

**FIT3036 PROJECT:**  
***INVESTIGATION OF GRAPH COLOURING AND  
GENETIC ALGORITHMS***

**Author: Jesse Chung**

**Student ID: 20279736**

MONASH UNIVERSITY, Clayton School of Information Technology Student Declaration for FIT3036 Submission. I, *Jesse Chung*, ID: 20279736 declare that this submission is my own work and has not been copied from any other source without attribution. I acknowledge that severe penalties exist for any copying of code without attribution, including a mark of 0 for this assessment.

## TABLE OF CONTENTS

<b>INTRODUCTION .....</b>	<b>3</b>
<b>ALGORITHM DESCRIPTIONS .....</b>	<b>3</b>
GENETIC ALGORITHM .....	3
NON GENETIC ALGORITHM .....	4
RANDOM GRAPH GENERATOR ALGORITHM .....	4
<b>PROJECT DOCUMENTATION .....</b>	<b>5</b>
USER INTERFACE/DOCUMENTATION .....	5
IMPLEMENTATION/TECHNICAL DOCUMENTATION .....	12
<b>EVIDENCE OF TESTING .....</b>	<b>17</b>
<b>ACHIEVEMENTS .....</b>	<b>18</b>
METHOD .....	18
RESULTS AND DISCUSSION .....	18
<b>CONCLUSION .....</b>	<b>21</b>
<b>BIBLIOGRAPHY .....</b>	<b>22</b>

## INTRODUCTION

The purpose for this task is to implement and investigate genetic algorithms for graph colouring. Graph colouring is labelling elements in a graph as colours, which are constrained so that no two elements adjacent to each other can be co-chromatic; for the task involved, this colouring is known as vertex colouring. However, the focus of this project will be on two-colouring, thus allowing improper colouring; where co-chromatic adjacent vertices exist.

A genetic algorithm is a search technique comparable to an evolutionary system[4], where it takes a population of individuals (in this case they are random colourings of a certain graph structure), and go through a process of mutation, crossover, fitness evaluation and selection, which are iterated until a certain requirement is met.

The purpose of the genetic algorithm is to attempt to achieve a solution that is a perfect colouring; where no two vertices are co-chromatic; or an improper colouring which reduce the size and number of monochromatic components (a subset of co-chromatic vertices which are connected in the graph, or chromon for short [1]) within a satisfactory timeframe.

The focus for this task is two-colouring of a graph of no more than four incident edges. A graph colouring with a non genetic algorithm will also be implemented and analysed. Both algorithms will be evaluated on its quality of solution and its run time.

## ALGORITHM DESCRIPTIONS

This section describes how each algorithm within this project is implemented.

### GENETIC ALGORITHM

The genetic algorithm implemented can be broken down into several compositions.

1. It first initialises a population of a user-defined size (or default size) of possible solutions (individuals of random colouring) to create diversity for the algorithm to run on.
2. An evaluation on each individual will be performed, measuring its chromon numbers and largest chromon found in the individual and assigning each with a fitness value calculated with a fitness function.
3. The algorithm will then store the fittest individual from the current population.
4. It will then perform mutation. The process of mutation consists of a user-changeable probability that a random vertex will change to its opposite colour in an individual, where a smoothing function is then applied to that vertex, to attempt to remove any large chromons that exists within the individual.
5. The crossover operation is then run in the algorithm. The crossover process begins with the breeding of a number of individuals within a population. The selection of

“parents” is random, though the user can change the probability which a parent is chosen within the population. If there is only one parent and the population has checked all individuals, then that parent is paired with the fittest individual.

Both parents are then compared, and if they are dissimilar one of the parent’s colouring are inverted. A child is created by keeping similarities in both parents (keeping the vertices which are the same colours from both parents), and a random colour is chosen for any vertices which do not match in both parents. At any vertices which a random colour is chosen, the smoothing function is run on it. The child produced will replace one of the parents, taking its place within the population.

6. After mutation and crossover, a selection of individuals for a new population begins. The way in which individuals are selected is based on fitness, where the higher the fitness the more chance that individual is chosen for the new population. However, this does not guarantee that all fitter individuals are chosen. The fitness of all the individuals is splayed out on a roulette wheel. This selection type gives less fit individuals a chance at being chosen for the new population and keep diversity in it. The fittest individual ever recorded is also guaranteed a place within each new population created.
7. The algorithm will iterate until the maximum number of generations (user-definable) is reached, or convergence (no change in results for a number of generations) (user-definable) occurs, where it will then return the best solution/individual found, displaying the individual’s colouring, chromon count, and largest chromon found in the individual, as well as its fitness value and generation where the individual is from.

In mutation and crossover, a smoothing function was used. The smoothing function takes in a vertex and checks all its neighbouring vertices and if they are more co-chromatic neighbours than not, the function will flip the vertex colour, and will recursively run its function on the neighbouring vertices.

### NON GENETIC ALGORITHM

The non genetic algorithm was designed around the smoothing function mentioned above. A solution is created via smoothing. It will randomly decide its first colour on the first vertex, and then move on to the neighbouring vertices, where it decides what colour to choose based on its neighbouring vertices, until all vertices have been coloured. When this operation is done, a second smoothing is applied over the solution, starting from the first vertex, and iterated through every vertex in the graph. The solution is then evaluated and displayed.

### RANDOM GRAPH GENERATOR ALGORITHM

The algorithm implemented to generate random four-regular graphs of  $n$  vertices is based on the Watts and Strogatz model [2]. The user defines how many vertices the graph requires and while every vertex does not have four incident edges, a new graph is instantiated with  $n$  vertices and every possible vertex point is added to a list.

Two vertex points are randomly chosen from the list, and if they are suitable (not a loop, an existing edge, or create a graph larger than a degree of four), will be added into the graph. The generator will continuously add edges to the graph, until either all vertex points are added, or no more suitable points can be added.

At this stage, the generator will check whether the graph is a four-regular graph, and if not, will reset the graph and all vertex points, and start again. Otherwise, it will return a randomly generated four-regular graph. The newly created graph is saved into a text file known as 'test.txt' within the program folder; this gives the opportunity to test whether the graph was properly completed.

## PROJECT DOCUMENTATION

### USER INTERFACE/DOCUMENTATION

System Details	
CPU	Intel Core 2 Duo E8200 at 3.8Ghz
Motherboard	Gigabyte X38-DS4
RAM	2GB Crucial Ballistix Tracers 950Mhz
Operating System	Windows XP Service Pack 3
Compiler	Eclipse Ganymede version 3.4 (Java)

#### File Requirements:

2 files are needed in order to run the program:

The latest Java Development Kit, *Java SE Development Kit (JDK) 6 Update 10*, which is available for download at their main website:

<http://java.sun.com/javase/downloads/index.jsp>

Eclipse Ganymede, *Eclipse IDE for Java Developers (85 MB)*, available at this website:

<http://www.eclipse.org/downloads/>

Documentation on the latest JDK can be found here:

