**University College London**
**Department of Computer Science**

BSc Computer Science – 3rd Year

COMP3091 – Individual Project: 2009 - 10

# Solving Travelling Salesman Problems using Genetic Algorithms

By Peter Tran

MAIN REPORT

30th April 2010

**Supervisor**

Dr. Robin Hirsch

*This report is submitted as part requirement for the BSc Degree in BSc Computer Science at University College London. It is substantially the result of my own work except where explicitly indicated in text.*

## Abstract

This project investigates how Genetic Algorithms can find near – optimal solutions to the *Travelling Salesman Problem*. A Genetic Algorithm works by selecting and combining parts of cheaper solutions from the search space. The Genetic Algorithm is inspired from biological concepts of Natural Selection, Crossover, Mutation, Inheritance, etc.

This project investigates methods for crossing over solutions. These methods are the *Order (OX), Cycle (CX)* and *Partially Mapped (PMX)* Crossover. Also investigate are mutation methods known as *Single Swap (SSM)* and *Inversion (IM)*. The aim is to see which of these *Path* representation methods are better in finding near – optimal solutions.

The investigations were conducted by practical experimentation and involved the construction of a Genetic Algorithm. Experiments were carried out on *bays29*, a benchmark graph representing the geographical distances between 29 cities. Similar work was carried out on *bays29* by Albert Jan Yzelman (University of Utrecht); the near-optimal solutions obtained were similar to my project's results.

The best set of results involved both *OX* and *IM* producing on average solutions within 3.6% of *bays29's* optimum. The *OX* tries to preserve *Ordering of Nodes* instead of *Absolute Position of Nodes* and maintains the concept of *Genetic Linkage* in which *IM* tries to facilitate. It was concluded that operators preserving *Ordering of Nodes* were more effective in contributing to near – optimal solutions as empirical analysis suggest that information about potential good node combinations aren't broken up. As a result there is a tendency for good node combinations to be inherited during crossover which is analogous to the concept of *Genetic Linkage*.

# Table of Contents

## 1. Introduction

The Travelling Salesman Problem (TSP) is an important combinatorial optimization problem. Originally it is about a sales-rep who has to visit a number of cities. Given the set of cities and any chosen starting point the TSP asks what is the best way of visiting every other city once and then returning to the start such that the distance travelled is minimized? The TSP can be represented as a complete graph with asymmetric edges where finding the Hamiltonian Circuit of minimal cost is required.

The problem sounds straight forward and the simplest method is to consider every Hamiltonian Circuit and select the cheapest one. Unfortunately this means that the algorithm runs in factorial time and would be unsuited for graphs with a large number of nodes. No algorithm is known that can solve exactly the TSP in polynomial - time. The TSP is NP – Hard as noted by D.S Johnson **[Page 234, 9]**.

The TSP is an abstract representation of many real – life problems. A scientist studying a crystal will involve moving it into thousands of different orientations to measure the intensity of X-ray diffraction **[Paragraph 1, 5]**. The scientist in question would like to conduct the investigation efficiently and so the ordering of crystal orientations matters. As the optimal solution can't be found in polynomial time settling nearer optimal solutions is advised. Genetic Algorithms are one method and this project will investigate how they can be applied to the TSP.

The Genetic Algorithm can find near-optimal solutions to intractable problems and was first introduced by John Holland in his book *"Adaptation in Natural and Artificial Systems (1975)"*. The fascination about Genetic Algorithms is that they are designed purely with biological concepts in mind.

A Genetic Algorithm runs for a number of generations and for each generation a population of solutions (Chromosomes) is maintained. Each Chromosome has a fitness that determines how good it is. Depending on the Chromosomes' fitness some are selected to undergo crossover to create new solutions (Offspring), the selection process is based Darwin's *"Theory of Natural Selection"*. Offspring inherit parts of their Parents' solution in the hope of generating better solutions. All solutions have a point on a search space which is like a landscape of hills and valleys with the global optimum at the deepest valley. Through

Natural Selection and Crossover alone the solutions would begin to converge onto a particular point on the search space and in most cases this isn't anywhere near the global optimum. Thankfully small alterations to solutions in the population can generate new points in the search space and this method is known as Mutation.

Solving a TSP using Genetic Algorithms requires a particular encoding for a solution. The *Path* representation is preferred which is simply a list of graph nodes in the order they should be visited. The *Path* encoding constrains the design of Crossover methods as there is a requirement that the resulting tour is legal (i.e. a graph node should appear only once in a tour sequence) whilst on the other hand ensuring the resulting tour stores information from its parents about good node orderings, combinations and positions.

The project focuses on 3 types of Crossover methods; they are the Order Crossover (OX), Cycle Crossover (CX) and Partially Mapped Crossover (PMX). These Crossovers have different characteristics.

The OX method constructs Offspring by choosing a sub-tour of one parent and then filling the remainder of the Offspring with nodes from the other parent not in the chosen sub-tour. When filling the remainder of the Offspring the OX method ensures that the node orderings from the other parent are relatively maintained. The OX method places more importance on *"Ordering of Nodes"*.

The CX method constructs an Offspring such that every node *n* and its position *p* come directly from one of its Parents. The CX method places more importance on *"Absolute Positioning of Nodes"*. On the other hand, the PMX method tries to implement Offspring based on both *"Ordering of Nodes"* and *"Absolute Positioning of Nodes"*.

The project also focuses on 2 types of Mutation methods; they are the Single Swap Mutation (SSM) and Inversion Mutation (IM). Single Swap Mutation selects two nodes in a tour and exchanges them. Inversion Mutation randomly selects two cut points and reverses the node orderings between the cut points, IM is analogous to *"Chromosomal Inversion"*.

Further details about these methods as well as the underlying theory and different components of a Genetic Algorithm are documented in Chapter 2 – Background Information and Related Work.

The aims of this project are as follows:

- To determine which Crossover method (OX, PMX and CX) is better in producing nearer optimal solutions to a TSP.
- To determine whether *Ordering of Nodes* or *Absolute Positioning of Nodes* is more important than the other for solving TSPs using Genetic Algorithms.
- To determine which Mutation method (SSM and IM) is better.
- Investigate different Crossover and Mutation probabilities.
- To elucidate design principles for solving TSPs using Genetic Algorithms.

To answer the above question practical experimentation is required. A program has been designed and implemented (Chapter 3 – Design and Implementation) after carrying out the necessary requirements and analysis (Chapter 2 – Requirements and Analysis).

Experiments are to be carried out on benchmark graphs. One graph is called "*bays29"* and is available from the University of Heidelberg's *Combinatorial Research Group*. "*bays29"* is a real representation of the geographical distances between 29 cities of Bavaria, Germany. An advantage of using this graph is that similar work has been carried out on it by Albert- Jan Yzelman from the University of Utrecht, Netherlands. This was helpful in elucidating various design principles of Genetic Algorithm components as there were differences in the programs implemented. Details of Yzelman's work can be found in Chapter 2.

After experiments (Chapter 5 – Results) were carried out conclusions were made (Chapter 6 – Summary and Conclusions). Briefly they are as follows:

- OX performed better than PMX and CX.
- PMX performed better than CX. *Ordering of Nodes* is a good approach for Crossover. This is due to the fact that *Ordering of Nodes* is associated with OX which performed better than PMX and CX.
- For Crossover the biological concept of Genetic Linkage (tendency for certain alleles to be inherited together) is relevant. The OX method can allow Offspring to inherit good linkage of nodes.
- The Crossover Probability should generally be high.
- The Mutation Probability should generally be low.

- Inversion Mutation performed better than Single Swap Mutation.
- For the best parameter settings, on average solutions within 3.6% of the optimum for *bays29* were found. Yzelman's experiments yielded similar results.
- After comparing the Genetic Algorithm structure of both projects, it was felt that the fitness function's role was underestimated. The way a fitness function is designed can have an impact on the population diversity and a Chromosome's selective pressure.

Finally the project is rounded off with recommendations for future work.

**N.B.** Links to references given as **[Location in Resource, No.]**. "No." corresponds to the list number in the Bibliography section which gives full details about the particular reference, "Location In Resource" refers. References are also simply given as **[No.]** in the case of for example full URL address.

# 2. Background Information and Related Work

## 2.1 The Travelling Salesman Problem

### 2.1.0 Why the Travelling Salesman Problem is Hard?

The obvious way to solve the TSP exactly is to consider every permutation of cities, calculate the total cost and select the cheapest route from them. For a large number of cities this method is like finding the needle in a haystack, because the number of steps to solve the TSP exactly increases rapidly as the number of cities increases. If we have ***n*** cities, the number of possible tours is ***n!***

**Table 1 TSP Complexity (Adapted from [Slide 15, 2])**

| Number of Cities | Number of Steps | Time (Example) |
|---|---|---|
| 1 | 1 | 1 microseconds |
| 2 | 2 | 2 microseconds |
| 3 | 6 | 6 microseconds |
| 4 | 24 | 24 microseconds |
| 5 | 120 | 120 microseconds |
| 6 | 720 | 70 milliseconds |
| 7 | 5040 | 5 milliseconds |
| 8 | 40320 | 0.04 milliseconds |
| 9 | 362880 | 0.36 seconds |
| 10 | 3628800 | 3.6 seconds |
| 12 | 479001600 | 7.92 minutes |
| 15 | $1.30 \times 10^{12}$ | 15 days |

No one has found an algorithm that computes the shortest tour of ***n*** cities in polynomial time. TSP is classified as NP-Hard and all known exact methods run in exponential time. Nobody knows how to solve NP-Hard problems in polynomial time **[Slide 17, 2]**. Since no solution can be found in polynomial time, settling for near optimal solutions is advised.

### 2.1.1 The Travelling Salesman Problem as a Graph Problem

We can represent TSP as a weighted graph "G". G = (V, f) where V is a set of vertices and a function *f* such that *f:* V×V → N, so between every pair of vertices there is an

edge "E" of weight ∈ N (note in the project's case real numbers are used). The graph is a complete one, such that an edge connects every pair of vertices **[Description: As a Graph Problem, 7]**.

## 2.2 Hill Climbing, Local and Global Optimum, Search Space

Hill Climbing algorithms find local and global optimums in a search space and are important concepts in GA. In TSP's case the goal of Hill Climbing is to minimize a function because we are trying to find the shortest Hamiltonian Circuit. A cost function is used to measure how good a solution is.

The following example is adapted from **[Paragraph 5, 4]**:

The cost function can be viewed as a landscape and altitude its measure. The landscape is composed of hills and valleys with the global optimum in the deepest valley.

The landscape is a search space holding all possible solutions. Each solution is a single point in this space. Now imagine you are lost after hiking in the dark and you are dehydrating. Your priority is to find a lake which is located downhill. You follow the direction of the steepest descent as it changes in order to lose altitude as quickly as possible. This is a form of adaptive search; information is gained from previous searches to figure out the next starting point to explore.

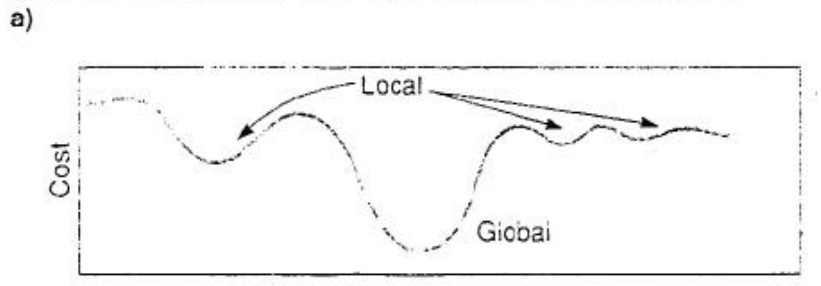



**Figure 1 Local and Global Optimum [4]**

Finding the global optimum is easy if we begin the search in the right area.

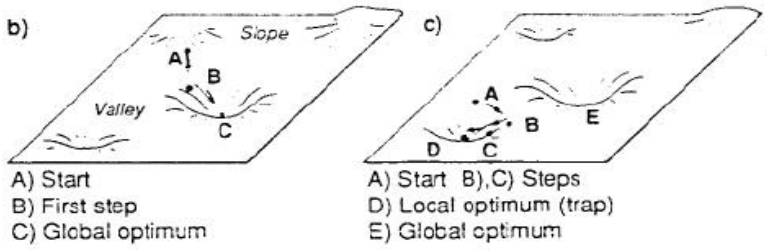



**Figure 2 Trapped in Local Optimum [4]**

In c) starting at "A" following the slope through "B", "C" will lead to the local optimum, "D". The global optimum is at E. In GA, convergence to a local optimum is a common problem.

## 2.3 Genetic Algorithms

### 2.3.0 Genetic Algorithm Structure

Genetic Algorithms are inspired by many biological concepts. GA can be any population model that uses selection, crossover and mutation operators to generate new sample points in a search space **[Paragraph 5, 8]**. Core parameters of a GA include population size, probabilities for copying, crossover and mutation. The probabilities limit how often these evolutionary operators are performed on the current population. A solution to a problem is known as a Chromosome. A GA typically works as follows **[Page 176, 1].**

1. Randomly create an initial population of Chromosomes.
2. Evaluate the fitness of each Chromosome using an appropriate function.
3. Based on each Chromosome's fitness use a selection method to pick Chromosomes into a "Selection Pool". The Chromosomes selected usually have an above average fitness.
4. For every pair of Chromosomes in the "Selection Pool" generate a random real number from [0.0, 1.0]. The cumulative probabilities of copying, crossover and mutation are represented from [0.0, 1.0], based on the generated real number apply the respective method to the pair. Add the result to the "Next Generation Pool".
5. Assign the "Next Generation Pool" to be the current population.
6. Repeat stages 2 – 5 for the newly generated Population until max epochs reached.

### 2.3.1 Biological Concepts

**Table 2 Related Biological Concepts**

| Biological Concepts | Relation to GA for TSP |
|---|---|
| Chromosome | A Chromosome represents a solution to the TSP. The solution is a Hamiltonian Circuit which has associated cost and fitness. |
| Gene | Genes are represented by graph nodes. |
| Loci | Each graph node has a position on the Hamiltonian Circuit. |
| Chromosomal Crossover | Two Hamiltonian Circuits can undergo crossover such that a new circuit is created. Information including node positions and node orderings from BOTH parents is transferred to the generated circuit. |
| Mutation | Changes the node at a particular position on the Hamiltonian Circuit. |
| Natural Selection | Solutions with above average fitness have a greater chance of undergoing crossover and surviving into the next generation |
| Genetic Linkage | In biology this is the tendency for some genes to be inherited together. In GA for TSP this is when nodes connected by edges can remain connected in the new solution generated by crossover. |

### 2.3.2 Selection Operator

Purpose of this operator is to select Chromosomes based on their fitness such that they can pass valuable genetic material into the next generation. Chromosomes can be selected more than once; this is fine as we would like individuals with above average fitness only to undergo the evolutionary process.

Only strictly allowing Chromosomes with higher levels of fitness to be selected causes premature convergence to a local optimum. This is exacerbated when Crossover is applied. The offspring generated will converge to a particular area and any individuals outside this area will experience increased “Selective Pressure”. Selective Pressure is the probability of a Chromosome being chosen compared to the average probability of selection for all Chromosomes **[Selection, 10]**.

Strong and weak selective pressure have implications as noted in **[Page 58 – 59, 9]**:

*“As selective pressure is increased, the search focuses on the top individuals in the population, but because of this exploitation genetic diversity is lost. Reducing selection pressure increases exploration because more genotypes and thus more schemata are involved in the search.”*

A popular method to produce a balanced approach is the Fitness Proportionate Selection (Roulette Wheel Selection).

Roulette Wheel Selection:

Each Chromosome number "$x$" has a fitness value calculate by the function $f_x$. The Chromosome with the cheapest Hamiltonian Circuit Cost has the highest fitness. This is done by finding the Chromosome with the most expensive circuit cost. Then for each Chromosome in the current population subtract their circuit weight from the most expensive one.

The sum of all fitness value for every individual is $\Sigma^{n}_{j=1} f_j$, where "**n**" is the population size. Each individual can have a probability of being selected.

$$selection\_prob_x = f_x / \Sigma^{n}_{j=1} f_j$$

We then map the Chromosomes with their respective cumulative frequencies. The algorithm proceeds to select a number of individuals. This is repeated according to the population size. For each iteration a random real [0.0, 1.0] is generated. For example 0.55 will mean Chromosome 5 is selected, 0.83 will mean Chromosome 6 is selected.

**Table 3 Roulette Wheel**

| **Chromosome Number** | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| **Fitness Value** | 2 | 3 | 1 | 4 | 5 | 5 |
| **Selection Probability** | 0.10 | 0.15 | 0.05 | 0.2 | 0.25 | 0.25 |
| **Cumulative Frequency** | 0.00 | 0.10 | 0.25 | 0.3 | 0.5 | 0.75 |

Elitism

This is when some of the best Chromosomes are automatically copied into the next generation. It is used to preserve the best result at each generation. Elitism is usually required as the best Chromosome at each generation is not guaranteed to be copied; performing crossover on this Chromosome doesn't always produce a better result and may also be destroyed by mutation. The consequences are that the GA will not search the right areas fully.

### 2.3.3 Copy Operator

If Chromosomes in the selection pool don't undergo mutation or crossover, they are automatically moved to the next generation.

### 2.3.4 Crossover Operator

Crossover is primarily responsible for exploring a decreasing part of the search space **[Page 50, 4]**. The offspring produced usually has a higher fitness level than its parents. This is done by extracting best parts of the parent to define the offspring's genetic sequence. Encoding a solution to TSP is an important question. Binary representation is not appropriate **[Page 211, 9]**:

*"Unfortunately there is no practical way to encode TSP as a binary string that does not have ordering dependencies or to which operators can be applied in a meaningful fashion. Simply crossing strings of cities produces duplicates and omissions…… The ideal recombination operator should recombine critical information from parent structures in a non destructive meaningful manner".*

The *Path* encoding for TSP is popular, we can number the nodes and the chromosome would be a list of these nodes in the order they should be visited, the last node in the sequence links directly back to the first node. Crossing over two tours using this encoding can't always produce another valid tour as noted in **[Page 50, 4]**. However a variety of methods have been invented.

Partially Mapped Crossover **(**PMX) **[Page 216, 9]**:

Consider a complete graph with 9 nodes labelled [0, 8] the nodes positions are also from [0, 8].

- Parent 1: (4, 8, 7, 0, 5, 1, 2, 3, 6) (and then back to 4).
- Parent 2: (5, 2, 1, 6, 3, 7, 0, 8, 4) (and then back to 5).

Two random cut points are chosen lets say node positions 2 and 5; a segment for each Parent is created.

- Parent 1: (4, 8, 7, 0, 5, 1, 2, 3, 6)
- Parent 2: (5, 2, 1, 6, 3, 7, 0, 8, 4)

The Offspring are created firstly by swapping the segments between the two parents.

- Offspring 1: (4, 8, 1, 6, 3, 7, 2, 3, 6)
- Offspring 2: (5, 2, 7, 0, 5, 1, 0, 8, 4)

Between the segments of the offspring, a bijective mapping exists for each node position. They are 1⇔ 7, 0 ⇔ 6 and 3 ⇔ 5.

Now for nodes outside the segment, where a node already appears in the sequence this has to be replaced by the corresponding mapping. If this produces a new conflict then use the mapping for that node to replace it. For example in the outer segment of Offspring 1, “3” is in conflict so replace it with “5”, “6” is in conflict so replace it with 0. Carry out the same process with Offspring 2.

The result is as follows, corresponding sub-tours of Offspring and Parents are highlighted.

**Table 4 PMX Crossover Result**

| **Parent 1** | **Parent 2** |
|---|---|
| (4, 8, 7, 0, 5, 1, 2, 3, 6) | (5, 2, 1, 6, 3, 7, 0, 8, 4) |
| **Offspring 1** | **Offspring 2** |
| (4, 8, 1, 6, 3, 7, 2, 5, 0) | (3, 2, 7, 0, 5, 1, 6, 8, 4) |

PMX produces offspring that has node orderings and/or node positions preserved from parents.

Order Crossover (OX) **[Page 217, 9]**:

The Order Crossover states that the ordering of nodes (not their exact positions on the tour) is important. Offspring is created first by preserving the sub-tour from one parent and then by preserving the relative ordering of nodes from the other parent.

Generate two random cut points, say nodes positions 2 and 5 again. Segments are created.

- Parent 1: (4, 8, 7, 0, 5, 1, 2, 3, 6)
- Parent 2: (5, 2, 1, 6, 3, 7, 0, 8, 4)

The segments are directly copied into the offspring as follows.

- Offspring 1: (x, x, 7, 0, 5, 1, x, x, x)
- Offspring 2: (x, x, 1, 6, 3, 7, x, x, x)

For Parent 1 concatenate the following segments in the following order 3rd, 1st and 2nd getting (2, 3, 6, 4, 8, 7, 0, 5, 1) then remove the nodes that appear in Parent 2's 2nd segment generating (2, 4, 8, 0, 5). Fill Offspring 2's blanks starting from the 3rd segment with (2, 4, 8, 0, 5). Similarly we can do the same for Parent 2 and Offspring 1.

**Table 5 OX Crossover Result**

| **Parent 1** | **Parent 2** |
|---|---|
| (4, 8, 7, 0, 5, 1, 2, 3, 6) | (5, 2, 1, 6, 3, 7, 0, 8, 4) |
| 0, 5 | 6, 3 |
| **Offspring 1** | **Offspring 2** |
| (6, 3, 7, 0, 5, 1, 8, 4, 2) | (0, 5, 1, 6, 3, 7, 2, 4, 8) |

Cycle Crossover (CX) **[Page 218, 9]**:

CX builds offspring such that nodes and their positions come directly from one of its parents.

- Parent 1: (1, 2, 3, 6, 0, 7, 8, 4, 5)
- Parent 2: (5, 8, 7, 1, 2, 0, 3, 6, 4)

Offspring 1 is created by firstly taking the node in the 1st position of Parent 1.

- Offspring 1: (1, x, x, x, x, x, x, x, x)

From Parent 1's 1st node position we look directly at the entry Parent 2's 1st node position, it's a 5. So we input this node in Offspring 1 in the position it originally appears in Parent 1.

- Offspring 1: (1, x, x, x, x, x, x, x, 5)

Similarly the process is repeated from the position of node 5 in Offspring 1.

- Offspring 1: (1, x, x, x, x, x, x, 4, 5)

And so on…..

- Offspring 1: (1, x, x, 6, x, x, x, 4, 5).

In the case above, from the position of node 6 in Offspring 1 we look at the node in the corresponding position of Parent 2, it's a **"1"** but this is already in Offspring 1, so the cycle is stopped. To complete the tour simply fill the blank positions with the nodes from the other Parent 2.

- Offspring 1: (1, 8, 7, 6, 2, 0, 3, 4, 5)

Similarly we can get Offspring 2. As you can see positioning of the nodes in the Offspring are exactly the same from either of the parents.

**Table 6 CX Crossover Result**

| **Parent 1** | **Parent 2** |
|---|---|
| (1, 2, 3, 6, 0, 7, 8, 4, 5) | (5, 8, 7, 1, 2, 0, 3, 6, 4) |
| **Offspring 1** | **Offspring 2** |
| (1, 8, 7, 6, 2, 0, 3, 4, 5) | (5, 2, 3, 1, 0, 7, 8, 6, 4) |

### 2.3.5 Mutation Operator

The purpose of Mutation is to create new points in the search space, preventing convergence to a local optimum. The following mutation methods are for the Path representation and have been used for solving the TSP.

Single Swap (SSM)

Parent 1 = (0, 5, 3, 2, 1, 4, 6, 8, 7).

Randomly choose two positions, say 2 and 4 then perform the swap.

Parent 1' = (0, 5, 1, 2, 3, 4, 6, 8, 7).

Insertion

Simply choose a node position (e.g. 6) then insert its entry it in a random place (e.g. position 2), re-align the other nodes appropriately.

Parent 1 = (0, 5, 3, 2, 1, 4, 6, 8, 7).
Parent 1' = (0, 5, 6, 3, 2, 1, 4, 8, 7).

Inversion (IM)

Select randomly two cut points (e.g. positions 2 and 6) and reverse the ordering of the nodes between the two. The cost of the sub tour between these positions is not affected. Swapping the nodes at the ends of the cut point is essentially a "Single Swap" as the cost of the circuit is affected in the same way. Inversion Mutation moves nodes in a segment around the chromosome altering the linkage between parts of the solution and is analogous to Chromosomal Inversion **[Page 50, 4]**.

Parent 1 = (0, 5, 3, 2, 1, 4, 6, 8, 7).
Parent 1'= (0, 5, 6, 4, 1, 2, 3, 8, 7).

## 2.4 How Genetic Algorithms Work?

### 2.4.0 Holland's Schema Theorem

A schema is a template that that defines a subset of strings with similarities at string positions **[Slide 24, 14]**. For example for a graph with 9 nodes labelled from [0, 8], a schema s1 can be:

(**7**, x, x, **4**, x, **3**, x, x, x) which defines a string of length 9 with "7" at position 0, "4" at position 3 and "3" at position 5.

The following strings are instances of s1.

P1 = (**7**, 1, 2, **4**, 0, **3**, 6, 8, 5)  P2 = (**7**, 8, 5, **4**, 2, **3**, 1, 0, 6)  P2 = (**7**, 0, 6, **4**, 1, **3**, 2, 8, 5)

- The *Order* (*s*) of a schema "s" is the number of fixed positions in the template. For example *Order* (*s1*) = 3.
- The defining length *def_len(s)* of a schema "s" is the distance between the first and last specific positions. For example *def_len* (*s1*) = 5.
- The fitness *fit*(*s*) of a schema "s" is the average fitness of all strings matching schema "s".
- The number of strings matching schema "s" at generation "g" is denoted as *num_strings*(*s, g*).
- The average fitness for population at generation "g" is denoted as *ave_fit_pop* (*g*).

Holland's Schema Theorem states that short, low-order, schemata with above-average fitness increase in successive generations. In other words if average fitness for schema's instance is above mean then the number of instances of the schema will grow over time **[Page 118, 15]**. Formally this is given as:

$$num\_strings(s, g+1) \geq [(num\_strings(s, g).\ fit(s)) / ave\_fit\_pop\ (g)] \times [1 - prob\_schema\_dest]$$

Where "*prob_schema_dest*" is the probability that crossover or mutation destroys schema "s". Formally this is given as:

$$prob\_schema\_dest = [[def\_len(s)/\ (length\_ofString - 1)] \times prob\_cross]$$

$$+ Order(s) \times prob\_mute$$

- The probability of crossover is *prob_cross* and the probability of mutation is *prob_mute*. The length of a string is simple *length_ofString* basically number of nodes in TSP graph. A schema "s" with a shorter *def_len(s)* is less likely to be destroyed.

### 2.4.1 The Building Block Hypothesis

Short, low-ordered, above – average schema undergo increased examination in future generations of a GA. As a result strings of higher fitness can be found by applying crossover to instances of above average schema. GA search for near optimal solutions through combining above average schema closely together **[Page 53, 9]**. The Building Block Hypothesis focuses on combining blocks of good solutions together, however this idea has been largely discredited **[Criticism of the Building Block Hypothesis, 9]**.

## 2.5 TSP Benchmarks

To ensure a fair test, my program will be tested benchmark graphs compiled by the "*Combinatorial Research Group*" at the "*University of Heidelberg*", Germany. An appropriate graph is called "*bays29"* **[16]**. The *bays29* graph is a representation of the geographical distances between 29 cities of Bavaria. The mean edge weight between 2 different nodes is 206 and the standard deviation of this is 99. The optimal cost is 2020, the full matrix can be found in the Appendices.

Another suitable graph compiled by the *Combinatorial Research Group* is the *"swiss42"*. The mean edge weight between 2 different nodes is 115 and its standard deviation is 58. The optimal cost is 1273.

## 2.6 Related Work

Similar work has been carried out by Albert Jan Yzelman from the University of Utrecht (Netherlands) **[17]**. The experiments involved *bays29*, Elitism, PMX, Insertion/Single Swap Mutations and Roulette Wheel Selection.

**Table 7 Yzelman's Parameters [Page 23, 17]**

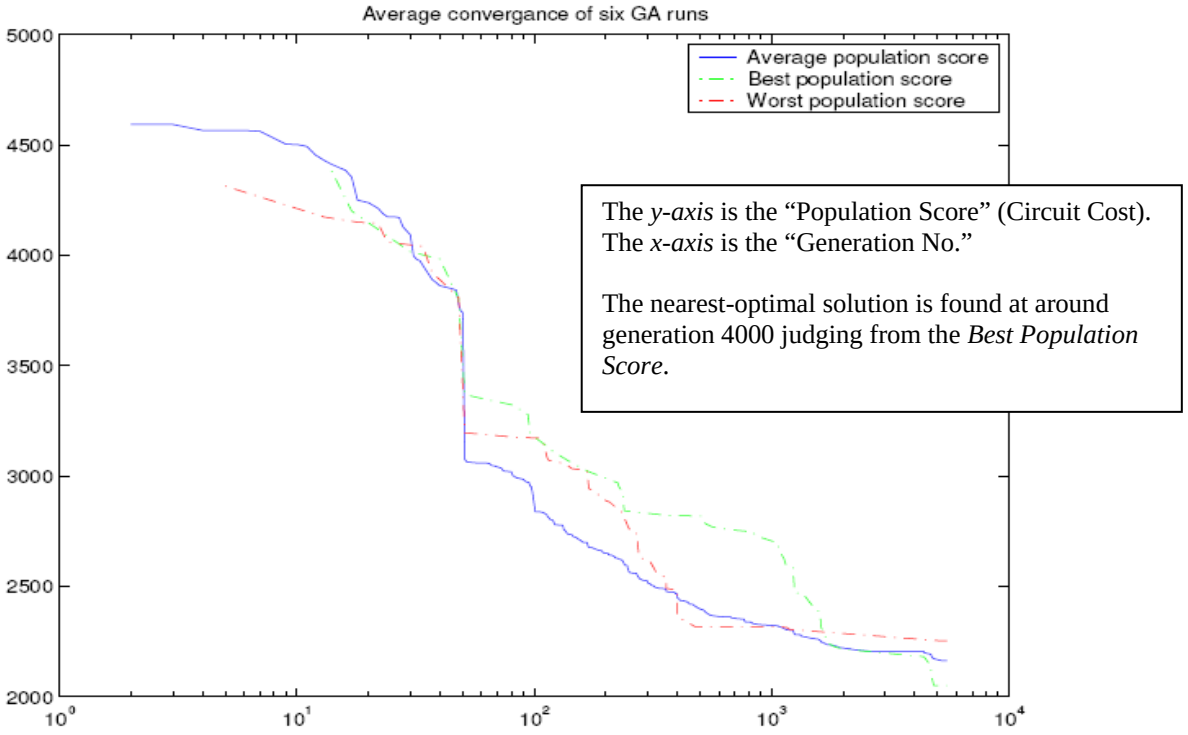
| | |
|---|---|
| **Generations** | 7500 |
| **Population Size** | 250 |
| **Elitism** | 1 |
| **Mutation Probability** | 0.12 |
| **Crossover Probability** | 0.30 |
| **Crossover** | PMX |
| **Mutation** | 33% Swap, 67% Insertion |

Six trials were performed and Cheapest Circuit Costs were: 2050, 2055, 2154, 2215, 2245 and 2253. Figure 3 shows the average convergence and is plotted on the logarithmic scale **[Page 25, 17]**.

