SEM6120 Assignment 2
Solving Travelling Salesman Problems using Genetic Algorithms

Daniel McGuckin
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 (GAs) and similar methods such as Ant Colony Optimisation are often used to generate optimal or near optimal solutions for the problem. GAs are based on genetics in nature, specifically natural selection: the fittest survives. They use techniques such as Mutation and Recombination to produce new generations of offspring in an attempt to adapt more optimal solutions. TSP adds extra complexity to GAs as each city may appear only once in a solution. Therefore when Mutating and Recombining different Chromosomes, care must be taken to retain legal solutions.

System Design

I chose to write the application in C#.NET. Visual Studio (VS) is, in my opinion, the best IDE and GUI creation with VS is also almost effortless.

TSP City Generator

This was a simple application that generated a user-defined number of cities between a min/max x/y co-ordinate area. It output these cities to a JSON file which was read in by the GA.

Genetic Algorithm Overview

Program Structure

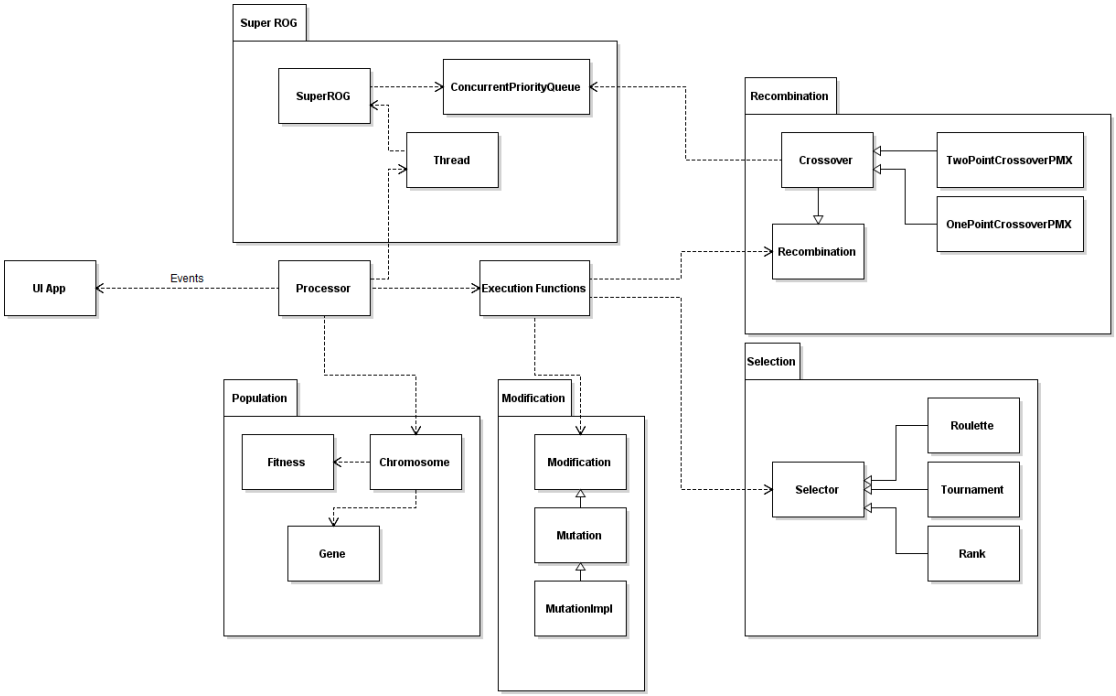

Figure 3

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.

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. This came in use as I used two UI's throughout the project: The console application and then the GUI later on.

Representation

To represent a candidate solution I used a 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.

Genetic Operators

Selection

For my Selection operator I decided to implement a variety: Rank, Roulette and Tournament. I suspected that Roulette would perform worse due to specific candidates dominating the population (as it turned out, the opposite was true) but I had no idea which of the other two would perform better so I implemented them both.

I had some initial implementation issues with Roulette when I realized that the lowest fitness candidates were the best and therefore they had to be weighted highest. To fix the weightings I made them inversely proportionate to the fitness. [ChShPa11-2]

Next I implemented Rank. To dynamically give each candidate a weighting I used the following equation:

$$Candidate\ weighting = \frac{n}{2^i}$$

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 worst.

The absolute worst 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.

Lastly I implemented Tournament by picking n number of Chromosomes from the population, selecting the best one and returning it.

Mutation

For Mutation the operator takes a single Chromosome and switches the positions of two of the genes. Mutation and Crossover both have a user specifiable probability of being activated per Chromosome/Pair of Chromosomes. This is defined in their implementation superclass.

Crossover

For Recombination I implemented One Point and Two Point PMX Crossover. I read about how One Point PMX Crossover works in [Ucoluk-3]. 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. It looks at the gene in the position of the random number in the second parent and then finds that gene in the first parent and swaps it with whatever is in the position of the random number in parent one. Swapping the genes as opposed to just changing the genes in that position to be equal 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 Crossover. 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.

Local Optimum Problem (With additional changes to Mutation and Crossover)

I noticed that whilst the algorithm now worked it converged to local (usually sub) optimal solutions after a few hundred iterations and then generally wouldn't find anything better. I decided this wasn't good enough.

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 adapting the Mutation Operator and 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 [RocNev-5] which talked firstly about Adaptive Mutation (Which I had pre-emptively implemented) 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 Mutation rate to overcome the local optimum, but I was instead just adding more random noise to the algorithm.

The paper went on to talk about Random Offspring Generation [RocNev-6]. 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 papers results showed that changing one of the parents produced better results than changing both [RocNev-8], likely because it better promoted good breeding.

I thought something should be done about the randomness of ROG as there are so many terrible Chromosomes in the search space. 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 Crossover sought a new random parent it would take from the Priority Queue, this meant that it was always using a decent Chromosome as the second parent. This promoted the aim of breeding together good Chromosomes. I call my version of ROG: Less ROG (or -ˈĒl- LROG).

Results

Please look to the Appendices for results tables and graphs.

Each set of results consists of 10 runs and a standard setup is used. 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
LROG	Off

When comparing results 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.

All Convergence rate figures are approximations, calculated by working out where the 'Best-Chromosome-Found' line went flat in each run graph and finding the average for the set.

Selectors

(Figures 4, 5)

Option	Value
Selector Method	Varied

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.

Rank Selector's Chromosomes were slightly better on average. However Rank was also almost a second slower than Tournament, most probably due to the population array being sorted by fitness on each call of the operator. Rank also takes much longer to converge to find a similar quality solution.

Roulette did not do well. I was surprised with the results, but more specifically the curve of the graph (Figure 6) especially compared to Tournament (Figure 7). 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. 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%. 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. Because of this Roulette almost never finds a solution better than what it started with.

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. 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 weighting and probability of being picked.

Tournament Contestants

(Figures 8, 9)

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.

Generally the algorithm converges quicker the more contestants are used. Likely because it focuses on good solutions and therefore travels faster down to the minimum. The speed is quite consistent with under a second difference between all the sets.

Crossover Operators

(Figure 10, 11)

Option	Value
Crossover Method	Varied

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. Convergence rates are similar with One Point converging slightly quicker on average. One Point also took slightly longer on average for its runs. This may be because on average One Point has to do more work during Crossover. It always has to Crossover down to the 0 position whilst Two Point may be able to stop before that.

Crossover Probability

(Figures 12, 13, 14, 15)

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 between 0.1 and 0.7 crossover probability, this also holds true for the speed of the runs. 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

(Figures 16, 17)

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). This does demonstrate how important Mutation is to the GA as without it the algorithm almost immediately gets stuck in a local optimum.

