Solving Travelling Salesman Problems using Genetic Algorithms

SEM6120 - Assignment 2

Alexander D Brown (adb9)

Contents

1 Introduction

This report investigates the use of Genetic Algorithms to solve the Travelling Salesman Problem, introducing both topics as well as previous research and techniques.

Following this the design for the solution implemented will be detailed, focusing from high level system design to annotated code section to explain the choice behind the implementation.

This implementation is then used to produced the results in section 3 which give an insight into the affects of different crossover and mutation schemes on the performance of the genetic algorithm, specifically looking at crossover operators, crossover rate and mutation rate.

1.1 Genetic Algorithms

Genetic algorithms are a biologically-inspired approach to heuristic search that mimic natural selection. Unlike many other evolutionary strategies and evolutionary programming, they are not designed to solve a specific problem, but are designed to solve the problem of optimisation which is made difficult by substantial complexity and uncertainty[1].

1.2 Travelling Salesman Problem

The Travelling Salesman Problem is a well-known NP-hard optimisation problem which asks the question: *"Given a number of cities and the distances between these cities, find the shortest touch which visits each city exactly once and returns to the first city."*

Assuming each city is connected to every other city, this problem reaches a complexity of $O(n!)$ and is very resource intensive to brute force a problem with any decent number of nodes quickly become too larger problem to solve within a reasonable amount of time.

There are many heuristic algorithms which have been applied successfully to the Travelling Salesman Problem, including both Evolutionary and Genetic Algorithms.

Though the nature of Genetic Algorithms are suited to the optimisation of a Travelling Salesman Problem, normal methods of crossover and mutation cannot, generally, be applied directly to the problem. The representation of chromosomes has to be ordered (i.e. each city must appear once and only once) and additional methods of crossover and mutation have had to be designed for these ordered chromosomes.

2 Design

2.1 System Design

As with most coding problems with multiple options, a modular approach is necessary to keep code quality high. This is typically done by defining high-level interfaces for the changeable elements. In this case the obvious three interfaces are:

1. Selection Scheme
2. Crossover Scheme
3. Mutation Scheme

From these high-level classes, concrete sub-classes can be written to perform the actual logic. An example of this would be a specific class for Order-1 based crossover.

This leaves the problem of how to accesses these classes based on an input string; the easiest method for this is to use factories to access these classes, cutting down the number of specific imports required and centralising the logic for creating them.

2.1.1 Language Choice

Python was the choice of language, it is a dynamically typed language which provides several programming paradigms to work with, including procedural, object-orientated and functional paradigms.

It is a language the author is very familiar with and has the advantage of having many open source libraries to perform different scientific functions; some of these libraries have been used in the course of this project, including the popular `numpy` library for number processing and `matplotlib` to produce graphs.

The strange choice of `pygame` was made for the choice of displaying the GUI, but this games library gives simple yet powerful access to OpenGL and also manages platform dependencies.

2.1.2 UML Class Diagram

Figure 1 shows the initial UML Class diagram for this project. There are some elements which break from typical object-orientated design, noticeably the representation of nodes in a graph as a map of integers to a turple of float, these parts are done so to re-use internal data structures of the Python programming language to speed up the implementation of many features.

This design is such that factories exist to create selection schemes, crossover schemes and mutators based on an input string to facilitate the switching of these elements via command line arguments.

This design was slowly improved through the project; the `CrossoverScheme` class implemented the `crossover` method, which then called a separate method, `do_crossover`, to generate `c1` based on `do_crossover(p1, p2)` and `c2` on `do_crossover(p2, p1)` to make the processing more uniform. Subclasses were still able to override the `crossover` method, but were encouraged to implement

