Solving Travelling Salesman Problems with Genetic Algorithms

Samuel Jackson, University of Aberystwyth

I. Introduction

A genetic algorithm (GA) is a search and optimisation method frequently used to find approximate solutions in challenging problem domains. Genetic algorithms are inspired by the biological concept of natural selection. This places genetic algorithms under the category of biologically inspired approaches to optimisation; along with genetic programming, ant colony optimisation, and particle swarm optimisation. Many traditional optimisation techniques rely on the calculation of derivatives and often requires good knowledge of the search space. GAs on the other hand only require a measure of solution quality, making them well suited to difficult optimisation problems where traditional techniques would otherwise fail.

In a genetic algorithm solutions to a problem are encoded as chromosomes. A chromosome in GA terminology is usually an array of binary, integer, or real numbers but other representations are possible. For example, an array of real numbers might represent the encoding of coefficients of a polynomial in a curve fitting problem. A gene in GA terminology is a single atomic component of a chromosome. In the previous curve fitting example a gene would be a single coefficient.

A GA proceeds by creating an initial population of randomly generated solutions. From this population a subset of candidates are selected which are used to generate the next population. New solutions are generated from this subset using genetic operators. Their are two fundamental types of genetic operators: crossover and selection. Crossover creates new chromosomes from two or more parent chromosomes by combining portions from each of the parent chromosomes together in some way. Crossover aims to preserve some information about what makes decent solutions between generations. Mutation randomly modifies a chromosome by altering one or more of its genes. The mutation operator aims to encourage more exploration of the search space to avoid local minima. Finally, each chromosome in the new population is evaluated according to its fitness. The fitness of a chromosome is how well the solution encoded by the chromosome solves the problem. The fitness function is the entirely dependant on the problem domain. In the curve fitting example above the fitness function could be the mean squared error between a dataset and the polynomial represented by a particular chromosome.

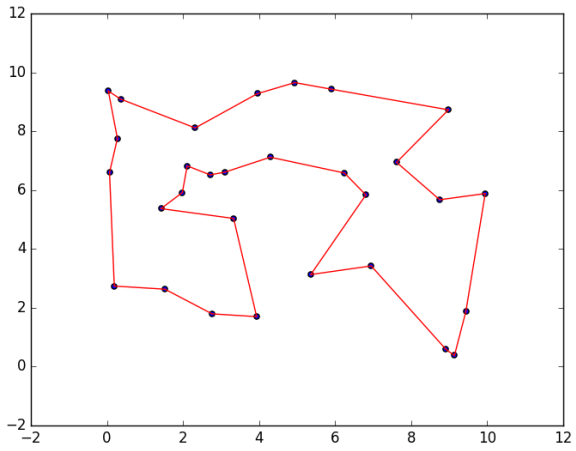

Fig. 1: Example of a solution to a TSP with 30 cities distributed uniformly at random. Each of the blue points represents a city. Each of the red lines indicates which city to travel to next. The solution shown is most likely optimal.

The travelling salesman problem (TSP) is a classic mathematical problem well suited for the application of GAs. The premise of the TSP is as follows:

> *Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?*

The problem is renowned for being simple to grasp but computationally intensive to calculate an exact solution. The TSP has been shown to be NP-hard and a brute force solution requires O(n!) time. Algorithms with factorial time complexity become unworkable with anything other than very small datasets. GAs easily lend themselves to the travelling salesman problem. A GA makes no guarantees about finding an optimal solution to a TSP, but can be used to find an approximate solution in a reasonable amount of time given decent parameters. It is able to do this faster than with a basic brute force search because a GA will only examine a subset of solutions in the search space that may or may not include the optimum result, but should with good parameters converge towards the optimum solution.

II. Program Description

My solution to this assignment is implemented as a Python package with a basic command line interface. The installation

of the package and the operation of the command line interface are described in detail in appendices **??** and **??** respectively.

The main implementation of the GA algorithm is within the sub-package *tspsolver.ga*. This package contains a separate module for each of the components of the genetic algorithm. It also includes the simulator module, which is responsible for composing each of the components and running the genetic algorithm itself.

