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.

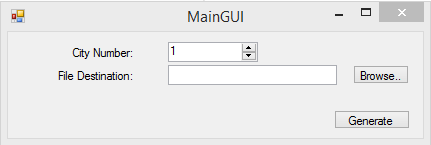


Figure 1

* 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.

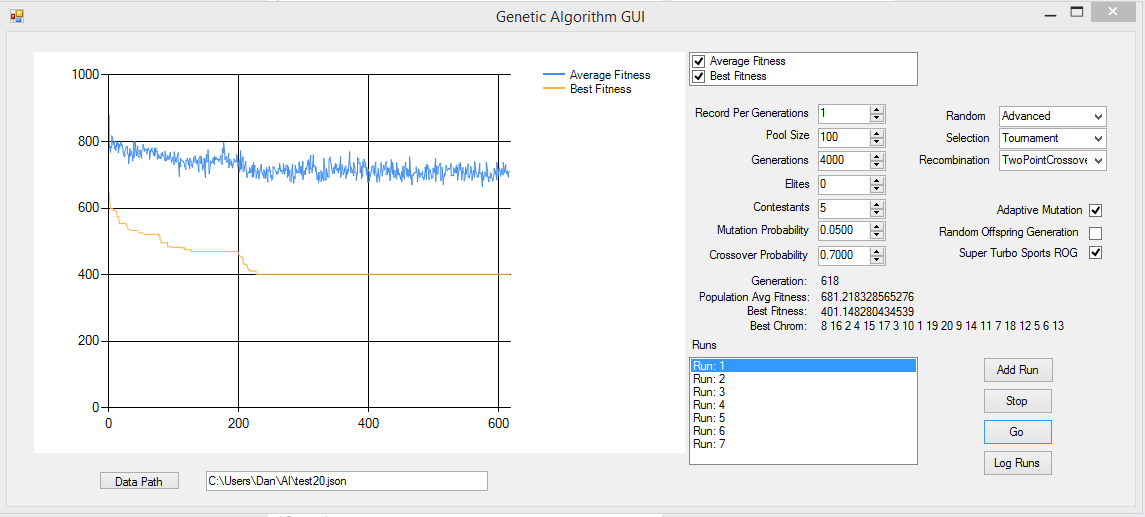


Figure 2

* 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

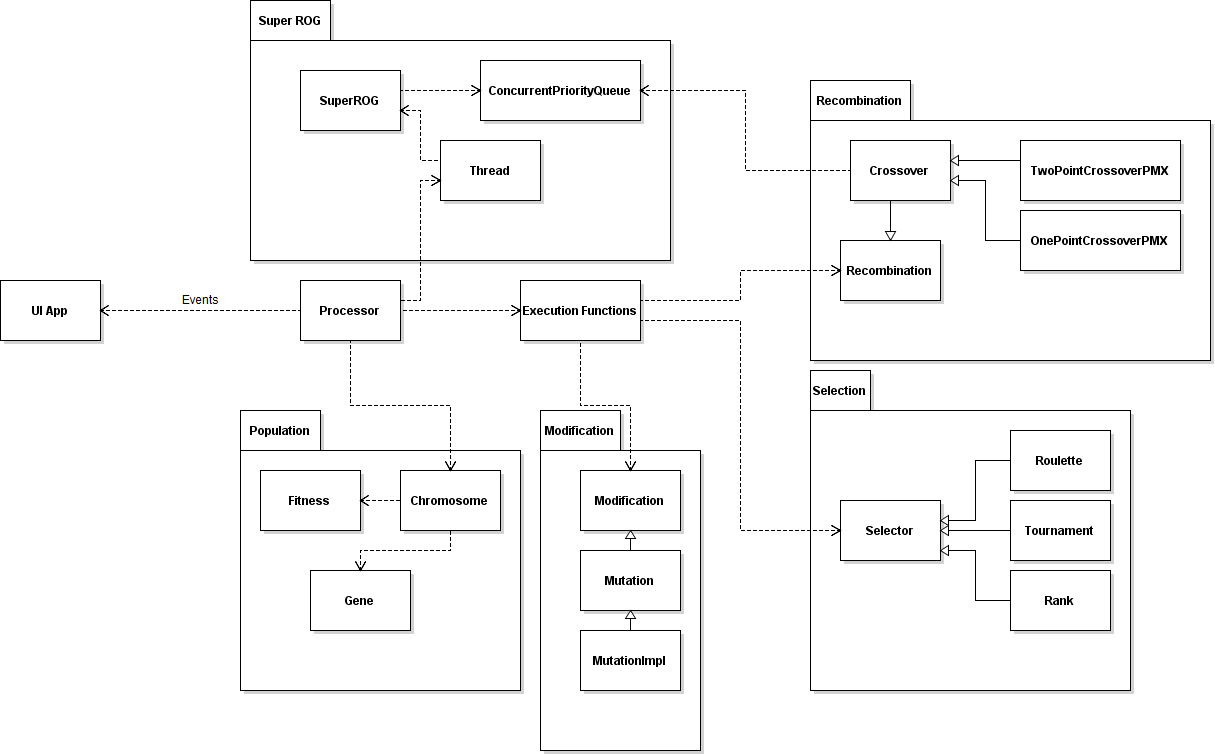


Figure 3

Figure 3 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 if they require altered functionality.

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. At runtime this representation boiled down to a list of City objects. 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. This would give the best few candidates a much higher change of being selected than worse candidates as the worse the candidate the less likely it will be selected, however there is still a chance.

The absolute worse candidate would be picked extremely rarely. For instance with a total weighting of 100 and 20 candidates, the worst candidate would have a (100/2^20) = 0.00001% probability of being picked. That being said, the 5th best candidate would have a 3.125% chance which is a fairly decent weighting for what is probably a fairly mediocre candidate.

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.

Next 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, but apparently 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.

# Results

For the results of my Genetic Algorithm I will first briefly compare some of the standard implementations, including the performance of the Selector and Crossover operators and also the impact of Elitism. Next I will look at how my attempts at getting past the local optimum problem have affected the performance of the Genetic Algorithm. Specifically my implementation of Adaptive Mutation, Random Offspring Generation and my own, altered version of this: SROG.

