Investigating Genetic Algorithms as a Means of Solving TSPs

Connor Goddard

Department of Computer Science, Aberystwyth University
Aberystwyth, Ceredigion, SY23 3DB
Email: clg11@aber.ac.uk

I. Introduction

Owing to their ability to perform mathematical calculation with great speed and accuracy, it has become routine practice nowadays to utilise computerised algorithms as a means of identifying possible solutions within complex problem spaces.

Whilst computerised solutions have proven to be very effective within domains for which both the problem and an adequate solution can be *fully defined* (e.g. a system to count buttons on a jacket), for other types of problem where an exact solution may either be unavailable or undesired, traditional computer-based solution techniques can prove to be both ineffective and computationally-expensive to perform.

One limitation in particular that can arise from the use of a prescriptive approach [1], focusses on the bounding constraints applied to any potential solution. These bounds come as a consequence of a solution being defined by human developers, who themselves will hold particular knowledge, intuition and prejudice that could in fact hinder a computer application from discovering solutions that may be potentially better, but fall outside of the considerations held by those original developers [1].

A well-publicised example of a problem that can be particularly challenging for traditional computational approaches is that of the Travelling Salesman Problem (TSP). A conceptually-simple problem, the Travelling Salesman Problem describes a scenario in which a salesman is required to travel to an arbitrary number of cities situated at random locations across a map of a confined size. On his journey, the salesman should only ever visit an individual city once, before returning to his starting location. Presented with a collection of city locations (typically represented as X-Y coordinates), an acceptable solution describes a route that successfully connects all of the specified cities, such that the *overall distance travelled by the salesman along the route is minimised*.

Whilst for small collections of cities (approximately 10 or less) it would be possible for a human to identify an optimal solution relatively quickly, as the number of cities increases, the difficulty in identifying the optimal shortest path between them quickly becomes far too challenging [2]. For computer-based solutions, this idea is no different. Categorised as an 'NP-hard' problem, attempts to solve a TSP using exhaustive search techniques can quickly become unfeasible due to the exceptional amount of time required to find an optimal solution [1].

For optimisation problems like that of the TSP, an alternative computational approach that can demonstrate much better overall performance in identifying 'good' solutions is that of a *Genetic Algorithm* (GA). Forming part of a larger class of algorithms known collectively as Evolutionary Algorithms (EA), a GA represents a type of search function that uses approaches inspired by evolution and natural selection to identify effective solutions with (typically) a much reduced overhead than that of exhaustive approaches.

It should be noted, that unlike an exhaustive search approach, the use of a genetic algorithm can never guarantee an *optimum* solution, owing to the necessary element of randomness they exhibit in their behaviour. However, for the types of problem that these algorithms are generally attempting to solve, it is typically deemed acceptable to identify a solution classified as 'good enough', in exchange for providing such a solution in good time.

II. Design & Implementation of the TSP-GA Solver

A. Chromosome Representation

One of the first, and most crucial tasks in implementing a genetic algorithm is to decide on an appropriate representation for an individual solution candidate (otherwise known as a *Chromosome*). Typically, the decision on which representation to choose will relate directly back to the original problem the genetic algorithm is trying to solve. This is the case, given that representation selected must ensure that any solution proposed by that GA correctly conforms to the constraints set forward by the problem definition.

In contrast to other representation types such as Binary Encoding, Tree Encoding or Real-value Encoding [3], permutation encoding focusses on preserving the *sequence* of the individuals that forma complete solution candidate. Being fundamentally an ordering problem, solutions proposed for a TSP are expected to take the form of a list sequence, with every item in that list representing an individual city along the route to be taken by the salesman. By representing a sequence of individuals, permutation encoding provides a suitable representation for a given TSP route.

B. Fitness Function

In generating 'good' solutions, a genetic algorithm can only ever perform as well as the measure it uses to determine how well a given solution performs at solving the problem. In order to evaluate a proposed solution, one must define an appropriate *fitness function* that the GA can apply in order to establish, as a single unit of measurement, how close that particular solution comes to providing an acceptable answer.

In the context of the TSP - where the aim is to find the shortest possible route that connects all cities located upon a map - the overall measure of the performance for a proposed route falls to its *overall distance*. Given that each city is assigned an X-Y coordinate, it is possible to calculate the *2D straight-line distance* between two cities given by:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

before summing these distances across all of the cities along a given route in order to calculate the total distance.

Given that in practice a greater distance (d) represents a worse fitness score (f), the relationship between the two requires inverting prior to acquiring the fitness score for a given TSP solution candidate:

$$f = \frac{1}{d}$$

C. Selection Operators

One of the key driving forces toward identifying 'better' solutions in a GA, stems from the notion of preserving the characteristics of stronger solutions from previous generations by actively selecting these for breeding in the next generation. Whilst aiming to evolve the stronger candidates from the previous population is of course desired, it also remains important to introduce a certain level of random exploration in the evolution process in order to prevent getting trapped within only a very small subset of possible solution configurations.

With this in mind, three varieties of selection technique were implemented with the aim of comparing how differences in selection behaviour can affect the final solutions generated by a genetic algorithm.

