Computational Intelligence Coursework 2020 Christopher Jones

40274924

ACM Reference format:
40274924. 2020. Computational Intelligence Coursework 2020
Christopher Jones. In *Proceedings of Coursework, Edinburgh Napier University, April 2020 (SET10107),* 6 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 APPROACH

Coming into this project I had previously done Emergent Computing for Optimisation with Emma Hart and my Honours project centered around evolutionary algorithms in soft robotics. So I had a good understanding of how evolutionary algorithms worked and what sort of methods I should be using in my design. Saying that I did some background reading into other selection methods that I could possibly use other than a tournament selection that I implemented before. This was where I found the roulette wheel selection which is a fitness proportionate selection method compared to the tournament which selects the highest fitted candidates by running several tournaments (survival of the fittest). I also did a bit of background reading into t-testing which I would be using for the analysis of my results. T-testing is a type of statistic that can be used to determine if there is significant difference between the means of two groups, why they may be related in certain features.

I added in several new operators to help improve the evolutionary algorithm. As the algorithm didn't feature a proper selection method I implemented both a tournament selection and roulette wheel selection. As mentioned previously the tournament selection runs tournaments to find the candidate which has the best fitness. Whereas the roulette wheel selection assigns a fitness to all possible solutions or chromosomes. This fitness level is used to associate a probability of selection with each individual chromosome.
I also added in a few different crossover methods as the current one that came with the code just returned an exact copy of both parents. I chose to add one point crossover, two point crossover and a uniform crossover. The one and two point crossover randomly select a point (2 points for two point) in the parents chromosome that will be split and placed in the child's chromosome. Uniform is similar but instead of splitting it in places, has a 50% change to take a gene from either parent and this goes along each gene in the two parents chromosomes until the end is reached.
I also added in some more mutation methods as the one that the algorithm started with wasn't the best for this chromosome. I added a creep mutation, a shuffle mutation, and a scramble mutation. The creep mutation works by picking a random gene in the child's chromosome and then adding/subtracting a random number between two values. The values chosen are typically custom to fit for each algorithm, really depends on how the chromosome is built. The shuffle mutate works by simply choosing a random gene in the chromosome and swapping it around with another gene in the child. Basically, is swapping the two places of those genes in chromosome. The final mutation method I implemented was the scramble mutation which simply takes two points in the chromosome and then scrambles the order of them. This intern means the genes from those two points are shuffled about to hopefully give a better fitness score.
Another operator I added was changing the replace function so that it replace by the worst fitness individual rather than the worst member in the population. This works by finding the worst individual in the population and then removing them. Then the new child can be added into the population at the index of the old worst individual.
The final operator I added was a bit of diversity into the algorithm in the form of the saw tooth algorithm. Diversity is good as it increases the exploration of the search space rather than just exploiting your best individuals which could intern get you stuck at false optima. This algorithm works by removing the individual with the worst fitness after X number of iterations. This can be custom to how many total iterations you plan on doing in the evolution. Once X has been met the worst is then removed and it will repeat until the population has normally reached half of what it started on. Once this point has been reached the algorithm will reinitialise the population back to the starting value. Overall increasing the exploration in the search space to hopefully find a better solution.

The evolutionary algorithms parameters were required to be changed and new parameters needed to be added. One such parameter was a "tournamentSize" parameter due to the fact I was using a tournament selection. This is to set the size of the tournaments that the selection uses to find the parents. This variable was set to 6 as you don't want the tournament size too be big or too small for the given population size.
I also added a "crossoverProbability" function as well which can be used to see if a crossover should be done on this iteration or not. Again this parameter can be custom to how frequently you want to the crossover to occur. To help with this I added a function that can be used to get a random double given the time system clock.
I also edited some of the other parameters during the parameter tuning such as the "mutateRate" which indicates how often how often a mutation of the genes for the child should occur. I also tuned the "crossoverProbability" and the "tournamentSize" parameters during the parameter tuning.

The neural network parameters were tuned and added accordingly

to give the best results possible. For instance the amount of hidden layers that the neural network uses was adjusted from 5 to 3 because that seemed to actually give better results on average. By doing this I am decreasing the amount of hidden nodes in the neural network to ensure I have the appropriate amount for this problem. Having a higher amount of nodes may improve accuracy or it might not, it depends on the complexity of the problem and I believe 5 is too many for this problem. By having too many then we are over fitting the problem and therefore cant generalise the new unseen data, it can only be doing the training data.
I changed the activation function as well for the neural network to relus algorithm. This will only return X if it is bigger than 0. If it is smaller than 0 then it will just return 0 instead. This is to help with the overall fitness by determining whether a neuron in the network should be activated("fired") or not, based on the X value.

