CT421: Project 1: Evolutionary Search (GAs)

1 Introduction

The **Travelling Salesman Problem** can be defined as:

> Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city? – This is an NP-hard problem in combinatorial optimization [1]

Genetic Algorithms (GAs) are inspired by the Darwinian theory of evolution. It's purpose is to generate and optimize solutions through processes mimicking natural selection (letting only the fittest/best solutions progress to the next round). GAs are particularly effective for NP-hard problems where the search space grows combinatorially with added parameters. [2]

2 Implementation Details & Design

2.1 Genome Representation

The code is set up to read in the TSP file and convert it into a dictionary where the city index serves as the key and the city's cooridinates are the corresponding values

For the representation (the encoding of potential solutions, which directly represents the candidate solutions in the search space) [2], I treat the genome as a list of numbers that represent the keys in the dictionary. For example, if the TSP file contains 3 cities it would be converted to a dictionary: {1: (565.0, 575.0), 2: (25.0, 185.0), 3: (345.0, 750.0)}, then the genome representations of potential solutions could be [1, 3, 2] or [2, 1, 3], and so on. For this problem, all genomes will have the same length (the number of cities in the TSP) and each genome will contain every unique number between 1 and the number of cities (inclusive).

2.2 Population

To start off, I've chosen to use a randomly generated population to ensure diversity and spread out the sample space as much as possible. It is possible to incorporate domain knowledge into the this problem, where if we know a subset of cities has a definite shortest path, we could use that information. However, doing so can limit the sample space or result in a local optimum (where the solution appears optimal within a certain neighborhood but may not be the global optimum).

2.3 Fitness

After we have generated the solutions (either via step 1 with random initialization or in later iterations through reproduction), we need some sort of evaluation function to discriminate between better and worse solutions [2].

For the Traveling Salesperson Problem, where the goal is to find the shortest possible route, I felt that the total distance of the solution was an appropriate metric for fitness – measuring the Euclidean distance between each city (I chose Euclidean distance because the TSP files specify the use of EUC_2D). With genetic algorithms you typically want to maximize fitness, whereas with distance, we aim to minimize it (since the shorter the distance the better the solution). I decided to implement a rank based fitness evaluation where solutions with the shortest distance is given the highest number and the longer the distance the lower the value assigned. So in a population of 6 the best solution would be assigned a fitness of 6 and the worst solution a fitness of 1.

2.4 Selection

After evaluating the fitness of our solutions, we need a selection function to determine which solutions should progress to the next round for reproduction. Typically, we want the best solutions to progress in order to create better offspring.

I chosen tournament selection, where a subset of the population are randomly chosen and of that subset the solution with the best fitness is selected as the parent. I chose this method because it's easy to implement, it works with the raw fitness values, and doesn't require fitness values to be scaled or normalized like roulette-wheel selection. It also ensures that best solutions are most likely to be selection but also allowing for the occasional weak solution to be selected in order to maintain diversity and keep an open sample space

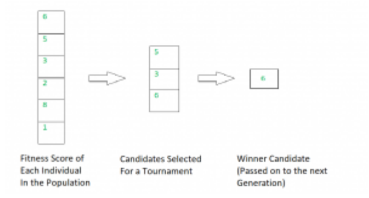

6
5
3
2
8
1
5
3
6
6
Fitness Score of Each Individual In the Population
Candidates Selected For a Tournament
Winner Candidate (Passed on to the next Generation)

Figure 1: Illustrating Tournament selection. [4]

2.5 Reproduction

After selecting the parents, we need to put the parents through the process of reproduction, which consists of crossover and mutation in order to generate new offspring. **Crossover** is used to combine the genetic information of two parents to generate new offspring. The idea is that combining parts of two existing *fit* solutions has the potential to produce better solutions over successive generations. **Mutation** then occurs at a much lower rate than crossover. It is still important as it introduces random changes to individual solutions in the population. This helps maintain diversity in the population and explore areas of the solution space that may not be reachable through crossover alone [2].