Each of the component modules contains an abstract base class for the particular type of component and several subclasses which provide concrete implementations of specific types of that component. For example, the crossover module contains a class *AbstractCrossoverOperator* which implements operations common to all crossover operators and provides an abstract method *_crossover_for_chromosomes* which all subclasses must implement in order to derive the class. Provided that each of the components implement the specified interface, the *Simulator* class will be able to setup and use the component without knowledge of the implementations details of the concrete component. In software engineering terms this architecture is known as the "strategy" design pattern. The strategy enables the behaviour of the algorithm to be dynamically selected at runtime.

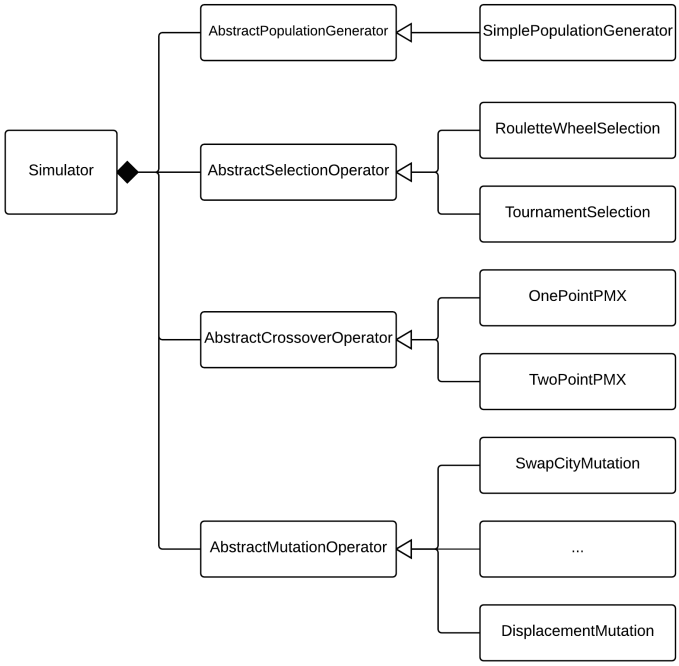

Fig. 2: Class diagram for the simulator class and its components using the strategy pattern. The simulator is composed with one of each type of component. The abstract classes provide and common interface for the components. Concrete implementations of the components derive from the relevant base class

My implementation takes this slightly further in that the *Simulator* class which composes the components of the GA together actually dynamically loads the correct components using Python's reflection capabilities according to user supplied string parameters. The parameters for the application are supplied via a command line interface in the form of a JSON file. Information regarding the structure of the parameter file can be found in the appendix **??**

Once instantiated the *Simulator* takes a 2D matrix with two columns (x and y coordinates) and n rows (equal to the number of cities) as a parameter to the evolve method. The evolve method runs converts the datasets into a distance matrix then approximates a solution using the genetic algorithm.

In addition to the main *tspsolver.ga* sub-package there are four other modules. These are really just supporting code for the running the simulator and producing the analysis. These modules are:

- **tsp_generator** - Implements a small class for generating TSP datasets with uniformly random cities.
- **command** - Implements the command line interface for running the simulator and includes a couple of file loading and saving routines.
- **plotting** - Defines some custom plotting functions for producing the graphs used in this report.
- **tuning** - A basic class for using a grid search to automatically select GA parameters.

III. Representation Discussion

For my experiments I chose to use a path based representation for chromosomes. In a path based representation each chromosome is represented as an array of integers. The value of each integer element represents a single city in the dataset. In my program this integer corresponds to the i^{th} row in the $2 \times n$ dataset. Each integer's position in the array indicates the order in which it will be visited. For example: in the array $[3, 4, 1, 5, 2]$ the first city to be visited would be city 3. For city 3 the tour then moves to city 4 and from city 4 to city 1.

Obviously since each city in the tour should only be visited once any valid solution should only contain a single occurrence of each city. Likewise because all cities must be visited exactly once each of cities in the dataset must be included in any valid solution. Therefore all valid solutions must be of size n where n is the number of cities in the dataset. In this representation format valid solutions are simply permutations of the enumeration of every city in the dataset.

