An investigation into Chromatic Graph Theory and optimal Graph Colouring techniques

Developing and implementing a graph coloring tool and analysing it's performance on a range of generic and special-case graphs

Student Name: Harry Alexander Richards

Supervisor Name: Professor Daniel Paulusma

Submitted as part of the degree of BSc in Computer Science and Mathematics within the Natural Sciences programme to the

Board of Examiners in the School of Engineering and Computing Sciences, Durham University

Abstract —

Context/Background

To colour a graph is to label it's vertices such that no two adjacent vertices have the same label, where each label represents a colour. In the graph colouring problem one seeks to colour a graph using as few colours as possible, the minimum number of colours required to do this is called the chromatic number. The graph colouring problem has been a focal point in the field of graph theory for many years, it is an NP-hard problem and the interest that it has provoked has stimulated a number of proofs within graph theory over time.

Aims

The aim of this study is to develop methods of acquiring near-optimal upper bounds for the chromatic number of a given graph, and implement these methods in a graph colouring tool. It is also desired that as I test the tool on a variety of graphs from various graph classes I gain an understanding of properties that make an algorithm successful in colouring graphs, and ensure my algorithms are successful for these graph classes.

Method

A number of algorithms (and metaheuristics) will be developed, the most significant of these methods being a genetic algorithm, simulated annealing and a hybrid algorithm encompassing aspects of both of the aforementioned algorithms. There is a vast array of benchmark graphs available to use in the development of the algorithms. The colourings and chromatic number upper bounds yielded will be compared with the chromatic numbers for benchmark graphs, and this will be a key indicator of the quality of the algorithms.

Proposed Solution

Models of a vast array of graphs will be input into *Python* implementations in DIMACS standard format. From here the implementations can utilise the *NetworkX* packages in order to manipulate the given graphs and perform the algorithms. The implementation will also make use of the *graphviz* and *multiprocessing* packages in order to visualise the graphs and parallelise the algorithms.

Keywords —

Adjacent*, *Bipartite graph*, *Chromatic number*, *Cycle*, *Degree*, *Graph class*, *Planar graph

I INTRODUCTION

The aim of this project is to identify optimal techniques for solving the graph colouring problem both in the general case (for any given graph) and for specific graph classes of interest. When referring to graphs in this context, I refer to a mathematical structure that has found many uses within computer science. The task of obtaining graph colouring is the assignment of colours to elements of a graph subject to certain constraints. In the case of the graph colouring problem one aims to use the smallest number of colours possible to colour the vertices of the graph whilst satisfying the constraint that no two adjacent vertices are the same colour.

Graph Colouring Origins

The graph colouring problem originates from the colouring of maps, when Francis Guthrie noticed that no more than four colours ever seemed necessary to colour a map such that neighbouring regions were coloured differently *(Chartrand, G. and Zhang, P. 2009)*. If one models regions of the map as vertices and connects each pair of neighbouring regions with an edge then a planar graph is obtained. A graph is planar if it can be drawn in the plane so that its edges intersect only at their end-vertices *(Paulusma, D., Johnson, M., Golovach, P. and Song, J. 2016)*. The four colour problem asks if every planar graph can be coloured with four colours. Over time, the four colour problem became one of the most difficult problems in graph theory and stimulated many research areas within the field. The search for a proof for the four colour problem was considered a major driving force for graph theory for a long time, until finally the four colour theorem was proved. Historically, the proof of the four colour theorem was the first instance of a famous mathematical problem being solved by extensive use of computers. Whilst being controversial at the time this revelation has had a huge effect on modern day Mathematics and Computer Science.

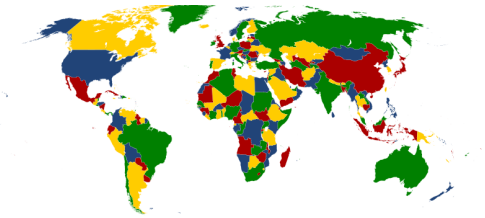

Figure 1: A depiction of the Four Colour Theorem applied to a simple map.

