An investigation into Chromatic Graph Theory and optimal Graph Colouring techniques

Student Name: Harry Alexander Richards

Supervisor Name: Professor Daniel Paulusma

Submitted as part of the degree of BSc in Computer Science and Mathematics to the

Board of Examiners in the Department of Computer Sciences, Durham University

Abstract —

Context/Background

To colour a graph is to label its vertices such that no two adjacent vertices have the same integer, where each integer 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 is a NP-hard problem that has been a focal point in the field of graph theory for many years.

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 desired that as we test the tool on graphs from various graph classes we determine properties of an algorithm that make them successful in colouring graphs.

Method

A number of algorithms (and metaheuristics) were developed, the most significant of these methods being a genetic-tabu algorithm, alongside a genetic-simulated annealing algorithm. Their performance was evaluated on a number of benchmark graphs and compared with existing methods.

Results

When performed on a variety of graphs the genetic-tabu algorithm produced upper bounds on the chromatic number which equal the current best known value in the majority of cases. In each case, the genetic-tabu algorithm produced colourings far superior to those produced by the genetic-simulated annealing hybrid and the *DSATUR* algorithm. The time taken to acquire proper colourings varied significantly across a number of graphs.

Conclusions

The results acquired by the genetic-tabu algorithm were far more consistent than the other methods investigated. Comparing the algorithms performance on a variety of graphs indicated that how well population diversity is maintained is critical in a population-based algorithm's performance. The times taken to acquire colourings for different graphs suggested the the underlying structure of a graph has a huge impact on the performance of the proposed algorithm.

Keywords — chromatic number, clique, diversity, genetic algorithm, NP-hard, search space, stochastic

I INTRODUCTION

The task of obtaining a graph colouring is the assignments of integers traditionally called '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 a graph whilst satisfying the constraint that no two adjacent vertices are the same colour.

Context and Background

The *graph colouring problem* originates from the colouring of maps, when Francis Guthries noticed that no more than four colours ever seem necessary to colour a map such that neighbouring regions were coloured differently. 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, using this method we can construct a planar graph from any map. Where a graph is considered planar if it can be drawn in the plane such that its edges intersect only at their end-vertices (Trudeau 1993). From this the four colour theorem was formulated, which states that any planar graph can be coloured using no more than four colours. Over time, solving the four colour theorem became one of the most difficult problems in graph theory and stimulated many research areas within the field. The search for a proof to the four colour 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 through extensive use of computers (Appel & Haken 1976). Whilst being considered controversial at the time, this revelation had a huge impact on modern day Mathematics and Computer Science and is considered common practice in modern-day science. Beginning with the origin of the Four Colour Problem in 1852, the field of graph colouring has developed into one of the most popular areas of graph theory (Chartrand & Zhang 2009).

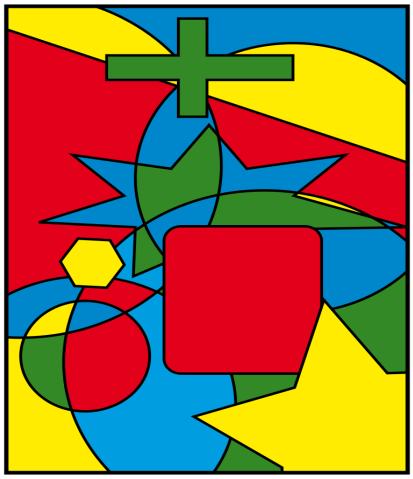

Figure 1: A map, to which the four colour theorem has been applied (Wikipedia 2007).

Graph colouring has proven useful in a host of real-world applications and continues to engage and motivate meaningful research. Graph colouring is often applied to optimisation problem where one must maximise a resource whilst satisfying a constraint, there are many such problems and graph colouring has proven useful in obtaining solutions to many of them. In these problems, we're often presented restrictions which impact the manner in which we can perform a task, perhaps there is a limited amount of resources available to us or we simply wish to be as efficient as possible with our available resources. In cases such as this graphs can often provide an abstraction of the problem, and provide us with a means of obtaining a solution. In this abstraction, vertices represent the subject matter in need of a resource. The resource may be anything, and the colours are simply an abstraction of the resource that is being optimised, whilst the presence of an edge between two vertices indicates a constraint (i.e. the same resource cannot be allocated to adjacent vertices). An intuitive example of this resides within compiler optimisation, this is the process of register allocation, the process of assigning a number of target