While many different representations of the TSP are possible [2], a path based representation remains the most natural. It is both intuitive for the beginner to understand (of which I am one personally) and has a large array of applicable genetic operators which can be utilised. Other approaches are possible as shown in ref. [2], but these are shown to either have poor results or have peculiarities in their representations. For example in a binary representation there are some seemingly valid encodings for which their are no valid cities.

However, this does not mean that a path based representation is without its downsides. The primary issues with encoding a set of cities as a chromosome for a TSP is ensuring that the genetic operators used will produce valid tours. As mentioned in the preceding paragraphs, a valid tour needs to be a permutation of the enumeration of all cities in the dataset. It is easy to see how traditional naive genetic operators would inevitably create invalid tours by producing a solution that visits the same city twice and excludes one or more cities.

IV. Algorithm Components & Genetic Operators Discussion

There are three main types of genetic operators that are typically used as part of a genetic algorithm. These are selection, crossover, and mutation. In my application I have implemented a variety of different genetic operators. Additionally I have used two different techniques for generating of the initial populations. This section outlines the implementations I have used in the experiments in section V and provides a justification as to why they were chosen.

A. Population Generation

My genetic algorithm implementation has two different methods of initialising the population. The first type of population generation technique is purely random. The *SimplePopulationGenerator* in the *population_generation* module simple create the desired population size by creating random permutations of the enumeration of the list of cities.

Random initialisation is very general and works fine for all genetic operators tested in my application. However, it is painfully evident that some initial solutions are going to better than others. In an optimal solution a city will have the city will the shortest distance from it next in the chromosome sequence. In general, over a small local neighbourhood, a good heuristic might be to use the closest neighbours to a city as the adjacent neighbours in the chromosome ordered from closest to largest.

This will not always yield great chromosome solutions. Contemplate the closest neighbours to a few points i 1 for a few moments and you will clearly see that the k^{th} nearest element is not necessarily the kth element in the optimal tour. However, especially for larger datasets, the cities in the immediate vicinity of one particular city are more likely to end up closer together in the chromosome.

With this intuition I created a second population generator *KNNPopulationGenerator* which uses the K-nearest neighbours of a city to generate a chromosome. Specifically the algorithm works as follows: for each element of the population to be created it picks a city at random. An ordered list of the K-nearest neighbours to that city are then found where K is equal to the total number of cities. The hope is that this is more likely to lead to initial chromosomes which have some portion of their chromosomes already in the correct place (or nearly in the correct place) than simply choosing at random.

B. Selection

In genetic algorithm terms, selection is the method by which individuals from the current population are in order to produce a new population through crossover and mutation. A good selection technique should yield more "good" chromosomes for reproduction than bad ones. This is because the best solutions in the current population are generally more likely to produce even better offspring in the following generation.

However, this does not mean that we should always just sort and take the best solutions because this rapidly leads to a lack of diversity. Consider a candidate TSP solution that yields a poor fitness because the just one of the connections geographically cuts from one side of the world to the other. In this scenario the fitness function ranks the solution as poor, but in reality it is quite close to being optimal! If a selection approach naively selects just the best looking solutions we are more likely to head towards local optima and potentially do not explore the full search space.

In my application, I have implemented two classic yet contrasting types of selection operator. The first selection procedure I implemented was *Roulette Wheel* selection [1]. Roulette wheel selection uses a probability distribution over each potential solution to be picked. Solutions are weighted proportionally to their fitness. *Roulette Wheel* selection selects chromosomes are random and proportionally to their weighting. *Roulette Wheel* selection gets its name from the fact that individuals are weighted in proportion to the area of a sector of a roulette wheel [1].

I chose to use *Roulette Wheel* selection for two main reasons: firstly, it is very simple to implement and secondly, being one of the most basic techniques, it makes a nice yard stick which other techniques can be compared against. However, this approach has some well know downsides. *Roulette Wheel* selection is well known to often be "too random". While candidate solutions are weighted such that more promising individuals are more likely to be picked there is still a huge amount of variation in which solutions actually get picked. This is especially true considering the shear number of solutions generated in even a moderately sized GA problem.