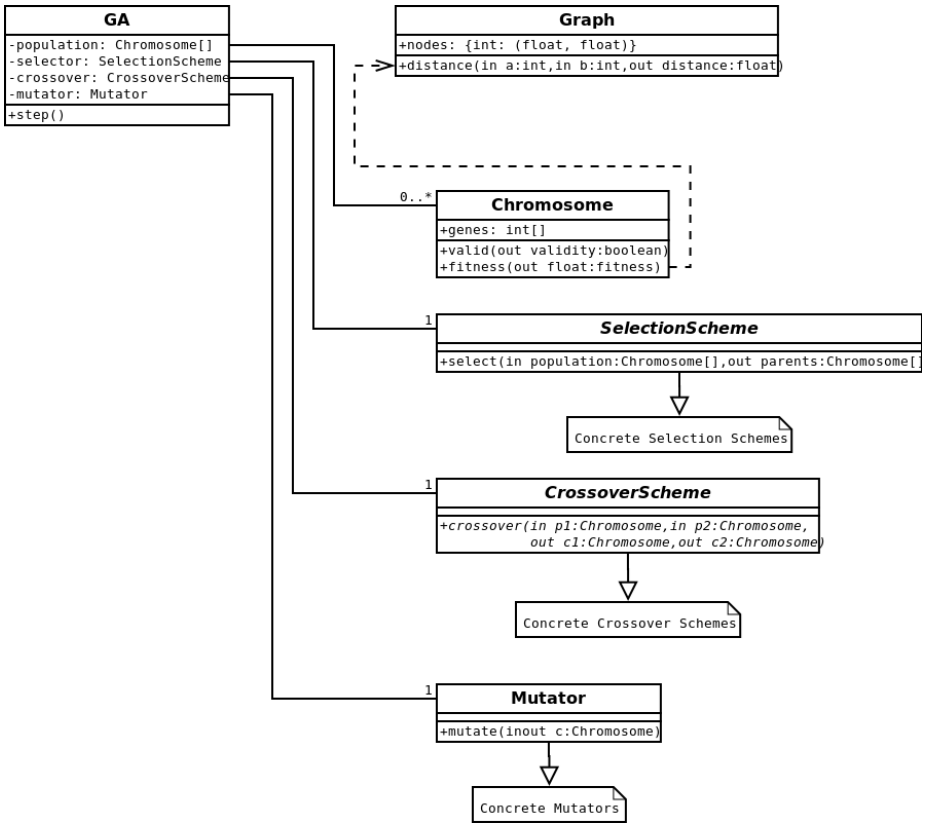


Figure 1: UML Class Diagram for the Genetic Algorithm

a `do_crossover` method unless the scheme required a different behaviour of child generation.

2.1.3 Representation of Graphs

The actual of representation of a graph is a map of the node identifier to the x and y co-ordinates of that node (as a tuple). This allows easy look up of nodes within the map to get the position of the node. This was chosen because of the representation of the problem in chromosomes - each gene represents a node in the graph.

With this representation, the fitness function would be the distance of the tour represented by these nodes:

$$d_{tour} = \sum_{i=0}^{N} \begin{cases} d(n_i, n_{i+1}) & \text{if } i + n < N \\ d(n_i, n_0) & \text{else} \end{cases} \tag{1}$$

Programatically, with the advantage of the functional and in-built elements of Python, this can be simplified to:

```
def d_tour():
  # Move the first element of the array to the end.
  shifted = nodes[1:] + nodes[:1]
```

```
return sum([distance(i, j) for (i, j) in zip(nodes, shifted)])
```

Listing 1: Distance of a tour

2.2 Use of Functional Programming Paradigms

As Python implements several different programming paradigms a lot of problems can be solved with a different approach than other languages can. Both genetic algorithms and the travelling salesman problem lend themselves towards a more functional approach with a lot of list processing. Using Pythons list comprehensions and the in-built list functions shortened the amount of code required and makes the code a lot easier to understand for those who are used to this approach.

A very good example of this is the method for evaluating chromosomes in a population, returning a list sorted from the best to the worst:

```
def eval(population):
  return sorted(population,
                key = lambda chromosome: chromosome.fitness())
```

Listing 2: Using function elements to improve sustinctness and readability

2.3 Crossover Strategies

Three main types of crossover strategies were implemented for this research:

1. Cycle Crossover,
2. Order Crossover Operator,
3. M-Crossover Operator.

All three are designed to work with ordered chromosomes, meaning they can be applied directly to the Travelling Salesman Problem without any modification.

2.3.1 Cycle Crossover

Cycle crossover is one of the simplest order-based crossover operators. Unlike many other forms of order-based operators it make no effort to preserve parts of the parent chromosomes.

To produce a child, c, from parents p_1 and p_2 a randomly selected point i is chosen. The gene at i in p_1 ($r = p_1(i)$) is removed and replaced with $p_1(i) = p_2(i)$. The position of the removed allele r is found in p_2 such that $i = index(r, p_2)$. This process is repeated until a cycle occurs (i.e. the removed allele r is the same as the initially removed allele).