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. I was also initially surprised that higher mutation meant faster convergence. However this does make sense as low mutation rates mean a lower rate of change, it will therefore take more time to find different solutions.

Population Size

Figures (18, 19)

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, although the high convergence rate suggests that possibly it may have found better solutions than it did given more generations to explore them.

Populations past 100 do not get better. One interesting result was the speed, a larger population has a large negative effect on speed as would be expected, however, it is almost a linear increase which suggests that the algorithm is quite efficient verging almost on O(N).

An optimal population number seems to be around 100. Higher populations converge faster but this is not linear. 5000 converges on a similar solution about 4x as fast (generations) and takes over 12x as long (time) to cover the same generations. A 5000 population is therefore only around a third as efficient as 100.

Elitism

(Figures 20, 21)

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.

I was surprised that using Elitism makes the algorithm take longer to converge. Possibly Elitism lowers the rate of change leading to a similar phenomenon to what low Mutation and Crossover rates cause.

Local Optimum Combative Strategies

(Figures 22, 23, 24, 25)

Option	Value
Cities	20 and 30
Generations	5000
Adaptive Mutation	Off for first set, then on.
Random Offspring Generation	On for third set.
LROG	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 LROG. 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 converge after a couple of hundred generations.

Adaptive Mutation (only) does terribly in both of its sets. 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 found the best Chromosome multiple times (411). Using ROG this was found a total of 6 times throughout the 10 runs and nothing more optimal was ever found using any method. The average best was only 4 above this 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 (LROG) performs even better. The difference with 20 cities looks fairly minor on the graph. However, LROG found this optimal solution 8 times out of the 10 runs and sported an average twice as close to this optimal solution as ROG and 33 (66/2) times closer than basic.

For 30 cities the difference between LROG and ROG was more extreme. This is likely because 20 cities was not complex enough to see substantial difference overall. LROG 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 LROG performed 43 points better than ROG and 75 better than basic. The promotion of **good**, new genetic material that I engineered into LROG seems to have a huge effect on the overall performance.

A characteristic to note is the difference in the average quality of the Chromosomes in the population (Figures 26, 27). 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 similar. In ROG and LROG the new data causes a more erratic average that is worse than the best Chromosome. This is a good thing as it means Chromosome variety.

There is a downside to using ROG and LROG 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 LROG 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 LROG 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 downside. Speed is very much secondary to results and if an algorithm can get to 5000 generations in half the time but converges at a local optimum after a fraction of that time then the speed increase really meant nothing. ROG and LROG on the other hand take much longer to converge but this is because they find a solution and instead of converging immediately on it they continue to find better ones.

The speed vs results trade-off between ROG and LROG is more debatable, for more complex routes LROG is not practically any slower, most probably because the algorithm takes more time and therefore isn't quick enough to empty the priority queue, LROG 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.

Concluding Comments

"…one of the major problems usually associated with the use of GAs is the premature convergence to solutions coding local optima" says M.Rocha and J.Neves [RocNev-1]. I saw based on my results that without additional strategies GAs suffer from converging on non-optimal solutions. Using different selectors and crossover operators, adding Elitism, finding optimal Mutation and Crossover rates may improve the results to some degree but the problem still persists. On the other hand adding new genetic material goes a long way towards solving this issue.

Fast convergence is not a positive attribute of the algorithm if it's converging on a bad solution. I believe a much slower convergence that delves deep into the search space to find quality solutions is far more useful. I therefore see the local optimum strategies that I employed to be absolutely necessary when developing a GA for TSP. Employing one of the foundations of Genetic Algorithms: Random Exploitation with the local optimum combative strategy further improved results. Using LROG, GAs are an adequately suitable solution for TSP, without a similar strategy to combat local optimum I do not believe they are for more complex problems.

Potential work for the future would be to create a Genetic Algorithm API with my solution. I would also like to further explore, adapt and refine my LROG local optimum combative strategy.

Bibliography

Papers

[Ucoluk] Gokturk Ucoluk, "Genetic Algorithm Solution of the TSP Avoiding Special Crossover and Mutation", Department of Computer Engineering Middle East Technical University 06531 Ankara, Turkey, https://www.ceng.metu.edu.tr/~ucoluk/research/publications/tspnew.pdf

The paper implements their own Crossover Operator, but also explains how PMX crossover works which I used to build my own implementation of it.

[RocNev] Miguel Rocha and Jose Neves, "Preventing Premature Convergence to Local Optima in Genetic Algorithms via Random Offspring Generation", Departamento de Informatica Universidade do Minho Largo do Paco, Portugal, http://www4.di.uminho.pt/~mpr/P078.pdf

This paper looks at ways to prevent 'premature convergence to local optima'. Included in the paper was an explanation of Adaptive Mutation and Random Offspring Generation.

[ChShPa11] Chetan Chudasama, S.M.Shah, Mahesh Panchal, "Comparison of Parents Selection Methods of Genetic Algorithm for TSP", International Conference on Computer Communication and Networks, 2011, http://research.ijcaonline.org/comnet/number1/comnet1019.pdf

The paper compares different selection methods. Specifically Roulette, Tournament, Rank and Elitism. It also gives brief explanations on how these are methods work.

Online Resources

1. http://scottlilly.com/create-better-random-numbers-in-c/

After reading an article about how many standard random number generator libraries are not actually that random I decided to implement a more advanced random number generator using the .NET cryptography library. This article is a tutorial on how to achieve this.

2. http://www.theprojectspot.com/tutorial-post/applying-a-genetic-algorithm-to-the-travelling-salesman-problem/5

I used this article to work out how to implement Elitism. I first implemented it like the tutorial however I realised that this tutorial never includes the elites in the mutation or crossover, they just sit there never lending their genetic material to the population. I ended up implementing it differently, the Elites would be recorded, then go through the algorithm crossing over with other candidates and being mutated, they would then be added back into the algorithm at the end.

Appendices

Further data about individual runs and their associated graphs can be found in the Results folder with the submitted program.

Data

Selector Operator Data

Operator	Best Chromosome	Average Best Chromosome	Average Population Fitness	Average Run Time (mm:ss:ms)	Average Convergence Rate (Gens)
Rank	550.17	624.71	733.97	00:10:58	636
Tournament	546.36	632.41	707.40	00:09:82	474
Roulette	1201.92	1294.80	2176.16	00:09:50	1 (Almost never finds a better Chromosome)

