Solving the Travelling TSP Salesman Problem using Evolutionary Computing and Optimized Programming - *The Explorer Gene*

Robert Bishop (#200647279)
Caleb Graves (#201652567)
Moustafa Elsisy (#201600418)
Vineel Nagisetty (#201647229)

***Abstract*— Evolutionary computing is an exciting sub-field of computer science that deals with creating robust algorithms that solve various problems. Many implementations of evolutionary computing algorithms exist for the travelling salesman problem. This project seeks to implement various selection and variation operators used by other papers to solve this problem. It also seeks to optimize the implementation using dynamic programming and other python language specific optimizations to achieve faster run-times. The code for this project is open and available [1].**

***Index Terms*— travelling salesman problem, evolutionary algorithm, optimized programming**

I. INTRODUCTION

The travelling salesman problem (TSP), is a common problem that has applications in many real-life situations. The problem is named after a common dilemna that a travelling salesman visiting many houses would have encountered. This salesman would want to find the shortest route that encompasses all these houses to sell his wares. Many approaches exist that aim to solve this problem, including various evolutionary computing approaches. This paper seeks to first implement several variation and selection operators in an evolutionary computing algorithm using optimized code, and then aims to compare these different variation and selection operators to find out the best combination of them, given our implementation.

[1]https://github.com/Zerocrossing/DRD4-7R

II. PROBLEM DESCRIPTION

A. The Travelling Salesman Problem

The Travelling Salesman Problem is a common NP-hard problem. Given a weighted undirected complete graph, the problem is finding the optimal (lowest-weight) Hamiltonian cycle. The number of possible solutions (assuming a fixed start or end location) is $\frac{(n-1)!}{2}$ where n is the number of nodes in the graph (locations to be visited), making the problem incredibly time-consuming to solve by brute-force. Various types of algorithms, including evolutionary computation exist to solve this problem.

B. Evolutionary Computation

Evolutionary computation is a sub-field of computer science that simulates natural evolution. It aims to develop automatic problem solvers that can be applied to a wide range of similar problems without the need for tailoring to the specific problem[1].

Evolutionary computation algorithms are stochastic and can be considered to be part of the *generate and test* algorithm group. There are several flavours of evolutionary computation such as genetic algorithms, evolutionary programming and genetic programming based on data structures the problems are represented as. Nowadays, it is more common to call them evolutionary computing and pick representations based on the problem. There are several key components to an evolutionary computation algorithms[2]:

1) **Representation:** describes how the various candidate solutions to the problem are encoded.
2) **Initialization:** describes how many candidate solutions to create and how to initialize them.
3) **Evaluation:** is the basis for selection and represents the objective function that the population should adapt to.
4) **Population:** represents how many candidate solutions to maintain over the course of the problem solving.
5) **Parent selection:** describes how to select parent solution candidates to generate offsprings.
6) **Mutation:** describes how to evolve some candidate solutions and how frequently. This is useful in order to add diversity to the candidate solution which can hopefully allow them to break out of local optima and lead to better solutions.
7) **Recombination:** describes how to merge information from the parents to the offspring.
8) **Survivor selection:** describes how to select which of the population to continue on to the next generation. They are used to reduce memory usage, and help focus on better candidates in order to find better candidate solutions.
9) **Termination:** describes when to stop the algorithm, and can be based on the number of generations or time or other advanced options.

C. *Related Works*

There are many papers detailing different implementations of evolutionary computing to solve TSP. Abdoun et. al 2012, compared several different mutation operators including twors (swap), reverse sequence (flip) and insertion mutation methods on TSP[3]. They found that the reverse sequence mutation (flip) performed the best, and we used this mutation as our standard one and tested it against several others.

Ahmed 2010 developed a new recombination method that he called sequential constructive crossover operator (scx) and compared it with generalized n-point crossover (gnx) and edge recombination crossover (erx)[4]. He found that evolutionary algorithm using scx performed better than the other two on TSP. We have implemented his crossover and tested it against several others.