For each set of results I use a standard setup, any variation for a set will be specified. The standard setup follows:

|  |  |
| --- | --- |
| Option | Value |
| Runs per Result Set | 10 |
| City Number | 30 |
| Pool Size (Chromosome number) | 100 |
| Generations | 1000 |
| Elites | 0 |
| Tournament Contestants | 5 |
| Mutation Probability | 0.05 |
| Crossover Probability | 0.7 |
| Selector Method | Tournament |
| Mutation Method | Two Point |
| Adaptive Mutation | Off |
| Random Offspring Generation | Off |
| SROG | Off |

10 runs are performed for each set (i.e. Measuring mutation rate, set = Mutation rate at 0.05). For each test performed a graph will detail the best overall chromosome and the average best chromosome. Data for each run will be included and a couple of the graphs from individual runs. This results data can be found in the appendices.

I believe that the average best is the best estimate of how well a particular setup performed. Where randomness is a factor - examples being Mutation and a randomly generated initial population - it is always possible to stumble blindly onto brilliant solutions once in a while so the overall best can be misleading as can individual runs.

## Selectors

|  |  |
| --- | --- |
| Option | Value |
| Selector Method | Varied |

The first set of results I took were on the performance of the different selectors.

Rank and Tournament selection perform about as well as each other. Rank has a better overall average whilst Tournament managed to find the best route over the course of the runs.

As I think it’s better to measure the algorithm setups on their performance consistency as opposed to specifically strong runs I would therefore give Rank Selector a slight edge.

Roulette did not do well. I was surprised with the results, but more specifically the curve of the graph. A run showing an example of this can be seen in Figure 5 and a Tournament run can be seen in Figure 6 to compare to. It almost looks like the algorithm is selecting the worst fitness. However, this does not seem to be the case after going through and debugging the code. An issue that does occur is that the weighting differences are very small in a pool of 100. A bad Chromosome may get around a 0.8% probability of being selected, whilst a good Chromosome may get a 1.2%. The difference in probability is small. On the other hand Rank gives a far higher probability of the best Chromosome being selected which more clearly promotes better breeding. This lack of positive breeding may be causing particularly good Chromosomes to be lost which pushes the average best Chromosome up.

A change that could be applied that may fix this would be to change the base from 0 to a small amount below the best fitness. For instance, a Chromosome with a fitness of 210 is twice as good as one with 420. This would be reflected in the pick probability. However in this case 210 is probably an amazing Chromosome whilst 420 is average or bad. Therefore more than a 2x pick probability would be more ideal. If the base was raised from 0 to 200, we could say that the fitness, from the Selectors perspective is 10 (210 – 200) against 220 (420 – 200). The difference in fitness is now a factor of 21, thus giving the good Chromosome a much higher probability of being picked.

## Tournament Contestants

|  |  |
| --- | --- |
| Option | Value |
| Tournament Contestants | 2  3  4  5  6  10  15  20 |

There is an obvious trend towards good results and around 4-6 contestants. The minimum contestant number, 2 is shown to be extremely unproductive and raising the contestant number beyond ten also less efficient, but to a lesser extent.

## Crossover Operators

|  |  |
| --- | --- |
| Option | Value |
| Crossover Method | Varied |

Figure 7 reveals the performance of the two Crossover methods over 10 runs. One Point has the slightly better average whilst Two Point found the far better overall best Chromosome. Whilst I said that the average is more important, it is hard to ignore this discrepancy in their best found Chromosome. Throughout the run Two Point actually found two Chromosomes better than anything One Point found. However it also crept far more often into the 700 fitness domain.

One reason I can think of concerning why Two Point may be more likely to find better Chromosomes is because One Point has a bias in Chromosome swapping. When the point is selected everything before that point gets swapped. This makes Genes further down the Chromosome less likely to be swapped. For instance the first Gene will always be swapped, the final Gene will only be swapped if the random number picks the highest number possible. This skewed crossover could limit the potential search space.

## Crossover Probability

|  |  |
| --- | --- |
| Option | Value |
| Crossover Probability | 1  0.85  0.7  0.5  0.3  0.1  0.01 |
| Generations | 1000  100 |

On the face of it these results were surprising. Using a very small crossover percentage causes very poor performance, but anything beyond and including 0.1 crossover rate creates more or less the same results.

This seemed odd so I looked at the individual runs and noticed that whilst after 1000 generations the results were similar, each of the crossover probabilities made the algorithm converge at slightly different rates, the higher the probability the faster the convergence on average. I ran another test with only 100 generations and it is clear that the lower probabilities are being cut off before convergence more savagely than the higher probabilities, they therefore generally have worse average best Chromosomes than higher probabilities.

## Mutation

|  |  |
| --- | --- |
| Option | Value |
| Mutation Probability | 0.5  0.3  0.1  0.05  0.01  0.005  0.001 |

Really low Mutation rates perform horribly. With a 0.001 probability the algorithm converges almost immediately, getting immediately stuck in a local optimum (if you can even call it an optimum). I wouldn’t have guessed 1300 was even in the better half of the search space.

As the probability is increased the results get progressively better until they stabilize at about 0.05 (5%). It was surprising that really high Mutation rates did not destroy the results.

# Population Size

|  |  |
| --- | --- |
| Option | Value |
| Pool Size | 5000  1000  500  100  10 |

A very small population evidently has a negative effect on the overall performance of the algorithm. That being said, populations past 100 do not get better. In fact, in these results the 100 population did the best. That being said, the variation is small enough that this could be attributed to chance. A larger population has an exponentially large negative effect on speed, therefore a population of around 100 should be optimal.

## Elitism

|  |  |
| --- | --- |
| Option | Value |
| Elitism | 0  1  2  5  10  20  40 |

Elitism adds the n best Chromosomes found so far back into the next iteration, replacing the n worst Chromosomes.

When a small number of Elites are used the average best Chromosome found improves. However, the best Chromosome found did not. ‘No Elites’ may have just been lucky as on average it was worse but the story may be that adding Elitism causes the algorithm to more easily fall into local minimum as it cannot discard good Chromosomes that don’t lead anywhere better. The fact that none of the 60 runs with Elitism used ever found a better Chromosome than in the set without Elitism is suspicious. It was for this reason I decided not to use Elitism in my standard setup.