2 EXPERIMENTS AND ANALYSIS

2.1 Initial Results

I ran the algorithm 10 times for each parameter that I changed and recorded the best fitness for the training environment and then the fitness for the test environment. I decided for both tournament and roulette wheel selection I would look at the spread of the fitness using the uniform crossover only. Each box plot graph down below shows how the mutation method affected the spread of the fitness's.

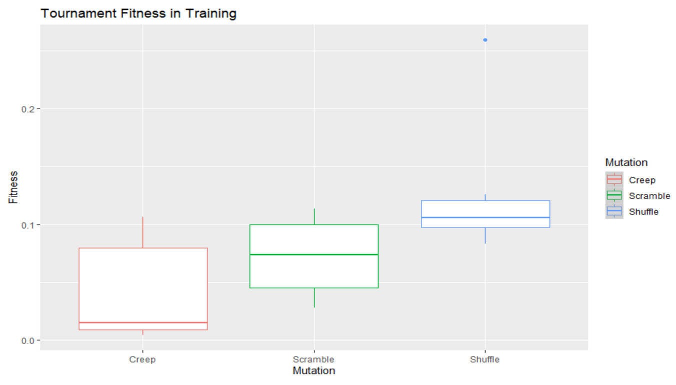

Figure 1: Tournament Selection in Training Fitness using uniform crossover only. Red = Creep Mutation, Green = Scramble Mutation, and Blue = Shuffle Mutation by the fitness achieved

Figure 1 shows the tournament selection spread of fitness's for the training environment. The shuffle method for this graph appears to have the smallest spread of fitness's compared to the others. Although it was the most consistent of the 3 mutations, it was also consistently worse than the other 2 in terms of fitness. The creep mutation had quite a big spread but it did get the smallest fitness achieved from the mutations and most of its values were to the lower quartile of the box plot. The scramble mutation was the middle man with it getting a spread that wasn't as big as the creep but not as small as the shuffle and it got fitness's that were in between the two as well.

Figure 2 shows the tournament selection spread of fitness's for the test environment. This graph paints a very different picture

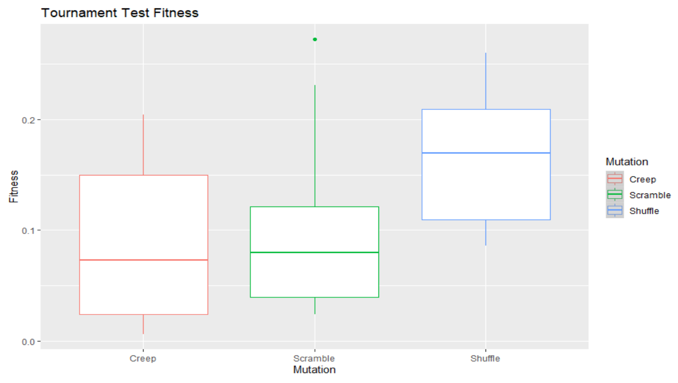

Figure 2: Tournament Selection on Test Fitness using uniform crossover only. Red = Creep Mutation, Green = Scramble Mutation, and Blue = Shuffle Mutation by the fitness achieved

compared to figure 1 which ran the training environment. The spread of the fitness's was a tad more even across with 3 mutations with them all getting far bigger spreads than their training counterparts. This time the scramble mutation was the mutation that had the worst fitness with an out liar value. However, the scramble mutation also had a big range difference between where the main bulk of the data was and the worst values it got. This shows the algorithm for the test environments may have reached a false optimas in while running the test. This does seem to be the case for the other mutations as well, with them all having quite a big range difference which shows them perhaps getting stuck on those runs. The creep mutation again had the best fitness overall but this time the data wasn't so low on the quatertile range compared to figure 1.