III. Implementation

The dataset was given by our professor, Dr. Hu. We primarily used the Uruguay data (734 cities) for our experiments, although we tested using Western Sahara (29), Canada (4663) and Italy (16862) and it worked great on them.

We implemented our code in python 3.5. The repository has a AGPL v3 license and available at https://github.com/Zerocrossing/DRD4-7R/. It is designed to be modular with separate modules for each of the processes so as to easily swap out various algorithms. A high level summary of our implementation is given in Table 1.

Representations:	Permutations
Recombination:	Order, PMX, SCX
Recombination %:	100%
Mutation:	Swap, Flip(inversion), Scramble
Mutation %:	20%
Parent selection:	Roulette Wheel with Elitism
Survival selection:	$(\mu + \lambda)$
Population size:	200
Offsprings #:	100
Initialisation:	Random
Termination condition:	2,000,000 Fitness Evaluations

Fig. 1: Summary of the algorithms chosen for our implementation

Detailed descriptions of the implementation is given below:

1) **Representation:** Given n locations, we can define the route traveled as a permutation of integers [1, n]. We are making a complete cycle, so we can fix the first number. Additionally, the direction we travel does not matter. Our total search space is then $\frac{(n-1)!}{2}$ possible solutions. We will define each solution as a permutation of the integers [1, $n-1$].
2) **Initialization:** We initialize by randomly generating *p* permutations.
3) **Fitness Evaluation:** The fitness of a route is defined as the *total distance travelled* in taking that route. In order to find individuals that maximize fitness, we negate the distance.
4) **Recombination:** We have implemented several crossovers:
    a) **Sequentially constructive crossover (SCX):** Ahmed (2010) introduced this novel crossover to solve TSP[4]. SCX is based on the adjacency information of the parents and aims to preserve the best edges. The algorithm for SCX is given by Algorithm **??** in the appendix.
    b) **Partially mapped crossover (PMX):** the main idea of this crossover is to preserve adjacency information[5]. The algorithm detailing this crossover is given by Algorithm **??** in the appendix.
    c) **Order crossover:** The idea of this crossover is to preserve relative order that elements occur[5]. The algorithm detailing this method is given by Algorithm **??** in the appendix.
5) **Mutation:** We have implemented several mutations, such as:
    a) **Flip (inverse):** Abdoun et. al (2012), called this the reverse sequence mutation and proved this works well on TSP instances[3]. In this mutation, a subset of genes are selected at random and reversed. Although this does not change the distance in the subset, it could help preserve the adjacency information and connect the subset to better edges, potentially leading to better solutions.
    b) **Swap:** involves selecting two allele values at random and swapping their positions[5].
    c) **Scramble:** involves picking a subset of genes at random and rearranging the alleles in those positions randomly[5].
6) **Parent selection (roulette-wheel w/ elitism) :** This algorithm balances selection pressure and exploiting highly fit individuals. It involves selecting the best individual from the population as a parent and selecting the other parents randomly, giving a better chance to stronger individuals. Even the weakest individual has a small chance of being selected, which may help break out of local optima at times. The algorithm from this implementation is given by Algorithm **??** in appendix.
7) **Survivor selection ($\mu + \lambda$):** This algorithm involves pooling all the current generation individuals with the generated offspring, ranking them and picking the top individuals from them to enter the next generation[6]. It is more exploitative since it only picks the best individuals and so needs to be partnered up with mutations and parent selections that are a little more exploratory.
8) **Termination:** We implemented terminations based on the number of generations.

A. Advanced Features

Several code optimization were implemented to run the program more efficiently and display results better:

1) **Checking accuracy:** We found a resource published by researchers at the University of Waterloo with optimal tours for our dataset[7]. We compared our results to this to get an objective insight into the performance of our program.
2) **Pre-calculation of distances:** Before we begin initializing the EA, we saved a significant amount of time by calculating the Euclidian distance between each of the locations and saving the result to a lookup table. Our table consisted of n^2 cells, and

this pre-calculation drastically reduced our fitness evaluation time. We were able to perform this calculation at significant speed by exploiting vectorized numpy operations.