1) Tournament Selection: The first of the three selection methods chosen for this investigation was Tournament Selection. This approach follows the idea of conducting a 'tournament' between a collection of randomly-selected solution candidates (acquired from the previous population), to return the individual that holds the highest fitness score.

This selection technique favours from being simple to implement, and is applicable for use on parallelised systems. By altering the number of solution candidates added into the tournament pool, it becomes possible to adjust the "selective pressure" [1] of the selection process, which in turn can either increase or decrease the chances of selecting and (by guarantee) returning the very strongest solutions from the previous population.

2) Roulette Wheel Selection: Roulette Wheel Selection (RWS) uses a probability *directly proportional* to the fitness score of a given solution candidate as the means of selecting the next parent for breeding. Whilst in theory all solution candidates have the potential to be selected, those with a higher fitness score sport a proportionally greater chance of being selected than those with a lower fitness score.

To simulate this selection approach, we can implement the following algorithm [4]:

Algorithm 1 RWS - Implementation

```
populationFitnessSum := 0

for i := 0 to population.length do
    populationFitnessSum += population[i].fitnessScore

randomPoint := random double between 0 and population-
    FitnessSum

currentSelectedCandidate := population[0]

partialSum := 0.0

i := 0

while (i <population.length  partialSum <randomPoint)
    do
    currentSelectedCandidate = population[i]
    partialSum += population[i].fitnessScore
    i = i + 1

return currentSelectedCandidate
```

Like Tournament Selection, this technique is relatively simple to conceptualise, which can make it a good starting choice for developers who have little experience of implementing genetic algorithms. Depending on the tournament size specified for Tournament Selection, RWS typically presents a greater chance of the same solution candidate being selected multiple times. Whilst this represents perfectly acceptable behaviour in terms of the GA (remembering that ideally the aim is for the majority of future offspring to be based upon the stronger set of parents), if the difference in fitness score between the best and worst candidates for the previous population is too great, this can result in the stronger chromosomes dominating the selection process, which in turn can lead to a poor level of variance and hence little room for random exploration.

3) Stochastic Universal Sampling: Holding much in common with Roulette Wheel Selection, Stochastic Universal Sampling (SUS) [5] aims to address the potential issue relating to the lack of variance sometimes found in RWS, by adopting a different style of selection approach that guarantees an individual candidate will only ever be selected the number of times directly proportional to its corresponding segment upon the roulette wheel [1].

The main difference between SUS and RWS relates back to how the sampling positions along the theoretical wheel are calculated. In RWS, the wheel is spun *every time* a solution candidate is to be selected. A *single pointer* is then used to determine the location along the wheel to select from, which in turn maps onto a particular solution candidate.

In the case of SUS, the wheel is spun *once and only once* for that *entire generation*. Rather than using only a single pointer to sample selection positions, SUS makes use of *multiple pointers* (equal to the number of selections required) that are positioned at *equally-spaced intervals* across the circumference of the wheel. When the time comes for the next candidate is to be selected, the process advances to the *next pointer* and returns the solution candidate at that position. [1].

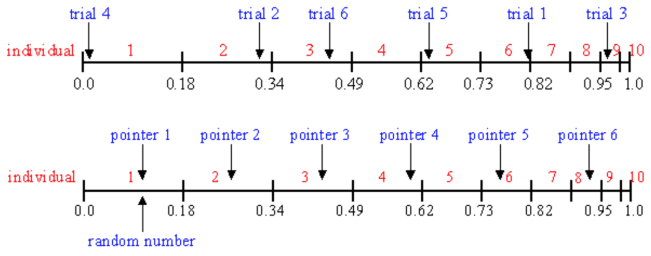


Fig. 1: Graphical depiction representing the difference in sampling technique demonstrated between RWS (top) and SUS (bottom) selection operators. [6]

Of course, as the quality of solutions becomes poorer, so to do the chances of those individuals being selected. Such solutions will occupy only very small segments of the wheel, and so run a high chance of being completely 'stepped over' by the sampling process. [1]. Figure 1 highlights the difference in sampling observed between RWS and SUS.

D. Crossover Operators

Like that of the fitness function, crossover operators must work within the bounds of the problem constraints in order to ensure that only valid solutions are produced at the end of the breeding process. In the case of the TSP, one such constraint dictates that a salesman *should only ever visit an individual city once along their route interconnecting all of the cities specified in the problem definition*.

This constraint can present an issue for a number of standard crossover techniques such as Single-Point, Two-Point and Uniform crossover, which use varying techniques to randomly select and combine individual solution 'genes' (cities in the case of TSP) from both parent solution candidates (chromosomes). The problem with adopting random selection and recombination alone as an approach, is that it fails to address the possibility of repeating genes appearing within the child solution candidates, which as we have established, would represent an invalid TSP solution.

The answer therefore was to select crossover operators that remain capable of recombining structural aspects from both parents, whilst also guaranteeing not to introduce any duplicates.

1) Order-One Crossover: Order-one Crossover (OX1) represents a conceptually-simple permutation crossover technique, in which a randomly-sized subset of individuals is copied over from Parent 1 to the new child, before the remaining values required to complete the child are copied over from Parent 2 *in the order in which they appear within the second parent* [7].