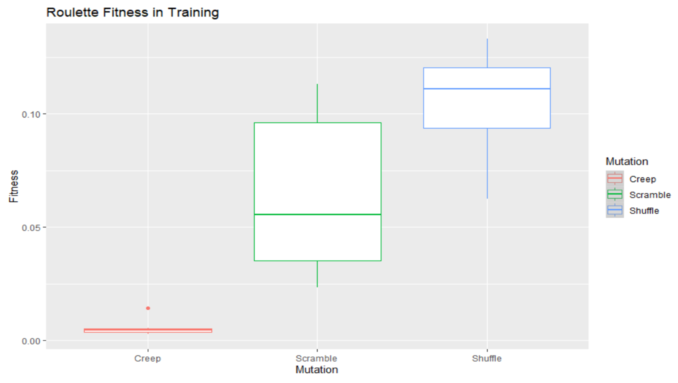

Figure 3: Roulette Wheel Selection in Training Fitness using uniform crossover only. Red = Creep Mutation, Green = Scramble Mutation, and Blue = Shuffle Mutation by the fitness achieved

Figure 3 shows the roulette wheel selection spread of fitness's for the training environment. This graph shows clearly that the creep mutation for this environment had a little to no spread at all meaning its fitness scores were very constant across the 10 runs. It had one out liar value but even it wasn't that far away from the bulk of the other fitness's. The scramble mutation had a much bigger spread of fitness values compared to the other two tournament environments. Its spread meant it had fitness scores that were good and bad but the worst scores came from the shuffle mutation. It had quite a mediocre spread of values and had fitness scores that were on par with the scramble mutation however, most of its values were in the upper quartile which shows that it was semi-consistently getting bad fitness's. Compared to the fitness scores of the tournament training environment the roulette wheel actually got a lower fitness but had much bigger spreads of values. This shows the difference between the two different ways of selecting parents.

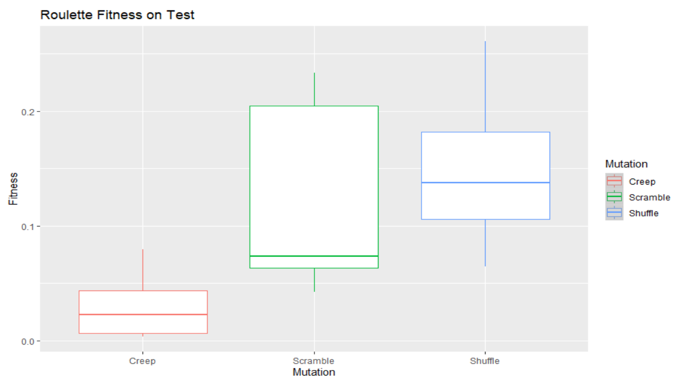

Figure 4: Roulette Wheel Selection on Test Fitness using uniform crossover. Red = Creep Mutation, Green = Scramble Mutation, and Blue = Shuffle Mutation by the fitness achieved

Figure 4 shows the roulette wheel selection spread of fitness's for the test environment. This graph shows the same trend as graph 3 with the creep mutation having a very small spread when compared to the other mutations. It did have some data that was more in the upper end of the quartile however, the bulk of its values were in the middle of the Q1 and Q3. The scramble mutation didn't have any out liars but a lot of the data was to the lower quartile. This shows that some of the values were consistent but it had a couple which increased its spread and made it quite a bit bigger. The shuffle mutation had values which very bad and some that were alright. It as per usual achieved the highest fitness out of all the mutations. The main bulk of the values was however like the creep very consistent but just had values that went from very high and semi low.

Overall, what we can see from these graphs is that my algorithm whether it is using a tournament or roulette wheel selection method should be using a creep mutation for the best results. The mutation is the parameter that will change the chromosome the most and has the best change of getting a better fitness. Another thing I have learned is just because a fitness looks good on paper doesn't mean it will be able to land on the landing in the simulator.

2.2 RTSNE

I used R Studio which is as an integrated development environment for R, a programming language for statistical computing and graphs along with the Rtsne package. This package can be used as a method for constructing a low dimensional embedding of high-dimensional data, distances or similarities. This means I can use to see if there are groupings of data from different parameters. The graphs use both tournament and roulette wheel selections but colour the data by mutations (figure 5) and by crossover (figure 6). Due to the amount of columns (which I limit to 5 tests for this) the perplexity could only be set to 2. The perplexity is able to make the graph more local or global depending on the value and in this case it is quite local. This is also a degree of randomness to these graphs so no two graphs will be the same.

