Fundamentals of Artificial Intelligence Assignment 2- Solving the Travelling Salesman Problem with Genetic Algorithms

Craig Heptinstall Crh13- 110005643
SEM6120
Institute of Computer Science
Aberystwyth University

Abstract—Genetic algorithms provide an optimization technique for a wide variety of problems, including that of the travelling salesman. The number of solutions for a TSP can become exponential, which genetic algorithms such as roulette, tournament and others can help reduce the number of evaluated solutions vastly.

I. INTRODUCTION

Genetic algorithms are considered in a wide array of problems where exact algorithms would struggle. The travelling salesman problem is a great example of a problem that does not have a means of finding the best solution first time, and would be increasingly time consuming to loop through all possible solutions to find the lowest cost (length of path).

A. Travelling Salesman Problem

To understand the travelling salesman problem, the reasons for its existence and early sources should be known. The travelling salesman problem was thought up in 1930 [?] by Merrill Flood whilst looking to solve a school bus routing problem. The problem stems off from the Hamiltonian cycle puzzle, requiring the solver to visit all node or cities once in a set of $\mathcal{N}$ cities. At this point in time, no general method of solution is known [?] (so is considered NP-hard) therefore only the best and shortest paths can be evaluated from several attempts.

Because of the nature of the problem, genetic algorithms appear as an idealistic means of finding optimised tours where a solution cannot be found using a mathematical means. Evolutionary algorithms used in genetics may never be guaranteed to find optimal solutions, though due to their random nature they will often find a good solution [?]. Genetic algorithms are called as so because of the similar way they act to natural selection, with the best and most efficient parents creating more advantageous children. In the instance of the travelling salesman, this could be reflected where two parents whose path distances are short are combined to form a child who uses similar paths to that of its parents, combined with a small form of randomness in the hope of improving upon a solution.
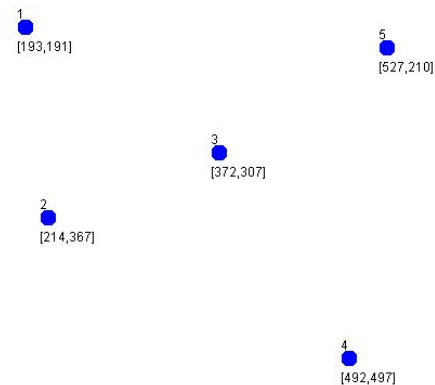

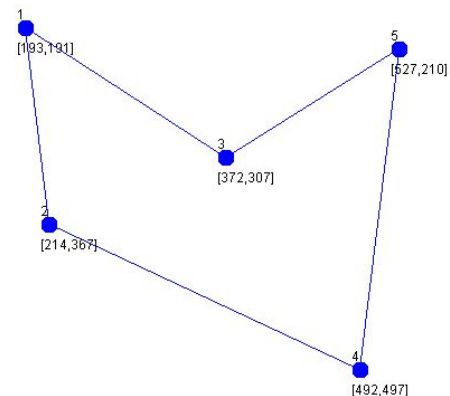Fig. 1. An example set of vertices where a travelling salesman problem is present.

Fig. 2. A possible solution to the TSP expressed above.

Figures **??** and **??** show an example TSP with a low number of points where an optimal solution can be found quite easily by exact, heuristic or GA means. An important aspect expressed in Figure **??** is that although all the points are visited, the final vertex visited must also be linked to the first in order to return to the starting position.

II. GENETIC ALGORITHMS CONSIDERED FOR TSP

Before looking at the possible genetic algorithms to be considered for such a puzzle as the TSP, it should be understood that a very basic generic algorithm can be used for such a purpose. In a paper in the Scientific World Journal [?], a team worked on using a traditional algorithm created back in 1989 by David Goldberg [?] and improving its efficiency to help solve the TSP quickly and as accurately as possible. The original genetic algorithm, which many algorithms are based from today goes through the following process:

1) Randomly generate an initial population of chromosomes.
2) Use the fitness function to select the fitter chromosomes.
3) Apply the crossover and mutation operators in order.
4) If a stopping criterion is satisfied, then stop and output the best chromosome
5) Go back to step 2

III. PROPOSED SOLUTION

Testing the capabilities of GA's using different means of selecting individual chromosomes (or tour solutions in the case of the travelling salesman problem) will allow statics and conclusions to be applied to the respective strategies, and have a higher chance of finding best possible results for the TSP problem. Alongside selection, the concept of elitism is looked at. Elitism may prevent strong solutions being lost on mutation and crossover, and therefore create better, faster-found routes.

