search space. These optima are points where the fitness gets better because it has a better solution. I used this to try and find areas of the search space that increased the fitness of the children to get me a better fitness score. It works by finding the child's current fitness and then mutating that child using a mutation method of your choice (I used creep for it). From there we get the child's new fitness and check if it is less than the old fitness and if it then we take the child's new fitness as it gives us a better fitness. if the new fitness is higher (higher is worse) then we want to mutate the child again until it is lower. By using the hill climber we are able to make our way up hills to lower our fitness however, the algorithms has its flaws because it relies on their actually being a better solution in its search space but, it may have reached a false peak or false optima of which the hill climber is now stuck. If we were to use simulated annealing this problem could be overcome but I didn't implement this. Overall, I used the hill climber to optimise my code by trying to find paths that could lead up the peaks to give a better fitness of which this algorithm can be good at finding.

1. Experiments and Analysis

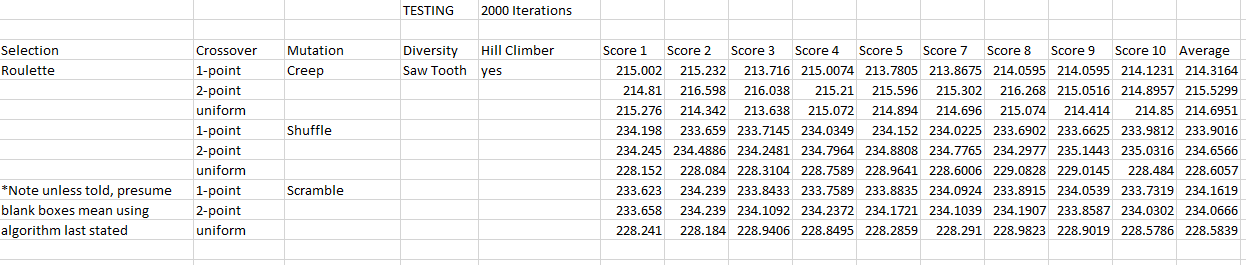
I ran 20 different tests with different parameters each time and then ran each of those 20 tests 10 times for a total of 200 tests. I conducted the tests with a baseline of 2000 iterations, so that the diversity and hill climber may be more useful to getting a lower fitness. I did these tests to investigate the performance of my algorithm using different types of mutations, crossovers and selections. This allowed me to find out which algorithms did more to improve the fitness than others and help to see what parts functioned the best and gave me the lowest fitness score. Here are the results I got using both Roulette wheel and Tournament selection with the respective parameters.

Figure 1: Roulette Wheel Selection

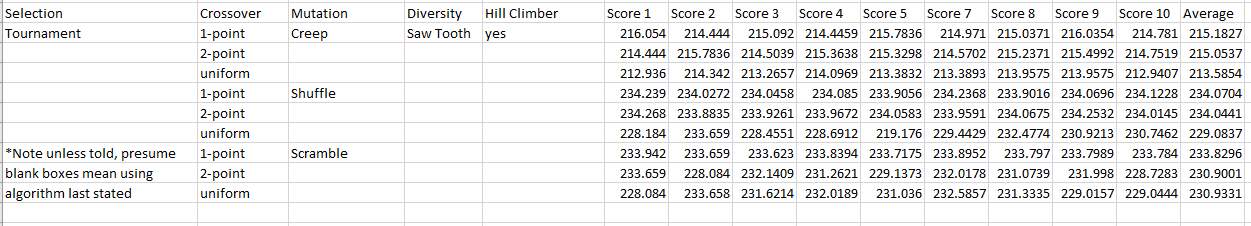
As can be seen in figure 1, the best algorithm consists of one of 3 crossovers with the creep mutation plus saw tooth and the hill climber. This is because the creep mutation doesn't just randomise or swap the value in the pacing strategy which is perfect for this kind of problem where we need to tweak the value. I did find however, that the crossover didn’t make that big if an impact on the average scores for the creep mutation, but uniform seemed to work better for the scramble and shuffle mutations. The shuffle and scramble mutations just don't seem to suit a task such as this and this is heavily reflected in their fitness scores. Basically, anything that isn't the creep mutation didn't do very well but, seeing as I didn't conduct these experiments without the diversity or the hill climber, I don't know how much they may have influenced or affected the results.

Figure 2: Tournament Selection

As can be seen in figure 2, the best algorithm consists of one 3 crossovers with the again the creep mutation plus saw tooth and the hill climber. As explained above the creep mutation is just apparently the best mutation method for this type of evolution. Even though the averages of the crossover were still quite close with the creep mutation there is almost a 2-fitness difference (215 and 213). This may not seem that significant but at those lower fitness's scores that is quite a huge leap and even on score 1 the uniform crossover achieved 212.93 which is a very good score. Every other kind of mutation and crossover was again lacklustre and achieved very high fitness's

From these tests we can see that the tournament selection did indeed get the lowest score using uniform crossover, creep mutation, saw tooth and hill climber with 2000 iterations but that doesn’t mean to say roulette is bad, it just wasn't able to use its speed to find a better solution faster. Another thing to note is that the tournament took almost double the amount of time as the roulette to get to 2000 iterations. This goes back to my problem with tournament and its very slow finding of parents.

1. Conclusion

Overall, the best race time I achieved throughout the entire coursework was 208.25 but, this was bear in mind after 50,000+ iterations because that is the kind of range where diversity can really start making the difference in the exploration. I achieved this score with the Roulette Wheel selection, Uniform crossover, Creep mutation, Saw Tooth and the Hill Climber. I believe the three main factors that helped me get this score were diversity, lowering of the creep +/- value to the pacing strategy and just a little bit of luck that I managed to get a good gene pool for that run and didn't get stuck on any false peeks.

I think my approach worked but, I feel I would have liked to have tried different diversities such as islands etc and another type of mutation like creep that would have impacted the algorithm a lot more than scramble or shuffle. I found it interesting how there is quite a lot of randomness in terms of how lucky a run can be and how unlucky another can be. One run may start getting really low fitness's much faster than another but then gets stuck because it is unable to find a better solution and that is ultimately, what evolution is in the end... sometimes you get lucky with the gene pools and other times you don't.

1. Future Work

If I was to look at this project again then I wouldn’t 100% look into islands and changing the fitness function some more as I didn’t mess around with the energyleft() function. This could have given me a lower fitness if I had perhaps played around with it, but I decided against it. I would have also liked to have added the simulated annealing which is basically a better version of the hill climber because it allows the algorithm to go back down a hill if it is stuck to hopefully find the true optima. I believe this could have found better solutions overtime because I think the hill climber did get stuck most of the time. I would have also liked to look at the replace function to see what I could have change there and how the algorithm replaces the worst fitness, but the worst fitness might only be the worst because they didn’t quite finish. So, I believe by changing that and the fitness function it could be possible to get sub 210 more frequently. The last thing I would change would be the how much creep mutation adds to the pacing strategy. I think with a bit more tweaking and debugging it I could have defined the iteration to switch at the most optimal time because now it is literally just random. As I mentioned in my conclusion, I think I would have also looked for some other mutations that could fit this kind of evolution. Overall, I am happy with how I approached the problem and will take my 208.25 fitness score.