Graph Colouring Applications

Graph colouring currently has a number of real-world applications and continues to engage exciting research. Graph colouring is often applied to problems involving the optimisation of a resource whilst satisfying a constraint, there are many such problems and graph colouring is often utilised in obtaining solutions to them. In the real-world graph colouring is often used to solve problems where there are a limited amount of resources available, or other restrictions. The resource may be anything and the colours are simply an abstraction of the resource that is being optimised, whilst the graph is an abstraction of the problem as whole (where the graphs edges indicate a constraint). An intuitive example of this resides within compiler optimisation, this is the process of register allocation *(Lewis, R. 2016)*. Register allocation is the process of assigning a number of target program variables into a fewer number of CPU registers *(Wikipedia. 2019)*.

This problem can be modelled as a graph, where vertices represent target variables and an edge is constructed between variables that are used simultaneously. The resource (registers) is abstracted as colours, and a proper colouring ensures that no two variables used at the same time are allocated the same register. An optimal colouring would ensure that no more than the minimum number of registers required are used, freeing up registers for other tasks. There are a number of further applications including radio frequency assignment, scheduling and pattern matching - the majority of which follow the same vein of optimising the resources we have available to us whilst satisfying constraints.

Computational Complexity

We saw from the four colour theorem that the chromatic number of a planar graph will be at most four, similarly we can see that the chromatic number of a cycle (a graph with vertices $\{v_1, \ldots, v_n\}$ and edges $\{v_1v_2, v_2v_3, \ldots, v_{n-1}v_n, v_nv_1\}$ for $n \geq 3$) is at most 3 (depending on if the number of vertices) and the chromatic number of a bipartite graph is 2. Clearly the properties of a graph and the graph class it belongs to have a huge impact on whether the chromatic number of the graph can be determined, and how difficult it is to do so. Solving the graph colouring problem in general is rarely simple. In fact, it is an NP-complete problem to colour a graph with $n \geq 3$ vertices with 3 colours thus whilst a solution can be verified efficiently there is no known way of solving the problem in polynomial time. Knowing which algorithms will perform well on any NP-complete problem is one of the fundamental problems in computer science today. The task of obtaining the chromatic number (obtaining an optimal vertex-colouring) of a given graph belongs to the set of NP-hard problems. Thus, it is at least as hard as the NP-complete problems, despite all of the limitations imposed by various algorithms on the type of graphs used *(Alexandru, M. 2019)*. As a consequence of this, it is very difficult to determine which algorithms will perform well on the graph colouring problem as the problem size grows. Even the best know approximation algorithm computes a colouring of size at most within a factor $O(n(\log n)^{-3}(\log\log n)^2)$ of the chromatic number. For all $\epsilon > 0$ approximating the chromatic number within $n^{1-\epsilon}$ is NP-hard.

Project Purpose

The focal point of the project is to acquire optimal/near-optimal upper bounds for the chromatic number of given graphs using algorithmic methods. In order for the upper bounds to be of sufficient quality they must be relatively close to (within 10%) of the best known upper bound (which is supplied in the benchmark data). In order for this to be feasible much research has been done to find fast graph colouring algorithms, often by taking advantage of special properties exhibited by particular graph classes *(W. Shen, J. 2019)*, further to this I aim to reduce runtime of the algorithms through parallelising them.

The problem domain of the graph colouring tool is simply any graph, regardless of the graph's properties and the graph class it belongs to. Due to this the graph colouring tool will assume input graph is represented in DIMACS standard format, as this allows any valid graph to be represented. Due to the nature of the problem domain I intend to test the tool on graphs from a number of graph class. Both the DIMACS standard format and the graphs we will test the implementations on will be specified in the design section.

Often within chromatic graph theory algorithms do not perform as expected on graphs provided, thus we expect that throughout the course of the project results will demand that algorithms

produced are altered and fine-tuned in order to improve their performance.

Deliverables

Basic Objectives

- **B1** Design a tool in which one can input benchmark graphs in DIMACS standard format.
- **B2** Implement the first-fit/greedy algorithm and perform it on the benchmark graphs, in order to see how far the results it yields is from optimal solutions.
- **B3** Use heuristics to alter the order in which vertices are considered when performing the greedy algorithm and see how this impacts the colourings produced.
- **B4** Implement a number of other graph colouring heuristics, such as determining if it belongs to a particular graph class, again analyse the effectiveness of these heuristics and the causal factors in poor and near-optimal colourings.
- **B5** Implement a graph colouring checker, which verifies that colourings are valid (that no pair of adjacent vertices are coloured the same).