Crossover

For the Traveling Salesperson Problem, classical crossover operators are unsuitable because a solution is represented as a permutation of cities, with the key constraint that each city must appear exactly once. Classical crossover methods, such as 1 point crossover, can produce offspring with duplicate cities or missing cities, thus violating the TSP constraints. Because of this I have chose **ordered crossover** and **cycle crossover** as my crossover operators [3]

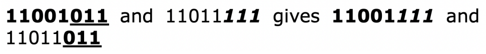

11001<u>011</u> and 11011***111*** gives **11001*111*** and 11011**<u>011</u>**

Figure 2: Illustrating 1 point crossover. [2]

2.5.1 Ordered Crossover

Ordered Crossover for Offspring 1, it cuts out a randomly selected subsequence from Parent 1 and fills the remaining positions in the order they appear in Parent 2, starting from the right side of the cutoff and skipping duplicates. Similarly, for Offspring 2, it cuts out a subsequence from Parent 2 and fills the remaining positions in the order they appear in Parent 1, starting from the right side of the cutoff. [3]

Parent1 - 2 | 7 1 6 | 8 3 4 5
Parent2 - 3 | 7 4 2 | 5 1 6 8

Gives

Offspring1 - 2 | 7 1 6 | 5 8 3 4
Offspring2 - 6 | 7 4 2 | 8 3 5 1

Figure 3: Illustrating ordered crossover.

2.5.2 Cycle Crossover

Cycle crossover identifies the cycle in Parent 1 and Parent 2, maps the numbers to their positions in the cycle, and then fills in the remaining gaps of Child 1 with the numbers in the order they appear in Parent 2. This is repeated for Child 2 by finding the cycle in Parent 2 and Parent 1. [3]

Parent 1: 2 7 1 6 8 3 4 5
Parent 2: 3 7 4 2 5 1 6 8

Step 1: Find the cycle 2 → 3 → 1 → 4 → 6
Offspring 1 (after cycle): 2 _ 1 6 _ 3 4 _

Step 2: Fill in the blanks in the order of Parent 2
Offspring 1 (final): 2 7 1 6 5 3 4 8

Offspring 2 (final): 3 7 4 2 8 1 6 5

Figure 4: Illustrating cycle crossover.

Mutation

Like crossover, mutation must also obey the constraints of TSP. A classical mutation operator, such as bit flipping (where a random gene in the genome is changed), would be inappropriate because we need to ensure there are no duplicate or missing cities. That's why I have chosen **swap mutation** and **scramble mutation** as the two mutation operators. These methods are effective at preserving the overall structure of the route while slightly altering it to explore nearby solutions.

2.5.3 Swap Mutation

Swap mutation randomly selects two genes in the genome and swaps them

Original Genome: 5 4 2 1 3
Randomly selects positions 1 and 4 (value 4 and 3)
Mutated Genome: 5 3 2 1 4

Figure 5: Illustrating swap mutation.

2.5.4 Scramble Mutation

Scramble mutation randomly selects a subset of the genome. This subset is then scrambled (shuffled).

Original Genome: 1 5 6 4 3 2

1. Randomly choose positions 1 to 4 (the selected subsequence is [5, 6, 4])
2. Shuffle the subsequence [6, 4, 5]

Mutated Genome: 1 6 4 5 3 2

Figure 6: Illustrating scramble mutation.

2.6 Genetic Algorithm

I designed the genetic algorithm method to accept a TSP file as a parameter as well as population size, mutation rate, and crossover rate. This makes it easy to test different parameter values.

At the very beginning, the method uses the `time()` function to start a timer for tracking execution time. It then reads the TSP file, converts it into a dictionary format for easy processing, and initializes a random population. Then it creates a `distance_matrix` this precalculates and stores the distance between each city in a matrix so that the distance between `cityA` and `cityF` can be retrieved by `distance_matrix[cityA][cityF]` – This greatly reduced my run time, as now we just have to calculate the distance between each city once, and then look up that value in a matrix instead of having to calculate the distance between each city in each genome, in each population in each generation.

Then in the generation, it calculates the total distance and fitness of each solution in the population. Then it moves on to reproduction where it calls upon the `select_parent` (tournament selection) method, to select two parents. The parents then have a `crossover_rate` probability of going through crossover to produce new offspring, and then the offspring have a `mutation_rate` probability of being mutated and then these offspring are added to the new population. It then gathers some stats on the old population e.g., best solution and average distance of that generation – this allows for analyzing how the algorithm performs over generations. Then the old population is replaced with this new population and this repeats for `generations` generations

3 Experimental Results and Analysis

3.1 Testing The GA

To test my genetic algorithm's accuracy, I created a TSP file with 5 cities, and one with 11. I then created a brute force script to find the best possible solution for each file to validate whether my GA could find the optimal solution or not.