## Local Optimum Combative Strategies

|  |  |
| --- | --- |
| Option | Value |
| Cities | 20 and 30 |
| Generations | 5000 |
| Adaptive Mutation | Off for first set, then on. |
| Random Offspring Generation | On for third set. |
| SROG | On for fourth set. |

I first ran a set (10 runs) without any of the additional strategies (basic), then one for Adaptive Mutation, Random Offspring Generation (ROG) and SROG. I did this with 20 cities and then again with 30. Each run went on for 5000 generations as a longer run would more easily reveal whether or not the strategies could get out of the local minimum or if the graph would go flat after a couple of hundred runs.

Figure 9 shows the results for the 4 sets, run with 20 cities and Figure 10 with 30 cities. Adaptive Mutation does terribly in both. The average run was worse and the best Chromosome it ever found was also worse than basic. 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, adding noise. Secondly, Mutation only makes a small change so even with massive Mutation there still wasn’t much fresh material.

Random Offspring Generation did brilliantly. Although it cannot be said with absolute certainty, the 411 fitness Chromosome is likely the most optimal route possible. Using ROG this was found a total of 6 times throughout the 10 runs and nothing more optimal was ever found using any method. If it is the most optimal Chromosome this means that when using ROG, an absolutely optimal Chromosome can be found up to 60% of the time using 20 cities. This is far better than the basic algorithm which never found a Chromosome as good as this. The average best was only 4 above the optimal, compared with basic at 66. This makes ROG, on average, 16.5 (66/4) times closer to the optimal solution than basic.

My own variation of ROG (SROG) performs even better. The difference with 20 cities looks fairly minor on the graph. However, SROG found the optimal solution 8 times out of the 10 runs and one of the other two was 412 giving it an up to 80% likelihood of finding the optimal route with 20 cities, an average twice as close to the optimal solution as ROG and 33 (66/2) times closer to the optimal solution than basic.

For 30 cities the difference between SROG and ROG was more extreme. This is likely because 20 cities was not complex enough to see substantial difference overall. SROG found the best solution overall by a long way at 479 fitness. 30 points better than standard ROG and 40 better than basic. On average SROG performed 43 points better than ROG and 75 better than basic. The promotion of good, new genetic material that I engineered into SROG seems to have a huge effect on the overall performance.

A characteristic to note in Figures 11 and 12 is the difference in the average quality of the Chromosomes in the population. This is what the blue line represents. In the basic algorithm the line hugs the best fitness. This is because most of the Chromosomes in the population are the best Chromosome so the average is fairly similar. In ROG and SROG the new data causes a more erratic average that is far worse than the best Chromosome. This is a good thing as it means Chromosome variety.

There is a downside to using ROG and SROG which is that their runtimes are longer, the basic algorithm on average was almost twice as fast as both. On 20 cities a similar result but SROG was slower than ROG. I believe this is probably because the Genetic Algorithm ate through the 100 element sized priority queue and had to wait, at times for the SROG thread to add to the queue. Increasing the queue size could potentially combat this so it could be a memory versus speed trade-off.

Ultimately I do not see this speed loss as a big downside. Speed is very much secondary to results and if an algorithm can get to 5000 generations in half the time but gets stuck forever at a local optima after a fraction of that time then the speed increase really meant nothing.

The speed vs results trade-off between ROG and SROG is more debatable, for more complex routes SROG is not practically any slower, most probably because the algorithm takes more time and therefore isn’t quick enough to empty the priority queue, for complex problems SROG is clearly worth it. On the other hand, for simpler problems a 20 second more time efficient algorithm over 5000 iterations may be worth the fairly small loss in result quality.

# Bibliography

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

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

# Appendices

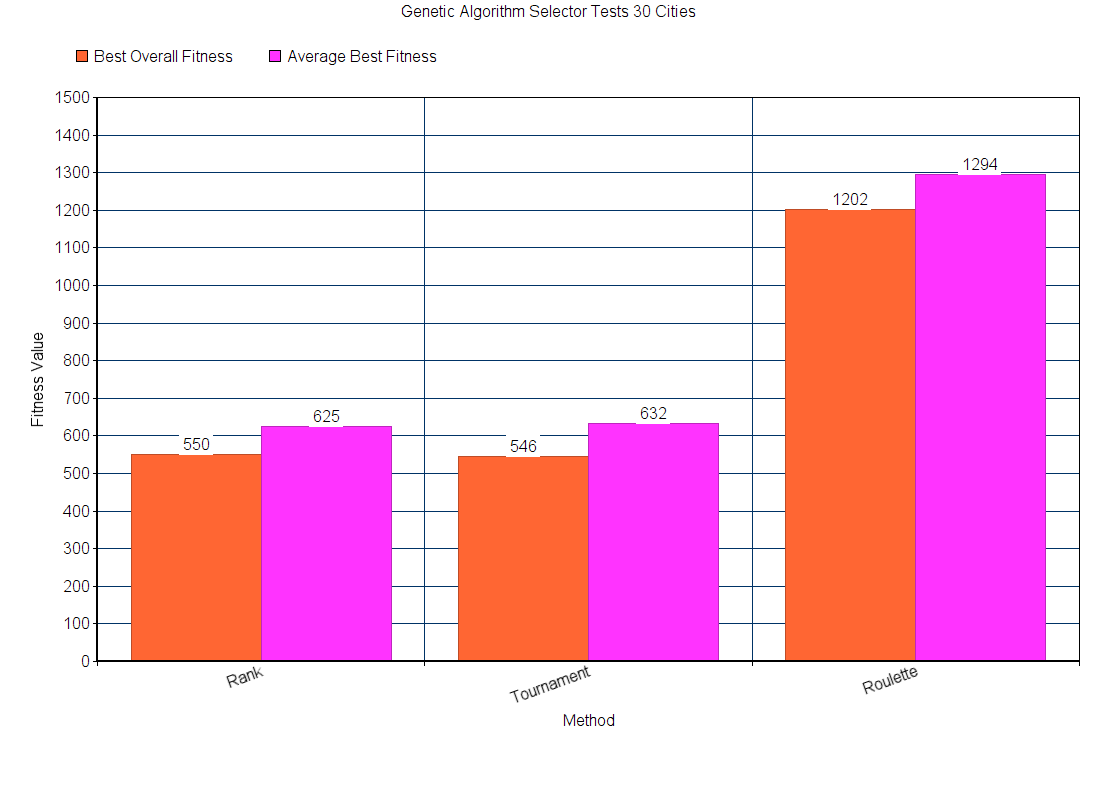


Figure 4 – Overall results comparing Selector performance after 10 runs.