program variables into a fewer number of CPU registers. This problem can be modelled as a graph, where each vertex represents a target variable and an edge is constructed between any pair of variables (vertices) that are used simultaneously (Chaitin 1982). A proper colouring of such a graph ensures that no two variables used at the same time are allocated the same register, and an optimal colouring ensures provides an assignment of variables to registers such that the minimum number of required registers are used. As a result of employing graph colouring in this way we can be vastly more efficient with the resources we have available to us, and as such graph colouring is utilised in many fields where optimisation and constraint satisfaction problems are prevalent. Graph colouring also features in a number of further applications such as radio frequency assignment, scheduling and pattern matching and its versatility for a number of uses have helped fuel interest in it as a research topic.

Motivation for new graph colouring techniques

The *graph colouring problem* is a NP-hard problem and determines the minimum number of colours required to colour a given graph, such that no two adjacent vertices have the same colour. Let $G = (V, E)$ be an undirected graph, where $V = \{v_1, v_2, \ldots, v_n\}$ is the set of vertices in G, and $E \subseteq V \times V$ is the set of edges in G. A k-colouring of G is an independent partition of the set of vertices V into k colour classes: $\{V_1, V_2, \ldots, V_k\}$, such that for any pair of vertices $v_x, v_y \in V$ with $\{v_x, v_y\} \in E$ then if $v_x \in V_z$ we must have $v_y \notin V_z$ - otherwise the k-colouring is invalid. The need for a new method to solve the *graph colouring problem* over existing approaches is justified by the complexity of the problem. If we're tasked with colouring the graph G with k colours, there will be k choices of colour for each of the n vertices, and as a result a solution space contains a total of k^n candidate solutions a function that is subject to combinatorial explosion for any $k > 1$, and as a consequence in most cases the number of candidate solutions to be checked will quickly become too large for even the most powerful computer to tackle (Marappan & Sethumadhavan 2017). If we ensure that the search space only contains distinct independent partitions of V, then the number of ways of partitioning n *vertices* independently into exactly k non-empty subsets (*colour classes*) can be calculated as $\left\{ {n \atop k} \right\} = \frac{1}{k!} \sum_{i=0}^{k} (-1)^{k-i} \binom{n}{k} i^n$ (Lewis 2015). Even though this significantly reduces the size of our search space, the number of potential solutions is still astronomically large for most values of k. Determining which algorithms or metaheuristics will perform well on a NP-hard problem is one of the fundamental issues in contemporary computer science, and determining which algorithms will provide sufficient solutions to the *graph colouring problem* becomes increasingly difficult as the problem size grows. Consequently, designing an algorithm to obtain optimal colourings is a complex process. Even the current best known approximation algorithm for the chromatic number computes a colouring of size at most within a factor $O(n(\log n)^{-3}(\log \log n)^2)$ of the chromatic number (Halldórsson 1993), and for all $\epsilon > 0$ approximating the chromatic number within $n^{1-\epsilon}$ is NP-hard (Zuckerman 2007).

Aims and Objectives

The focal of this project is to acquire optimal/near-optimal upper bounds for the chromatic number of given graphs, to fulfil this requirement we shall investigate a number of approaches for combinatorial optimisation. In order to remove complexities from obtaining our solution and make our project objective less intimidating, a number of deliverables were defined.

These deliverables ranged from basic objectives which it was essential the project fulfil to more difficult advanced objectives which enhanced the quality of our solution but were not necessary in order to fulfil our project's purpose. Each of the deliverables are stated below.

Basic Objectives

- **B1** Design a tool in which one can input benchmark graphs in DIMACS standard format.
- **B2** Implement a simple greedy algorithm, such as first-fit.
- **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 more advanced graph colouring heuristics that are more suited to specific graph classes.
- **B5** Implement a graph colouring checker, which verifies that colourings are valid.

Intermediate Objectives

- **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.
- **I4** Analyse the colourings yielded by the algorithms thus far and interpret how the algorithms exploit properties of particular graph classes.
- **I5** Produce a hybrid algorithm that utilises aspects of both the genetic algorithm and simulated annealing.