3.1.1 5 Cities

5 cities means there are 5! (120) possibilities of routes and with **brute force** it found

- **Optimal Route:** (3, 2, 1, 5, 4)
- **shortest:** 2314.5526975124203m
- **Time:** 0.00006s

And the **GA** only using 10 generations and a population size of 10, found the correct shortest distance in 0.0016s

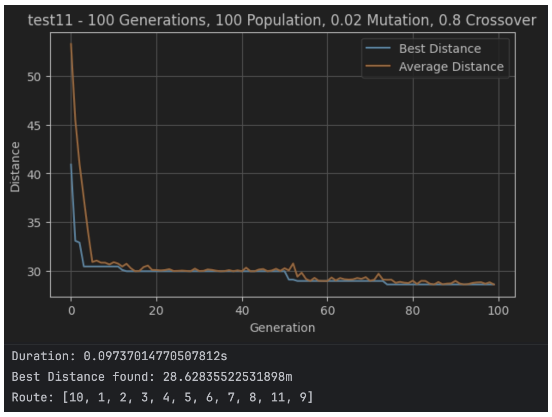
3.1.2 11 Cities

11 Cities now means there are 11! (39,916,800) possibilities and with **brute force** it found

- **Optimal Route:** (1, 10, 9, 11, 8, 7, 6, 5, 4, 3, 2)
- **Optimal Distance:** 4038.437912795235m

- **Time:** 31.58s (**very long**)

And the **GA** found this solution in 0.176s using 100 generations and a population size of 100



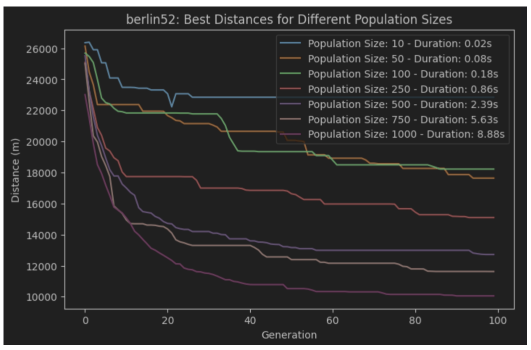
```
Duration: 0.09737014770507812s
Best Distance found: 28.62835522531898m
Route: [10, 1, 2, 3, 4, 5, 6, 7, 8, 11, 9]
```

Figure 7: Illustrating the GA running test11.tsp

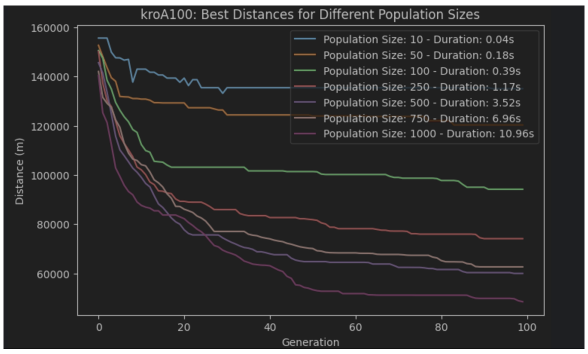
Seeing these results gave me strong evidence to believe that me genetic algorithm was working correctly and was logically sound

