Solving Travelling Salesman Problems with Genetic Algorithms

Samuel Jackson, University of Aberystwyth

I. Introduction

A genetic algorithm (GA) is an 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 require detailed 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 GA solutions are encoded as chromosomes. A chromosome 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 and used to generate the next population. New solutions are generated from this subset using crossover and selection operators. Crossover creates new chromosomes from two or more parent chromosomes by combining portions from each of the parent chromosomes together. 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 exploration of the search space and avoid local minima. Finally, each chromosome in the new population is evaluated according a measure of quality known as the "fitness". 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 of the polynomial represented by a particular chromosome.

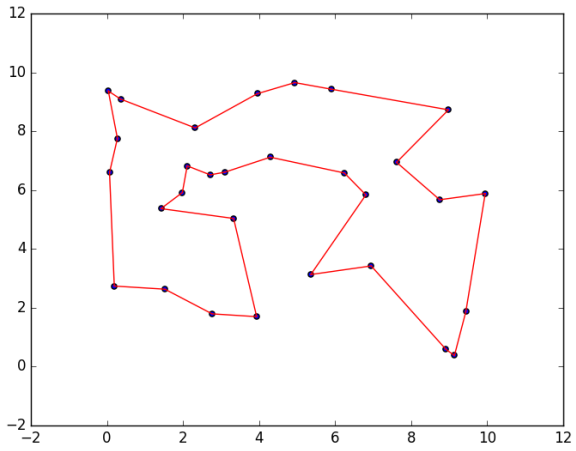

Fig. 1: Example of a solution to a TSP with 30 cities distributed uniformly at random. Each of the blue points represents a city. The red lines indicate the order of traversal between cities.

The travelling salesman problem (TSP) is a classic mathematical problem. The TSP is defined as follows:

> *The traveling salesman problem is a problem ... requiring the most efficient (i.e., least total distance) Hamiltonian cycle a salesman can take through each of n cities.*[7]

The problem is simple to understand but computationally intensive to calculate an exact solution. The TSP has been shown to be NP-hard [7] and a brute force solution requires $O(n!)$ time. 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. This is faster than 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 converge towards the optimum solution with good parameters.

II. Program Description

The program for this assignment is implemented as a Python package with a basic command line interface (CLI). The installation of the package and operation of the CLI are described in detail in appendices A and B.

The main implementation of the GA algorithm is within the sub-package *tspsolver.ga*. This contains a separate module

for each of the components of the GA. It also includes the *simulator* module, which is responsible for composing each of the components and running the GA itself.

Each of the modules contains an abstract base class and several subclasses providing concrete implementations of types of the 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. This allows the *Simulator* class to use each component without knowledge of the implementation details of the concrete component. In software engineering terminology this is known as the "strategy" design pattern. The strategy pattern enables the behaviour of the algorithm to be dynamically selected at runtime.

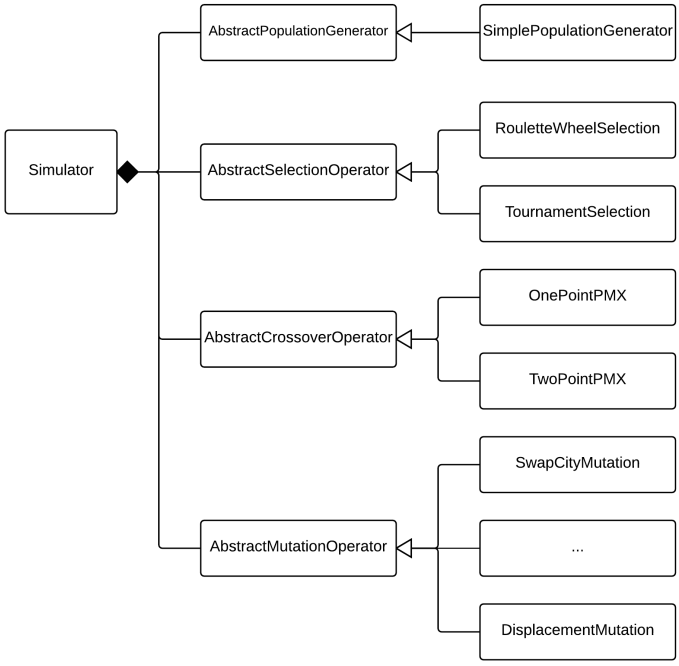

Fig. 2: Class diagram for the simulator class and its components using the strategy pattern. The simulator is composed with one of each type of component. The abstract classes provide a common interface for each of the components. Concrete implementations of the components derive from the relevant base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