Because of the limitations of *Roulette Wheel* selection I chose to also implement *Tournament* selection [1]. In *Tournament* selection a subset of the total population of chromosomes is chosen at random. The chromosomes in this random subset are then compared against each other. In my implementation I have decided to make the simplest choice an implement *strict Tournament* selection where the chromosome with the biggest fitness is always the "winner" of the tournament. An alternative formulation is the use of *soft* tournaments where the winner is probabilistically selected according to their fitness.

Tournament selection is a very practical technique that is simple to implement, scales well, and can potentially be parallelised. Another key advantage over *Roulette Wheel* selection is that *Tournament* selection provides a parameter that directly adjusts the selection pressure applied by the selection operator. The selection pressure is the likelihood that an average individual will be selected over the fittest individual. Changing the size of the tournament influences how likely it is that weaker chromosomes will survive the selection process. Bigger tournaments lead to an increase chance of a better individual entering the tournament; therefore increasing the probability that they'll be selected and visa versa. However, it is also worth noting that *Tournament* selection still suffers from the same issues as *Roulette Wheel* selection. The random nature of the algorithm can mean that the distribution you get is skewed and not fully representative.

C. Crossover

The crossover operator is used to recombine selected individuals to form a new population with solutions that are

different (and hopefully better) than the previous population. A good crossover operator should attempt to preserve the best portions of the selected chromosomes. Without the counter effect of a mutation operator a good crossover operator should eventually cause the GA to converge to a (possibly not optimal) solution.

The implementations of the crossover operators that I have used in my application can be found in the *crossover* module. I have implemented three different crossover operators, two of which are fairly similar to one another.

The first two operators I have implemented are *OnePointPMX* and *TwoPointPMX*. Both of these are variants of the *partially mapped crossover* (PMX). As their names suggest the only different in their implementation is the number of pivot points used to define which parts of the chromosomes get recombined (similar to regular one and two point crossover). PMX crossover begins by copying some sub section of the parent chromosome to the child. This inevitably leads to an invalid tour because there will be duplicate cities in resulting solution. The PMX operator then repairs the new chromosomes using the mapping of the elements replaced by copying over the sub-tour. The algorithm proceeds by iterating over the parts of the chromosome outside of the copied sub-tour. If a duplicate element is found then it is replaced by inserting the corresponding element that was in original chromosome overwritten by copying the sub-tour using the mapping. PMX is both conceptually simple to understand and to implement. It is similar to regular one and two point crossover but produces valid TSP tours. This makes it a good base line operator to compare other techniques with.

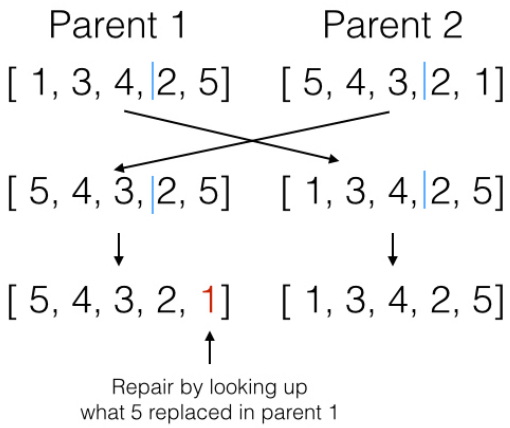

Fig. 3: Shows how the PMX operator creates new valid children. First, sub-tours from the parent are copied to the children. Then the chromosomes are repaired using by looking at what was replaced in the original parent.

The other crossover operator that was implemented for this assignment is *OrderCrossover* [3] (OX). Order crossover begins in a similar manner to PMX. Sub-tours from both chromosomes are chosen from the parents at random and copied to the children. The algorithm differs in how it repairs the chromosomes. To repair the chromosome each element starting from the second pivot point is copied from the second parent to the child in order, skipping those which are already in the copied sub tour. The OX operator has an advantage over PMX because it attempts to preserve the order of the chromosomes that were not copied for the original tour. This should help to prevent the crossover accidentally being too destructive.