3.2 Evaluating the Affects of Different Parameter Values

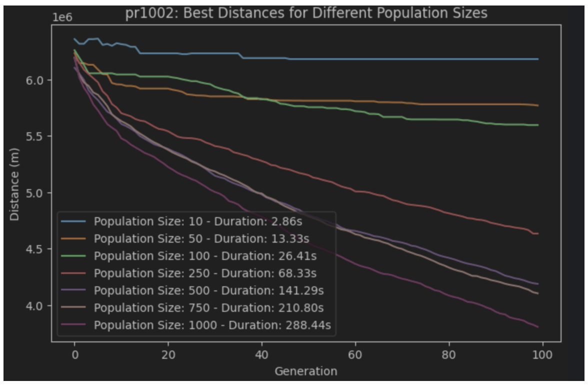
3.2.1 Varying Population Sizes



(a) berlin52



(b) kroA100

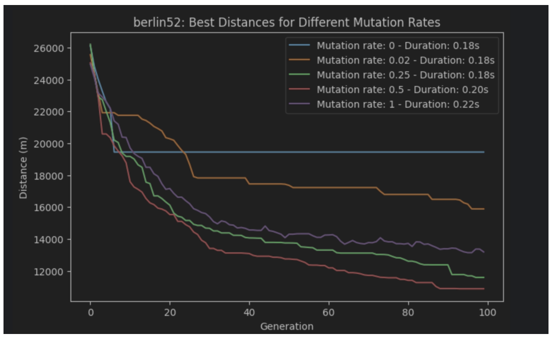


(c) pr1002

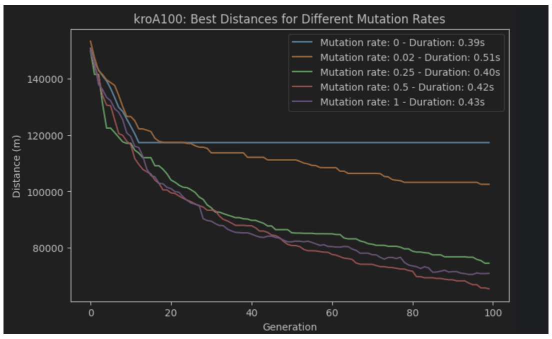
Figure 8: Illustrating the effects of varying the population size on the different tsp files

As illustrated in the graphs, the larger populations yield better solutions but require more computational time and the smaller populations require less computational time and converge faster but on suboptimal solutions. As well the size of the dataset impacts the scalability, the computational cost for high populations and large datasets increases drastically

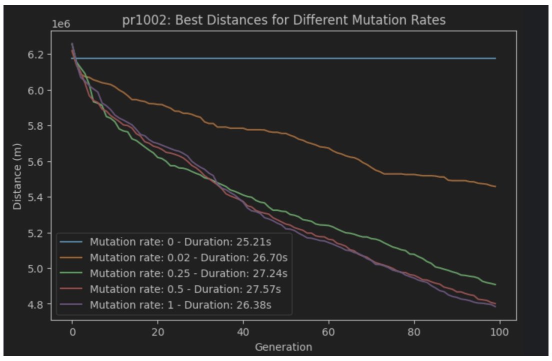
3.2.2 Varying Mutation Rates



(a) berlin52



(b) kroA100

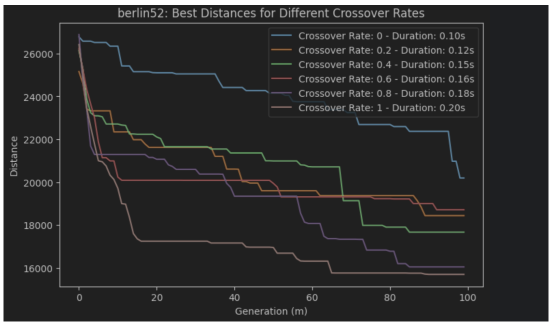


(c) pr1002

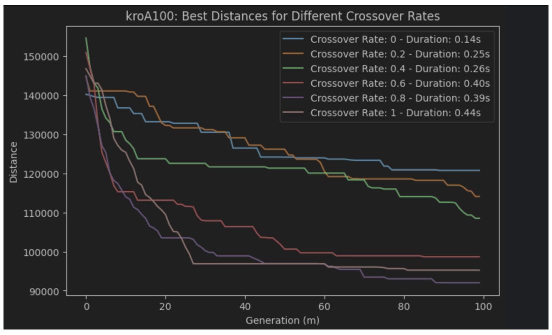
Figure 9: Illustrating the effects of varying the mutation rate on the different tsp files

As illustrated in the graphs, when the mutation rate is 0, the solutions converge too early, likely because no randomness is introduced to the sample space. Across the three TSP datasets, the mutation rate of 0.5 appears to perform the best, followed closely by 0.25 and then 1.0. Mutation rates of 0.5 and 0.25 make sense as these introduce occasional randomness to maintain diversity in the sample space. Mutation rate 1.0 took me by surprise, as this means every offspring is subjected to random changes. However this could be because my chosen mutation operators do not completely change the genome as they still obey the constraints of tsp