A. Considered and selected GA representations

As already discussed in the previous section of the paper, most GA's have been based off of older ones such as chromosome representation [?]. For the purpose of this paper, the algorithm used for the TSP will be also based on this algorithm though different aspects such as crossover, mutation and elitism types will be looked at. To keep the algorithm at a simple chromosome state will allow easier implementation of the algorithm in code form, therefore will be easier to manipulate and change specific aspects to allow for statistical research.

There are three parts of the genetic algorithm that will be heavily considered in this paper, each of which will be modified and replaced by other means to compare and find the most optimal solutions to the travelling salesman problem. The three which are integral to any good GA, and which will be explained the need for in the TSP in the following sections are:

1) Selection- Selects two 'parents' from the current population, usually picking two strong chromosomes is ideal in order to ensure a strong 'child' is created. This like nature, as the strongest usually survive and again produce offspring which in turn makes the population stronger as a whole.

2) Crossover- The algorithm responsible for mating the two parents selected in the previous state of the GA. By mixing the two parents as efficiently as possible, two strong child chromosomes should be created using traits from both parents.

3) Mutation- Once the population has been updated with its new chromosomes formed from previous solutions, there should be some small amount of mutation to some of the the chromosomes. This allows the random chance that the solution could get better in some cases, therefore when selected and mated again the overall population may get better.

Each of these components of the GA to be used in the application will be described in the following section, each of which provided alongside how the application could implement them.

1) Selection methods: In order to create the most efficient child chromosomes (or path around TSP nodes in this case), there a range of available algorithms that can be used to select parent chromosomes. There are subcategories of selection strategies which are explained in some depth on Marek Obitko's 'Introduction to genetic algorithms' [?]. The first of which should be explored is the simple but effective roulette wheel selection technique. This works by imagining all the solutions are placed as roulette wheel numbers, with each solution having a proportion of the wheel decided by its fitness value.

A randomly generated number between zero and the total fitness of the population is then selected, and starting with the chromosomes with the higher fitness values, there fitness is subtracted from the random number until that number equals or is below zero. This process stops and then that chromosome is returned as a parent. This could be compared to the roulette ball stopping on a track.

By having the roulette wheel broken up by the size of fitness values, the larger the fitness value means the better chance of being selected. When a random number is selected, Figure ?? shows how some chromosomes could be placed onto a roulette wheel system.
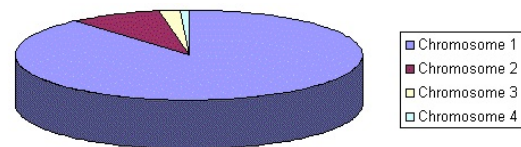

Fig. 3. A roulette wheel example showing a single chromosome with a far better fitness than the others.

To implement this into a TSP solver application, this will be done by simply looping through all of the paths, getting a total fitness and then looping again deducing each fitness value until zero is reached. This will be the simplest of the algorithms to implement and should be made to be built upon for other such algorithms like rank based ones.

The issue with roulette selection which is solved by the second selection method which will be used by a proposed

TSP solver application is that if there is a clear best chromosome with a fitness value that differs largely from the others then its chance of getting selected to be a parent may be two high, therefore children formed may always be from parents which are the same. This defeats the purpose of the crossover.

To solve this, a function called rank selection was created to keep in mind that although chromosomes with better fitness should have a higher chance of being selected, but they should have more of an exponential system to keep chances fair. Therefore, rank based system simply assigns the best fitness chromosome with a higher standard number, and this number is decreased as the rank of each chromosome is less. For example, with a population of 5, the best chromosome would be assigned the rank 5, and the next 4 and so on. This then means that although the roulette ball will still have the possibility of landing on any chromosome, the chances of landing on higher and lower fitness chromosomes will be biased more fairly. Figure **??** shows an example rank system in use with a set of chromosomes.
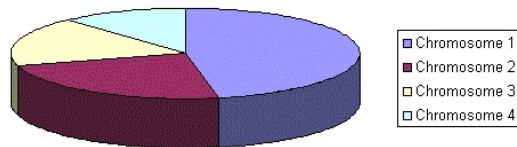

Fig. 4. An improved form of the roulette wheel with a fairer system of laying out chromosomes.

In the case of this TSP solver application, this could be done by firstly comparing each of the path lengths in the population, and then ordering them by lowest fitness first. Once this is complete, a fitness can be assigned to each starting with the population size, and decreasing the value of the fitness by one with each iteration. Once they are ranked, the roulette wheel method can use the edited population.

The third and final selection technique to be explored that will be tested in a TSP solver application will be tournament selection method. This uses a different mechanism to the previous two, by only evaluating a given number of solutions in a population rather than the entire population. Using the defined number of randomly selected chromosomes, the algorithm then gets the fitness of each and returns the winning chromosome. The tournament selection method ensures that weaker chromosomes do not make it through to the next evolution. In the case of the proposed application, this would be a simple implementation of a method which takes a population, a requested tournament size, and then uses a random number generator to select solutions before competing them together to return a winner.

