CT421 Artificial Intelligence - Project 1

Evan Murphy | Student ID: 21306306
Github Repository: https://github.com/xEvanM/ct421-ai-ga-tsp

1. Implementation details and design choices

1.1 Introduction & Outline

A genetic algorithm is an algorithm based on the ideas of natural selection and genetics. They are commonly used to create solutions for problems in the areas of search and optimisations. The process of natural selection is implemented in the algorithm by adapting initial solutions for a number of generations. Each solution is considered to be an individual and the goal is to have the best or fittest solutions 'mate' to create even better child solutions.[1]

For this project, I will be applying a genetic algorithm to the travelling salesman problem. This is an NP-hard problem that asks "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?"[2]

I initially wrote my genetic algorithm and other relevant components as a jupyter notebook (.ipynb file) for the experiment, however I found this to be ineffective as computational times were long, and it was more difficult to follow a logical flow.

I decided to rewrite my program as a python script (.py file) with various improvements allowing me to make my workflow more efficient.

This file consists of utility functions of writing to files, data processing functions that set up the data and build a distance matrix, and components of the genetic algorithm such as mutation functions, crossover functions, fitness calculation, and the algorithm implementation itself.

1.2 Initialising data, building a distance matrix, and creating a fitness function

A function 'prepare_tsp_files' was used to create a dictionary of x,y coordinates for each TSP. A distance matrix was then computed to simply calculations.

The 'calculate_fitness' function computes the total distance of a given route in the Traveling Salesman Problem (TSP) using the aforementioned distance matrix, summing the distances between consecutive cities in the route and adds the distance from the last city back to the first, ensuring a complete tour.

1.3 Crossover Functions

The crossover functions that I used are ordered crossover (OX1) and partially mapped crossover (PMX).

Ordered crossover[3]: This method selects a random segment from one parent and preserves its order in the child, while filling the remaining positions with genes from the second parent in the order they appear, avoiding duplicates. This results in offspring made up of a meaningful sequence of cities from both parents while maintaining feasibility.

Partially mapped crossover[4]: PMX swaps a random segment between parents and creates a mapping of swapped values to maintain valid city assignments.Missing positions are then filled following this mapping, ensuring the child maintains valid city sequences as well as the positional relationships from both parents.

1.4 Mutation Functions

The mutation functions that I used are swap mutation and scramble mutation.

Swap mutation[5]: This method simply selects two random cities in the route and swaps their positions. It's an effective way to introduce variation while preserving the majority of the route's structure.

Scramble mutation[5]: A random segment of the route is selected, and the cities within this segment are shuffled at random. This results in a significantly changed portion of the route but keeps all cities included, making it useful for escaping local optima.

1.5 Genetic Algorithm

Simply put, the idea of a genetic algorithm is to create an algorithm that evolves. There are some key features of my genetic algorithm I wish to highlight in this report.

The parameters of the algorithm are:

- Distance_matrix: a matrix storing distances between cities, which we use to calculate route fitness
- Population_size: the number of routes (solutions) that are being used
- Generations: the maximum number of algorithm iterations
- Mutation_rate: the probability that a mutation occurs in an offspring's route
- Crossover_rate: the chance that crossover occurs between two parent routes
- Crossover: the function used to combine two parent routes, generating an offspring
- Mutation: the function used to introduce changes in the offspring routes
- Tournament_size: the number of routes selected for a tournament-based parent selection, impacting 'selection pressure'

Other key things to highlight:

- Stagnant_generations: the algorithm tracks generations without improvement, and terminates early if we don't find a better solution within a certain threshold (10% of total generations)
- Printing if generation % 10 == 0: an effective way to see the algorithm processing without spamming with console logs.

2. Experimental results and analysis

2.1 Initial Tests - Jupyter Notebook

My initial tests, ran with my 'old' code in the jupyter notebook were ineffective. The following parameter grid was used for berlin25 and kroa100:

crossover_methods = [ordered_crossover, pmx_crossover]
mutation_methods = [swap_mutation, scramble_mutation]
population_sizes = [100, 200, 500]
num_generations = [100, 200, 500]
mutation_rates = [0.01, 0.05, 0.1]

This resulted in a fitness score of 8460.44 for berlin25, and a score of 33317.68 for kroa100.

In both cases, a population size of 500 with 500 generations was found to be the optimal, which is unsurprising as this number is relatively low.

A reduced parameter grid was used for the larger TSP pr1002, which resulted in a score of 4908260.77. This problem preferred a mutation rate of 0.05 whilst the first two problems solved found the best fitness with a mutation rate of 0.01.