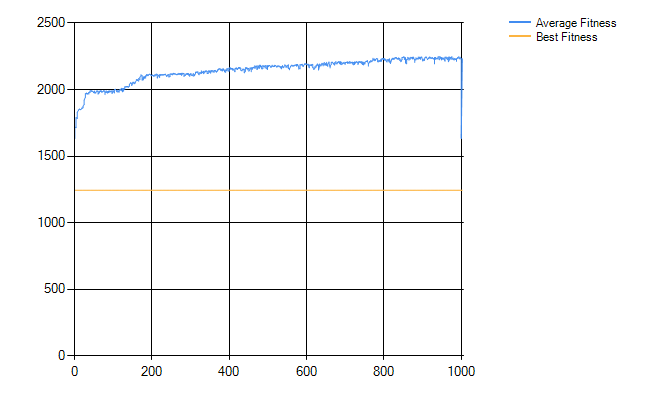


Figure 5 – Roulette run example.

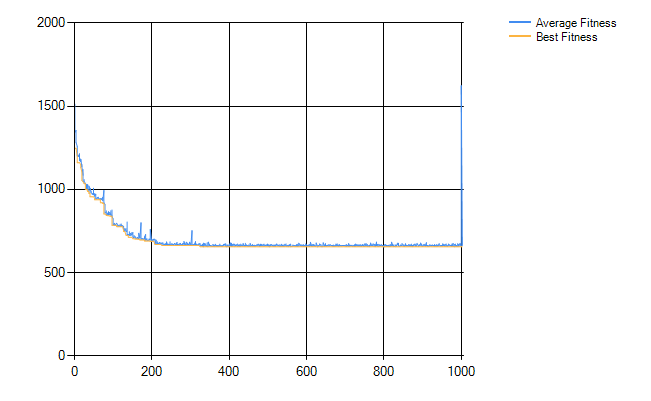
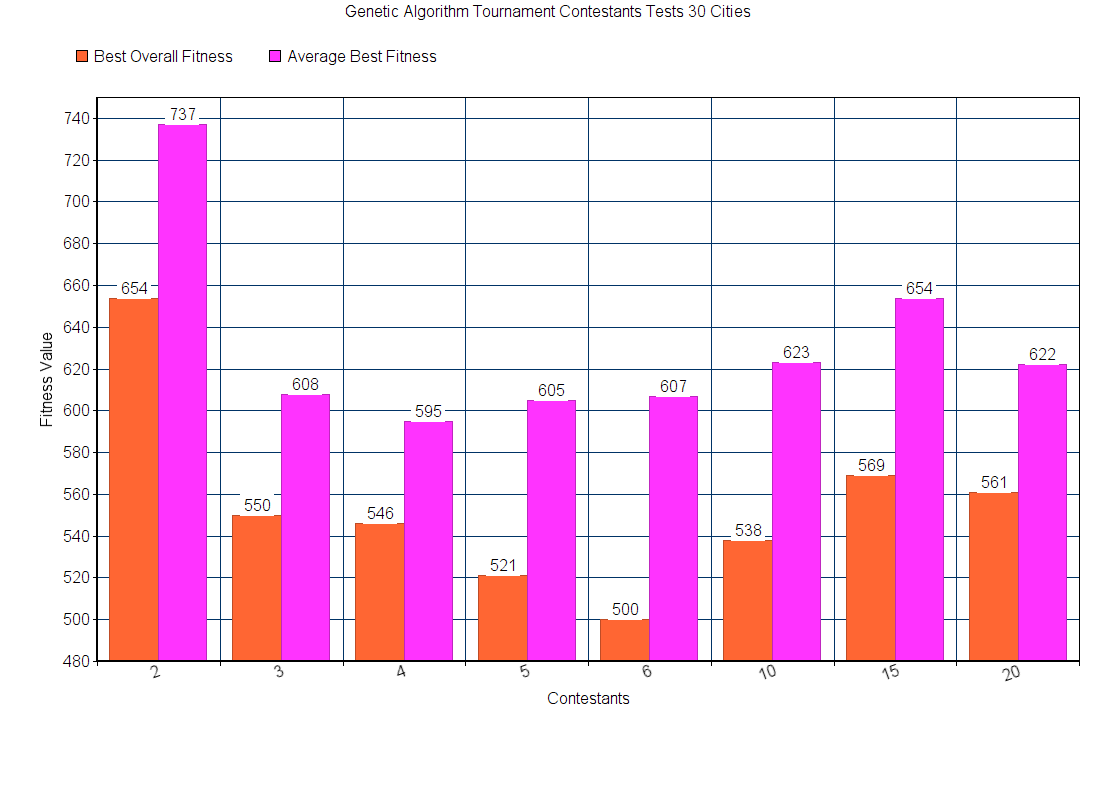


Figure 6 – Tournament run example.