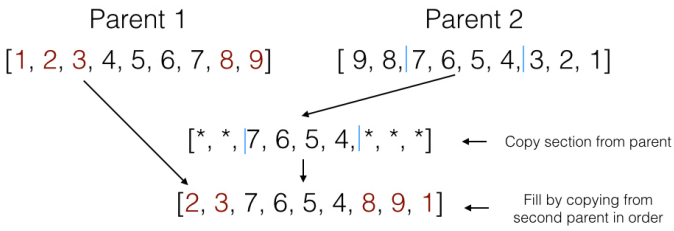

Fig. 4: Shows how the OX operator creates new valid children. First, sub-tours from the parent are copied to the children. Then the chromosomes are repaired using by copying elements from the second parent in the order they appear, excluding ones which are already included.

D. Mutation

The mutation operator is used to randomly modify selected individuals, usually after crossover. Mutation operators, in contrast to crossover, aim to prevent the algorithm from converging too early and attempt to diversify the population.

In my program I have implemented four different types of mutation operator which are suitable for TSP problems [2]. They are: *SwapCity*, *Displacement*, *Inversion* and *Insertion* mutations.

The *SwapCityMutation* is the most simple mutation that I have implemented for this assignment. This operator chooses two cities at random in the chromosome and swaps their positions. This implementation is perhaps the simplest operator possible and therefore makes a good base line to compare other operators against.

The *DisplacementMutation* is the next logical step up from the *SwapCityMutation*. In this operator, instead of swapping two cities within the chromosome a whole sub-tour is moved within the chromosome. The motivation for this operator is that swapping individual cities is often too destructive. Moving a portion of the sub tour should result in being less destructive as good paths are potentially preserved while still exploring the search space.

The *InversionMutation* takes the *DisplacementMutation* another step further. In this mutation type a sub-tour is randomly moved within the chromosome but is also inserted in the reverse order. Again this has the advantage of potentially not being too destructive to good solutions. Additionally this can help solve cases where the GA has found a section good route, but is hampered by a a few long connections at either end of the tour. By reversing the sub-tour before re-insertion could solve this by removing some of the long connections through chance.

Finally, the *InsertionMutation* is an alternative operator which operates in a similar to the *SwapCityMutation*. With the *InsertionMutation* a single gene is removed from a chromosome and reinserted in another place. This is likely to be

mutator crossover	DisplacementMutation	InsertionMutation	InversionMutation	SwapCityMutation
OnePointPMX	89.824891	90.873543	89.093593	94.777183
OrderCrossover	84.346435	83.070859	78.645158	93.692092
TwoPointPMX	86.838374	87.167188	85.360710	96.247643

TABLE I: Median fitness of running each of the different types of crossover and mutation with each other over 5 randomly generated datasets each containing 50 points. All parameter sets had a crossover rate equal to 0.9 and a mutation rate of 0.1. Tournament selection was used with population size of 20 and tournament size of 5. Each was run for a total of 1000 generations.

mutator_pmutate crossover_pcross	0.01	0.05	0.10	0.20
0.6	125.636980	94.648238	87.416766	82.935809
0.7	120.357263	95.173636	88.066053	80.483827
0.8	116.652596	93.404031	84.183019	81.736283
0.9	112.598065	91.804595	81.899921	80.405973

TABLE II: Median fitness of running each of the different types of crossover and mutation parameters with *OrderCrossover* and *InversionMutation*. Tournament selection was used with population size of 20 and tournament size of 5. Each was run for a total of 1000 generations.

useful in cases where a single city is linked to other cities much further away from itself. By mutating randomly inserting the city in a different place it may be that the distance travelled to and from it is reduced.

V. Experiments Performed

This section explains the experiments carried out using the aforementioned system and components described in the preceding sections. For all experiments in this section of the document, unless otherwise noted, the following setup is used: In order to fairly compare the operators a parameter grid is generated for each type of operator. Each set of parameters is tested on a total of 5 different randomly generated datasets each with 50 cities. This number was chosen to be high enough to show the differences between parameters and operators, but low enough to make computation time realistic. All parameter sets are tested on the same random datasets to ensure consistency. The median fitness value is taken to represent the test as a whole. This is to ensure that the outcome of the test was not just random chance or affected by a single outlier.

A. Crossover and Mutation operators

The first experiment examines which of the crossover and mutation operators work best together. Table I shows the summary results of comparing the crossover operators against mutation operators. Figure 5 shows the fitness of each operator combination over a number of 1000 epochs for a single trial.