Figure 2 shows this process step by step on two ordered chromosomes. Note that it only by chance that parts of the tour are preserved.

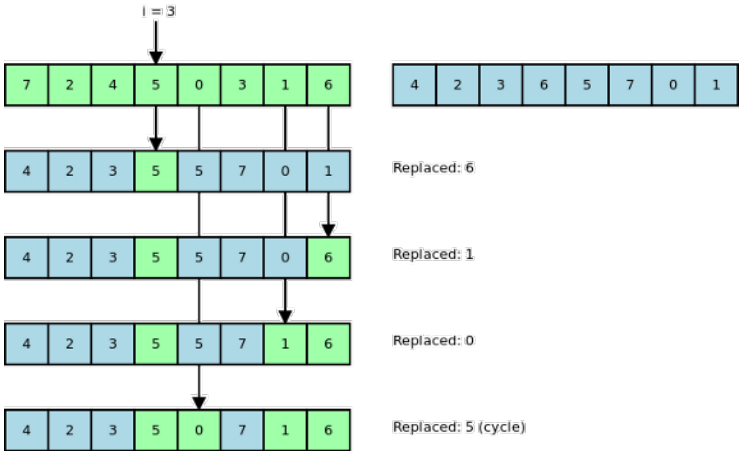


Figure 2: Cycle Crossover

2.3.2 Order Crossover Operator

The order crossover operator is another simple order-based crossover operator. Unlike cycle crossover, it preserves at least part of a tour from one of the parents.

To produce a child, c, from parents p_1, p_2, a random segment from p_1 is appended to the remaining genes from p_2, omitting any alleles that are also in the segment from p_1.

Figure 3 shows this process step by step on two ordered chromosomes. Note that at least a part of the tour is always preserved and that it is often the case that a part of a tour from p_1 and p_2 is preserved in c.

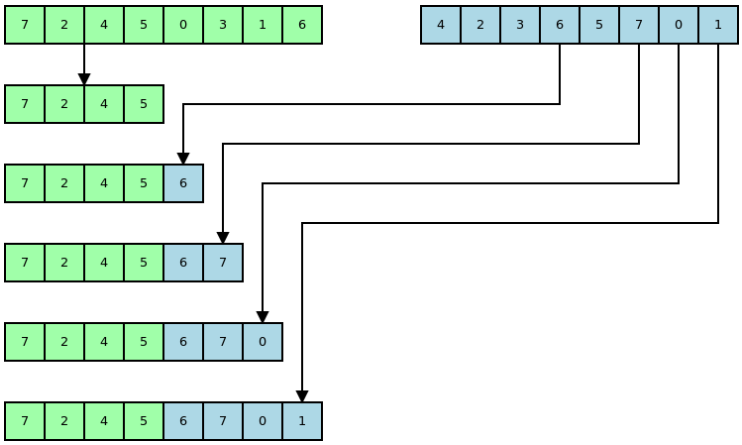


Figure 3: Order Crossover Operator

2.3.3 M-Crossover Operator

The m-crossover operator[2] produces multiple offspring (C) from parents p_1, p_2 then selects the best two from this process.

Like with the order crossover operator, this is based on segmenting the chromosomes. However, both chromosomes are split into several segments. Every segment of p_1 is inserted at any point in front, behind or between the segments of p_2 and vice versa. The new, elongated chromosome is then processed such that any allele in the added segment is removed from all other segments to produce the child at that point.

The fitness of these children is then evaluated and the best two are carried forward as the offspring of these parents.