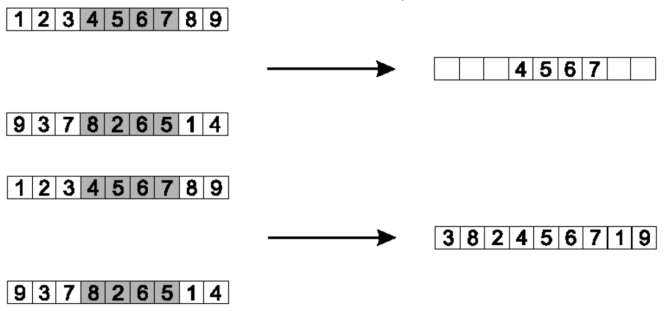


Fig. 2: Graphical example of generating a child chromosome using Order-One crossover. [8]

Figure 2 presents an example of this process, where a random subset is copied down from Parent 1, before any missing individuals are added from Parent 2 in the same sequential order.

2) Cycle Crossover: In contrast to OX1 crossover which works by copying whole blocks of parent individuals over to the children, Cycle Crossover "identifies a number of so-called cycles between the two parent chromosomes", before alternating between the individual items extracted from both Parent 1 and Parent 2 in order to generate new child solution candidates [9].

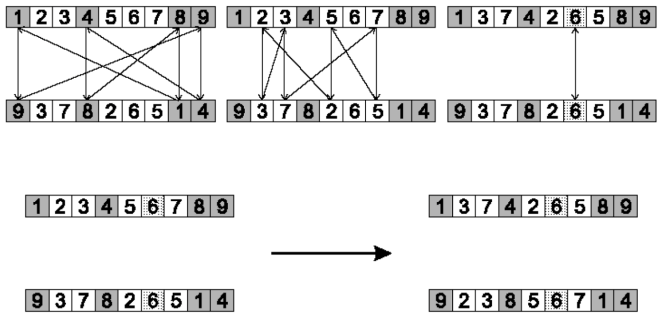


Fig. 3: Graphical example of generating a child chromosome using Cycle crossover. [8]

In establishing a 'cycle', the process identifies relationship mappings between the *position* of matching individuals within the parent candidates, which in turn prevent duplicates from appearing by restricting each individual to form part of only a single cycle.

E. Mutation Operators

As this investigation decided to focus attention on other types of GA operators and parameters in its experiments, only a single mutation operator was implemented. In the same way as the crossover operators, the mutation operator chosen for the final implementation had to guarantee against introducing duplicates within the modified solution candidates.

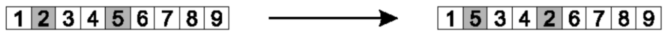


Fig. 4: Graphical example of child chromosome mutation using Swap Mutation. [8]

The mutation operator eventually selected was that of Swap Mutation. Proving easy to understand and implement, swap mutation simply selects two individuals from the target solution candidate and exchanges their positions. As the candidate in question is already known to not contain any duplicates (as guaranteed by the crossover operator in the previous stage), this process requires little computational overhead and thereby performs well over a large number of generations. Figure 4 shows an example of this process within a solution candidate.

F. Application Structure & Programming Language

Considering that the underlying operators and processes for a genetic algorithm remain independent of the specific problem it is applied to, it was identified that an inheritance-based application structure would provide the best basis from which to implement a genetic algorithm in computer code. The decision was taken to implement the *TSP-GA Router* application using the Java programming language owing to its wide community support, extensive collection of data structure and utility libraries, and (perhaps most importantly) the author's existing familiarity and experience with the language.

To enable development efforts to focus upon the implementation of the genetic algorithm and its operators, a selection of 3rd party Java libraries were utilised within the final *TSP-GA-Router* application. These were as follows:

- GRAL Java Graphing Library [10] - Provided graphical plotting capabilities.
- Apache Commons CLI [11] - Provided common CLI argument parsing facilities.
- Apache Commons IO [12] - Provided common file creation and directory management facilities.
- Apache Commons Lang [13] - Provided facilities relating to the custom parsing of Date/Time properties.

Describing the general structure of the application, a collection of 'GA'-prefixed abstract classes each represent a core common component of the genetic algorithm; including instances of an individual gene ('GAGene'), a chromosome ('GAChromosome') and population ('GAPopulation').

By deliberately defining these as abstract classes, it became possible to pre-define and implement functionality common to all instances of a genetic algorithm, whilst at the same time forcibly delegating all problem-specific functionality to be implemented within custom class definitions inheriting from those abstract classes. Examples of these problem-specific class definitions are represented in the *TSP-GA Router* application by classes prefixed with the 'TSP' notation.

A complete UML class diagram providing detail on the application design is provided in Appendix A (Figure 10).

III. Experimental Setup

All of the experiments in this investigation were performed under automated test conditions, with the specific parameters for each individual experiment provided via a pre-prepared XML configuration file.