3) **Parallelization and just-in-time compilation:** We used numba, an open source just in time compiler that translates a subset of python and numpy code to fast machine code[8]. Numba also allows for parallelizing code for CPUs, which we took advantage of. We found this sped up the execution by over 5 times.
4) **Animating the solution:** We have also implemented an animation of the best solution every set number of generations as the program tries to solve the problem. This helps visualize the improvements over time and makes for a more intuitive visualization. The animation is included in the source code.
5) **Space Requirements**: We relied on 16-bit unsigned integers to represent our vectors, which slashed our memory usage by quarter compared to the 64-bit integers which numpy defaults to. Unsigning the integers gives us a max representation of $2^{16} = 65536$, which is enough for the number of cities for each country in our data set. In addition, unsigned integers require less machine instructions to perform mathematical operations, furthering our speed[9].

B. Setting up the Experiment

We used a jupyter notebook to set up the experiment. The code for experiment can be found at https://github.com/Zerocrossing/DRD4-7R/Report Viz.ipynb. Different recombination and mutation operators were utilized to run the experiment and compare results for.

IV. RESULTS

A. Recombination Operators

We selected flip mutation operator as the default, due to its promising results from Ahmed et. al 2010, and ran 10 trials using 10,000 generations each using order crossover, scx and pmx recombinations. The average overall results from this trial can be found in figure 2.

Overall, scx has been observed to initially yield in better quality solution than order and pmx. As