For the purpose of keeping this report concise and focusing on meaningful results, the remainder of the report will be focused on results from my python script, starting from section 2.2 below. You can explore the Jupyter Notebook in the Github repository.

2.2 Further Testing - Python Script

As mentioned previously in the report, I rewrote my genetic algorithm and the associated components as a python script rather than a jupyter notebook. This included improvements such as additional parameters, and corrections for aspects that were overlooked initially.

2.2.1 Grid Search Process (Running 'Experiments')

Many different permutations of my parameters were used in the experimenting process which can be found in the source code, but for the purpose of keeping this report relatively condensed I am focusing on the parameter grid seen below that consists of a large 810 permutations. This parameter grid was tested on the berlin52 TSP.

```
crossover_methods = [ordered_crossover]
mutation_methods = [swap_mutation]
population_sizes = [100, 200, 500, 1000, 2000, 5000]
num_generations = [1000, 2000, 5000]
mutation_rates = [0.01, 0.05, 0.1]
crossover_rates = [0.1, 0.5, 1.0]
tournament_size = [2, 3, 5, 8, 10]
```

Both *pmx_crossover* and *scramble_mutation* were excluded from this testing as they were found to be ineffective in my implementation. This was in order to reduce the number of permutations required. The same goes for lower numbers of generations and smaller population sizes which were tested in previous implementations. A lower number of generations would also be captured thanks to the stopping condition implemented for stagnant generations.

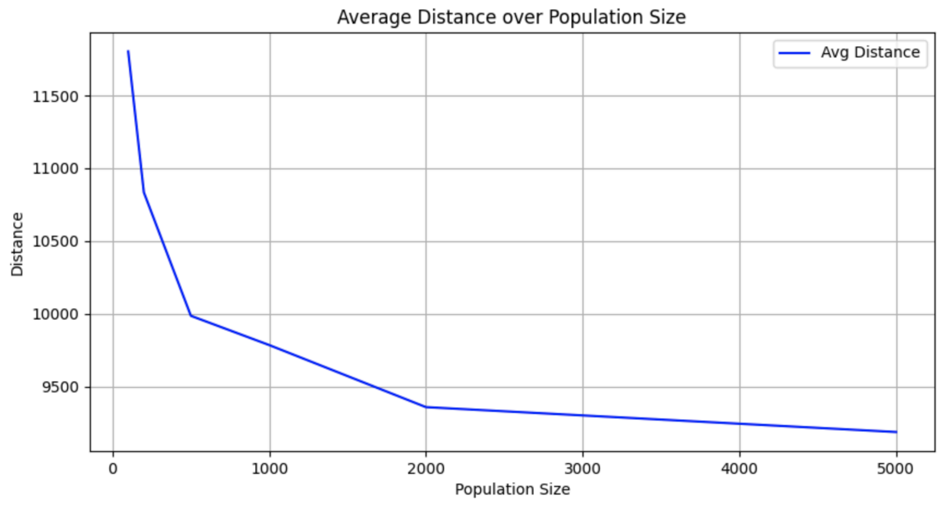
The grid search process took 7 hours, 16 minutes and 25 seconds, and resulted in a best fitness of 7765.64. This was achieved with a crossover rate of 1.0 using ordered crossover, a mutation rate of 0.01 using swap_mutation, a population size of 2000, which reached stopping criteria after 176 generations. The tournament size in this case was 5, and execution time was 17.02 seconds.

The optimal result found for berlin52 was then used on the other datasets tested, as this was considered to be the 'optimal' genetic algorithm for the purpose of solving the travelling salesman problem. This yielded results of 26821 for kroA100 and 2703139.08 for pr1002. I would have liked to perform grid search on these TSPs also however this was computationally expensive with a high number of permutations, especially when running with relatively large populations for a lot of generations each time.