2) Crossover methods: The reason for only selecting two crossover methods stems from the reason that although many crossover algorithms exist, many of them are inconsistent with a problem such as travelling salesman. For instance, take the crossover example in Figure **??**, after a simple one-point crossover, although the chromosomes have been mixed, the solutions in the children are no longer valid since not all the

points will be visited. A paper by Kylie Bryant which looks at both general genetic algorithms and how they can be applied to the travelling salesman problem mentions that algorithms should be made to make sure that children chromosomes do not finish as illegal solutions, and that 'in this way, crossover is very problem specific' [**?**].

The first to be considered is ordered crossover. The same paper explains this type of crossover well, explaining that this type of crossover is very similar to the PMX (partially mapped crossover) method, except instead of repairing chromosomes which contain repeat nodes, the rest of the nodes are rearranged to provide a legal path. In the application, this will be implemented by starting with a switch of part of two chromosomes between two crossover points. With this complete, then the child chromosomes will be constructed using genes from the start of the second crossover point, and these are to be added until the parent is looped over and reach the first crossover point. Finally the selected crossover genes will be placed into the child chromosomes in the position of the first crossover point.

The second crossover algorithm that will be implemented by the TSP solver application will be cyclic crossover. Good examples of the cyclic crossover method can be found on various online tutorials [**?**] and explained well on the apache commons library[**?**] with:

- Start with the first gene of parent 1
- Look at the gene at the same position of parent 2
- Go to the position with the same gene in parent 1
- Add this gene index to the cycle
- Repeat the steps 2-5 until we arrive at the starting gene of this cycle

3) Mutation methods: For the same reason that crossover options are limited due to the NP-Hard nature of the travelling salesman problem, only a short amount of available mutation types were available for consideration. For the purposes of this paper and the application constructed, only one mutation type will be used, swap mutation. Swap mutation is a very ideal solution to mutating points along a TSP because it allows routes to branch off and explore paths gradually further down the route as more optimum solutions are found. One author of an online article exploring a similar problem to this paper, states that using simple swap mutation makes the GA 'unable to achieve an optimum quickly but can prevent convergence into a local optimum' [**?**].

4) Evolution stop criteria: The final part of any GA and one that has been considered while planning the implementation the TSP solver is deciding when to stop the evolutions of chromosomes, or in other words stopping the looping of selection, crossover and mutation. In a chapter from the book 'Advances in Artificial Intelligence   SBIA 2004' [**?**], the three most commonly and widely used stopping criteria are described:

- An upper limit for number of generations is reached
- An upper limit on the number of evaluated fitness functions is reached

• The chance of significant improvement is reached

For the purpose of the research in this paper, the first will be implemented in order to allow more flexibility when testing and to allow the user to increase the runtime in order to compare the time required for each selection and crossover to hit a local minimum. This will also provide the ability to run more stressful testing to the algorithms such as increasing the number of cities to larger numbers.

B. Application structure

Following the completion of research into various implementation options for different parts of the TSP solver application, the structure of the application itself can be detailed. In order to do this, a high level class diagram, alongside a description of communication between classes has been provided in Figure **??**. The language chosen for this implementation was Java, for its extensive library collections and vast amount of tutorials similarly using the language for other genetic algorithm examples.
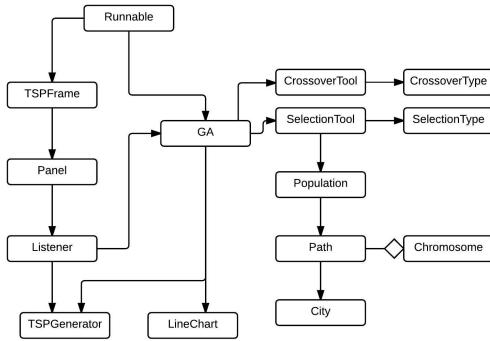

Fig. 5. High level class diagram of the TSP solver application.

As it can be seen in the diagram, there is a GUI in the design which is separated from the main logic of the application. This was done in order to allow a user wanting to run the application without the front end and opt for the option to use it as a console application only. The application has been made in an object orientated form as much as possible, such as creating objects for cities, paths, and types of crossover/ selection.

Starting by looking at key classes in Figure **??**, the main class 'Runnable' is responsible for triggering a run of the algorithm or trigger the opening of the GUI. If the user runs the jar using the 'java' command and passes correct arguments, then the output is printed to the command line alongside an export of the result csv file, and graphs from runtime. Otherwise the application frame is opened, and the user can perform actions through controls available.