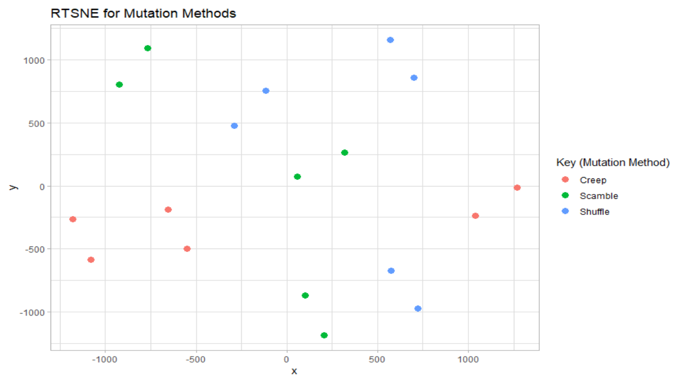

Figure 5: RTSNE graph for Mutation data. Red = Creep Mutation, Green = Scramble Mutation, and Blue = Shuffle Mutation

Figure 5 shows the R-tsne data for every selection and crossover method but is coloured by the mutation method. This was done to see if any of the parameters share a similarity in the fitness that they ended up achieving. From the graph you can see that most of the creep mutation in a little group in the corner. This shows that their chromosomes may share some descriptors even though they may have been formed using different algorithms but ended up in roughly the same area of the search space. Every other mutation is seen to be in pairs, this could be down to the fact that changing the crossover actually doesn't do too much to the overall fitness. Therefore multiple algorithms that used the same mutation could end up being grouped together even if they used different crossover methods. I am assuming the pairs are the caused by using the same crossover and mutation method but changing the seleciton method which slightly changes the fitness achieved.

Figure 6 shows the R-tsne data for every selection and mutation method but is coloured by the crossover method. This graph is very like figure 5 however, the 1 point and 2 point crossover data is much closer grouped than the uniform. This does make sense because those methods are very similar in how they work where one makes one split and the other makes two. Its again like figure 5 comes in pairs of data however, this can't be said for uniform selection method as this is the only parameter in to have one of its pairs not very close to each other. This suggests that selection methods for

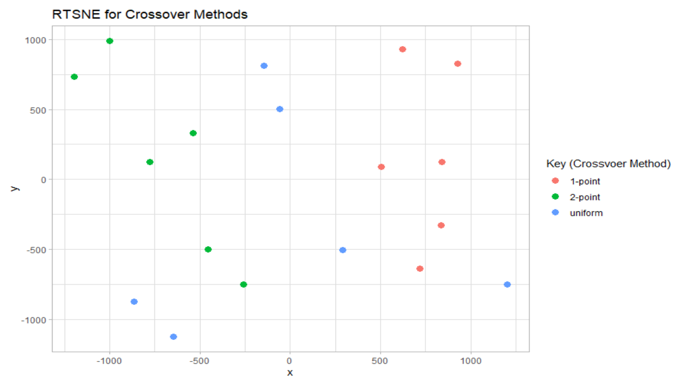

Figure 6: RTSNE graph for Crossover data. Red = 1-point, Green = 2-point, and Blue = uniform

that parameter achieved very different fitness's compared to other parameters selections methods which got similar.

2.3 Parameter Tuning

For the purpose of the tuning I will be using the algorithm that gave me the best results which was a tournament selection, uniform crossover, and creep mutation. I will also be using the fitness that was evaluated in the test environment.

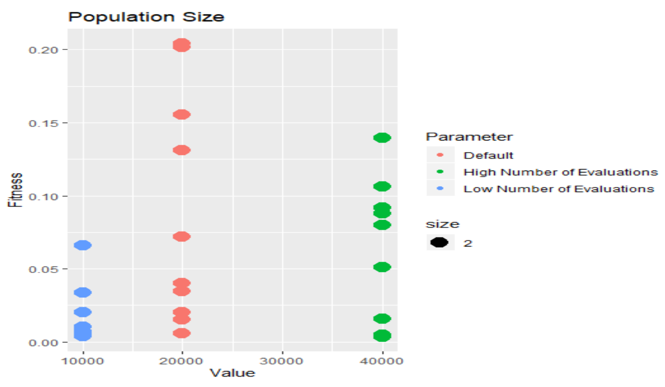

Figure 7: Population Size Parameter Tuning: Low = 20 (Green) , Default = 40 (Red), and High = 80 (Blue)

2.3.1 Population Size. Figure 7 shows the 10 runs for each of values that I tuned for the population size parameter. In the case of population I went for a higher and lower values than the default value of 40. I chose 20 for the low end and 80 for the high end. The graph shows that in the high end the fitness was much better and less spread out compared to the default which has values all over the graph. It also shows that lowering the population also can stop the population being so spread out. This is because the chance of a chromosome getting picked in a smaller population is much higher than a bigger population.