This implementation takes the pattern slightly further. The *Simulator* class also dynamically loads the correct components using Python's reflection capabilities according to user supplied parameters. The parameters for the application are supplied via the CLI in the form of a JSON file. Information regarding the structure of the parameter file can be found in the README and information on the parameters in appendix C.

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 parameters to the *evolve* method. This method converts the datasets into a distance matrix which is used to approximate a solution using the GA.

In addition to the main *tspsolver.ga* sub-package there are four other modules. These contain supporting code for running the simulator and setting up the CLI. These modules are:

- **tsp_generator** - Implements a class for generating TSP datasets with uniformly random cities.
- **command** - Implements the CLI and includes a couple of file I/O routines.
- **plotting** - Defines a couple of custom plotting functions.
- **tuning** - Defines a class for running a grid search over a range of GA parameters.

III. Representation Discussion

The experiments in this report use a path based representation. In a path based representation each chromosome is an array of positive integers. The value of each integer represents a single city. In this program an integer corresponds to the i^{th} row in the dataset. Each integer's position in the array indicates the order in which it will be visited. For example: in $[3, 4, 1, 5, 2]$ the first city to be visited would be city 3, followed by city 4 and then city 1 etc.

Since each city in the tour must only be visited once any valid solution only contains a single occurrence of each city. Therefore all valid solutions must be of size n where n is the number of cities. In this representation valid solutions are therefore simply permutations of the enumeration of every city in the dataset.

While many different representations of the TSP are possible [3], a path based representation remains the most natural. It is both intuitive to understand and has many different genetic operators which can be utilised. Other approaches are possible as discussed in ref. [3], but these are shown to either have poor results or have implementation difficulties. For example, a binary representation has some seemingly valid encodings which represent no valid city.

However, this does not mean that a path based representation is without limitations. 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. It is easy to see how traditional naive genetic operators would create invalid tours by producing a solution that visits the same city twice.

IV. Algorithm Components & Genetic Operators Discussion

There are three main types of genetic operators that are typically used as part of a GA: selection, crossover, and mutation. This application implements a variety of different genetic operators. Additionally two different techniques for generating the initial populations are provided.

A. Population Generation

The first type of population generation technique is purely random. The *SimplePopulationGenerator* in the *population_generation* module creates the desired population size by creating random permutations of the list of cities.

Random initialisation is a sensible, general baseline. However, it is obvious that some initial solutions will be better

than others. In an optimal solution each city will have a city with a short distance from it next in the sequence. In general, a good heuristic may be to use the closest neighbours to a city as the adjacent entires in the chromosome.

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

With this intuition a second population generator *KNNPopulationGenerator* [5] is implemented which uses the K-nearest neighbours (KNN) of a city to generate a chromosome. The algorithm works as follows: for i^{th} member of the population it picks the i^{th} city. An ordered list of the KNN's to that city are found where K is equal to the n. The aim is that this is more likely to lead to initial chromosomes which have some part of their chromosome already close to the correct place compared with choosing at random. To prevent a dramatic reduction in diversity, only a portion of the population is initilised in this way.

B. Selection

In GAs, selection is how individuals from the current population are chosen to produce a new population. 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 more likely to produce even better offspring in the following generations.

This doesn't mean that the best solutions should always be chosen because this rapidly leads to a lack of diversity. Consider a candidate TSP solution that yields a poor fitness because just one of the connections geographically cuts across the whole "world". In this scenario the solution is ranked as poor, but in reality it is quite close to being optimal! A selection approach that naively selects just the best looking solutions is more likely to head towards local optima and would potentially fail to explore the full search space.