**N.B:-**

- The "*Best Population Score*" is a particular GA run where the best near-optimal solution is found, the average circuit cost at each generation is plotted. Hence the "*Worst Population Score*" is the run where the worst near-optimal solution is found. The "*Average Population Score*" is the average circuit cost at each generation for the 6 runs.
- As a result the *Best Population Score* can appear higher than the *Worst Population Score for each generation*.

**Figure 3 Average Convergence of Yzelman's PMX Experiments [Page 25, 17]**



The same parameters in Table 7 were used for another experiment was but this time *Greedy Crossover* **[Page 19, 17]** was used.

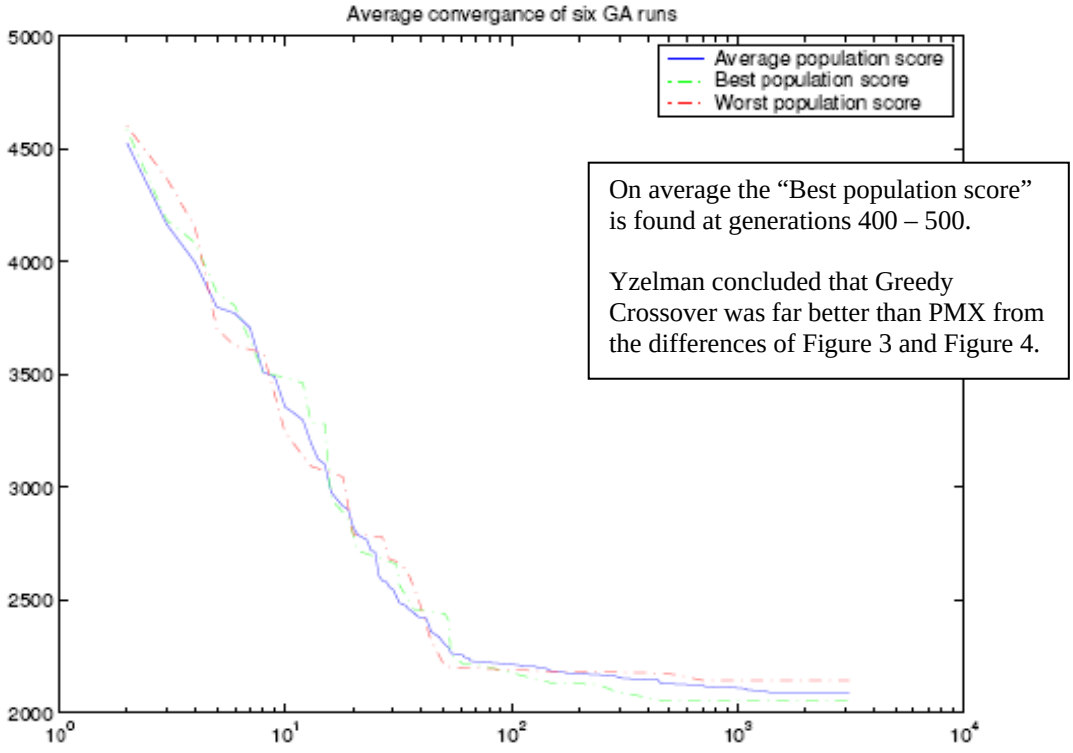
Greedy Crossover

- Constructs a new tour "Offspring 1" firstly by selecting the first node of Parent 1 as the starting point.
- From then on the next node is chosen from either Parent 1 or Parent 2 using the following criteria:

    - For the last node added to Offspring 1. Let *l1* be the location of this node in Parent 1 and let *l2* be the location of this node in Parent 2. Then note the nodes *n1* and *n2* following *l1* and *l2* respectively.
    - If *n1* is already in Offspring 1 then pick *n2* to be the next node in Offspring 1, vice versa.
    - If both *n1* and *n2* are already in Offspring 1, then randomly pick a node that is not currently in Offspring 1 to be the next node in Offspring 1.
    - Else if adding *n1* to the current Offspring 1 tour sequence produces a cheaper tour cost than adding *n2* then add *n1* to be the next node in Offspring 1, vice versa.

Cheapest Circuit Costs were: 2056 (Found at Generation 464), 2068, 2068, 2080, 2120 and 2145. Figure 4 shows the average convergence rates.

**Figure 4 Average Convergence of Yzelman's Greedy Crossover Experiments [Page 24, 17]**




The GA structure is slightly different to the one used in this project (2.4.0). In his version when a Chromosome is selected a decision is made to as whether it should be copied, crossed over or mutated. If the decision is crossover then another Chromosome is selected from the population to perform this operation **[Page 11, 17]**. In this project all Chromosomes will be selected into a "Selection Pool" then operators are applied for every pair in this pool.

My project will look at the OX, PMX and CX crossovers with Single Swap and Inversion mutations. The GA to be implemented will use Elitism.

## 2.7 Literature Study

The publication *"Genetic Algorithms for the Travelling Salesman Problem: A Review of Representation and Operators* **[18]***"* reports attempts made to solve the TSP using Genetic Algorithms. Similar results will be noted in the Conclusions section of the report. The review details various Crossover and Mutation operators for the Path representation.

# 3. Requirements and Analysis

## 3.1 Requirements

### 3.1.0 Functional Requirements

The following are functional requirements needed for the program in order to provide a basis for my study.

Note:

- **Must Have**: Requirements essential to the program.
- **Should Have:** Important requirements that may be omitted.
- **Could Have:** Requirements that are optional (if time is available to implement).
- **Won't Have:** Desirable requirements but preferred for future developments of the system.

**Table 8 Functional Requirements**

| ID | Description | Priority |
|---|---|---|
| 1 | The program shall perform the GA cycle specified in 2.4.0. | Must Have |
| 2 | The program shall perform the Evolutionary operations mentioned in 2.4. | Must Have |
| 3 | The program shall allow the User to specify the following input parameters:<br>TSP Graph File, Population Size, Copy Rate, Crossover Rate, Mutation Rate, Number of Generations required and Crossover Type. | Must Have |
| 4 | The program shall be able to read a Graph presented as a 2D matrix. | Must Have |
| 5 | The program shall have Path Crossovers for TSP. | Must Have |
| 6 | The program shall print out all Chromosomes' information in a population for every generation. | Must Have |
| 7 | The program shall print out the statistics about every Generation (e.g. Cheapest Circuit Length, Average Circuit Length) | Must Have |
| 8 | The program shall perform schemata analysis. | Won't Have |
| 9 | The program shall perform on symmetric TSP graphs. | Must Have |

### 3.1.1 Non - Functional Requirements

These are basically constraints on the program.

**Table 9 Non Functional Requirements**

| ID | Description | Priority |
|---|---|---|
| 1 | The program shall be implemented in Java. | Should Have |

| | | |
|---|---|---|
| 2 | The program shall be operated through an Integrated Development Environment (IDE). | Must Have |
| 3 | The program shall be easy to copy and paste the output onto a spreadsheet. Hence the requirement for an IDE. | Must Have |
| 4 | The program shall allow easy configuration of GA parameters. | Must Have |
| 5 | The program shall be operated from Netbeans IDE. | Should Have |
| 6 | The program shall be tested on TSP Benchmarks. Subsequently the original format of the Benchmark graphs needs to be formatted appropriately. | Must Have |
| 7 | The program shall be architected to be extensible; it should be easy to replace evolutionary operators. | Must Have |
| 8 | The program shall be appropriate for graphs of maximum size 50 nodes. | Must Have |

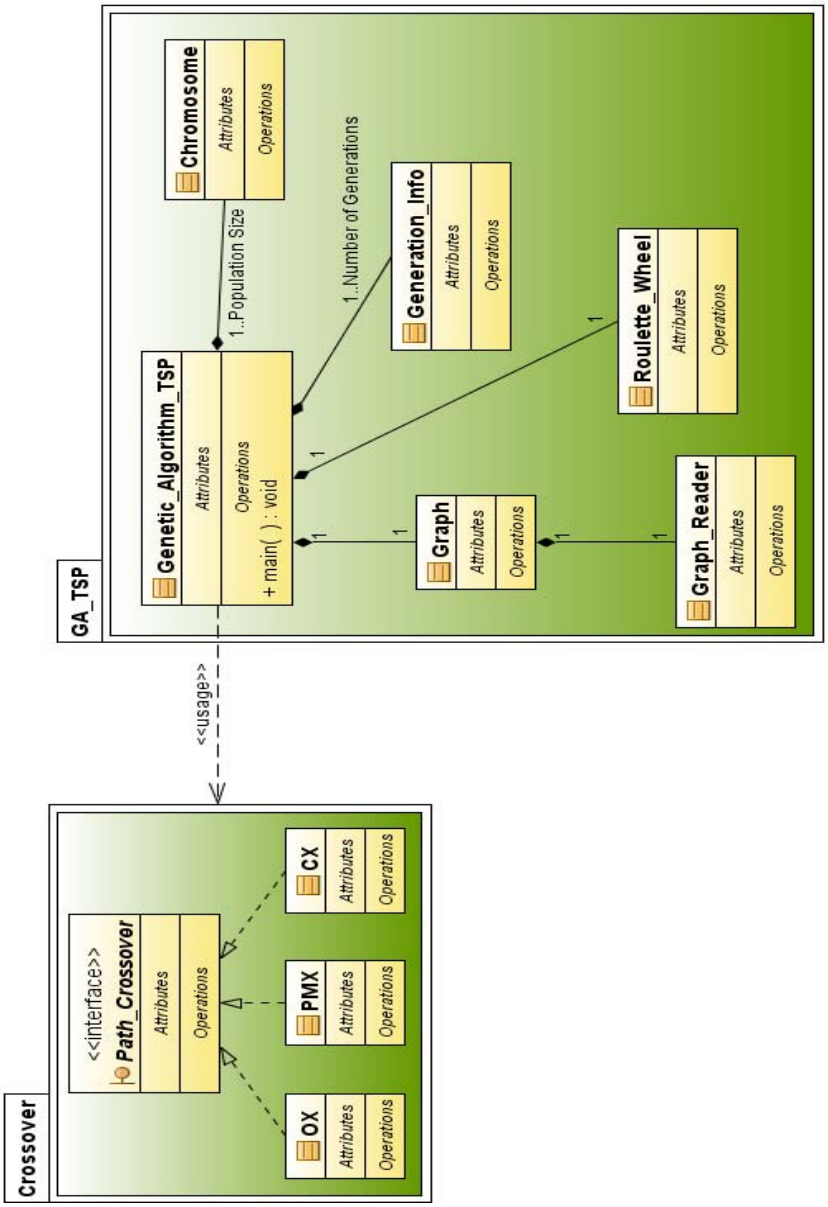
## 3.2 Object Oriented Analysis

### 3.2.0 Initial Domain Model

The following domain model has been deduced. It shows the Classes required and the relationships between them.

- The Crossover package contains the interface Path_Crossover which will specify a set of public features for implementing classes to specialise. There are many types of encoding for using GA on TSP such as the ordinal and adjacency representation **[Page 212 – 216, 9]** but this project will only implement crossovers for path encoding.
- The user will specify the crossover used. So a method with return type Path_Crossover will return the object representing the crossover that was called for. This is an advantage of interfaces, it saves writing a method for every type of Crossover in Genetic_Algorithm_TSP, thus making the program shorter and extensible.
- The main class Genetic_Algorithm_TSP will import the Crossover package.
- Class Roulette_Wheel can be replaced by another selection class when required.

**Figure 5 Initial Domain Model for Analysis**




### 3.2.1 Classes, Responsibilities and Collaborations Analysis

This technique was used to understand the Classes' responsibilities. Collaborators of a particular Class are basically Classes required that provide the functionality in order for it operate as specified. The technique is preparation for Design and Implementation. Results of this process are in the Appendices section under Requirements and Analysis – CRC.

# 4. Design and Implementation

## 4.1 UML Class Diagram

### 4.1.0 Crossover Package

The UML diagram for the classes and interfaces making up the Crossover Package are shown on the next page. This package is common with all Genetic Algorithms created in the project. Some methods are not shown which the "crossOver (int [] offspring, int [] parentX, int [] parentY)" calls to carry out necessary operations.

To create "offspring1" the following is called:

- .. crossOver (offspring1, parent1, parent2). The ordering of parameter passing matters.

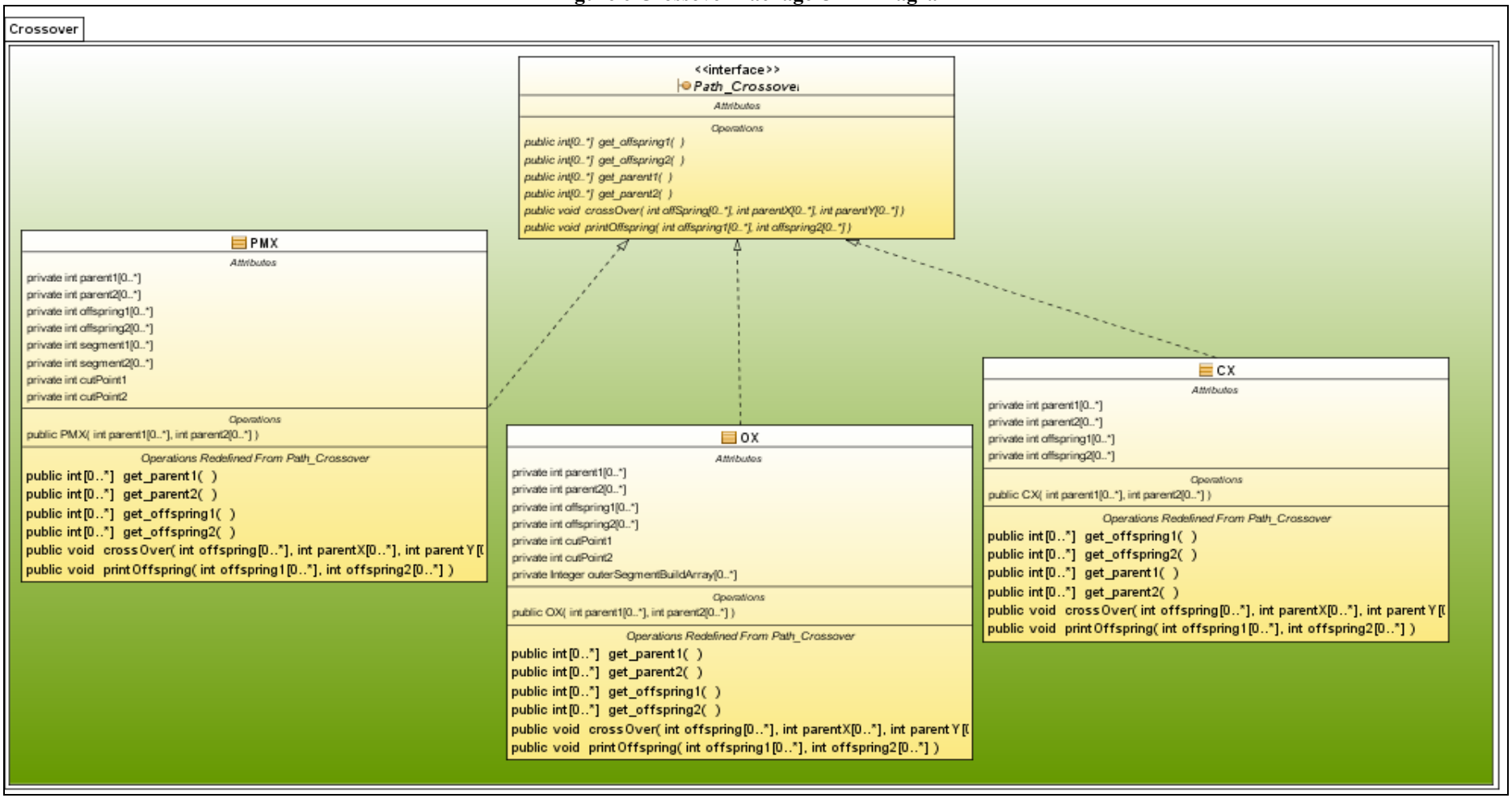
Similarly "offspring2" is created by:

- .. crossOver (offspring2, parent2, parent1). The ordering of parameter passing matters.

The above two method procedures are actually called in the constructers of a Path_Crossover type (e.g. OX). So it is assumed for example ox_object: OX has been instantiated by calling OX ox_object = new OX (chrom1.getHamiltonianCircuit (), chrom2.getHamiltonianCircuit ()) where chrom1 and chrom2 are object references of Chromosome. The "getHamiltonianCircuit ()" returns the path encoding of a TSP solution which is an integer array type.

The constructer instantiates "parent1" and "parent2" to chrom1.getHamiltonianCircuit () and chrom2.getHamiltonianCircuit () respectively, then instances "offspring1" and "offspring2" are created in the constructor. Therefore in Genetic_Algorithm_TSP, ox_object.getOffspring1 () and ox_object.getOffspring2 () will only need to be called to get both offspring produced when crossover is required.

**Figure 6 Crossover Package UML Diagram**



### 4.1.1 GA_TSP Package

**Figure 7 GA_TSP Package UML Diagram**

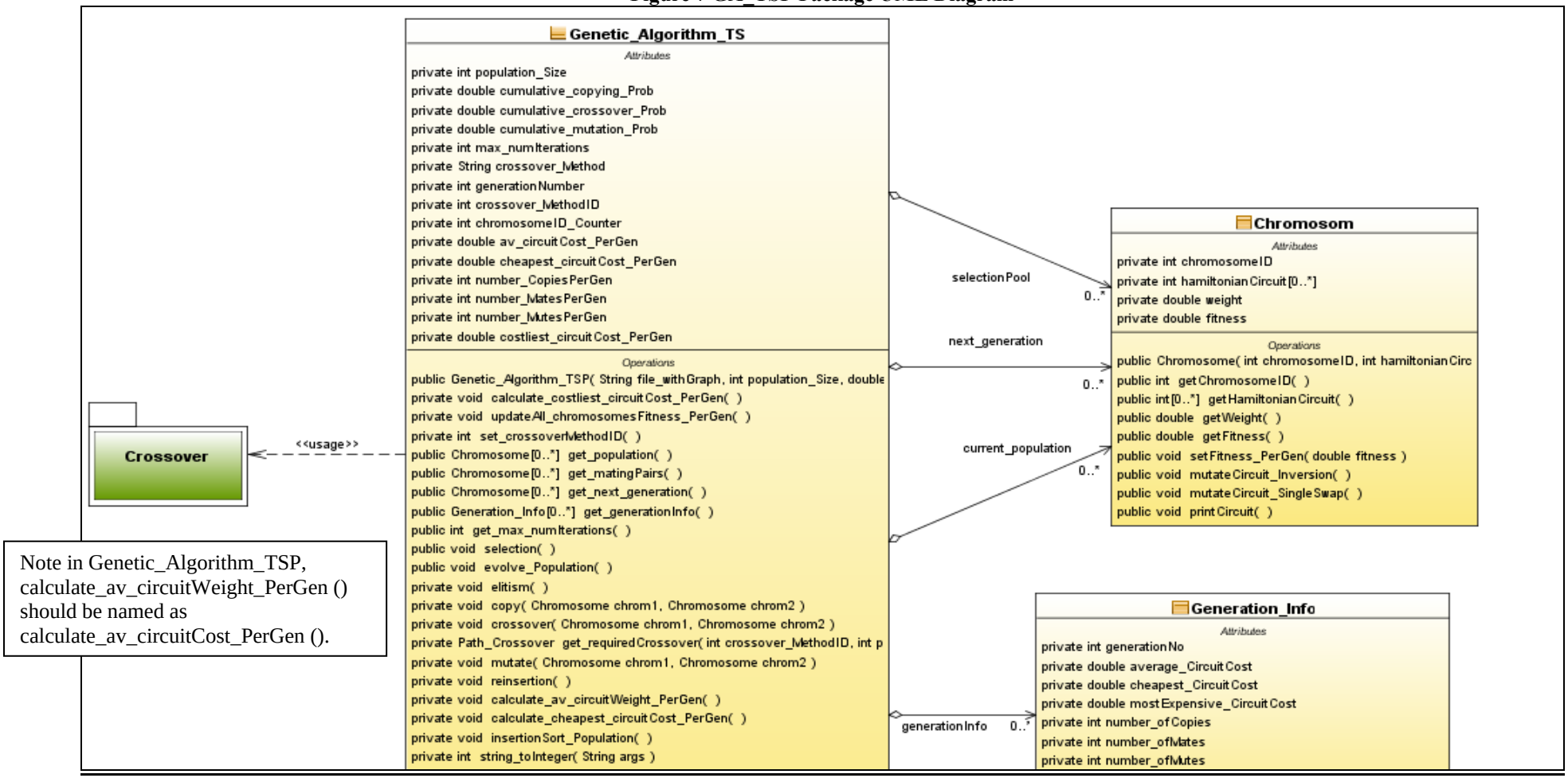
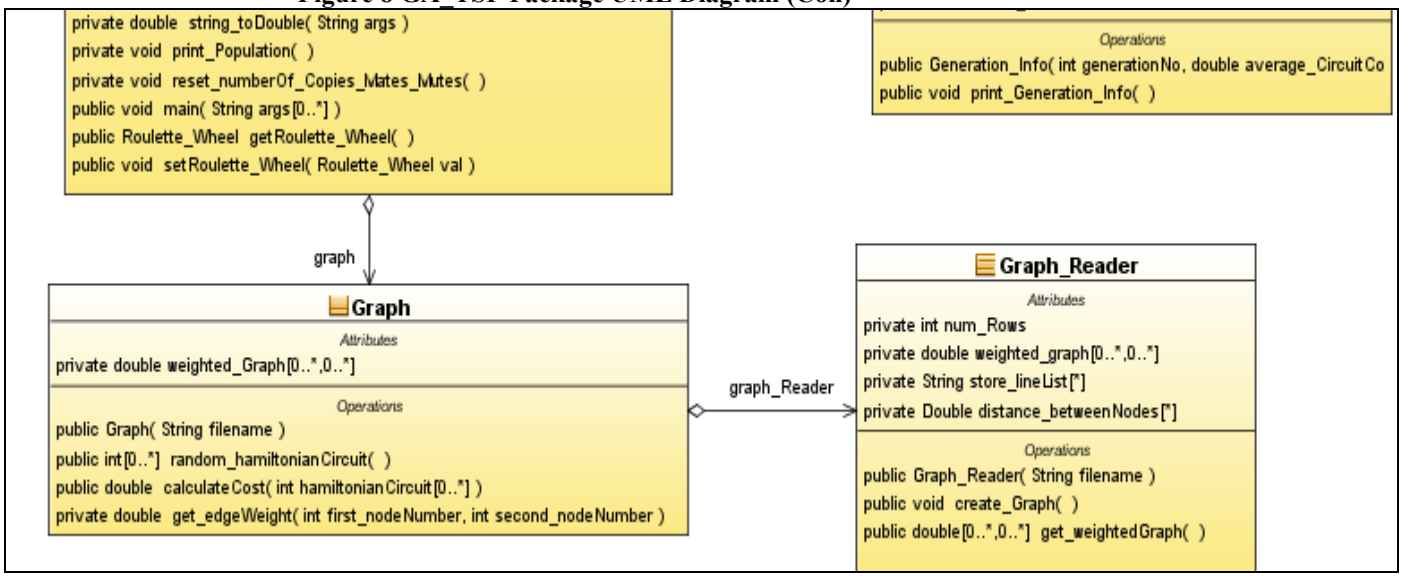


**Figure 8 GA_TSP Package UML Diagram (Con)**



The UML diagram is for GA_Final_RWS_Elitismv1.1 which will be used for conducting experiments.

A chromosome object chrom: Chromosome is initially instantiated when solutions are created through the initial population or crossover. The fitness is initially set to 0.0. After all Chromosomes have been determined for the next generation's population, a Chromosome's fitness is calculated by subtracting its circuit weight from the costliest circuit weight. The costliest circuit weight is calculated by sorting (insertion sort) the population: ArrayList<Chromosome> in descending order in terms of the Chromosome's circuit weight.

GA_RWS_Elitismv1.1 requires odd population sizes *n*. After the best Chromosome is saved the next generation, *n - 1* Chromosomes will be chosen to make up the selection pool. The method evolve_Population () is applied to the selection pool. For every pair of chromosomes in this pool either the copy, crossover or mutation operator is applied. The advantage for applying operators to every pair of Chromosomes is that should crossover be required local areas of 2 search points (possibly with above average fitness) can be searched. This could increase the speed in which a near optimal solution is found.

## 4.2 Unit Testing

TestNG plug-in for Netbeans IDE was used to carry some tests. For example, ensuring that the correct cost is calculated for a Hamiltonian Circuit and ensuring the 2D array dimensions is correct for the graph specified. This was achieved by comparing the calculations on input with the known result using a set of predicates supplied by the TestNG. This predicate should return "true" (or be highlighted green) to indicate a passed test and return "false" (or be highlighted green or red) otherwise.

Another important technique was static inspection of code and output of methods (system print outs). This was particularly helpful in validating the correctness of crossover and mutation operators. Some Crossover classes had additional methods to provide information on where the cut points were generated such that the operations can be traced by hand.

Static inspection of code and system output was essential as most of the GA operators operate randomly (e.g. Roulette Wheel Selection or Mutation) i.e. using TestNG is difficult.

Detailed information about the Unit Testing process is available in the “Unit Testing” section of the Appendices.

GA_Final_RWS_Elitismv1.1 was a result of extensive refactoring of earlier versions and as a result errors in previous versions were noted before designing and implementing the final program.

# 5. Results

## 5.1 Experiments

**N.B**:

- GA_Final_RWS_Eltismv1.1 was used.
- 6 trials were conducted for each experiment so that some comparisons made with Yzelman's experiments.
- Graphs:
  - *Average Cheapest Circuit Cost*: Average cost of the cheapest circuit over the 6 populations for each generation.
  - *Average Costliest Circuit Cost*: Average cost of the costliest circuit over the 6 populations for each generation.
  - *Average Circuit Cost*: Average of the 6 populations' average circuit cost for each generation.
  - *Optimal Circuit Cost*: The global optimal solution for the TSP instance.

### 5.1.0 PMX vs. CX vs. OX on *bays29*

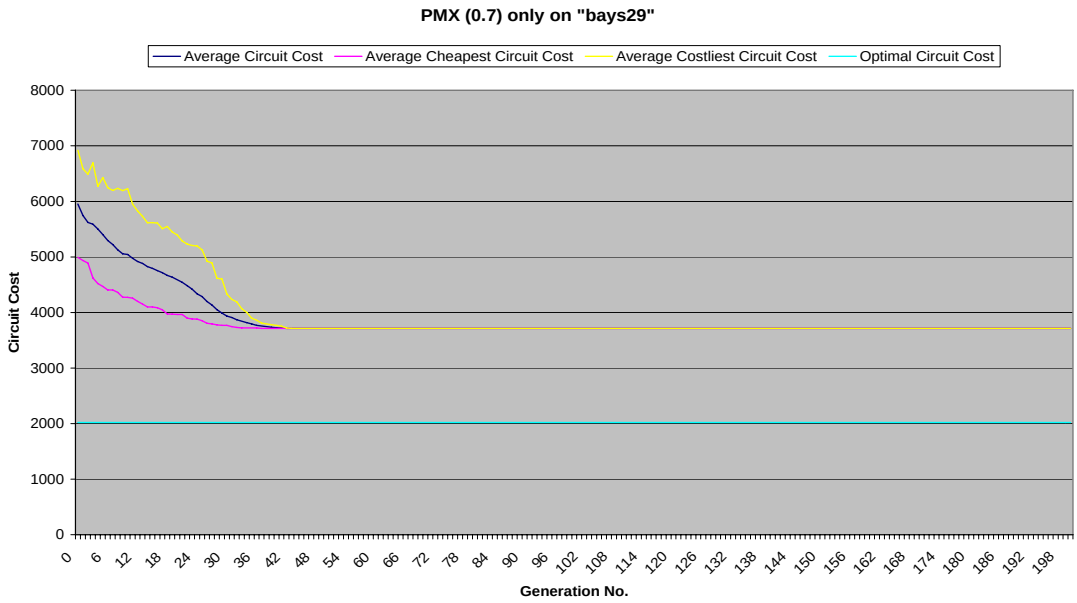
1. PMX (Probability 0.7) only on *bays29*

**Table 10 Parameters for 5.1.0 - 1**

| | |
|---|---|
| **Graph** | bays29 |
| **Population Size** | 61 |
| **Elites** | 1 |
| **Copy Probability** | 0.3 |
| **Crossover (PMX) Probability** | 0.7 |
| **Mutation Probability** | 0.0 |
| **Generations** | 200 |

The cheapest circuit costs obtained from 6 trails were 3897, 3252, 3908, 3723, 3721 and 3780. The average convergence rates are shown in Figure 9. The population converges after 45 generations and no further evolution takes place.