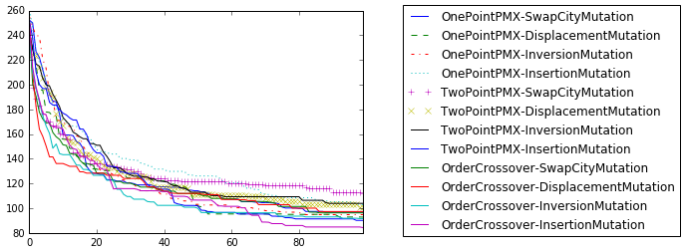

Fig. 5: Convergence of each of the crossover and mutation operator pairs on a sample dataset. Each line is the minimum (best) fitness achieved for every 10 generations. It can be seen that Order crossover with an Displacement and Inversion mutation are the quickest to make progress initially, while in this instance Order crossover with Insertion mutation finds the best solution

B. Crossover and Mutation Parameters

Motivated by the results of this experiment and constrained by the size limits of this report and the time complexity of calculating many runs on multiple parameters sets the findings for the remaining experiments will use *OrderCrossover* with *InversionMutation* with mutation and crossover parameters selected using a grid search as described above over a small range of sensible parameter choices.

This section presents the results of searching for those parameters over a range of possible values. Table II shows the results of running a grid search over a small range of possible values for the probability of crossover and mutation. The general trend in towards a larger crossover and mutation rate.

generator_population_size selector_tournament_size	10	20	30	40
3	95.170012	80.785238	77.076346	77.999664
5	87.522664	81.151858	75.956841	73.441637
10	83.614738	74.550498	74.843452	72.645017

TABLE III: Median fitness of running each of the different types parameter values for population size and tournament size with *TournamentSelection*. For all tests *OrderCrossover* and *InversionMutation* are used. All runs used a 0.9 crossover rate and a 0.2 mutation rate based on the results in section V-B. Each was run for a total of 1000 generations.

	generator_population_size	fitness
0	10	196.904247
1	20	202.703760
2	30	195.786973
3	40	197.601059

TABLE IV: Median fitness of running each of the different population sizes with *RouletteWheelSelection*. For all tests *OrderCrossover* and *InversionMutation* are used. All runs used a 0.9 crossover rate and a 0.2 mutation rate based on the results in section V-B. Each was run for a total of 1000 generations. No results successfully converged

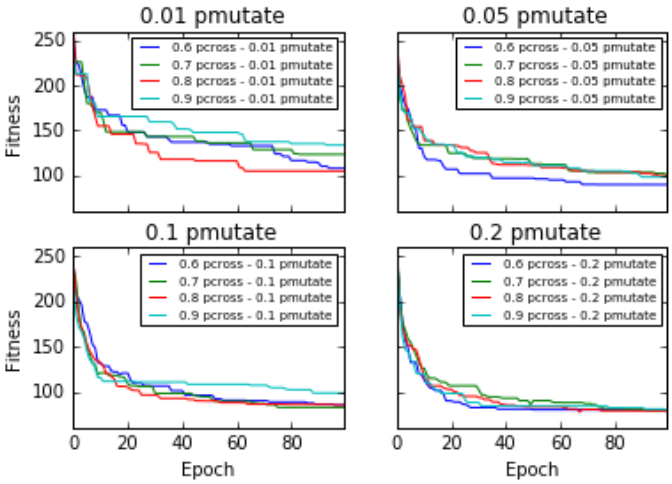

Fig. 6: Convergence of the GA over a range of different parameters for crossover and mutation. All trials used *OrderCrossover* and *InversionMutation*. Each line is the minimum (best) fitness achieved for every 10 generations. Selection was performed using tournament selection with tournament size of 5 and a population size of 20. The number of points used for each trial was 50.

C. Selection Operators

This section looks at the effect of modifying the selection parameters for both *TournamentSelection* and *RouletteWheelSelection*. Once again this has been limited to just using *OrderCrossover* and *InversionMutation*. The crossover and mutation rates for each run are fixed to 0.9 and 0.2 based on the results of prior testing in the previous section. The method used in this section is the same as the previous sections. 50 data points are used for each run. Sadly *RouletteWheelSelection* failed to converge in all trials. Table III shows the results for the trails on population size versus tournament size for *TournamentSelection*. Table IV shows the results of different population sizes for *RouletteWheelSelection*. Figure 7 shows a single trial run with varying population sizes and tournament sizes.