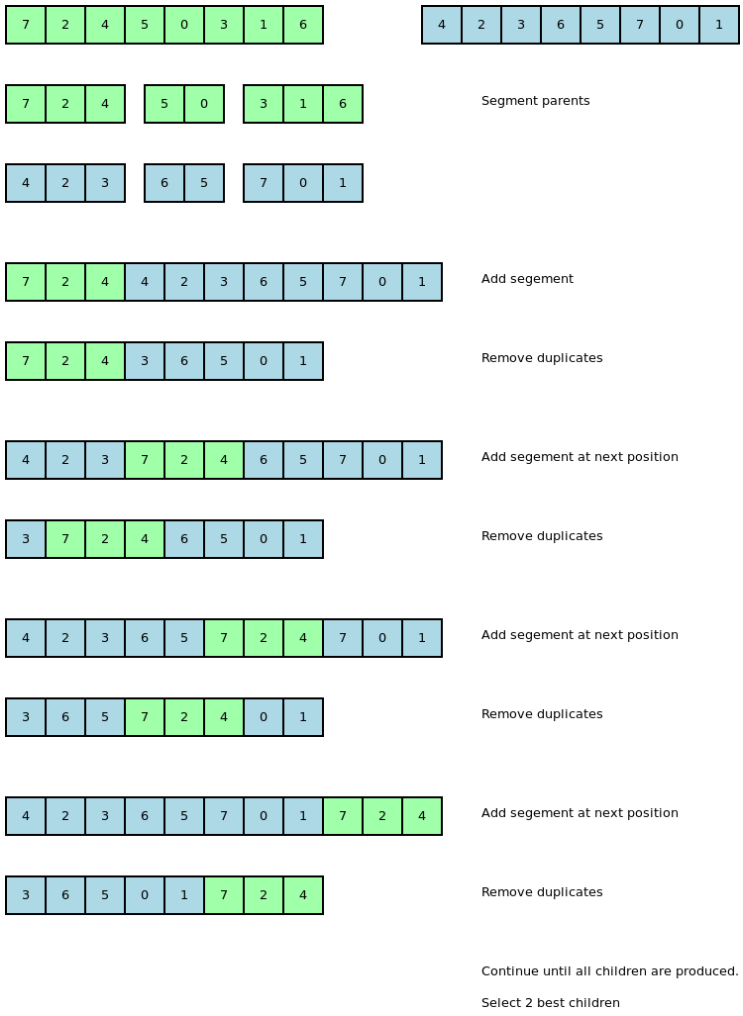


Figure 4: M-Crossover Operator

2.4 Mutation Operators

Two mutation operators were implemented for this problem. Again, as with crossover, normal mutation operators cannot be directly applied to a genetic algorithm with ordered chromosomes.

One fairly standard mutation operators for ordered chromosomes is the swapping of genes. To do this, two genes are selected at random and are swapped, ensuring that the chromosome is always still valid.

This mutation operator does have the problem that it can significantly change tours and though this encourages variety, it may introduce too much.

To help reduce this, a second mutation operator was designed so that only adjacent genes are swapped. This helps to preserve most of the tour whilst changing small parts of it.

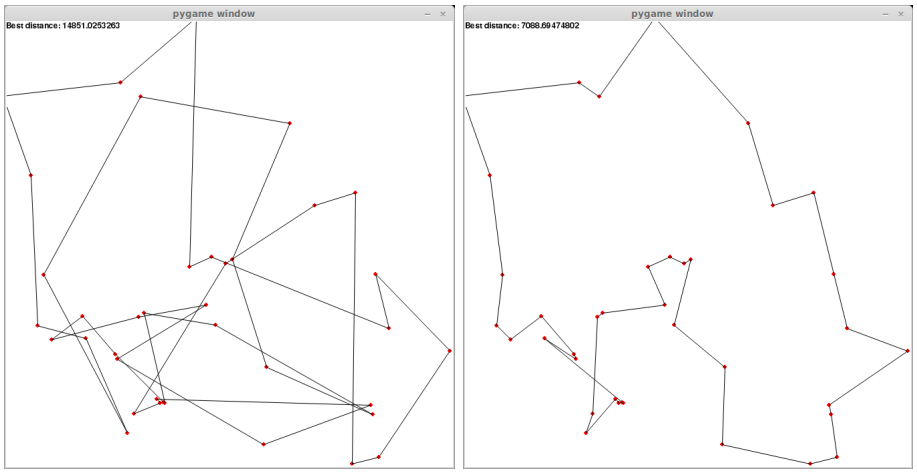
2.5 Parallelisation

Although genetic algorithms benefit greatly from a multi-threaded approach, there was not enough time to make the code parallel. The attempts made to do so led

3 Results

To gain some insight from the implementation the impacts of several aspects of the GA were investigated: the crossover strategy, the crossover rate, the mutation rate and the mutation scheme.

Figure 5 shows an example of the genetic algorithm in progress.



(a) At early generation. (b) At a late generation.

Figure 5: The Genetic Algorithm at different points in the runtime.