**Figure 9 (5.1.0 – 1) Average convergence of 6 GA runs**



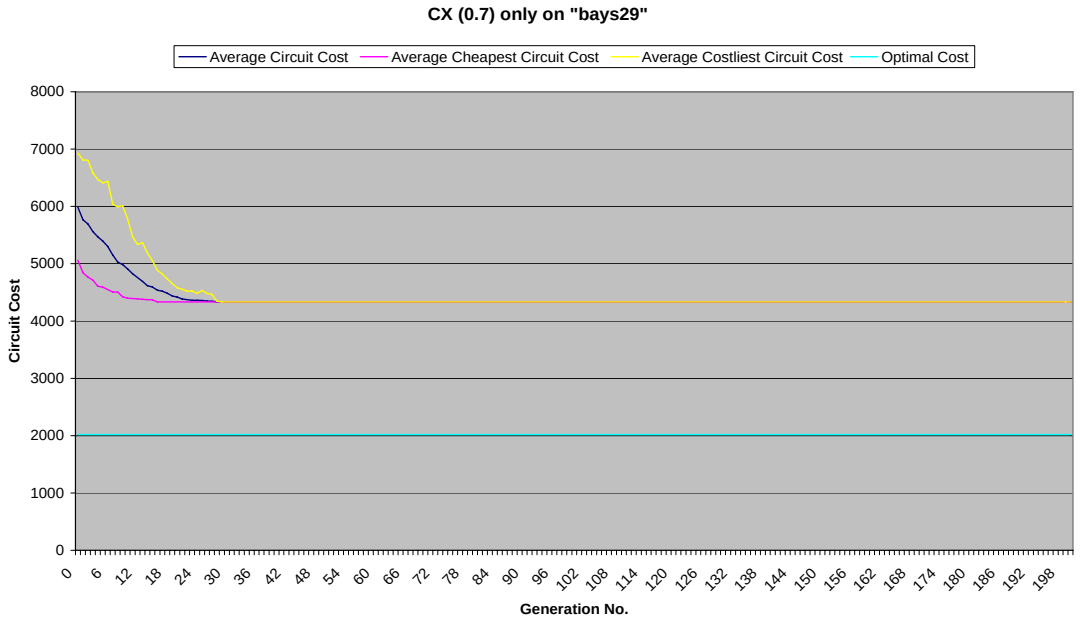

2. CX (Probability 0.7) only on *bays29*

**Table 11 Parameters for 5.1.0 - 2**

| | |
|---|---|
| **Graph** | bays29 |
| **Population Size** | 61 |
| **Elites** | 1 |
| **Copy Probability** | 0.3 |
| **Crossover (CX) Probability** | 0.7 |
| **Mutation Probability** | 0.0 |
| **Generations** | 200 |

The cheapest circuit costs obtained from 6 trails were 4603, 4319, 4510, 4135 and 4541. The average convergence rates are shown in Figure 10. The population converges after 30 generations and no further evolution takes place.

**Figure 10 (5.1.0 – 2) Average convergence of 6 GA runs**




Compared to the PMX method, the CX seems less effective. A reason is that in some cases when two parents undergo CX the offspring produced are actually copies of their parents. As a result the trials on average produce a faster convergence.

For example (in a different graph):

- Parent 1: (4, 2, 3, 6, 5, 1, 8, 0, 7) -> Offspring 1: (4, 2, 3, 6, 5, 1, 8, 0, 7)
- Parent 2: (7, 5, 0, 1, 3, 4, 2, 6, 8) -> Offspring 2: (7, 5, 0, 1, 3, 4, 2, 6, 8)

3. OX (Probability 0.7) only on *bays29*

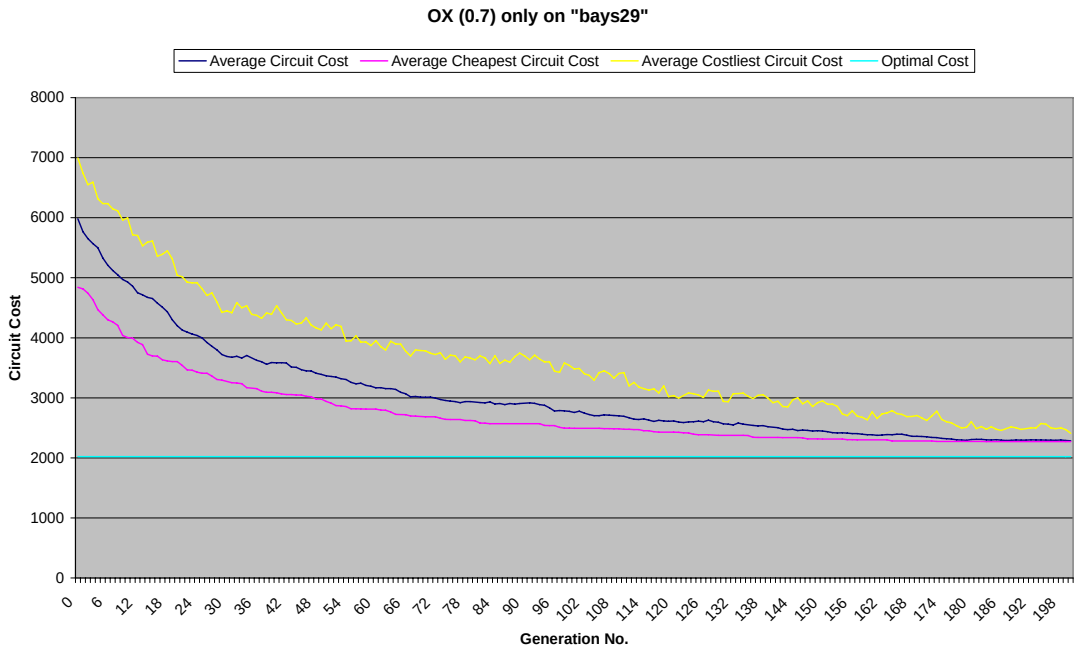
**Table 12 Parameters for 5.1.0 - 3**

| | |
|---|---|
| **Graph** | bays29 |
| **Population Size** | 61 |
| **Elites** | 1 |
| **Copy Probability** | 0.3 |
| **Crossover (OX) Probability** | 0.7 |
| **Mutation Probability** | 0.0 |
| **Generations** | 200 |

The cheapest circuit costs obtained from the 6 trials were 2262, 2397, 2172, 2378, 2278 and 2146. The average cheapest circuit cost of these 6 trials is 2272 and the standard deviation comes to 102. The average convergence rates are shown in Figure 11. The

population hasn't converged after 200 generations, so therefore a few more iterations will be required. Greater number of generations will be considered when different OX rates are tested out.

**Figure 11 (5.1.0 – 3) Average convergence of 6 GA runs**




### 5.1.1 Experimentation of OX Probabilities on *bays29*

In these experiments the OX method was examined further with different crossover probabilities to see the possible effects on producing near – optimum solutions.

1. OX (Probability 0.6) only on *bays29*

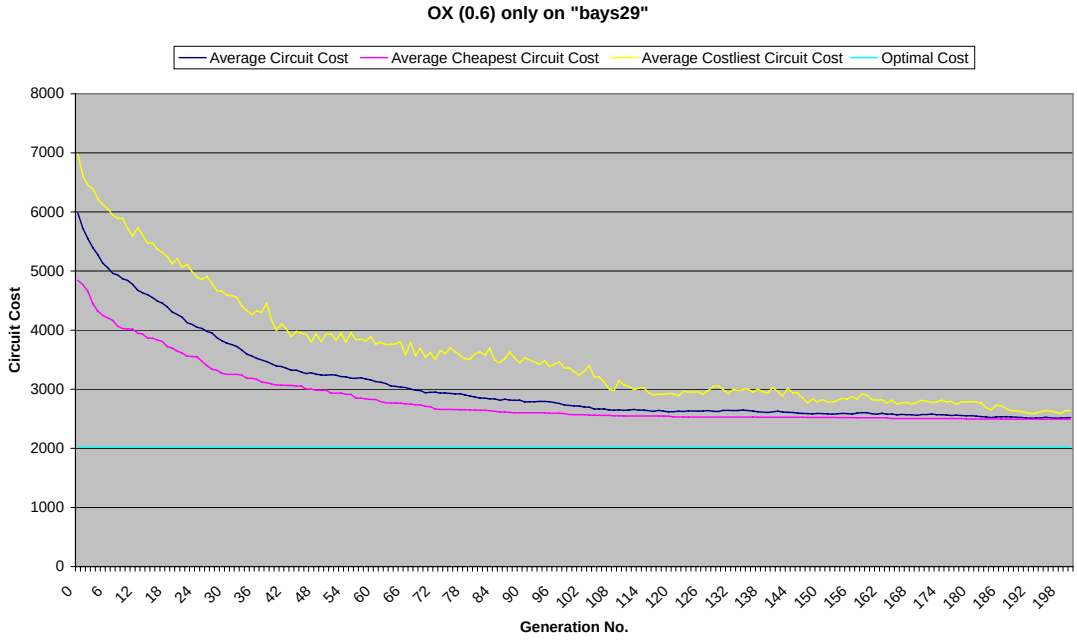
**Table 13 Parameters for 5.1.1 - 1**

| | |
|---|---|
| **Graph** | bays29 |
| **Population Size** | 61 |
| **Elites** | 1 |
| **Copy Probability** | 0.4 |
| **Crossover (OX) Probability** | 0.6 |
| **Mutation Probability** | 0.0 |
| **Generations** | 200 |

The cheapest circuit costs obtained from the 6 trials were 2675, 2544, 2299, 2608, 2471 and 2370. The average cheapest circuit cost of these 6 trials is 2495 and the standard

deviation comes to 143. The average convergence rates are shown in Figure 12. There is slightly more convergence during the last 10 iterations than OX at 0.7.

**Figure 12 (5.1.1 – 1) Average convergence of 6 GA runs**



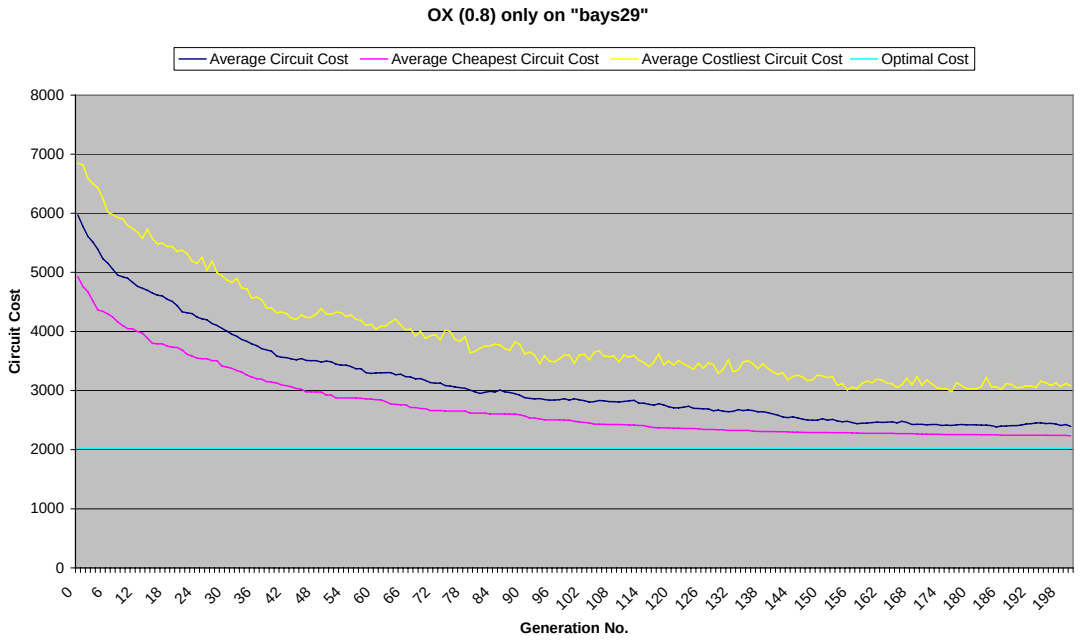

2. OX (0.8) on *bays29*

**Table 14 Parameters for 5.1.1 - 2**

| | |
|---|---|
| **Graph** | bays29 |
| **Population Size** | 61 |
| **Elites** | 1 |
| **Copy Probability** | 0.2 |
| **Crossover (OX) Probability** | 0.8 |
| **Mutation Probability** | 0.0 |
| **Generations** | 200 |

The cheapest circuit costs obtained from the 6 trials were 2150, 2339, 2409, 2219, 2178 and 2107. The average cheapest circuit cost of these 6 trials is 2234 and the standard deviation comes to 117. The average convergence rates are shown in Figure 13. There is less convergence than OX at 0.6 and 0.7. This is owed to the increase in population diversity which is evident from the wider gap between average cheapest circuit cost and average costliest circuit cost for generations 150 onwards.

**Figure 13 (5.1.1 – 2) Average convergence of 6 GA runs**



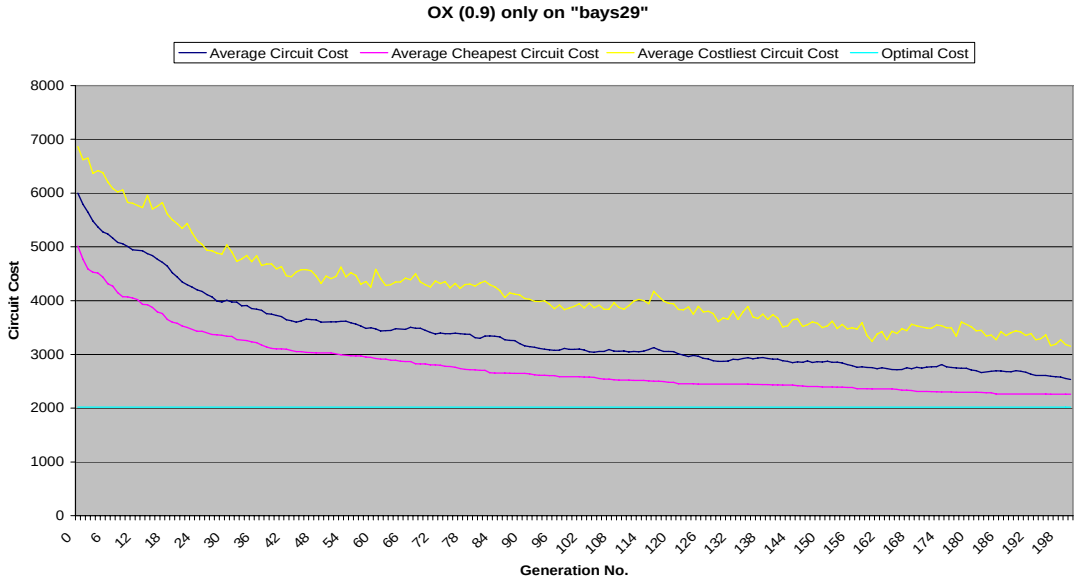

3. OX (0.9) on *bays29*

**Table 15 Parameters for 5.1.1 - 3**

| | |
|---|---|
| **Graph** | bays29 |
| **Population Size** | 61 |
| **Elites** | 1 |
| **Copy Probability** | 0.1 |
| **Crossover (OX) Probability** | 0.9 |
| **Mutation Probability** | 0.0 |
| **Generations** | 200 |

The cheapest circuit costs obtained from the 6 trials were 2201, 2350, 2460, 2090, 2238 and 2261. The average cheapest circuit cost of these 6 trials is 2261 and the standard deviation comes to 128. The average convergence rates are shown in Figure 14. The population diversity is more about similar to OX at 0.8.

**Figure 14 (5.1.1 – 3) Average convergence of 6 GA runs**



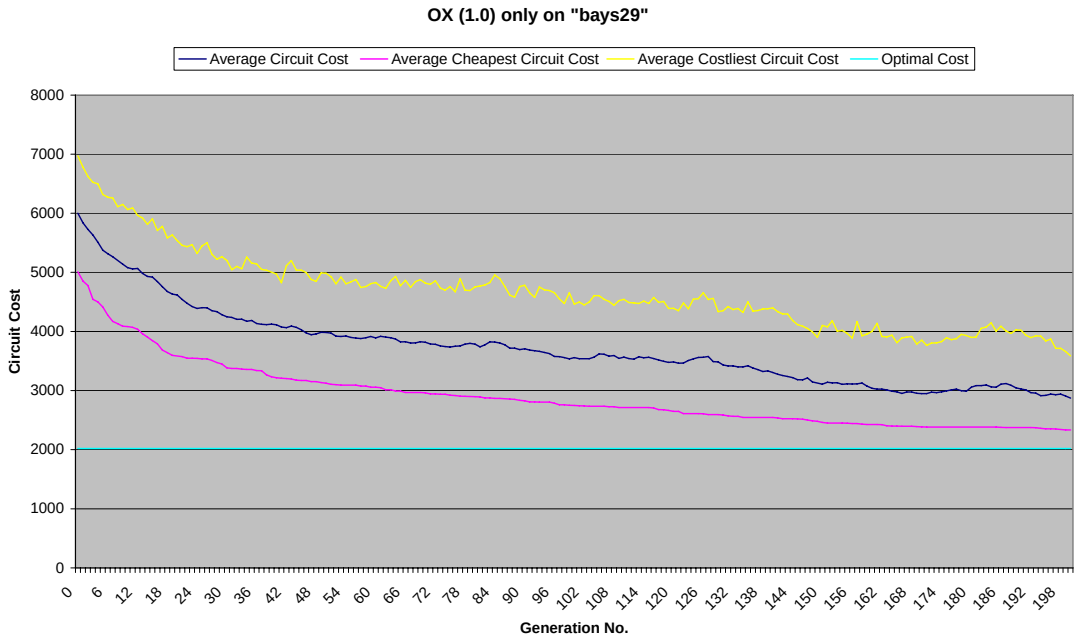

4. OX (1.0) on *bays29*

**Table 16 Parameters for 5.1.1 - 4**

| | |
|---|---|
| **Graph** | bays29 |
| **Population Size** | 61 |
| **Elites** | 1 |
| **Copy Probability** | 0.0 |
| **Crossover (OX) Probability** | 1.0 |
| **Mutation Probability** | 0.0 |
| **Generations** | 200 |

The cheapest circuit costs obtained from the 6 trials were 2178, 2187, 2264, 2464, 2480 and 2425. The average cheapest circuit cost of these 6 trials is 2333 and the standard deviation comes to 140. The average convergence rates are shown in Figure 15. There is far greater population diversity here than OX at 0.8 and 0.9, this is due to no copying.

**Figure 15 (5.1.1 – 4) Average convergence of 6 GA runs**




Which OX probability is better?

The best set of results came with probabilities 0.8 and 0.9 yielding average cheapest circuit costs of 2117 and 2133 respectively. Since OX at 0.8 had the lower standard deviation, it was decided that in the attempt for searching near – optimal solutions for *bays29* the OX probability shall be based at 0.8. In general it appears that OX at 0.8 and 0.9 is suitable as the other rates failed to produce better results.

### 5.1.2 OX with Mutation on *bays29*

Purpose of these tests is to examine the effects of 2 mutation types. Mutation probabilities of 5% and 1% will be compared. Crossover probability will be altered to accommodate the changes.

1. OX (0.65) and Single Swap Mutation (0.05) on *bays29*

**Table 17 Parameters for 5.1.2 - 1**

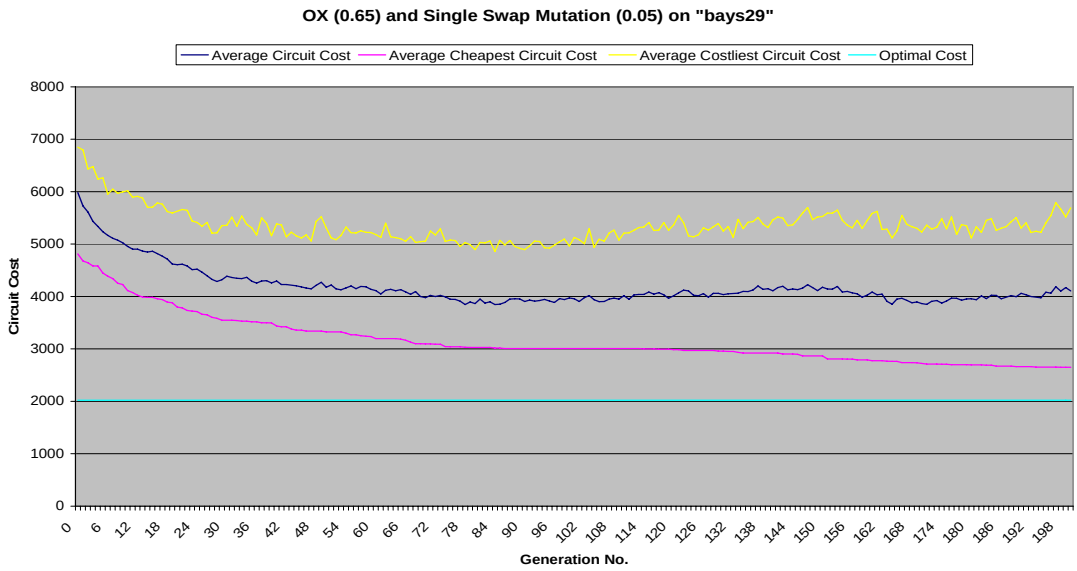
| | |
|---|---|
| **Graph** | bays29 |
| **Population Size** | 61 |
| **Elites** | 1 |
| **Copy Probability** | 0.3 |
| **Crossover (OX) Probability** | 0.65 |

| | |
|---|---|
| **Mutation Probability (Single Swap)** | 0.05 |
| **Generations** | 200 |

The cheapest circuit costs obtained from the 6 trials were 2657, 2283, 2702, 2715, 3134 and 2401. The average cheapest circuit cost of these 6 trials is 2649 and the standard deviation comes to 296. The average convergence rates are shown in Figure 16.

As you can see a mutation rate of 0.05 produces far too much population diversity, the better Chromosomes are disrupted. This is evident from the averages for circuit cost and costliest circuit cost not moving downhill. Compared with just OX at 0.6 and 0.7, this experiment produces poor results with the standard deviation being significantly higher.

**Figure 16 (5.1.2 – 1) Average convergence of 6 GA runs**



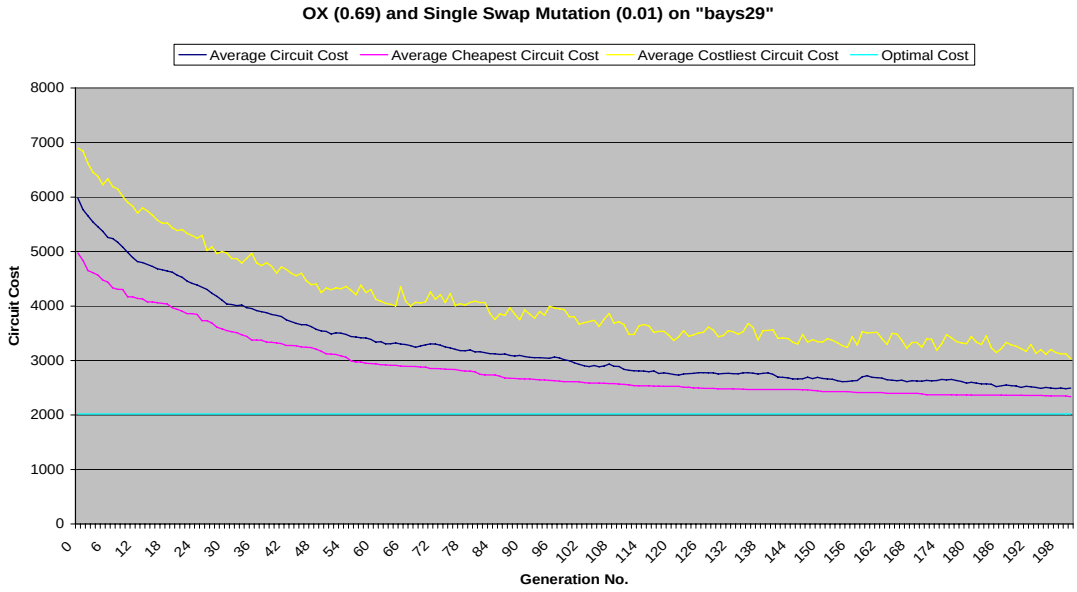

2. OX (0.69) and Single Swap Mutation (0.01) on *bays29*

**Table 18 Parameters for 5.1.2 - 2**

| | |
|---|---|
| **Graph** | bays29 |
| **Population Size** | 61 |
| **Elites** | 1 |
| **Copy Probability** | 0.3 |
| **Crossover (OX) Probability** | 0.69 |
| **Mutation Probability (Single Swap)** | 0.01 |
| **Generations** | 200 |

The cheapest circuit costs obtained from the 6 trials were 2379, 2267, 2396, 2294, 2257 and 2431. The average cheapest circuit cost of these 6 trials is 2337 and the standard deviation comes to 74 (lower than SSM at 0.1, thereby reflecting less volatility). The average convergence rates are shown in Figure 17. As you can see a SSM rate at 0.1 is preferable as the population diversity is under control. It produces slightly higher circuit costs than OX at 0.7 but this was expected.

**Figure 17 (5.1.2 – 2) Average convergence of 6 GA runs**




3. OX (0.65) with Inversion Mutation (0.05) on *bays29*

**Table 19 Parameters for 5.1.2 - 3**

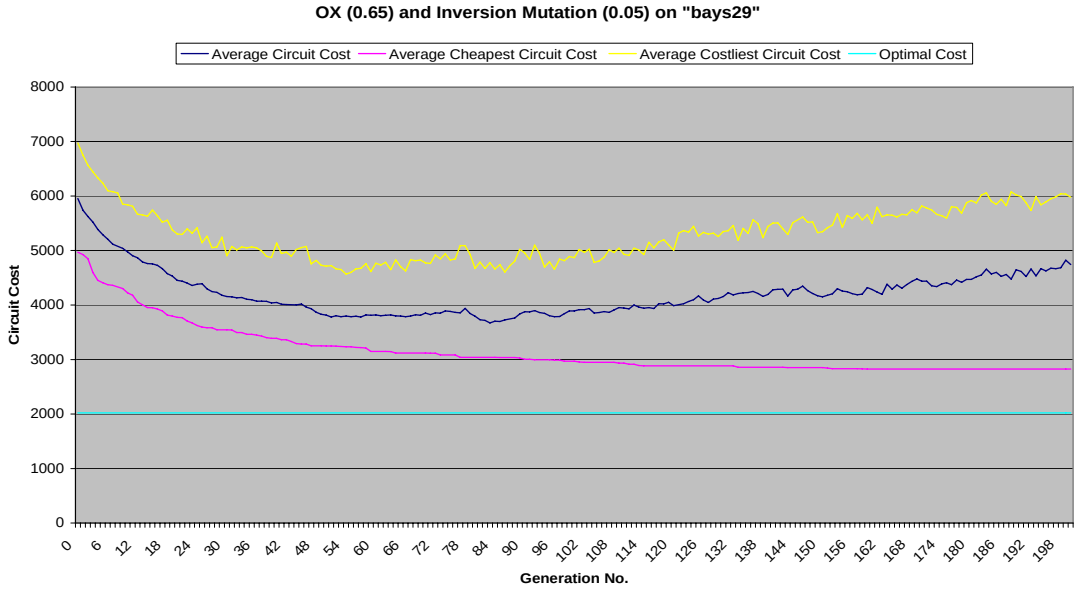
| | |
|---|---|
| **Graph** | bays29 |
| **Population Size** | 61 |
| **Elites** | 1 |
| **Copy Probability** | 0.3 |
| **Crossover (OX) Probability** | 0.65 |
| **Mutation Probability (Inversion)** | 0.05 |
| **Generations** | 200 |

The cheapest circuit costs obtained from the 6 trials were 2821, 2889, 2849, 3269, 2505 and 2628. The average cheapest circuit cost of these 6 trials is 2827 and the standard deviation comes to 262. The average convergence rates are shown in Figure 18.

The average convergence of IM at 0.05 is somewhat similar to SSM at 0.05; only real difference is that for IM at 0.05 the averages for circuit cost and costliest circuit cost are

rising. It happens to be that lots of mutation causes an increase in some tour costs and so subsequent mutations on these increases averages for circuit cost and costliest circuit cost.

**Figure 18 (5.1.2 – 3) Average convergence of 6 GA runs**



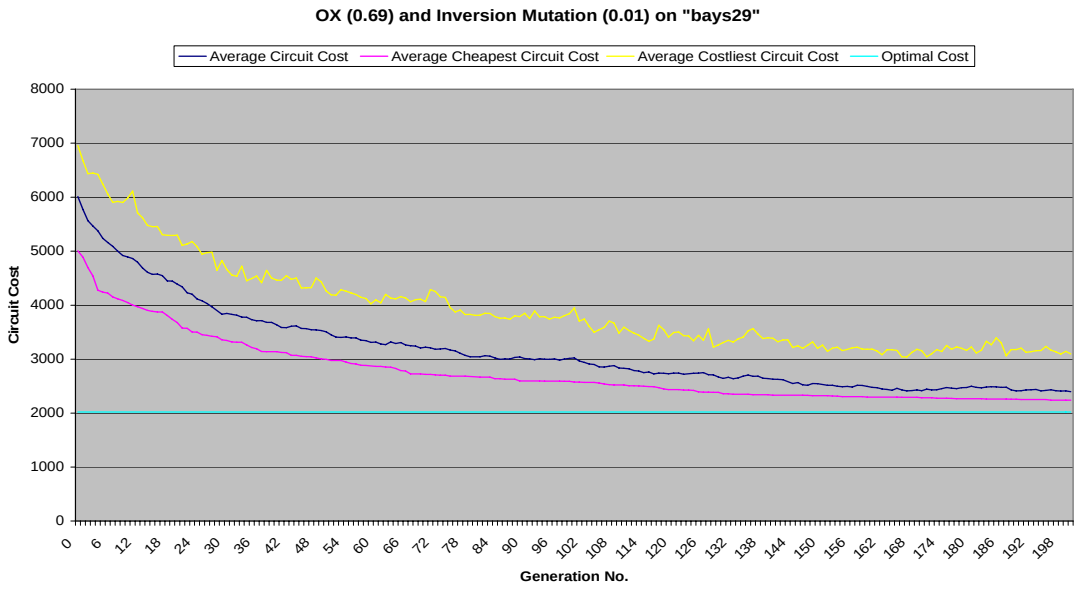

4. OX (0.69) and Inversion Mutation (0.01) on *bays29*

**Table 20 Parameters for 5.1.2 - 4**

| | |
|---|---|
| **Graph** | bays29 |
| **Population Size** | 61 |
| **Elites** | 1 |
| **Copy Probability** | 0.3 |
| **Crossover (OX) Probability** | 0.69 |
| **Mutation Probability (Single Swap)** | 0.01 |
| **Generations** | 200 |

The cheapest circuit costs obtained from the 6 trials were 2238, 2218, 2252, 2254, 2209 and 2278. The average cheapest circuit cost of these 6 trials is 2242 and the standard deviation comes to 25 (lower than IM at 0.1, thereby reflecting less volatility). The average convergence rates are shown in Figure 19. As you can see IM at 0.01 brings the population diversity under control.

**Figure 19 (5.1.2 – 4) Average convergence of 6 GA runs**




General Comments

The following experiments have some interesting findings. OX at 0.7 produced average cheapest circuit costs (ACCC) of 2272 with the standard deviation (SD) of this being 102. OX at 0.69 and SSM at 0.01 produced ACCC of 2337 with the SD being 74 whereas OX at 0.69 and IM at 0.01 produced ACCC of 2242 with the SD being 25.