Intermediate

- **I1** Develop a genetic algorithm that will be performed on the benchmark graphs.
- **I2** Develop a simulated annealing algorithm that will be performed on the benchmark graphs.
- **I3** Alter the algorithms developed up to this point so to improve their performance on a number of special-case graph classes. Examples of these graph classes are graphs that are 'almost bipartite' or graphs with bounded maximum degree.
- **I4** Analyse the colourings yielded by the algorithms thus far and interpret how the algorithms exploit properties of particular graph classes, use this to further develop the algorithms.
- **I5** Utilise my genetic algorithm and simulated annealing in order to produce a hybrid algorithm that utilises aspects of both.

Advanced

- **A1** Parallelise the algorithms, the aim behind this is to improve the efficiency of my algorithms and reduce the required run time to obtain a colouring of sufficient quality.
- **A2** Visualise the colouring process of the algorithms, as this may be useful in seeing the characteristics of the colourings and further developing the algorithms.
- **A3** Abstract the problem of 'Sudoku solving' to a graph colouring problem, and analyse the performance of the algorithms (after some alterations) as 'Sudoku solvers'. This will involve the graphs starting with a fixed initial colouring and colouring the remaining vertices without altering the initial colouring.

II DESIGN

This section will continue the description of my proposed solution of the graph colouring problem in detail. The requirements for the implementations will be explicitly specified as well as the design plans for experimentation once the project has functionality. I will present specific system requirements and other technicalities for the project and how I intend to generate high quality colourings for given graphs. I will also discuss the input data and how I determine to assess the efficiency and general quality of the algorithms in comparison to other existing algorithms.

Requirements

Table 1: Functional Requirements

ID	Requirement	Priority
FR1	Be capable of inputting any given graph into the graph colouring tool.	High
FR2	We must be able to verify whether a colouring obtained is valid.	High
FR3	The greedy algorithm, genetic algorithm, simulated annealing and hybrid algorithm must be able to colour large graphs (with $4000+$ vertices) efficiently, regardless of how connected the graph is.	High
FR4	Store a representation of a graph, and perform a colouring algorithm on the representation, which should then translate to a proper colouring of the input graph.	High
FR5	Maintain an appropriate fitness (or temperature) variable which indicates the quality of the colouring and allows the genetic algorithm (or simulated annealing) to terminate once the colouring produced is of sufficient quality.	High
FR6	Ensure that we are able to generate sufficiently random initial colourings at the beginning phase of each algorithm. This may mean allowing invalid colourings up to some point in the algorithm.	High
FR7	Store the initial graph and a number of colourings in memory simultaneously.	High
FR8	Maintain efficient and thorough error-handling throughout manipulation of the input graphs and execution of the algorithms	High
FR9	Ensure that we are not stuck in a local minima by introducing an element of randomness.	Medium

Table 2: Non-Functional Requirements

ID	Requirement	Priority
NR1	Store various indicators of quality of performance for the different algorithms on the same set of graphs.	Medium
NR2	Parallelise the algorithm in order to reduce the run time.	Medium
NR3	Visualise the final graph colouring (for smaller graphs).	Low

NR4	Record the time it takes for each algorithm to reach a sufficient colouring on various graph classes.	Low
NR5	Visualise the graph colouring process as it occurs.	Low
NR6	Alter the algorithms such that they can successfully solve a valid Sudoku puzzle.	Low

System Architecture

Input Data

The input data for the graph colouring tool is a text file containing the graph structure for the input graph. We see an example of expected valid input data in Figure **??**. This representation of a graph is known as the DIMACS standard format and it is as follows: the first line of the file shall contain the total number of vertices and edges, each line proceeding this states the start-vertex and end-vertex of an edge in the graph structure. The DIMACS standard format is by far the most common method of representing graphs and all of the benchmark graphs sourced for this project will be found in this form. Thus, it is only logical that we make the assumption that any graph input into the colouring tool will be in the DIMACS standard format. In order to satisfy deliverable **B1** we will need to be able to parse the benchmark graphs from text files DIMACS format into the graph colouring tool.

```
p edge 11 20
e 1 2
e 1 4
e 1 7
e 1 9
e 2 3
e 2 6
```

Figure 2: An extract of an input graph in DIMACS standard format.

The benchmark graphs acquired represent graphs from 8 different graph classes, with the edges in each graph ranging from 12458 to over 4 *million*. The vast array of input graphs should give us a more general overview of the performance of the algorithms when drawing conclusions.