The experiments in this report only utilise a single type of selection operator. Originally, two types of selection operator were going to be compared. These were *roulette wheel selection* (RWS) and *tournament selection* (TS) [1]. However RWS was dropped due to slow performance and poor convergence on larger datasets in comparison with TS. RWS is well known to suffer from stochastic noise leading to a large variation in the expected distribution.

In TS a small subset of the total population is chosen at random and compared against each other. This program implements *strict* TS, where the chromosome with the best fitness always "wins" the tournament. An alternative formulation would be to use *soft* tournaments where the winner is probabilistically selected proportionally to their fitness.

Tournament selection is a very practical technique that is simple to implement, fast, and can be parallelised. Another key advantage over RWS is that TS provides a parameter that adjusts the selection pressure applied by the operator. Selection pressure is the likelihood that an average individual will be chosen over the fittest individual. Changing the tournament size influences how likely it is that weaker chromosomes will survive the selection process. Bigger tournaments lead to an increased chance of a better individual entering the tournament causing them to be selected and visa versa. However, it is worth noting that TS, like RWS, still suffers from stochastic noise but the effects are dampened thanks to selection pressure.

C. Crossover

The crossover operator is used to recombine selected individuals to form a new population that is distinct from (and hopefully better) than the previous population. A good crossover operator should preserve the best portions of a chromosome. Without the counter effect of mutation a good crossover operator should naturally cause the GA to converge to a (possibly optimal) solution.

The crossover operators implemented in this application can all be found in the *crossover* module. There are three distinct operators, two of which are fairly similar. The first two operators are *OnePointPMX* and *TwoPointPMX*. Both of these are variants of *partially mapped crossover* (PMX). As their names suggest the only different in their implementation is the number of pivot points used to produce sub-tours.

PMX crossover begins by copying a sub-tour of one parent chromosome to the child. This can lead to an invalid tour because there will be duplicate cities in the resulting solution. PMX repairs the new chromosomes using the mapping between the elements replaced when copying 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 in the parent chromosome 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 to compare other techniques against.

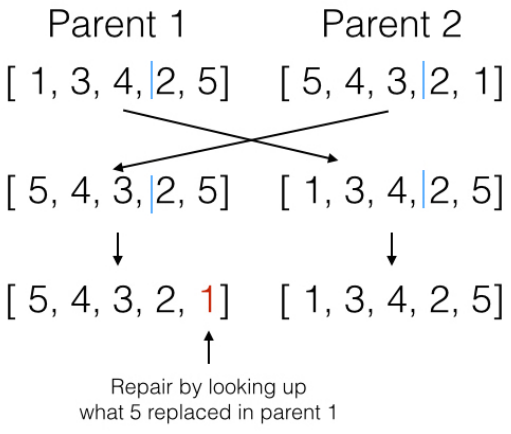

Fig. 3: Shows how the *TwoPointPMX* operator creates new valid children. First, sub-tours from the parent are copied to the children. Then the chromosomes are repaired by looking at what was replaced in the original parent.

The other crossover operator is *OrderCrossover* (OX) [4].

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 from the original tour. This should help to prevent the crossover accidentally being too destructive.

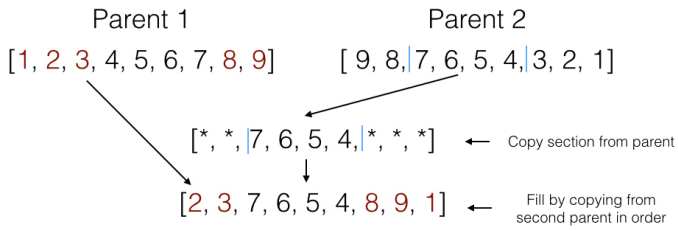

Fig. 4: Shows how the OX operator creates new valid children. First, sub-tours from the parent are copied to the child. Then the chromosome is repaired using by copying elements from the second parent in the order they appear, excluding ones which are already included.