The parameter settings with mutation produced lower standard deviations of ACCC; with IM used the ACCC was less compared with OX at 0.7. The reason is that Chromosomes not undergoing mutation will have an even higher selection probability in the next generation.

### 5.1.3 Single Swap Mutation vs. Inversion Mutation on *bays29*

Slight changes are made to the parameters, this time Crossover Probability will be 0.79, Mutation Probability will be 0.01 and the GA will run for 400 Generations. Single Swap and Inversion mutations were performed with CX but results proved inconclusive in determining the better method. Details of these trials are in "Miscellaneous Results - 2" of the Appendices.

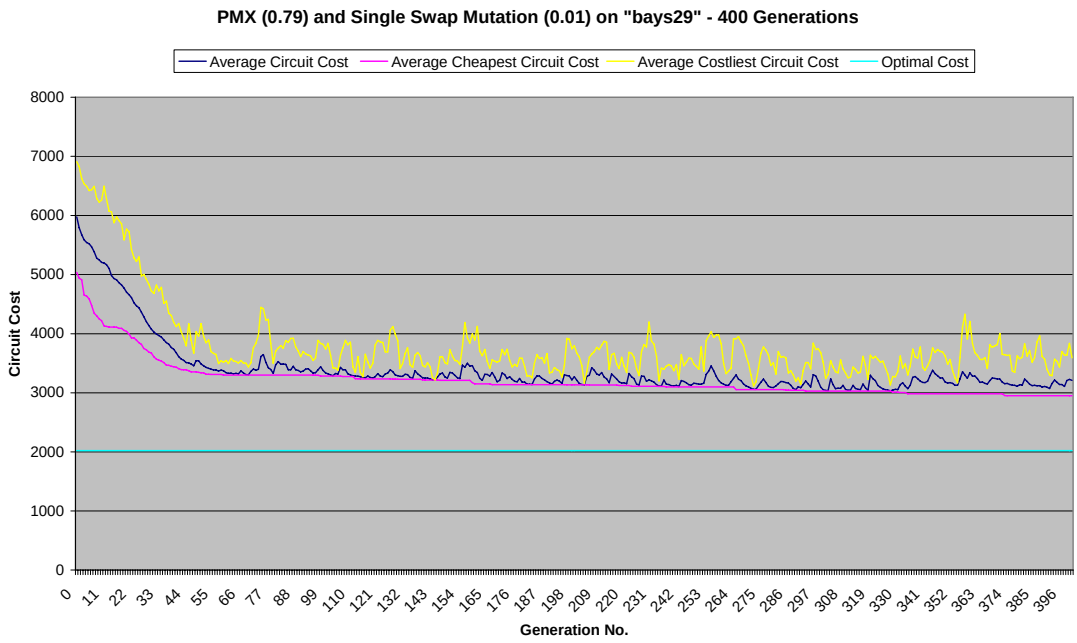
1. PMX (0.79) and Single Swap Mutation (0.01) on *bays29*

**Table 21 Parameters for 5.1.3 - 1**

| | |
|---|---|
| **Graph** | bays29 |
| **Population Size** | 61 |
| **Elites** | 1 |
| **Copy Probability** | 0.2 |
| **Crossover (PMX) Probability** | 0.79 |
| **Mutation Probability (Single Swap)** | 0.01 |
| **Generations** | 400 |

The cheapest circuit costs obtained from the 6 trials were 3216, 3069, 3069, 2806, 2921 and 2612. The average cheapest circuit cost of these 6 trials is 2949 and the standard deviation comes to 157. The average convergence rates are shown in Figure 20. Previously with the GA running PMX alone, the population fully converges at after 50 or so generations. With mutation in place the search is prolonged.

**Figure 20 (5.1.3 – 1) Average convergence of 6 GA runs**



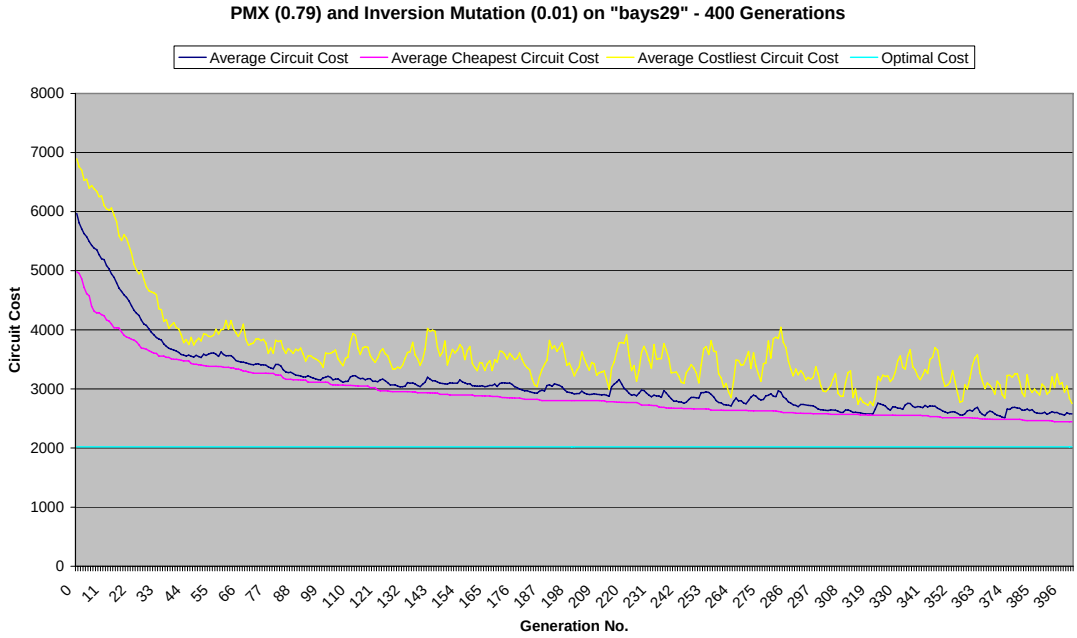

2. PMX (0.79) and Inversion Mutation (0.01) on *bays29*

**Table 22 Parameters for 5.1.3 - 2**

| | |
|---|---|
| **Graph** | bays29 |
| **Population Size** | 61 |
| **Elites** | 1 |
| **Copy Probability** | 0.2 |
| **Crossover (PMX) Probability** | 0.79 |
| **Mutation Probability (Inversion)** | 0.01 |
| **Generations** | 400 |

The cheapest circuit costs obtained from the 6 trials were 2380, 2350, 2371, 2643, 2494 and 2445. The average cheapest circuit cost of these 6 trials is 2447 and the standard deviation comes to 110. The average convergence rates are shown in Figure 21. IM helps prolong the search. Compared to SSM, IM produces better results clearly by identifying that the average circuit cost for IM reaches around 2500 compared to SSM's which only reaches around 3000. The SD for IM was lower than SSM's, indicating that Chromosomes not undergoing mutation will have higher selection probabilities in the next generations. The results are in line when mutations were applied with OX that is SSM producing higher ACCC and SD and IM producing lower ACCC and SD.

**Figure 21 (5.1.3 – 2) Average convergence of 6 GA runs**




### 5.1.4 Nearest Optimal Solution for *bays29*

The best near optimum solution recorded was **2068** and was found by using the following settings for GA_Final_RWS_Eltismv1.1.

**Table 23 Parameters for 5.1.4**

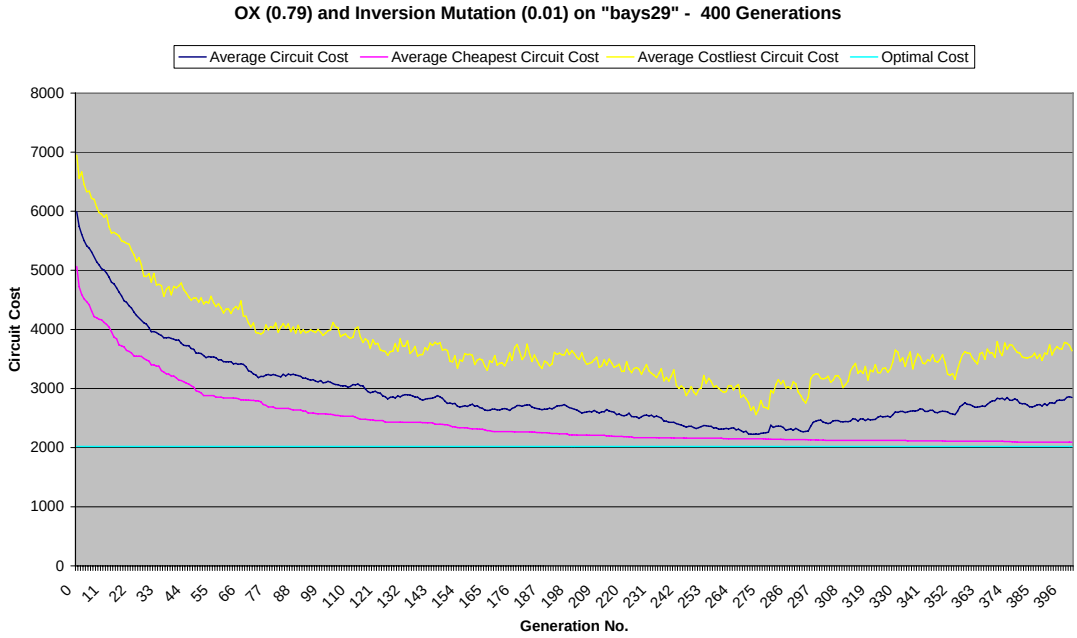
| | |
|---|---|
| **Graph** | bays29 |
| **Population Size** | 61 |
| **Elites** | 1 |
| **Copy Probability** | 0.2 |
| **Crossover (OX) Probability** | 0.79 |
| **Mutation Probability (Inversion)** | 0.01 |

| **Generations** | 400 |
|---|---|

The cheapest circuit costs obtained from the 6 trials were 2089, **2068** (2.3% within optimum), 2087, 2145, 2080 and 2092. The average cheapest circuit cost of these 6 trials is 2094 (on average 3.6% within optimum) and the standard deviation comes to 27. The average convergence rates are shown in

Figure 22.

**Figure 22 (5.1.4) Average convergence of 6 GA runs**




## Comments

Order Crossover and Inversion Mutation are proving to be effective Genetic Algorithm operators for solving Travelling Salesman Problems. By comparing the Average Circuit Costs for 5.1.3 – 2 and 5.1.4 it appears that the OX method adds to population diversity.

This experiment was repeated but with Single Swap Mutation used instead. The results backed up the case that Inversion Mutation is more effective than Single Swap Mutation as the average cheapest circuit cost came to 2177 (standard deviation of 89). Details of this experiment can be found in “Miscellaneous Results - 3” of the Appendices.

# 6. Summary and Conclusions

## 6.1 OX vs. PMX vs. CX

OX is able to search a local area more deeply than PMX and OX. This is evident by comparing the GA with just OX alone (5.1.0 – 3) against performances of the GA with just PMX (5.1.0 – 1) and CX (5.1.0 – 2) alone. For 5.1.0 – 3 OX manages to maintain the search for the 200 generations specified and produced on average solutions within 12% of bays29's optimum. In both 5.1.0 – 1 and 5.1.0 – 2 the GA converges in under 50 generations.
An explanation to as why the GA using CX converges rapidly has already been given in 5.1.0 – 2.

It can also be concluded that PMX is better than CX by comparing the results of experiments 5.1.3 – 2 (PMX with Inversion) and Appendices: 2 – 2 (CX with Inversion). The literature study also reports similar findings that OX is better than PMX and that PMX is better than CX **[Page 46 - 3rd Paragraph, 18]**

## 6.2 Ordering of Nodes vs. Absolute Position of Nodes

The Order Crossover states that the ordering of nodes (not their exact tour positions) is important **[Page 15, 18]**. In contrast Cycle Crossover produces offspring such that for every node's position on a tour, one of its parents has exactly the same node in the specified position. Partially Mapped Crossover produces offspring such that both ordering and positions of nodes are preserved as much as possible.

The OX method creates offspring by preserving a chosen segment from Parent 1 and refills the other 2 segments with nodes from Parent 2 that are not in the selected segment. The unselected segments are filled such that the order of nodes from Parent 2 is relatively maintained. In effect the offspring is produced by combining parents' sub-tours that are as long as possible. This means certain linkage of nodes, possibly contributing to cheaper circuit costs have a higher chance of staying together during the cutting and splicing actions of crossover. As a result information about good building blocks of a solution is unlikely to be destroyed.

The OX method is not a Greedy Algorithm as it doesn't examine the cost of different node orderings. From Conclusion 6.1, OX is better than PMX and CX, so a method taking account node orderings is the right approach for crossing over solutions of the TSP.

## 6.3 OX Probability

After seeing positive performances of OX on bays29 (experiment 5.1.0 – 3), 4 set of experiments in 5.1.1 determined which crossover rate is preferable.

From the 5 experiments involving different OX probabilities from [0.6 to 1.0] rates of 0.7, 0.8 and 0.9 performed better.

**Table 24 Summary of Results - OX Probabilities on "bays29"**

| 1. OX Probability | 2. Corresponding Graph | 3. Av Cheapest Circuit Cost | 4. Standard Deviation (of 3.) |
|---|---|---|---|
| 0.6 | Figure 12 | 2495 | 143 |
| 0.7 | Figure 11 | 2272 | 102 |
| 0.8 | Figure 13 | 2234 | 117 |
| 0.9 | Figure 14 | 2261 | 128 |
| 1.0 | Figure 15 | 2333 | 140 |

The graph of OX at 0.7 had a greater convergence rate compared to OX at 0.8 and 0.9. OX at 0.7 has a higher average cheapest circuit cost than OX at 0.8 and 0.9. However the graphs of OX at 0.8 and 0.9 have a much slower convergence rate than 0.7.

From the results OX at 0.8 is slightly better than OX at 0.9 as it produced a lower average cheapest circuit cost and standard deviation. This could be due to higher copy probabilities forcing better Chromosomes to be carried into the next generation thus undergoing more chances for crossover.

Crossover probability should generally be high, but this should depend on the methods used in a Genetic Algorithm and the TSP instance.

## 6.4 Mutation Probability

In GA_Final_RWS_Elitismv1.1 premature convergence to a point on the search space occurs when all Chromosomes in the population become identical through natural

selection. Crossing over identical chromosomes produces identical offspring. Identical Chromosomes will have the same circuit cost therefore each Chromosome will have a fitness of 0.0 (as Chromosome's fitness is calculated by subtracting its circuit cost from the most expensive in the current population). A roulette wheel can't be created as the total fitness is "0.0" in this case the first Chromosome in the population is picked by default.

The effects of mutation can be observed when all Chromosomes' fitness becomes 0.0. When mutation occurs some Chromosomes' tours will be changed, this means that every Chromosome's circuit cost will not also be the most expensive circuit cost. As a result the Roulette Wheel can be constructed and so the search is resuscitated.

For experiment 5.1.2 – 1 (OX at 0.65, SSM at 0.05) the average cheapest circuit costs (ACCC) after 200 generations was 2649. In experiment 5.1.0 – 3 (OX at 0.7, No Mutation) the ACCC was 2170. Mutation at 0.05 means that the GA loses its convergence; this can be seen by comparing the graphs of 5.1.2 and 5.1.0 respectively. In contrast experiment 5.1.2 – 2 (OX at 0.69, SSM at 0.01) produces an ACCC of 2237, similar results occurred when Inversion Mutation was used.

Mutation probability should generally be low, but this should depend on the methods used in a Genetic Algorithm and the TSP instance.

### 6.5 Inversion Mutation vs. Single Swap Mutation

When Inversion Mutation (IM) was combined with OX and PMX, the Genetic Algorithm performed well in contrast to the combinations of Single Swap Mutation (SSM) with OX and PMX.

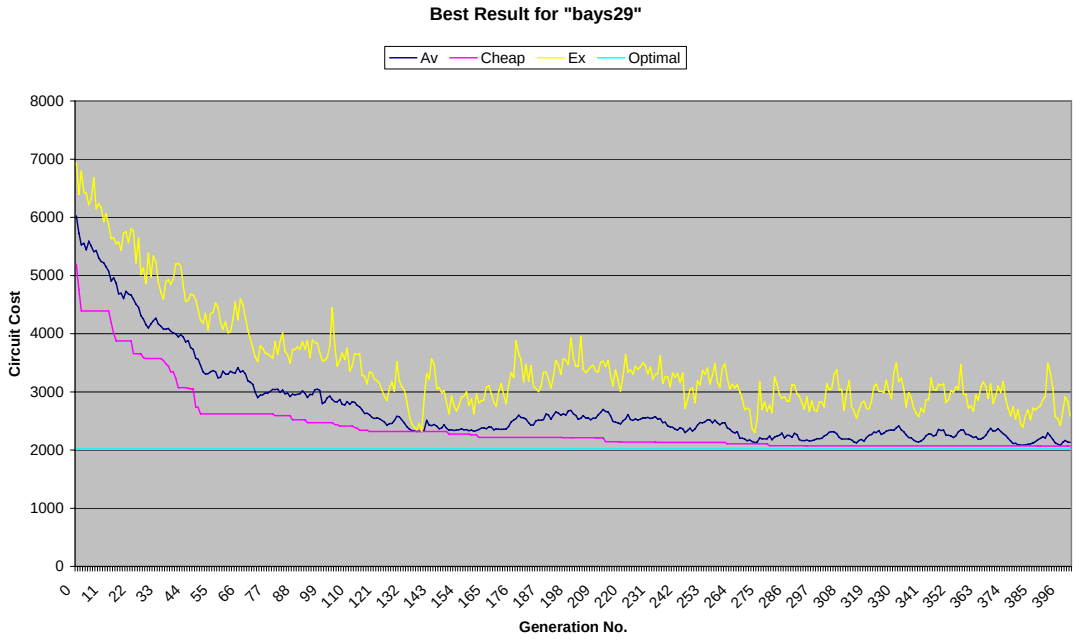
The results that determined IM is better than SSM came from testing these operators with PMX (experiments 5.1.3 - 1 and 5.1.3 – 2). For experiment 5.1.3 – 1 (SSM and PMX) the ACCC was 2949 and in 5.1.3 – 2 (IM and PMX) the ACCC was 2447.

Inversion reverses node orderings in a given segment. This means that nodes just outside the given segment will experience a different linkage. Linkage of these nodes may have circuit costs within the graph's average edge weight and nearer - optimal solutions can be produced by crossover if these linkages are inherited.

## 6.6 Near - Optimal Solutions for *bays29* and Comparisons to Related Work

The best set of results came from experiment 5.1.4. On average GA_Final_RWS_Elitismv1.1 managed to find solutions within 3.6% of the optimum. The best near optimal solution obtained was 2068 (2.3% within optimum). Figure 23 shows the average convergence rates for the best trial.

**Figure 23 Average Convergence for Best Trial**




Elitism plays a part in ensuring the Cheapest Circuit Cost decreases as the Generation No. increases. At Generation No. 141 premature convergence occurs (The Average, Cheapest and Costliest Circuit Costs are at 2319) but IM at 0.01 plays its part in resuscitating the search. The effects of mutation are evident by the costliest circuit cost line being the most volatile. The cheapest Hamiltonian Circuit is found at Generation No. 390.

Here is a comparison between the global optimal circuit and the near – optimal circuit obtained, Figure 24. Highlighted are similarities in node linkages between the two. As you can see there is a focus on finding and rearranging good sub-tours together.

**Figure 24 Hamiltonian Circuits: Global Optimum vs. Near Optimum**

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8 | 4 | 25 | 28 | 2 | 1 | 20 | 19 | 9 | 3 | 14 | 17 | 16 | 13 | 21 | 10 | 6 | 24 | 18 | 15 | 12 | 23 | 26 | 22 | 7 | 0 | 27 | 5 | 11 | and back to 8 | Near Optimum |
| 0 | 27 | 5 | 11 | 8 | 4 | 25 | 28 | 2 | 1 | 19 | 9 | 3 | 14 | 17 | 16 | 13 | 21 | 10 | 18 | 24 | 6 | 22 | 26 | 7 | 23 | 15 | 12 | 20 | and back to 0 | Global Optimum |

Yzelman's experiment involving Greedy Crossover (Section 2.6) can be partially compared to 5.1.4.

| | **Yzelman's (Section 2.6)** | **5.1.4** |
|---|---|---|
| **Generations** | 7500 | 400 |
| **Population Size** | 250 | 61 |
| **Selection** | Roulette Wheel | Roulette Wheel |
| **Elites** | 1 | 1 |
| **Crossover Method and Probability** | Greedy Crossover (0.3) | Order Crossover (0.79) |
| **Mutation Method and Probability** | Mutation (0.12) split between Single Swap at 0.33, Insertion at 0.67 | Inversion Mutation (0.01) |
| **Copy Probability** | 0.58 | 0.2 |
| **Average Cheapest Circuit Cost** | 2090 | 2094 |
| **Cheapest Circuit Cost Found** | 2056 (at Generation No. 464) | 2068 (at Generation No. 390) |

The Greedy Crossover appears to be a strong method as its crossover probability is only 0.3. The Order Crossover in GA_Final_RWS_Elitismv1.1 requires a higher crossover probability to find ever decreasing points on the search space.

There isn't a lot of difference between the Average Cheapest Circuit Cost. Also there isn't much difference in the Cheapest Circuit Cost found considering that Yzelman's best solution required more generations than **5.1.4**.

From examining the respective graphs (Figure 4 and Figure 22), experiment 5.1.4 seems to lose its convergence (average circuit cost doesn't move downwards). Yzelman's experiment maintains convergence **[Page 32 – Basic GA with…., 17]** this can be due to the fact that Greedy Crossover is an edge based operator.

The plots for average circuit costs ("*average population score*" in Yzelman's case) show differences in population diversity. Note that in Yzelman's case the population size and mutation rates are higher than 5.1.4. Higher population sizes and mutation rates increase population diversity, however an important factor in maintaining convergence is the chance for better Chromosomes to undergo crossover. This could be due to the fitness function and there are differences between the two programs. Yzelman's fitness function assigns a value between 1 (low) to 10 (high) for a Chromosome depending on the other Chromosomes in the

population **[Page 9, 17]**. This reduces the selective pressure for solutions with cheaper circuit costs and so the search is concentrated more in a particular area.

## 6.7 Recommendations for Further Work

- **Population Size:** The population size was kept constant throughout. Would recommend that different population sizes are tried out.
- **Fitness Function:** The fitness function has been overlooked and its importance has been underestimated. Would recommend investigating similar work, note the various fitness functions used and carry out trials on them.
- **Literature Review:** The authors of the literature review **[18]** carried out experiments on different combinations of crossover and mutation. Unfortunately comparisons can't be made between the review's results due to the vast differences in parameter settings. This was due to the timing on acquiring the publication.
- **Heuristic Methods:** The crossover operators used in this project aren't heuristic methods. Greedy Crossover as mentioned in Yzelman's study is a heuristic method as it is edge based. Carrying out a study into methods that have more heuristics within them is recommended.
- ***"swiss42"* Trials:** As a matter of interest six trials were made on *"swiss42"*, a 42 node graph with the optimal Hamiltonian Circuit cost being 1272. Parameter settings from 5.1.4 were used. Mixed results were produced as the Cheapest Circuit Costs were 1967, 2116, 1725, 2186, 1476 and 1453. So repeating the projects experiments to see which aspects of GA_Final_RWS_Elitismv1.1 could be improved is recommended.

## 7. Bibliography

[1] **Heaton, Jeff** – "*Introduction to Neural Networks for Java (Second Edition)*", St Louis - USA: Heaton Research, 2007. [ISBN 978-1-60439-008-7]. *"Ch 6: Understanding Genetic Algorithms"*. Also available at [http://www.heatonresearch.com/course/intro-neural-nets-java].

[2] **Letchford, Adam –** "*The Travelling Salesman Problem*". Available at [www.lancs.ac.uk/staff/letchfoa/talks/TSP.pdf].

[3] **Mitchell, Melanie** – "*An Introduction to Genetic Algorithms*", Cambridge (Massachusetts) – USA: The MIT Press, 1998. [ISBN 0-262-63185-7]. *"Ch 1: Genetic Algorithms: An Overview", "Ch 4: Theoretical Foundations of Genetic Algorithms", "Ch 5: Implementing a Genetic Algorithm"*.

[4] **Radcliffe, Nick** and **Wilson, Greg** – "*Natural Solutions Give Their Best*", New Scientist, Issue 1712, Pages 47 – 50, 14 April 1990.

[5] **Sangalli, Arturo** – "*Why Sales Reps Pose a Hard Problem*", New Scientist, Issue 1851, Pages 24 - 28, 12 December 1992.

[6] **Wikipedia –** "*Genetic Algorithm*". Available at [http://en.wikipedia.org/wiki/Genetic_algorithm].

[7] **Wikipedia –** "*Travelling Salesman Problem*". Available at [http://en.wikipedia.org/wiki/Travelling_salesman_problem].

[8] **Whitley, Darrel** – "*Genetic Algorithms and Evolutionary Computing*". Available at [http://www.cs.colostate.edu/~genitor/2002/encyclo.pdf].

[9] **Zbigniew, Michalewicz –** "*Genetic Algorithms + Data Structures = Evolution Programs (Third Revised Extended Edition)*", Berlin - Germany: Springer, 1996. [ISBN 3-540-60676-9]. *"Ch 1: GAs: What Are They?" "Ch 2: GAs: How Do They Work?" "Ch 3: GAs: Why Do They Work?" "Ch 4: GAs Selected Topics" "Ch 10: The Travelling Salesman Problem"*.

[10] **Pohlheim, Hartmut –** "*Evolutionary Algorithms 3 Selection*". Available at [http://www.geatbx.com/docu/algindex-02.html#P363_18910].

[11] **Boukreev, Konstantin** – "*Genetic Algorithms and the Travelling Salesman Problem*". Available at [http://www.generation5.org/content/2001/tspapp.asp].

[12] **Obikto, Marek** – "*Genetic Algorithms*". Available at [http://www.obitko.com/tutorials/genetic-algorithms/index.php].

[13] **Wikipedia –** *"Chromosomal Crossover"*. Available at [http://en.wikipedia.org/wiki/Chromosomal_crossover].

[14] **Widak, Kristov** and **Chatterjee, Kaushik -** *"Genetic Algorithms II"*. Available at [http://www.cs.sunysb.edu/~cse634/spring2009/Genetic_Algorithms_2.pdf].

[15] **Russell, Stuart** and **Norvig, Peter –** *"Artificial Intelligence: A Modern Approach"*, Prentice Hall, 2nd Edition. ISBN 0-13-080302-2.

[16] **Research Group Combinatorial Optimization, University of Heidelberg**, - *"TSPLIB – Symmetric Travelling Salesman Problem (TSP)"*. Available at [http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95].

[17] **Yzelman A.N**, "*Genetic Algorithms*". Available at [http://academic.trancethrust.nl/ga_tsp.pdf]. Plus his homepage at [http://www.math.uu.nl/people/Yzelman].

[18] **Larranaga, Kuijpers, Inza** and **Dizdarevic**, *"Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators"*, Artificial Intelligence Review, 1999. Available at [http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.35.8882].