Figure 4

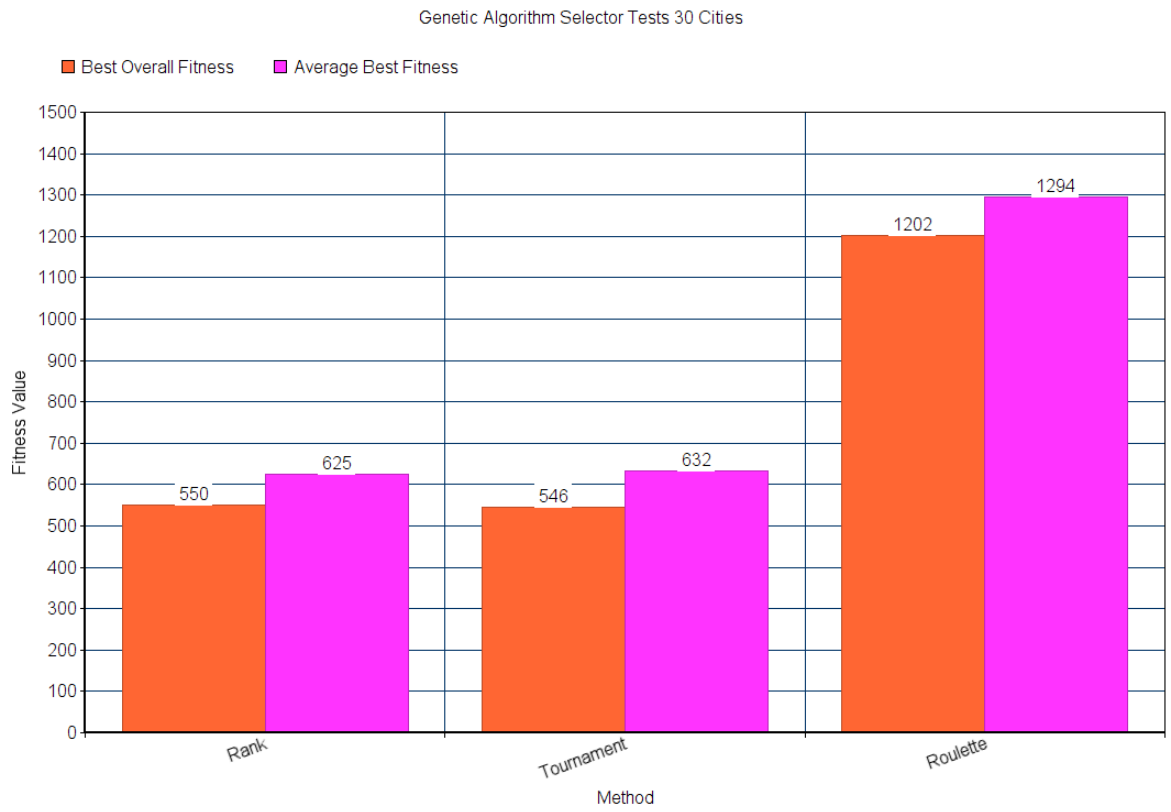

Figure 5 – Results comparing Selector performance after 10 runs.

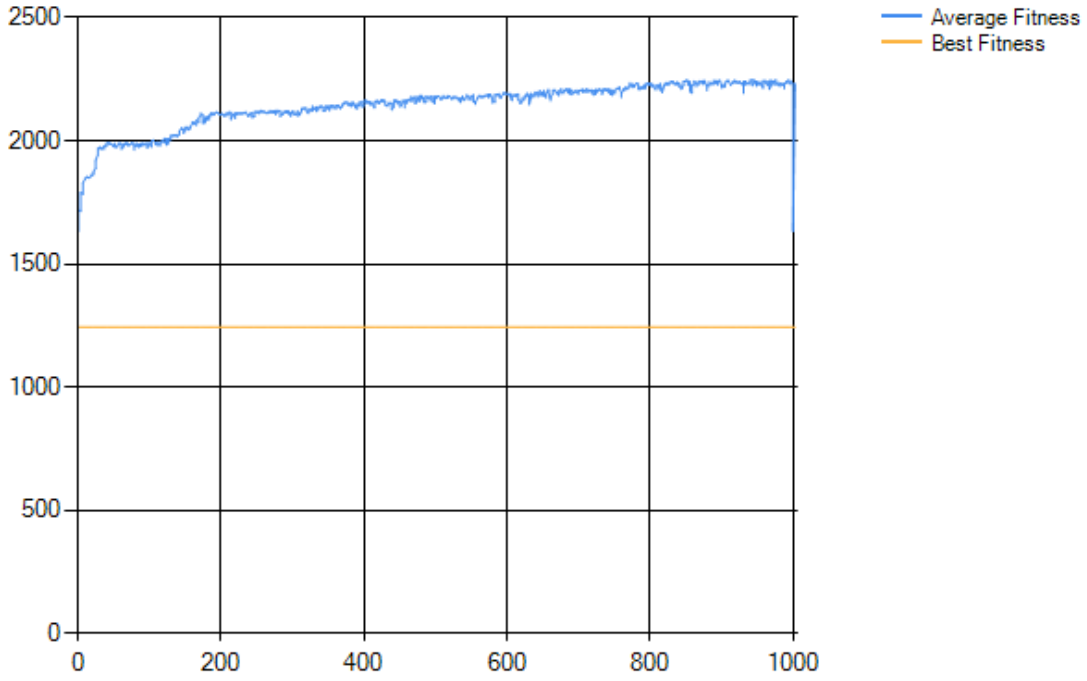

Figure 6 – Roulette run example.

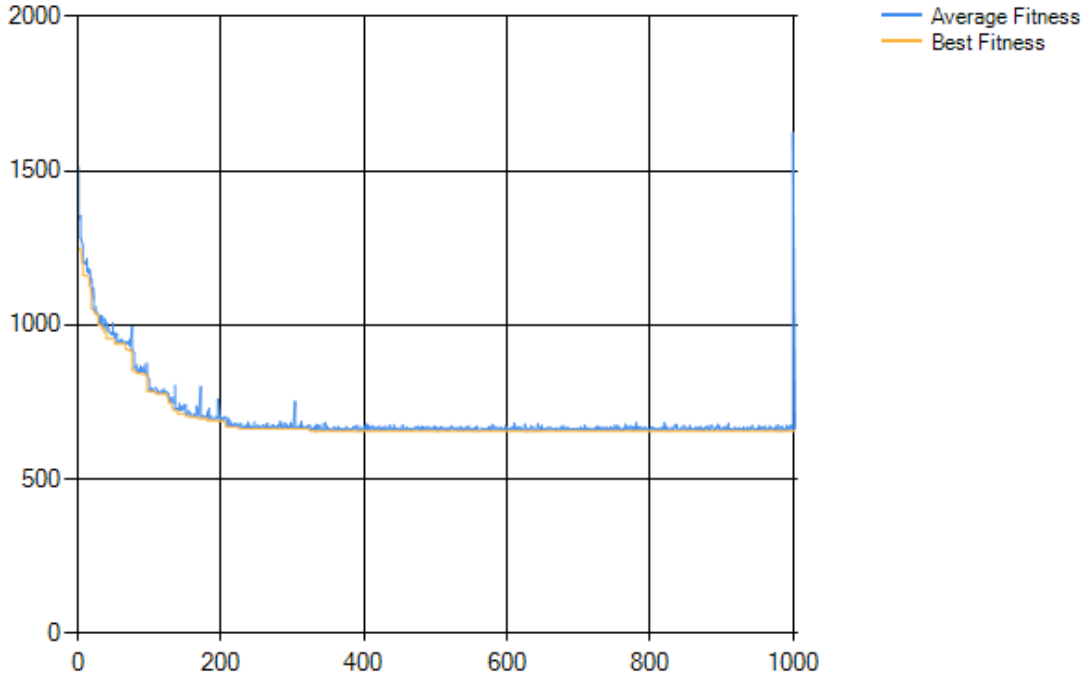

Figure 7 – Tournament run example.