To keep the GA class as simple as possible, a lot of the logic for performing the solving of TSP's was placed into helper classes such as selection and crossover 'tools'. Both these classes contain all the implemented methods for

crossovers and selections, and are selected via enumerated types for both respectably. All of the algorithms implemented can be found in the previous subsection of this paper, and all have graphical and command line options to use each.

The TSP generator class is especially important, making use of the Java Secure Random library to give less chance of duplicated points. This class has the ability to import and export paths to CSV and is used when the user generates points in the GUI (exporting points), or used when running the application through the command line (reading points in).

As for representations of the actual paths, it was ensured that both chromosomes (paths), and genes (cities) would be done in the most object orientated way possible. As shown in the class diagram in Figure **??**, a population is a collection of Paths, with each of these extending an abstract class of type chromosome. This has been done to show the relationship to genetics that the GA has. Within each path, a collection of cities (which represent genes) form the solutions to the given TSP.

For running the application, in the appendix under section **??**, a brief description is available. The same description can also be found in the readme file in the application source files.

C. Comparison of application results

There will be a range of different test set ups here, all of which will have graphs and statistics attached, to help compare and decide the best set up for the travelling salesman problem in terms of a genetic algorithm. Initial set ups (listed by selection type) will be described below, followed by tabled statistics of modifications of those set ups.

1) Rank based selection: Starting with a problem space of 12, the solution produced can be found in Figure **??**. Figure **??** shows the same setup, but with a doubled evolution amount, to 1000.

Fig. 6. Result using rank based selection, ordered crossover and an evolution amount of 500.
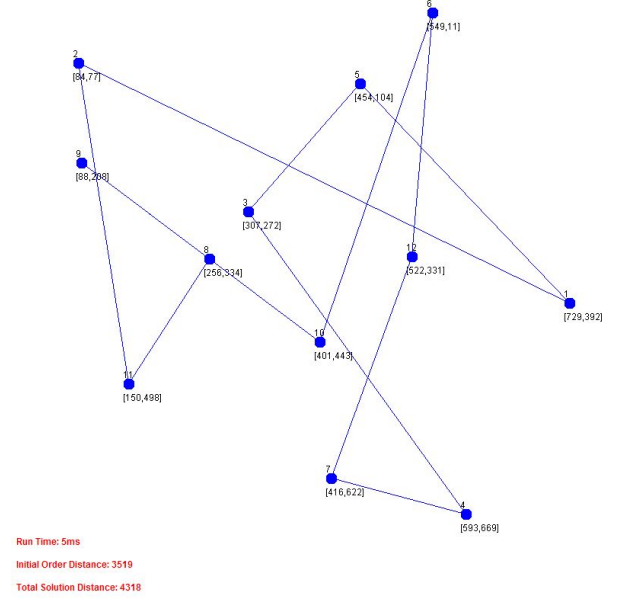
Fig. 8. Result using roulette based selection, ordered crossover and an evolution amount of 500.
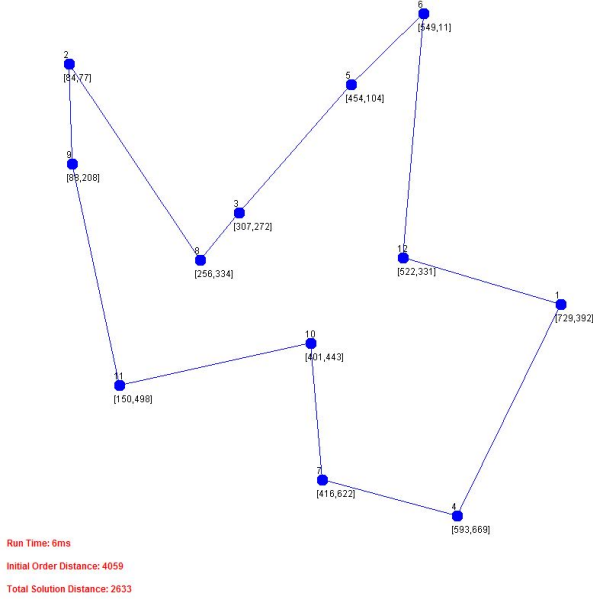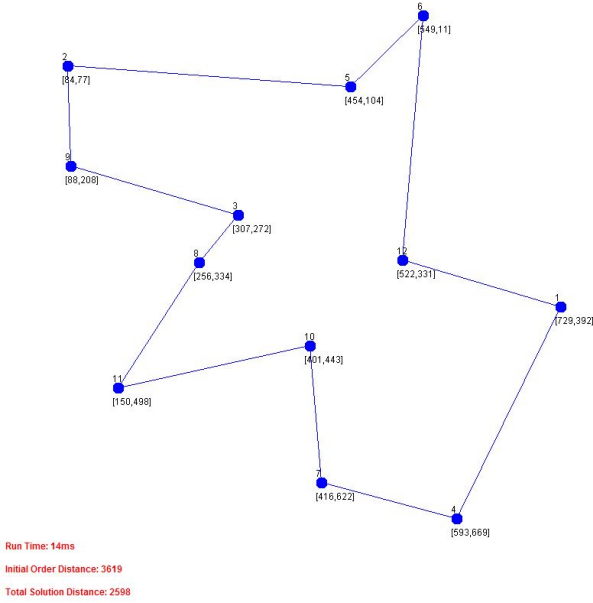

