SEM6120 Assignment 2

Solving Travelling Salesman Problems using Genetic Algorithms

Daniel McGuckin

[dam44@aber.ac.uk](mailto:dam44@aber.ac.uk)

110059989

# Introduction

The Traveling Salesman Problem (TSP) is about finding the shortest route for a salesman when touring a group of cities. The salesman may only ever travel through each city once. TSP is a NP-Complete problem, there is no efficient way to calculate an optimal or near optimal solution and as the number of cities increases the search space rises exponentially, and as a result, the time taken to find a good solution.

Genetic Algorithms and similar methods such as Ant Colony Optimisation are often used to generate optimal or near optimal solutions for the problem. When using Genetic Algorithms for the TSP problem the Chromosome will be a combination of all the cities, an individual city will be the gene. Each gene may appear only once in the route as otherwise this solution will be invalid.

Chromosomes will be continually modified and recombined in attempts to find better solutions for the problem.

# System Design

I chose to write the application in C#.NET as this is the language I feel most comfortable in. In my opinion it does everything Java does as well or better and in a cleaner and easier to use IDE (Visual Studio).

When designing the system I wanted to make it easily modifiable and to be able to be used like an API with the potential to turn it into one in the future.

## .NET Projects Overview

I used six separate projects for the algorithm:

* TSPCityGenAPI

This generated a user-defined amount of cities and output them to a JSON file to be read by the genetic algorithm.

* TSPCityGenGUI

A user interface which used the TSPCityGenAPI application, a user can enter a number of cities and the destination they want the file output to. The application will then feed these into the API. Figure 2 depicts the GUI application.

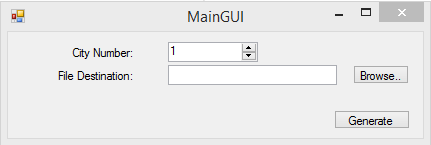


Figure 1

* GeneticAPI

This was the main project, it contained all the code concerning the Genetic Algorithm. Figure 3 is based on the design for this project.

* TSPGenGUI

A GUI which allows users to customise their run of the Genetic Algorithm. As it runs it displays a graph detailing how the algorithm is going. At the end of a run it writes the graph to a file and also outputs details about the run. Multiple runs can be queued which allowed for automated taking of results. Figure 3 depicts this GUI application.

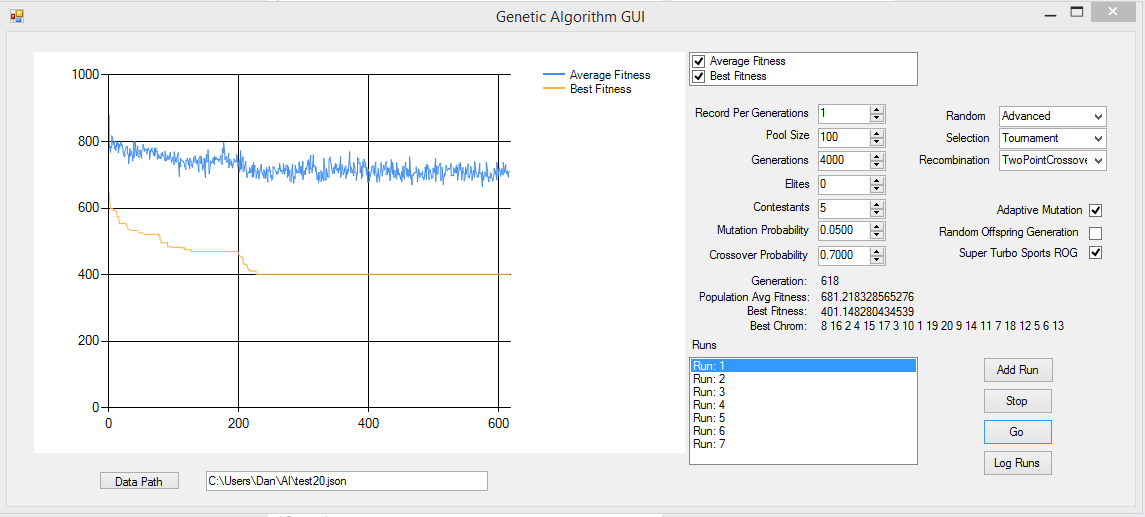


Figure 2

* TSPGenSandbox

A basic console app which was used to test the Genetic Algorithm before the later development of the GUI.