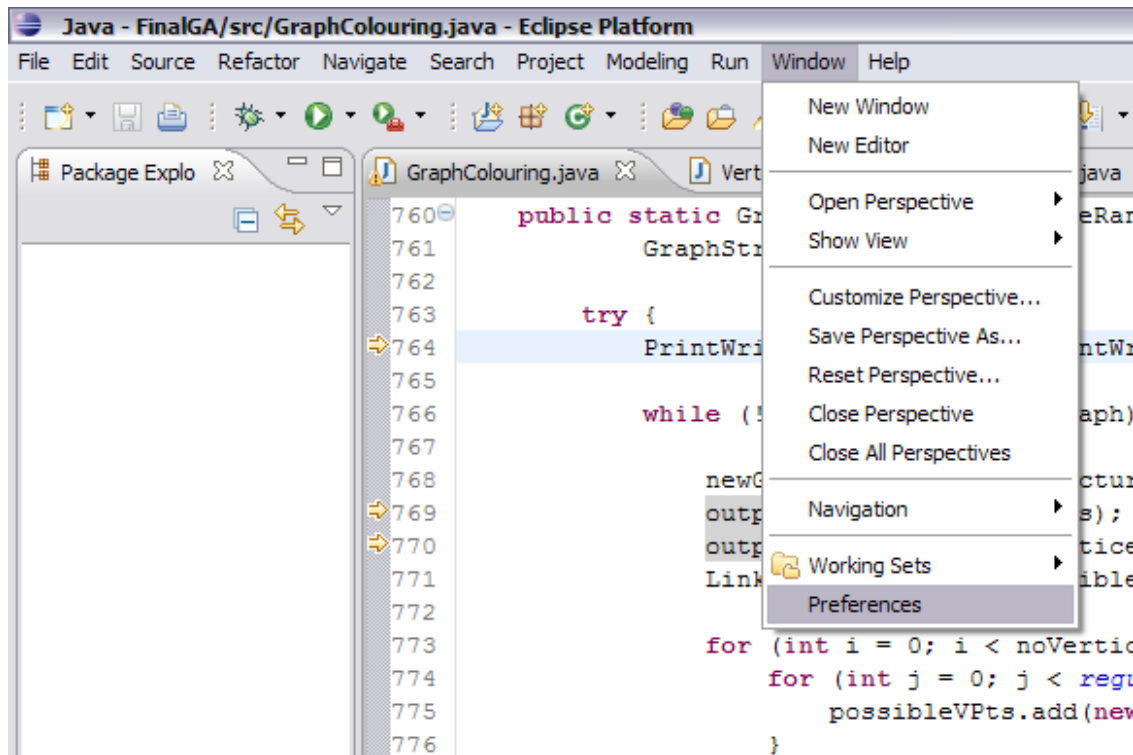
<http://java.sun.com/javase/6/webnotes/install/index.html>

Note: The operating system that the program was run in is under Windows XP. Use other operating system's version of the above at your own risk.

## Compiling and Running the program

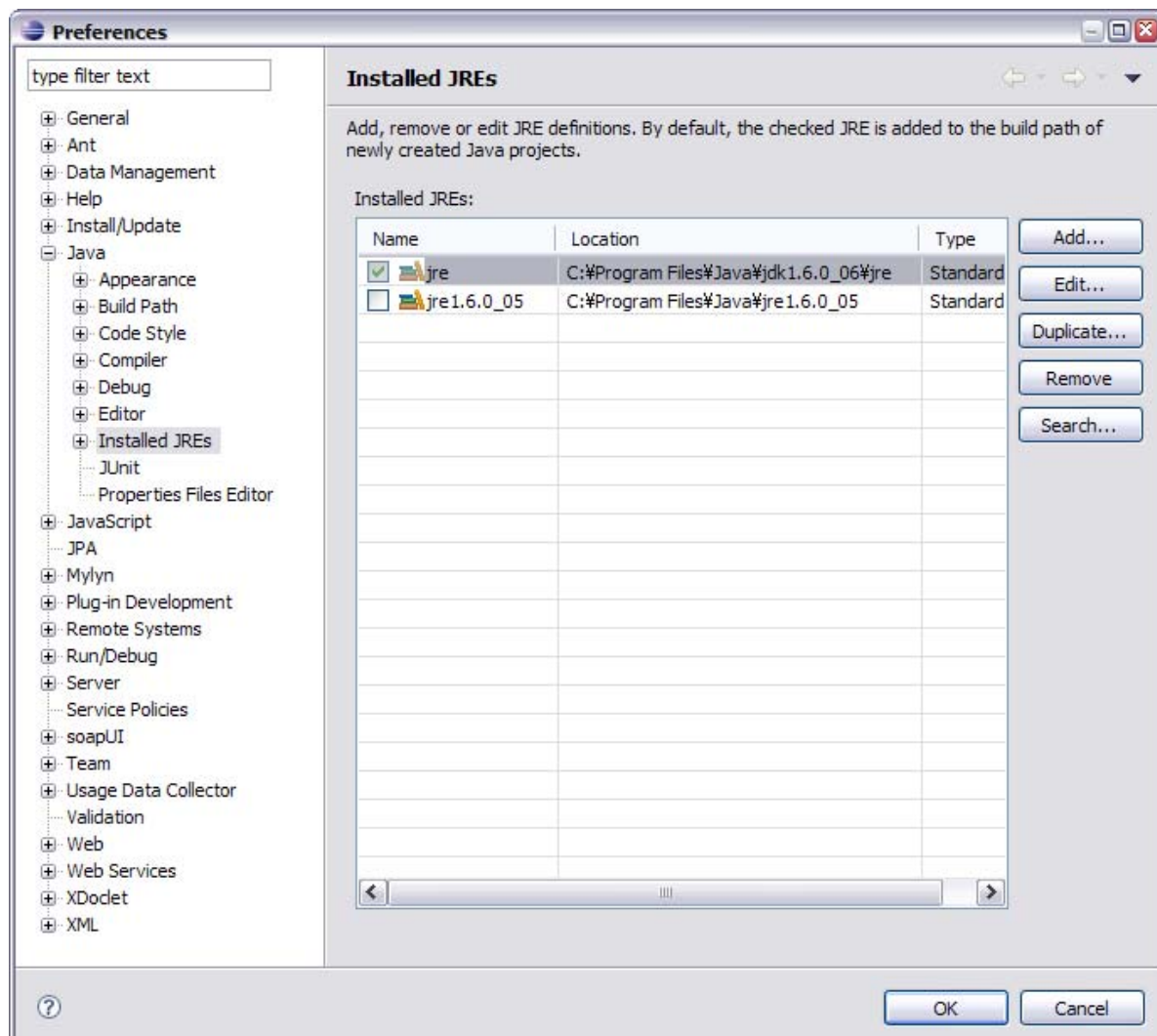
Simply run the exe file for the JDK to install it.

After Eclipse has been unzipped to its destination folder, it can be run by simply double clicking the eclipse.exe. In order for eclipse to use the recently downloaded JDK, you will need to go into the preference menu in Eclipse, shown in **Figure 1.1** below.



**Figure 1.1** – Accessing the preference menu

Refer to **Figure 1.2** for the following steps. Once in the preference menu, expand the *Java* menu option on the left side by clicking the + sign, and select *Installed JREs*. From the list shown in the main section of the preference menu, check the box for the latest JDK. If it is not on the list, use the search button to find the directory where the JDK is installed. Click OK when finished.



**Figure 1.2** – Selecting a JRE

Once the above is completed, create a new java project by right clicking and selecting *New* → *Java Project* in the *Package Explorer* to the left side of the main interface in Eclipse, as done in **Figure 1.3**. A new window will appear to create a new Java project, like the one in **Figure 1.4** below. Enter a name for the project name and select the *Create a project from existing source* option, and browse for the directory where the project is in. Once the directory is selected, click *Finish*.