D. Mutation

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

This program implements four types of mutation operator suitable for TSP problems [3]. They are: *SwapCity*, *Displacement*, *Inversion* and *Insertion* mutations.

The *SwapCityMutation* chooses two cities at random and swaps their positions. This is perhaps the simplest mutation operator and therefore makes a good base line to compare other operators against.

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

The *InversionMutation* takes *DisplacementMutation* another step further. *InversionMutation* works identically to *DisplacementMutation* but the sub-tour is inserted in the reverse order. In addition to the benefits of *DisplacementMutation* this can help solve cases where the GA has found a good route, but is hampered by a few long connections at the end of a tour. Reversing the sub-tour before insertion can solve this by removing some of the long connections through chance.

Finally, in *InsertionMutation* a single gene is removed from a chromosome and reinserted in another place. This is likely to be useful in cases where a single city is linked to other cities much further away from itself. By 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 describes the experiments performed using the system described in the preceding sections. For all experiments, unless otherwise noted, the following setup is used: A parameter grid is generated for each combination of parameters. Each set of parameters is tested on a total of 5 randomly generated datasets each with 50 cities. This number was chosen to be high enough to show the differences between parameters, but low enough to make computation time realistic. All parameter sets in the test use the same datasets to ensure consistency. The median fitness value is taken to represent the whole parameter set. This ensures 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 how crossover and mutation operators work together. Table I shows the results comparing the crossover operators against mutation operators. Figure 5 shows the fitness of each operator combination over 1000 epochs for a single trial.

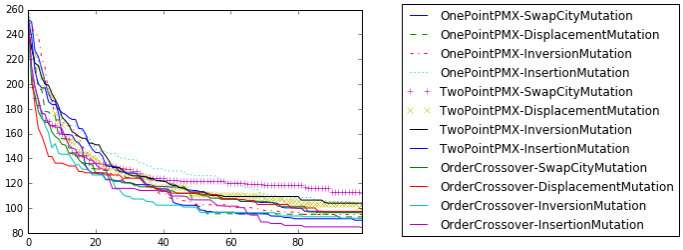

Fig. 5: Convergence of each of the crossover and mutation operator pairs on a sample dataset. Each line is the minimum (best) fitness achieved for every 10 generations. It can be seen that Order crossover with an Displacement and Inversion mutation are the quickest to make progress initially, while in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

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 is towards a larger crossover and mutation rate.

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.

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 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.

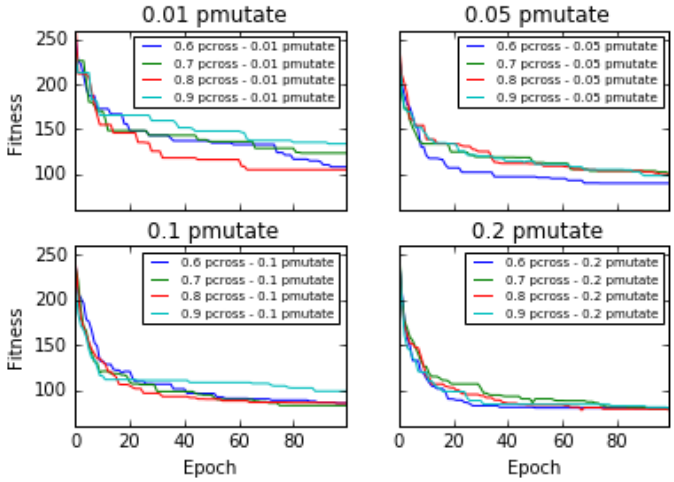

Fig. 6: Convergence of the GA over a range of different parameters for crossover and mutation. All trials used *OrderCrossover* and *InversionMutation*. Each line is the minimum (best) fitness achieved for every 10 generations. Selection was performed using tournament selection with tournament size of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