Representation

Whilst the DIMACS standard format is a convenient and easily readable representation, it is not suitable for use in the algorithms as it would increase the runtime dramatically. When it comes to the algorithms we need a representation of the graph that we can easily manipulate and iterate over, in order to ensure that an algorithm can be performed quickly. The obvious choices for this seemed to be either an adjacency list or an adjacency matrix due to their simplicity and the speed at which they can be iterated over. Alternatively, I considered the *NetworkX* package which allows me to represent the graph more quickly using existing built-in functions. *NetworkX* makes use of node and edge lists to represent graphs and is very flexible in terms of the data structures used in the implementations. I decided upon using *NetworkX* for its suitability for fulfilling requirements **FR1**, **FR3**, **FR4**, **NR3** and **NR5** as well as contributing to the fulfilment of basic deliverables **B1** and **B5** with some of its built-in functions (such as the ease of checking neighbours). *NetworkX* shall be discussed in further detail later in the design.

Algorithms and Heuristics

There are a lot of algorithms that take different approaches to graph colouring, and many heuristics such as considering different lower or upper bounds for the chromatic number that have had varying success. Throughout this project we aim to develop algorithms that are able to efficiently obtain solutions to the graph colouring problem. More than this, we hope that the colourings obtained by said algorithms provide near-optimal upper bounds for the chromatic number of any given graph. The algorithms that I will be experimenting with are a genetic algorithm, simulated annealing and a hybrid algorithm which will incorporate aspects of both algorithms. With aims to satisfy deliverables **B2**, **B3** and **B4**, I shall also develop a first-fit/greedy algorithm and various heuristics that will provide a clear indication of how effective the developed algorithms are compared to a basic technique. When referring to a first-fit algorithm we refer to an algorithm that goes through each vertex and colours it with the first available colour that is not the same as the colour of any adjacent vertex. Figure **??** exhibits colourings obtained by a greedy algorithm performed on a bipartite graph, to obtain the two colourings the order in which the vertices are coloured is altered. The contrast in results exemplifies the huge impact that heuristics (in this case vertex order) have on the quality of the colouring obtained by the algorithm - with one colouring clearly being optimal colouring and the other far from it.

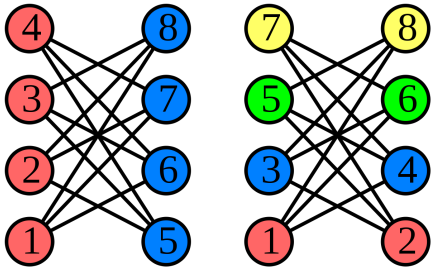

Figure 3: A comparison of two colourings acquired by a greedy algorithm performed on a bipartite graph.

The main algorithms that I have decided on for my project are the genetic algorithm, simulated annealing and the hybrid algorithm. The first of these, the genetic algorithm, is a metaheuristic inspired by the process of natural selection that belongs to the larger class of evolutionary algorithms. They are commonly used to generate high-quality solutions to optimisation problems by relying on biologically-inspired operators such as mutation, crossover and selection, *(Suryansh, S. 2018)*. The genetic algorithm initially obtains a population of colourings for the graph colouring problem. After considering the generation of this population I decided it would be best to colour each graph of n vertices at random using m colours (where $1 \leq m \leq n-1$ is an integer) and m alters for each individual colouring in the population. The issue with colouring the graphs using a method such as the greedy algorithm (with random vertex orderings) is that there would not be enough variance in the colourings in the initial population to optimise the colouring effectively, at least not for many of the graph classes we will be testing with. The genetic algorithm assigns 'fitness' values to each solution in the population, where a higher 'fitness' value indicates a better

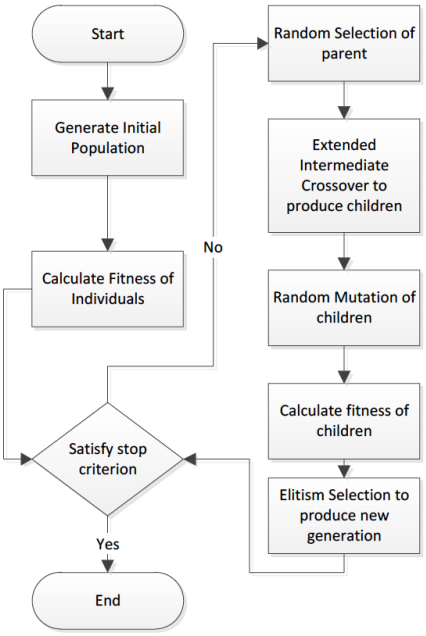

Figure 4: A data flow diagram for a genetic algorithm *(Chen, P. and Shahandashti, S. 2008)*.