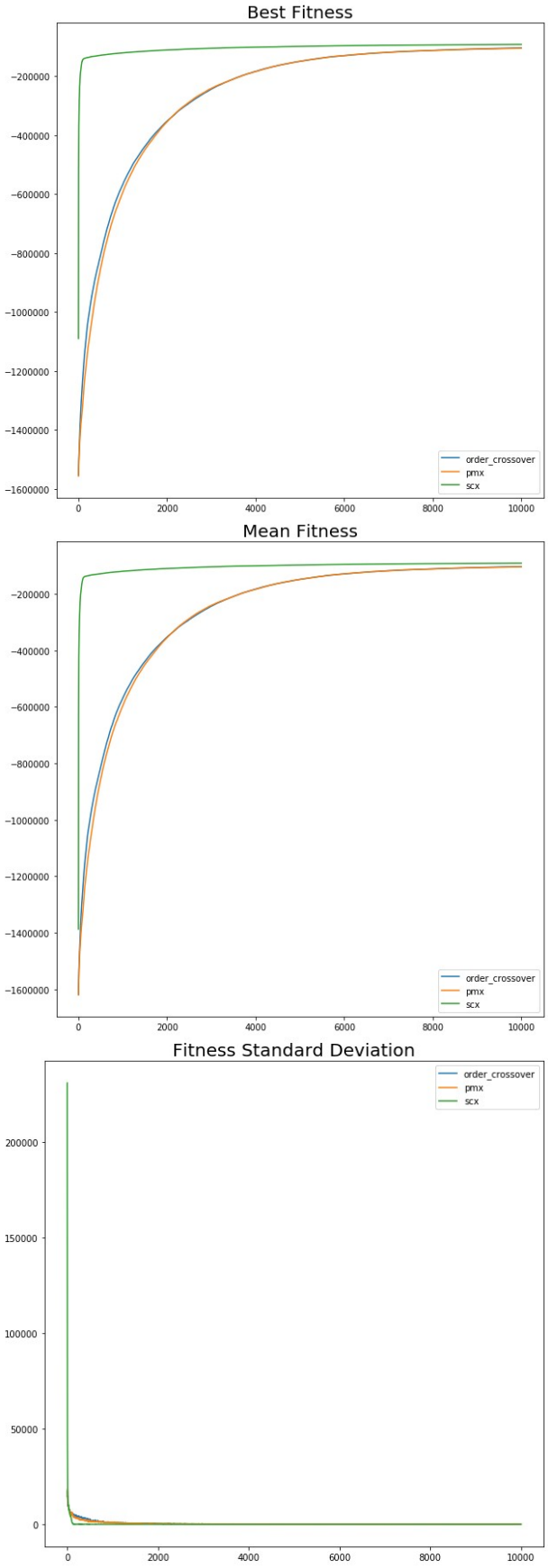

Fig. 2: Crossover methods details

can be seen from the best fitness and mean fitness graphs, scx very quickly lets the solutions improve to a good quality, almost like a smart initialization. Order and pmx crossover perform similarly, and catch up to scx after around 8000 generations. This means scx would be better to use when a quick good solution is needed, and order and pmx could potentially be better for longer runs.

The results of the time taken to compute these operators can be found in figure 3. Scx takes the longest time to compute. Order takes less than half the compute time as scx for 10,000 generations.

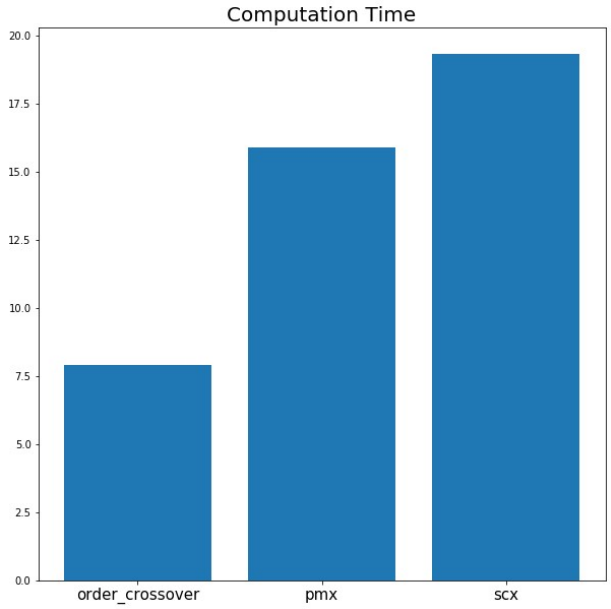

Fig. 3: Crossover methods time taken

B. *Mutation Operators*

We selected order crossover operator as the default, and ran 10 trials using 10,000 generations using flip (inversion), swap and insert mutation methods. The overall results from this trial can be found in figure 4.

Overall, flip has been observed to yield better quality solutions, followed by swap and then scramble. As can be seen from the best fitness and mean fitness graphs, the three perform similarly till around the 200 generation mark, when flip mutation starts performing better than the other two. Due to how destructive it is, and when combined with $(\mu + \lambda)$ survivor selection, scramble mutation does not seem to contribute much to the diversity in the population (as can be seen from the fitness