Following the completion of each experiment, the results were automatically exported to a new folder location created specifically for that individual experiment. These results consisted of the following items:

- Graphical representation describing the state of the best TSP solution acquired following the initialisation of the first GA population.
- Graphical representation describing the state of the best TSP solution acquired following the completion of the GA evolution process (i.e. representing the best overall solution to be found in that individual experiment)
- Text file containing details relating to the input parameters for the experiment, and result information including total elapsed time for the GA run and the fitness score for the first and last instances of the current best solution candidate.
- Graph plot indicating the trend for the average, best and worst fitness scores for the current population across the full generational lifespan of the GA.

IV. Results & Discussion

A. Experiment 1: Crossover Operator Performance

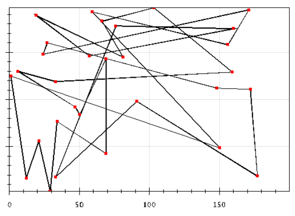
To ensure a fair comparison between the performance of different crossover operators, all tests performed as part of experiment one were conducted using the same fixed parameters. Solving a TSP consisting of 30 locations, these parameters specified: a population size of 100; a crossover rate of 75%; a mutation rate of 1.5%, with swap mutation as the mutation operator; a tournament size of 10; and elitism set to automatically copy over the two best parents from the previous population. All tests were run over 500 generations, with each being repeated 100 times. The results presented represent the averages calculated across all of the test runs.

TABLE I: Average results over 100 test runs for Experiment 1.

Selection Type	**Initial Fitness**	**Final Fitness**	**Duration (Milliseconds)**
Tournament	2817	1247	299
SUS	2558	2121	138
RWS	2883	2119	117

From observing the results indicated in Figure 6, it becomes clear that Tournament selection consistently outperforms both the SUS and RWS selection operators. Moving toward a final solution close to that of the optimal, Tournament selection shows much less of an observed difference between the average fitness and the best fitness for a population. This reduced range in fitness variance compared to RWS and SUS is not of great surprise, given that with a tournament size set to

10% of the population size, it would be expected that at least a couple of representatives originating from the stronger subset of solution candidates would make it into the tournament pool. Owing to its behaviour of always selecting the strongest candidate out of the acquired tournament pool, Tournament selection therefore provides a comparatively weaker chance of poorer, but potentially more diverse solution candidates being brought forward for crossover.


(a) Initial Solution State

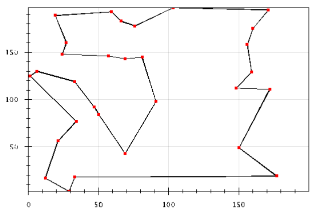

(b) Final Solution State

Fig. 5: Graphical examples of start and final solutions generated by GA using Tournament selection.

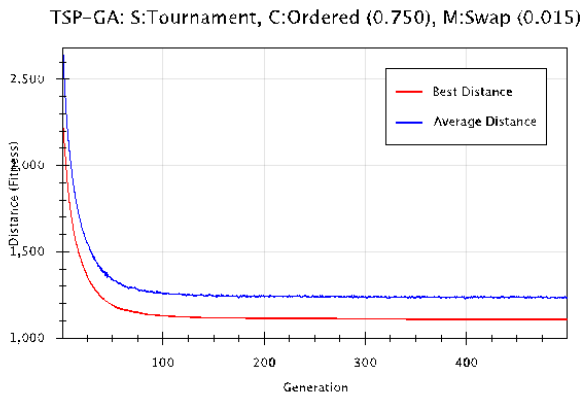
While SUS does see a reduction in the overall route distance, it takes a much more gradual approach to reducing the overall solution fitness by any notable degree, although it is important to recognise that this reduction does remain consist across the entire length of the GA. One could expect that given longer to complete, this approach may have been eventually reached the same level of fitness improvement as that observed from Tournament Selection.

Unlike RWS, SUS guarantees that the stronger solutions will on average be selected more times than those of weaker fitness. It would therefore be expected that on average RWS would display a greater level of random exploration than that seen of SUS. This behaviour is observed in Figure 6c, where as a probable consequence of both too much random exploration, and too little preservation of stronger solutions (despite elitism being in use), the GA using RWS fails to ever arrive towards any form of stable convergence.

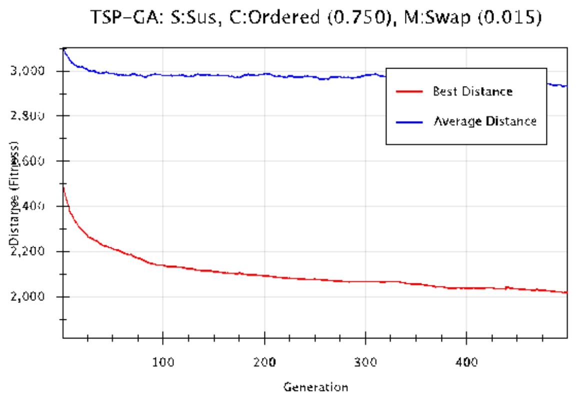
B. Experiment 2: Crossover Operator Performance

Solving a TSP consisting of 30 locations, this experiment compared the performance of the two implemented crossover operators over all three selection operators (incorporating elitism) using the following fixed parameters: a population size of 100; a crossover rate of 75%; and a mutation rate of 1.5%, with swap mutation as the mutation operator. All tests were run over 500 generations, with each being repeated 100 times. The results presented represent the averages calculated across all of the test runs.