quality colouring. In this case the value of 'fitness' of a colouring will be determined by the number of colours used and the number of conflicts in a colouring, where a conflict is a pair of adjacent vertices that are coloured the same. The fitness function will be something similar to $f = \frac{1}{number\ of\ colours\ used} + \frac{1}{(number\ of\ conflicts)^2}$ the intuition behind this is that we aim to minimise number of colours used whilst also minimising the number of conflicts in each colouring. However, experimentation and the results we obtain will dictate how we alter the fitness function. Each individual of the next generation of colourings is acquired through 'merging' a pair of colourings in the current generation. Selection of the pair of colourings used to generate the new colouring is done probabilistically where 'fitter' colourings are have a higher probability of being selected. The process of 'mating' or 'merging' two colourings is known as crossover and it is done by generating a random number, and ensuring that all of the vertices up to this vertex are coloured as they were in one of the parents and the remaining vertices are coloured as they were in the other parent. Through using this method it is hoped that the new colouring will exhibit characteristics of the colourings used to generate it. In order to ensure that the algorithm doesn't quickly reach a local minima there is a random probability that the colouring generated will be 'mutated', there are many methods of 'mutation' but it tends to be that the colourings of a number of vertices in the graph are swapped with the colourings of other vertices elsewhere in the graph. This process of generating populations is repeated until a member of a population is a sufficiently fit colouring, in other words a sufficiently low upper bound for the chromatic number.

The second of the approaches is the simulated annealing method, this again is a technique for solving optimisation problems. The method models the physical process of heating a material and then slowly lowering the temperature to decreases defects. The simulated annealing process begins by setting an the current temperature to an initial temperature value, to what is usually a very high value, in this case it will be set in relation the the number of vertices in the graph. For example, we may decide *current temp* = *number of vertices* $*$ 10 but this will be altered through experimentation and testing. In iteration of the simulated annealing algorithm/metaheuristic, a new colouring is randomly generated *(Akoglu, L. n.d.)*. If this random colouring is an improvement on the current colouring then we replace the current colouring with it, if not then we replace the current colour-

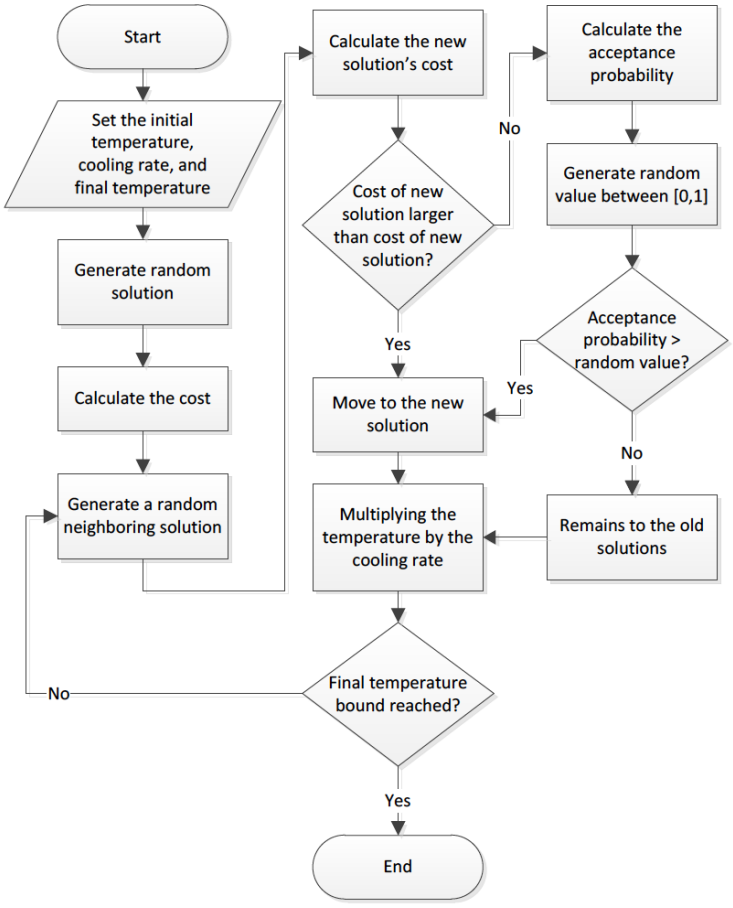

Figure 5: A data flow diagram for simulated annealing *(Chen, P. and Shahandashti, S. 2008)*.