3.1 Impact of Crossover Strategies

The crossover strategy is one of the biggest influences on the performance of genetic algorithm, the first set of experiments should focus on the differences between them.

All results are run over 500 generations and averaged over 100 experimental test runs.

Figure 6 shows the experimental results for all three of the crossover operators.

Cycle crossover consistently performs worse than the other two, likely because it doesn't preserve good parts of tours. It also has very little variation between individuals of the population as the average fitness is very close to the best fitness.

It should be noted from the visualisation of the GA that cycle crossover tends quickly towards a local minima, but has difficulty finding lower minima. It is likely that with a higher mutation rate cycle crossover would perform better as more varied possibilities would be considered.

The m-crossover operator performs fairly well but, as with cycle crossover, converges quickly which slows the rate of performance after a small number of generations. This is fairly intuitive as it is constantly selecting the best possible offspring from the parents.

It seems that the m-crossover operator is good for getting a quick estimate of the problem in a low number of generations. But for finding a solution as

close to the optimal as possible it would seem that it isn't the best crossover operator.

The other issue with the m-crossover operator is that it consumes a lot more resources and time than the other two, as it has to generate extra children, evaluate them and then only choose two of them.

By far the best operator for reaching a solution close to the optimal is the order crossover operator. It manages to keep plenty of variety in the population whilst still managing to get the best fitness very close to the global minima.

All three operators would be trivial to make parallel (though due to time constraints the author did not do so) and so could be made more efficient in term of runtime.

Of the three it feels the order crossover operator is the most useful for general application, whilst the m-crossover operator would suit a problem which would a good solution quickly. The cycle crossover is very simple, but should have a performance similar to the order crossover operator.

3.2 Impact of Crossover Rate

The next step was to investigate the impact of crossover rate on the performance of a genetic algorithm.

All results are run over 500 generations and averaged over 100 experimental test runs using the order crossover operator which was the best of the three crossover operators chosen.

Figure 7 shows the experimental results for with three different crossover rates: 0.1, a very low crossover rate; 0.7, a commonly used crossover rate; and 0.9, a very high crossover rate.

It should be noted that this implementation is such that crossover rate defines not only the chances of crossover being performed, but also the ratio of the population which is made up of newly generated offspring. For example, if the crossover rate is 0.7 then the offspring will make up 70% of the new population and the rest will be the best 30% of the old population.

The results show slightly interesting results. The results from the crossover rate of 0.1 is intuitive, a low crossover rate is expected to elongate the number of generations it takes to optimise a solution. It also has a profound affect on the convergence of the population in that it converges quickly, which appears to be uncommon for the order crossover operator.

Where the results get interesting is that the higher crossover rate of 0.9 reaches a better best solution than the standard crossover rate of 0.7. This may be a result of the implementation of crossover rate, or because it works well with the order crossover operator. However, as the other crossover operators perform similarly, one could say that it is because of the implementation of the genetic algorithm.

The other possibility is that a high crossover rate is good for the travelling salesman problem, however most literature in the area would suggest otherwise.

3.3 Impact of Mutation Rate

All results are run over 500 generations and averaged over 100 experimental test runs using the order crossover operator and a crossover rate of 0.7.

Figure 8 show the experimental results for three mutation results: 0.01, a standard mutation rate for small populations; 0.25 a medium mutation rate; and 0.5, a relatively high mutation rate.

Interestingly the change of mutation rate have very little affect, other than to increase the variance of the population. For the higher mutation rate it does the speed of optimisation as well as increasing the variance. However, it is interesting that the medium mutation rate of 0.25 produces a better result than the standard mutation rate of 0.01. However as the population size is 100, it is on the cusp of needing a standard mutation rate of 0.1.

3.4 Impact on Mutation Operator

All results are run over 500 generations and averaged over 100 experimental test runs using the order crossover operator and a crossover rate of 0.7.

Figure 9 show the experimental results for the swap mutator operator and the swap adjacent operator.

As is fairly intuitive, the swap adjacent operator reduces the variety in the population. However, it doesn't allow the GA to get as close to the optimal solution as the swap operator does.

Although the swap operator doesn't preserve tours as well, it seems to allow the genetic algorithm to easier get out of local minima, whilst limiting the swap to only adjacent genes is too limiting and the algorithm can often get caught in local minima.