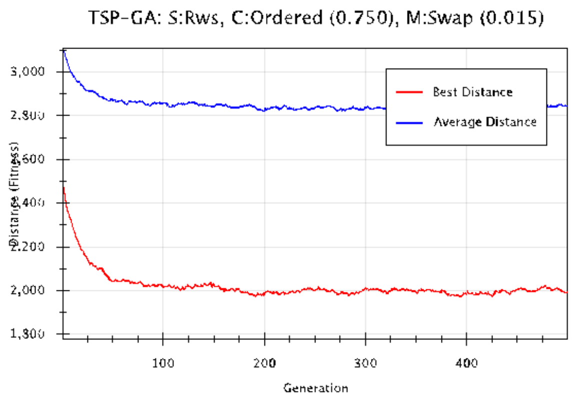
In the case of the Tournament selection, the observed difference in performance between that of ordered crossover (Figure 7a) and cycle crossover (Figure 8a) appears to be minimal. This is most likely due to the fact that as the selection operator actively selects only the very best solution from a typically healthy collection of tournament candidates, the differences in generation technique between the two crossover operators do little to alter the chances of finding and exploiting the generally stronger candidates in future generations.



(a) Best and average TSP route distances (averaged over 100 runs) using Tournament selection.



(b) Best and average TSP route distances (averaged over 100 runs) using SUS selection.



(c) Best and average TSP route distances (averaged over 100 runs) using RWS selection.

Fig. 6: Graphical examples of start and final solutions generated by GA using Tournament selection.

For the SUS and in particular RWS operators, the performance increase observed following the switch from ordered to cycle crossover is notable, subsequently causing both instances (Figures 8c and 8b respectively) to reach the kind of

TABLE II: Average results over 100 test runs for Experiment 2.

Selection Type	Crossover Type	Initial Fitness	Final Fitness	Duration (Milliseconds)
Tournament (Elitism)	Ordered	2559	1164	267
SUS (Elitism)	Ordered	2577	2122	111
RWS (Elitism)	Ordered	2561	2028	115
Tournament (Elitism)	Cycle	2558	1174	172
SUS (Elitism)	Cycle	2560	1849	162
RWS (Elitism)	Cycle	2557	1512	192

near-optimal solutions only previously demonstrated by that of Tournament selection. It is suspected that the behaviour observed from cycle crossover, which unlike ordered crossover chooses not to copy whole swathes of individuals from a parent down to the new child, is likely to result in a generally greater level of solution diversity that subsequently cures SUS of becoming trapped within a local minima, and RWS of failing to track and exploit stronger solutions.

It should be noted these thoughts come with a significant level of trepidation attached, given that intuitively one would expect that ordered crossover would stand a better chance of preserving those aspects of a parent solution that enables it to be deemed a 'good solution'. Of course, it may also be argued that this same characteristic could equally cause ordered crossover to preserve the poorer characteristics of a given parent solution, in which case the original notion as to the increase in performance seen by cycle crossover would stand up.

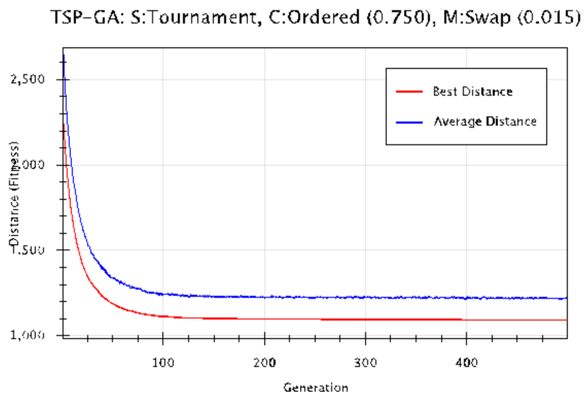
C. Experiment 3: Mutation Rate Performance

This experiment compared the performance observed under a selection of three pre-determined mutation rate settings: a typical rate of 1.5%, a reasonably high rate of 10% and a very high rate of 25%. In order to focus the comparison of results, the best performing selection operator (Tournament) in conjunction with the best performing crossover operator (Ordered) were selected for use with the following fixed parameters: a population size of 100; a crossover rate of 75%; and swap mutation as the mutation operator. All tests were run over 500 generations, with each being repeated 100 times. The results presented represent the averages calculated across all of the test runs.

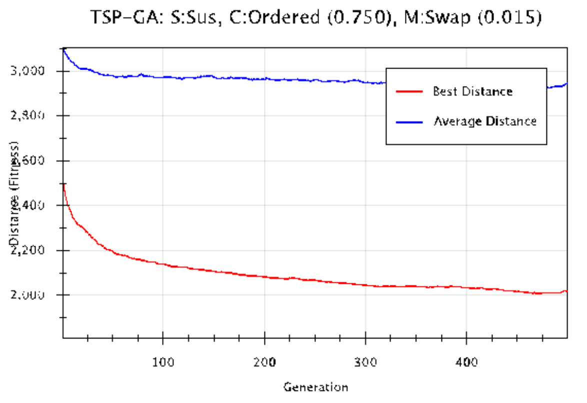
TABLE III: Average results over 100 test runs for Experiment 3.