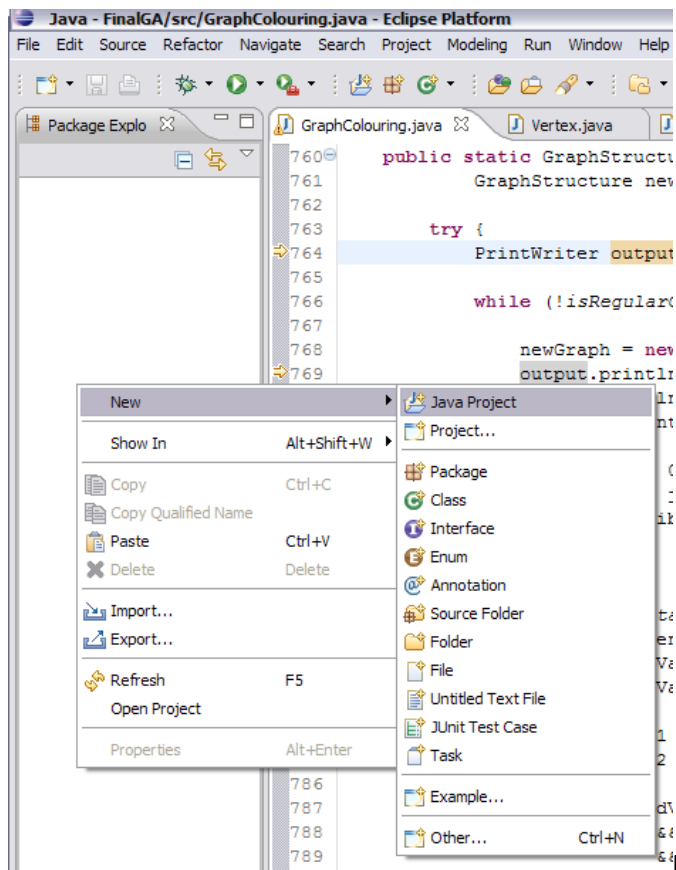


Figure 1.3 – Creating a new Project (left)

