CT421 Evolutionary Search

Brian Moyles – 21333461

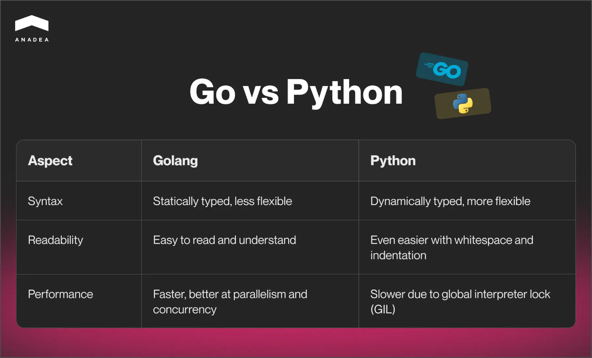
**Project Goal**

The Goal of this assignment is to “implement a genetic algorithm to solve the Traveling Salesman Problem (TSP)”, using Parameter tuning, implementing different algorithms, and analysing performance.

**Overview**

Why I Chose Golang

My primary reason for choosing to do this project in Golang was due to it being my preferred language. Taking on a task such as the TSP would be sensible to do with a language preference, rather than an unfamiliar language. Another reason I chose Golang was due to its speed and efficiency. Go is known to be an efficient language with faster performance, which will be very important when dealing with a genetic algorithm such as TSP with large data. For example, a personal test during this assignment showed me that Golang was able to perform 16 times faster than a replica python script, further justifying my language choice.



<https://www.google.com/url?sa=i&url=https%3A%2F%2Fanadea.info%2Fblog%2Fgolang-performance%2F&psig=AOvVaw0BloqnQCl5yeXb72anHGgX&ust=1739833026686000&source=images&cd=vfe&opi=89978449&ved=0CBQQjRxqFwoTCKDn2p2lyYsDFQAAAAAdAAAAABA1>

**Crossover**

*Crossover Script*

[https://github.com/Moylzee/ai-assignment-1/blob/main/crossover/crossover.*go*](https://github.com/Moylzee/ai-assignment-1/blob/main/crossover/crossover.go)

The purpose of crossover functions in the TSP is to combine solutions from two tours to create a new potential solution. “*Crossover happens whenever two individuals are recombined to create new individuals and finally they are copied into the new population*” – *( [1] ResearchGate, Sarkar, Subham. (2018). Implementation of Parallel Genetic Algorithm)*. It aims to take beneficial routes from the two parent tours and combine them into one candidate solution

Ordered Crossover (OX)

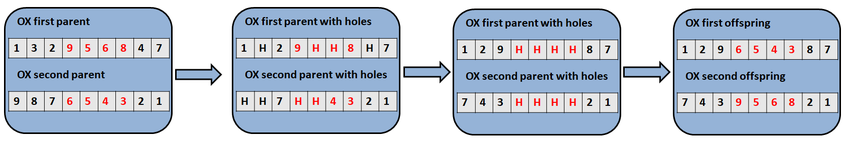
OX works by taking 2 parents, P1 and P2, and copies a segment of the tour from P1. It then fills in the remaining missing cities using P2, while keeping their order. The steps involved are:

1. Choose a segment from P1

2. Copy the segment into the child in their same positions

3. Use P2 to fill the remaining cities, keeping their sequence

4. Use only unique cities not already included

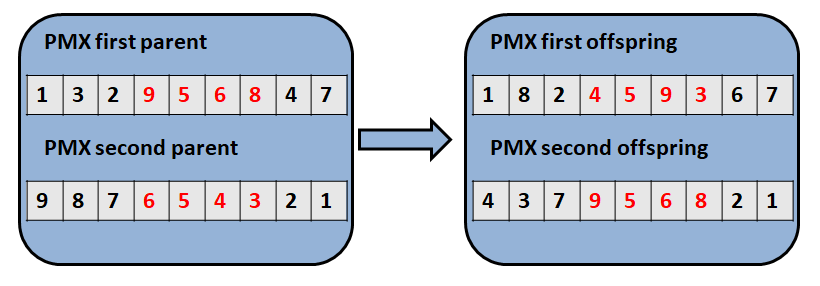


*Example of OX* [*https://www.researchgate.net/publication/357549651/figure/fig3/AS:11431281105541286@1670447568260/Example-of-the-Order-Crossover-OX-operator.ppm*](https://www.researchgate.net/publication/357549651/figure/fig3/AS:11431281105541286@1670447568260/Example-of-the-Order-Crossover-OX-operator.ppm)