Advanced Objectives

- **A1** Visualise the colouring process of the algorithms.
- **A2** Handle large graphs with $100s - 1000s$ of vertices, and very dense graphs.
- **A3** Produce a hybrid algorithm of genetic algorithm and tabu search.

Each of the deliverables contributes is designed to aid us in developing a more robust method of solving the *graph colouring problem* which achieves such k-colourings on a more consistent basis and obtains near-optimal colourings for any given graph, regardless of the graph class it belongs to, advancing on the long-standing existing methods (Galinier & Hao 1999) and (Resende & Sousa 2004). As we saw earlier, our there has been substantial effort made towards limit the solution space in order to colour a given graph, and in general this has been of little avail. Thus we aimed to design algorithms and heuristics which considered the graph colouring problem logically and efficiently in order to attain optimal or near-optimal colourings. In terms of quantitative requirements for these colourings, we aimed for the developed hybrid algorithms parallel current best efforts to colour the *DIMACS* challenge graphs, i.e. we aimed to acquire upper bounds on the chromatic number $\chi(G)$ which equal or improve upon the best-known current upper bounds for the given graphs.

II RELATED WORK

The *graph colouring problem* is an age-old problem, originating hundreds of years ago, and during its existence a number of methods have been proposed in solving it. However, despite the tremendous effort put into finding solutions to the *graph colouring problem* the solutions presented are still, for the most part, unable to produce colourings of sufficient quality. Some methods have experienced a degree of success but often fall short in terms of their robustness, due to their performance fluctuating greatly dependent on the properties of the graph that the method is applied to. Our aim is to produce a method that produces near-optimal colourings for all graphs, independent of the graph class the graph belongs to. Below we will discuss a variety of graph colouring methods of varying complexity, and the extent of their success.

Integer Linear Programming

Many integer linear programming models have been applied to the *graph colouring problem*. A typical structure for such a model consists of an objective function minimising the number of colours used, and a number of constraints imposing that each vertex is assigned a colour and that no pair of adjacent vertices are given the same colour (Malaguti & Toth 2010). Such models have been used extensively, to varying degrees of success. The model is often altered by applying further constraints, with aims of reducing symmetries in the solution space and consequently prevent the program considering redundant colourings. There a number of constraints that can be applied in order to make models more efficient, but ultimately all of the existing models suffer with the same limitation that the search space is still massive.

Greedy Methods

The principle of a greedy method is to colour the vertices of a graph successively, in a some order $v_{i_1}, \ldots, v_{i_n}$, according to some criteria. At each iteration of a greedy algorithm, we consider the next vertex in the specified order and a colour is assigned to the vertex. In general, we assign to v the smallest available colour that is not used by any of v's neighbours, and if necessary we assign to v an entirely new colour (Mitchem 1973). The quality of the resulting colouring is entirely dependent on the chosen ordering. For instance, Figure 2 exhibits two colourings of the same graph obtained by two greedy algorithms, and the order in which each vertex is considered during the greedy algorithms execution. The sole difference between these two algorithms is the order in which the vertices are considered. Clearly, whilst the algorithms used to obtain these colouring work from the same principles one algorithm acquires an optimal colouring in this case, whilst the other does not. The efficacy of the two algorithms is a consequence of the order in which vertices are considered, and the use of heuristics in order to influence the order in which vertices are considered has a huge impact on the quality of the colourings obtained.

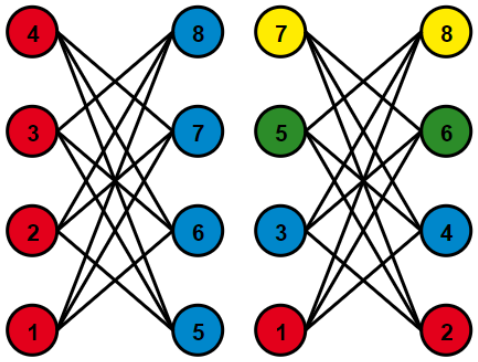

Figure 2: Two colourings of the same graph obtained by greedy algorithms. Created using draw.io.