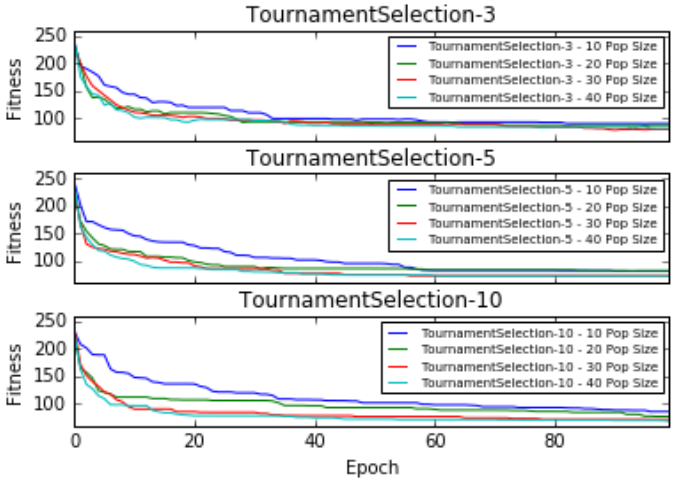

Fig. 7: Convergence of each of the different tournament sizes over a range of different population sizes for a single trial run. Each line is the minimum (best) fitness achieved for every 10 generations. Generally larger population and tournament sizes produce quicker convergence.

	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 IV: The fitness for a batch of 5 datasets each containing 100 cities with different population initialisation strategies.

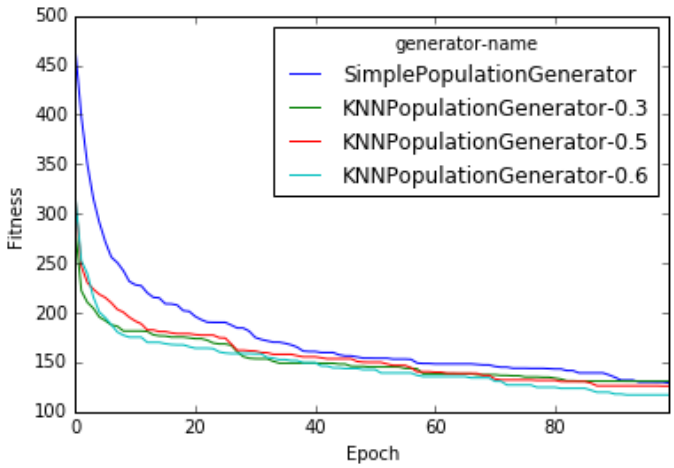

Fig. 8: Shows the convergence of the different settings for population generation for one trial over a dataset of 100 cities. It can be clearly seen that the KNN approach leads to faster convergence.

C. Selection Operators

This experiment examines the selection parameters *TournamentSelection*. Once again this has been limited to just using *OrderCrossover* and *InversionMutation*. Crossover and mutation rates are fixed to 0.9 and 0.2 based on the results of the previous section. Table III shows the results for the trials on population size versus tournament size for *TournamentSelection*. Figure 7 shows a single trial run with varying population and tournament sizes.

D. Population Initialisation

Table IV 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 8 shows the convergence of one trial for each parameter set. In both cases datasets containing 100 cities were used. This is to better accentuate the differences.

VI. Discussion and Analysis

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

Comparing the mutation operators shows that *InversionMutation* appears to produce the best results and *SwapCityMutation* produces the worst. Logically the *SwapCityMutation* should be more destructive than the displacement and inversion operations which keep whole portions of the chromosome in order. As order is key to solving the TSP these mutations must be able to push the GA to explore the search space effectively without being overly destructive. It's worth noting that the *InsertionMutation* achieves results that are very similar to the *DisplacementMutation* and much better than just swapping cities. This was 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.

The crossover and mutation rate results (section V-B) clearly shows a tendency towards larger crossover and mutation rates. Looking at table II it can be 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 rate has a larger effect on the results than the crossover rate. Comparing this with figure 6 shows that increasing the mutation rate increases the chance of the GA finding a better solution.

The effect of the crossover rate appears to be more subtle. Looking at figure 6 shows that 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 high mutation rate, the convergence becomes more consistent, although not identical.

