**Andrew Turner**

**301249607**

**CMPT 310 Surrey Spring 2020**

**Genetic Algorithm Framework**

The general framework I made myself but took inspiration from - <https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3>

I created several helper functions to help with the process these include functions for reading in the a challenge file, reading in a set of permutations, randomly shuffling a section of a list, calculating average fitness, calculating fitness score and a function to save permutations to files for future use.

For the main driver of the program it has 5 arguments it can take

1. The starting permutations for the program either randomly generate or supplied by the user.
2. The number of generation to make before completion default 1000 but can be changed by the user.
3. Crossover function to be used by the program chosen by user.
4. Mutation function to be used by the program chosen by user.

This program can be run using the following format

TSP\_GeneticAlgorithm.py “generations” “cross-over” “mutation” “Permutations” “Permutations”

Generations: valid integer >= 1

Cross-over Options: ‘1’ for Ordered cross over, ‘2’ Partially Mapped Crossover

Mutations: ‘1’ for Random swapping, ‘2’ for shuffle worst half, ‘3’ for shuffle chunk, ‘4’ Mixed mutation

Permutations: file name of permutation you would like to load if two are supplied combines them into a single set taking the top 2 of each.

**Attempted Ideas**

For the crossover functions I used OX and PMX my first attempts used OX with varying mutations and weights I found the best success in these earlier attempts while using the random swap mutation with a number of swaps ~1. The issue I found with this was after a while it began to converge as eventually they all became closer and closer to the best permutation. The mutations worst half and 100 chunks were experiments of mine worst half shuffled the half of the permutation that performed the worst so had the higher score. The issue I found with this is it shuffled away anything good on that half as well overall it was too large of a mix-up to be constant.

The chunk shuffle was the next experiment I tried but this time I randomly chose a point I then added n to it where n is the size of my chunk if this would exceed bounds I subtracted instead. After that I now had my upper and lower bounds. I took the slice in the bounds and shuffled it then passed back into the permutation. This performed better then worst shuffle but still worse the random swapping since the cities were stagnant and didn’t move around the permutation very much relying on OX for that.

I also tried a method of doing several mutations at once namely worst shuffle and random swap. The idea behind this was since I was finding that my permutations would get stagnant after a while I was hoping this would shake things up and help me move past local minimums but I found it still had the issue that with shuffling so many at once it basically made new permutations.

So I decided to try that principle with chunk shuffling to try and shake off local minimums without creating radically different permutations. With this I mutated with the random swap ~1 swaps then I create two copies of that permutation except I randomly shuffled a chunk ~30. I did this for each permutation generated from the crossover. I then modified this method slightly further by doing a variable chunk swap so I did the one swap ~30 but also performed a larger swap ~60. I also did a similar thing with the random swap so instead of swapping just once I also did a swap for 10 and 25 times. This method was slightly slower than some alternatives but I found I got the best results I refer to this method as mixed mutation

I also tried all of the above methods with PMX for that one I used the implementation from tsp.py in the notes which I found to be the most effective

Reference sites:

<https://www.hindawi.com/journals/cin/2017/7430125/>

**Challenge Problem**

Best Solution Distance: 211,174.1967848562

Total Time: ~9hrs of run time

To achieve this I used PMX with the mixed mutation selecting the top 2 for the next generation originally this got me two solution one at 456957 after 10,000 iterations and another at ~550000 after 1000 iterations. I then used my helper function combine\_permutations to combine the top results of each into a single list of permutations. I then reran the above method with this new list for 100,000 overnight to get a permutation score of 213,578.

I attempted to improve this by creating two permutation lists starting from random using the above method but taking 4 parents instead of 2 which gave me two scores just under 500,000. I then combined that list and ran it for 10,000 iterations which gave me a score just over 300,000. Finally I took my top two permutation sets and combined those together and ran PMX with mixed mutation one more time for 1000 iterations but unfortunately saw only minor improvements giving me my final solution.