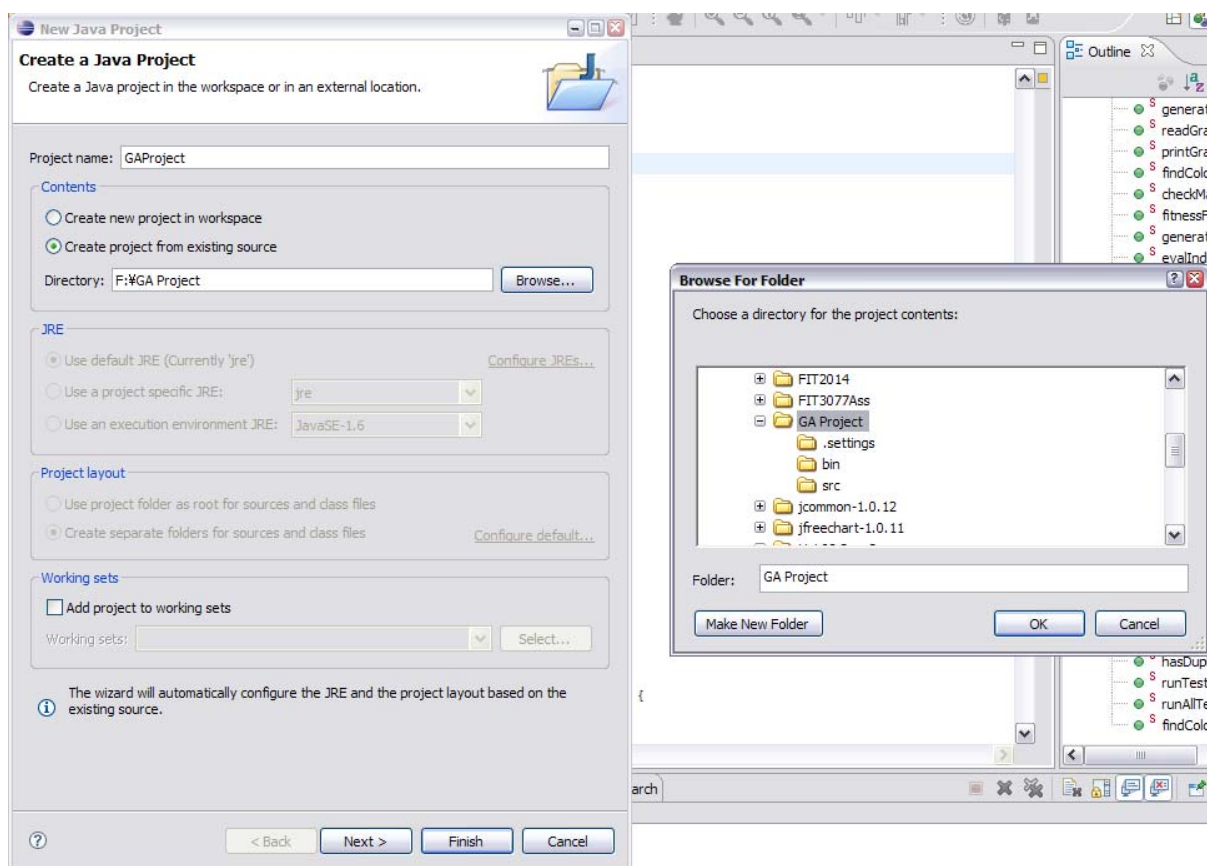


Figure 1.4 – Creating a project from an existing source



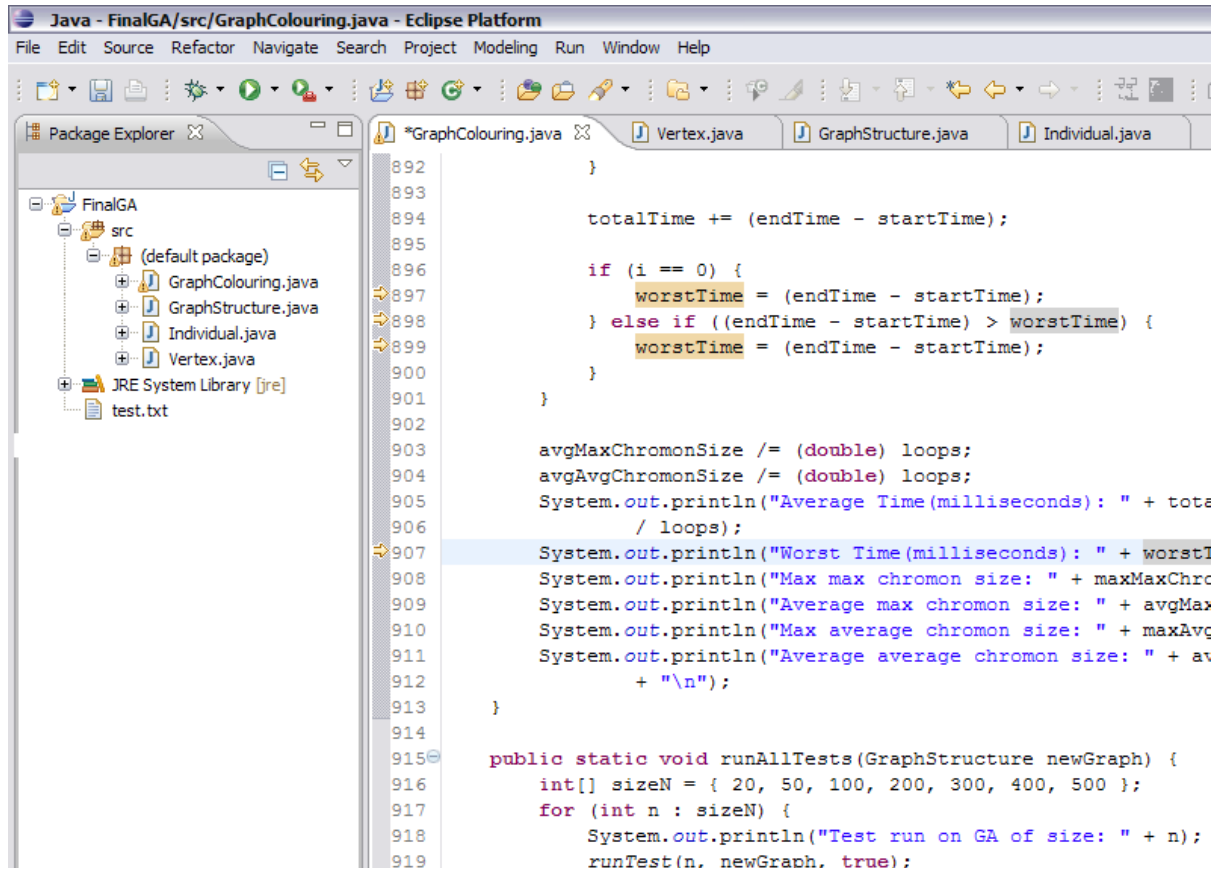


Figure 1.5 – Newly created Project

When your project has been created, it will be shown on the left in the Package Explorer, as shown in **Figure 1.5**. All the java classes can be viewed by double clicking them in the default package in the *src* folder within the project. To run the program, double click the *GraphColouring.java* class in the default package, and click the drop down menu just to the right of the green play button highlighted in yellow in **Figure 1.6**, and select *Run As* → *1 Java Application*.

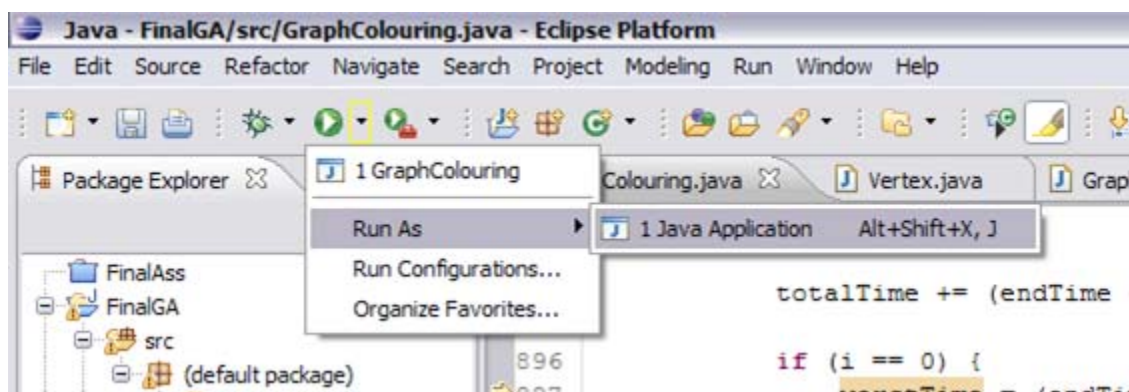
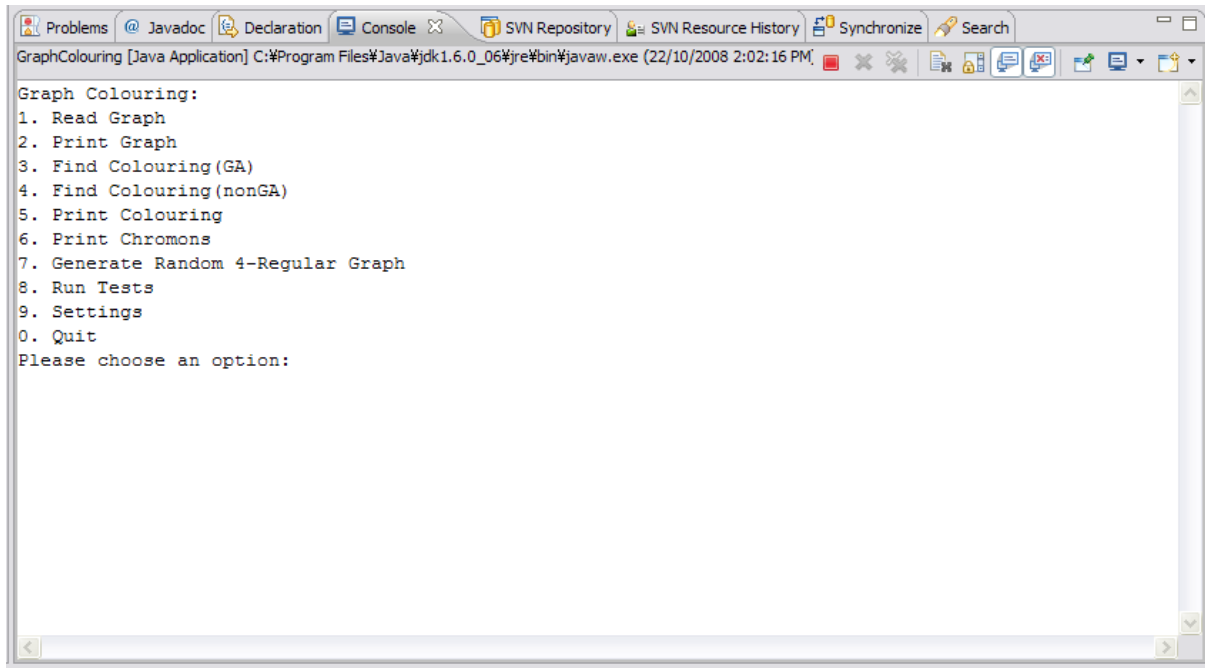


Figure 1.6 – Running the program

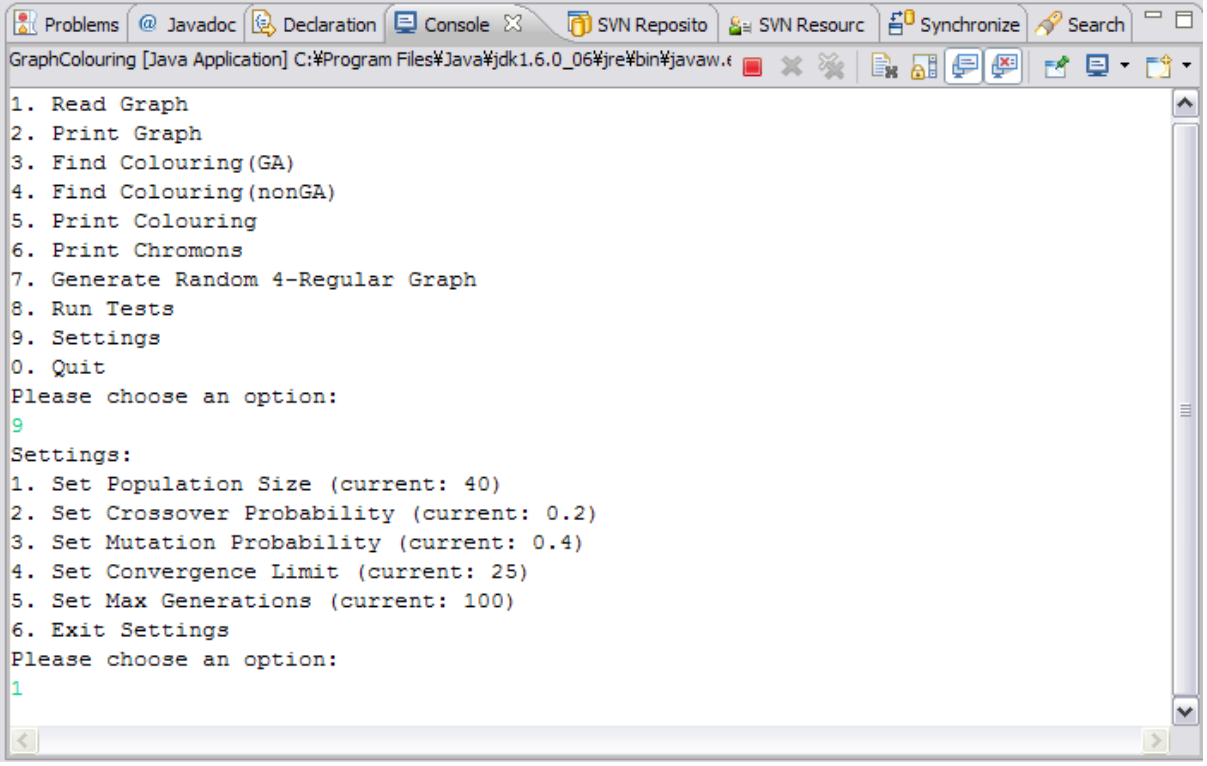


**Figure 1.7** – Program menu

The program will start and display in the console in Eclipse when it is run. To select an option, enter a numeral corresponding to each selection, from 0-9, and click Enter to execute that task. There are 10 options available as shown in **Figure 1.7** above, each performing their own function:

1. Read Graph – Selecting this option, you will be prompted to enter a filename which is in a graph readable format. This can be either the entire location of the file such as "C:\example.txt" or if the file is within the program directory just enter "example.txt". The graph will be saved within the program.
2. Print Graph – If there is a graph within the program, it will print all vertices and the adjacent vertices in each vertex.
3. Find Colouring (GA) – If a graph exists, the program will prompt if the user would like to run till max generations, entering 1 for yes or 2 for no. Once the user has selected an option, the program will run a genetic algorithm over the graph, and display a solution.
4. Find Colouring (non GA) – if a graph exists, a non genetic algorithm will run over the graph and display the results of the solution.
5. Print Colouring – If a graph colouring algorithm was run recently, this option will print the colouring of each vertex in the graph of the last result.
6. Print Chromons – Prints all the chromon sets from the last graph colouring solution.
7. Generate Random 4-Regular Graph – This option prompts the user of the size of the graph to be generated, and once the user has inputted a value greater than four, it will generate a new graph of a user-specified size of  $n$  vertices and store it within the program. It will also save the generated graph into 'test.txt'.

8. Run Tests – Runs tests on the genetic and non genetic algorithms over a number of different sized randomly generated graphs several times and displays the results and its run times for each graph and algorithm.
9. Settings – This option will open another menu where certain variables can be changed and each setting will display current values, as seen in **Figure 1.8**:
  1. Set Population Size – Changes the population size within the genetic algorithm if the value inputted is greater than zero.
  2. Set Crossover Probability – Changes the crossover probability as long as it is between zero and one.
  3. Set Mutation Probability – Changes the mutation rate as long as it is between zero and one.
  4. Set Convergence Limit – Changes the convergence limit in a genetic algorithm, if the value is greater than zero.
  5. Set Max Generations – Changes the generation in which the genetic algorithm will continue to run till, as long as convergence does not occur. Value inputted must be greater than zero.
  6. Exit Settings – leaves the settings menu and goes back to the main menu.
0. Quit – This will exit the program, and the last graph and solution stored in the program will be lost.



```
Problems @ Javadoc Declaration Console SVN Reposito SVN Resourc Synchronize Search
GraphColouring [Java Application] C:\Program Files\Java\jdk1.6.0_06\bin\javaw.exe
1. Read Graph
2. Print Graph
3. Find Colouring(GA)
4. Find Colouring(nonGA)
5. Print Colouring
6. Print Chromons
7. Generate Random 4-Regular Graph
8. Run Tests
9. Settings
0. Quit
Please choose an option:
9
Settings:
1. Set Population Size (current: 40)
2. Set Crossover Probability (current: 0.2)
3. Set Mutation Probability (current: 0.4)
4. Set Convergence Limit (current: 25)
5. Set Max Generations (current: 100)
6. Exit Settings
Please choose an option:
1
```

**Figure 1.8** – User-changeable settings

Error Messages	
Message	Reason
<i>Invalid option. Try again.</i>	User entered a non existing option
<i>Invalid option or incorrect input. Try again.</i>	User entered an incorrect value or a non existing option.
<i>Invalid graph: There are no Vertices.</i>	The file read has no vertices in the graph.
<i>Invalid graph: Loop exists.</i>	The graph being made has a looping edge.
<i>Invalid graph: Duplicate Edges.</i>	The graph being made has duplicate edges.
<i>Invalid graph: larger than a 4-Regular Graph.</i>	The graph has one or more vertices with more than four adjacent vertices.
<i>Incorrect format.</i>	The file read is not in a graph readable format.
<i>Invalid graph: inconsistent edge count.</i>	The file read has an incorrect edge count compared to the amount of actual edges that were added into the graph.
<i>File not found or could not be opened.</i>	The file name inputted by the user is incorrect or the file does not exist.
<i>No graph exists.</i>	User attempted to run an option that requires a graph, without a graph stored in the program.
<i>Cannot print as no colouring algorithm was performed.</i>	User selected an option where a colouring algorithm needed to be done beforehand.

## IMPLEMENTATION/TECHNICAL DOCUMENTATION

The data structure used to store the graph is an *adjacency list* structure. Each vertex within the graph has an adjacency list of vertices which it is connected to. This structure was used due to the sparseness of the graphs as it grew larger in size and because the graphs were only at most a degree of four, thus saving space and memory. Each vertex also stores a Boolean to check whether that vertex has been visited or not so that certain functions can run properly.

The possible solutions are stored in a data structure known as *Individual* for each solution, which contains all the attributes needed for the solution: colouring, chromon count, largest chromon size, fitness, and generation. The population needed for the genetic algorithm is an array of the *Individual* data structure.

### Functions:

The following are non-trivial functions within this program.

#### GraphStructure

FUNCTION NAME: clearVisits

DESCRIPTION: Clears all visited vertices

PARAMETERS: none

RETURN VALUE: none

**Individual**

FUNCTION NAME: toString

DESCRIPTION: Displays the individual's attributes

PARAMETERS: none

RETURN VALUE: String

**GraphColouring**

FUNCTION NAME: menu

DESCRIPTION: Displays the main menu for the program

PARAMETERS: myScanner: Scanner, newGraph: GraphStructure

RETURN VALUE: none

FUNCTION NAME: setSettings

DESCRIPTION: Displays user-changeable settings menu

PARAMETERS: myScanner: Scanner

RETURN VALUE: none

FUNCTION NAME: generateColour

DESCRIPTION: Generates a random colour. Black or White

PARAMETERS: none

RETURN VALUE: char

FUNCTION NAME: readGraph

DESCRIPTION: Reads a graph, checks for any errors and stores it into program

PARAMETERS: fileInput: String, newGraph: GraphStructure

RETURN VALUE: boolean

FUNCTION NAME: printGraph

DESCRIPTION: Prints the graph stored in the program

PARAMETERS: newGraph: GraphStructure

RETURN VALUE: none

FUNCTION NAME: findColouringGA

DESCRIPTION: Uses a Genetic algorithm on a graph to find the best "individual"

PARAMETERS: newGraph: GraphStructure, maxGenEnabled: boolean, print: boolean

RETURN VALUE: none

FUNCTION NAME: checkMaxGen

DESCRIPTION: Checks whether it will run till convergence and if not checks whether if it has reached max generations

PARAMETERS: enabled: boolean, isNotMaxGen: boolean

RETURN VALUE: boolean

FUNCTION NAME: fitnessFunction

DESCRIPTION: Fitness evaluation for individuals. A great fitness is above 5 (Perfect colouring is 10). formula:  $(\text{no. of chromons} + (1/\text{chromon size})) / N \times 10.0$