Tournament Contestant Data

Contestants	Best Chromosome	Average Best Chromosome	Average Population Fitness	Average Run Time (mm:ss:ms)	Average Convergence Rate (Gens)
2	654.01	736.54	926.84	00:07:74	848
3	549.60	608.25	680.77	00:08:40	504
4	546.44	595.17	675.07	00:08:52	428
5	520.75	605.10	675.28	00:07:99	411
6	499.71	607.08	676.82	00:08:46	382
10	538.23	623.11	687.03	00:08:00	419
15	568.66	653.86	716.96	00:08:02	532
20	561.25	621.57	694.97	00:08:63	516

Figure 8

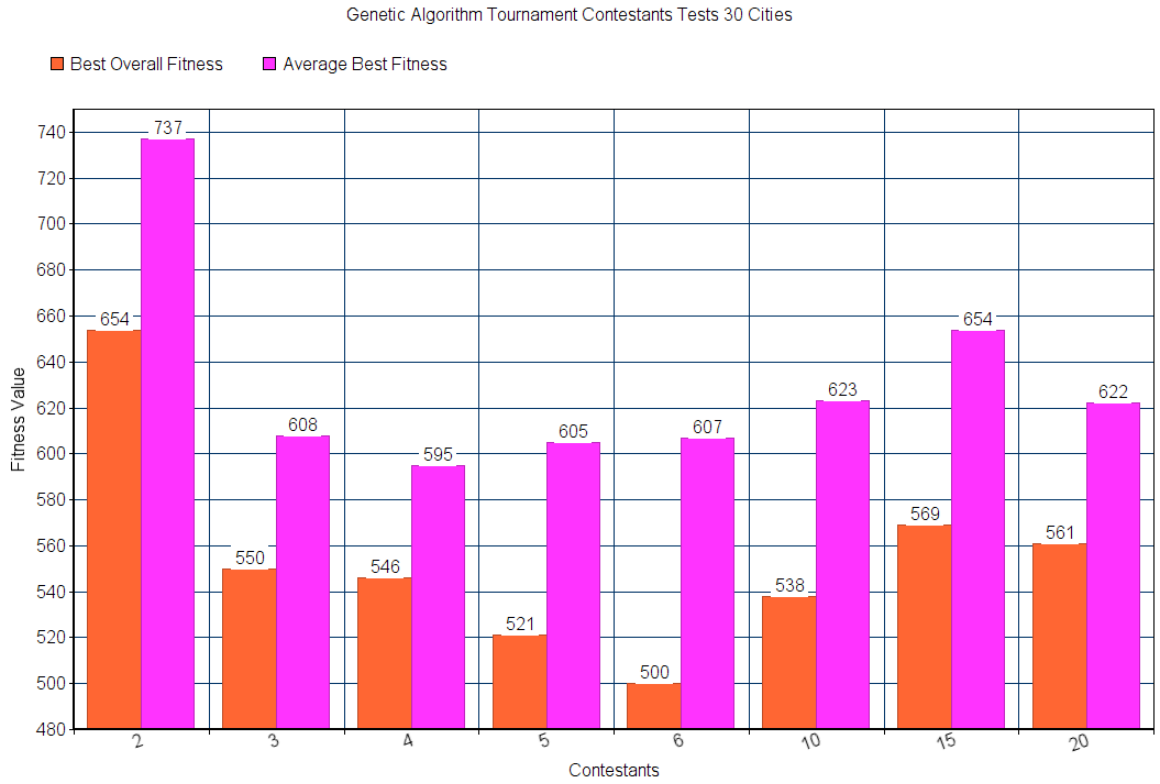

Figure 9 - Results comparing Tournament Contestant number performance after 10 runs.

Crossover Operator Data

Operator	Best Chromosome	Average Best Chromosome	Average Population Fitness	Average Run Time (mm:ss:ms)	Average Convergence Rate (Gens)
One-Point	566.34	614.68	694.71	00:06.70	481
Two-Point	536.85	621.59	693.33	00:06:51	498

Figure 10 – Crossover Operator test results.

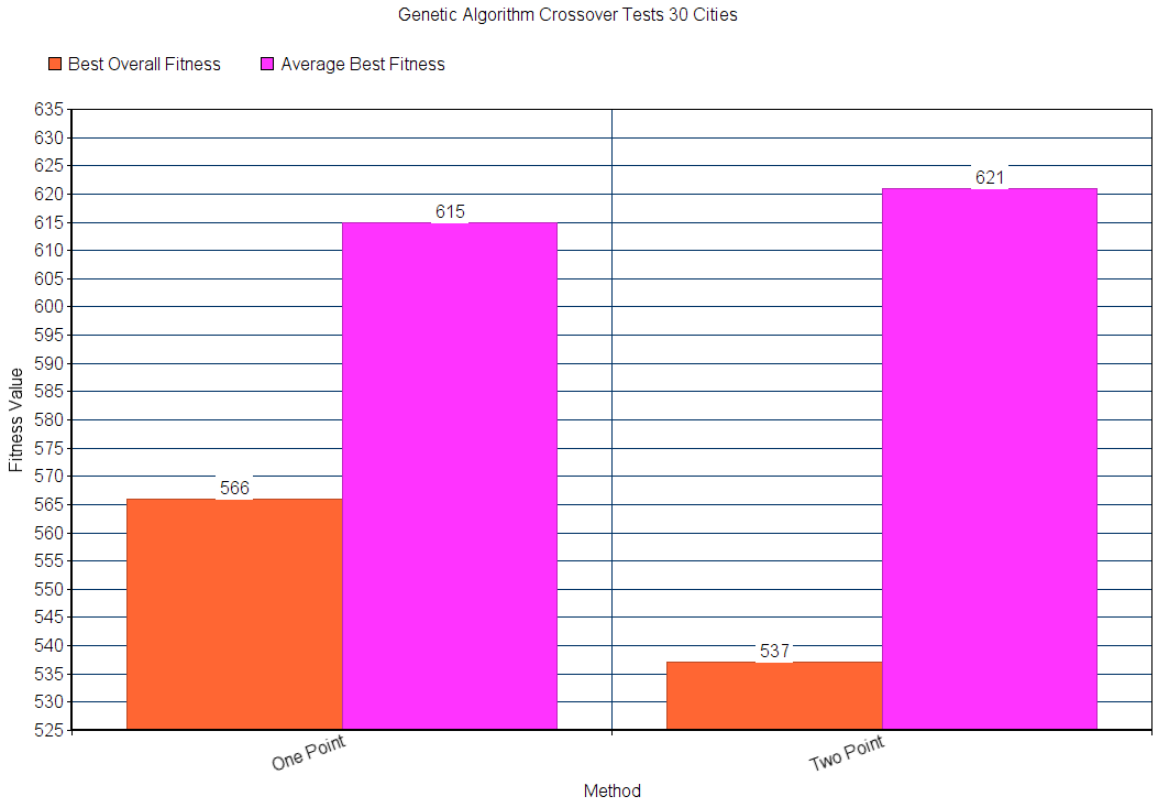

Figure 11 – Results comparing Crossover performance after 10 runs.

Crossover Probability Test Data

1000 Generations

Crossover Probability	**Best Chromosome**	**Average Best Chromosome**	**Average Population Fitness**	**Average Run Time (mm:ss:ms)**	**Average Convergence Rate (Gens)**
0.01	1100.82	1156.44	1573.69	00:06:71	355
0.10	544.22	627.83	829.84	00:07:60	798
0.30	542.18	643.24	743.17	00:07:15	532
0.50	525.25	637.16	714.64	00:06:55	467
0.70	564.95	617.12	690.49	00:06:33	448
0.85	549.20	618.54	697.12	00:07:85	532
1.00	554.58	614.76	693.92	00:06:93	513

Figure 12 – Crossover Probability 1000 generations test results.