ing with some probability (where the probability is related to the '*current temperature*'). The probability should be such that when the *current temperature* is high the probability is higher, and when the temperature is cooler (and we're closer to an optimal solution) the probability is low so there is less chance of 'swapping' for a worse colouring. The benefit of 'swapping' to worse solutions at different stages of the algorithm is again that we are able to avoid local minima. We reduce the temperature and repeat the process until the temperature has cooled sufficiently - at which point the current colouring should provide a high quality upper bound for the chromatic number of the graph. The rate at which we reduce the temperature is referred to as the cooling schedule, a typical cooling schedule is $T_k = \frac{initial\ temperature}{1+\log(1+k)}$ where k is the number of iterations we have performed thus far (including the current iteration).

The third main approach is the hybrid algorithm that I intend to develop, the hybrid algorithm in question will consist of methods from both the genetic and simulated annealing algorithms. As present in Figure **??** the hybrid algorithm will initiate by executing the genetic algorithm and later utilising simulated annealing. The rationale behind this is that the genetic algorithm is well suited to searching huge search spaces and relatively quickly obtaining an above quality colouring through making use of the populations it generates. Once we arrive at this stage we then begin to execute the simulated annealing approach and exploit local search areas *(Chen, P. and Shahandashti, S. 2008)*. So where the simulated annealing would usually take a random colouring as its initial start point, it's input is now the best colouring produced by the genetic algorithm (within a set amount of time). Due to this, the probability of 'swapping' from this colouring to a worse colouring would have to be dramatically reduced - in order to avoid undoing the work of the genetic algorithm. Each of the methods described above (Genetic Algorithm, Simulated Annealing and the Hybrid Algorithm) are key in **B4** and all of the intermediate objectives (**I1**-**5**).

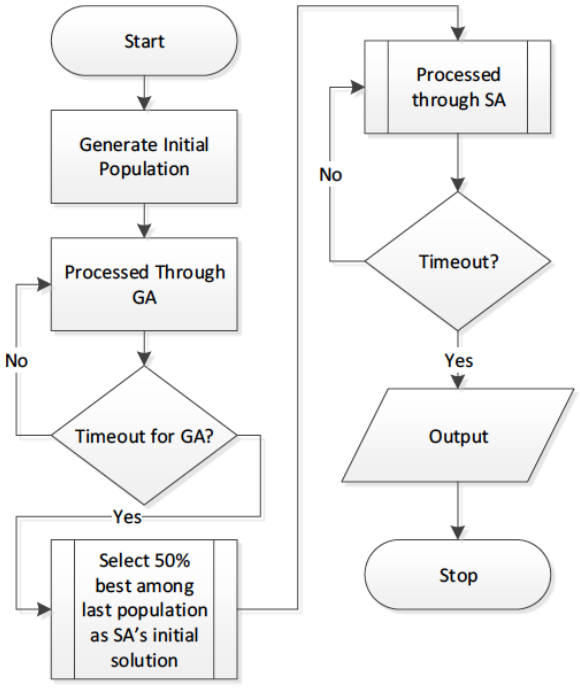

Figure 6: A data flow diagram for a hybrid algorithm involving a Genetic Algorithm and Simulated Annealing *(Chen, P. and Shahandashti, S. 2008)*.

Implementation

Python

Due to the nature of the program it makes sense that the program utilised follows OOP principles. Since Python is a multi-paradigm programming language we're able to choose the paradigm that

best suits the problem at hand, mix different paradigms in one program, and/or switch from one paradigm to another as the program evolves *(realpython.com. 2018)*. This flexibility alongside my familiarity with the programming language immediately made Python a strong contender. I also appreciated the fact that Python is very well documented and in my experience the syntax is relatively flexible. On top of this I was aware of a number of useful Python packages/utilities which were key in my decision to use Python as they would save a lot of time when performing some standard graph operations which there are built-in functions for.