* TSPModel

The Genetic Algorithm used a generic data type so that it could be more customisable in the future. The TSPModel contained the concrete type (City) which was used for the TSP assignment.

## Genetic Algorithm Overview

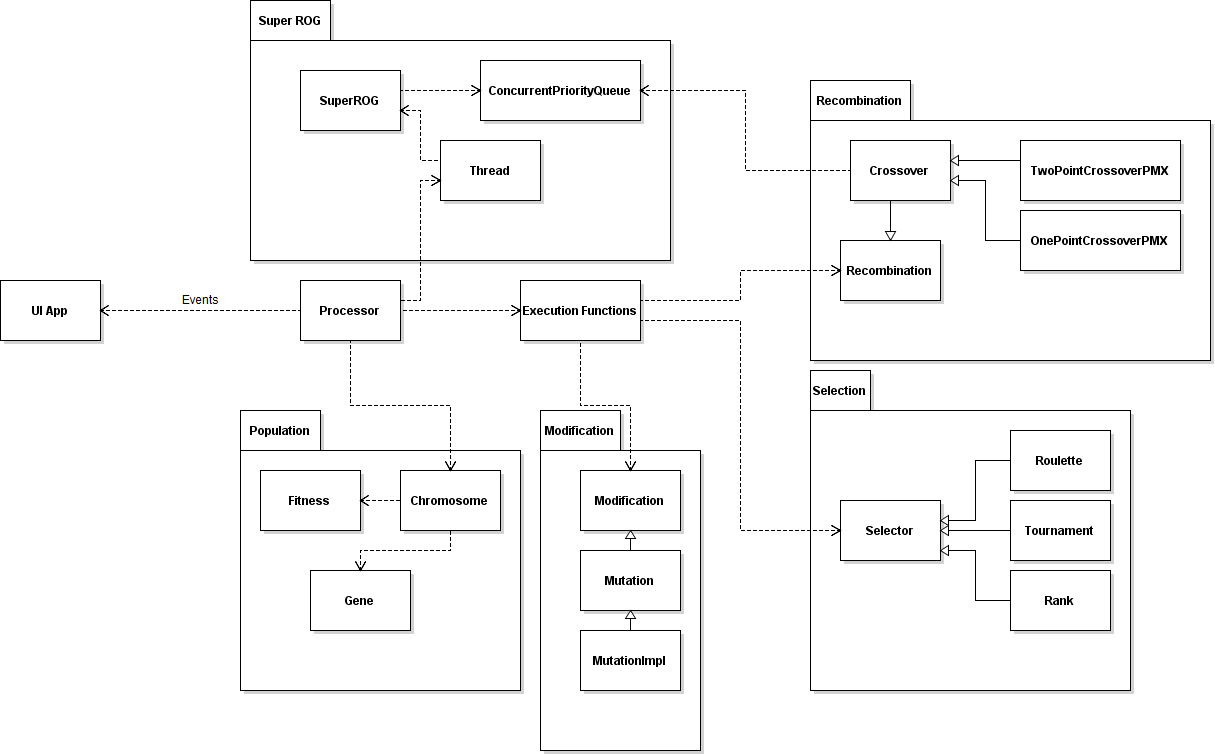


Figure 3

Figure 1 depicts the important objects that make up the Genetic Algorithm.

The processor class runs the algorithm. It initializes the population and then Selects, Recombines, Modifies and Evaluates the population in a loop by calling methods in the ‘Execution Functions’ class which is used to interact with the different parts of the system.