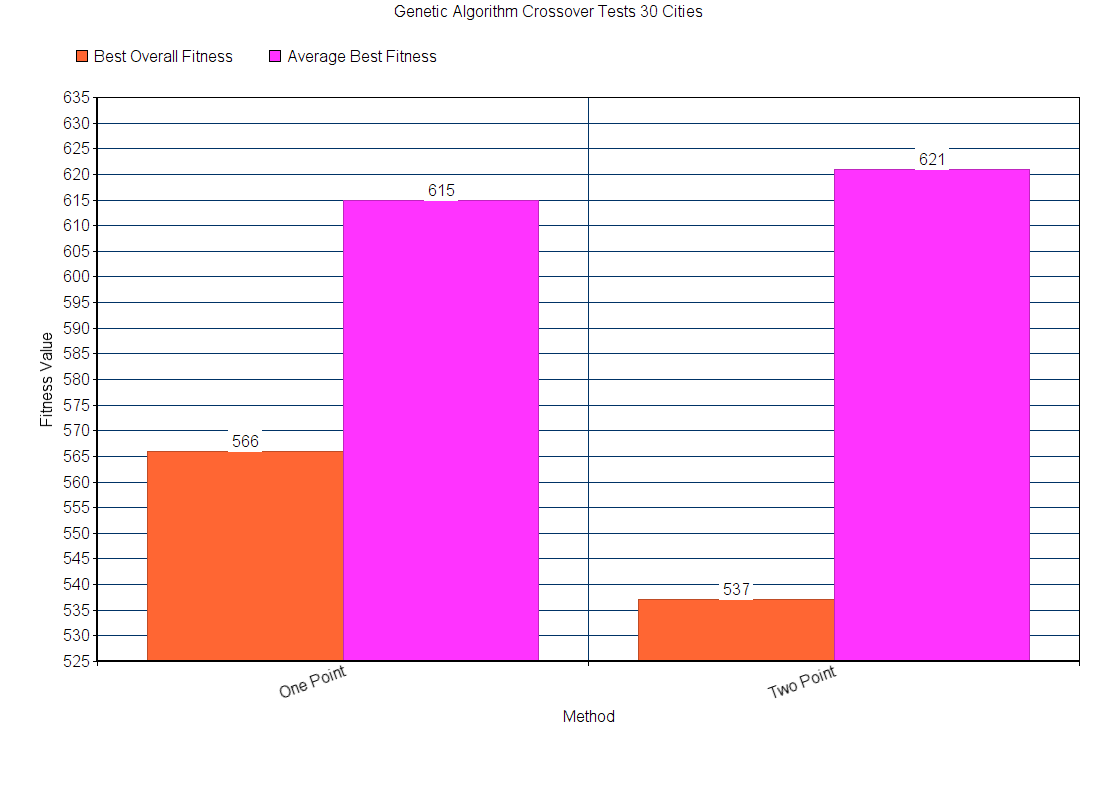
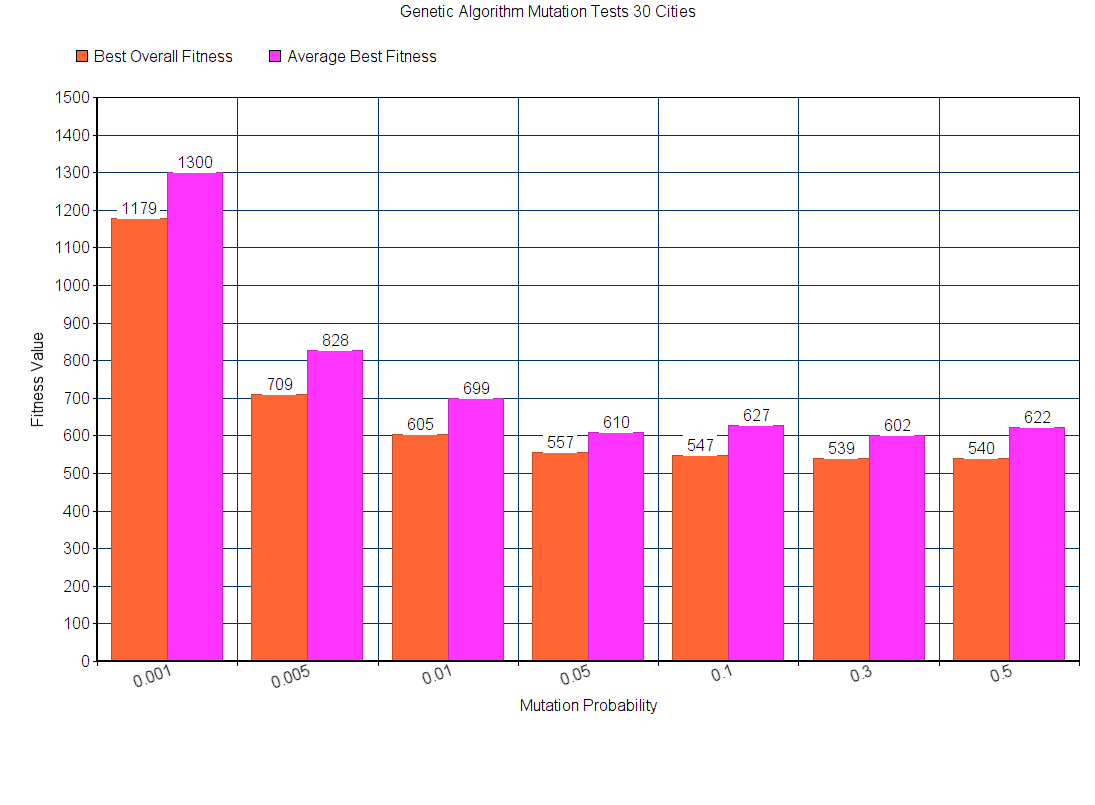
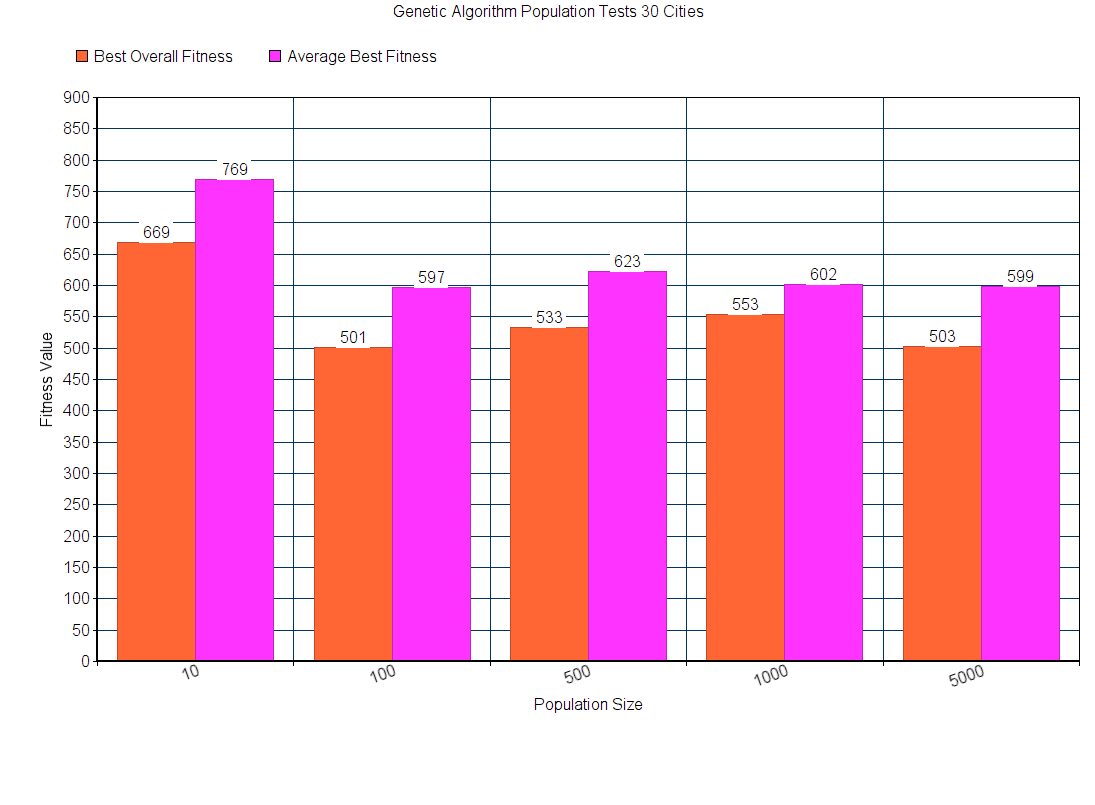