Fig. 7. Result using rank based selection, ordered crossover and an evolution amount of 1000.
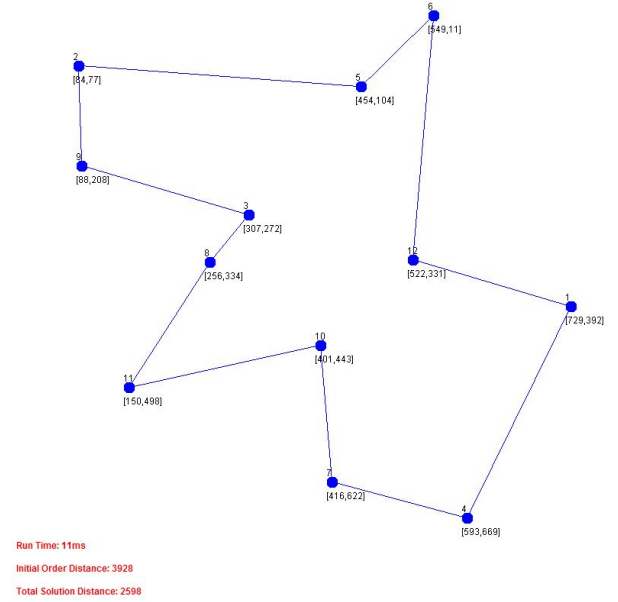
Fig. 9. Result using roulette based selection, ordered crossover, elitism enabled and an evolution amount of 1000.

2) Roulette based selection: With the same, fair amount of nodes, the solution produced can be found in Figure **??**. As with the previous run, Figure **??** shows an increase to 1000 evolutions. Elitism has also been enabled on the second run, to avoid the paths becoming completely random due to such a short amount of nodes.

3) Tournament based selection: Starting with a problem space of 12, the solution produced can be found in Figure **??**. Figure **??** shows the same setup, but with a doubled evolution amount, to 1000.

Fig. 10. Result using tournament based selection, ordered crossover and an evolution amount of 500.


Fig. 11. Result using tournament based selection, ordered crossover and an evolution amount of 500.

4) Comparisons of initial results: Following the completion of the first set of testing, Table **??** shows the summarised results using the listed selection types. From the results shows in the table, it can be seen that increasing evolutions decreases the distances in solutions, especially in the case of the roulette selection GA. Although overall runtime looks to exponentially increase, it seems necessary that future GA testing should use more evolutions to ensure that the final solution fitness hits a local minimum. What can also be seen in the figure previously is that roulette wheel requires elitism to be most effective, otherwise all routes change, resulting in the population to have a very slow evolution of fitness.

Selection Type	Num. of Evolutions	Solution Distance	Run Time (ms)
Rank	500	2643	6
Rank	1000	2598	14
Roulette	500	4318	5
Roulette	1000	2598	11
Tournament	500	2784	6
Tournament	1000	2633	12

The same tests were performed but with cyclic crossover, and are displayed in Table **??**. Comparing the two sets of runs, cyclic crossover has appeared to both produce slightly longer paths, and was performed at a slow run time. Similar to the previous runs, the rank based selection technique proved the best, getting slightly better results that roulette overall.

Selection Type	Num. of Evolutions	Solution Distance	Run Time (ms)
Rank	500	2809	7
Rank	1000	2772	16
Roulette	500	4174	5
Roulette	1000	2772	11
Tournament	500	2809	7
Tournament	1000	2786	14

In order to refine the best GA, the next few tests will only consider ordered crossover. There was also an increase in cities to 30 in the next set of experiments to both push the algorithms further, and to allow better statistics to be gained considering average and best fitness/ distance levels. Mutation rates will also be increased in the following tests, to see how changing the paths more effect the different selection types, and the genetic algorithm as a whole.