It may be the case that limiting the swap to the two (or more) adjacent nodes would help it to avoid this problem and this may be an interesting, if small, piece of research to perform.

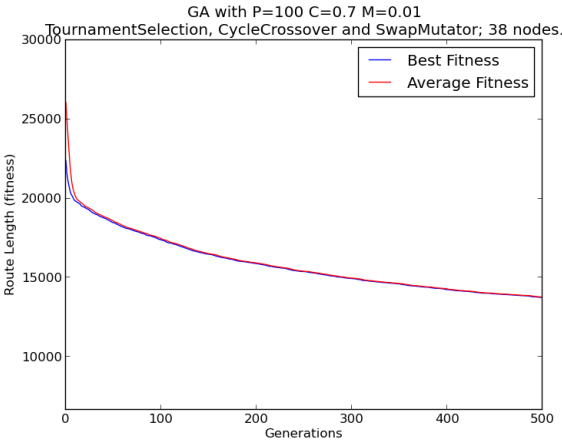
3.5 Applying the GA to larger problems

Another important application of a GA is the ability to apply it to larger data sets. Due to time limitations the number of runs the results are averaged over has been reduced to 25, but the number of generations has been increased to 1000 to allow the GA more opportunity to reach optimal values. The problem used had a total of 194 nodes.

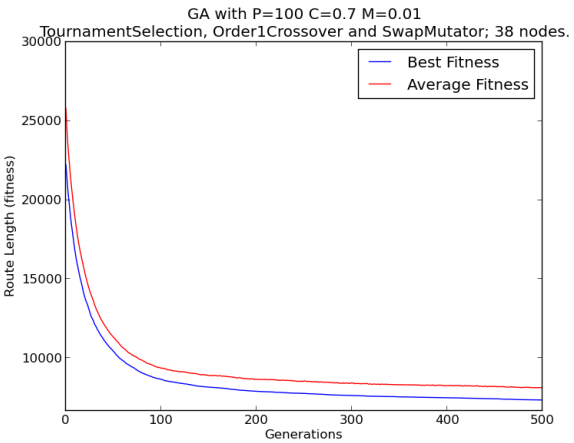
To further compare the crossover schemes from earlier, the time taken to execute was also recorded using the GNU Tools `time` command. These results are shown in table 1 and the graphs are shown in figure **??**.

Crossover Operator	**User**	**System**	**CPU**	**Total**
Cycle Crossover				
Order Crossover Operator	1665.10s	0.12s	99%	27:46.47
M-Crossover Operator				

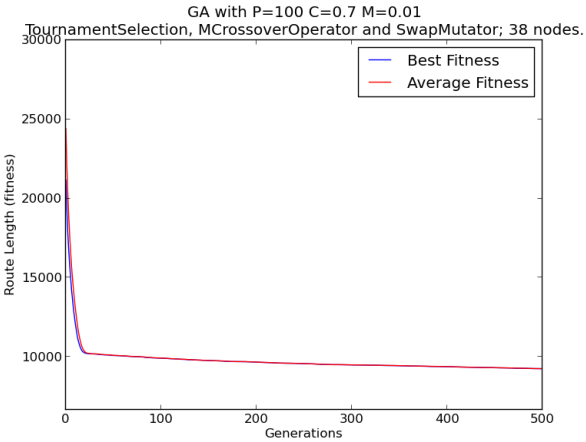
Table 1: Results of the `time` command for the different crossover operators on a GA with a population of 100, crossover rate of 0.7, mutation rate of 0.1 and with the swap mutation operator.