2.2.2 Experiment: Population Size

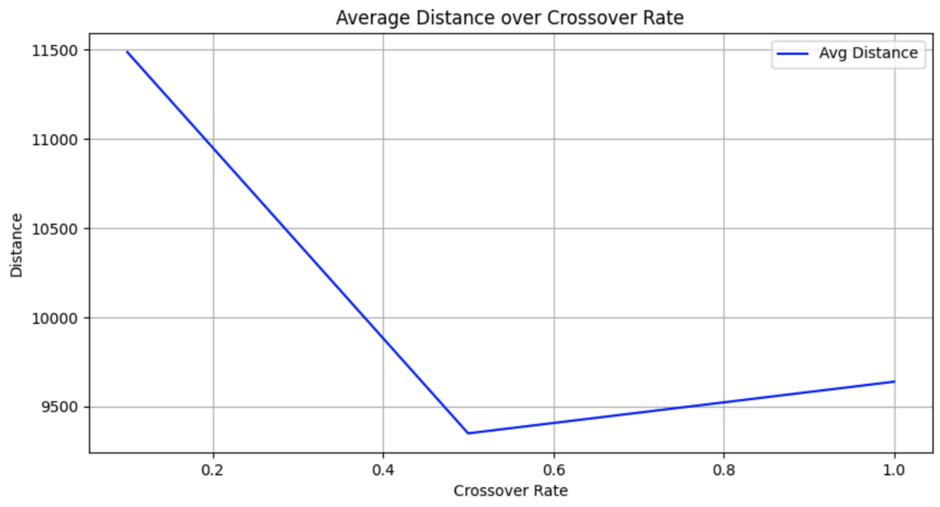
The first experiment that I conducted was with the population size. The goal of this experiment, as with the others, is to see how changing the value of some parameter impacts our ability to achieve a near-optimal solution.

The graph below shows how our the distance (fitness) changes with a greater population size. There is a strong improvement in fitness from 100 population to 200 population, as well as from 200 population to 500 population. The improvement is more gradual from 1000, to 2000, and then to 5000. The diminishing return is probably largely due to the gene pool being saturated. After 2000 population, it appears that new individuals are introducing little unique genetic material, and in fact, they are probably very similar which will reduce evolutionary pressure and impact the effectiveness of the mutation and crossover functions.



2.2.3 Experiment: Crossover Rate

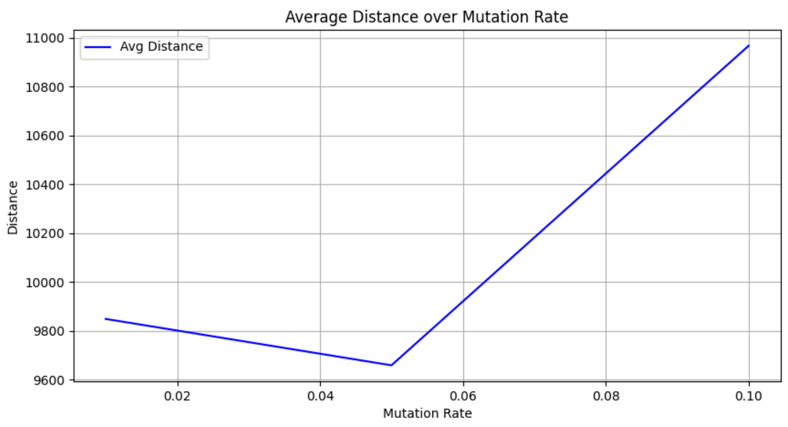
The next experiment I want to highlight is changing crossover rate, which is the rate at which crossover occurs. Crossover refers to the combining of genetic material from two parents to create an offspring, which allows us to explore the search space. The values that I used for crossover rate were 0.1, 0.5, and 1.0. This graph is a bit more interesting to look at, as we can see that the average distance is lowest with a crossover rate of 0.5, however, later in the report you will find that the best results were achieved for a crossover rate of 1.0 in the case of berlin25, for which these experiments were conducted.



At a crossover rate of 0.5, we have a good balance between exploring new solutions and exploiting good solutions, which leads to consistently lower average distances over generations. However, with a crossover rate of 1.0, every individual undergoes crossover, which increases exploitation. This can lead to stronger convergence, which will likely increase the average distance, but with enough generations provides the algorithm with a chance to find a near-optimal solution.

2.2.4 Experiment: Mutation Rate

The final experiment that I want to take about is mutation rate values. The values tested in this experiment were 0.01, 0.05, and 0.1. This is the rate at which random changes are introduces in solutions, with a goal of maintaining diversity and avoiding premature convergence.



From the above graph we can see the impact of mutation rates on fitness. A mutation rate of 0.01 maintains diversity, whilst allowing gradual improvements, and as such it leads to a good fitness score. An even better score is seen with a mutation rate of 0.05, as we prevent premature convergence whilst efficiently refining our solutions. However, increasing the mutation rate to 1.0 introduces too much randomness, which causes disruption to good solutions and leads to suboptimal paths being found, and as such, a poorer fitness score (greater distance) .