Test one using:

- Rank based selection
- Ordered crossover
- 1000 Evolutions
- Mutation of 0.25
- Elitism disabled

Shown in Figure **??**, the result of this test, showing that although the solution looks as if it is linking some correct cities together, crossed lines are still appearing showing signs that the solution could be better completed. The second graphic, Figure **??** shows the best distance chromosome (path) in the population after each evolution. Because of the increase in mutation rate, the best distance path in each evolution differs more than a lesser mutation rate. Overall though, the graph does eventually show a decrease in path distance.
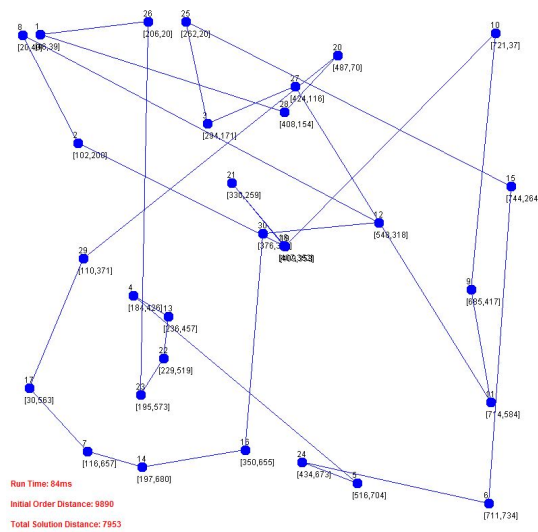
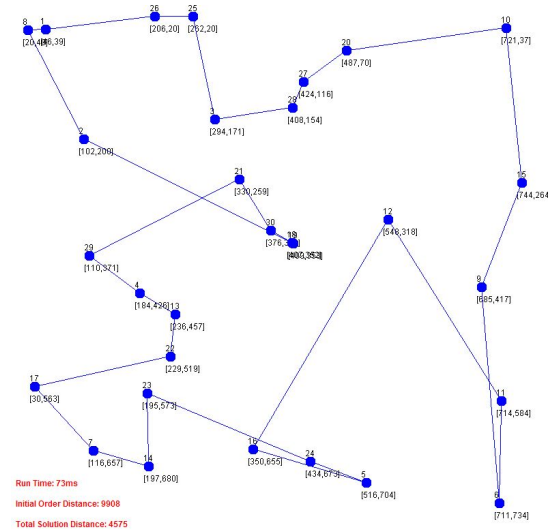Fig. 12. Solution path from running the GA with test one parameters.

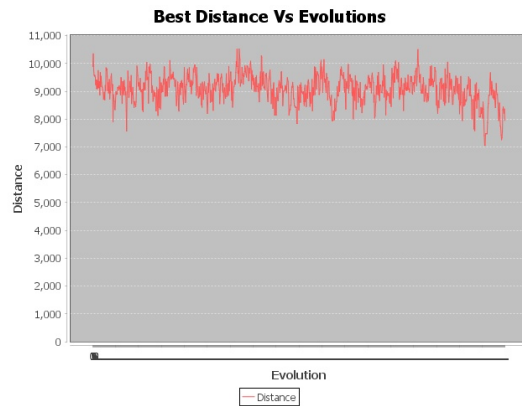Fig. 14. Solution path from running the GA with test two parameters.

Fig. 13. Graph produced showing best distances per evolution.
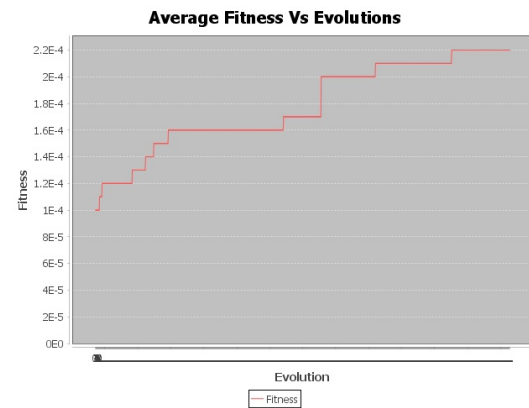

Fig. 15. Graph produced showing average fitness levels (inverted) per evolution.

Test two using:

- Roulette wheel based selection
- Ordered crossover
- 1000 Evolutions
- Mutation of 0.25
- Elitism enabled

Figure **??** shows the result of running roulette wheel selection, and shows clearly a better solution. Not only is the solution path shorter, but a reduction in crossed lines can be seen. Figure **??** shows that over the span of a 1000 evolutions, the fitness level gets better sharply before tapering off. Elitism was enabled for this test, otherwise the roulette wheel selection would always choose any random chromosome, therefore never really creating a steady decrease in distances of solutions.