PARAMETERS: newGraph: GraphStructure, noChromons: double, chromonSize: double

RETURN VALUE: double

FUNCTION NAME: generateNewPop

DESCRIPTION: Generates a new population

PARAMETERS: newGraph: GraphStructure, gen: int

RETURN VALUE: Individual[]

FUNCTION NAME: evalIndividual

DESCRIPTION: Initial method of evaluating an individual in the population

PARAMETERS: newGraph: GraphStructure, individual Individual, print: boolean, set: boolean

RETURN VALUE: none

FUNCTION NAME: evalIndividual

DESCRIPTION: Recursive method of evaluating an individual

PARAMETERS: input: Vertex, newGraph: GraphStructure, chromonSize: int, individual: Individual, print: boolean

RETURN VALUE: int

FUNCTION NAME: evalBestIndividual

DESCRIPTION: Finds best individual in a population

PARAMETERS: none

RETURN VALUE: boolean

FUNCTION NAME: evalBestIndividual

DESCRIPTION: Compares an individual with the best and keeps the individual if it is better

PARAMETERS: individual: Individual

RETURN VALUE: boolean

FUNCTION NAME: mutation

DESCRIPTION: Initial method for mutation, checks whether mutation is needed

PARAMETERS: newGraph: GraphStructure

RETURN VALUE: none

FUNCTION NAME: mutate

DESCRIPTION: Method which mutates an individual

PARAMETERS: newGraph: GraphStructure, ind: Individual

RETURN VALUE: none

FUNCTION NAME: crossOver

DESCRIPTION: Initial crossover method which checks every individual for possible breeding

PARAMETERS: newGraph: GraphStructure

RETURN VALUE: none

FUNCTION NAME: crossOver

DESCRIPTION: Crossover method which creates a new child between 2 parents

PARAMETERS: newGraph: GraphStructure, parentA: Individual, parentB: Individual

RETURN VALUE: Individual

FUNCTION NAME: smooth

DESCRIPTION: Smoothing function which goes through the adjacent vertices

PARAMETERS: input: Vertex, newGraph: GraphStructure, ind: Individual

RETURN VALUE: none

FUNCTION NAME: smoothing

DESCRIPTION: smoothing function which checks the neighbours, before deciding to flip the vertex

PARAMETERS: inputVertex: Vertex, newGraph: GraphStructure, ind: Individual

RETURN VALUE: none

COMMENTS: Also part of the non genetic algorithm with modifications in it

FUNCTION NAME: flip

DESCRIPTION: Flip method for flipping a vertex colour

PARAMETERS: ind: Individual, vertInd: int

RETURN VALUE: none

FUNCTION NAME: flip

DESCRIPTION: flip method for the non genetic algorithm

PARAMETERS: ind: Individual, vertInd: int, colour: char

RETURN VALUE: none

FUNCTION NAME: printColouring

DESCRIPTION: Prints out the colouring of the best individual last stored

PARAMETERS: none

RETURN VALUE: none

FUNCTION NAME: printChromons

DESCRIPTION: Prints out all chromon sets of the last stored best individual

PARAMETERS: newGraph: GraphStructure

RETURN VALUE: none

FUNCTION NAME: clearGraph

DESCRIPTION: Clears the graph

PARAMETERS: newGraph: GraphStructure

RETURN VALUE: GraphStructure

FUNCTION NAME: generateRandomGraph

DESCRIPTION: generates a random 4-Regular graph

PARAMETERS: noVertices: int, newGraph: GraphStructure

RETURN VALUE: GraphStructure

FUNCTION NAME: isSuitable

DESCRIPTION: checks whether a generating graph is still suitable

PARAMETERS: possibleVPts: LinkedList<Integer>, newGraph: GraphStructure

RETURN VALUE: boolean

FUNCTION NAME: isRegularGraph

DESCRIPTION: checks whether all vertices in the graph has 4 adjacent vertices

PARAMETERS: newGraph: GraphStructure

RETURN VALUE: boolean

FUNCTION NAME: isLargerThanRegular

DESCRIPTION: checks whether adding an edge will create a graph with degree of more than 4

PARAMETERS: newGraph: GraphStructure, vert1: int, vert2: int

RETURN VALUE: boolean

FUNCTION NAME: hasDuplicateEdge

DESCRIPTION: checks if there is a duplicate edge before adding an edge into a graph