2.3.2 Tournament Size. Figure 8 shows the 10 runs for each of values that I tuned for the Tournament size parameter. In this case

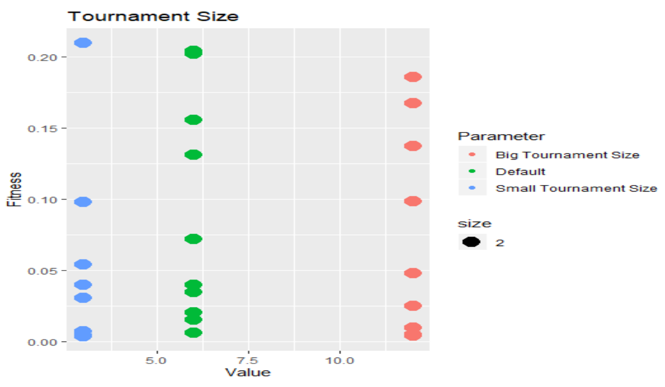

Figure 8: Tournament Size Parameter Tuning: Low = 3 (Blue) , Default = 6 (Green), and High = 12 (Red)

the the lower value was set to 3, the default left at 6 and the high value was set to 12. By looking at the graph the tournament size really hasn't affect the fitness much. It has very sightly reduced the spread in the high value area but it is marginal. The low value seems to have one big our liar from the rest of the data which seems weird. This could be down to the fact having smaller tournaments means an individual may be picked that necessarily have the best fitness and therefore become an out lair. However, this seems to be the only case for the low value, the rest of the values aren't anyway near as spread.

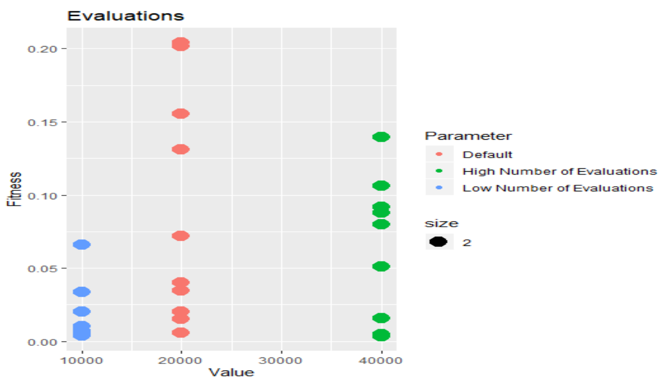

Figure 9: Max Evaluations Parameter Tuning: Low = 10,000 (Blue), Default = 20,000 (Red), and High = 40,000 (Green)

2.3.3 Evaluations. Figure 9 shows the 10 runs for each of values that I tuned for the max amount of evaluations parameter. The values were set to 10,000 for the low end, 20,000 for the default, and 40,000 for the high end. From the graph it is possible to see that the default has the biggest spread and the worst fitness compared to the others. This is weird considering you would think that having fewer iterations would mean the algorithm doesn't have enough time to properly evolve the chromosomes. However, in this case it has actually got better fitness's than the default parameter which ran for longer. The lowest value also beat out the high maximum amount of iterations as that also had a big spread but not as big as the default. The high value didn't have as consistent as the lower range value did but did improve on from the default. Overall, this one is very weird because running the simulator for longer should allow for the chromosomes to be evolved for longer as we have more exploration and exploitation going on, unless the algorithm on these runs reached a false optima.

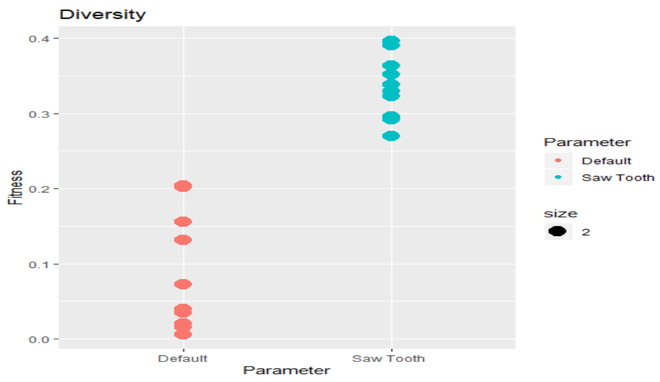

Figure 10: Diversity Parameter Tuning: SawTooth = on (Blue) and Default = off (Red)