NetworkX

As mentioned earlier *NetworkX* is a Python package for the creation, manipulation, and study of the structure, dynamics , and functions of complex networks. It provides appropriate structures in which we can store graphs, vertices, edges and even colours for each vertex. It also provides many of the standard graph algorithms which will save time on the developing front as there will be no need to write these functions myself. Some of the code for *NetworkX* is written in 'C, C++ and FORTRAN' meaning it can perform considerably faster than many would typically expect for a Python package, the speed at which the implementation operates will be key in the success of the project - particularly as the problem is scaled (number of vertices is increased) *(Developers, N. 2019)*. Whilst on the topic of scalability it is important to note that there are many examples of *NetworkX* being utilised in implementations involving tens of thousands of vertices and it's reliability when dealing with large graphs is necessary to satisfy **FR3**.

GraphViz and PyDot

GraphViz is open source graph visualisation software used to render graphs as image files *(PyPI. 2018)*. When we talk about graph visualisation here, we refer to a technique of representing structural information as diagrams of abstract graphs and networks. Alongside GraphViz I intend to use PyDot, which is an interface to GraphViz written in pure Python. PyDot can parse and dump into the DOT language used by GraphViz, allowing us to visualise the benchmark graphs somewhat simply. NetworkX is able to convert it's graphs to PyDot, this is highly convenient for the project and limit the of visualising the graphs. The software often seems to have trouble visualising graphs with large numbers of nodes (1000s), however as it becomes very difficult to analyse graphs of this size by looking at them anyway so graph visualisation of any kind would be redundant. *GraphViz* and *PyDot* are essential in satisfying **A2**.

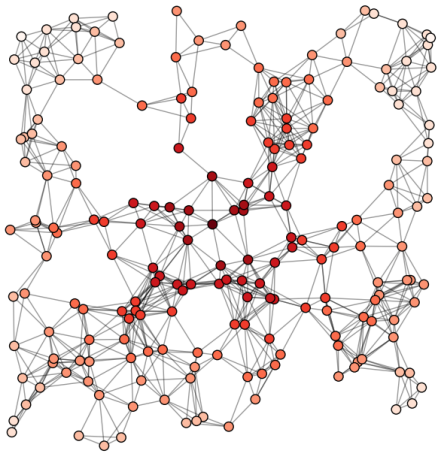

Figure 7: An example of a visualisation of a *NetworkX* graph.

multiprocessing and parallel programming

Python's 'multiprocessing' is a package that supports spawning process and offers both local and remote concurrency, this enables me to fulfil **A1**. It allows the programmer to fully leverage mult processors on a given machine providing it has a Unix or Windows operating system *(Raschka, S. 2014)*. Through parallel programming I am hopeful that we will be able to significantly reduce the runtime of the implementation/algorithms, particularly for benchmark graphs with a large number of vertices. Due to the overhead associated with spawning and managing threads the sequential implementation may even run more quickly than the parallel program for graphs involving a low number of vertices. It is my expectation that as the we scale the problem (involve more vertices) there will come a point when the runtime of the parallel program is far superior to that of the sequential program.

System functionality

When the graph colouring tool is first initiated an input graph must be provided and it is expected that the input graph is given in a text file, and the graph representation within that text file complies with the DIMACS standard format. From this we are able to use *NetworkX* formulate a representation of the graph that we're able to manipulate.

Once the input graph has been modelled it seems appropriate to visualise the graph - this will be useful in giving myself and others an idea of the structure of the graph being coloured. Following this the tool will proceed to perform one of the colouring algorithms on the graph - dependent on the algorithm selected. The result of the algorithms conclusion will be a proper colouring of the input graph, as well as outputting an upper bound for the chromatic number. At this point the colouring of the graph should be visualised and the chromatic number upper bound stored. At this point the colouring tool terminates.