Despite the common conception that greedy algorithms yield poor colourings when applied to the *graph colouring problem*, when implemented with effective heuristics there are several documented cases of them performing well. Due to their relative simplicity they're often able to acquire accurate approximations for the chromatic number $\chi(G)$ in little time (Brélaz 1979) and (Leighton 1979). However, the consensus that no existing greedy algorithm are contenders for solving the *graph colouring problem* in the general case is well founded. As we investigate dense graphs with many vertices, greedy algorithms begin to yield k-colourings which are far from optimal. Consequently, we utilise a notorious greedy algorithm *DSATUR* (Brélaz 1979) (described in the solution), to obtain an initial estimate for the chromatic number k_0. However, we require complex operators to improve estimates for the chromatic number $\chi(G)$ in general.

Successively building colour classes

There are many different approaches to solving the *graph colouring problem*. To find a k-colouring of a graph $G = (V, E)$ some consider a mapping $\phi : V \mapsto \{1, \ldots, k\}$ where if $\{v_i, v_j\} \in E$ we must have $\phi(v_i) \neq \phi(v_j)$. However, the colours themselves have no meaning and it is more intuitive to consider it as a partition of V into k subsets (colour classes) as we do. This approach simplifies the problem and allows us to consider each of the k subsets at a time, rather than considering each of the vertices and its corresponding colour. One approach to solve the colouring problem that shares this view is to build successively the different colour classes by identifying each time a maximal independent set and removing its vertices from the graph, and so solving the *graph colouring problem* is reduced to solving the *maximal independent set problem* (Galinier & Hao 1999). This method has proven to be one of the most effective approaches for colouring large random graphs. There are numerous different techniques for finding maximal independent sets (Tarjan & Trojanowski 1976) including the aforementioned 'greedy algorithms' and 'local search algorithms', which we will discuss in more detail below.

Local Search

A number of local search algorithms have been proposed in an attempt at solving the *graph colouring problem*, and the best of them have proven to be very successful (Marco Chiarandini & Stützle 2003). Two algorithms which have proven very successful are based on simulated annealing or tabu search (Lewis 2015), and it is these algorithms that we have investigated specifically during the course of this project. Both simulated annealing and tabu search work by starting with an initial potential solution (k-colouring), and at each iteration considering neighbouring solutions and 'moving' from solution to neighbouring solution if particular criteria is met. The aim behind a local search algorithm is to move from solution to solution in the space of candidate solutions (the search space), by applying local changes, until an acceptable (optimal) solution is found - at which point we hope to have found a valid k-colouring. The methods in which our simulated annealing and tabu search differ is the cost and neighbourhood functions. The neighbourhood function determines the local changes at each iteration and hence determines the neighbouring colouring that is considered. Whereas the cost function helps the local search to escape local minima. Both tabu search and simulated annealing have been analysed extensively, and a number of different neighbourhood and cost functions have been compared (Marco Chiarandini & Stützle 2003). One effective local search algorithm was developed by Morgenstern (Johnson & Trick 1993), offering a stark contrast to many of the existing methods.

Morgenstern's method aims to obtain a proper k-colouring of a given graph $G = (V, E)$, in order to do so it considers the set of legal partial colourings of the graph G. Each partial k-colouring is a partition of the set of vertices V into $k + 1$ independent subsets of V, $V_i \subseteq V$. For each $V_i \subseteq V$ with $i < k$ we have for all if $v_i, v_j \in V_i$ then $\{v_i, v_j\} \notin E$, i.e. if a pair of vertices are adjacent they do not belong to the same subset. The exception to this is V_k, since in general we cannot assign all vertices to subsets V_i without assigning adjacent vertices we assign any unassigned vertices to the same subset V_k regardless of whether they are adjacent. The algorithm aims to minimise the number of vertices in V_k, and as a result eventually obtain a proper k-colouring. The algorithm works well, and we utilise similar techniques in our solution.