Chromosomes evaluate their own fitness by calling methods in the Fitness class whenever they are created or modified. My future vision for the system is to make it a proper API and outsource implementation specific to a project, i.e. how the fitness is evaluated, how the chromosomes can be modified and recombined to the developer.

This is why I used a generic type for the data and also used inheritance extensively. For instance I use a Recombination abstract-class, the Crossover abstract-class inherits from this and then the specific implementations of Crossover inherit from that. Crossover takes the entire population and then feeds two chromosomes at a time to the implementation which returns two new chromosomes. The Recombination class is largely irrelevant to the application, but during the design I was considering the future and potentially a developer could want to use the Genetic Algorithm but recombine in a different way to crossover, they could therefore inherit from Recombination and write their own method. I used a similar plan when implementing Modification.

To completely decouple the Genetic Algorithm from any user interface I used events. The UI can subscribe to the Genetic Algorithm, the GA will then send information about the run to the UI application such as the current best chromosome, the average fitness and the current iteration. This came in use as I used two UI’s throughout the project: The console application and then the GUI later on.

To represent a candidate solution I used the Chromosome class. This class stored its fitness and a list of Gene objects. Gene objects held the ‘data’ which was a generic type, specified as a ‘City’ by the UI. The generic type implemented the IData interface which specified that the methods x, y and id should be implemented to find the location of the data and to be able to reference it as a unique Gene. Ultimately the representation boiled down to a list of City objects, each containing an x and y co-ordinate to detail where they were and an Id. I felt that this simple representation was perfectly suitable and gave the added bonus of making Chromosomes easily human readable (more so than, for instance, a binary representation) which was helpful when debugging, i.e. checking that crossovers were creating valid candidates. I was concerned that this representation would be heavy on the system and cause the algorithm to run slowly, however, after implementing and running it I found that it ran at a perfectly acceptable speed.

For my Selection operator I implemented Rank, Roulette and Tournament methods. I initially started implementing Roulette. I got stuck part way through implementing it when I realized that the lowest fitness candidates were the best and therefore they had to be weighted highest. I changed to Tournament and implemented that, I made the Tournament contestant number variable as I was unsure what the optimal contestant number would be.

Later I came back to Roulette and realized that [PAPERS] to get the weightings correct, the weighting had to be inversely proportionate to the fitness.

I was initially working out an individual’s percentage weighting by doing:

return (100 / totalfitness) \* (individual.fitness))

But changed it to make it inversely proportionate:

return (100 / (1/totalfitness)) \* (1/individual.fitness))

This would give lower fitness candidates higher weightings as the lower the fitness the better the candidate as this dictates that their route is a shorter distance.

After implementing Roulette I used that as a basis to implement Rank. To dynamically give each candidate a weighting I used the following equation:

Where n is the total weighting of all candidates and i is the iteration count, counting through the population, ordered from best fitness to worst.

For my Recombination I implemented One Point and Two Point PMX Crossover. I read about how One Point PMX Crossover works in [1]. The paper said that PMX was one of the fastest Crossover operators and as the explanation was quite detailed I decided that this would be a good Crossover operator to try to implement myself.

PMX works by taking two parents with numbered ID’s and generating a random number between zero and the total number of Genes in the parent Chromosomes. We look at the gene in the position of the random number in the second parent. We then find that gene in the first parent and swap it with whatever is in the position of the random number in parent one. Swapping the genes as opposed to just changing the gene in the random number position in parent one to the gene in that position in parent two keeps the chromosome valid. By using HashMaps (Dictionary’s) I managed to keep the algorithm on an O(N) complexity.

I then used that implementation to create Two Point PMX Crossover. This was a very simple change, instead of counting down to 0 like the paper suggested, I counted down to a second randomly generated number smaller than the first. Everything between those values would be crossed over. The Crossover implementations also have a user specifiable probability of being used per two parents.

For Mutation I chose two random positions on the Chromosome and swapped them. Similarly to Crossover there was a user specifiable probability of this activating per Chromosome.