Mutation Rate	Initial Fitness	Final Fitness	Duration (Milliseconds)
1.5% (0.015)	2563	1142	286
10% (0.1)	2558	1998	137
25% (0.25)	2563	2399	115

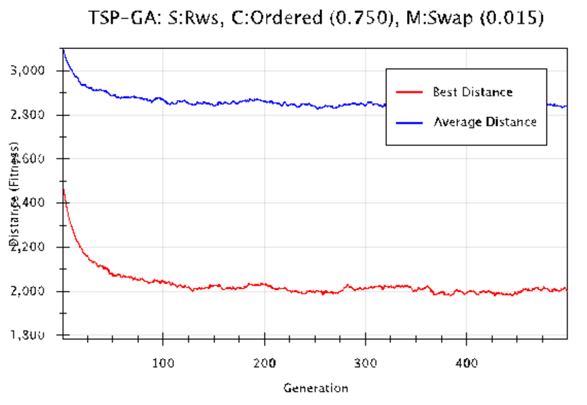
The results depicted in Table III and Figure 9 highlight the behaviour that would typically be expected given the values



(a) Best and average TSP route distances (averaged over 100 runs) using Ordered crossover with Tournament selection.



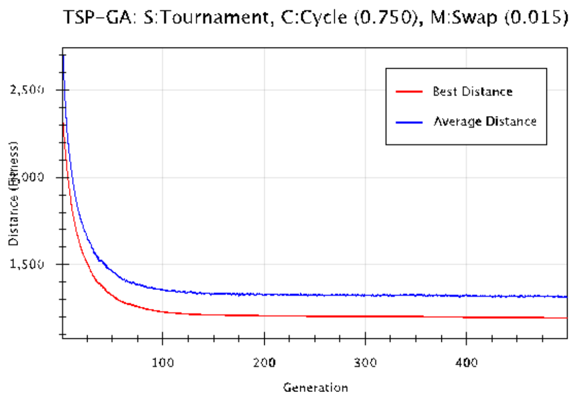
(b) Best and average TSP route distances (averaged over 100 runs) using Ordered crossover with SUS selection.



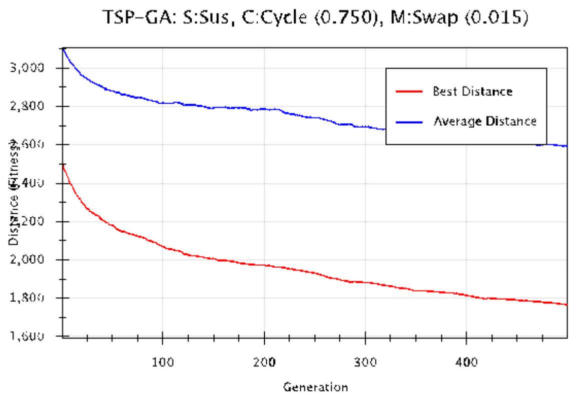
(c) Best and average TSP route distances (averaged over 100 runs) using Ordered crossover with RWS selection.

Fig. 7: Graphical examples of start and final solutions generated by GA using Tournament selection.

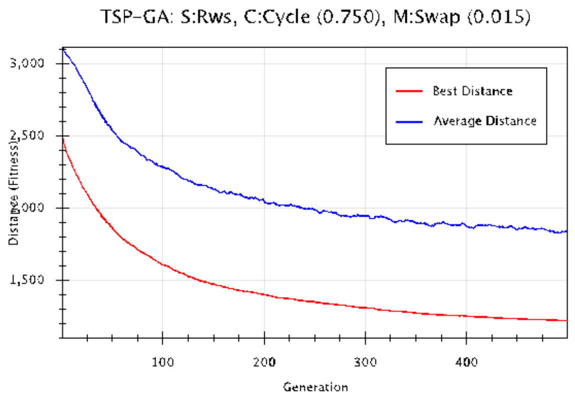
provided for the mutation rate. Generally suggested as a useful starting value from which to begin testing [14], a mutation rate of 1.5% combined with Tournament selection and Ordered crossover provides a good balance between solution *exploitation* and *exploration*, with a near-optimal solution being found very soon into the evolution process.



(a) Best and average TSP route distances (averaged over 100 runs) using Cycle crossover with Tournament selection.

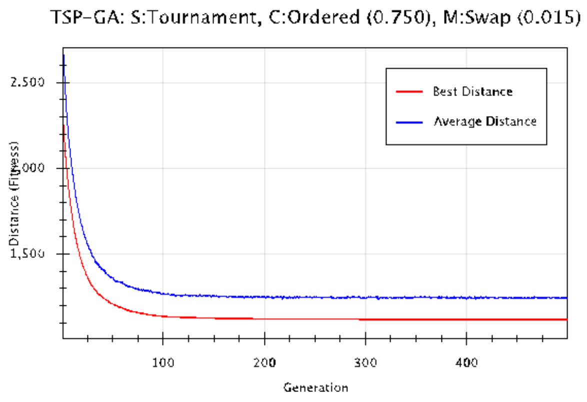


(b) Best and average TSP route distances (averaged over 100 runs) using Cycle crossover with SUS selection.