2.3.4 Diversity. I added in the Saw Tooth algorithm in the effort to try and increase exploration of the search space and to avoid exploiting the same individuals over and over again which gets you stuck at a false optima. As I mentioned at the beginning of this report the duration of the evolution isn't long enough to take full advantage of the exploration that the diversity gives you. So from figure 10 it looks as though having diversity on gives far worse fitness's than not having it. However, this is just down to the fact that it was just too fast and not long enough as you needed to remove the worst individual out the population every 100 iterations but this is way too fast as it meant you didn't have time evolved the chromosomes of the new individuals before they were removed. Overall, this project wasn't the best for diversity but as I mention in my future work it would be good to increase the max evolution's quite a lot more than I did and see if diversity can be used to its full affect.

2.4 T-Testing

T-testing is a type of statistic that can be used to determine if there is a significant difference between the means of two groups, why may be related in certain features. The t-test is one of the many tests that can be used to show the purpose of hypothesis testing in statistics. Calculating a t-test requires three key data values and they include the difference between the mean values from each data set, the standard deviation of each group, and the number of data values of each group. The T-test is measured by the the p-value which ranges from 0 (no chance) to 1 (absolute certainty) of significance. So 0.5 means that there is a 50% of some sort of significance or similarity and 0.05 means there is a 5% chance. Thankfully, R Studio has a built in package that is able to do the maths behind a t-test.
Table 1 shows all of the t-testing that was done for each parameter

Selection	Crossover + Mutation	T-Test Score
Tournament	1-point Creep + 2-point Creep	0.8644
	1-point Creep + Uniform Creep	0.5965
	1-point Creep + 1-point Shuffle	0.1013
	1-point Creep + 2-point Shuffle	0.1121
	1-point Creep + Uniform Shuffle	0.1543
	1-point Creep + 1-point Scramble	0.152
	1-point Creep + 2-point Scramble	0.0374
	1-point Creep + Uniform Scramble	0.8699
	2-point Creep + Uniform Creep	0.4346
	2-point Creep + 1-point Shuffle	0.1009
	2-point Creep + 2-point Shuffle	0.1052
	2-point Creep + Uniform Shuffle	0.155
	2-point Creep + 1-point Scramble	0.1606
	2-point Creep + 2-point Scramble	0.03016
	2-point Creep + Uniform Scramble	0.7136
	Uniform Creep + 1-point Shuffle	0.01785
	Uniform Creep + 2-point Shuffle	0.01375
	Uniform Creep + Uniform Shuffle	0.02367
	Uniform Creep + 1-point Scramble	0.03443
	Uniform Creep + 2-point Scramble	0.00357
	Uniform Creep + Uniform Scramble	0.6919
	1-point Shuffle + 2-point Shuffle	0.7675
	1-point Shuffle + Uniform Shuffle	0.645
	2-point Shuffle + 1-point Scramble	0.956
	2-point Shuffle + 2-point Scramble	0.3597
	2-point Shuffle + Uniform Scramble	0.05004
	Uniform Shuffle + 1-point Scramble	0.8261
	Uniform Shuffle + 2-point Scramble	0.2854
	Uniform Shuffle + Uniform Scramble	0.07672

Table 1: T-Testing results: Every parameter has been compared against each other to give a t-test score

using the data from the 10 runs in the test environment. The significance of some parameters was far greater than others, an example of this is the 1 point crossover and the creep mutation against the uniform crossover and the scramble mutation. The p-value for them was 0.8699, this shows a great significance and means they have some sort descriptors in common. Another example of a very, very high p-value is the 2-point crossover and shuffle mutation against the 1-point crossover and scramble mutation. It got a p-value of 0.956 which is very close to being the exact same. This could suggest that these four parameters favoured a certain path during the evolution seeing as their p-values are the highest significance I manged to get.

An example of a parameter having a low significance which means they have very little or nothing in common is the uniform selection with a creep mutation against the 2 point selection with a scramble mutation. It got a p-value of 0.00357 which is getting very close to 0. The closer the value is to 0 then the less significant the two parameters are. So in this case these two parameters didn't share any descriptors.