100 Generations

Crossover Probability	**Best Chromosome**	**Average Best Chromosome**	**Average Population Fitness**	**Average Run Time (mm:ss:ms)**	**Average Convergence Rate**
0.10	763.48	845.27	1103.38	00:00:62	NA
0.30	744.73	845.02	1037.33	00:00:65	NA
0.50	674.43	791.38	977.48	00:00:81	NA
0.70	682.92	794.88	964.78	00:00:79	NA
0.85	700.17	773.67	948.00	00:00:77	NA
1.00	697.04	789.23	970.00	00:00:77	NA

Figure 13 – Crossover Probability 100 generations test results.

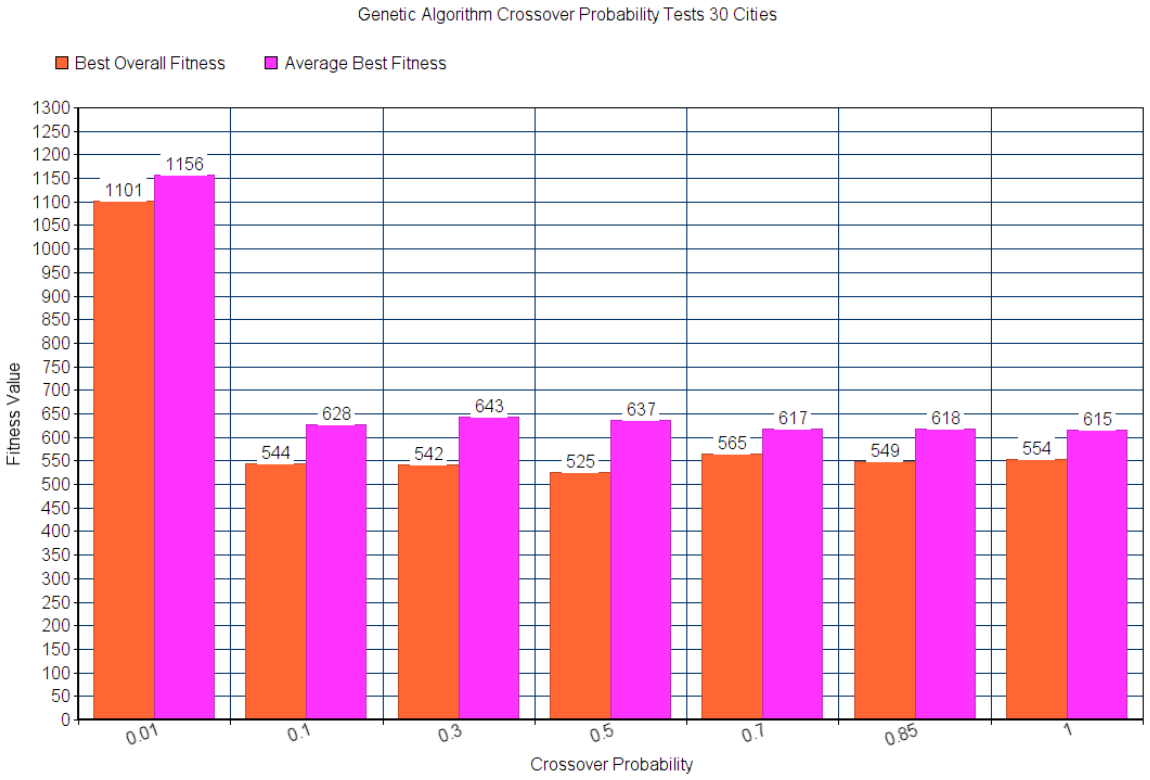

Figure 14 – Results comparing Crossover Probability, 1000 generations.

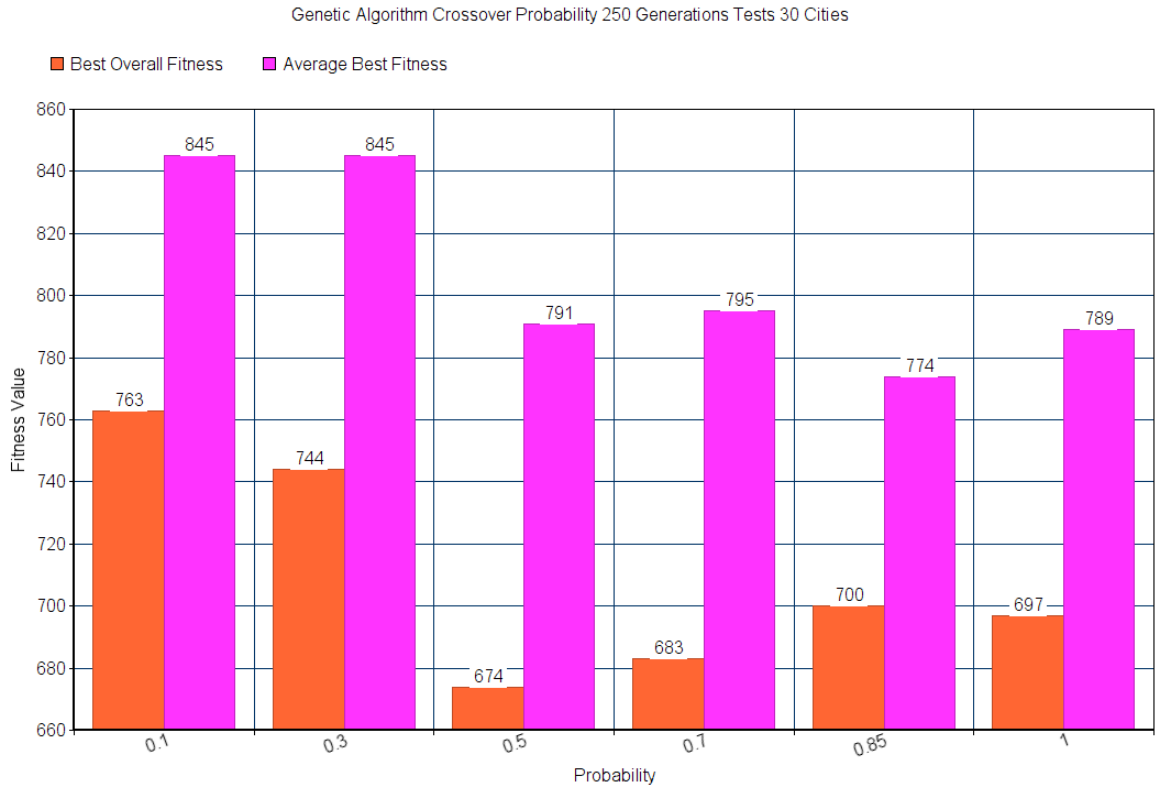

Figure 15 – Results comparing Crossover Probability, 250 Generations.

Mutation Probability Test Data

Mutation Probability	Best Chromosome	Average Best Chromosome	Average Population Fitness	Average Run Time (mm:ss:ms)	Average Convergence Rate (Gens)
0.001	1179.47	1300.27	1318.11	00:09:22	1 (Converges almost immediately)
0.005	709.21	827.65	966.14	00:07:52	860
0.010	605.23	698.72	842.36	00:06:77	851
0.050	556.89	609.61	684.71	00:06:29	464
0.100	546.74	627.45	684.56	00:07:30	342
0.300	539:34	602.38	684.62	00:06:42	159
0.500	540:04	622:52	748:17	00:06:12	281

Figure 16 – Mutation Probability test results.