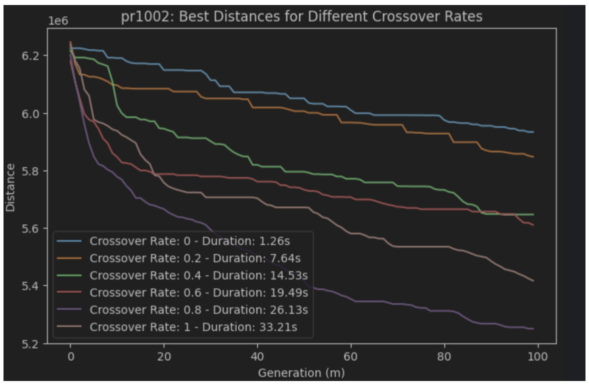
3.2.3 Varying Crossover Rates



(a) Berlin52



(b) KroA100



(c) pr1002

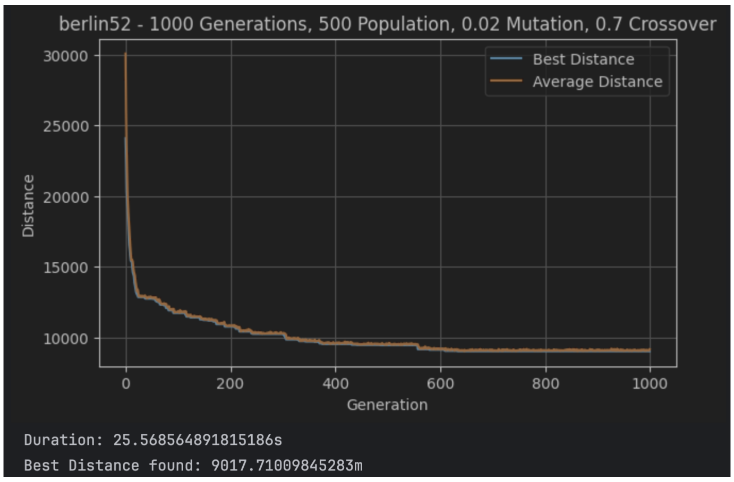
Figure 11: Illustrating the effects of varying the crossover rate on the tsp files

As illustrated in the graphs, a crossover rate of 0 converges early on suboptimal solutions. This is because, without crossover, the algorithm relies solely on mutation (at a rate of 0.02) to introduce new genetic material, which limits its ability to effectively explore the solution space. Across the three datasets, a crossover rate of 0.8 appears to perform the best, especially in pr1002, which has 1002 cities. This is likely because a crossover rate of 0.8 strikes a balance between introducing new solutions through offspring while preserving some solutions from previous generations, rather than replacing the entire population with completely new solutions.

4 Performance comparison with known optimal solutions

4.1 Berlin52

The known optimal solution for Berlin52 is a distance of 7542m [5]. On the first run, my GA record 9017m (a difference of $\sim$ 20% from the optimal solution) which I found to be reasonably efficient. Then due to the random nature of the GA, I decided to run it 10 times to see its results on the different iterations would be. The resulting distances ranged from 8971m to 10,904m, which I also considered to be reasonably efficient given the complexity of the problem

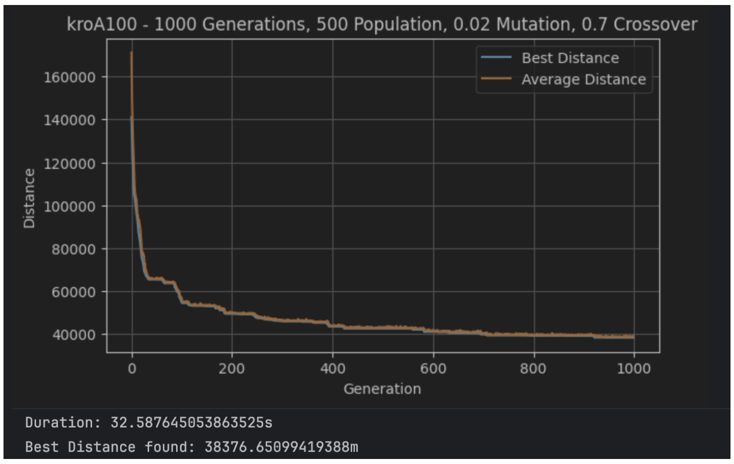


```
Duration: 25.568564891815186s
Best Distance found: 9017.71009845283m
```

Figure 12: Illustrating the GA's performance on Berlin52.