Genetic Algorithms

The traditional genetic algorithm is a metaheuristic inspired by the process of natural selection, and belongs to the class of evolutionary algorithms (Vose 1999). Genetic algorithms are commonly used for optimisation and search problems and naturally there has been significant research and experimentation around their suitability to solving the *graph colouring problem*. Traditionally, the genetic algorithm works by initialising a population of a number of potential solutions, in our case k-colourings - where the population size is dependent on the implementation. The algorithm then iteratively evaluates the quality of population using a 'fitness operator', and uses a crossover operator to generate new solutions through combining pairs of 'fitter' solution. Through using these methods we progressively acquire higher quality solutions and eventually we hope that we acquire a sufficient solution. With some probability each new solution is mutated in some way in order to prevent the algorithm from being 'stuck' in local optima. Despite being common place in other NP-hard problems and optimisation problems pure genetic algorithms seems to yield relatively poor results to the *graph colouring problem* (Davis 2003), and implementations of pure genetic algorithms are few and far between. The genetic algorithm is reliant on an efficient crossover operator in order to transfer properties of quality parent colourings to their offspring, however these crossover operators seems ill-suited to the *graph colouring problem* and resultant algorithms perform slowly and produce insufficient colourings (Davis 2003). In our solution we consider an alternate crossover operator as proposed by (Galinier & Hao 1999), which ensures that an offspring colouring inherits its parents properties more consistently than other traditional crossover operators.

Hybrid Algorithms

Hybrid algorithms are a more recent technique applied to the graph colouring problem, with the first proper hybridisation being produced by C. Fleurent and J.A. Ferland (Johnson & Trick 1993). Morgenstern later proposed a population-based hybrid algorithm (Johnson & Trick 1993), which utilised local search but was void of any crossover operator and produced excellent results on benchmark graphs. The hybrid algorithms integrating local search and crossover initially showed that crossovers can improve the performance of local search, but this only occurred for a few graph instances and required significant computational effort. Since then there have been several reports implementing hybrids of various population-based algorithms with local search techniques (Galinier & Hao 1999), and these implementations have yielded very impressive results in a short time frame but on a small sample of graphs. Our solution builds on such algorithms and investigates a number of alternative algorithms on a range of graphs.

III SOLUTION

Throughout this section we discuss the implementations of our 'genetic-tabu algorithm' and 'genetic-simulated annealing algorithm' implementations. Since the 'genetic aspects' of these algorithms are identical, in the interest of efficiency we first examine our 'genetic-tabu algorithm' and later the simulated annealing operator of the 'genetic-simulated annealing algorithm'.

Representing our graphs: DIMACS standard format and NetworkX

In order to solve the *Graph Colouring Problem*, and fulfil deliverables **B1** and **A2**, we must first be able to represent our graphs using a structure to which colouring algorithms can be efficiently applied. The input representation of the benchmark graphs utilised in this project is known as the *DIMACS* standard format, this is the most common and universal method of representing graph instances within graph theory and practically any well-known graph instance can be found in *DIMACS* form. In this representation the first line of the file contains the total number of *vertices* followed by the total number of *edges*; each line proceeding this states the start-vertex and end-vertex of an edge in the graph.

Whilst being an intuitive method of easily representing any graph, it cannot be easily manipulated and even simple operations will take a significant amount of time as the graph size increases. Consequently, our colouring algorithms would be considerably slower, and less effective, if applied to a graph in this format. In order apply our algorithms to a range of *DIMACS* graphs, we alter the input representation of our graphs to a structure in which it is easy to identify each *vertex's* neighbours. There are a number of candidate representations including adjacency lists and adjacency matrices. In our case we make use of the *python* package *NetworkX*, a package designed for the creation, manipulation, and study of the structure of complex networks. *NetworkX* uses a 'dictionary of dictionaries of dictionaries' as the basic network data structure. This method of representing graphs allows for the fast lookup of *vertices* and their *neighbours*, which is imperative in providing a quality *graph colouring* implementation. Not only this but it *NetworkX* provides reasonable storage for given graphs, even in the worst case when the input graph is a very large sparse graph. As well as facilitating the efficient representation of large and complex graphs, *NetworkX* also provides graph drawing tools, allowing us to visualise the structure of given graphs and any colourings we obtain. In order to do so, we map the vertices in each colour class to a distinct colour and *NetworkX* allows using an algorithm to determine each vertex's position in the visualised colouring. This facilitates the fulfilment of deliverable **A3**, and allows us to identify and understand how an algorithm transitions from one colouring to a superior one. Due to the extensive functionality of *NetworkX* in both visualising graphs, and the various operations that it provides we utilise *Python* and *NetworkX* for the entire duration of this project.