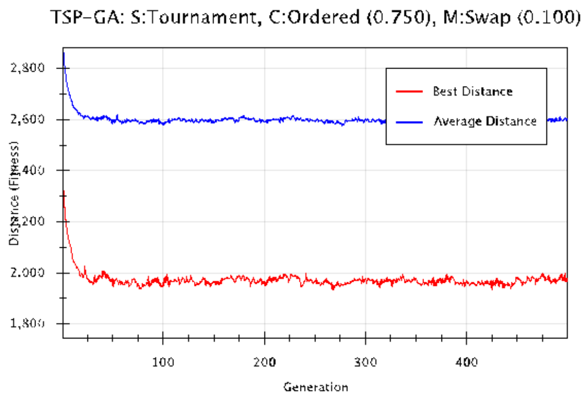


(c) Best and average TSP route distances (averaged over 100 runs) using Cycle crossover with RWS selection.

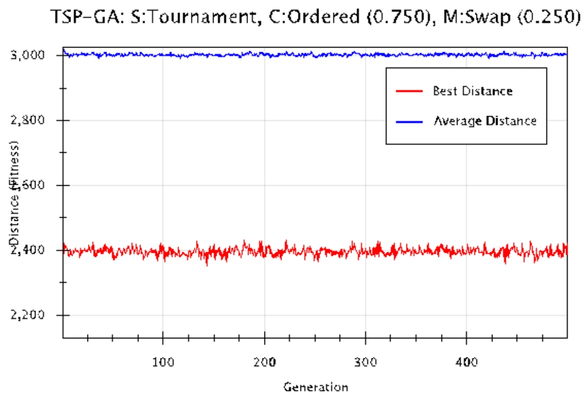
Fig. 8: Graphical examples of start and final solutions generated by GA using Tournament selection.



(a) Best and average TSP route distances (averaged over 100 runs) using 1.5% mutation rate.



(b) Best and average TSP route distances (averaged over 100 runs) using 10% mutation rate.



(c) Best and average TSP route distances (averaged over 100 runs) using 25% mutation rate.

Fig. 9: Graphical examples of start and final solutions generated by GA using Tournament selection.

Unsurprisingly, increasing the mutation rate to a factor of 10 and above beyond 1.5%, results in a significant reduction in the amount of information that is preserved from the stronger parents caused by an unsatisfactory level of randomness being re-introduced back into the population per generation cycle. Consequently, this prevents the GA from identifying strong leads on better solutions, thereby reducing the overall quality of the final solution.

V. Conclusions & Future Work

Using a well-configured genetic algorithm, this investigation has found that for a small to medium-sized Travelling

Salesman Problem, it is possible to provide an acceptable (and in most cases near-optimal) solution within a generally respectable degree of time. This is owed at least in part to the choice of genetic operators including Ordered Crossover and Swap Mutation which both uphold the specific constraints presented by the TSP whilst also being computationally efficient to perform.

From analysing the results, this report concludes that a combination of Tournament Selection with Ordered Crossover demonstrated the best overall performance for a TSP consisting of 30 nodes, however, following the discovery of comparable results for both Roulette Wheel Selection and Stochastic Universal Selection after switching over to Cycle Crossover in Experiment 2, one must stress the importance of making sensible and appropriate choices regarding the configuration of a GA in order to ensure the very best performance is extracted.

Owing to time constraints, it was unfortunately not possible to perform further experimentation on alternative order-based mutation operators such as Inversion Mutation and Scramble Mutation [15]. Given more time, it would also be desirable to conduct a fuller investigation looking into grounds for a relationship between a lack of selection variance, and the poor performance demonstrated by both of the proportionate-based selection operators.

Additional future work could also focus on analysing the performance of the various operators when applied to much larger TSPs ($>$100 locations). Given the current results, one would predict that Tournament selection would be the first to find an acceptable solution, however owing to their greater acceptance for lower-scoring solutions, it would fall to RWS, or more likely SUS to find the closest solution to the optimum, although in all likelihood taking longer to do so.

References

[1] D. W. Dyer, *Evolutionary Computation in Java - A Practical Guide to the Watchmaker Framework*, http://watchmaker.uncommons.org/manual/, 2010.

[2] J. N. MacGregor and T. Ormerod, "Human performance on the traveling salesman problem," *Perception & Psychophysics*, vol. 58, no. 4, pp. 527–539, 1996.

[3] M. Obitko, "Introduction to Genetic Algorithms - Encoding," 1998. [Online]. Available: http://www.obitko.com/tutorials/genetic-algorithms/encoding.php

[4] ——, "Introduction to Genetic Algorithms - Selection," 1998.

[5] J. E. Baker, "Reducing Bias and Inefficiency in the Selection Algorithm," in *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*. Hillsdale, NJ, USA: L. Erlbaum Associates Inc., 1987, pp. 14–21. [Online]. Available: http://dl.acm.org/citation.cfm?id=42512.42515

[6] H. Pohlheim, "Selection - GEATbx: Genetic and Evolutionary Algorithm Toolbox for use with MATLAB," Jul. 1997.