### Local Optimum Problem

At this point I had a working Genetic Algorithm which converged! Very quickly. Usually after a couple of hundred iterations. I also noticed that although it consistently found better Chromosomes than what it had when it started, it generally didn’t find particularly optimal ones compared to what I knew were out there. I knew these were out there as on the rare occasion it did stumble on these gems, just not usually.

For me, this wasn’t good enough. I wanted an algorithm that consistently found these almost-perfectly optimal solutions. I debugged the algorithm and saw that it was quickly finding a half decent solution and the entire population was then transformed into copies of this Chromosome. And of course, when two of the same Chromosomes Crossover they produce an identical child. Once this had happened it was highly unlikely to ever find a better solution whether it was run for 500 or 50,000 iterations.

Mutation, even at that low rate could have theoretically found a better one and I did see this happen, once. I decided to explore this concept by implementing what I later found out was called Adaptive Mutation. I measured the population every few iterations and looked to see how many copies of the same Chromosome were present. For each copy I increased the Mutation probability by a small factor. It didn’t achieve what I was hoping. The issue was that firstly by hugely increasing the Mutation rate I wasn’t promoting the breeding of good Chromosomes, I was just changing them randomly. Secondly, Mutation only makes a small change so even with massive Mutation there still wasn’t much fresh material.

I started searching for papers about fighting the local optimum. My theory was that I needed to generate new Chromosomes and add them to the population when the population was converging on an optimum. I came across [2] which talked firstly about Adaptive Mutation (I was happy that I had pre-emptively thought about and implemented a solution the paper proposed) but it did say that ‘a high value to this parameter (Mutation Rate) introduces a certain degree of noise into the system, thus creating serious obstacles to the convergence process.’ Unfortunately, I was relying on a high level of the Mutation rate to overcome the local optimum, really I was just adding a higher amount random noise to the algorithm.

The paper went on to talk about Random Offspring Generation. This was exactly the sort of thing that I’d been looking for. When Crossover of two parents is about to take place the parents are compared. If the parents are the same then in one version of ROG both parents are replaced with randomly generated new Chromosomes, in the second version only one of them is changed. The first version fell into the same issue that I had with my Adaptive Mutation, it did not promote good breeding. The other version did as one of the Chromosomes was good enough that there were multiple versions of it in the population. The papers results showed that changing one of the parents produced better results than changing both.

After implementing this and running the algorithm the results were astonishing. It consistently found far better solutions and would not usually hang forever at a local optimum. These results will be discussed later.

Whilst I liked that the new random offspring’s were creating more genetic diversity in the algorithm I thought something should be done about the randomness of ROG as there are so many terrible Chromosomes. So I went further than the paper and engineered my own version of ROG. I decided that I could use the time between introducing new Chromosomes to find better Chromosomes to introduce. Add some exploitation to the ROG. To do this I created a second thread running parallel to the Genetic Algorithm thread. This thread constantly generated random Chromosomes and then added them to a Concurrent Priority Queue that I wrote (.NET does not have a standard implementation of this). I set the queue to have a limit of 100, each Chromosome added was either placed somewhere in the queue or rejected if it wasn’t good enough. When the Genetic Algorithm required a new parent it would take from the Priority Queue, this meant that it was always using a decent Chromosome as the second parent. This promoted to a much higher degree the aim of breeding together good Chromosomes. I call my version of ROG: Super Turbo Sports ROG (or SROG). SROG did further improve the performance of the Genetic Algorithm.

# Bibliography

1. https://www.ceng.metu.edu.tr/~ucoluk/research/publications/tspnew.pdf

2. http://www4.di.uminho.pt/~mpr/P078.pdf

SROG vs ROG (10 runs each):

SROG:

"best": 395.19519846580954,

"averagebest": 407.50013353104,

"average": 730.4088190779656

(Found best 4x)

ROG:

"best": 395.1951984658096,

"averagebest": 414.6266365301675,

"average": 778.8048243149173

(Found best 2x)

Basic:

"best": 417.0953416460349,

"averagebest": 453.8466070138508,

"average": 467.75017185522336

(Found best 0x)