# A. Appendices

## Background Information and Related Work

### TSP Benchmark *bays29*

| | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 107 | 241 | 190 | 124 | 80 | 316 | 76 | 152 | 157 | 283 | 133 | 113 | 297 | 228 | 129 | 348 | 276 | 188 | 150 | 65 | 341 | 184 | 67 | 221 | 169 | 108 | 45 | 167 |
| 107 | 0 | 148 | 137 | 88 | 127 | 336 | 183 | 134 | 95 | 254 | 180 | 101 | 234 | 175 | 176 | 265 | 199 | 182 | 67 | 42 | 278 | 271 | 146 | 251 | 105 | 191 | 139 | 79 |
| 241 | 148 | 0 | 374 | 171 | 259 | 509 | 317 | 217 | 232 | 491 | 312 | 280 | 391 | 412 | 349 | 422 | 356 | 355 | 204 | 182 | 435 | 417 | 292 | 424 | 116 | 337 | 273 | 77 |
| 190 | 137 | 374 | 0 | 202 | 234 | 222 | 192 | 248 | 42 | 117 | 287 | 79 | 107 | 38 | 121 | 152 | 86 | 68 | 70 | 137 | 151 | 239 | 135 | 137 | 242 | 165 | 228 | 205 |
| 124 | 88 | 171 | 202 | 0 | 61 | 392 | 202 | 46 | 160 | 319 | 112 | 163 | 322 | 240 | 232 | 314 | 287 | 238 | 155 | 65 | 366 | 300 | 175 | 307 | 57 | 220 | 121 | 97 |
| 80 | 127 | 259 | 234 | 61 | 0 | 386 | 141 | 72 | 167 | 351 | 55 | 157 | 331 | 272 | 226 | 362 | 296 | 232 | 164 | 85 | 375 | 249 | 147 | 301 | 118 | 188 | 60 | 185 |
| 316 | 336 | 509 | 222 | 392 | 386 | 0 | 233 | 438 | 254 | 202 | 439 | 235 | 254 | 210 | 187 | 313 | 266 | 154 | 282 | 321 | 298 | 168 | 249 | 95 | 437 | 190 | 314 | 435 |
| 76 | 183 | 317 | 192 | 202 | 141 | 233 | 0 | 213 | 188 | 272 | 193 | 131 | 302 | 233 | 98 | 344 | 289 | 177 | 216 | 141 | 346 | 108 | 57 | 190 | 245 | 43 | 81 | 243 |
| 152 | 134 | 217 | 248 | 46 | 72 | 438 | 213 | 0 | 206 | 365 | 89 | 209 | 368 | 286 | 278 | 360 | 333 | 284 | 201 | 111 | 412 | 321 | 221 | 353 | 72 | 266 | 132 | 111 |
| 157 | 95 | 232 | 42 | 160 | 167 | 254 | 188 | 206 | 0 | 159 | 220 | 57 | 149 | 80 | 132 | 193 | 127 | 100 | 28 | 95 | 193 | 241 | 131 | 169 | 200 | 161 | 189 | 163 |
| 283 | 254 | 491 | 117 | 319 | 351 | 202 | 272 | 365 | 159 | 0 | 404 | 176 | 106 | 79 | 161 | 165 | 141 | 95 | 187 | 254 | 103 | 279 | 215 | 117 | 359 | 216 | 308 | 322 |
| 133 | 180 | 312 | 287 | 112 | 55 | 439 | 193 | 89 | 220 | 404 | 0 | 210 | 384 | 325 | 279 | 415 | 349 | 285 | 217 | 138 | 428 | 310 | 200 | 354 | 169 | 241 | 112 | 238 |
| 113 | 101 | 280 | 79 | 163 | 157 | 235 | 131 | 209 | 57 | 176 | 210 | 0 | 186 | 117 | 75 | 231 | 165 | 81 | 85 | 92 | 230 | 184 | 74 | 150 | 208 | 104 | 158 | 206 |
| 297 | 234 | 391 | 107 | 322 | 331 | 254 | 302 | 368 | 149 | 106 | 384 | 186 | 0 | 69 | 191 | 59 | 35 | 125 | 167 | 255 | 44 | 309 | 245 | 169 | 327 | 246 | 335 | 288 |
| 228 | 175 | 412 | 38 | 240 | 272 | 210 | 233 | 286 | 80 | 79 | 325 | 117 | 69 | 0 | 122 | 122 | 56 | 56 | 108 | 175 | 113 | 240 | 176 | 125 | 280 | 177 | 266 | 243 |
| 129 | 176 | 349 | 121 | 232 | 226 | 187 | 98 | 278 | 132 | 161 | 279 | 75 | 191 | 122 | 0 | 244 | 178 | 66 | 160 | 161 | 235 | 118 | 62 | 92 | 277 | 55 | 155 | 275 |
| 348 | 265 | 422 | 152 | 314 | 362 | 313 | 344 | 360 | 193 | 165 | 415 | 231 | 59 | 122 | 244 | 0 | 66 | 178 | 198 | 286 | 77 | 362 | 287 | 228 | 358 | 299 | 380 | 319 |
| 276 | 199 | 356 | 86 | 287 | 296 | 266 | 289 | 333 | 127 | 141 | 349 | 165 | 35 | 56 | 178 | 66 | 0 | 112 | 132 | 220 | 79 | 296 | 232 | 181 | 292 | 233 | 314 | 253 |
| 188 | 182 | 355 | 68 | 238 | 232 | 154 | 177 | 284 | 100 | 95 | 285 | 81 | 125 | 56 | 66 | 178 | 112 | 0 | 128 | 167 | 169 | 179 | 120 | 69 | 283 | 121 | 213 | 281 |
| 150 | 67 | 204 | 70 | 155 | 164 | 282 | 216 | 201 | 28 | 187 | 217 | 85 | 167 | 108 | 160 | 198 | 132 | 128 | 0 | 88 | 211 | 269 | 159 | 197 | 172 | 189 | 182 | 135 |
| 65 | 42 | 182 | 137 | 65 | 85 | 321 | 141 | 111 | 95 | 254 | 138 | 92 | 255 | 175 | 161 | 286 | 220 | 167 | 88 | 0 | 299 | 229 | 104 | 236 | 110 | 149 | 97 | 108 |
| 341 | 278 | 435 | 151 | 366 | 375 | 298 | 346 | 412 | 193 | 103 | 428 | 230 | 44 | 113 | 235 | 77 | 79 | 169 | 211 | 299 | 0 | 353 | 289 | 213 | 371 | 290 | 379 | 332 |
| 184 | 271 | 417 | 239 | 300 | 249 | 168 | 108 | 321 | 241 | 279 | 310 | 184 | 309 | 240 | 118 | 362 | 296 | 179 | 269 | 229 | 353 | 0 | 121 | 162 | 345 | 80 | 189 | 342 |
| 67 | 146 | 292 | 135 | 175 | 147 | 249 | 57 | 221 | 131 | 215 | 200 | 74 | 245 | 176 | 62 | 287 | 232 | 120 | 159 | 104 | 289 | 121 | 0 | 154 | 220 | 41 | 93 | 218 |
| 221 | 251 | 424 | 137 | 307 | 301 | 95 | 190 | 353 | 169 | 117 | 354 | 150 | 169 | 125 | 92 | 228 | 181 | 69 | 197 | 236 | 213 | 162 | 154 | 0 | 352 | 147 | 247 | 350 |
| 169 | 105 | 116 | 242 | 57 | 118 | 437 | 245 | 72 | 200 | 359 | 169 | 208 | 327 | 280 | 277 | 358 | 292 | 283 | 172 | 110 | 371 | 345 | 220 | 352 | 0 | 265 | 178 | 39 |
| 108 | 191 | 337 | 165 | 220 | 188 | 190 | 43 | 266 | 161 | 216 | 241 | 104 | 246 | 177 | 55 | 299 | 233 | 121 | 189 | 149 | 290 | 80 | 41 | 147 | 265 | 0 | 124 | 263 |
| 45 | 139 | 273 | 228 | 121 | 60 | 314 | 81 | 132 | 189 | 308 | 112 | 158 | 335 | 266 | 155 | 380 | 314 | 213 | 182 | 97 | 379 | 189 | 93 | 247 | 178 | 124 | 0 | 199 |
| 167 | 79 | 77 | 205 | 97 | 185 | 435 | 243 | 111 | 163 | 322 | 238 | 206 | 288 | 243 | 275 | 319 | 253 | 281 | 135 | 108 | 332 | 342 | 218 | 350 | 39 | 263 | 199 | 0 |

The optimal route cost is 2020 and the Hamiltonian Circuit is: {0, 27, 5, 11, 8, 4, 25, 28, 2, 1, 19, 9, 3, 14, 17, 16, 13, 21, 10, 18, 24, 6, 22, 26, 7, 23, 15, 12, 20}

Considering that the 29 nodes are labelled from [0 – 28].

### TSP Benchmark *swiss42*

The optimal route cost is 1272. The optimal Hamiltonian Circuit hasn't been supplied by the research group.

## Requirements and Analysis

**CRC**

| | |
|---|---|
| **Interface Name** | Path_Crossover |
| **Implementing Classes** | PMX, OX, CX |
| **Implementing Classes Contractual Obligations** | ▪ Must implement a get Offspring1 method.<br>▪ Must implement a get Offspring2 method.<br>▪ Must implement a get Parent1 method.<br>▪ Must implement a get Parent2 method.<br>▪ Must implement a Crossover method for 2 Parents to produce an Offspring.<br>▪ Must implement a print Offspring method to print the 2 offspring produced. |
| **Collaborators** | None |
| **Class Name** | PMX |
| **Responsibilities** | ▪ Implements a get Offspring1 method and Offspring2.<br>▪ Implements a get Parent1 and a Parent2 method.<br>▪ Implements a Crossover method for 2 Parents to produce an Offspring by Partially Mapped Crossover.<br>▪ Implements a print Offspring method. |
| **Collaborators** | None |
| **Class Name** | OX |
| **Responsibilities** | ▪ Implements a get Offspring1 method and Offspring2.<br>▪ Implements a get Parent1 and a Parent2 method.<br>▪ Implements a Crossover method for 2 Parents to produce an Offspring by Order Crossover.<br>▪ Implements a print Offspring method. |
| **Collaborators** | None |
| **Class Name** | CX |
| **Responsibilities** | ▪ Implements a get Offspring1 method and Offspring2.<br>▪ Implements a get Parent1 and a Parent2 method.<br>▪ Implements a Crossover method for 2 Parents to produce an Offspring by Cycle Crossover.<br>▪ Implements a print Offspring method. |
| **Collaborator** | None |
| **Class Name** | Chromosome |
| **Responsibilities** | ▪ Represents a solution to a Travelling Salesman Problem. That is storing the Hamiltonian Circuit, Circuit Cost and the Derived Fitness of the Solution.<br>▪ Has a mutate method to randomly change the Hamiltonian Circuit produced.<br>▪ Prints it Circuit. |
| **Collaborators** | None |
| **Class Name** | Roulette Wheel |
| **Responsibilities** | ▪ Responsible for selecting Chromosomes to be |

| | |
|---|---|
| | evolved by Roulette Wheel selection. Returns the Index of the Parent Chosen in the Population. |
| **Collaborators** | None |
| **Class Name** | Generation_Info |
| **Responsibilities** | ▪ Responsible for storing information about each Generation. Data required are Generation Number, Average Circuit Fitness, Cheapest & Most Expensive Circuit Cost, Number of Copies, Crossovers and Mutations. |
| **Collaborators** | None |
| **Class Name:** | Graph |
| **Responsibilities** | ▪ Responsible for representing the TSP Graph as a 2D array.<br>▪ Plays a role in generating random Circuits for the initial randomly generated population.<br>▪ Used to calculate the cost of a Chromosome's circuit. |
| **Collaborators** | Graph_Reader |
| **Class Name** | Graph_Reader |
| **Responsibilities** | ▪ Reads in a graph stored as a full matrix in a text file and represents it as a 2D array. |
| **Collaborators** | None |
| **Class Name** | Genetic_Algorithm_TSP (Main) |
| **Responsibilities** | ▪ To create an initial population.<br>▪ To carry out the evolutionary operators on the population (Selection, Copy, Crossover and Mutation).<br>▪ To calculate the fitness of every Chromosome in the Population.<br>▪ Print Chromosomes in the population for each generation.<br>▪ Print and calculate statistics about each generation. |
| **Collaborators** | Graph, Path_Crossover (PMX, OX, CX), Chromosome, Roulette_Wheel and Generation_Info. |

## Design and Implementation

**Unit Testing**

| Class | Test | Method of Testing | Purpose and Details of Testing | Results/ Conclusions |
|---|---|---|---|---|
| Graph | calculateCost(…): double<br>search_forEdgeWeight(…): double | TestNG<br>Graph_Test.java | Tests conducted in Graph_Test.java<br><br>**testCalculateCost():-**<br><br>The expected result of two Hamiltonian Circuits' cost for "tube_9stations.txt" is given.<br><br>assertEquals (calculateCost (…), expected result) needs to return true.<br><br>**testCalculateCost_bays29():-**<br><br>Same as method testCalculateCost () but with two Hamiltonian Circuits' cost for this TSP benchmark. | Passed.<br><br>The representation of the Hamiltonian Circuit is correct.<br><br>The method for calculating the Hamiltonian Circuit cost is correct.<br><br>As a result search_forEdgeWeight (…): double works correctly as this is used in calculateCost (…): double. |
| Graph_Reader | create_Graph()<br>get_weightedGraph(): double[][] | TestNG<br>Graph_Reader_Test.java<br><br>Inspection<br>Graph_Reader_Test.java | Tests conducted in Graph_Reader_Test.java<br><br>Graph_Reader reads the text file specified by the user.<br><br>**test_Size17(), test_size_bays29() , etc:-**<br><br>These tests ensure that the 2D array representation is created correctly by making sure the correct dimensions are met, assertEquals (…) is used. | Passed.<br><br>create_Graph () and get_weightedGraph (): double [] [] works. |

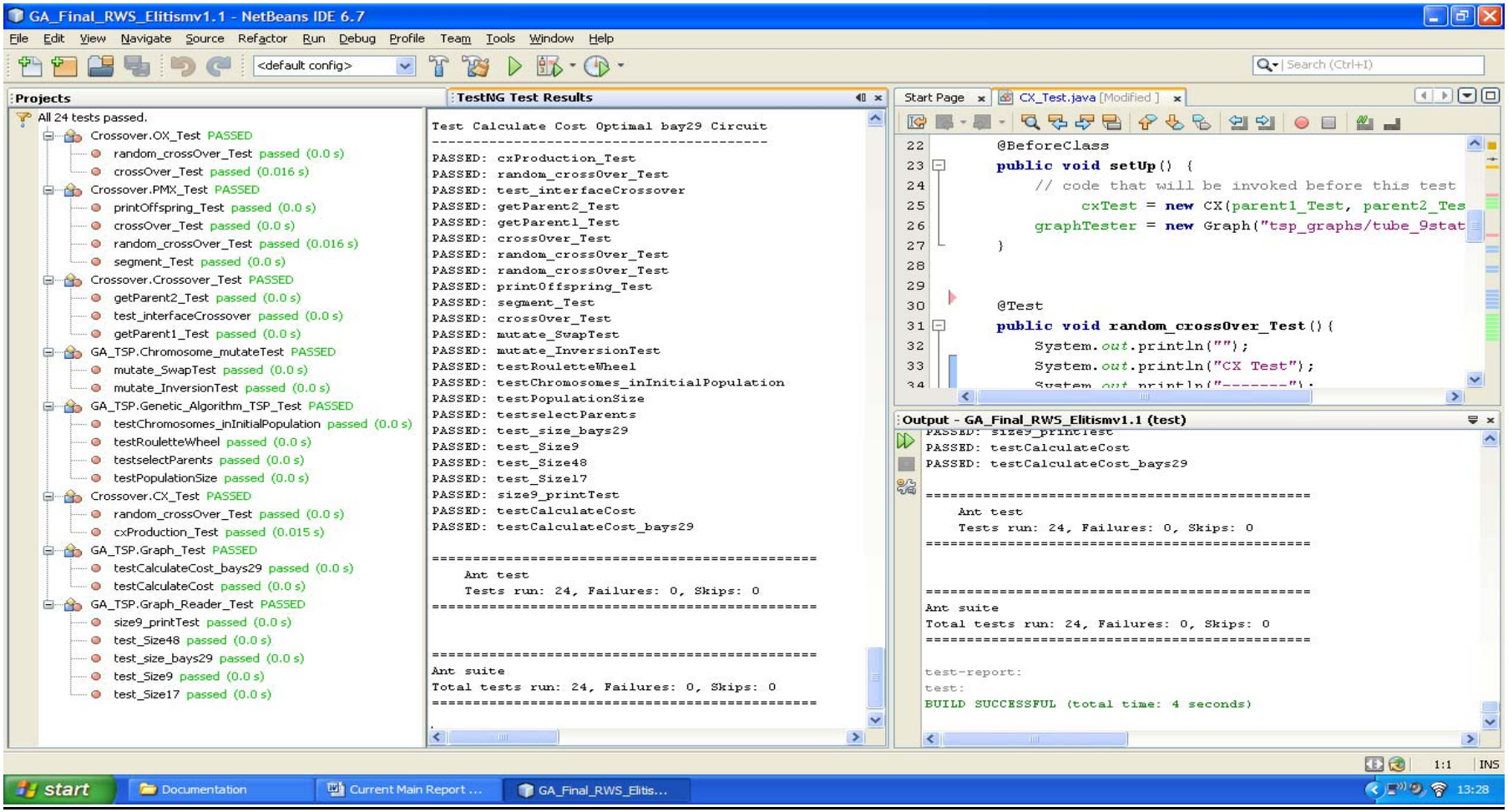
| | | | | |
|---|---|---|---|---|
| | | | **size9_printTest():-**<br><br>Print out 2D array representation of tube9_stations.txt. Also tests whether get_weightedGraph () works. | |
| Chromosome | mutateCircuit_SingleSwap()<br>mutateCircuit_Inversion() | Inspection<br>Chromosome_mutateTest.java | Tests conducted in Chromosome_mutateTest.java<br><br>Test to see if Chromosome's mutate method works.<br><br>**mutate_InversionTest()** and **mutate_SwapTest():-**<br><br>Inspect print outs to see if any changes are made. | Passed.<br><br>mutateCircuit_SingleSwap () and mutateCircuit_Inversion () works. |
| Genetic_Algorithm_TSP | Test the class's collaboration with Chromosome and Graph. | Inspection<br>Generic_Algorithm_TSP_Test.java<br><br>TestNG<br>Generic_Algorithm_TSP_Test.java | Tests conducted in Genetic_Algorithm_TSP_Test.java<br><br>Test to see if Graph's **random_hamiltonianCircuit():-**<br><br>int[] works (used to generate random initial population).<br><br>Test to see if all Chromosomes' "get" methods work.<br><br>**testChromosomes_inInitialPopulation():-**<br><br>The above 2 tests are performed; the | Passed.<br><br>Chromosomes' attributes are correctly printed out.<br><br>Population size is the same from supplied arguments. |

| | | | | |
|---|---|---|---|---|
| | | | Chromosomes in the initial population are printed out.<br><br>**testPopulationSize()**:-<br><br>To test if initial population is the same from arguments supplied. Uses assertEquals (...) | |
| Genetic_Algorithm_TSP | Test the class's collaboration with Roulette_Wheel. | Inspection<br>Generic_Algorithm_TSP_Test.java<br><br>TestNG<br>Generic_Algorithm_TSP_Test.java | Tests conducted in Genetic_Algorithm_TSP_Test.java<br><br>**testRouletteWheel():-**<br><br>Prints out the roulette wheel. The last cumulative probability should be arbitrary close to 1.0 or be 1.0.<br>The Roulette Wheel should be of length population size + 1. Uses assertEquals (...).<br><br>**testselectParents():-**<br><br>Prints out the random variable and corresponding selected parent's index in population. Inspect by hand that the index of parent corresponds to the random variable generated in the appropriate cumulative frequency interval. | Passed.<br><br>Roulette_Wheel is correctly implemented. |
| Path_Crossover (Implementing Classes) | Test the classes' "get" methods specified by the Path_Crossover interface. | Inspection<br>Crossover_Test.java | Tests conducted in Crossover_Test.java<br><br>**getParent1_Test()** and **getParent2_Test():-**<br><br>Prints out the parents passed into the implementing classes' constructors. | Passed.<br><br>Crossover classes are instantiated correctly. |

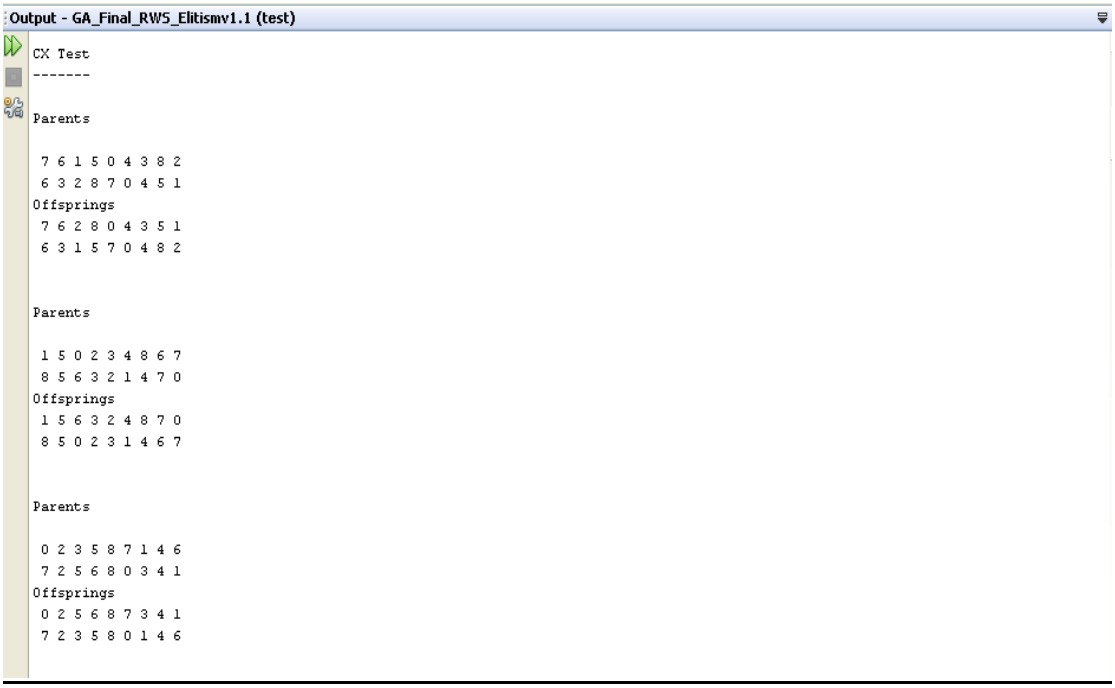
| | | | | |
|---|---|---|---|---|
| | | | **get_offspring1()** and **get_offspring2():-**<br><br>Prints out the offspring created when the implementing classes are instantiated. Crossover occurs in implementing classes' constructors. | |
| PMX | Test the PMX method. | Inspection<br>PMX_Test.java | Tests conducted in PMX_Test.java<br><br>**crossOver_Test()** and **random_crossOver_Test():-**<br><br>These tests print out the parents followed by the produced offspring. In addition the cut points (array indexes) are printed out such that the crossovers can be analyzed by hand. | Passed.<br><br>PMX methods works. |
| OX | Test the OX method | Inspection<br>OX_Test.java | Tests conducted in OX_Test.java<br><br>**crossOver_Test()** and **random_crossOver_Test():-**<br><br>Same method as PMX testing. | Passed.<br><br>OX methods works. |
| CX | Test the CX method | Inspection<br>CX_Test.java<br><br>TestNG<br>CX_Test.java | Tests conducted in CX_Test.java<br><br>In contrast to the PMX and OX methods the CX method is not random so assertEquals (...) can be used.<br><br>**cxProduction_Test ():-**<br><br>Two parents and their expected offspring are already given. Perform crossover on the parents. Make sure offspring produced is the same as expected result by using assertEquals (...). | Passed.<br><br>CX methods works. |

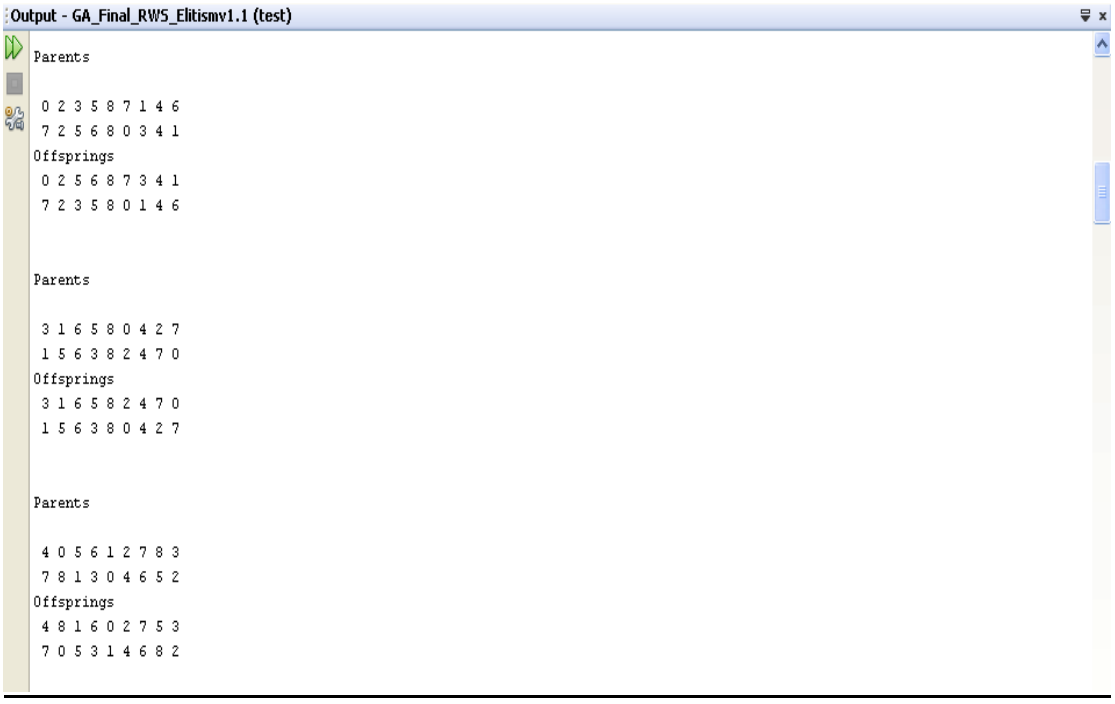
| | | | | |
|---|---|---|---|---|
| | | | **random_crossOver_Test ():-**<br><br>Randomly produced parents and their crossover performed. Available for inspection method of testing. | |