3. Comparison to Optimal Solutions

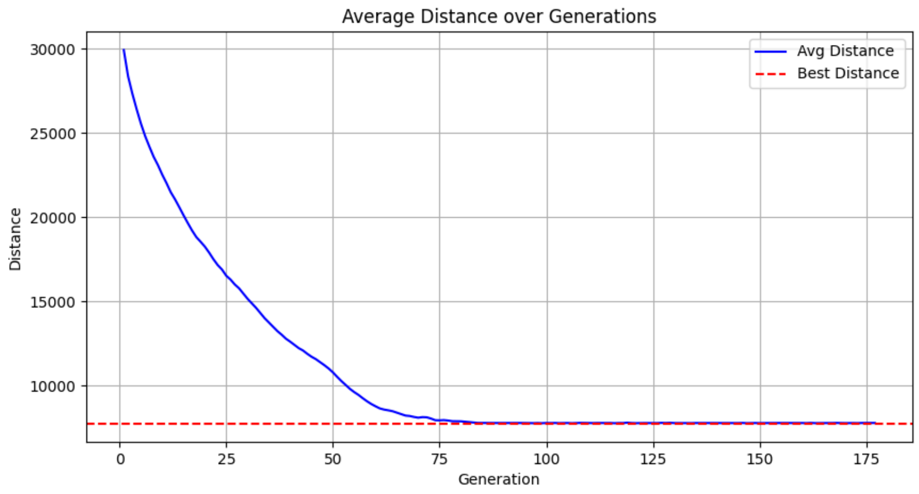
The optimal fitness scores were taken from:
http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/STSP.html

Algorithm	Optimal Fitness	My Fitness	Difference
Berlin52	7542	7765.64	2.97%
KroA100	21282	26821.06	26.03%

Pr1002	259045	2703139.08	943.50%

3.1 Berlin52

Berlin52 is the TSP that I performed the grid search operation on, and as such forms the basis for most of the data in this report. As mentioned, a score of 7765.64 was achieved from the grid search process after 176 generations.



We can see from the graph above that after around 75 generations the improvement in distance is very small, and in fact it's probably not worth running additional generations. This indicates to me that my stopping criteria should have been further refined to reduce computational cost.

The table below shows some interesting statistics from the grid search process.

Parameters	**Best Distance (7765.64)**	**Worst 'Best' Distance (19889.55)**
Population	2000	200
Generations	176	254
Crossover Function*	ordered_crossover	ordered_crossover
Crossover Rate	1.0	0.1
Mutation Function*	swap_mutation	swap_mutation
Mutation Rate	0.01	0.1
Tournament Size	5	2
Execution Time	17.02s	1.11s

An important note from the above, is that because early testing rendered my alternative crossover function and mutation function to be ineffective there was only one of each tested during the grid search process, so they are really ignored here.

My initial thoughts are that the mutation function was having a negative effect as we can see we get the best distance with a lower mutation rate, but it also shouldn't be ignored that the tournament size is greater which will impact which candidates survive selection.

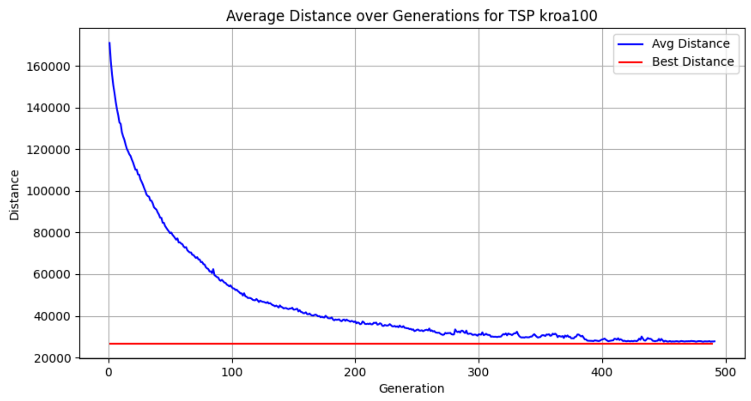
Some other statistics to consider from the grid search process include:

- **Execution time:** A mean execution time of 32.327s, and a longest execution time of 316.27s. Interestingly we can see that our best results came with an execution time of 17.02s which might indicate that more generations or a greater population is NOT necessarily better.
- **Best distance:** As mentioned, a best distance of 7765.64 was achieved, with a worst distance of 19889.55, and an average of 10.158.55. The average execution time is not exceptionally far from the optimal solution which indicates the algorithm was generally providing good solutions. I would attribute this, in part, to the high population values and large numbers of generations used for the majority of the grid search process.

3.2 KroA100

In the case of KroA100, I achieved a best fitness of 26821.06 after 490 generations. This fitness score is 26.03% greater than the known optimal distance. This had an execution time of 76.48 seconds.