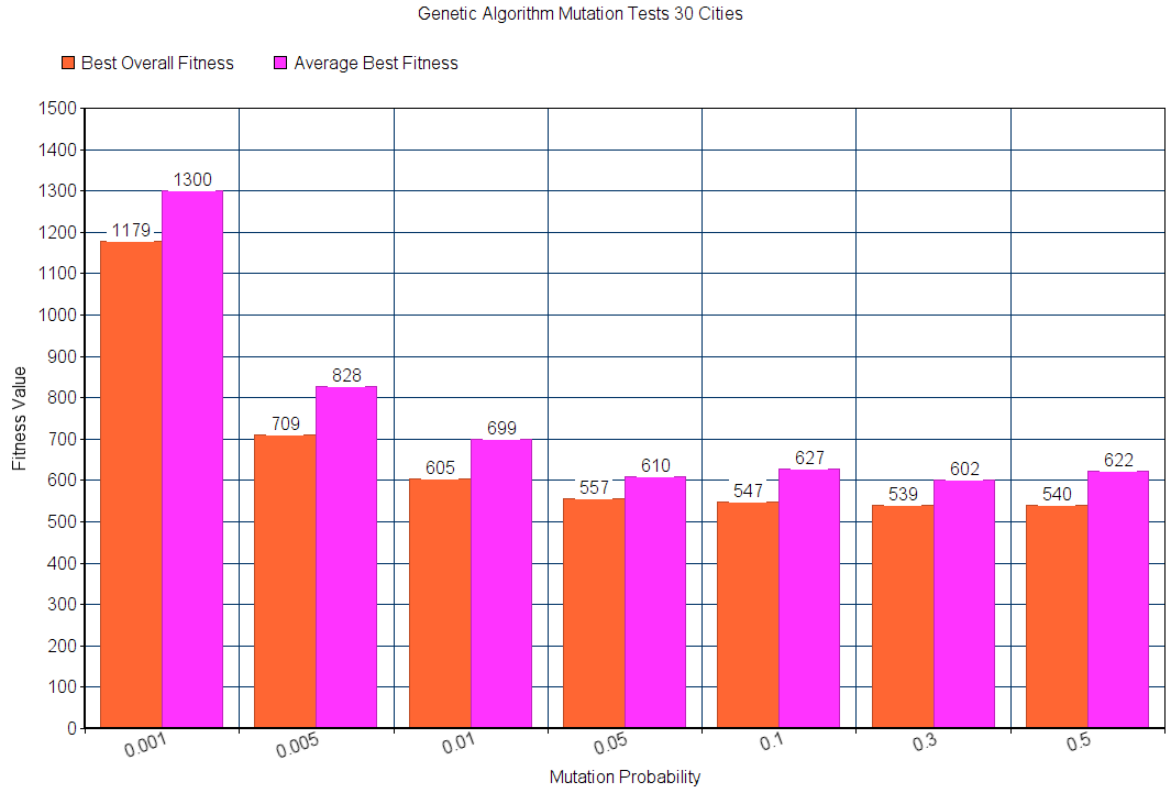

Figure 17 – Results comparing Mutation Probability.

Population Test Data

Population Size	Best Chromosome	Average Best Chromosome	Average Population Fitness	Average Run Time (mm:ss:ms)	Average Convergence Rate (Gens)
10	669.31	769.01	950.97	00:06:88	887
100	501.48	597.62	679.62	00:07:33	518
500	532.94	623.31	662.58	00:10:73	207
1000	552.75	602.19	646.84	00:19:89	186
5000	503.64	599.18	664.44	02:05:73	142

Figure 18 – Population test results.

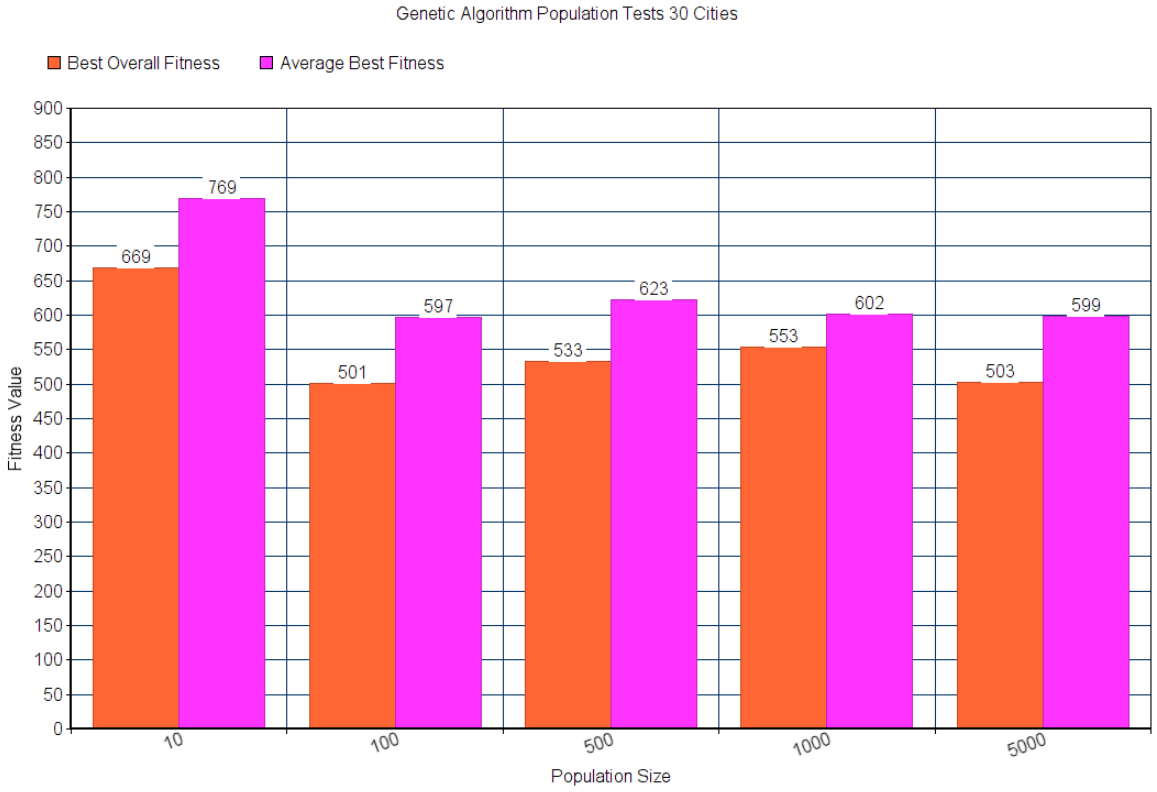

Figure 19 – Results comparing Population numbers.

Elitism Test Data

Crossover Probability	Best Chromosome	Average Best Chromosome	Average Population Fitness	Average Run Time (mm:ss:ms)	Average Convergence Rate (Gens)
0	508.16	648.56	714.71	00:11:16	384
1	555.36	640.03	703.00	00:08:27	422
2	531.48	611.04	679.90	00:11:00	477
5	526.08	614.76	679.52	00:08:35	339
10	570.15	621.88	707.03	00:08:36	599
20	606.00	663.34	737.62	00:07:93	465
40	582.76	629.32	698.70	00:08:28	434

Figure 20 – Elitism test results.