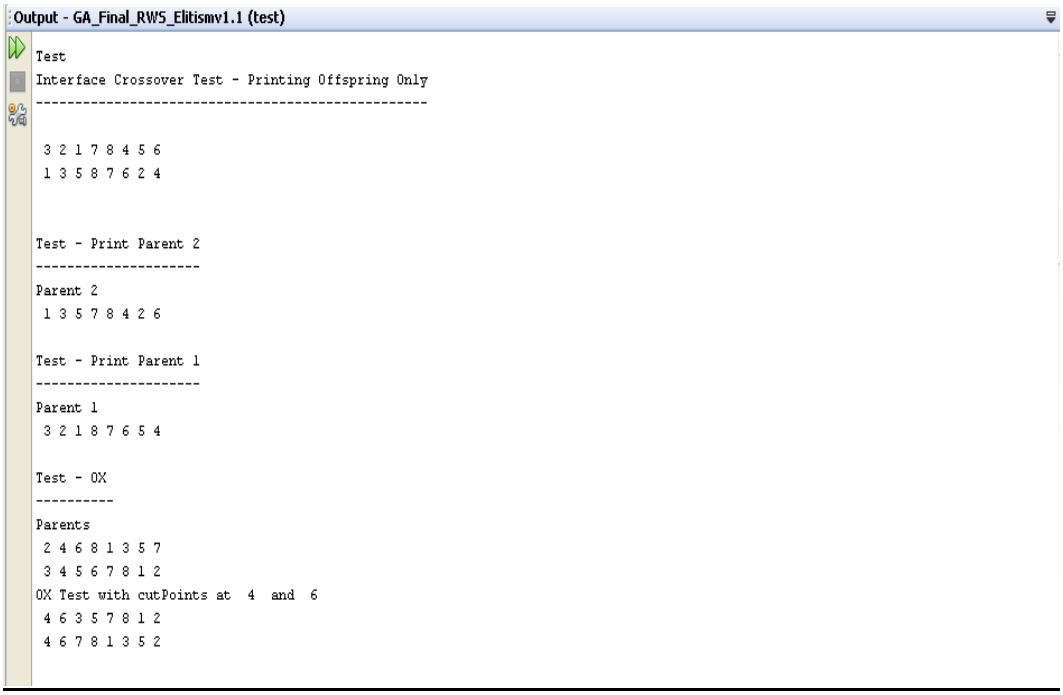
## TestNG Screenshot



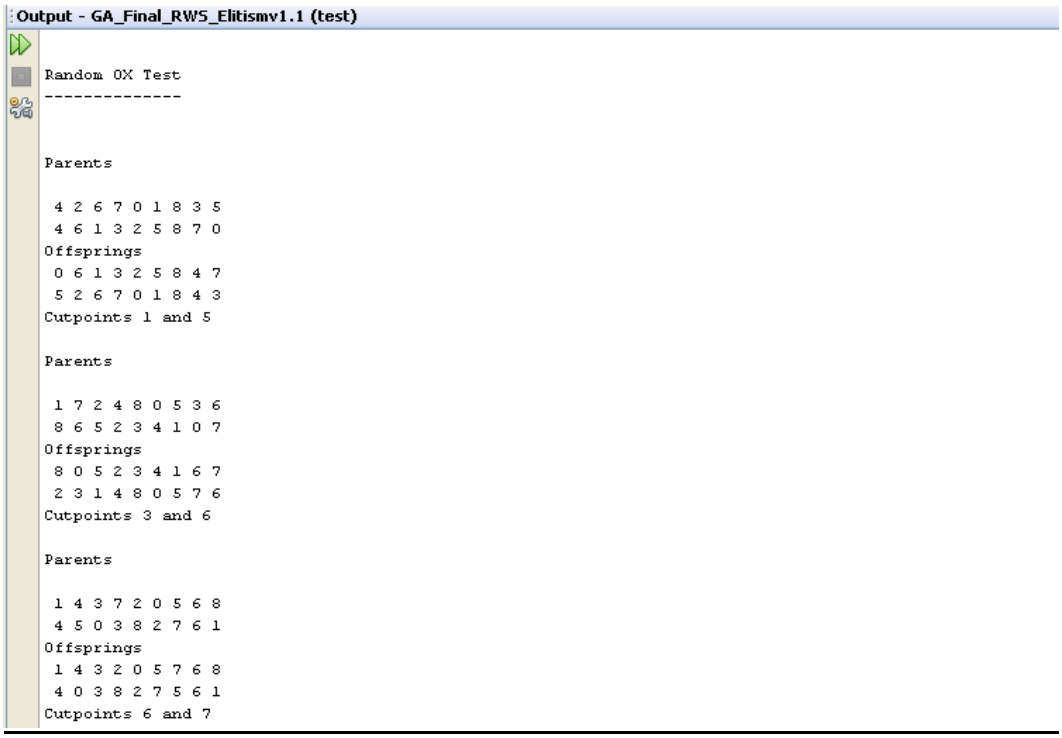
## Screenshots of Inspection Tests



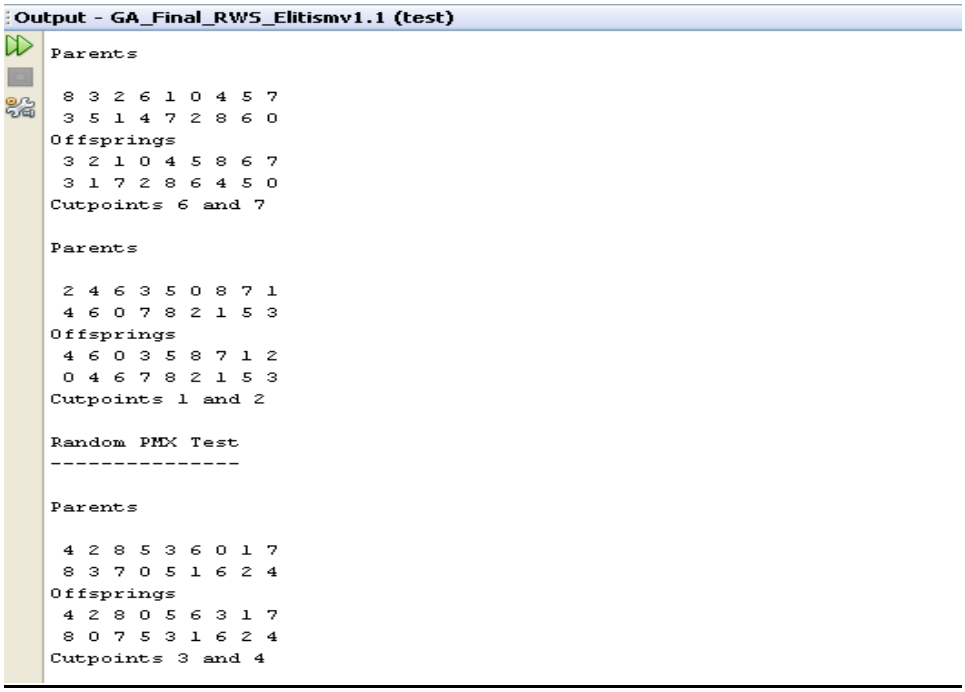





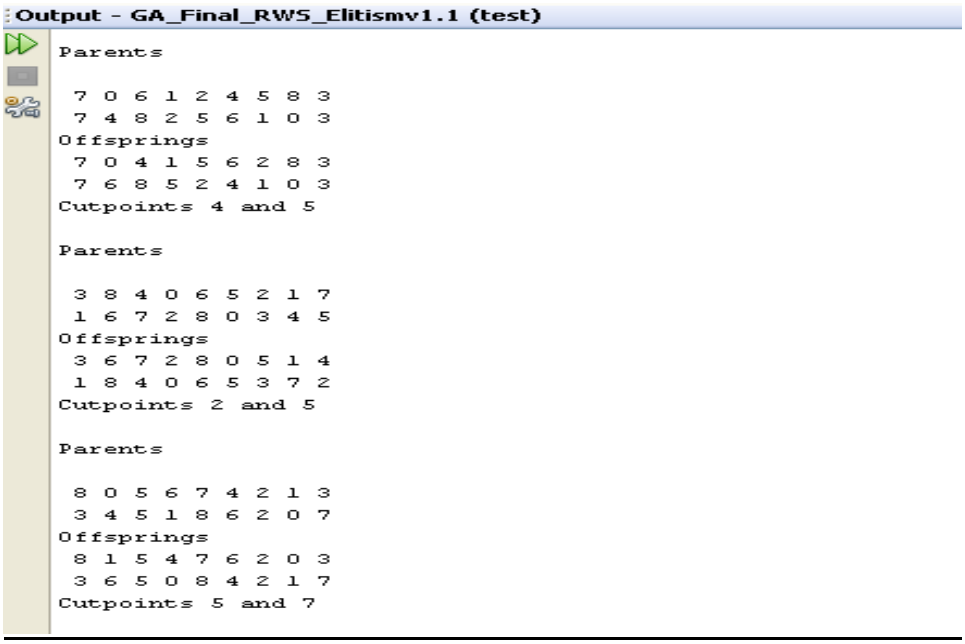
```
Output - GA_Final_RWS_Elitismv1.1 (test)
Test
Interface Crossover Test - Printing Offspring Only
--------------------------------------------------

 3 2 1 7 8 4 5 6
 1 3 5 8 7 6 2 4


Test - Print Parent 2
---------------------
Parent 2
 1 3 5 7 8 4 2 6

Test - Print Parent 1
---------------------
Parent 1
 3 2 1 8 7 6 5 4

Test - OX
----------
Parents
 2 4 6 8 1 3 5 7
 3 4 5 6 7 8 1 2
OX Test with cutPoints at  4  and  6
 4 6 3 5 7 8 1 2
 4 6 7 8 1 3 5 2
```



```
Output - GA_Final_RWS_Elitismv1.1 (test)

Random OX Test
--------------


Parents

 4 2 6 7 0 1 8 3 5
 4 6 1 3 2 5 8 7 0
Offsprings
 0 6 1 3 2 5 8 4 7
 5 2 6 7 0 1 8 4 3
Cutpoints 1 and 5

Parents

 1 7 2 4 8 0 5 3 6
 8 6 5 2 3 4 1 0 7
Offsprings
 8 0 5 2 3 4 1 6 7
 2 3 1 4 8 0 5 7 6
Cutpoints 3 and 6

Parents

 1 4 3 7 2 0 5 6 8
 4 5 0 3 8 2 7 6 1
Offsprings
 1 4 3 2 0 5 7 6 8
 4 0 3 8 2 7 5 6 1
Cutpoints 6 and 7
```



```
Output - GA_Final_RWS_Elitismv1.1 (test)
Parents

 8 3 2 6 1 0 4 5 7
 3 5 1 4 7 2 8 6 0
Offsprings
 3 2 1 0 4 5 8 6 7
 3 1 7 2 8 6 4 5 0
Cutpoints 6 and 7

Parents

 2 4 6 3 5 0 8 7 1
 4 6 0 7 8 2 1 5 3
Offsprings
 4 6 0 3 5 8 7 1 2
 0 4 6 7 8 2 1 5 3
Cutpoints 1 and 2

Random PMX Test
---------------

Parents

 4 2 8 5 3 6 0 1 7
 8 3 7 0 5 1 6 2 4
Offsprings
 4 2 8 0 5 6 3 1 7
 8 0 7 5 3 1 6 2 4
Cutpoints 3 and 4
```



```
Output - GA_Final_RWS_Elitismv1.1 (test)
Parents

 7 0 6 1 2 4 5 8 3
 7 4 8 2 5 6 1 0 3
Offsprings
 7 0 4 1 5 6 2 8 3
 7 6 8 5 2 4 1 0 3
Cutpoints 4 and 5

Parents

 3 8 4 0 6 5 2 1 7
 1 6 7 2 8 0 3 4 5
Offsprings
 3 6 7 2 8 0 5 1 4
 1 8 4 0 6 5 3 7 2
Cutpoints 2 and 5

Parents

 8 0 5 6 7 4 2 1 3
 3 4 5 1 8 6 2 0 7
Offsprings
 8 1 5 4 7 6 2 0 3
 3 6 5 0 8 4 2 1 7
Cutpoints 5 and 7
```

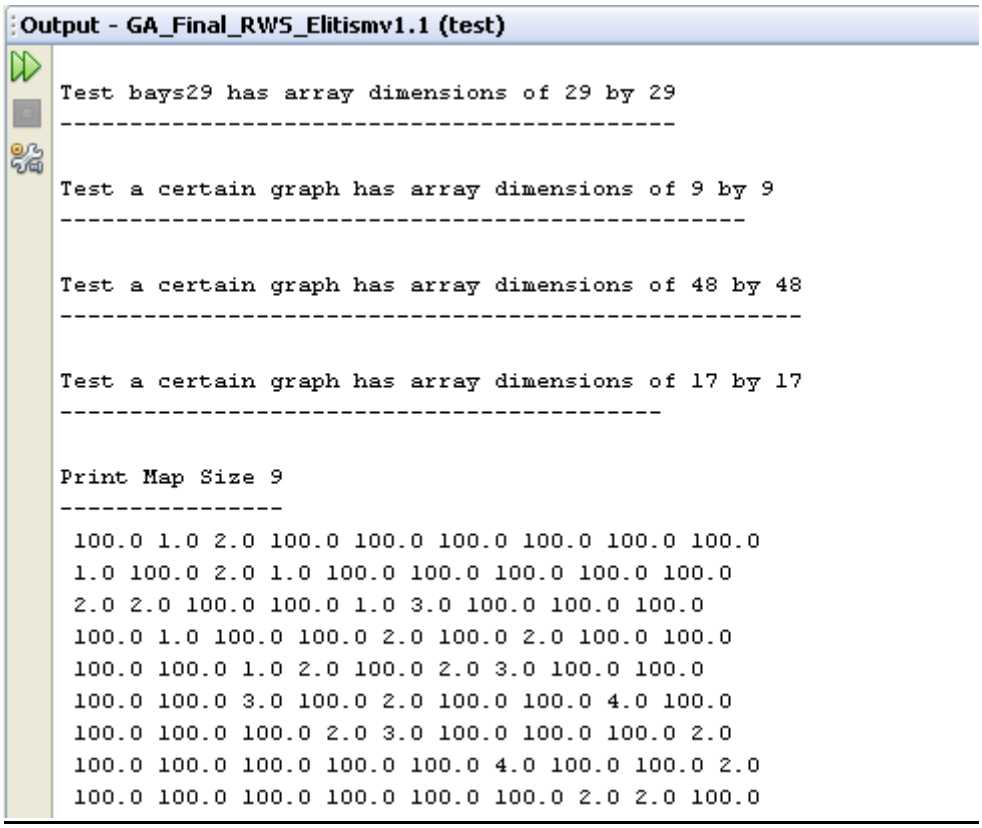
```
Parents

 1 4 3 7 8 0 2 6 5
 1 8 2 5 7 3 0 6 4
Offsprings
 1 4 2 7 8 3 0 6 5
 1 8 3 5 7 0 2 6 4
Cutpoints 5 and 6

Parents

 2 4 6 8 1 3 5 7
 3 4 5 6 7 8 1 2
Offsprings
 3 4 6 8 1 2 5 7
 2 4 5 6 7 8 1 3

PMX Segment Test 1
------------------
 Cutpoint1 0
 Cutpoint2 1
 2 4
PMX Segment Test 2
------------------
 Cutpoint1 0
 Cutpoint2 1
 3 4
```

Output - GA_Final_RWS_Elitismv1.1 (test)

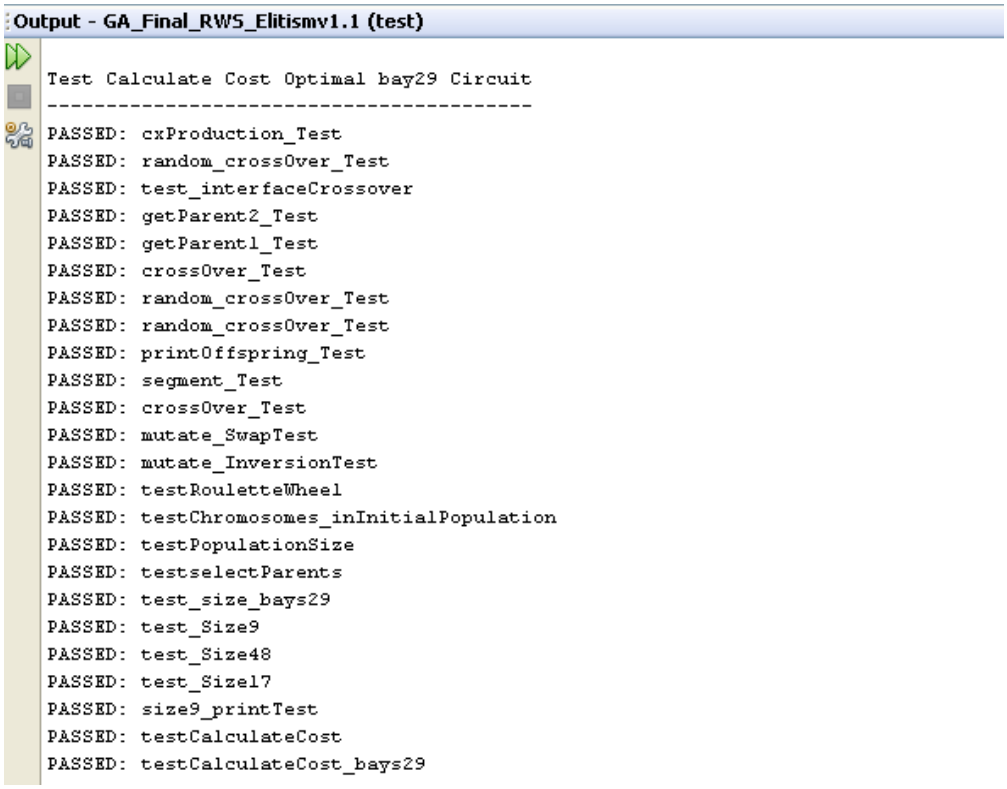
```
 3 4 5 6

PMX Test
--------
PMX Test with cutPoints at 0 and 3
Parents
 2 4 6 8 1 3 5 7
 3 4 5 6 7 8 1 2
Offspring

 3 4 5 6 1 2 8 7
 2 4 6 8 7 5 1 3

Mutate Test Swap
----------------

 1 3 5 6 7 2 4 8 0 Before
 3 1 5 6 7 2 4 8 0 After


Mutate Test Inversion
---------------------

 0 1 2 3 4 5 6 7 8 Before
 4 3 2 1 0 5 6 7 8 After
```

```
Output - GA_Final_RWS_Elitismv1.1 (test)

Test Roulette Wheel
--------------------
Total Fitness: 3615.0
0: 0.0
1: 0.0
2: 0.026832641770401105
3: 0.05394190871369295
4: 0.1078838174273859
5: 0.16182572614107885
6: 0.21604426002766253
7: 0.27026279391424624
8: 0.3247579529737207
9: 0.3792531120331951
10: 0.4603042876901799
11: 0.5416320885200554
12: 0.6489626556016598
13: 0.7562932226832642
14: 0.8650069156293223
15: 1.0
```

```
Output - GA_Final_RWS_Elitismv1.1 (test)

Test Chromosome
------------------
 3 5 8 4 0 7 1 6 2 Weight 900.0 Fitness 0.0 ID: 2
 3 2 5 0 7 6 1 4 8 Weight 803.0 Fitness 97.0 ID: 11
 8 4 5 1 6 0 3 7 2 Weight 802.0 Fitness 98.0 ID: 6
 7 5 1 6 2 4 0 8 3 Weight 705.0 Fitness 195.0 ID: 5
 1 7 8 2 3 0 4 6 5 Weight 705.0 Fitness 195.0 ID: 7
 6 3 0 8 4 5 1 7 2 Weight 704.0 Fitness 196.0 ID: 3
 3 1 4 8 5 2 6 7 0 Weight 704.0 Fitness 196.0 ID: 10
 6 5 1 0 4 7 8 3 2 Weight 703.0 Fitness 197.0 ID: 8
 3 8 5 6 0 4 2 1 7 Weight 703.0 Fitness 197.0 ID: 14
 7 1 2 3 4 6 5 8 0 Weight 607.0 Fitness 293.0 ID: 1
 6 5 2 1 0 3 8 4 7 Weight 606.0 Fitness 294.0 ID: 12
 6 4 1 8 0 7 5 2 3 Weight 512.0 Fitness 388.0 ID: 0
 7 5 8 3 2 1 0 6 4 Weight 510.0 Fitness 390.0 ID: 4
 8 2 1 0 6 3 5 7 4 Weight 509.0 Fitness 391.0 ID: 9
 8 6 4 1 3 5 7 0 2 Weight 412.0 Fitness 488.0 ID: 13


Test Select Parents
-------------------
Refer to Roulette Wheel produced in testRouletteWheel
Random No.1 for Parent 1 Selection: 0.7921114899132664
Random No.1 refers to Population Index: 13
Parent(ID): 9
```



```
Output - GA_Final_RWS_Elitismv1.1 (test)

Test bays29 has array dimensions of 29 by 29
---------------------------------------------

Test a certain graph has array dimensions of 9 by 9
---------------------------------------------------

Test a certain graph has array dimensions of 48 by 48
-----------------------------------------------------

Test a certain graph has array dimensions of 17 by 17
-----------------------------------------------

Print Map Size 9
----------------
 100.0 1.0 2.0 100.0 100.0 100.0 100.0 100.0 100.0
 1.0 100.0 2.0 1.0 100.0 100.0 100.0 100.0 100.0
 2.0 2.0 100.0 100.0 1.0 3.0 100.0 100.0 100.0
 100.0 1.0 100.0 100.0 2.0 100.0 2.0 100.0 100.0
 100.0 100.0 1.0 2.0 100.0 2.0 3.0 100.0 100.0
 100.0 100.0 3.0 100.0 2.0 100.0 100.0 4.0 100.0
 100.0 100.0 100.0 2.0 3.0 100.0 100.0 100.0 2.0
 100.0 100.0 100.0 100.0 100.0 4.0 100.0 100.0 2.0
 100.0 100.0 100.0 100.0 100.0 100.0 2.0 2.0 100.0
```



```
Output - GA_Final_RWS_Elitismv1.1 (test)

Test Calculate Cost Optimal bay29 Circuit
-----------------------------------------
PASSED: cxProduction_Test
PASSED: random_crossOver_Test
PASSED: test_interfaceCrossover
PASSED: getParent2_Test
PASSED: getParent1_Test
PASSED: crossOver_Test
PASSED: random_crossOver_Test
PASSED: random_crossOver_Test
PASSED: printOffspring_Test
PASSED: segment_Test
PASSED: crossOver_Test
PASSED: mutate_SwapTest
PASSED: mutate_InversionTest
PASSED: testRouletteWheel
PASSED: testChromosomes_inInitialPopulation
PASSED: testPopulationSize
PASSED: testselectParents
PASSED: test_size_bays29
PASSED: test_Size9
PASSED: test_Size48
PASSED: test_Size17
PASSED: size9_printTest
PASSED: testCalculateCost
PASSED: testCalculateCost_bays29
```

## Miscellaneous Results

### 1. Repeats of 5.1.1 and 5.1.2 - 300 Generations

These experiments examining the effects of OX (with or without mutation) on *bays29* but this time the GA runs for 300 Generations.

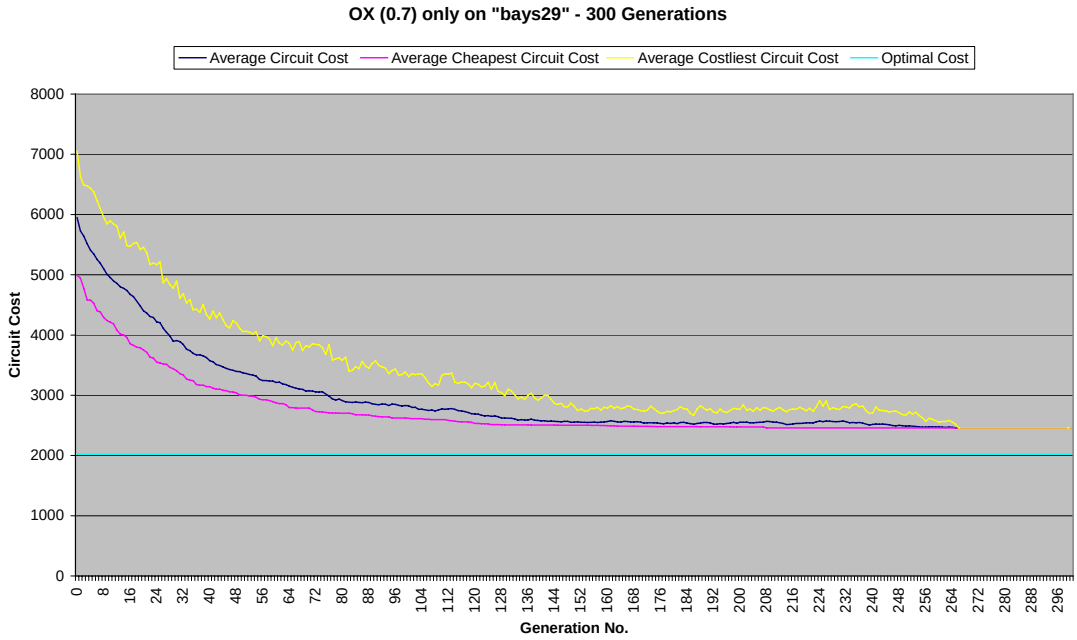
1. OX (0.7) only on *bays29* – 300 Generations

**Table 25 Parameters for 1 - 1**

| | |
|---|---|
| **Graph** | bays29 |
| **Population Size** | 61 |
| **Elites** | 1 |
| **Copy Probability** | 0.3 |
| **Crossover (OX) Probability** | 0.7 |
| **Mutation Probability** | 0.0 |
| **Generations** | 300 |

The cheapest circuit costs obtained from the 6 trials were 2347, 2565, 2384, 2726, 2370 and 2352. The average cheapest circuit cost of these 6 trials is 2457 and the standard deviation comes to 155. The average convergence rates are shown in Figure 25.

**Figure 25 (1 – 1) Average convergence of 6 GA runs**



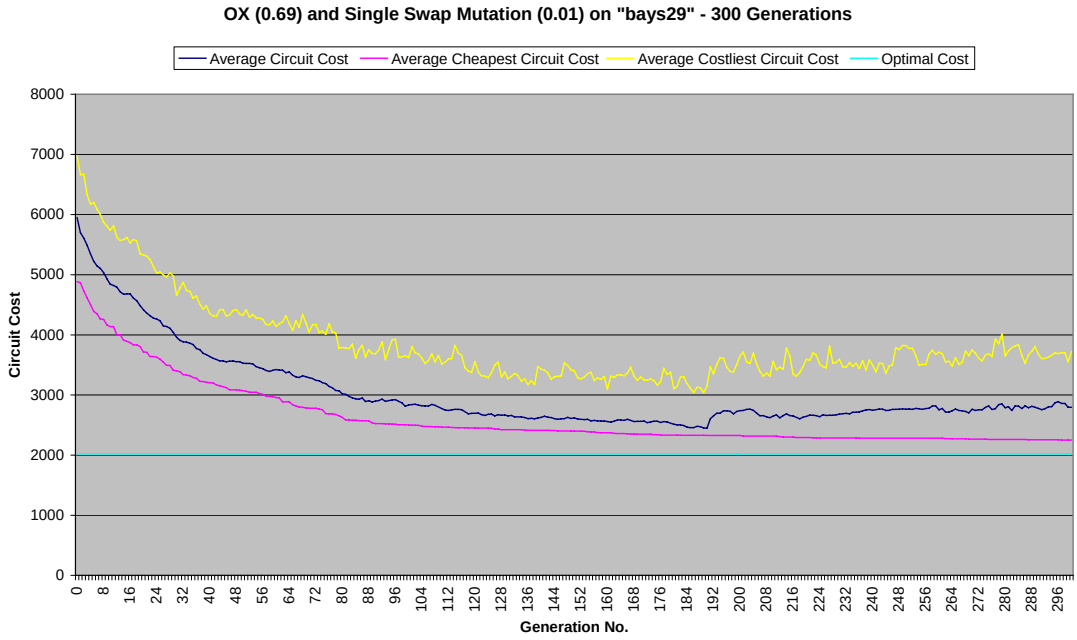
2. OX (0.69) and Single Swap Mutation (0.01) on *bays29* – 300 Generations

**Table 26 Parameters for 1 - 2**

| | |
|---|---|
| **Graph** | bays29 |
| **Population Size** | 61 |
| **Elites** | 1 |
| **Copy Probability** | 0.3 |
| **Crossover (OX) Probability** | 0.69 |
| **Mutation Probability (Single Swap)** | 0.01 |
| **Generations** | 300 |

The cheapest circuit costs obtained from the 6 trials were 2382, 2163, 2204, 2237, 2238 and 2269. The average cheapest circuit cost of these 6 trials is 2249 and the standard deviation comes to 75. The average convergence rates are shown in Figure 26.

**Figure 26 (1 - 2) Average convergence of 6 GA runs**



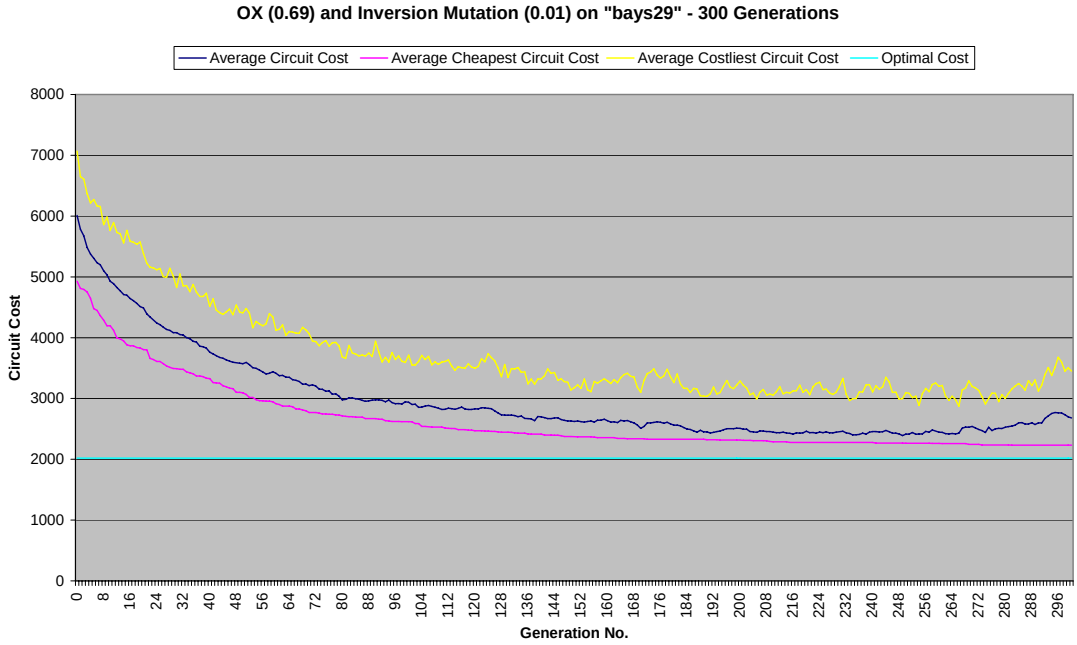

3. OX (0.69) and Inversion Mutation (0.01)

**Table 27 Parameters for 1 - 3**

| | |
|---|---|
| **Graph** | bays29 |
| **Population Size** | 61 |
| **Elites** | 1 |
| **Copy Probability** | 0.3 |
| **Crossover (OX) Probability** | 0.69 |
| **Mutation Probability (Inversion)** | 0.01 |
| **Generations** | 300 |

The cheapest circuit costs obtained from the 6 trials were 2242, 2177, 2158, 2316, 2166 and 2334. The average cheapest circuit cost of these 6 trials is 2232 and the standard deviation comes to 78. The average convergence rates are shown in Figure 27.

**Figure 27 (1 - 3) Average convergence of 6 GA runs**




## 2. CX and Mutation on *bays29*

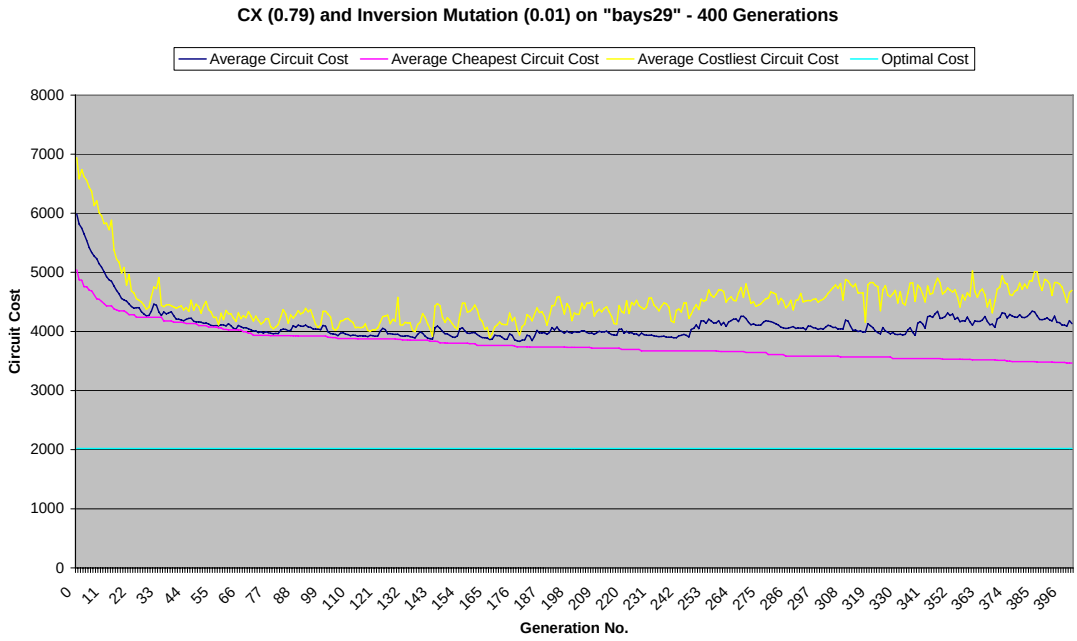
1. Single Swap Mutation (0.01) and CX (0.79) on *bays29*

**Table 28 Parameters for 2 - 1**

| | |
|---|---|
| **Graph** | bays29 |
| **Population Size** | 61 |
| **Elites** | 1 |
| **Copy Probability** | 0.2 |
| **Crossover (CX) Probability** | 0.79 |
| **Mutation Probability (Single Swap)** | 0.01 |
| **Generations** | 400 |

The cheapest circuit costs obtained from the 6 trials were 3720, 3934, 3398, 3531, 3475 and 3719. The average cheapest circuit cost of these 6 trials is 3630 and the standard deviation comes to 198. The average convergence rates are shown in Figure 28.

**Figure 28 (2 – 1) Average convergence of 6 GA runs**



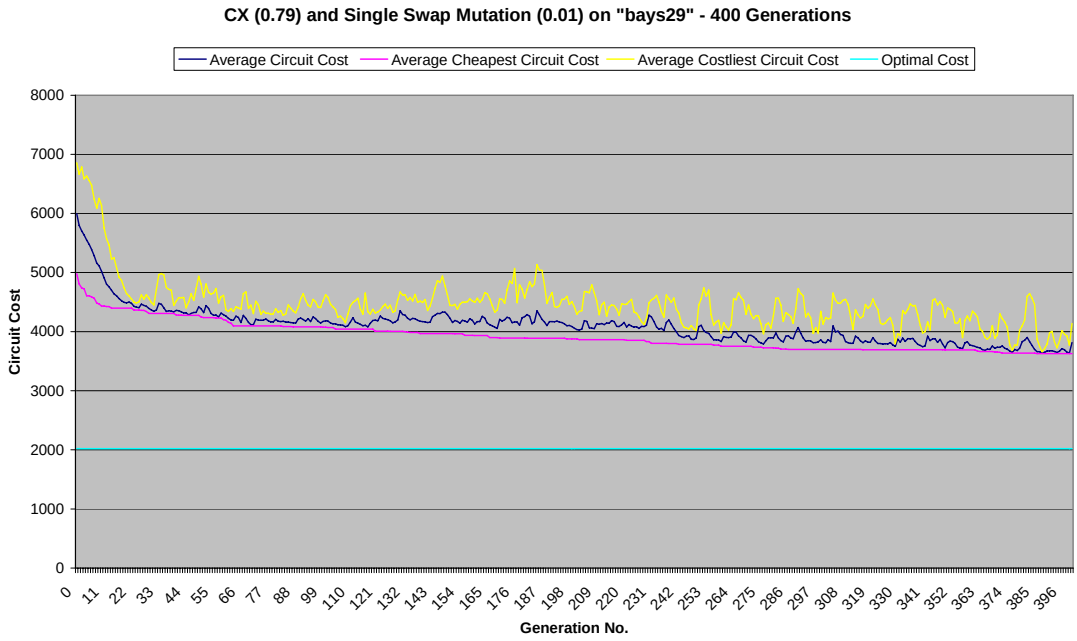

2. Inversion Mutation (0.01) and CX (0.79) on *bays29*

**Table 29 Parameters for 2 - 2**

| | |
|---|---|
| **Graph** | bays29 |
| **Population Size** | 61 |
| **Elites** | 1 |
| **Copy Probability** | 0.2 |
| **Crossover (CX) Probability** | 0.79 |
| **Mutation Probability (Single Swap)** | 0.01 |
| **Generations** | 400 |

The cheapest circuit costs obtained from the 6 trials were 3340, 3887, 3232, 3314, 4073 and 2951. The average cheapest circuit cost of these 6 trials is 3466 and the standard deviation comes to 425. The average convergence rates are shown in Figure 29.