Test three using:

- Tournament based selection
- Ordered crossover
- 1000 Evolutions
- Mutation of 0.25
- Elitism enabled

The final test looks at tournament selection, with elitism enabled. Without it, there would be too many random solutions used to create children chromosomes. In this test result as shown in figures **??** and **??**, not only was a better path selected, but the rate at which the fitness improved was much faster.
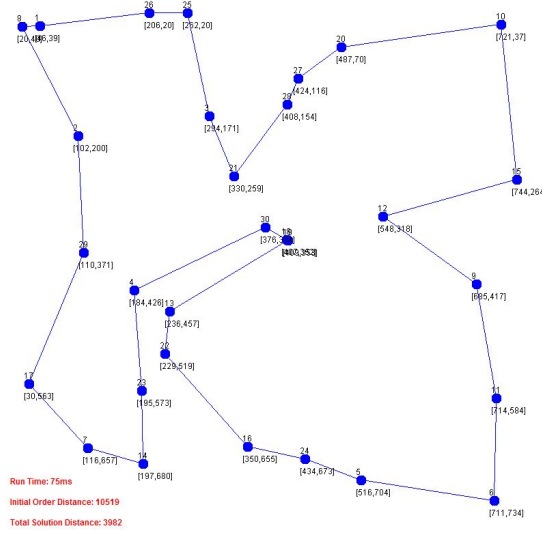
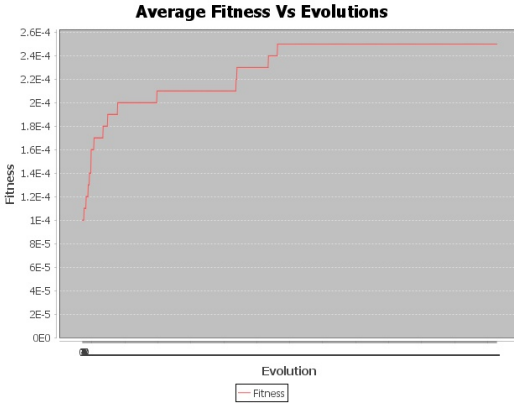Fig. 16. Solution path from running the GA with test three parameters.

Fig. 17. Graph produced showing average fitness levels (inverted) per evolution.

IV. OVERALL FINDINGS AND CONCLUSION

Starting with selection mechanisms, the initial testing phase showed that all three selection types had a similar ending solution when using a low level of nodes. The differences only started to show when more nodes were added, with both roulette wheel and tournament selection performing better. Roulette wheel however did operate faster than the other techniques, due to less logic being required to choose chromosomes. Unlike rank and tournament, all the chromosomes in a population did not have to be reordered before deciding which one to choose, therefore making it a quicker process.

In addition to this, both roulette wheel and tournament were found to require elitism in order to make them both as effective as possible. With too many random nodes selected, both fell in terms of performance greatly. Roulette however started stronger, but then tournament appeared to perform better after a certain amount of cities. This is also reflected in Figure **??**.
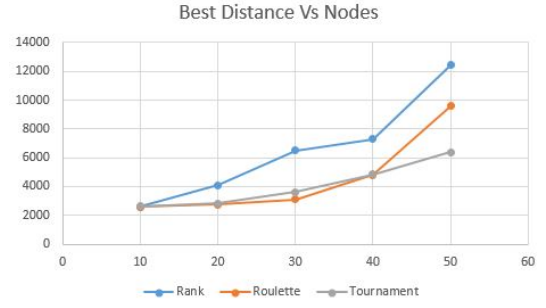

Fig. 18. Graph showing comparison of different selection types by amount of nodes(cities) in a path.

As for the crossover means, this was harder to test, but by comparing the speed at which both algorithms ran ordered crossover was found to be slightly quicker.

The mutation probability was also a parameter in the GA that was changed during testing, and tests showed that the higher (0.025) 25 in 1000 chromosomes to be modified helped selection techniques such as roulette wheel benefit. Of course on the other hand, rank based selection performed slightly worse due to it preferring as many higher ranked chromosomes as possible.

Using all the information gained, the suggested set ups for a TSP solver using the application options available described in this paper are as follows:

For problems with smaller amounts of cities (0-20):

- Roulette wheel selection
- Ordered crossover
- Less than 1000 Evolutions
- Mutation of 0.15
- Elitism enabled

For problems with larger amounts of cities (20+):

- Tournament selection
- Ordered crossover
- At least 1000 Evolutions
- Mutation of 0.25
- Elitism enabled

A. Improvements to proposed solution

In addition to the proposed solutions, there are a few considered improvements that may be possible with further testing and implemented selection/ crossover methods. Starting with selection methods, although three were implemented having more to test would allow further options to increase the capabilities of the GA. For crossover's, with more time this could be the case too. Many papers have been written on comparisons of crossover types, such as one from the International Journal of Latest Research in Science and Technology [**?**], which found the PMX crossover method (The only one out of the three mentioned in that paper that was not implemented in this application) to be a better solution. With more time,

perhaps trying out this crossover method could improve the effectiveness of the GA.