Overall, the table shows that actually quite a lot of parameters in some way shape or form shared significance. This tells us that the parameters during the evolution may gone down roughly the same paths to reach their final fitness scores. This is mostly true with the shuffle and scramble mutations due to them having quite high fitness's overall (as seen in figure 1-4). It could be that they at one point were in the same search space or close to each other to get such similar fitness's and therefore t-test scores.

3 CONCLUSION

3.1 Mean Fitness for 10 Runs

Run	Selection	Crossover	Mutation	Set	Fitness
1	Tournament	Uniform	Creep	Training	0.012009245
2					0.101510719
3					0.003691746
4					0.007826515
5					0.01215054
6					0.016709729
7					0.072052833
8					0.106421033
9					0.081601302
10					0.006926357
				Average Fitness:	0.042090002
1	Tournament	Uniform	Creep	Test	0.034543719
2					0.201942953
3					0.006006232
4					0.039984634
5					0.015209828
6					0.131276273
7					0.155607692
8					0.209257191
9					0.204046766
10					0.020509266
				Average Fitness:	0.101838455

Table 2: Shows what parameters I used, the results I got with them and the set that the results came from

3.2 Parameters Used

I used a tournament selection with a uniform crossover and a creep mutation to achieve my results. I think they were a very good combination of parameters due to the fact that I was able to land all 180 rockets in the "Go Nuts" section plus landing all 8 in the test, training and random environments.

Overall, I believe my approach to this problem was a success as I was able to get the best result possible in the shape of landing all the rockets. An interesting observation that I made was the fact that the creep mutation was the most consistent at getting the best fitness. This is probably because it is the method that changes the chromosome the most in the way of possible values. From the RTSNE and T-testing it is clear that there are groupings and pairs of data that have a lot of significance and therefore perhaps have a similar chromosome or descriptors. The T-testing also showed me that quite a lot of the operator combinations are actually semi-significant and only a couple really weren't. In conclusion, this shows that even with different EA parameters it is still able to reach an area of the search space that another parameter has arrived at.

3.3 Activation Function

I changed the original activation to instead use Relus activation function which as previously mentioned will set X to equal 0 if X is lower than 0 otherwise if it is greater than just return X. Mathematically, this is defined as $y = \max(0, x)$ and is most commonly used as an activation function in neural networks, especially in Convolution Neural Networks (CNN's). This activation function turned out to be better than the one that was pre-written with it also being significantly faster per an iteration.

4 FUTURE WORK

To improve this algorithm even further I would have liked to ran it for longer to see if I could get diversity to have more of an impact. This is because diversity is designed to be used on algorithms that go on for longer due to them needing to actually be evolved and evaluated. This algorithm goes too fast for the diversity to actually have an affect on the evolution as the new individuals can't be exploited fully in the time before they are removed or algorithm is completed. Following on from this I would like to have added an island model in as well. This allows for groups of individuals to be evolved on separate islands and then after a certain amount of time and have been exploited enough then some individuals are moved onto a new island. This allows for these individuals to pick up traits from that islands population and vise versa. By doing this we are increasing the exploration of the search space. However, just like the saw tooth algorithm it would require the program to run a longer to increase the chances that all of the search space is searched and to avoid reaching false optimas.

I would have also liked to have added simulated annealing in which is a better version of the hill climber. Simulated annealing allows the algorithm to go back down the hill just in case it has reach a false optima (exploring the search space). That is the problem with the hill climber, it can get stuck at false optimas and can't move there after.

I think the replace function is something else that I would have liked to looked into further. In my current build I remove the worst fitness however, although it may have the worst fitness it actually might have been close to landing but ran out fuel just before etc. This means it could be exploited further to make it better but it gets removed before it has a chance. This can also be the case with the best fitness individual as it could have a very good fitness but actually at the peak fitness it can achieve due to too much exploitation. Without, actually seeing the fitness function it is hard to say how exactly it evaluates the rockets but this could be quite good to look at and see if it is possible to make it better.

If I had a bit more time I would have liked to tuned my creep mutation a bit more so it isn't such a big jump between the stages. At the start it should be quite big as the fitness will be bad but once it gets to a certain iteration or fitness then the creep should start changing the chromosome with a smaller value. This is because it requires more fine tuning when the fitness gets to a certain stage and don't need big jumps.

Overall, I think the rest of my choices for methods in the evolutionary algorithm would good choices for this project seeing as I did manage to find a chromosome that was able to land all 180 on the "Go Nuts" section.