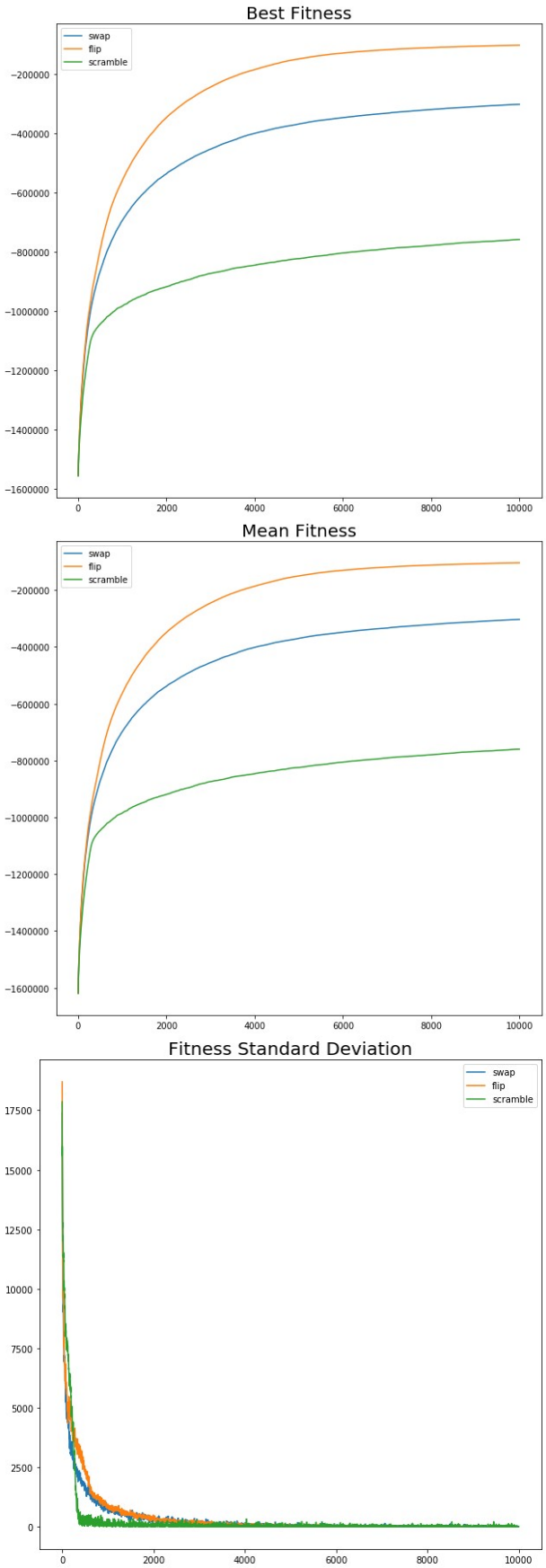

Fig. 4: Mutation methods details

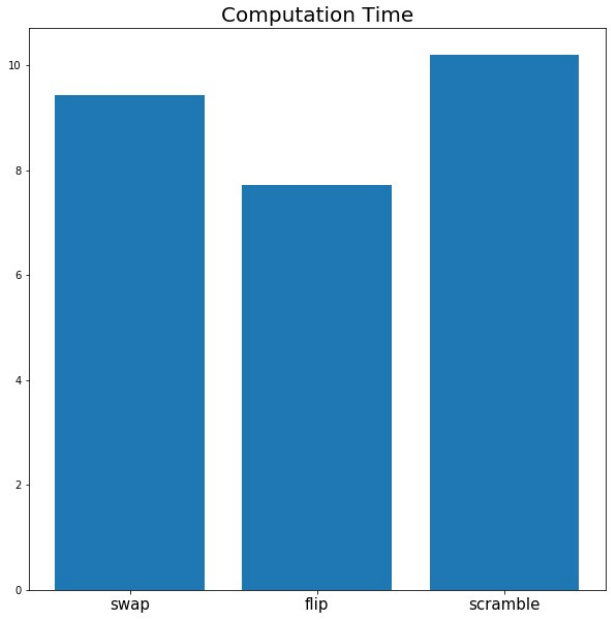

Fig. 5: Mutation methods time taken

standard deviation results after about generation 200), since the mutated individuals tend to have a rather low fitness, and thus stand almost no chance of surviving through selection.

The results of the time taken to compute these operators can be found in figure 5. Flip is found to take the least time to compute, followed by swap and then scramble. Based on this, flip has been found to be the best performing mutation in our experiment.