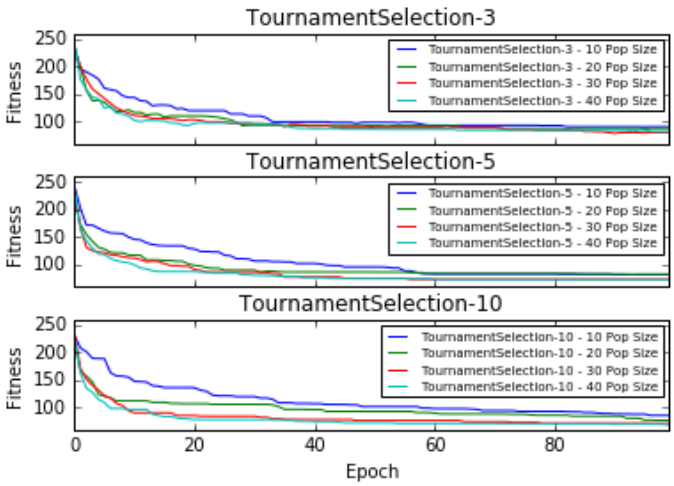

Fig. 7: Convergence of each of the different tournament sizes over a range of different population sizes for a single trial run. Each line is the minimum (best) fitness achieved for every 10 generations. Generally larger population and tournament sizes produce quicker convergence.

D. Elitism

This section shows the effects of adding elitism to the genetic algorithm. Elitism allows all algorithm to copy x number of chromosomes unaltered from one generation to another. This ensures that the algorithm doesn't accidentally "forget" the its best result.

	num_elites	fitness
0	0	74.018956
1	1	73.651241
2	2	74.376653

TABLE V: This shows a trial run on three different settings for the elites parameter. This run uses a similar setup to the proceeding section, but uses tournament size of 10 and population size of 40. This suggests that maintaining one elite chromosome between populations produces the best results.

	generator	generator_random_proportion	fitness
0	SimplePopulationGenerator	NaN	134.464263
1	KNNPopulationGenerator	0.3	128.815815
2	KNNPopulationGenerator	0.5	126.743674
3	KNNPopulationGenerator	0.6	120.437607

TABLE VI: The fitness for a batch of 5 datasets each containing 100 cities.

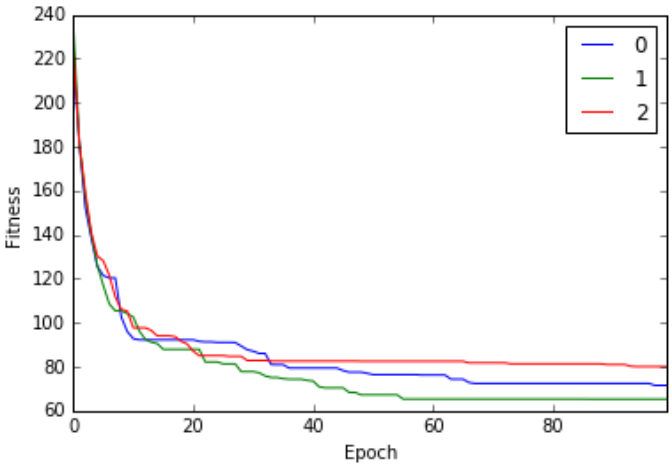

Fig. 8: Shows the convergence of one run for each setting of the elite parameter. Again, like other trials this used 50 cities.