Testing the selection operator shows some interesting insights. Looking at table III and figure 7 a clear relationship can be seen between the population size used and the tournament size. Best results were achieved with both a large population and tournament size. Another interesting point is that like the relationship between the mutation and crossover parameters the difference between the values becomes smaller as the parameters increase. This suggests two things 1) there is an optimum setting between the two parameters and 2) that increasing them indefinitely will probably not significantly improve GA performance. Also, in figure 7 it can be seen that a very small tournament size (3) stifles the diversity and makes it harder for the algorithm to make progress.

The results in table 8 show the a direct relationship where the fitness of the population decreases in proportion to the number of KNN based chromosomes used. This is backed up by figure 8 which shows a difference in the initial rate of convergence between random population generation and all settings of KNN population generation. As predicted this

difference was more prominent with a greater number of cities hence why 100 cities where used.

VII. Conclusions and Future Work

In conclusion, a few interesting global trends have been identified in applying GAs to the TSP. One trend appears to be that the best choice of crossover and mutation operators are ones which maximally preserve the order of sub-tours. This might prevent the algorithm from loosing good portions of candidate solutions and convergence on local minima.

Secondly, it seems that high crossover and mutation rates are preferred. In particular the mutation rate seems to make the highest impact on the rate at which a GA stabilises on a particular solution. The crossover rate also has an impact, but it's effect is less prominent.

TS appeared was more stable than RWS. With tournament selection a larger population size and tournament size increase performance. However, the tournament size and in particular the population size significantly impacts the time complexity. Combined with the fact that increasing these parameters doesn't linearly increase performance confirms that a balance is required between selection parameters and computation time.

The use of KNN population generation can provide a useful increase to the performance of an algorithm. KNN population generation can lead to faster convergence and better solutions by starting with better chromosomes. But a balance must be struck to prevent decreasing diversity.

GAs appear to be well suited to the problem of solving the TSP. With this system good solutions could be found for 30 to 50 cities, but quality deteriorated beyond this. However, the requirement that the genes must be unique and that ordering is important is a handicap for GAs. It would be interesting to investigate the effect of ant colony optimisation on the TSP as this approach more naturally lends itself to path finding in comparison to a GA.

Finally, there are several areas which have not been fully explored here, but which could be explored in future work. This includes modifying the implementation to use multi-processing. GAs naturally lead themselves to parallelisation and such an implementation would make it easier to run larger parameter sets. Another idea would be to explore modification of the population and parameters over time. This could use approaches such as simulated annealing to control crossover & mutation parameters or to reintroduce diversity using approaches such as random offspring generation [6] or social disasters [2].

APPENDIX A
INSTALLATION OF PROGRAM

Installation is easiest using the *pip* package manager. Python version 2.7.9+ automatically ships with *pip*. If you're using an older version of Python you can find the installation instructions at http://pip.readthedocs.org/en/stable/installing/.

To install the program locally *cd* into the top level directory of the project and run the following:

```
pip install -e .
```

All of the project dependancies should be installed automatically. If for some reason they are not, the full list of dependancies are:

- numpy
- scipy
- pandas
- scikit-learn
- matplotlib
- click

APPENDIX B
COMMAND LINE INTERFACE

There are three major commands: *generate*, *solve* and *tune*.

A. Generating Datasets

The generate command will create a uniformly random TSP dataset and output it to the specified CSV file.

```
tspsolver generate dataset.csv
```

Optionally, it can also take a parameter specifying the number of cities to generate (default is 10):

```
tspsovler generate -n 30 dataset.csv
```

B. Solving TSP problems

The *solve* command can be used to run the genetic algorithm and produce solutions to TSP problems. The command takes a JSON parameter file as an argument. Optionally you can provide a dataset file (such as produced by the *generate* command) or specify a random number of cities to generate. When the command terminates, two plots are produced. One shows the min, mean, and max fitness across all generations, the second shows the best solution found over all datasets.