4.2 kroA100

The known optimal solution for kroA100 is a distance of 21282m. On first run, my GA recorded 38376m (a difference of $\sim$ 80% from the optimal solution) [5] indicating that as the problem gets increasingly harder, the GA gets increasingly inefficient. Again I ran the problem 10 times and the resulting distances ranged from 37569m to 46106m (now the range is increasing too)

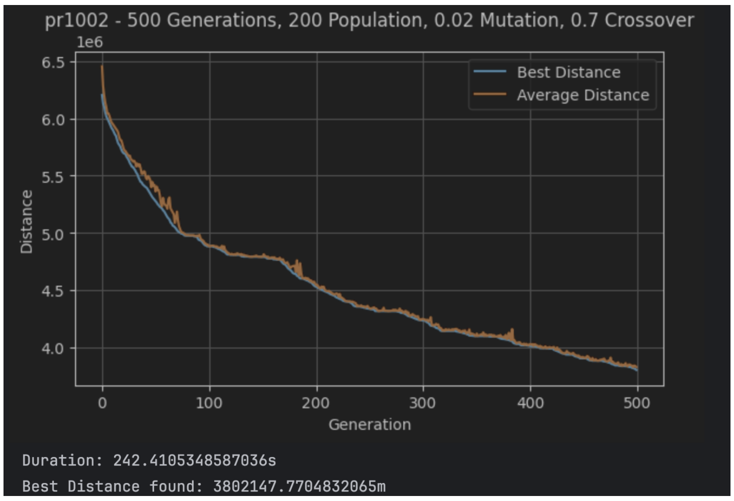


```
Duration: 32.587645053863525s
Best Distance found: 38376.65099419388m
```

Figure 13: Illustrating the GA's performance on kroA100.

4.3 pr1002

I then ran pr1002 with less generations and a smaller population time to berlin52 or kroA100 because of its required computational costs. The known optimal solution for pr1002 is a distance of 259,045m. On first run, my GA recorded 3,802,147m (a difference of $\sim$ 1400% from the optimal solution)[5]. Given stronger indication that has the problem gets more complex the GA becomes less efficient. While this result is far from optimal, it is important to note the incredible complexity of the pr1002 problem. It involves 1002 cities, resulting in a solution space of 1002! possible routes, a number with 2,574 digits [6]. Given this, it is understandable that the GA struggled to approach the optimal solution. Also, it does not appear that the GA reached convergence, suggesting that further iterations might improve the solution. This would come at a significant computational cost and would still almost certainly fall short of the optimal distance. Again I ran the solution 10 times resulting in distances ranging from 3,688,945m to 3,976,688m



```
Duration: 242.4105348587036s
Best Distance found: 3802147.7704832065m
```

Figure 14: Illustrating the GA's performance on pr1002.

5 potential improvements

5.1 Time Complexity

Calculating the distance of each genome is the most time consuming process in the program. To address this I did add a distance matrix to greatly reduce the workload, but other additional improvements could include:

1. **Memoization** By storing the calculated distances of genomes in a cache, the program could retrieve precomputed distances for genomes that reappear, avoiding redundant calculations.

2. **Concurrency** Instead of calculating the distances of genomes sequentially, this process could be parallelized to calculate the distances of genomes in the population simultaneously for faster execution.

5.2 Solution Efficiency

To enhance the quality of solutions produced by the algorithm, I could have implemented **elitism**. This approach ensures that the top subset of the population, representing the best solutions in each generation, is preserved and guaranteed to progress to the next round without modification. This could help maintain high-quality solutions

Link to GitHub Repository

References

[1] Wikipedia Contributors (2019). *Travelling salesman problem*. [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/Travelling_salesman_problem..(accessed: 25 January 2025)

[2] Colm O'Riordan. (2024). *Learning and IR_p1*. [online] Available at: https://universityofgalway.instructure.com/courses/31932/files/1988333?module_item_id=631165. (accessed: 25 January 2025)

[3] Hussain et all (2017). Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator. *Computational Intelligence and Neuroscience*, [online] available at https://onlinelibrary.wiley.com/doi/10.1155/2017/7430125. (accessed: 26 January 2025)

[4] GeeksforGeeks. (2018). *Tournament Selection (GA)*. [online] https://www.geeksforgeeks.org/tournament-selection-ga/ (accessed: 26 January 2025)

[5] comopt.ifi.uni-heidelberg.de. (n.d.). *Symmetric TSPs*. [online] Available at: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/STSP.html (accessed: 29 January 2025)

[6] Zeptomath.com. (2023). *1,002! - Factorial of 1002*. [online] Available at: https://zeptomath.com/calculators/factorial.php?number=1002 (Accessed 31 Jan. 2025).