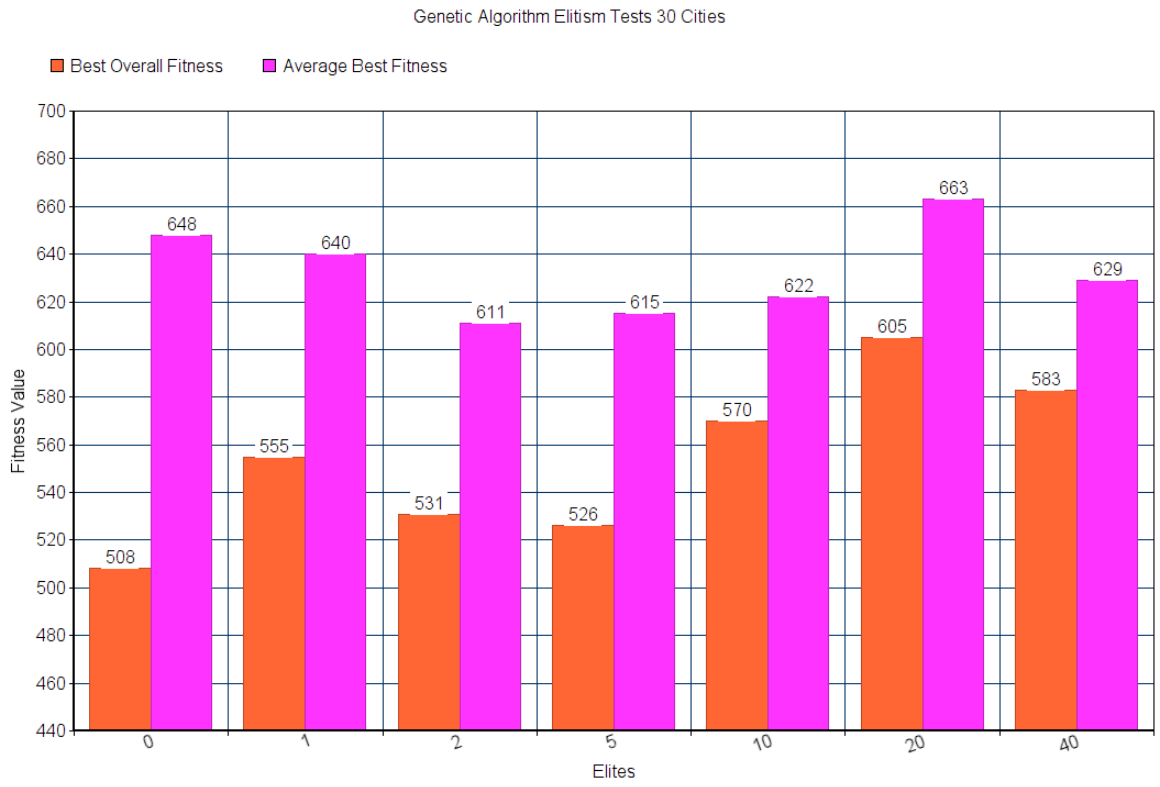

Figure 21 – Compares two sets, one with Elitism on and the other with it off.

Local Optimum Combative Strategy Test Data

20 Cities

Local Optimum Combative Strategy	**Best Chromosome**	**Average Best Chromosome**	**Average Population Fitness**	**Average Run Time (mm:ss:ms)**	**Average Convergence Rate (Gens)**
Basic	417.92	476.91	490.46	00:27:47	188
Adaptive Mutation	425.71	487.34	505.16	00:40:33	535
ROG	411.13	415.50	857.02	00:54:05	2345
LROG	411.13	413.21	780.29	01:14:00	2300

Figure 22 – Local Optimum Strategy test data for 20 cities.

30 Cities

Local Optimum Combative Strategy	**Best Chromosome**	**Average Best Chromosome**	**Average Population Fitness**	**Average Run Time (mm:ss:ms)**	**Average Convergence Rate (Gens)**
Basic	519.20	602.73	629.80	00:56:41	695
Adaptive Mutation	544.05	618.26	639.40	00:38:60	455
ROG	509.01	571.00	1230.34	01:43:23	2610
LROG	478.61	527.86	1138.81	01:43:90	3120

Figure 23 – Local Optimum Strategy test data for 30 cities.

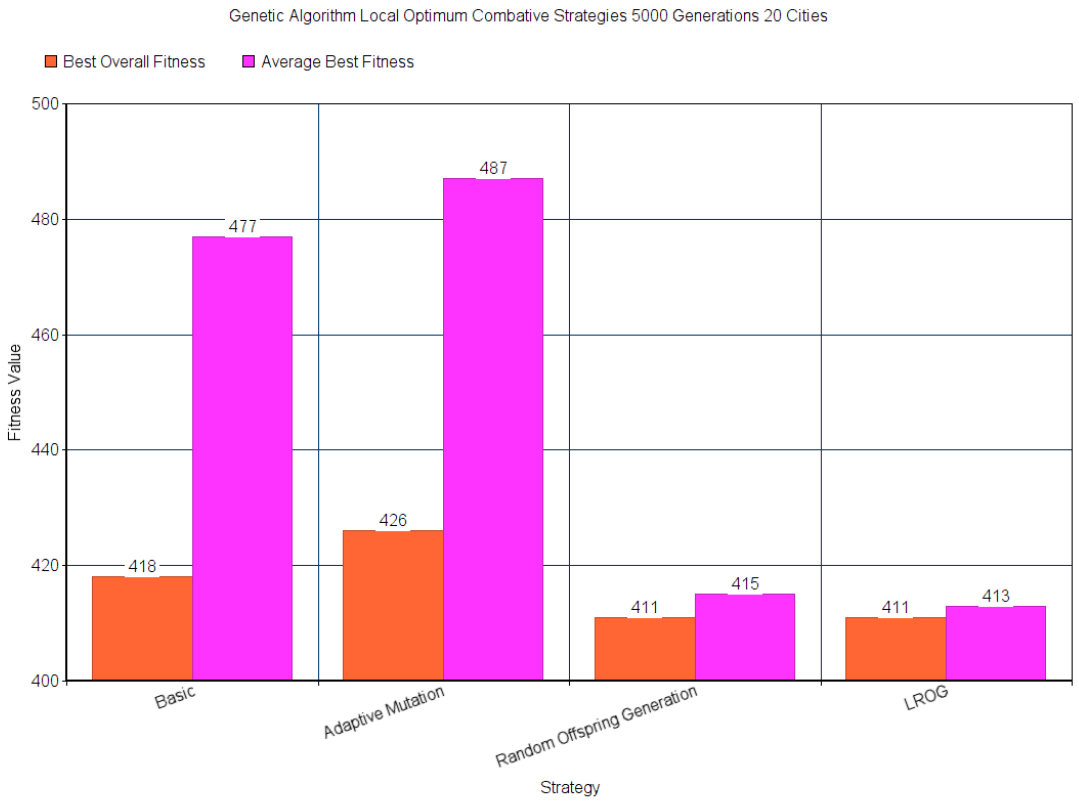

Figure 24 – Combative strategies, 5000 generations, 20 cities.