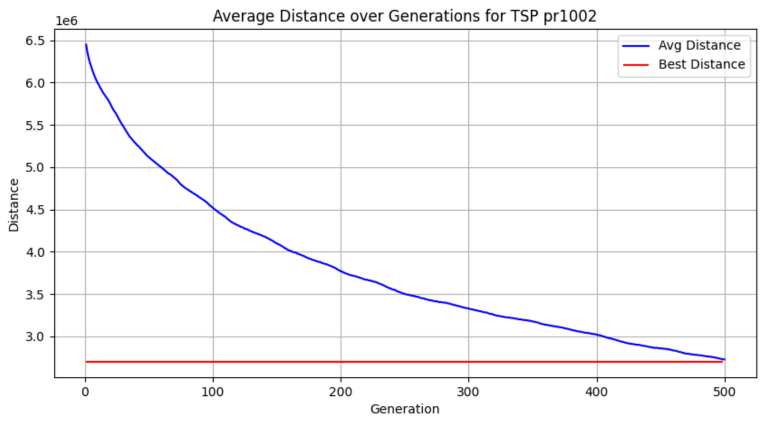
The crossover rate was 0.5 using ordered crossover, the mutation rate was 0.1 for swap mutation, with a population size of 2000 and a tournament size of 5.



In the case of this algorithm, we can observe that the number of generations has a greater impact than seen for the smaller berlin52 dataset. This is unsurprising, as the other parameters remained the same, and with each solution being comprised of more 'cities' it makes sense that a greater number of generations was required to reach our best fitness.

3.3 Pr1002

Finally, for the pr1002 dataset, I achieved a best fitness of 2703139.08 after 500 generations. This fitness score is 943.50% greater than the known optimal distance. This had an execution time of 4208.11 seconds. This fitness score is disappointing but unsurprising as each individual here is much longer than the previous two datasets explored.



In the case of the above graph, we can really see how an increase in generations leads to an improvement in average distance. Although the graph appears to level off slightly at the end, it's a reasonable assumption to state that further generations would have seen a better fitness score. However, it shouldn't be overlooked that this dataset is computationally complex and each iteration of the algorithm takes a long time to run.

4. Potential Improvements for GA

4.1 Greater grid search:
Expansion of the grid search performed to include more permutations would provide more insight into how different parameter combinations impact performance. It's possible that I neglected PMX crossover and scramble mutation by not testing them with sufficiently large population sizes or a greater number of generations, potentially missing better-performing configurations that would provide a closer to optimal solution.

4.2 More generations:
Increasing the number of generations ran to a much larger number may allow the algorithm more time to refine solutions, escaping local optima and ultimately providing a stronger solution.. This wasn't done due to large computational cost and the tradeoff between computational cost and improvement in fitness.

4.3 Larger populations:
Using larger population sizes in the GA could enhance diversity among routes and prevent convergence from happening early. A larger number of routes would provide a broader search space which could lead to to better solutions,though this would increase computational costs.

4.4 Additional datasets:
Testing the algorithm on more datasets, including larger and more complex TSPs, would improve the algorithms ability to find the best solutions. Evaluating performance across varying problem sizes would help determine how certain parameter settings generalise.

4.5 Adaptive mutation and crossover rates:
My research has indicated that adaptive rates over generations can help maintain diversity early on while refining solutions in later stages. This would allow the algorithm to balance exploration and exploitation dynamically depending on fitness values for each generation, particularly when working with 'solved' problems.

4.6 Better stopping conditions:
Improving the stopping conditions would make the GA more computationally efficient and reduce the time taken to create a solution for these problems. The current stopping conditions are based on 10% of the number of generations, however in the case of a very large number of generations where we're unlikely to see improvement after X generations, we are unnecessarily computing fitness without improvement for a long time.

4.7 Elitism:
Introducing elitism, where the best solutions are preserved across generations, could prevent regression in solution quality which is frequently seen in cases where we have a high number of generations. In tournament selection, the best solutions can be pitted against each other and we lose some good quality solutions in this process due to the nature of randomness.

5. References Appendix

1. GeeksforGeeks (2017). Genetic Algorithms. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/genetic-algorithms/.
2. Wikipedia Contributors (2019). Travelling salesman problem. [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/Travelling_salesman_problem.
3. Alseda, L., Alseda, J., & Calsina, À. (n.d.). Genetic operations in evolutionary algorithms. Universitat Autònoma de Barcelona. Available at: https://mat.uab.cat/~alseda/MasterOpt/GeneticOperations.pdf
4. Helmy, M. (2023). Baeldung. [online] Baeldung on Computer Science. Available at: https://www.baeldung.com/cs/ga-pmx-operator.
5. tutorialspoint.com (2019). Genetic Algorithms Mutation. [online] www.tutorialspoint.com. Available at: https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_mutation.htm.