[7] B. Webster, "Order 1 Crossover," 2010.

[8] N. Sadawi, "An Introduction to Genetic Algorithms," Jan. 2015.

[9] B. Webster, "Cycle Crossover Operator," http://www.rubicite.com/Tutorials/GeneticAlgorithms/CrossoverOperators/CycleCrossoverOperator.aspx, Rubicite Interactive, 2010.

[10] D. Erichseifert, "GRAL - Java Graphing Library," http://trac.erichseifert.de/gral/wiki, 2014.

[11] "Apache Commons CLI," https://commons.apache.org/proper/commons-cli/, The Apache Software Foundation, 2015.

[12] "Apache Commons IO," https://commons.apache.org/proper/commons-io/, The Apache Software Foundation, 2015.

[13] "Apache Commons Lang," https://commons.apache.org/proper/commons-lang/, The Apache Software Foundation, 2015.

[14] M. Obitko, "Introduction to Genetic Algorithms - Recommendations," http://www.obitko.com/tutorials/genetic-algorithms/recommendations.php, 1998.

[15] A. E. Eiben and J. E. Smith, *Introduction to evolutionary computing*. Springer Science & Business Media, 2003.

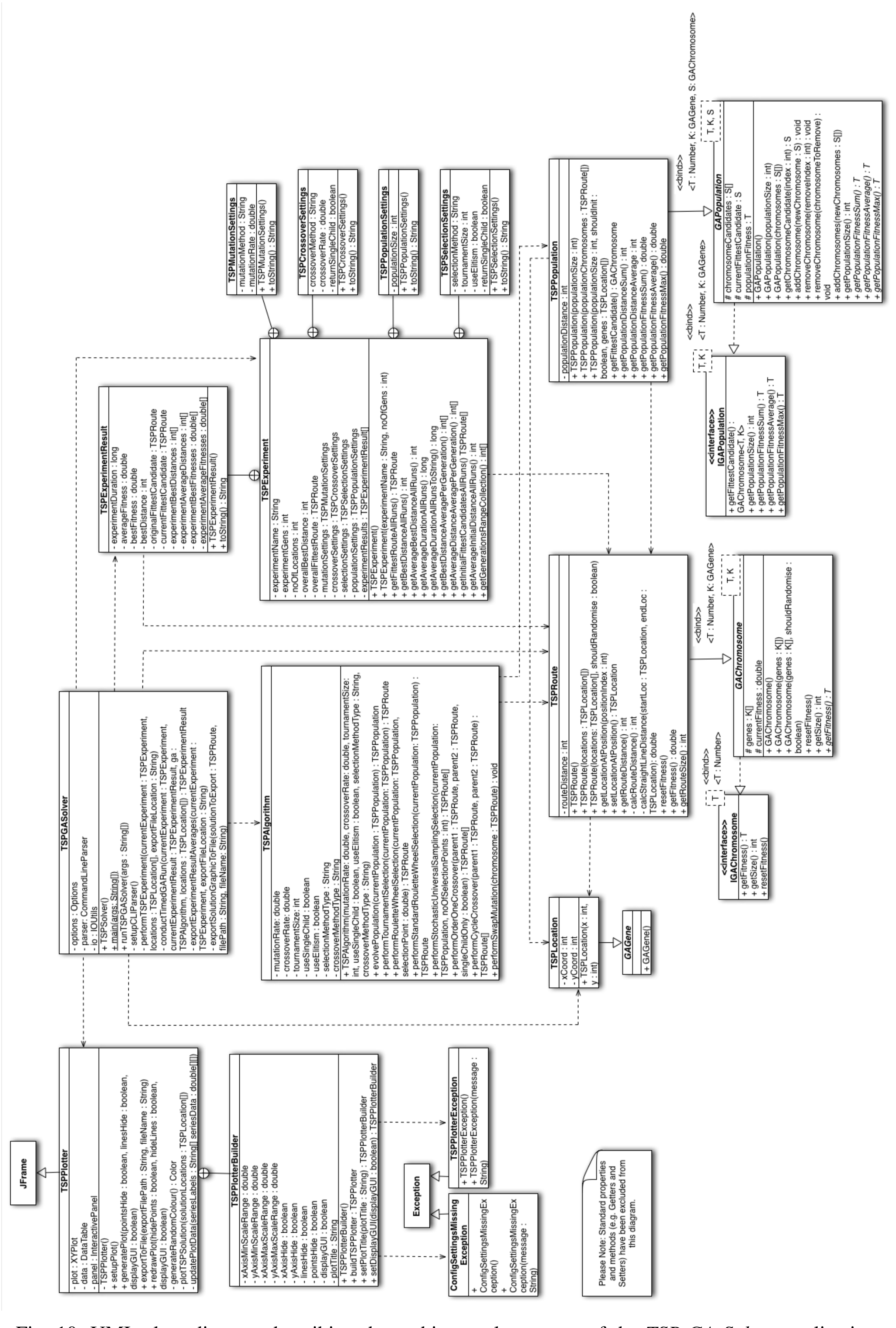


Fig. 10: UML class diagram describing the architectural structure of the *TSP-GA Solver* application.