PARAMETERS: newGraph: GraphStructure, vert1: int, vert2: int

RETURN VALUE: boolean

FUNCTION NAME: runTest

DESCRIPTION: runs a test on an graph colouring algorithm on a graph of size n

PARAMETERS: n: int, newGraph: GraphStructure, isGA: boolean

RETURN VALUE: none



FUNCTION NAME: runAllTests

DESCRIPTION: runs all tests over different sized graphs and on both algorithms

PARAMETERS: newGraph: GraphStructure

RETURN VALUE: none

FUNCTION NAME: findColouringNonGA

DESCRIPTION: a non genetic algorithm for graph colouring

PARAMETERS: newGraph: GraphStructure

RETURN VALUE: none

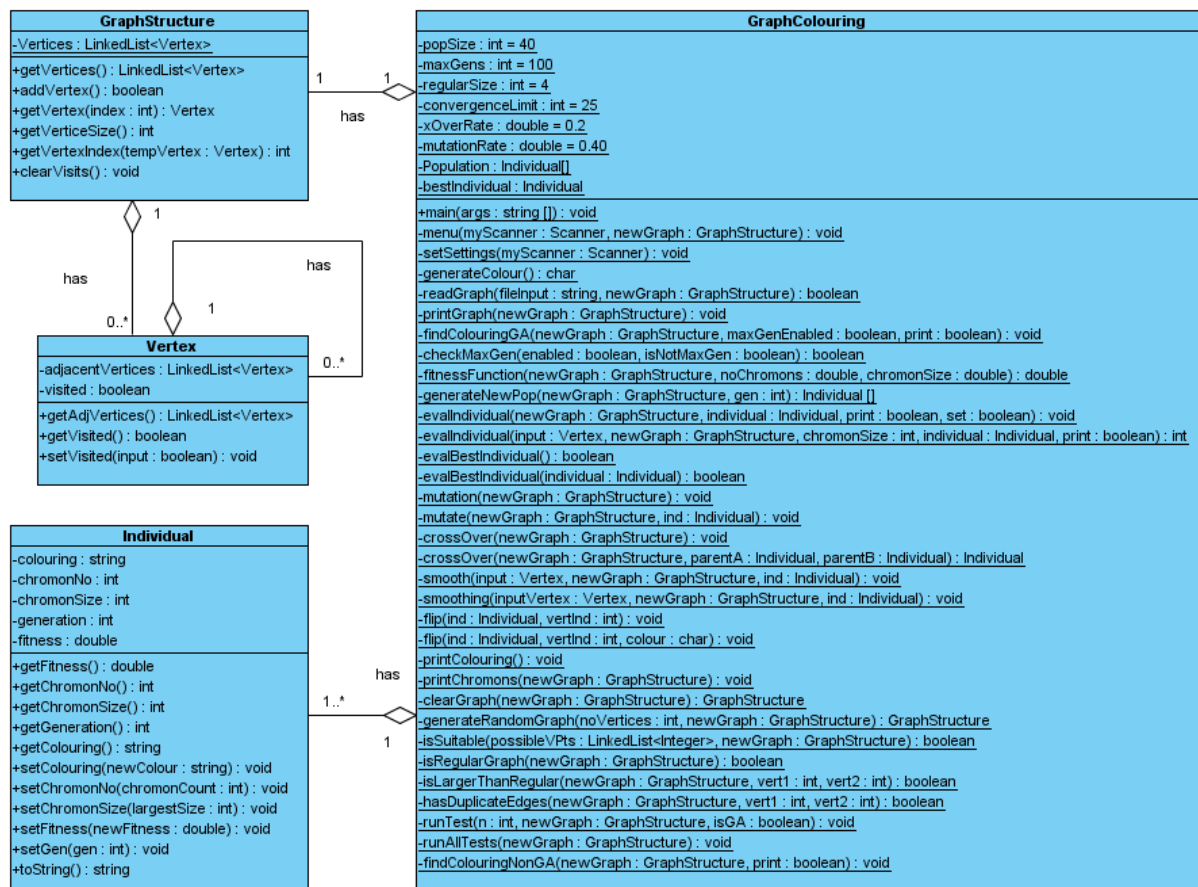


Figure 1.9 – Class Diagram

## EVIDENCE OF TESTING

When testing, the main factor that helps is what generation the solution has come from. Using the generation as a measurement and using the *runAllTests()* function, the rates of mutation and crossover were adjusted to produce a better solution, as well as adjusting population size and maximum generations, whilst minimising run time. Another factor is to start at higher rates and numbers, and slowly decrease the values, so as to be able to keep quality of solution whilst diminishing run time.

Having a larger mutation rate didn't improve current solutions, but it greatly effected run time. A good balance of run time and suitable results for mutation rate was found at around 35% (0.35).

As for crossover rates, at 70% it could triple the run time of the algorithm, though it produced better results. At 50% solutions were not degraded, and run time was reduced by half. At a suitable rate of 20%, quality of solution was stable, while reducing run time by half again. For population sizes, at 200 results were varying and only served to increase run time. Slowly decreasing the population size to 50 is when results were stable and run time was greatly reduced. Lower than 50 is when the quality of solution starts to diminish, due to lack of diversity in the population, and an early convergence.

Increasing the maximum generations will yield better results as long as there is sufficient size on the convergence limit. However the results will depend on the convergence limit. If it converges early the results may not be a good solution. The convergence limit also accommodates for smaller graphs, where convergence may occur in early generations as the algorithm has found a near-optimal solution. A maximum generation of 150 and a convergence limit of 25 were suitable for the algorithm's quality of solution.

When testing, it was noted that trying to decrease rates and size of the population to a certain point will keep the quality of solution whilst reducing its runtime. However, increasing the max generation, especially for larger graphs will produce better results, but also increase runtime.

## ACHIEVEMENTS

### METHOD

For each n-sized graph, the test will create a randomly generated graph and each algorithm will run on this graph ten times, before moving to the next n-sized graph. After each iteration, the results for that n size will be displayed on the console.

### RESULTS AND DISCUSSION

#### Settings for results:

Crossover probability: 20%

Mutation probability: 35%

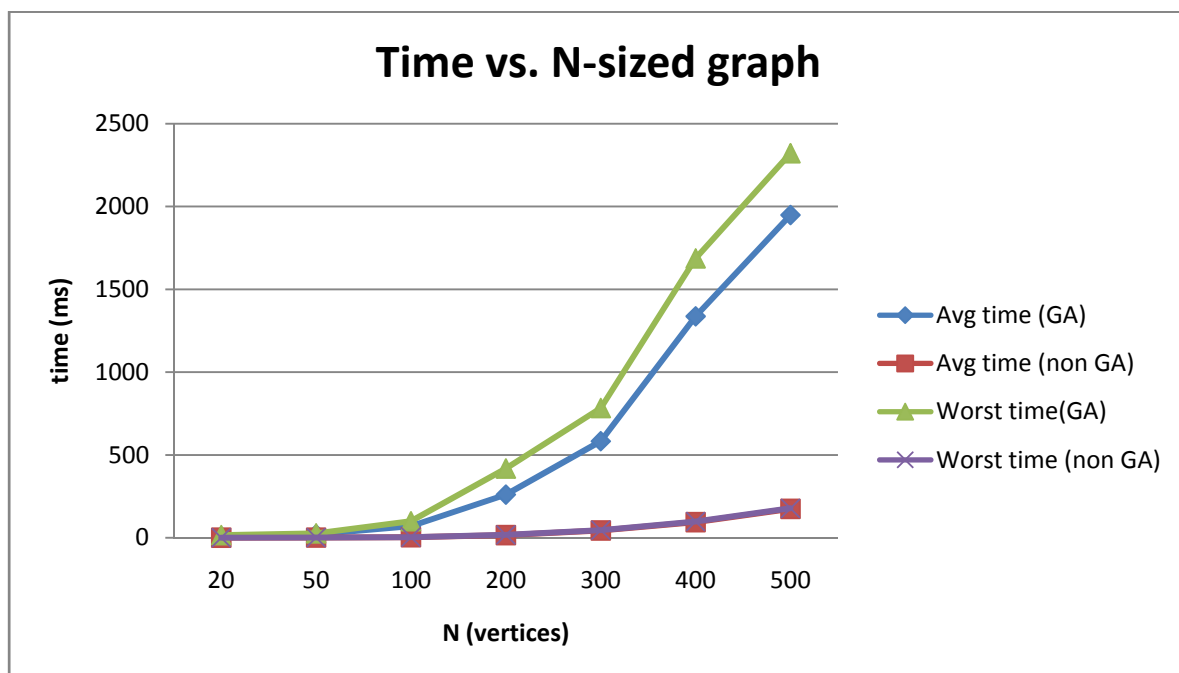
Population size: 50

Convergence limit: 25

Maximum generations: 150

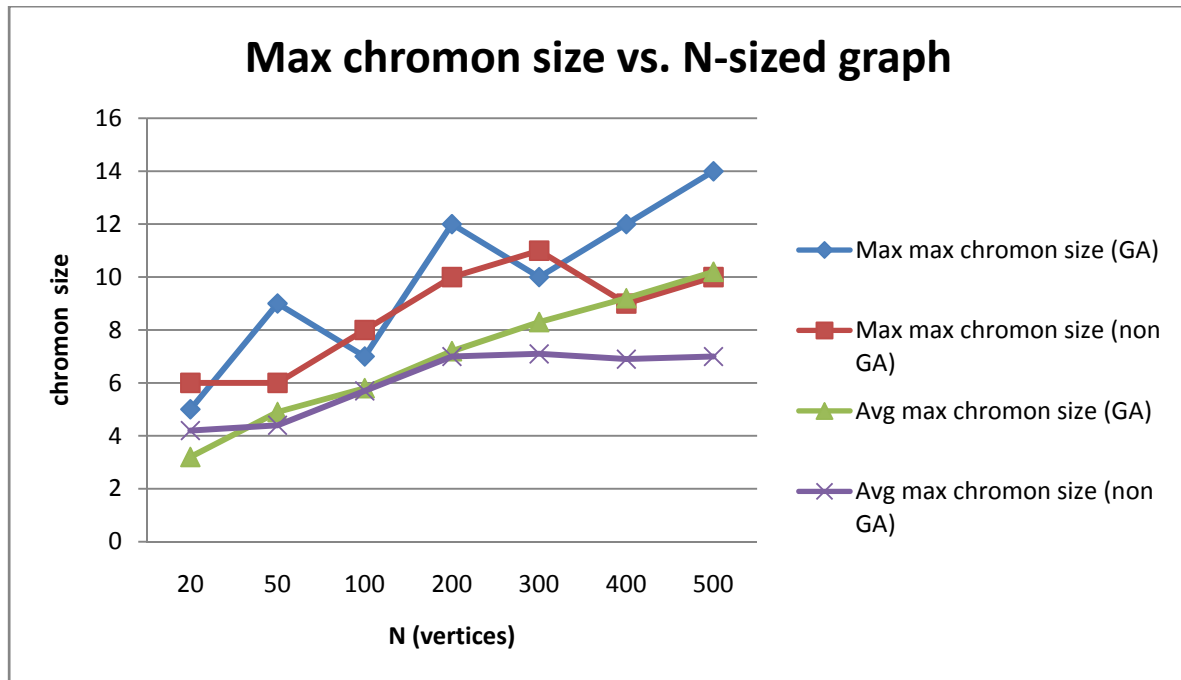
Results						
	Avg time (ms)	Worst time (ms)	Max max chromon size	Avg max chromon size	Max avg chromon size	Avg avg chromon size
<b>N = 20</b>						
GA	7.68	15.83	5	3.2	1.67	1.56
Non GA	0.15	0.23	6	4.2	2.22	1.81
<b>N = 50</b>						
GA	20.2	26.82	9	4.9	1.79	1.6
Non GA	0.74	0.91	6	4.4	1.92	1.68
<b>N = 100</b>						
GA	69.37	99.7	7	5.8	2	1.75
Non GA	3.21	3.48	8	5.7	1.79	1.66
<b>N = 200</b>						
GA	260.37	418.04	12	7.2	1.89	1.77
Non GA	15.47	17.44	10	7	1.82	1.7
<b>N = 300</b>						
GA	582.71	783.13	10	8.3	1.94	1.81
Non GA	43.63	45.87	11	7.1	1.76	1.67
<b>N = 400</b>						
GA	1337.11	1686.75	12	9.2	1.92	1.8
Non GA	93.76	97.84	9	6.9	1.79	1.7
<b>N = 500</b>						
GA	1948.93	2322.23	14	10.2	2.01	1.86
Non GA	173.99	177.6	10	7	1.75	1.67