Example Parameter File:

```
{
    "num_epochs": 1000,
    "num_elites": 2,
    "generator": "SimplePopulationGenerator",
    "generator_population_size": 20,
    "selector": "RouletteWheelSelection",
    "selector_tournament_size": 5,
    "crossover": "OrderCrossover",
    "crossover_pcross": 0.9,
    "mutator": "InversionMutation",
    "mutator_pmutate": 0.1
}
```

Solving a TSP problem with a randomly generated TSP dataset:

```
tspsolver solve -n 20  params.json
```

Solving a TSP problem with an exisitng dataset:

```
tspsolver solve -f dataset.csv params.json
```

C. Tuning Parameters

The final command can be used to run a range of parameter configurations over a number of different datasets. This can be useful to examine the effects of different parameter datasets. Each configuration is run on n different randomly generated datasets and the median result is taken to represent the whole. The results for all datasets are saved to a CSV file. The configuration that produced the best results is also saved to a JSON file.

This command takes a special parameter file that specifies ranges of parameters. An example is shown below:

```
{
    "num_epochs": [1000],
    "num_elites": [0],
    "generator": ["SimplePopulationGenerator"],
    "generator_population_size": [20],
    "selector": ["TournamentSelection"],
    "selector_tournament_size": [5],
    "crossover": ["OrderCrossover"],
    "crossover_pcross": [0.6, 0.7, 0.8, 0.9],
    "mutator": ["InversionMutation"],
    "mutator_pmutate": [0.01, 0.05, 0.1, 0.2]
}
```

This will run the genetic algorithm with a varying range of crossover and mutation probabilities. An example of running the tuning command is as follows:

```
tspsolver tune -d 5 -n 50 tuning_params.json results
    .csv best.json
```

In the above command the *-d* command specifies the number of datasets to generate for each parameter configuration. The *-n* flag specifies the number of random generated points to use for each dataset. *tuning_params.json* is the special parameter file with ranges. *results.csv* is the CSV file created with all parameter results. *best.json* is the generated parameter file containing the parameters that produced the best run.

APPENDIX C
PARAMETERS

This appendix provides an overview of the different parameters that can be used with the system with example values and a description of what the parameter controls.

Parameter	Example Value	Description
num_epochs	1000	The number of epochs to run the genetic algorithm for.
num_elites	2	The number of elites to carry over the the next generation.
generator	SimplePopulationGeneration	The type of generator to use to initialise the population. This can either be *SimplePopulationGeneration* or *KNNPopulationGenerator*.
generator_population_size	20	Specifies the population size of the genetic algorithm
generator_random_proportion	0.5	When using KNN population generation this controls the ratio of individuals selected using KNN relative to the number of random chromosomes. Must be in the range $0 \leq x \leq 1$
selector	TournamentSelection	Specifies which type of selection to use. This can either be *TournamentSelection* or *RouletteWheelSelection*
selector_tournament_size	5	When using tournament selection this controls the size of the tournament used.
crossover	OrderCrossover	Specifies the crossover operator to use. This can be one of: *OrderCrossover*, *OnePointPMX*, or *TwoPointPMX*.
crossover_pcross	0.9	This controls the probability that crossover will occur. Must be in the range $0 \leq x \leq 1$
mutator	InversionMutation	Specifies the mutation operator to use. This can be one of: *SwapCityMutation*, *InversionMutation*, *InsertionMutation*, *DisplacementMutation*.
mutator_pmutate	0.1	This controls the probability that mutation will occur. Must be in the range $0 \leq x \leq 1$

REFERENCES

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

[2] V Kureichick, AN Melikhov, VV Miagkikh, OV Savelev, and AP Topchy. Some new features in genetic solution of the travelling salesman problem. In *Adaptive Computing in Engineering Design and Control*, volume 96, 1996.

[3] 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.

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

[5] Wayne Pullan. Adapting the genetic algorithm to the travelling salesman problem. In *Evolutionary Computation, 2003. CEC'03. The 2003 Congress on*, volume 2, pages 1029–1035. IEEE, 2003.

[6] Miguel Rocha and José Neves. Preventing premature convergence to local optima in genetic algorithms via random offspring generation. In *Multiple Approaches to Intelligent Systems*, pages 127–136. Springer, 1999.

[7] Eric W Weisstein. Traveling salesman problem. 2006.