**Figure 29 (2 – 2) Average convergence of 6 GA runs**




## 3. OX and Single Swap Mutation on *bays29* – 400 Generations

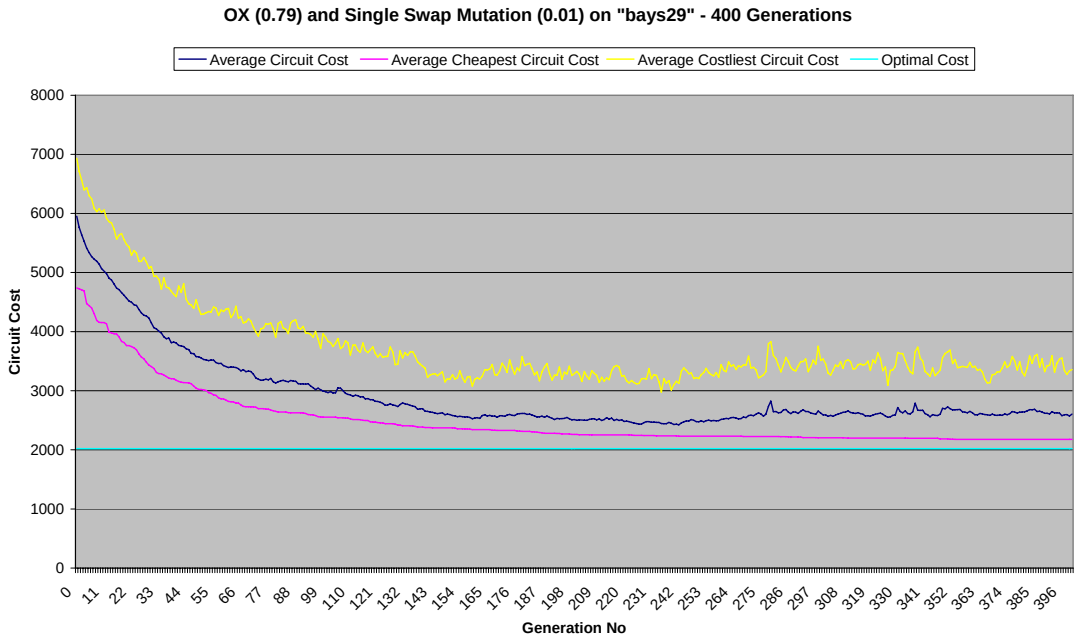
1. Single Swap Mutation (0.01) and OX (0.79) on *bays29* – 400 Generations

**Table 30 Parameters for 3 - 1**

| | |
|---|---|
| **Graph** | bays29 |
| **Population Size** | 61 |
| **Elites** | 1 |
| **Copy Probability** | 0.2 |
| **Crossover (OX) Probability** | 0.79 |
| **Mutation Probability (Single Swap)** | 0.01 |
| **Generations** | 400 |

The cheapest circuit costs obtained from the 6 trials were 2101, 2243, 2181, 2312, 2151 and 2074. The average cheapest circuit cost of these 6 trials is 2177 (on average 4% within optimum) and the standard deviation comes to 89. The average convergence rates are shown in the Figure 30.

**Figure 30 (3 – 1)**


OX (0.79) and Single Swap Mutation (0.01) on "bays29" - 400 Generations
Average Circuit Cost
Average Cheapest Circuit Cost
Average Costliest Circuit Cost
Optimal Cost
Circuit Cost
8000
7000
6000
5000
4000
3000
2000
1000
0
0
11
22
33
44
55
66
77
88
99
110
121
132
143
154
165
176
187
198
209
220
231
242
253
264
275
286
297
308
319
330
341
352
363
374
385
396
Generation No

## Project Plan

| | |
|---|---|
| **Student** | Peter L Tran |
| **Project Title** | Solving Travelling Salesman Problems using Genetic Algorithms |
| **Areas of Study** | Theoretical Computer Science (Primary)<br>Genetic Algorithms (Primary)<br>Techniques in Artificial Intelligence (Secondary) |
| **Supervisor** | Dr. Robin Hirsch |

**Aims and Objectives**

**Aim:**

To understand how Natural Computation, in particular Genetic Algorithms (GA) can be used to find solutions to the "Travelling Salesman Problem (TSP)". I am interested in how the various stages of GAs work and how to design an effective evolutionary algorithm for this purpose. I also aim to understand the biological inspirations to genetic algorithms for example genetics and natural evolution.

**Objectives:**

1. Review the TSP, particularly on why it is a hard problem and important one.
2. Review GAs and how some of their attributes are suited to solving the TSP.
3. Design and implement a computer program to represent the TSP.
4. Design and implement a genetic algorithm to find a near - optimal solution for visiting a set of TSP graph nodes once only.
5. Fully test the program.
6. To evaluate my GA and to be critical such that changes can be made to improve solutions. I will also be interested in exploring different techniques in the GAs' selection, crossover and mutation methods.
7. Implement such changes.
8. Gradually increasing the complexity of the TSP in context and re-evaluate GA.
9. Final review on GAs strengths and weaknesses for TSPs.

**Deliverables**

- A GA developed to solve the project's problem in context.
- Documentation of graphical, pictorial representation of the GAs performance on our data set. For example this could be a graph to show the mapping of the chromosomes on a fitness landscape, showing how much chromosomes converge to a point after crossover etc.
- A design specification for the software application.
- A fully documented and functional piece of software.
- A strategy for testing and evaluating the software.
- Project Plan (This) – **18th November 2009.**
- Interim Report – **27th January 2009.**
- Final Report, with copy of program on disk – **30th April 2010.**

**Work Plan**

Dates shown are estimates only but the various stages of the work plan are as accurate as possible.

**Background Reading and Feasibility Study**

12 October 2009 – 18 November 2009

- Literature search and review.
- Feasibility study of original project proposal. To include both theoretical and empirical (prototyping) considerations.

**System Requirements**

11th November 2009 – 18th November 2009

- System requirements of program. In the format of "MOSCOW" prioritization.

**System Analysis and Design – "Program Platform"**

18th November 2009 - 9th December 2009

- Continue with prototyping of SMALL instances of the problem.

- Mainly analysis and design the "Program Platform" i.e. Defining exactly how the Genetic Algorithms' attributes and entities would interact
- Specifying and designing how user would interact with the system.

**Implementation and Testing – "Program Platform"**

9th December 2009 – 21st December 2009

- Implementation of Program Platform. Unit and Functional Testing here.

**Design, Implementation and Evaluation of different GAs'**

21st December 2009 - 22nd February 2010

This section will be very iterative to suit my exploratory and experimental development of different GAs. The GAs will differ by having different selection, crossover and mutation strategies. The goal of this to find the best GA that will provide the best solutions to my problem specified.

For example GA 1 will have "Elitism" (Selection), "Edge Recombination" (Crossover) and some mutation method. GA 2 will have "Roulette Wheel" (Selection), "One Point" (Crossover) and some mutation method. GA 3 will have …. And so on.

All GAs will undergo routine unit and functional tests. In addition they will have testing in a more scientific nature. This testing must be fair. Scientific tests will be carried out after each creation of a GA, each test result for a GA will be compared with other existing GA test results. Results of tests will be communicated using suitable statistical methods. Results will help with my evaluation.

**Final Report**

22nd February 2010 – 31st March 2010

- Work on final report.

**Supervisor's Signature**

Dr. Robin Hirsch _____ Date ____ / ____ / ____

## Interim Report

**The Interim Report – 27th January 2010**

| | |
|---|---|
| **Name** | Peter Tran |
| **Project Title** | Solving The Travelling Salesman Problem using Genetic Algorithms |
| **Supervisor** | Dr. Robin Hirsch |

**Progress Made to Date**

Background Reading:

A sufficient amount of background reading has been undertaken. A draft version of my findings has been documented and was completed last term. A full list of bibliographic references has been maintained.

Design and Implementation:

A Genetic Algorithm has been created and is written in Java. It does the following:

- Inputs a complete graph which is stored on a .txt file. The graph is then represented using 2D - array.
- A random initial population of solutions is created at the first iteration. The population size is specified by the user.
- The solutions are represented by the Chromosome object; that stores the following information, the Hamiltonian circuit, the circuit cost and the associated fitness (calculate by the fitness function).
- The selection method used is the Roulette Wheel (fitness proportionate selection).
- Crossover methods implemented and to be tried out individually are Partially Mapped Crossover (PMX), Order Crossover (OX) and Cycle Crossover (CX).
- A mutation operator known as random swap has been implemented.
- The Genetic Algorithm runs for a set number of iterations.

- The following parameters can be configured; the graph to be used, Population Size, Crossover Rate, Mutation Rate and number of iterations required.

Unit Testing:

Methods and Classes have been tested, in particular the Crossover operators. Techniques used for testing include methods introduced earlier in the course such as TestNG (for test cases). Other techniques include for example printing out the results of applying a crossover operator to a pair of Chromosomes.

Functional Testing:

Putting all components together when running the algorithm for a graph with just crossover I have noticed that the average fitness increases as the number of iterations increases and as expected the GA converges to a local optimum.

**Further Work Required**

Testing and Evaluation

I am particularly interested in discussing the merits of the PMX, OX, CX operators in helping to find the nearest optimum solutions to a TSP. I would therefore have three different Genetic Algorithms distinguished by the crossover methods used. I can compare these algorithms by asking for example what parameter settings are required to find the best solution.

I am also interested to answer how these Crossover operators seem to contribute to producing better solutions.

TSP benchmarks will be used for these experiments. 1 suitable graph has been found which has 29 nodes used; other suitable graphs required may need formatting. The maximum number of nodes I would consider for this project would be around 50.

Another area of study can be the selection methods used, perhaps the copying method (known as elitism), tournament selection or stochastic universal sampling (variation of the Roulette Wheel).

All results of experiments need to be presented as statistical graphs.

Final Report

Sections Required:

1. Introduction
2. Background Information
3. Requirements and Analysis Design and Implementation
4. Testing
5. Evaluation
6. Bibliography – Currently maintained.
7. Appendices

**Supervisors Signature**

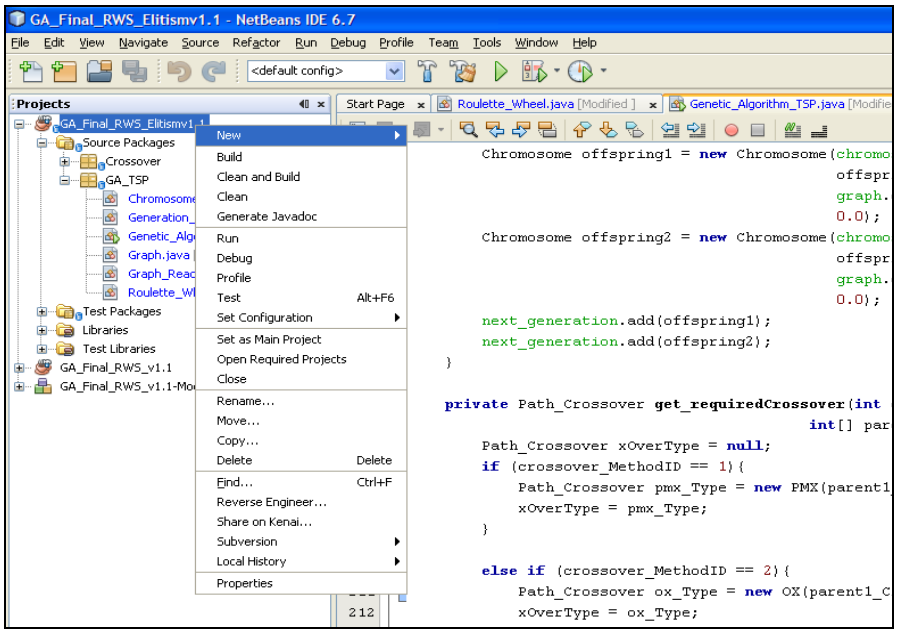
**Dr. Robin Hirsch** : _____ **Date:** _____

## User and System Manual

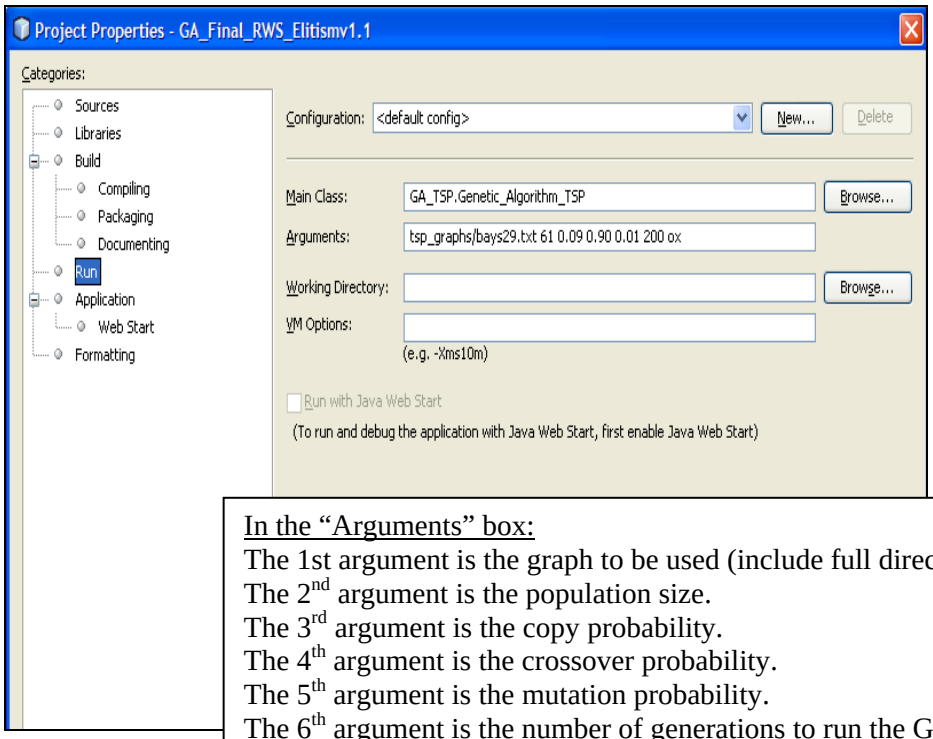
### Running GA_Final_RWS_Elitismv1.1

The recommendation is to run GA_Final_RWS_Elitismv1.1 in Netbeans IDE or a similar program (e.g. Eclipse). Netbeans IDE is available on UCL CS systems. To open GA-Final_RWS_Elitismv1.1, simple go to File > Open Project then select the required program.

To clean and build the project, simply choose the required options as shown below:




To set the GA parameters do the following as shown below after selecting "Properties" (see above screenshot):




In the "Arguments" box:
The 1st argument is the graph to be used (include full directory)
The 2nd argument is the population size.
The 3rd argument is the copy probability.
The 4th argument is the crossover probability.
The 5th argument is the mutation probability.
The 6th argument is the number of generations to run the GA.
The 7th argument is the crossover method (either "pmx", "ox", or "cx")

After parameters have been set, the GA is ready to run. This is done by selecting "run" see 1st screenshot. NOTE: Must ensure copy, crossover and mutation probabilities add up to 1.0.

## Changing the Mutation Operator

You will need to go into the class Genetic_Algorithm_TSP and under "mutate (….): void" method change the mutation operators called on "chrom1" and "chrom2". Then you will need to "clean and build" GA_Final_RWS_Elitismv1.1 again.

```
private void mutate(Chromosome chrom1, Chromosome chrom2){
    chrom1.mutateCircuit_SingleSwap();
    chrom2.mutateCircuit_SingleSwap();
    next_generation.add(chrom1);
    next_generation.add(chrom2);
}
```

## Adding a New Path_Crossover

The new path crossover will need to implement the Path_Crossover interface.

After this has been completed the new path crossover needs to be assigned with an integer id and a string id.

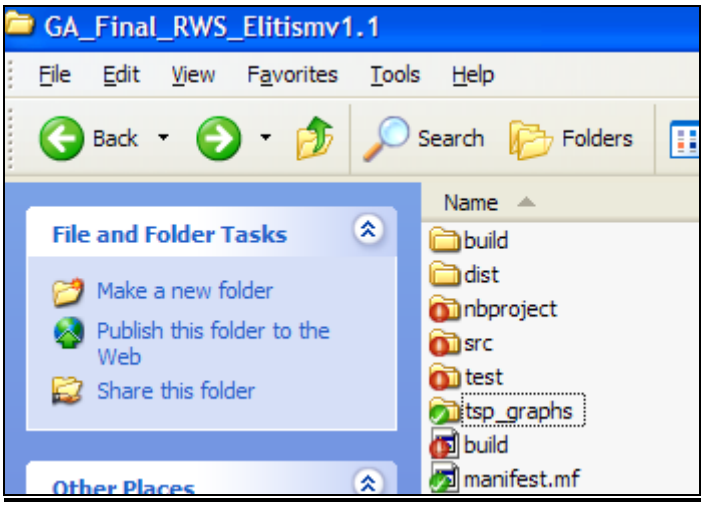
```
private int set_crossoverMethodID(){
    int xoverMethodID = -1;
    String pmx_Cross  = "pmx"; // 1 //
    String  ox_Cross  = "ox";  // 2 //
    String  cx_Cross  = "cx";  // 3 //

    if(crossover_Method.equals(pmx_Cross))     {  xoverMethodID = 1; }

    else if(crossover_Method.equals(ox_Cross)){  xoverMethodID = 2; }
    else if(crossover_Method.equals(cx_Cross)){  xoverMethodID = 3; }

    else { System.out.println("Unknown Crossover Method: " + xoverMethodID);
           System.exit(0);
    }

    return xoverMethodID;
}
```

Then an entry for the new path crossover needs to be in get_requiredCrossover (…..): Path_Crossover, such that the GA_Final_RWS_Elitismv1.1 knows exactly which Path_Crossover to return as specified by the user. The advantage of using the interface is that a crossover method doesn't need to be implemented for every recombination operator created.

```
private Path_Crossover get_requiredCrossover(int crossover_MethodID, int[] parent1_Circuit,
                                             int[] parent2_Circuit){
    Path_Crossover xOverType = null;
    if (crossover_MethodID == 1){
        Path_Crossover pmx_Type = new PMX(parent1_Circuit, parent2_Circuit);
        xOverType = pmx_Type;
    }

    else if (crossover_MethodID == 2){
        Path_Crossover ox_Type = new OX(parent1_Circuit, parent2_Circuit);
        xOverType = ox_Type;
    }

    else if (crossover_MethodID == 3){
        Path_Crossover cx_Type = new CX(parent1_Circuit, parent2_Circuit);
        xOverType = cx_Type;
    }

    return xOverType;
}
```

**Adding a Graph**

Firstly ensure that the graph is edited such that a full matrix of the distances between nodes is only shown. See the "bays29" screenshot. The graph needs to be saved as a ".txt" file in the folder "tsp_graphs" as shown below.



## Code Listings – GA_Final_RWS_Elitismv1.1

### Class: Chromosome

```
1  package GA_TSP;
2  /**
3   * @author Peter Tran
4   */
5  import java.util.Random;
6
7  public class Chromosome
8  {
9      private int chromosomeID;           // chromosome id
10     private int[] hamiltonianCircuit;   // a Hamiltonian Circuit //
11     private double weight;              // weight of circuit //
12     private double fitness;             // chromosome fitness //
13
14     public Chromosome(int chromosomeID, int [] hamiltonianCircuit,
15                                         double weight, double fitness){
16         this.chromosomeID = chromosomeID;
17         this.hamiltonianCircuit = hamiltonianCircuit;
18         this.weight = weight;
19         this.fitness = fitness;
20     }
21
22     public int getChromosomeID(){
23         return chromosomeID;
24     }
25
26     public int [] getHamiltonianCircuit(){
27         return hamiltonianCircuit;
28     }
29
30     public double getWeight(){
31         return weight;
```

```
32      }
33
34      public double getFitness(){
35          return fitness;
36      }
37
38      public void setFitness_PerGen(double fitness){
39          this.fitness = fitness;
40      }
41
42      // Mutation by single swap //
43      public void mutateCircuit_SingleSwap(){
44          Random randInt = new Random();
45          int index1st = randInt.nextInt(hamiltonianCircuit.length);
46          int index2nd = randInt.nextInt(hamiltonianCircuit.length);
47          int temp = hamiltonianCircuit[index1st];
48          hamiltonianCircuit[index1st] = hamiltonianCircuit[index2nd];
49          hamiltonianCircuit[index2nd] = temp;
50      }
51
52      public void mutateCircuit_Inversion(){
53          // Aim say p1 = { 0, 1, 2, 3, 4, 5, 6} //
54          //  mutate p1 = { 0, 1, 5, 4, 3, 2, 6} //
55          // i.e reverse ordering of nodes between 2 cutpoints //
56          Random randInt = new Random();
57          int index1st = randInt.nextInt(hamiltonianCircuit.length);
58          int index2nd = randInt.nextInt(hamiltonianCircuit.length);
59          while(index2nd == index1st){
60              index2nd = randInt.nextInt(hamiltonianCircuit.length);
61          }
62
63          if(index1st > index2nd){
64              int temp = index1st;
65              index1st = index2nd;
66              index2nd = temp;
67          }
68
```

```
69
70          while(true){
71              int test = index2nd - index1st;
72              // case p1 = { 0, 1, 2, 3, 4, 5, 6, 7 } reverse from 2 to 5 //
73              // when 3 and 4 is reached difference is 1, perform final swap then stop //
74              if(test == 1){
75                   int temp = hamiltonianCircuit[index1st];
76                   hamiltonianCircuit[index1st] = hamiltonianCircuit[index2nd];
77                   hamiltonianCircuit[index2nd] = temp;
78                   break;
79              }
80
81              else if(test == 2){
82              // case p1 = { 0, 1, 2, 3, 4, 5, 6, 7 } reverse from 2 to 6 //
83              // when 3 and 5 is reached difference is 2, perform final swap then stop //
84                  int temp = hamiltonianCircuit[index1st];
85                  hamiltonianCircuit[index1st] = hamiltonianCircuit[index2nd];
86                  hamiltonianCircuit[index2nd] = temp;
87                  break;
88                }
89
90              int temp = hamiltonianCircuit[index1st];
91              hamiltonianCircuit[index1st] = hamiltonianCircuit[index2nd];
92              hamiltonianCircuit[index2nd] = temp;
93              index1st++;
94              index2nd--;
95          }
96
97      }
98
99      public void printCircuit(){
100         for(int i = 0; i < hamiltonianCircuit.length; i++){
101             System.out.print(" " +hamiltonianCircuit[i]);
102         }
103     }
104 }
```

**Class: Generation_Info**

```
1  package GA_TSP;
3  /**
4   * @author Peter Tran
5   */
6  public class Generation_Info
7  {
8      private int generationNo;
9
10     private double average_CircuitCost;
11     private double cheapest_CircuitCost;
12     private double mostExpensive_CircuitCost;
13     private int number_ofCopies;
14     private int number_ofMates;
15     private int number_ofMutes;
16
17     public Generation_Info(int generationNo,       double average_CircuitCost,
18                                                    double cheapest_CircuitCost,
19                                                    double mostExpensive_CircuitCost,
20                                                    int number_ofCopies,
21                                                    int number_ofMates,
22                                                    int number_ofMutes){
23
24         this.generationNo = generationNo;
25         this.average_CircuitCost  = average_CircuitCost;
26         this.cheapest_CircuitCost = cheapest_CircuitCost;
27         this.mostExpensive_CircuitCost = mostExpensive_CircuitCost;
28         this.number_ofCopies = number_ofCopies;
29         this.number_ofMates = number_ofMates;
30         this.number_ofMutes = number_ofMutes;
31     }
32
33     public void print_Generation_Info(){
34         System.out.print(" " + generationNo);
```

```
35          System.out.print(" " + average_CircuitCost);
36          System.out.print(" " + cheapest_CircuitCost);
37          System.out.print(" " + mostExpensive_CircuitCost);
38          System.out.print(" " + number_ofCopies);
39          System.out.print(" " + number_ofMates);
40          System.out.print(" " + number_ofMutes);
41      }
42  }
```

**Class: Graph**

```
 1  package GA_TSP;
 2  /**
 3   * @author Peter Tran
 4   */
 5  import java.util.ArrayList;
 6  import java.util.Random;
 7
 8  public class Graph
 9  {
10      private double[][] weighted_Graph;
11
12      private Graph_Reader graph_Reader;
13
14      public Graph(String filename){
15          graph_Reader = new Graph_Reader(filename);
16          graph_Reader.create_Graph();
17          weighted_Graph = graph_Reader.get_weightedGraph();
18      }
19
20      public int [] random_hamiltonianCircuit(){
21
22          int [] hamiltonianCircuit = new int[weighted_Graph.length];
23          ArrayList<Integer> nodes_notPicked = new ArrayList<Integer>();
24
25          // Add all Nodes (labelled 0 to |N-1|) //
```

```
26          // to nodes_notPicked: ArrayList<Integer> //
27
28          for(int start_Node = 0; start_Node < weighted_Graph.length; start_Node++){
29              nodes_notPicked.add(start_Node);
30          }
31
32          // Randomly pick a node from nodes_notPicked
33          // Add this node to hamiltonianCircuit[nodePosition]
34
35          for(int nodePosition = 0;
                    nodePosition < weighted_Graph.length; nodePosition++){
36              Random rand_nextNode = new Random();
37              int nodeSelect = rand_nextNode.nextInt(nodes_notPicked.size());
38              hamiltonianCircuit[nodePosition] = nodes_notPicked.get(nodeSelect);
39              nodes_notPicked.remove(nodeSelect);
40          }
41
42          return hamiltonianCircuit;
43      }
44
45      public double calculateCost(int [] hamiltonianCircuit){
46          double cost = 0.0;
47          for(int currentNode = 0;
48                  currentNode < hamiltonianCircuit.length - 1; currentNode++){
49
50              cost = cost + get_edgeWeight(hamiltonianCircuit[currentNode],
51                                           hamiltonianCircuit[currentNode + 1]);
52          }
53          // Find edge weight from last node in sequence back to start
54          // and add to cost
55          cost = cost + get_edgeWeight(hamiltonianCircuit[(hamiltonianCircuit.length)- 1], hamiltonianCircuit[0]);
57          return cost;
58      }
59
60       private double get_edgeWeight(int first_nodeNumber, int second_nodeNumber){
61          double weight = weighted_Graph[first_nodeNumber][second_nodeNumber];
62          return weight;
```

```
63      }
64  }
```

**Class: Graph_Reader**

```
 1  package GA_TSP;
 2  /*
 3   * @author Peter Tran
 4   */
 5  import java.io.BufferedReader;
 6  import java.io.DataInputStream;
 7  import java.io.FileInputStream;
 8  import java.io.InputStreamReader;
 9  import java.util.StringTokenizer;
10  import java.util.ArrayList;
11
12  public class Graph_Reader
13  {
14      private int num_Rows;                // number of lines in the file //
15      private double[][] weighted_graph;
16      private ArrayList<String> store_lineList; //stores each line of a file as a string  //
17      private ArrayList<Double> distance_betweenNodes;
18
19      // The constructor creates a 2D matrix weighted_graph //
20      // Each line in the file is store as a String in "store_lineList" //
21
22      public Graph_Reader(String filename){
23          num_Rows = 0;
24          store_lineList = new ArrayList<String>();
25          distance_betweenNodes = new ArrayList<Double>(); // TEMP //
26          try {
27              FileInputStream fileInputStream = new FileInputStream(filename);
28              DataInputStream dataInputStream = new DataInputStream(fileInputStream);
29              BufferedReader bufferedReader   = new BufferedReader
30                                      (new InputStreamReader(dataInputStream));
31              // Read Line by Line
```

```
32              String stringLine;
33              while((stringLine = bufferedReader.readLine()) != null){
34                  store_lineList.add(stringLine);
35                  num_Rows++;
36              }
37              dataInputStream.close();
38
39          }
40          catch(Exception exception){
41              System.out.println("ERROR: " + exception.getMessage());
42          }
43
44          weighted_graph = new double[num_Rows][num_Rows];
45      }
46
47      public void create_Graph(){
48          // Each string in store_lineList can be broken up into words //
49          // i.e. in the Graph's context each word is a distance between a pair of nodes //
50
51
52          ArrayList<Object> temp = new ArrayList<Object>();
53          for(int i = 0; i < store_lineList.size(); i++){
54              StringTokenizer lineParser = new StringTokenizer(store_lineList.get(i));
55              while(lineParser.hasMoreTokens()){
56                  temp.add(lineParser.nextElement());
57              }
58          }
59
60          // these words as store as Double in "temp" ArrayList //
61          for(int x = 0; x < temp.size(); x++){
62              Object obj = temp.get(x);
63              String str_distance = obj.toString();
64              double distance = Double.parseDouble(str_distance);
65              distance_betweenNodes.add(distance);
66          }
67
68          // fill weighted_graph with "temp" entries //
```

```
69          int tempIndex = 0;
70          for(int row = 0; row < weighted_graph.length; row++){
71              for(int column = 0; column < weighted_graph.length; column++){
72                 weighted_graph[row][column] = distance_betweenNodes.get(tempIndex);
73                  tempIndex++;
74              }
75          }
76      }
77
78      public double[][] get_weightedGraph(){
79          return weighted_graph;
80      }
81
82  }
```