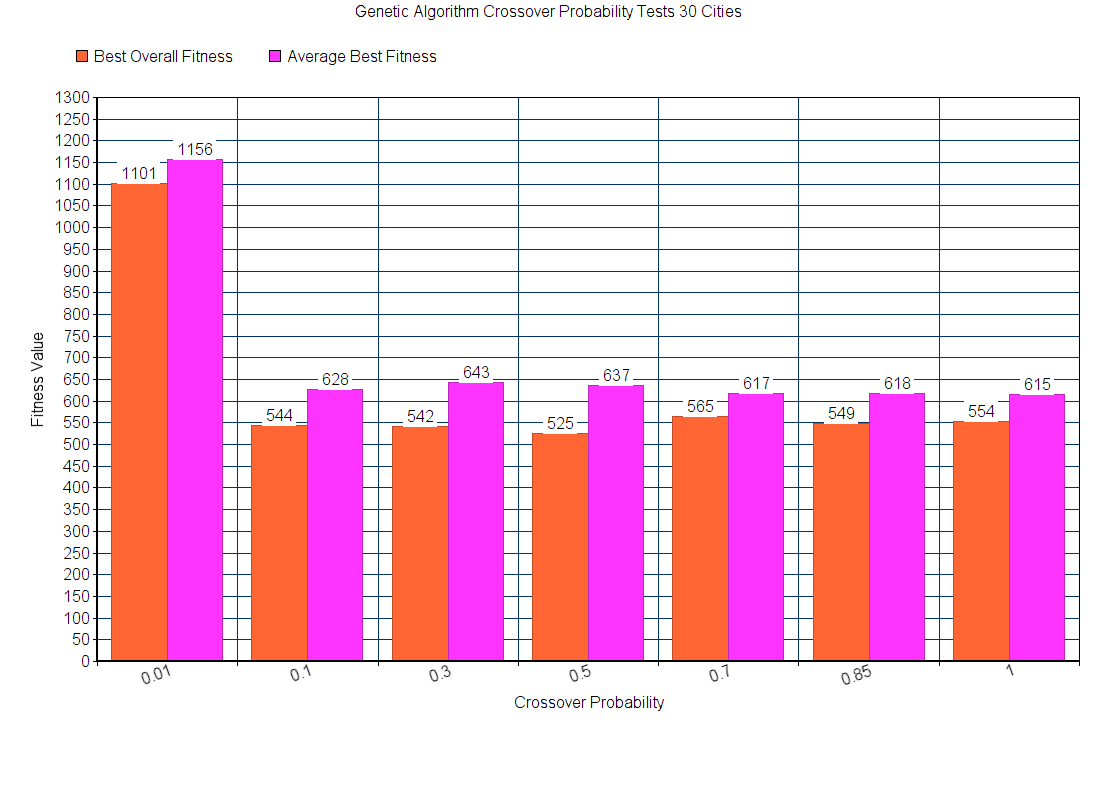
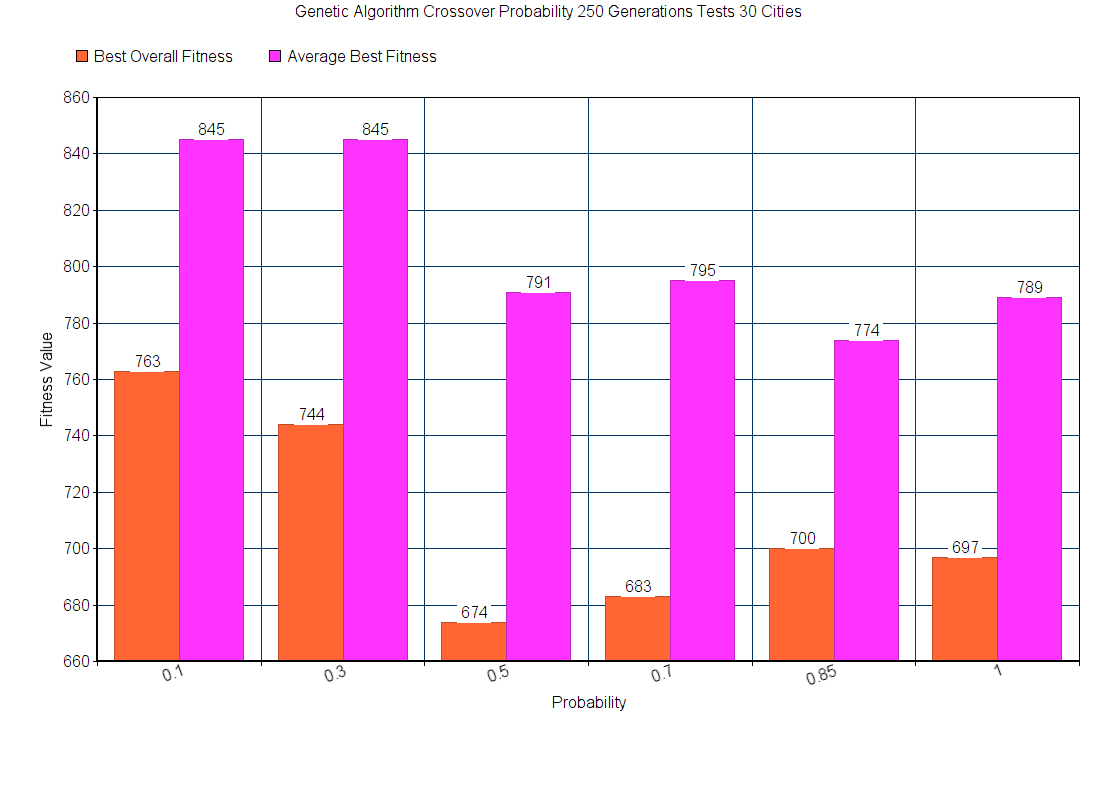