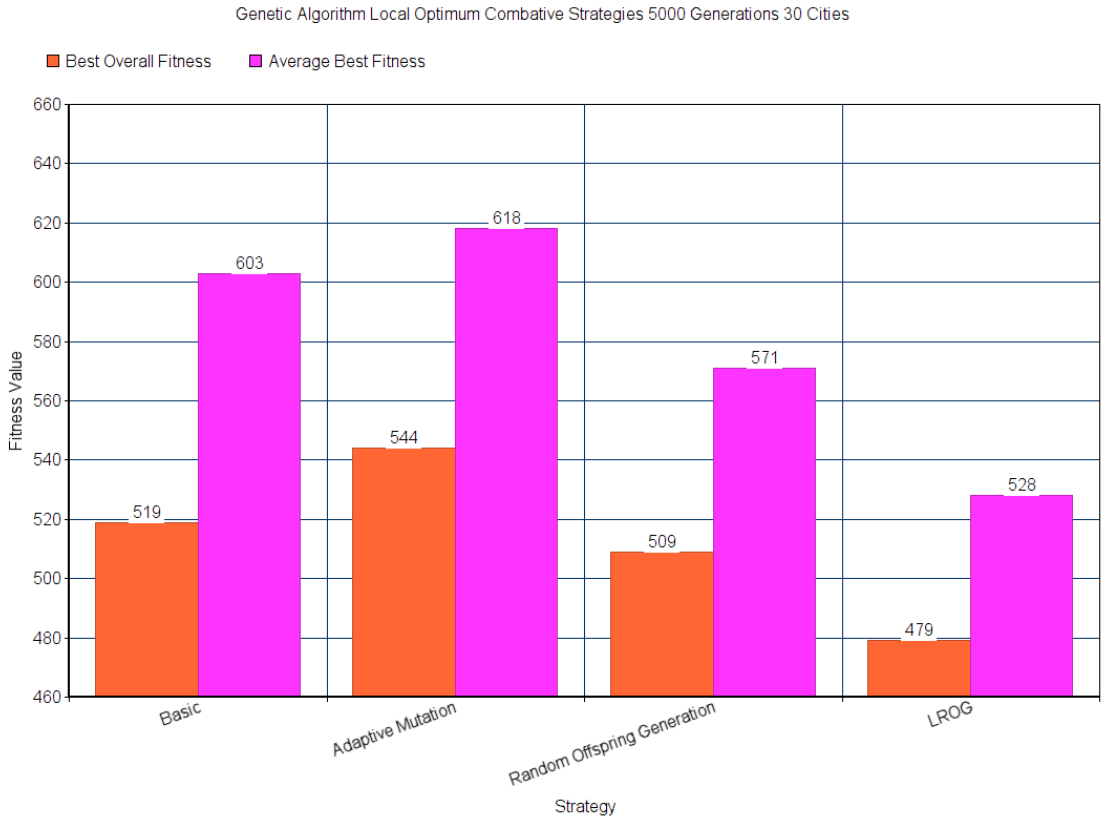

Figure 25 – Combative strategies, 5000 generations, 30 cities.

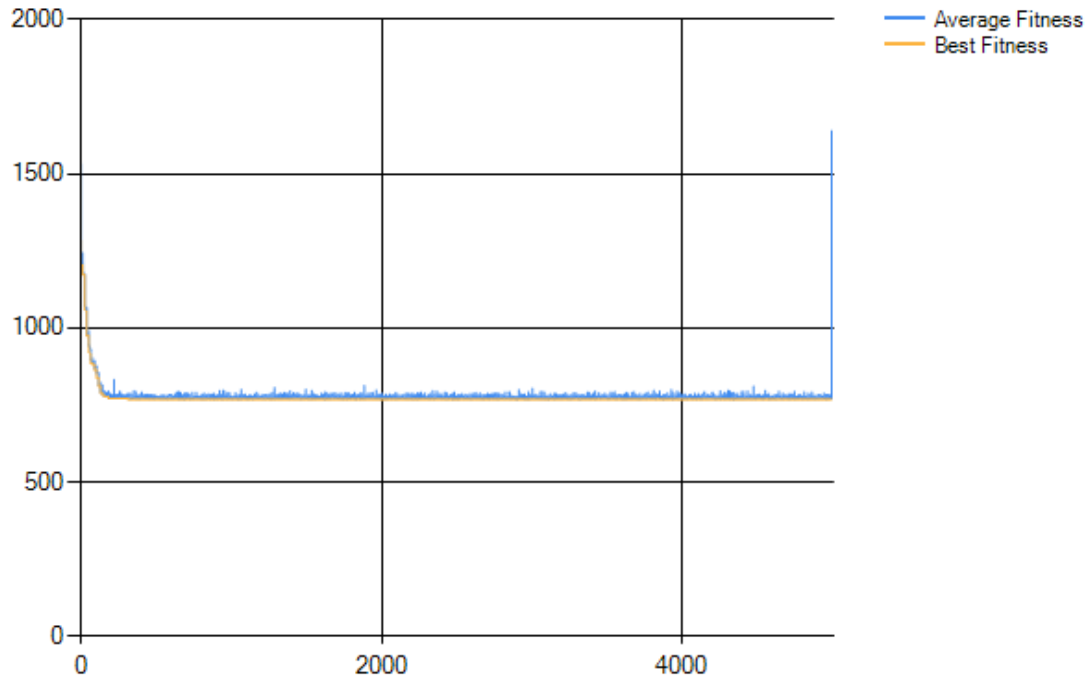

Figure 26 – Basic local optima example.

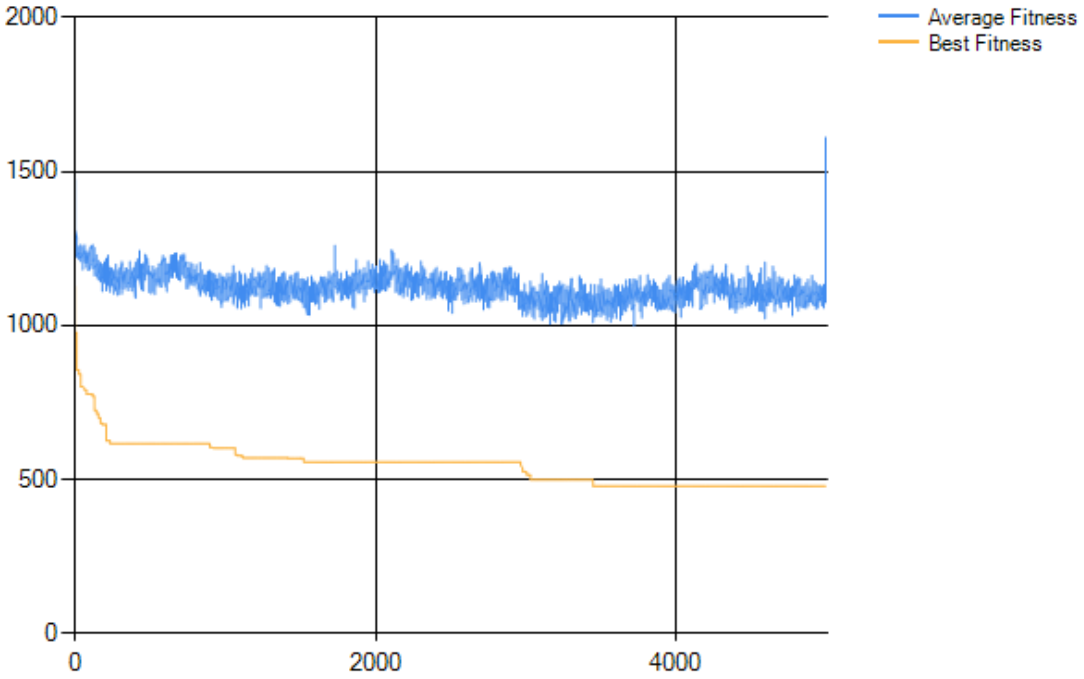

Figure 27 – LROG beating the local optimum example.