Partially Matched Crossover (PMX)

PMX is used by mapping segments between parents. This is done to maintain relationships and resolves conflicts by using a mapping table. The Steps involved are:

1. Create a table that maps relationships between parents’ positions
2. Finds conflicts between cities in a parent that would violate the relationships
3. Resolve these conflicts by using the Mapping table



*Example of PMX*

[*https://www.researchgate.net/publication/357549651/figure/fig4/AS:11431281105550090@1670447568424/Example-of-the-Partially-Matched-Crossover-PMX-operator.ppm*](https://www.researchgate.net/publication/357549651/figure/fig4/AS:11431281105550090@1670447568424/Example-of-the-Partially-Matched-Crossover-PMX-operator.ppm)

**Mutation**

Mutation Functions in the TSP are used to prevent premature convergence while also allowing new solutions to be explored. This is done by using a controlled randomness to modify existing tours but while also maintaining a valid tour.

A screenshot of a computer program

AI-generated content may be incorrect.

Swap Mutation

The Swap mutation picks 2 cities at random, and swaps them in the tour. It maintains all the other connections so that there is minimal disruption. Its computational cost is O(1) and is ideal for when diversity in population is high, efficiency is important and to fine-tune proven solutions.

Inverse Mutation

The inverse mutation function takes a randomly selected segment from the tour and reverses the order of the cities in that segment. It has slightly higher disruption level, with a computational cost of O(k), where k is the length of the segment. It is best utilized when population diversity is low and to explore larger solution spaces.

Inverse Mutation and Swap Mutation are the 2 Mutation functions that I decided to use for my Project.

**Selection**

I decided to use the tournament selection algorithm for my Assignment. The tournament selection algorithm works by taking a segment of the population randomly, comparing the fitness scores between the chosen segment. The highest fitness will be chosen and will continue. This process is repeated until sufficient individuals are selected.

Tournament Selection was chosen as it is a more straightforward selection algorithm. I also conducted experimentation with a roulette wheel algorithm and noticed no significant improvement on the results. Hence, due to its robustness, adaptability, straightforwardness, and other reasons such as personal preference, I decided on using the Tournament Selection Algorithm.

**Utilities**

The utilities file is used to store functions used in the main file that are not considered essential to the TSP. These include functions that read/save data from files, functions that check if a value is contained inside a map, and calculation functions. This utility script ensures a structured and clean codebase. Allowing for separation of functions is a practice that I try to implement wherever possible.

**Files Used**

The files used for this assignment were the ones suggested in the assignment brief:

Berlin52

The Berlin52 file is the smallest of the files used, which includes 52 cities. I used this file as a foundation to determine if my genetic algorithm and functions worked. Once I started receiving scores using the Berlin52 file, I began to optimize my code. This file has a known optimal score of around 7544.

KroA100

This is the medium sized file I will be using for the assignment, with a known optimal score of around 21282

Pr1002

The largest file to be used in this assignment with a known optimal score of around 259045.

**Variables for files**

The Variables used for this assignment were stored in a ‘vars.go’ file. For each file used (berlin, kroA, pr) they had their own set of variables that would be loaded in from the file before the genetic algorithm started. This was done because, although ideally the variables would remain consistent between files, there were instances where they had to be changed to improve performance due to the size difference of the files. For example, the pr1002 needed more generations than berlin52.

Variables Used

The Variables used consisted of the following:

* PopulationSize
  + The number of individuals in the population
* Generations
  + The number of generations (instances) the algorithm will run
* TournamentSize
  + The number of individuals in each tournament selection
* CrossoverRate
  + The chance at which crossover operations are used in the genetic algorithm
* MutationRate
  + The chance at which mutation operations are used in the genetic algorithm
* ElitismCount
  + The number of top individuals that are brought to the next generation
* CrossChance
  + Variable used to set the probability of OX to PMX being used

Storing these variables in a separate file and using separate structs for each TSP file gave me complete control over the manipulation of these values for each runtime. Allowing smoother testing, tuning and better performance.