**Class: Roulette_Wheel**

```
 1  package GA_TSP;
 2
 3  import java.util.ArrayList;
 4  import java.util.Random;
 5
 6  /**
 7   * @author Peter Tran
 8   */
 9
10  // Implements the roulette wheel required for each generation //
11
12  public class Roulette_Wheel
13  {
14      private double [] cumulative_prob_ofSelection;
15      private double totalFitness;
16      private int index_ofParent;      // Changes When pickChromosome is Called //
17      private double randomProb;
18
```

```
19
20      public Roulette_Wheel(ArrayList<Chromosome> population){
21          randomProb = 0.0;
22          index_ofParent = -1;
23          totalFitness = 0.0;
24          cumulative_prob_ofSelection = new double[population.size() + 1];
25
26          // Calculate Total Fitness //
27          for(Chromosome ch: population){
28             totalFitness = totalFitness + ch.getFitness();
29          }
30
31          int next_index = 1; // current  Index //
32          int prev_index = 0; // previous Index //
33          cumulative_prob_ofSelection[0]= 0.0;
34
35          // If totalFitness = 0.0 then shouldn't divide by 0.0 //
36          if(totalFitness == 0.0){
37              for(Chromosome chrom: population){
38                  cumulative_prob_ofSelection[next_index] = 0.0;
39                  next_index++;
40              }
41          }
42
43          else {
44          // Now Calculate Cumulative Fitness of Chromosomes //
45              for(Chromosome chrom: population){
46                  double selectionProb = chrom.getFitness()/totalFitness;
47                  cumulative_prob_ofSelection[next_index] =
48                            cumulative_prob_ofSelection[prev_index] + selectionProb;
49                  prev_index++;
50                  next_index++;
51              }
52          }
53      }
54
55      // Chromosome is Picked, Index of Parent is Calculated //.
```

```
56      public void pickChromosome(){
57          Random double_generator = new Random();
58          randomProb = double_generator.nextDouble();
59          index_ofParent = getLocation_ofParent(randomProb);
60      }
61
62      private int getLocation_ofParent(double randomDouble){
63          int location_ofParent = 0; // Default - Required if RW Entries are 0.0
64                                     // as explained earlier //
65          for(int index = 0; index < cumulative_prob_ofSelection.length - 1; index++){
66              if((randomDouble >= cumulative_prob_ofSelection[index]) &&
67                 (randomDouble <= cumulative_prob_ofSelection[index + 1])){
68                  location_ofParent = index;
69                  break;
70              }
71          }
72          return location_ofParent;
73      }
74
75      public int get_indexOfParent(){ return index_ofParent; }
76
77      public double [] get_rouletteWheel(){ return cumulative_prob_ofSelection; }
78
79      // For Testing Purposes Only
80      public double get_totalFitness(){ return totalFitness; }
81
82      // For Testing Purposes Only //
83      public double get_randomNo(){ return randomProb; }
84
85  }
```

**Interface: Path_Crossover**

```
 1  package Crossover;
 2
 3  /**
```

```
 * @author Peter Tran
 */
public interface Path_Crossover
{

    public int[] get_offspring1();
    public int[] get_offspring2();

    public int[] get_parent1();
    public int[] get_parent2();

    public void crossOver(int[] offSpring, int[] parentX, int[] parentY);

    // Used in Testing //
    public void printOffspring(int [] offspring1, int [] offspring2);
}
```

**Class: PMX**

```
package Crossover;
/**
 * @author Peter Tran
 */

import java.util.Random;

public class PMX implements Path_Crossover
{
    private int[] parent1;
    private int[] parent2;
    private int[] offspring1;
    private int[] offspring2;
    private int[] segment1;
    private int[] segment2;
    private int   cutPoint1;
    private int   cutPoint2;
```

```
18
19      public PMX(int [] parent1, int [] parent2){
20          this.parent1 = new int[parent1.length];
21          this.parent2 = new int[parent2.length];
22          for(int index = 0; index < parent1.length; index ++){
23             this.parent1[index] = parent1[index];
24             this.parent2[index] = parent2[index];
25          }
26          Random firstRNum  = new Random();
27          Random secondRNum = new Random();
28
29          // special value randomNo_Boundray required
30          // as firstRNum.nextInt(parent1.length) generates a random number
31          // from >=0 <= parent1.length, number used as index. However parent1.length is
32          // never an array index as aray index are numbered 0 to (parent1.length - 1)
33
34          int randomNo_Boundary = (parent1.length) - 1;
35          offspring1 = new int[parent1.length];
36          offspring2 = new int[parent2.length];
37          cutPoint1 = firstRNum.nextInt(randomNo_Boundary);
38          cutPoint2 = secondRNum.nextInt(randomNo_Boundary);

39          while(cutPoint1 == cutPoint2){
40              // Make sure cutPoints are not identical to each other //
41              cutPoint2 = secondRNum.nextInt(randomNo_Boundary);
42          }
43          if(cutPoint1 > cutPoint2){
44               int temp = cutPoint1;    // Make sure CutPoint1 is greater than
45              cutPoint1 = cutPoint2;    // cutPoint2 //
46              cutPoint2 = temp;
47          }
48          create_Segments(cutPoint1, cutPoint2);
49          crossOver(offspring1, parent1, parent2);
50          crossOver(offspring2, parent2, parent1);
51
52      }
53
```

```
54      public int get_cutPoint1()   { return cutPoint1;  } // For Testing Purposes //
55      public int get_cutPoint2()   { return cutPoint2;  } // For Testing Purposes //
56
57      public int[] get_segment1()  { return segment1;   } // For Testing Purposes //
58      public int[] get_segment2()  { return segment2;   } // For Testing Purposes //
59
60      public int[] get_parent1()   { return parent1;    }
61      public int[] get_parent2()   { return parent2;    }
62
63      public int[] get_offspring1(){ return offspring1; }
64      public int[] get_offspring2(){ return offspring2; }
65
66      // For an Element given by its index check that it doesn't appear twice //
67      private boolean check_forDuplicates(int [] offspring, int indexOfElement){
68          for(int index = 0; index < offspring.length; index++){
69              if((offspring[index] == offspring[indexOfElement]) &&
70                      (indexOfElement != index) ){
71                  return true;
72              }
73          }
74          return false;
75      }
76
77      // If Element is Duplicated, replace it by using its mapping //
78      private void sort_Duplicates(int [] offspring, int indexOfElement){
79          for(int index = 0; index < segment1.length; index++){
80              if(segment1[index] == offspring[indexOfElement]){
81                  offspring[indexOfElement] = segment2[index];
82              }
83              else if(segment2[index] == offspring[indexOfElement]){
84                  offspring[indexOfElement] = segment1[index];
85              }
86          }
87      }
88
89      private void create_Segments(int cutPoint1, int cutPoint2){
90          int capacity_ofSegments = (cutPoint2 - cutPoint1) + 1;
```

```
91          segment1 = new int[capacity_ofSegments];
92          segment2 = new int[capacity_ofSegments];
93          int segment1and2Index = 0;
94          for(int index = 0; index < parent1.length; index++){
95            if((index >= cutPoint1) && (index <= cutPoint2)){
96               int x = parent1[index];  int y = parent2[index];
97               segment1[segment1and2Index] = x;
98               segment2[segment1and2Index] = y;
99               segment1and2Index++;
100           }
101         }
102     }
103
104     private void insert_Segments(int[] offspring, int[] segment){
105         int segmentIndex = 0;
106         for(int index = 0; index < offspring.length; index++){
107            if((index >= cutPoint1) && (index <= cutPoint2)){
108                offspring[index] = segment[segmentIndex];
109                segmentIndex++;
110            }
111        }
112    }
113
114    // offspring2 gets segment 1, offspring1 gets segment2 //
115    public void crossOver(int [] offspring, int[] parentX, int[] parentY){
116        if(offspring == offspring1){
117            int[] segment = segment2;
118            insert_Segments(offspring, segment);
119        }
120
121        else if(offspring == offspring2){
122            int [] segment = segment1;
123            insert_Segments(offspring, segment);
124        }
125
126        for(int index = 0; index < offspring.length; index++){
127            if((index < cutPoint1) || (index > cutPoint2)){
```

```
128                 offspring[index] = parentX[index];
129             }
130         }
131 
132         for(int index = 0; index < offspring.length; index++){
133             if((index < cutPoint1) || (index > cutPoint2)){
134                 while(check_forDuplicates(offspring, index)){
135                     sort_Duplicates(offspring, index);
136                 }
137             }
138         }
139     }
140 
141      public void printOffspring(int [] offspring1, int [] offspring2){
142               // Basically Prints Offspring… See Source Code for Details //
166      }
167 
168 }
```

**Class: OX**

```
 2 package Crossover;
 3 
 4 import java.util.Random;
 5 import java.util.ArrayList;
 6 /**
 7  * @author Peter Tran
 8  */
 9 public class OX implements Path_Crossover
10 {
11     private int[] parent1;
12     private int[] parent2;
13     private int[] offspring1;
14     private int[] offspring2;
15     private int cutPoint1;
```

```
16      private int cutPoint2;
17
18      private ArrayList<Integer> outerSegmentBuildArray;
19
20      public OX(int [] parent1, int [] parent2){
21          outerSegmentBuildArray = new ArrayList<Integer>();
22          this.parent1 = new int[parent1.length];
23          this.parent2 = new int[parent2.length];
24
25          for(int index = 0; index < parent1.length; index ++){
26             this.parent1[index] = parent1[index];
27             this.parent2[index] = parent2[index];
28          }
29
30          offspring1 = new int[parent1.length];
31          offspring2 = new int[parent2.length];
32          Random cP1 = new Random();
33          Random cP2 = new Random();
34          // Generate Random cut points, must be unique from each other //
35          // cutPoint2 should be greater than cutPoint1 //
36          int length = parent1.length - 1;
37          cutPoint1 = cP1.nextInt(length);
38          cutPoint2 = cP2.nextInt(length);
39
40          while(cutPoint2 == cutPoint1){
41               cutPoint2 = cP2.nextInt(length);
42          }
43
44          if(cutPoint1 > cutPoint2){
45              int temporary = cutPoint1;
46              cutPoint1 = cutPoint2;
47              cutPoint2 = temporary;
48           }
49
50          crossOver(offspring1, parent1, parent2);
51          crossOver(offspring2, parent2, parent1);
52      }
```

```
53
54      public int[] get_parent1(){ return parent1; }
55
56      public int[] get_parent2(){ return parent2; }
57
58      public int[] get_offspring1(){ return offspring1; }
59
60      public int[] get_offspring2(){ return offspring2; }
61
62      public int get_cutPoint1(){ return cutPoint1; } // FOR TESTING PURPOSES //
63
64      public int get_cutPoint2(){ return cutPoint2; } // FOR TESTING PURPOSES //
65
66
67      private void remove_SpecifiedElement(int elementToRemove){
68          for(int index = 0; index< outerSegmentBuildArray.size(); index++){
69              if(outerSegmentBuildArray.get(index) == elementToRemove){
70                  outerSegmentBuildArray.remove(index);
71                  break;
72              }
73          }
74      }
75
76      public void crossOver(int [] offspring, int [] parentX, int [] parentY){
77          int tempIndex = 0;
78          int index = cutPoint2 + 1;
79          // if index - cutPoint2 + 1  == parentX.length
80          // add all parentX elements directly to  outerSegmentBuildArray ArrayList.
81          if(index == parentX.length) { // e.g. (1 2 3 | 4 5 6 7 8| )
82              for(int x = 0; x < parentX.length; x++){
83                  outerSegmentBuildArray.add(parentX[x]);
84              }
85          }
86
87         // Else block here concatenates segments in the following order 3rd then (1 and 2)
88         // outerSegmentBuildArray
89          else {
```

```
90              for(index = cutPoint2 + 1; index < parentX.length; index++){
91                  outerSegmentBuildArray.add(tempIndex, parentX[index]);
92                  tempIndex++;
93              }
94              for(index = 0; index <= cutPoint2; index++){
95                  outerSegmentBuildArray.add(tempIndex, parentX[index]);
96                  tempIndex++;
97              }
98
99          }
100
101
102         for(int indexInSegment = cutPoint1; indexInSegment <=cutPoint2; indexInSegment++){
103             // for ArrayList temp remove elements that appear in parentY mid segments
104             remove_SpecifiedElement(parentY[indexInSegment]);
105         }
106
107         for(int x = cutPoint1; x <= cutPoint2; x++){
108             // copy mid segment from parent designated as Y,
109             // into offspring to be created.
110             offspring[x] = parentY[x];
111         }
112
113
114         // Belows section copies remaining elements in temp into offspring
115         // starting from 3rd segment of offspring.
116         tempIndex = 0;
117         for(int y = cutPoint2 + 1; y < offspring.length; y++){
118             if(y == offspring.length){ break; }
119             offspring[y] = outerSegmentBuildArray.get(tempIndex);
120             tempIndex++;
121         }
122
123         // after end of offspring reach, copy elements from temp haven't been copied
124         // into offspring from 1st segment.
125         for(int z = 0; z < cutPoint1; z++){
126             if(z == offspring.length){ break; }
127             offspring[z] = outerSegmentBuildArray.get(tempIndex);
```

```
128             tempIndex++;
129         }
130     }
131
132     // Used for Testing //
133     public void printOffspring(int [] offspring1, int [] offspring2){
134         // Basically Prints Offspring… See Source Code for Details //
157     }
158 }
```

**Class: CX**

```
 1 package Crossover;
 3 /**
 4  * @author Peter Tran
 5  */
 7 public class CX implements Path_Crossover
 8 {
 9     private int[] parent1;
10     private int[] parent2;
11     private int[] offspring1;
12     private int[] offspring2;
13
14     public CX(int[] parent1, int[] parent2){
15         this.parent1 = new int[parent1.length];
16         this.parent2 = new int[parent2.length];
17         for(int index = 0; index < parent1.length; index ++){
18            this.parent1[index] = parent1[index];
19            this.parent2[index] = parent2[index];
20         }
21         offspring1 = new int[parent1.length];
22         offspring2 = new int[parent2.length];
23         for(int index = 0; index < offspring1.length; index++){
24             offspring1[index] = -1;
25             offspring2[index] = -1;
26         }
```

```
27          crossOver(offspring1, parent1, parent2);
28          crossOver(offspring2, parent2, parent1);
29
30      }
31
32      public int[] get_offspring1(){  return offspring1; }
33      public int[] get_offspring2(){  return offspring2; }
34      public int[] get_parent1()   {  return parent1;    }
35      public int[] get_parent2()   {  return parent2;    }
36
37
38      // (1 x x 5 ) eg. element to search is 5 in 1st parent after 1 matches to 5..
39      // (5 x x x )  // its position in parent 1 is 3.
40
41      private int getPosition_ofSecondParentElement_infirstParent
42                                      (int [] firstParent, int element_toSearch){
43          int position = 0;
44          for(int index = 0; index < parent1.length; index++){
45              if(firstParent[index] == element_toSearch){
46                 position = index;
47                 break;
48              }
49          }
50          return position;
51      }
52
53      // (1 x x 5 ) eg. element to search is 1, after look for it in 2nd parent.
54      // (5 x x 1 )  // 1 has already been filled so return true.
55
56      private boolean element_already_inOffspring(int [] offspring, int element){
57          for(int index = 0; index < offspring.length; index++){
58              if(offspring[index] == element){
59                  return true;
60              }
61          }
62          return false;
63      }
```

```
64
65      public void crossOver(int [] offspring, int [] parentX, int [] parentY){
66          int index = 0;
67          while(!element_already_inOffspring(offspring, parentY[index])){
68             offspring[index] = parentX[index];
69             int position = getPosition_ofSecondParentElement_infirstParent
70                                                    (parentX, parentY[index]);
71             offspring[position] = parentY[index];
72             index = position;
73          }
74
75          for(int offspring_index = 0; offspring_index < offspring.length; offspring_index++){
76              if(offspring[offspring_index] == -1){
77                  offspring[offspring_index] = parentY[offspring_index];
78              }
79          }
80      }
81
82       // For Testing //
83       public void printOffspring(int [] offspring1, int [] offspring2){
84          // Basically Prints Offspring… See Source Code for Details //
107      }
108 }
```

**Class: Genetic_Algorithm_TSP**

```
1  package GA_TSP;
2
3  /**
4   * @author Peter Tran
5   */
6
7  import java.util.ArrayList;
8  import java.util.Random;
9  import Crossover.Path_Crossover;
10 import Crossover.PMX;
```

```
11 import Crossover.OX;
12 import Crossover.CX;
13
14 public class Genetic_Algorithm_TSP
15 {
16     private final Graph graph; // 1st Argument is the file containing the graph //
17     private final int population_Size;                    // 2nd Argument  //
18     private final double cumulative_copying_Prob;         // 3rd Agrument  //
19     private final double cumulative_crossover_Prob;       // 4th Argument  //
20     private final double cumulative_mutation_Prob;        // 5th Argument  //
21     private final int max_numIterations;                  // 6th Argument  //
22     private final String crossover_Method;                 // 7th Argument  //
23     private int generationNumber;
24     private ArrayList<Chromosome> population;          // population, updated every gen //
25     private ArrayList<Chromosome> selectionPool;       // selection pool //
26     private ArrayList<Chromosome> next_generation;
27     private ArrayList<Generation_Info> generationInfo;    // Generation Information //
28     private final int crossover_MethodID;
29     private int chromosomeID_Counter;
30     private double av_circuitCost_PerGen;
31     private double cheapest_circuitCost_PerGen;
32     private int number_CopiesPerGen;
33     private int number_MatesPerGen;
34     private int number_MutesPerGen;
35     private double costliest_circuitCost_PerGen;
36
37     public Genetic_Algorithm_TSP(String file_withGraph, int population_Size,
38                                  double copying_Prob,    double crossover_Prob,
39                                  double mutation_Prob,   int max_numIterations,
40                                  String crossover_Method){
41         graph = new Graph(file_withGraph);
42         this.population_Size = population_Size;
43
44         // Set Cumulative Values for Copying, Crossover, Mutation Probabilities //
45         cumulative_copying_Prob   = copying_Prob;
46         cumulative_crossover_Prob = cumulative_copying_Prob + crossover_Prob;
```

```
47          cumulative_mutation_Prob  = cumulative_crossover_Prob + mutation_Prob;
48
49          this.max_numIterations = max_numIterations;
50          this.crossover_Method  = crossover_Method;
51          generationNumber = 0;
52          population  = new ArrayList<Chromosome>();
53          selectionPool = new ArrayList<Chromosome>();
54          next_generation = new ArrayList<Chromosome>();
55          generationInfo  = new ArrayList<Generation_Info>();
56          crossover_MethodID = set_crossoverMethodID();
57          chromosomeID_Counter = 0;
58          av_circuitCost_PerGen = 0.0;
59          cheapest_circuitCost_PerGen = 0.0;
60          number_CopiesPerGen = 0;
61          number_MatesPerGen = 0;
62          number_MutesPerGen = 0;
63          costliest_circuitCost_PerGen = 0;
64
65          // Create Initial Population //
66          while(chromosomeID_Counter != population_Size){
67              int[] randCircuit = graph.random_hamiltonianCircuit();
68              double circuit_cost = graph.calculateCost(randCircuit);
69              Chromosome chromosome = new Chromosome(chromosomeID_Counter, randCircuit,
70                                                     circuit_cost, 0.0);
71              population.add(chromosome);
72              chromosomeID_Counter++;
73          }
74          insertionSort_Population();
75          calculate_costliest_circuitCost_PerGen();
76          updateAll_chromosomesFitness_PerGen();
77          calculate_av_circuitWeight_PerGen();
78          calculate_cheapest_circuitCost_PerGen();
79          Generation_Info genInfo = new Generation_Info(generationNumber,
80                                                        av_circuitCost_PerGen,
81                                                        cheapest_circuitCost_PerGen,
82                                                        costliest_circuitCost_PerGen,
83                                                        number_CopiesPerGen,
```

```
84                                                      number_MatesPerGen,
85                                                      number_MutesPerGen);
86          generationInfo.add(genInfo);
87      }
88
89      private void calculate_costliest_circuitCost_PerGen(){
90         // Chromsome with Most Expensive circuit is
91         // first element of ArrayList<Chromosome> population.
92         // (Assumming that population has been sorted using insertionSort(...).
93         costliest_circuitCost_PerGen = population.get(0).getWeight();
94      }
95
96      private void updateAll_chromosomesFitness_PerGen(){
97         for(Chromosome chrom: population){
98              double fitness = (costliest_circuitCost_PerGen) - chrom.getWeight();
99              chrom.setFitness_PerGen(fitness);
100        }
101     }
102
103     private int set_crossoverMethodID(){
104         int xoverMethodID = -1;
105         String pmx_Cross  = "pmx"; // 1 //
106         String  ox_Cross  = "ox";  // 2 //
107         String  cx_Cross  = "cx";  // 3 //
108
109         if(crossover_Method.equals(pmx_Cross))    {  xoverMethodID = 1; }
110
111         else if(crossover_Method.equals(ox_Cross)){  xoverMethodID = 2; }
112         else if(crossover_Method.equals(cx_Cross)){  xoverMethodID = 3; }
113
114         else { System.out.println("Unknown Crossover Method: " + xoverMethodID);
115                System.exit(0);
116         }
117
118         return xoverMethodID;
119     }
120
```

```
121     public ArrayList<Chromosome> get_population(){
122         return population;
123     }
124
125     public ArrayList<Chromosome> get_matingPairs(){
126         return selectionPool;
127     }
128
129     public ArrayList<Chromosome> get_next_generation(){
130         return next_generation;
131     }
132
133     public ArrayList<Generation_Info> get_generationInfo(){
134         return generationInfo;
135     }
136
137     public int get_max_numIterations(){ return max_numIterations; }
138
139     public void selection(){
140         Roulette_Wheel rws = new Roulette_Wheel(population);
141         int num_spins = 0;
142         int num_spinsRequired = population_Size - 1;
143         // Due to Elitism = 1, num_spinsRequired is population_Size - 1 //
144         while(num_spins < num_spinsRequired){
145             rws.pickChromosome();
146             selectionPool.add(population.get(rws.get_indexOfParent()));
147             num_spins++;
148         }
149     }
150
151     public void evolve_Population(){
152         // Perform Elitism  //
153         elitism();
154         Random randDouble = new Random();
155         // Generate random double [0.0, 1.0] //
156         for(int index = 0; index < selectionPool.size(); index = index + 2){
157             double evolChoice = randDouble.nextDouble();
```

```
158
159             // Copy Option //
160             if((evolChoice >= 0.0)&&(evolChoice <= cumulative_copying_Prob)){
161                 copy(selectionPool.get(index), selectionPool.get(index + 1));
162                 number_CopiesPerGen = number_CopiesPerGen + 2;
163             }
164
165             // Recombination (Crossover) Option //
166             else if((evolChoice >= cumulative_copying_Prob)
167                             &&(evolChoice <= cumulative_crossover_Prob)){
168                 crossover(selectionPool.get(index), selectionPool.get(index + 1));
169                 number_MatesPerGen = number_MatesPerGen + 2;
170             }
171
172             // Mutation Option //
173             else if((evolChoice >= cumulative_crossover_Prob)
174                             &&(evolChoice <= cumulative_mutation_Prob)){
175                 mutate(selectionPool.get(index), selectionPool.get(index + 1));
176                 number_MutesPerGen = number_MutesPerGen + 2;
177             }
178         }
179     }
180
181     private void elitism(){
182         // Save Top Chromosome in Current Generation //
183         next_generation.add(population.get(population_Size - 1));
184
185     }
186
187     private void copy(Chromosome chrom1, Chromosome chrom2){
188         next_generation.add(chrom1);
189         next_generation.add(chrom2);
190     }
191
192     private void crossover(Chromosome chrom1, Chromosome chrom2){
193         Path_Crossover xover_method = get_requiredCrossover(crossover_MethodID,
194                                                             chrom1.getHamiltonianCircuit(),
195                                                             chrom2.getHamiltonianCircuit());
```

```
196
197         int[] offspring1_Circuit = xover_method.get_offspring1();
198         int[] offspring2_Circuit = xover_method.get_offspring2();
199         // Create offspring chromosomes, fitness is initially set to 0.0,
200         // but will be updated when all Chromosome making up the
201         // new generation has been determined.
202         Chromosome offspring1 = new Chromosome(chromosomeID_Counter++,
203                                                offspring1_Circuit,
204                                                graph.calculateCost(offspring1_Circuit),
205                                                0.0);
206         Chromosome offspring2 = new Chromosome(chromosomeID_Counter++,
207                                                offspring2_Circuit,
208                                                graph.calculateCost(offspring2_Circuit),
209                                                0.0);
210         next_generation.add(offspring1);
211         next_generation.add(offspring2);
212     }
213
214     private Path_Crossover get_requiredCrossover(int crossover_MethodID,
215                                                  int[] parent1_Circuit,
216                                                  int[] parent2_Circuit){
217         Path_Crossover xOverType = null;
218         if (crossover_MethodID == 1){
219             Path_Crossover pmx_Type = new PMX(parent1_Circuit, parent2_Circuit);
220             xOverType = pmx_Type;
221         }
222
223         else if (crossover_MethodID == 2){
224             Path_Crossover ox_Type = new OX(parent1_Circuit, parent2_Circuit);
225             xOverType = ox_Type;
226         }
227
228         else if (crossover_MethodID == 3){
229             Path_Crossover cx_Type = new CX(parent1_Circuit, parent2_Circuit);
230             xOverType = cx_Type;
231         }
232
```

```
233         return xOverType;
234     }
235 
236     private void mutate(Chromosome chrom1, Chromosome chrom2){
237         chrom1.mutateCircuit_Inversion();
238         chrom2.mutateCircuit_Inversion();
239         next_generation.add(chrom1);
240         next_generation.add(chrom2);
241     }
242 
243     private void reinsertion(){
244         population = null;
245         population = next_generation;
246         next_generation = new ArrayList<Chromosome>();
247         selectionPool = new ArrayList<Chromosome>();
248     }
249 
250     private void calculate_av_circuitCost_PerGen(){
251         double totalcircuitWeight = 0.0;
252         for(Chromosome chrom : population){
253             totalcircuitWeight = totalcircuitWeight + chrom.getWeight();
254         }
255         av_circuitCost_PerGen =  Math.round(totalcircuitWeight/population.size());
256     }
257 
258     private void calculate_cheapest_circuitCost_PerGen(){
259         cheapest_circuitCost_PerGen = population.get(population_Size - 1).getWeight();
260     }
261 
262     private void insertionSort_Population(){
263     // Reference http://www.dreamincode.net/code/snippet516.htm
264     // Sorts population, such that Individual with
265     // cheapest circuit weight is at the bottom.
266 
267         double temp;
268         int index;
269         for(int position = 1; position < population.size(); position++){
```

```
270             if(population.get(position).getWeight()
271                             > population.get(position - 1).getWeight())
272             {
273                temp = population.get(position).getWeight();
274                Chromosome chromTemp = population.get(position);
275                index = position;
276                  do {
277                        population.set(index, population.get(index - 1));
278                        index--;
279                     }
280                  while((index > 0) && (population.get(index-1).getWeight() < temp));
281                population.set(index, chromTemp);
282             }
283         }
284     }
285
286     private static int string_toInteger(String args){
287         String argument = args;
288         int int_ofString = Integer.parseInt(argument);
289         return int_ofString;
290     }
291
292     private static double string_toDouble(String args){
293         String argument = args;
294         double double_ofString = Double.parseDouble(argument);
295         return double_ofString;
296     }
297
298     private void print_Population(){
299         System.out.println("");
300         System.out.println("Generation Number: " + generationNumber);
301         System.out.println("");
302         System.out.println("");
303         for(Chromosome ch: population){
304             System.out.print("  Circuit: ");
305             ch.printCircuit();
306             System.out.print("  Weight: " + ch.getWeight());
```

```
307             System.out.print("  Fitness: " + ch.getFitness());
308             System.out.print("  ID: " + ch.getChromosomeID());
309             System.out.println("");
310         }
311         System.out.println("Copies in Generation: "+ number_CopiesPerGen);
312         System.out.println("Mates in Generation: " + number_MatesPerGen);
313         System.out.println("Mutations in Generation: " + number_MutesPerGen);
314         System.out.println("Average Circuit Cost: " + av_circuitCost_PerGen);
315         System.out.println("BEST CHROMOSOME (ID): " +
316                            population.get(population.size()-1).getChromosomeID());
317     }
318
319     private void reset_numberOf_Copies_Mates_Mutes(){
320         number_CopiesPerGen = 0;
321         number_MatesPerGen  = 0;
322         number_MutesPerGen  = 0;
323     }
324
325     public static void main(String[] args){
326         String dataFile     = args[0];
327         String popArgument  = args[1];
328         String copyRate     = args[2];
329         String xoverRate    = args[3];
330         String muteRate     = args[4];
331         String numGenerations   = args[5];
332         String crossover_Method = args[6];
333
334         Genetic_Algorithm_TSP gaTSP = new Genetic_Algorithm_TSP(dataFile,
335                                                 string_toInteger(popArgument),
336                                                 string_toDouble(copyRate),
337                                                 string_toDouble(xoverRate),
338                                                 string_toDouble(muteRate),
339                                                 string_toInteger(numGenerations),
340                                                 crossover_Method);
341         System.out.println("Graph: " + dataFile);
342         System.out.println("Population Size: " + popArgument);
343         System.out.println("Copying Rate: "   + copyRate);
344         System.out.println("Crossover Prob: " + xoverRate);
```

```
345         System.out.println("Mutation Prob: " + muteRate);
346         System.out.println("Number of Generations to Run: " + numGenerations);
347         System.out.println("Crossover Method: " + crossover_Method);
348         gaTSP.print_Population();
349         int iter = 0;
350         while(iter < gaTSP.get_max_numIterations())
351         {
352             gaTSP.generationNumber++;
353             gaTSP.selection();
354             gaTSP.evolve_Population();
355             gaTSP.reinsertion();
356             gaTSP.insertionSort_Population();
357 
358             gaTSP.calculate_costliest_circuitCost_PerGen();
359             gaTSP.updateAll_chromosomesFitness_PerGen();
360             gaTSP.calculate_av_circuitWeight_PerGen();
361             gaTSP.calculate_cheapest_circuitCost_PerGen();
362             gaTSP.print_Population();
363             Generation_Info genInfo = new Generation_Info(gaTSP.generationNumber,
364                                                           gaTSP.av_circuitCost_PerGen,
365                                                           gaTSP.cheapest_circuitCost_PerGen,
366                                                           gaTSP.costliest_circuitCost_PerGen,
367                                                           gaTSP.number_CopiesPerGen,
368                                                           gaTSP.number_MatesPerGen,
369                                                           gaTSP.number_MutesPerGen);
370             gaTSP.get_generationInfo().add(genInfo);
371             gaTSP.reset_numberOf_Copies_Mates_Mutes();
372             iter++;
373         }
374         System.out.println(" Generation Statistics: ");
375         System.out.println(" Column 1: Gen Number - ");
376         System.out.println(" Column 2: Av Circuit Cost/Gen - ");
377         System.out.println(" Column 3: Cheapest Circuit Cost/Gen - ");
378         System.out.println(" Column 4: Most Expensive Circuit Cost/Gen - ");
379         System.out.println(" Column 5: Number Copies/Gen - ");
380         System.out.println(" Column 6: Number Mates/Gen - ");
381         System.out.println(" Column 7: Number Mutes/Gen - ");
382 
```

```
383         for(Generation_Info genInfo : gaTSP.get_generationInfo()){
384             genInfo.print_Generation_Info();
385             System.out.println("");
386         }
387 
388     }
390 }
```