Solving the Graph Colouring Problem: Specification, design and method

To solve the *Graph Colouring Problem*, we adopt the approach of applying an algorithm to find a k-colouring for an appropriate number of colours $k = k_0$. Whenever a k-colouring is found, we simply re-apply the same algorithm to look for a k-colouring with a decreasing number of colours $k = k - 1$. Therefore, the graph colouring problem is reduced to solving increasingly difficult k-colouring problems.

In order to find a k-colouring of a given graph $G = (V, E)$, we partition the set of vertices $V = \{V_1, \ldots, V_k\}$ into independent colour classes (subsets) $V_i \subseteq V$, such that each V_i does not contain any pair of adjacent vertices. As we decrease k this becomes exponentially harder to do. When we cannot obtain a k-colouring initially, we allocate as many vertices as possible to colour classes, whilst satisfying the constraint of the *Graph Colouring Problem*. Once we reach a point when we can no longer allocate any vertices, we randomly each of the unassigned vertices to a colour class. Our algorithm relies on ranking 'attempted' k-colourings by the number of conflicting vertices they have, where a vertex is considered conflicting if there exists an adjacent (neighbour) vertex in the same colour class as it. Over the course of the algorithms execution our solution employs stochastic methods which aim to progressively reduce the number of conflicts in a k-colouring for a given k. The algorithm terminates when a colouring produced is conflict-free or a fixed number of iterations is exceeded.

Genetic-tabu algorithm

The following presents an outline of the general procedure of the colouring algorithm produced.

Algorithm 1 Genetic-tabu hybrid algorithm

input: *graph* $G = (V, E)$, *number_of_colours*
output: *valid_colouring*, *visualised_colouring*

1: *population* = **initialise_population**(|*population*|)
2: **while** *number_of_conflicts* $\neq 0$ **do**
3: *number_of_conflicts* = **evaluate_population**(*population*)
4: $(parent_1, parent_2)$ = **selection**(*population*)
5: *child* = **crossover**$(parent_1, parent_2)$
6: *child* = **tabu_search**(*child*)
7: *population* = **update_population**$(population, parent_1, parent_2, child)$
8: **end while**

The algorithm first build an initial populations of *population size* colourings, using the *initialise population* method, and following this the initial population is improved over a number of generations. At each generation, two *parent colourings* are chosen at random, $parent_1$ and $parent_2$. A crossover operator is then applied to $parent_1$ and $parent_2$, combining them to produce a *child* colouring. The *tabu search* operator is then applied to improve the *child* colouring for a fixed number of iterations. This process repeats until a fixed number of iterations is exceeded without producing a new colouring, once the algorithm terminates the best k-colouring is output and the coloured graph is visualised. This section is focused on the fulfilment of deliverables **I1** and **A3**. The fundamental difference between our hybrid algorithm **A3** and a traditional genetic algorithm **I1** is that the typical mutation operator of a genetic algorithm is replaced with a tabu search. The function of each operator is presented in the pseudocode below.

Initialise population. The operator **initialise population**(*population size*) initiates the population with *population size* colourings. In order to create each colour, we use an adaptation of the *DSATUR* algorithm (Brélaz 1979) - a simple but very effective greedy algorithm.

The *DSATUR* algorithm works as follows. When provided with a given graph G it initially finds the largest clique $V_{max} \subseteq G$ of G and assigns each vertex $v \in V_{max}$ a distinct colour class.

Following this it begins to colour each of the remaining vertices, belonging to $V_{remaining} = V - V_{largest\,clique}$, in order to do so it iteratively selects a vertex $v \in V_{remaining}$ which is adjacent to the largest number of distinctly coloured vertices and assigns v to the lowest indexed colour class that contains no vertices adjacent to v. If there are no such colour classes, then a new colour class is created containing v. After this occurs v is removed from $V_{remaining}$ and the next vertex is considered, the algorithm proceeds until $V_{remaining} = \emptyset$, at which point all vertices have been coloured (Brélaz 1979). Through implementing this algorithm, we fulfilled **B3** and **B4**, and advanced upon **B2**. Below, we describe our adaption of *DSATUR*.