var BerlinVariables = vars{

PopulationSize: 1000,

Generations: 2500,

TournamentSize: 5,

CrossoverRate: 95,

MutationRate: 5,

ElitismCount: 2,

CrossChance: 100,

}

var krVariables = vars{

PopulationSize: 1000,

Generations: 2500,

TournamentSize: 5,

CrossoverRate: 95,

MutationRate: 10,

ElitismCount: 2,

CrossChance: 0.95,

}

var prVariables = vars{

PopulationSize: 1000,

Generations: 100000,

TournamentSize: 3,

CrossoverRate: 95,

MutationRate: 10,

ElitismCount: 2,

CrossChance: 100,

}

The population size remained consistent throughout the assignment, as it didn’t seem to have a major effect on the results. Generations was increased for pr1002 as, due to the overall complexity of the file, it required more generations to get to its most optimal point. Tournament size was also reduced for it. The crossover rate, mutation rate and elitism count also remained consistent throughout the assignment. The main variable with change was the cross chance. This was due to the performance of OX compared to PMX. OX is better used for berlin and kroA, as it performs significantly better with smaller datasets , while PMX is used with pr1002 as it performs better with more complex data.

**Main File**

**Plot File**

**Experimentation & Testing**

Experimentation with this assignment involved adjusting the parameters to attempt to get results closest to the known optimal solution. For example, Berlin52 was used with multiple different parameter combinations in order to find the best parameters.

Experimentation Results (Berlin52)

A screenshot of a computer

AI-generated content may be incorrect.

A group of blue bars

AI-generated content may be incorrect.

The experimentation with parameters, as shown by the results above, demonstrates that the best result achieved was 7544.37 and the worst result was 12206.82. Further detailed analysis reveals that increasing the population size leads to improved solution quality, though at the cost of computation time. Specifically, the smallest population tested (50) produced a distance of 8796.58, while the largest population (1000) achieved the optimal result of 7544.37, demonstrating that larger populations consistently yield better solutions.

Regarding crossover operators, the experiments show distinct performance characteristics. When CrossChance is set to 100 (always using OX), the mean distance falls below 8k, confirming OX's superior optimization capabilities. In contrast, PMX, while offering faster execution, produces less optimal results.

Additional experimentation conducted on the Berlin52 file provided valuable insights for optimizing the implementation. Testing with this smaller dataset enabled precise parameter calibration before scaling to larger instances. Notably, when applying these optimized parameters to the larger pr1002 instance, the results revealed an interesting dynamic: while OX excelled on the smaller Berlin52 dataset, PMX proved more effective for the larger-scale pr1002 problem, highlighting the importance of operator selection based on problem size.

OX v PMX performance on pr1002

**PMX**: Generation 500: Best Distance = 4151191.663684

**OX**: Generation 500: Best Distance = 4427440.719194

The above results show a 300k improvement from PMX compared to OX for pr1002, and, although this varied every runtime, the result of PMX performing better than OX remained consistent.

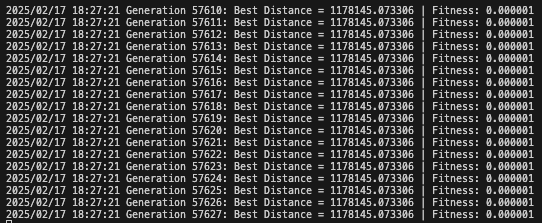
Population Performance on pr1002

Below also shows the importance of parameters on the files. The example shows two runtimes of the pr1002 file, with different population values. Although they were started at the exact same time, it is seen that figure 1 has only managed to reach 15,000 Generations performing much slower than figure 2, with 57,000 generations. This shows the difference in performance speed a parameter can have, and why testing different parameters is vital, especially when it is observed that, although it has managed 40,000 less generations, it has achieved a better score for distance, thus showing how essential it is to experiment with different parameters and functions, in order to find the perfect balance between performance speed and effectiveness to achieve optimal results.

**A screen shot of a computer

AI-generated content may be incorrect.**

*Population: 200 | Generation: 1530 | Distance: 979634*

****

*Population: 50 | Generation: 57620 | Distance: 1178145*

Experimentation was also vital for other aspects besides parameter tuning, as it also helped me optimize my code. Throughout the testing process, it became evident that my code was not performing the best that it could.