Future work would primarily focus on making the paths more efficient as a priority. By increasing the speed at which the GA operates, would allow more vigorous testing with a higher amount of cities to be tested with. The current state of the application allows the user to generate and run solutions for up to 100 cities. In the future it would be good to be able to handle a much larger amount of nodes, in order to put the GA to more practical and real life situations.

APPENDIX

Running the application

There are two simple ways of running the application, both through the runnable Jar file. Figure **??** shows the result of simply opening the Jar to open up the GUI. The other option is shown in Figure **??** by running the Jar through the command line to produce a command line response. The primary parameters for this are: name of csv file containing coordinates; a mutation rate; an evolution amount; whether elitism should be enabled; a selection type; and a crossover type. It should be noted that in order to run the command line version, the GUI version may need to be run if the user wishes to make use of the random point generation operation, which will only export to CSV whilst run in the GUI.

As with any good application, the TSP solver created is well commented, and Javadoc is available in the 'dist/Javadoc' folder. A mention should also be made to an online article on the project spot [**?**], which provided a good example on how to structure general GA's in the Java OO format.

REFERENCES

[1] E.L. Lawler, *The Traveling salesman problem: a guided tour of combinatorial optimization*, 1985

[2] E.W. Weisstein, *Hamiltonian Cycle*, *http://mathworld.wolfram.com/HamiltonianCycle.html*, 2015

[3] D.W. Dyer, *When are Evolutionary Algorithms Useful?*, *http://watchmaker.uncommons.org/manual/ch01s02.html*, 2008

[4] C.W. Tsai, S.P Tseng, M.C. Chiang, C.S. Yang, T.P. Hong, *A High-Performance Genetic Algorithm: Using Traveling Salesman Problem as a Case*, The Scientific World Journal Volume 2014, 2014

[5] D.E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley Longman Publishing Co., 1989.

[6] M. Obitko, *Introduction to Genetic Algorithms, Selection*, University of Applied Sciences. Czech Technical University., 1998.

[7] K. Bryant , *Genetic Algorithms and the Traveling Salesman Problem*, Department of Mathematics. Harvey Mudd college., 2000.

[8] B. Webster, *Crossover Technique: Cycle Crossover*, *http://www.rubicite.com/Tutorials/GeneticAlgorithms/CrossoverOperators/CycleCrossoverOperator.aspx*, 2015.

[9] Apache, *Apache Commons CycleCrossover Method*, *http://commons.apache.org/proper/commons-math/apidocs/org/apache/commons/math3/genetics/CycleCrossover.html*, 2015.

[10] K. Boukreev, *Genetic Algorithm and Traveling Salesman Problem*, *http://www.generation5.org/content/2001/tspapp.asp*, 2001.

[11] M. Safe , *On Stopping Criteria for Genetic Algorithms*, Advances in Artificial Intelligence SBIA 1004., 2004.

[12] Object Refinery Limited, *JFreeChart*, *http://www.jfree.org/jfreechart/download.html*, 2013.

[13] L. Jacobson, *Applying a genetic algorithm to the traveling salesman problem*, *http://www.theprojectspot.com/tutorial-post/applying-a-genetic-algorithm-to-the-travelling-salesman-problem/5*, 2012.

[14] N. Kumar, K. Karimbir, R Kumar, *A Comparative Analysis of PMX, CX and OX Crossover operators*, International Journal of Latest Research in Science and Technology Vol.1 Issue.2, 2012.
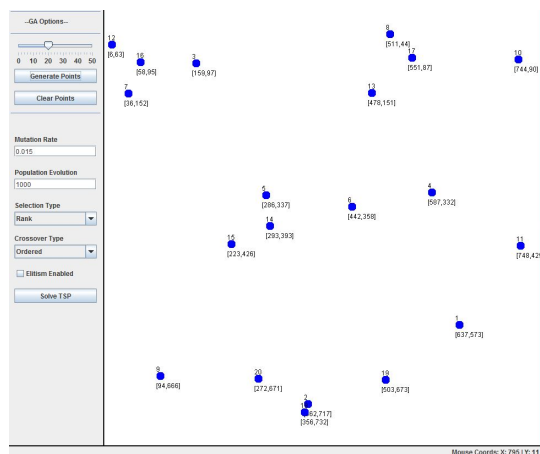
Fig. 19. The application running with it GUI.

Fig. 20. Example run of the console with example parameters. Output of ordered cities is placed into 'output.csv' file.