(a) Cycle Crossover

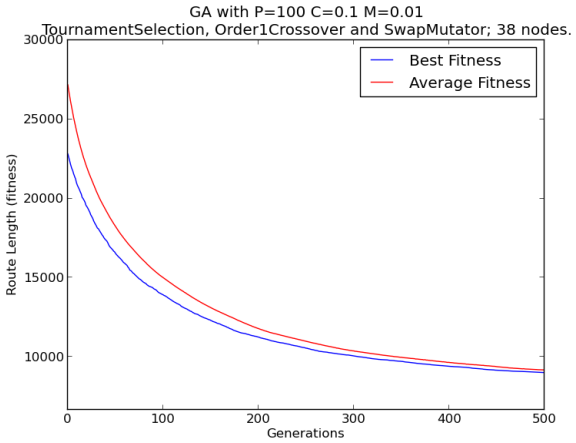


(b) Order Crossover Operator

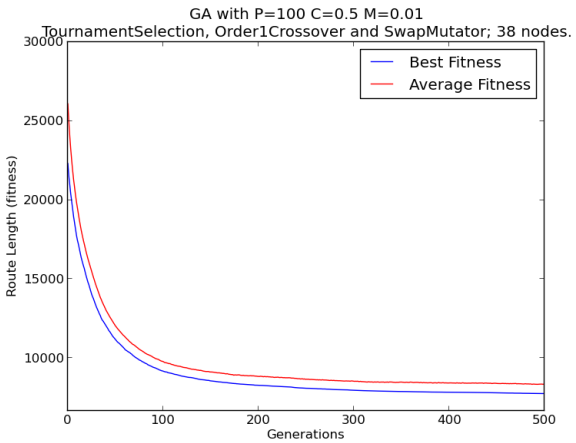


(c) M-Crossover Operator

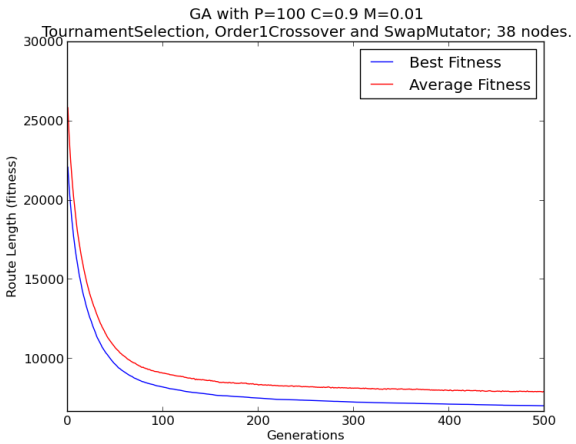
Figure 6: Results on different crossover operators on a data set with 38 nodes. All used a population of 100, crossover rate of 0.7, mutation rate of 0.01 and the swap mutator



(a) $c = 0.1$

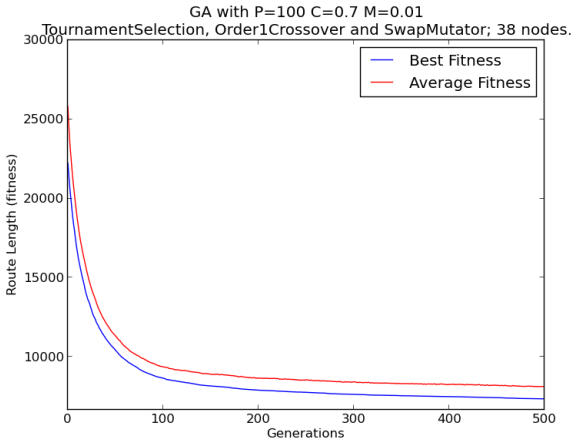


(b) $c = 0.7$

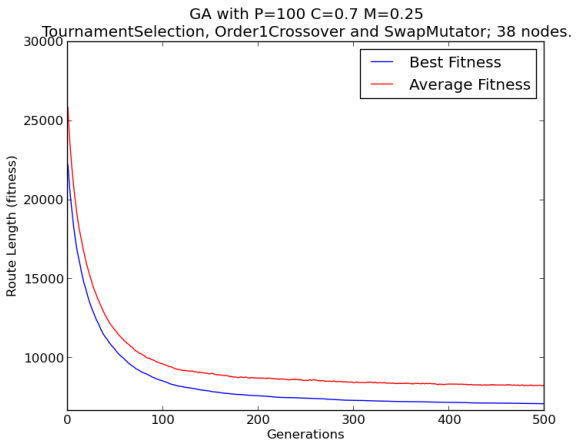


(c) $c = 0.9$

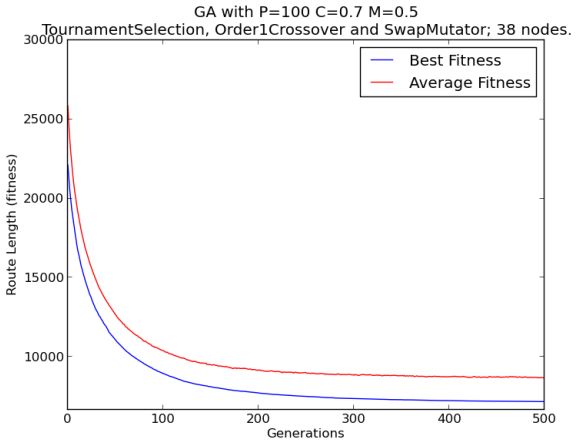
Figure 7: Results on different crossover rate (c) on a data set with 38 nodes. All used a population of 100, mutation rate of 0.01 with the order crossover operator and the swap mutator