Using a bigO calculator online ( <https://www.bigocalc.com/> ), I tested my functions and attempted to improve their time complexity. This was necessary testing as I found issues in my code that was making the code run longer than it should, an example can be seen below, where my crossover function was improved, significantly increasing performance speed.

A computer screen shot of a program code

AI-generated content may be incorrect.A screen shot of a computer code

AI-generated content may be incorrect.

Big O(n^2) compared to Big O(n)

**Discussion of Results**

(Note: The Graphs are also available in the results.md file on Github)

**Berlin52**

|  |  |  |  |
| --- | --- | --- | --- |
| **Best Distance** | **Best Fitness** | **Parameters** | **Time Taken** |
| 7544.37 | 0.000133 | **PopulationSize**: 1000, **Generations**: 2500, **TournamentSize**: 5 **CrossoverRate**: 100, **MutationRate**: 5, **ElitismCount**: 2, **CrossChance**: 0.95 | **22.243628096s** |

Discussion Compared to known optimal solution

The optimal solution was achieved using the parameters specified above. The use of OX provided precise tuning and refinement. OX, alongside the low mutation rate and large population size, allowed for a balance between exploiting and exploring.

Overall, The algorithm reached the known optimal solution, justifying my parameter choices and reinforcing the abilities of my code. Below are graphs of the results for Berlin52.

**A graph with blue dots

AI-generated content may be incorrect.A graph with a line going up

AI-generated content may be incorrect.**

**A graph showing the growth of a number of people

AI-generated content may be incorrect.**

**KroA100**

|  |  |  |  |
| --- | --- | --- | --- |
| **Best Distance** | **Best Fitness** | **Parameters** | **Time Taken** |
| 22239 | 0.000045 | **PopulationSize**: 1000, **Generations**: 2500, **TournamentSize**: 5  **CrossoverRate**: 95, **MutationRate**: 10, **ElitismCount**: 2, **CrossChance**: 0.95 | **47.035707192s** |

Discussion Compared to known optimal solution (21,282)

(known Optimal: <https://www.sciencedirect.com/science/article/pii/S266672072400136X>)

My Solution achieved a Score of 22,239, which is 957 away from the known optimal solution. This represents a deviation of around 4.4%. This was accomplished using a balanced configuration of a population size of 1000 and a small mutation rate of 10% and a strong elitism preservation of 2.

A score 957 away from optimal is good considering the scope of the problem, although a fitness of 0.000045 suggests a strong presence of convergence. Overall, this algorithm, aligned with the parameters used, shows very strong performance, achieving near optimal results.

**A graph showing a blue line

AI-generated content may be incorrect.A graph with blue dots and lines

AI-generated content may be incorrect.**

**A graph with a line going up

AI-generated content may be incorrect.**

**Pr1002**

|  |  |  |
| --- | --- | --- |
| **Best Distance** | **Best Fitness** | **Parameters** |
| 7544.37 | 0.000133 | **PopulationSize**: 1000, **Generations**: 2500, **TournamentSize**: 5  **CrossoverRate**: 100, **MutationRate**: 5, **ElitismCount**: 2, **CrossChance**: 0.95 |

Discussion Compared to known optimal solution

**A graph with a line going up

AI-generated content may be incorrect.**

**A map of a group of dots and lines

AI-generated content may be incorrect.**

**A blue line on a gray background

AI-generated content may be incorrect.**

(Note: This Graph was massive and can be run locally if needed to be visualized better)

**Conclusion & Remarks**

**Discussion of potential improvements**

Potential Improvements for my code would include ensuring the time complexity of functions are optimal from the beginning, rather than allowing for functions to become over-complicated early. As Golang is an extremely efficient language, I am satisfied with my choice of language and would not change that. I would instead focus on implementing different functions such as more mutation and crossover functions, in an attempt to find more optimal results.  
I would also implement a caching mechanism to help speed up the process. This could be used to cache the distances between nodes, or for any other use that would provide fast lookup times and help improve the overall performance.  
I would also automatically run my python scripts to analyse and provide graphs of the results, rather than having to run them myself manually after each runtime.

Overall, my potential improvements would aim to improve overall efficiency, leveraging performance times with potential to expand upon the codebase with more effective functions.

**References and Links**

* [1] <https://www.researchgate.net/figure/Steps-of-TSP-iv-Crossover-The-probability-of-crossover-is-calculated-using-the-fitness_fig1_340579712>