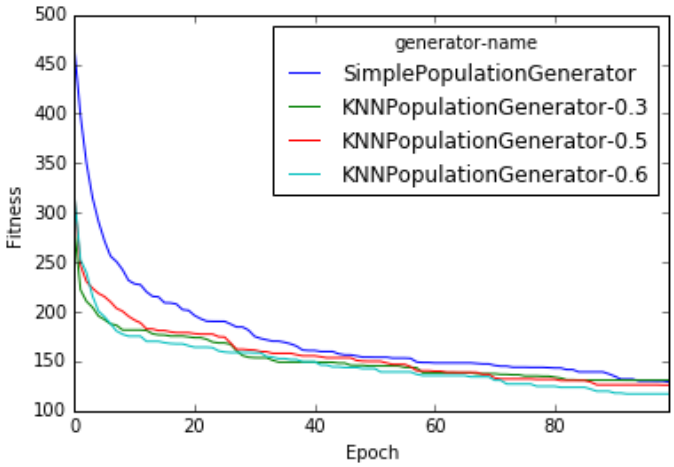

Fig. 9: Shows the convergence of the different settings for population generation for one trail over a dataset of 100 cities. It can be clearly seen that the KNN approach leads to faster convergence.

E. Population Initialisation

This section shows the effects of using different population initialisation strategies. Table VI shows the effect of running the same experimental setup with different population generators. For the *KNNPopulationGenerator* a parameter specifies the proportion of the chromosomes that are generated by taking the KNN neighbours. Figure 9 shows the convergence of one trial for each parameter set. In both cases datasets containing 100 cities were used. This is so that the difference can be clearly seen. The setup for these trials used the same parameters as section V-C but also include a single elite chromosome in each generation.

VI. Discussion and Analysis

Section V-A compared the differences between a variety of different crossover and mutation operators. In these experiments *OrderCrossover* was consistently the best crossover operator out of the three. This matches my expectations because order crossover is designed to better preserve the order of chromosome elements. This is obviously very important in problems such as the TSP where the order is perhaps the most import part of the solution rather than just the producing valid tours. I would suggest that order based crossover better encodes heuristics about the problem domain than PMX.

Comparing the mutation operators shows that *InversionMutation* appears to produce the best results and *SwapCity* mutation produces the worst. This seems to be a reasonable result. Logically the *SwapCity* mutation should be more destructive than the displacement and inversion operations which both keep whole portions of the chromosome in order. As order is the most important aspect of the TSP these mutations must be able to push the GA to explore the search space effectively without being overly destructive, hence the better results. It's also worth noting that the insertion mutation achieves results

that are very similar to the displacement mutation and much better than just swapping cities. I found this to be slightly unexpected. It could just be that this operator performed better because it's slightly less destructive to the order or it could just be a statistical fluke.

In terms of the probability of crossover and mutation parameters used results in section V-B clearly show a tendency to prefer larger crossover and mutation rates. Looking at table II it can be clearly seen that the results definitively become better towards the bottom right. What is also interesting in this result is that the effect of changing the mutation parameter has a larger effect on the results of the experiment than the probability of crossover. Comparing this with figure 6 shows that increasing the mutation rate increases the chance of the genetic algorithm finding a better solution. The effect of the crossover rate appears to be a little more subtle. Looking closely at figure 6 shows having a very high crossover rate (0.9) causes the GA to plateau earlier. I.e. a high crossover rate causes the GA to converge faster but not necessarily to the best solution. However, when combined with a hight mutation rate, it can be seen that the convergence rate becomes more consistent, although not identical.

The results of testing the selection operator show some interesting insights. Unfortunately here roulette wheel selection failed to converge in any tests. I believe this is down to the fact that the GA becomes "too random". In contrast to tournament selection, roulette wheel selection is less likely to draw a consistent distribution of chromosomes. Therefore while it may find a good solution, there's less of a guarantee that the solution will make it to the next generation than with tournament selection. This combined with the destructive effects of mutation most likely combine to make convergence difficult. I ran some additional trials (not included) with a much lower mutation rate (0.01) and larger population size (50) in order to encourage a better result with no success.

VII. Conclusions and Future Work

REFERENCES

[1] R Reeves Colin and ER Jonathan. Genetic algorithms-principles and perspectives, a guide to ga theory, 2002.

[2] Pedro Larrañaga, Cindy M. H. Kuijpers, Roberto H. Murga, Inaki Inza, and Sejla Dizdarevic. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial Intelligence Review*, 13(2):129–170, 1999.

[3] Pablo Moscato et al. On genetic crossover operators for relative order preservation. *C3P Report*, 778, 1989.