(a) $m = 0.01$

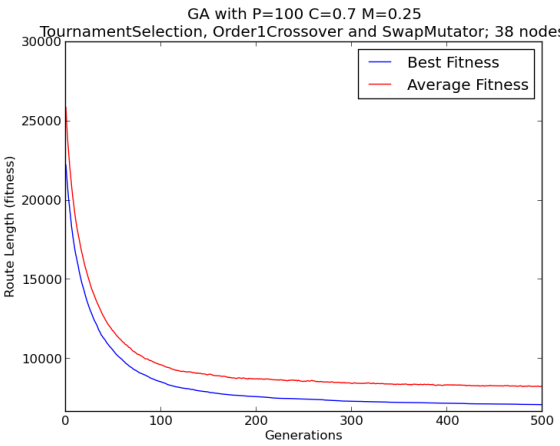


(b) $m = 0.25$

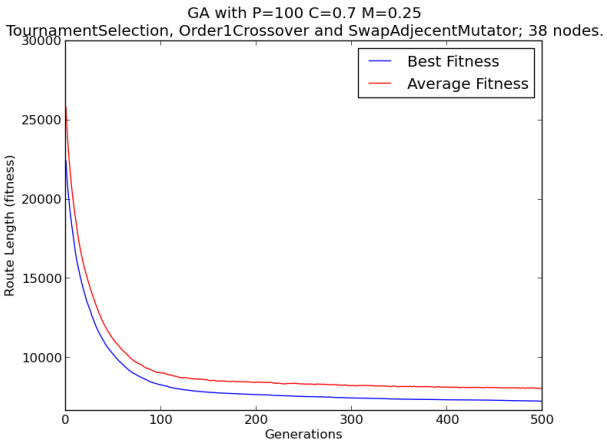


(c) $m = 0.5$

Figure 8: Results on different mutation rate (m) on a data set with 38 nodes. All used a population of 100, crossover rate of 0.7 with the order crossover operator and the swap mutator



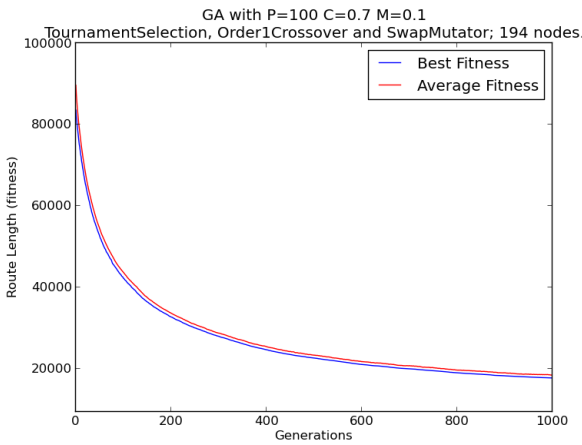
(a) Swap Mutation Operator



(b) Swap Adjacent Mutation Operator

Figure 9: Results on different mutation operators on a data set with 38 nodes. All used a population of 100, crossover rate of 0.7, mutation rate of 0.25 and with the order crossover operator

(a) Cycle Crossover



(b) Order Crossover Operator

(c) M-Crossover Operator

Figure 10: Results on different crossover operators on a data set with 194 nodes. All used a population of 100, crossover rate of 0.7, mutation rate of 0.1 and the swap mutator

4 Conclusions

References

[1] John H. Holland. *Adaptation in natural and artificial systems : an introductory analysis with applications to biology, control, and artificial intelligence.* MIT Press, April 1992.

[2] D. N. Mudaliar and N. K. Modi. Unraveling travelling salesman problem by genetic algorithm using m-crossover operator. In *Signal Processing Image Processing & Pattern Recognition (ICSIPR), 2013 International Conference on*, pages 127–130. IEEE, February 2013.