V. DISCUSSION

A. Further Research Ideas

1) Initialization: One large unexplored area of optimization for us is initialization. We simply use a random permutation for our initialization, and while SCX provides a significant jump in the first few generations, some initialization techniques can start the algorithm off from within a reasonable margin of an optimal path. Paul & Dhavachelvan used a technique called Equi-begin and Vari-diversity to initialize their candidates for TSP, and showed many candidates were instantiated at greater than 80% convergence to the optimal solution [10]. The tactic uses an ordered distance vector containing each city's ordered distance to each other city and uses a random offset at each stop to ensure variance of route. In this manner they were able to initialize their population with a significant genetic diversity while still maintaining a high level of fitness. It's unclear if the computation time involved in initializing the population in this manner is more beneficial than simply running extra generations, and it's possible that the increased genetic diversity of totally random initialization helps us avoid local optima better than a calculated population of this sort.

2) Larger Datasets: Datasets are available for Vietnam & China, containing 22,775 and 71,009 cities, respectively. Further optimization may be required to work with these datasets, as the size severely increases the computation time required for each generation.

ACKNOWLEDGMENT

The authors acknowledge the contributions of Dr. Ting Hu for introducing to the fascinating field of evolutionary computing.

REFERENCES

[1] T. Hu, "Introduction to evolutionary computation." Course Material, 9 2018. Slides.
[2] T. Hu, "Evolutionary algorithm basics." Course Material, 9 2018. Slides.
[3] O. Abdoun, J. Abouchabaka, and C. Tajani, "Analyzing the performance of mutation operators to solve the travelling salesman problem," *arXiv preprint arXiv:1203.3099*, 2012.
[4] Z. H. Ahmed, "Genetic algorithm for the traveling salesman problem using sequential constructive crossover operator," *International Journal of Biometrics & Bioinformatics (IJBB)*, vol. 3, no. 6, p. 96, 2010.
[5] T. Hu, "Other representations." Course Material, 9 2018. Slides.
[6] T. Hu, "Selection and population management." Course Material, 9 2018. Slides.
[7] U. of Waterloo, "National travelling salesman problems," 2017.
[8] Numba, "Numba: A high performance python compiler," 2018.
[9] "c++ - performance of unsigned vs signed integers - stack overflow." https://stackoverflow.com/questions/4712315/performance-of-unsigned-vs-signed-integers, January 2011. (Accessed on 12/09/2018).
[10] P. V. Paul, P. Dhavachelvan, S. Abiramy, and R. Baskaran, "Effective ev population initialization technique for genetic algorithm,"

APPENDIX

Input: parents p1 and p2

c = first node in child = first node in p1;

while *child is not filled* **do**
    α = node after c from p1;
    β = node after c from p2;
    if α *is already in child* **then**
        α = random node not in child
    if β *is already in child* **then**
        β = random node not in child
    if *c to* α *distance* $\leq$ *c to* β *distance* **then**
        set next node in child as α
    else
        set next node in child as β
    end
    set c to next node;
end
return *child*;

Algorithm 1: SCX Crossover Algorithm

Input: parents p1 and p2

set_1 = random subset from p1;
fill child with set_1;
out_set = elements from p1 not in set_1;
for *every element in out_set, starting from element in index of p2 right of set_1* **do**
    fill element in child using the order of p2 and wrap around;
end

return *child*;

Algorithm 2: Order Crossover Algorithm

Input: population p, fitness f,
mating_pool_size s

add parent with maximum f to mating_pool;
a = cumulative probability distribution of f;
c = 1;
while $c \leq s$ **do**
    pick random value r uniformly $\in [0, 1]$;
    i = 1;
    while $a_i < r$ **do**
        i = i + 1;
    end
    mating_pool[c] = parents[i];
    c = c + 1;
end

return *mating_pool*

Algorithm 3: Roulette Wheel Algorithm

Input: parents p1 and p2

copy random subset from p1 to child;
for *every element i in subset indices in p2 not in child* **do**
    find element j from p2 in index of i;
    place i in index occupied by j;
    if *index of j is filled in child by element k* **then**
        place i in index occupied by k
end
fill rest of the elements in child from p2;

return *child*;

Algorithm 4: PMX Crossover Algorithm