Figure 7 – Overall results comparing Crossover performance after 10 runs.









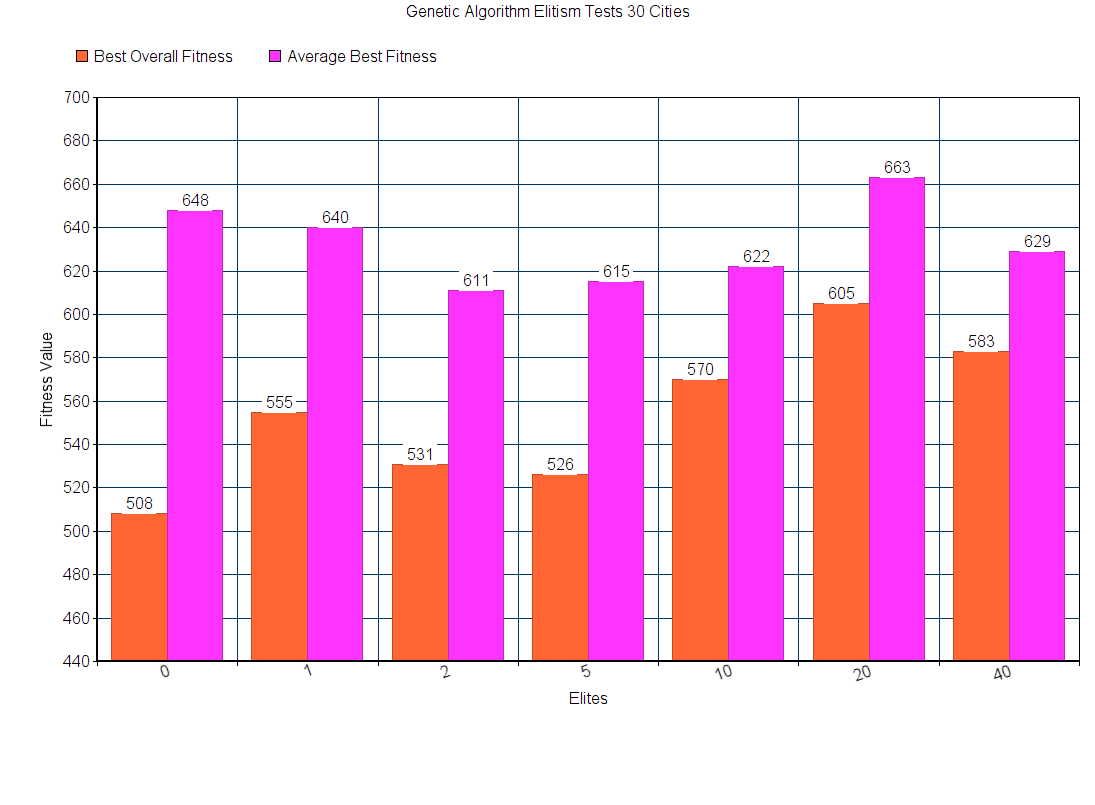


Figure 8 – Compares two sets, one with Elitism on and the other with it off.