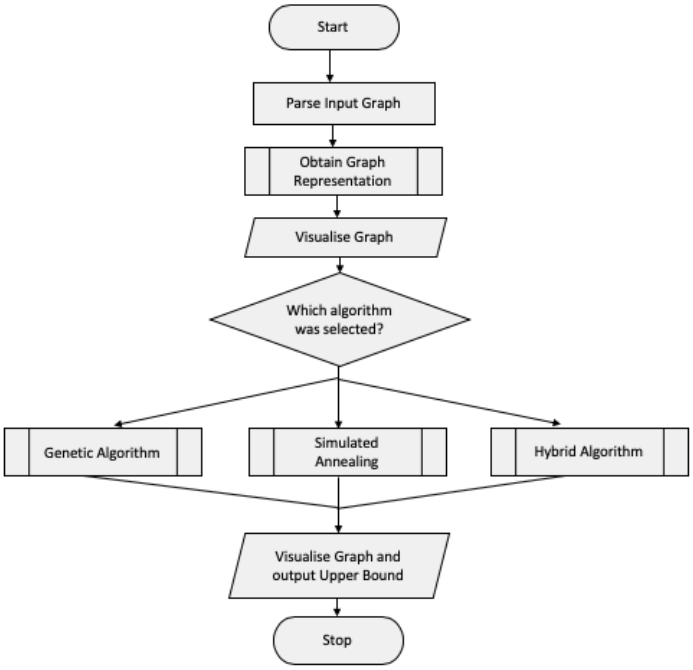

Figure 8: A basic illustration of the processes occurring in the graph colouring tool.

Analysing performance

When analysing the performance of the algorithms we will need to account for the stochastic nature of both the genetic algorithm, simulated annealing and consequently the hybrid algorithm. The fact that each method involves an element of random probability means that each of their performances will vary even when performed on the same benchmark graphs. Thus when obtaining results for the analysis of the algorithm I will calculate both the mean runtime and the mean

chromatic number upper bound for each graph from several runs of the algorithm. I will also calculate the variance/standard deviation of the upper bounds obtained, this way we can gauge how consistently the algorithm performs. The numerical results obtained by the algorithms will be compared to upper bound yielded by other algorithmic methods such as the greedy algorithm. We will also compare the results to the chromatic number for benchmark graphs and best known upper bounds for the chromatic number. Throughout the development of the algorithms we expect that performance is likely to vary,from the upper bounds obtained and the runtime of each algorithm I am likely to make changes to the implementation - however I will maintain the fundamental structure of each algorithm described in this report.

References

Chen, P. and Shahandashti, S. (2008). Hybrid of genetic algorithm and simulated annealing for multiple project scheduling with multiple resource constraints. [online] Available at: https://www.sciencedirect.com/science/article/pii/S0926580508001635?via%3Dihub#! [Accessed 19 Dec. 2018].

Suryansh, S. (2018). Genetic Algorithms + Neural Networks = Best of Both Worlds. [online] Towards Data Science. Available at: https://towardsdatascience.com/gas-and-nns-6a41f [Accessed 21 Dec. 2018].

Chartrand, G. and Zhang, P. (2009). Chromatic graph theory. Boca Raton: Chapman & Hall/CRC, pp.1-27.

Paulusma, D., Johnson, M., Golovach, P. and Song, J. (2016). A Survey on the Computational Complexity of Colouring Graphs with Forbidden Subgraphs. [online] Available at: 'url-https://arxiv.org/pdf/1407.1482.pdf [Accessed 17 Jan. 2019].

Lewis, R. (2016). A Guide to Graph Colouring.

Wikipedia. (2019). Register allocation. [online] Available at: https://en.wikipedia.org/wiki/Register_allocation [Accessed 18 Dec. 2018].

Alexandru, M. (2019). A software tool to demonstrate graph coloring. BSc (Hons) Computer Science. Manchester University.

W. Shen, J. (2019). Solving the Graph Coloring Problem using Genetic Programming. Stanford University.

Akoglu, L. (n.d.). What is Simulated Annealing?. [online] Cs.cmu.edu. Available at: http://www.cs.cmu.edu/afs/cs.cmu.edu/project/learn-43/lib/photoz/.g/web/glossary/anneal.html [Accessed 27 Dec. 2018].

realpython.com. (2018). Object-Oriented Programming (OOP) in Python 3 Real Python. [online] Available at: https://realpython.com/python3-object-oriented-programming [Accessed 27 Dec. 2018].

Developers, N. (2019). Overview NetworkX 1.10 documentation. [online] Networkx.github.io. Available at: https://networkx.github.io/documentation/networkx-1.10/overview.html [Accessed 24 Jan. 2019].

(PyPI. (2018). pydot. [online] Available at: https://pypi.org/project/pydot/ [Accessed 4 Dec. 2018].

Raschka, S. (2014). An introduction to parallel programming using Python's multiprocessing module. [online] Available at: https://sebastianraschka.com/Articles/2014_multiprocessing.html [Accessed 20 Jan. 2019].