**Figure 2.1** – Comparative Results of genetic and non genetic algorithm



**Figure 2.2** – Time vs. N-sized graph

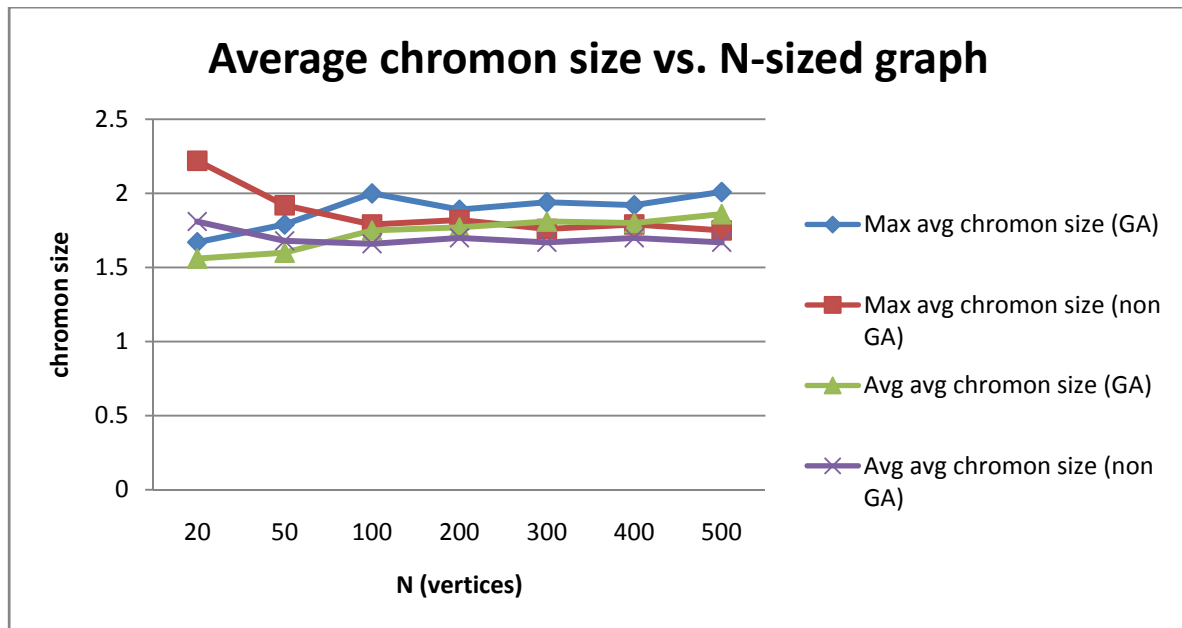
From **Figure 2.2**, it can be seen that there is a steady rapid increase in time as  $n$  gets larger for the genetic algorithm, whilst the non genetic algorithm barely increases in its runtime as  $n$  increases. This describes a lot about the efficiency of the non genetic algorithm, being more than 10 times faster than the genetic algorithm in most cases.



**Figure 2.3 – Max chromon size vs. N-sized graph**

Comparing both algorithms in terms of maximum chromon size, the genetic algorithm fluctuates a bit in terms of its maximum largest chromon size as  $n$  increases, but eventually stabilises into a steady increase. Analysing its average largest chromon size, it can be seen that there is a steady constant increase as  $n$  gets larger.

The non genetic algorithm produced steadier results for its maximum largest chromon size but dropped unusually when  $n$  was at 400 and 500. When looking at its average maximum chromon size, it maintains a constant chromon size when  $n$  was between 200 and 500. All these observations can be seen above in **Figure 2.3**.



**Figure 2.4** – Average chromon size vs. N-sized graph

In **Figure 2.4**, as  $n$  gets larger the genetic algorithm performed slightly worse, its average chromon size getting larger. Compared to the genetic algorithm, the non genetic algorithm had smaller average chromon size as  $n$  got larger.

## CONCLUSION

It can be deduced that the non genetic algorithm far outperforms that of the genetic algorithm in terms of quality of solution and runtime efficiency, where in most cases runs more than 10 times faster than the genetic algorithm. However, it can be noted that the genetic algorithm produced slightly better results with smaller graphs. Due to the genetic algorithm's complex nature, that of simulating an evolutionary system, there are too many variables to manipulate to produce quality solutions, without effecting runtime efficiency. For this task, there are probably more efficient algorithms available to use than a genetic algorithm.

To reduce the unpredictability of some functions within the genetic algorithm, mutation could be improved, so that it mutates the largest chromon found within the individual instead of randomly flipping a vertex. Also, if early convergence occurs, maybe increase the rate at which an individual is mutated for several generations to create diversity. Furthermore, investigation is needed on what order mutation, crossover and selection should be in, to produce better solutions.

Additional features that could be added at a later date is storing more than one graph, and being able to choose which graph to view and run an algorithm on. Also, all solutions can be

accumulated and stored to be viewed and compared later. Further improvements can be made by creating a GUI for the program, and have dynamically updated graphs to compare both algorithms, and a pie chart for the roulette wheel selection when generating new populations, to view the chances of individuals being selected.

## BIBLIOGRAPHY

1. K. Edwards and G. Farr. On monochromatic component size for improper colourings. *Discrete Applied Mathematics*, 148:89-105, 2005.
2. A. Steger and N.C. Wormald. Generating random regular graphs quickly. *Combin. Probab. Comput.*, 8:377-396, 1999.
3. AI Topics/ Genetic Algorithms  
<http://www.aaai.org/AITopics/pmwiki/pmwiki.php/AITopics/GeneticAlgorithms>
4. G.F. Luger. *Artificial Intelligence, Structures and Strategies for Complex Problem Solving*, Fourth Edition, pg 471, 2002.