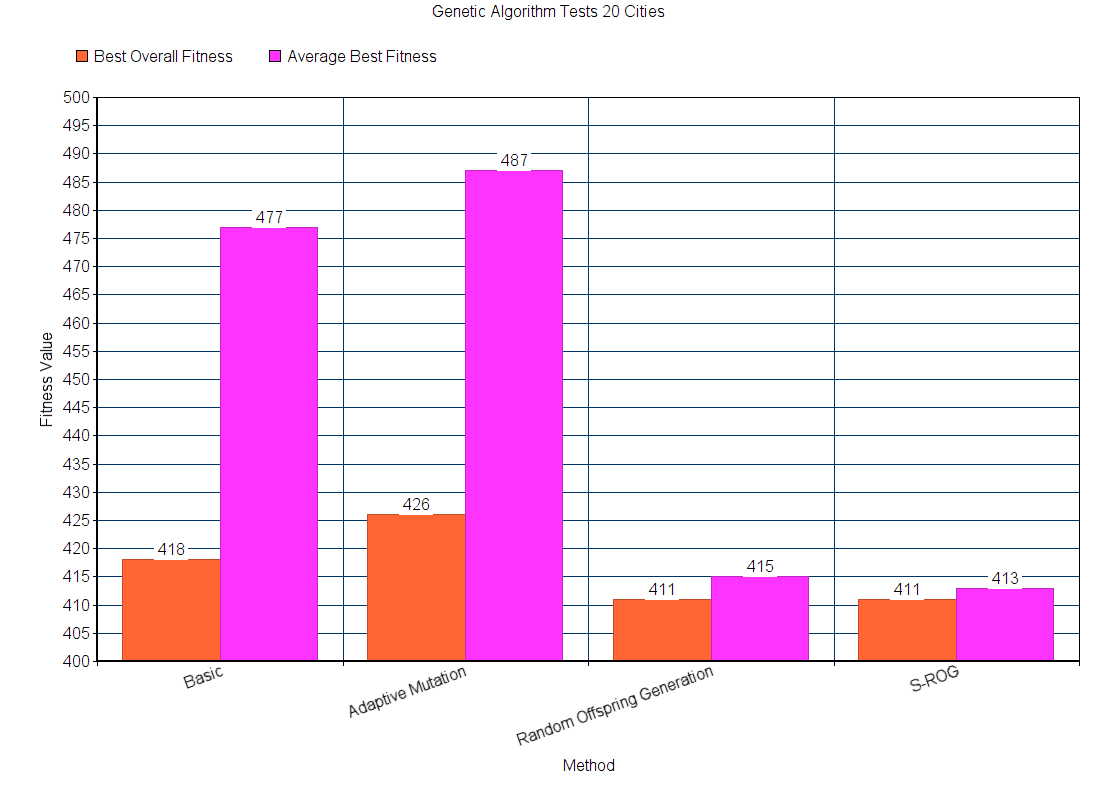


Figure 9 – Combative strategies, 5000 generations, 20 cities.

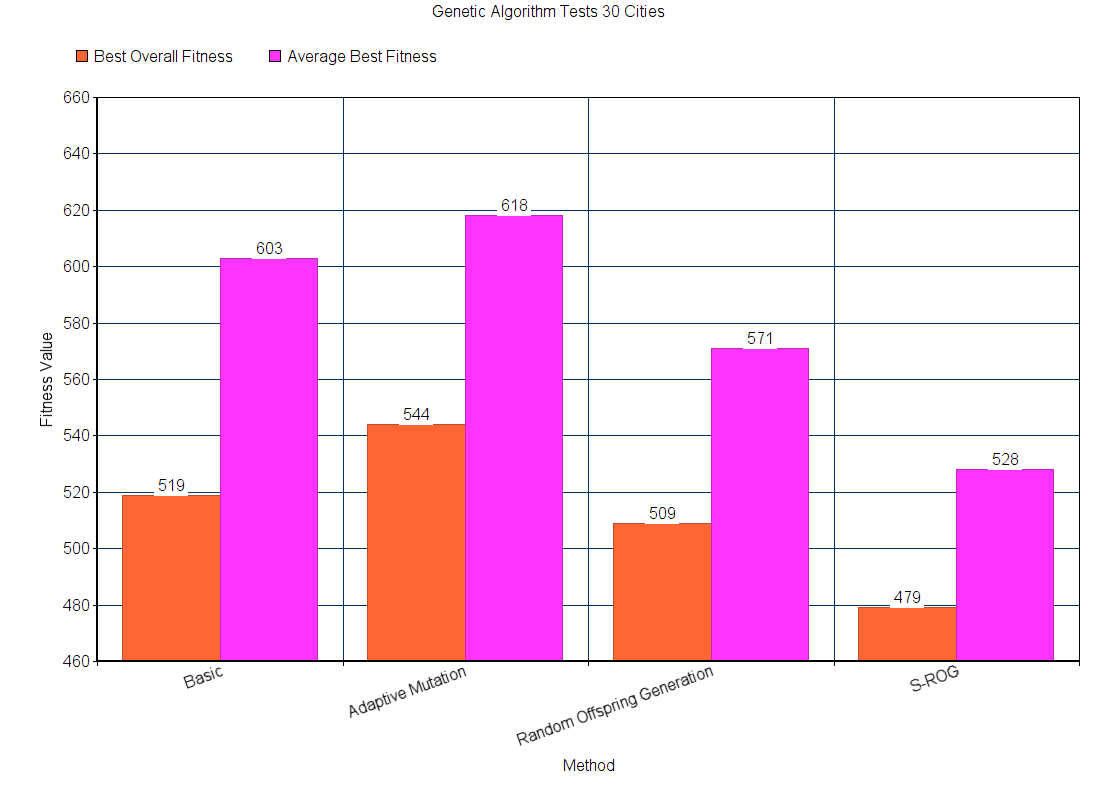


Figure 10 – Combative strategies, 5000 generations, 30 cities.

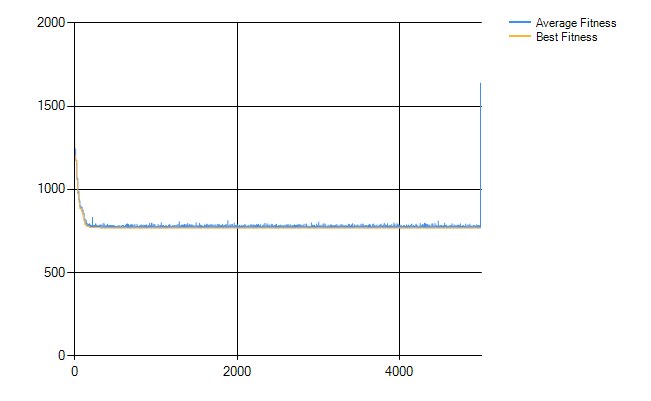


Figure 11 – Basic local optima example.

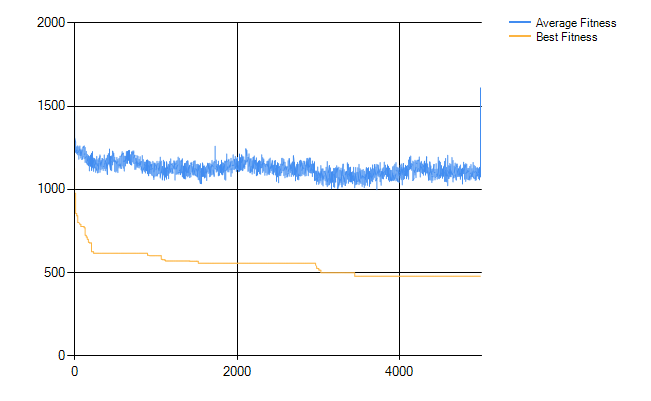


Figure 12 – SROG beating the local optima example.