When creating a k-colouring, we are partitioning the vertices V of G into k colour classes - as a result we begin our algorithm with k empty colour classes $V_1 = \ldots = V_k = \emptyset$. During each iteration we choose a vertex $v \in V$ such that v has the minimal number of allowed colour classes, where a colour class is allowed if it does not contain a vertex adjacent to v. We then assign v to a colour class, in order to do so we choose the colour class V_i that has minimal index i. In general, this process cannot assign all the vertices. We take each unassigned vertex in turn and assign it to a randomly chosen colour class. Once we have performed the above *population size* times, we have an initial population. Due to the randomness of both our adapted '*DSATUR*' algorithm and the *tabu search* which is later performed on individuals in the population, we ensure that we maintain a diverse population. This is key to the success of the genetic aspects of our algorithm, as homogeneous population cannot evolve efficiently (Galinier & Hao 1999).

Selection and Crossover. In order to evolve our population we select a pair of parent colourings at random, and apply the crossover operator in order to produce a child colouring. The purpose of the crossover operator is to, given two parent colourings produce a child colouring which inherits characteristics from both parent colourings. The crossover implemented here works in the following way:

Crossover operator

input: $parent_1 = \{V_1^1, \ldots, V_k^1\},\ parent_2 = \{V_1^2, \ldots, V_k^2\}$
output: *child*

1: *child* $= \emptyset$
2: $k =$ *number_of_colours*
3: **for** i in $1 \leq i \leq k$ **do**
4: **if** i is odd **then**
5: *colour* = colour class with highest cardinality in $parent_1$
6: **else**
7: *colour* = colour class with highest cardinality in $parent_2$
8: **end if**
9: add *colour* to *child*
10: remove the vertices coloured using *colour* for $parent_1$ and $parent_2$
11: **end for**
12: Assign randomly any vertices that are not present in *child*

The algorithm builds, step by step, k colour classes $V_1, \ldots, V_k$ of the offspring. At each iteration, a colour class for the *child* colouring is built in the following way. We start by considering $parent_1$, and at each iteration we consider $parent_1$ or $parent_2$, whichever we have least recently considered. In the considered parent, we choose the colour class containing the maximum

number of vertices, and add this colour class to the *child* colouring. In order to ensure vertices aren't added to two different colours, we remove all of the vertices in the *child* colouring from both $parent_1$ and $parent_2$. After k iterations have occurred, a number of vertices may remain unassigned. These vertices are assigned to randomly chosen colour classes.

In Table 1 we see a worked example of the crossover operator described above, at each iteration we see that the colour class of largest cardinality in the considered *parent* is added to the child, and that the vertices belonging to the colour class are removed from both *parents*. In the case when we select $\{1\}$ rather than $\{7\}$ as the maximal colour class in $parent_1$, we choose randomly - since both colour classes have the same cardinality the choice between the two is arbitrary. We see from the above that after performing k iterations there is still an unassigned vertex 7, as stated in our method we assign 'leftover'/unassigned vertices randomly, and as a result 7 can be assigned to any of the 3 colour classes.

Table 1: Worked example of crossover of two parent 3-colourings:

$parent_1$	$\{12\}$	$\{\mathbf{3456}\}$	$\{789\}$
$parent_2$	$\{\mathbf{1}4\}$	$\{2\mathbf{3}6\mathbf{5}89\}$	$\{\mathbf{5}7\}$
$child$	$\emptyset$	$\emptyset$	$\emptyset$

Step 1: $\{3456\}$ is selected to be added to $child$ since it is the most used colour in $parent_1$.

$parent_1$	$\{12\}$	$\emptyset$	$\{789\}$
$parent_2$	$\{1\}$	$\{289\}$	$\{7\}$
$child$	$\{\mathbf{3456}\}$	$\emptyset$	$\emptyset$

Step 2: Since $3, 4, 5$ and 6 are present in $child$, they are removed from $parent_1$ and $parent_2$.

$parent_1$	$\{\mathbf{12}\}$	$\emptyset$	$\{\mathbf{789}\}$
$parent_2$	$\{1\}$	$\{\mathbf{289}\}$	$\{7\}$
$child$	$\{\mathbf{3456}\}$	$\emptyset$	$\emptyset$

Step 3: $\{289\}$ is selected to be added to $child$ since it is the most used colour in $parent_2$.

$parent_1$	$\{1\}$	$\emptyset$	$\{7\}$
$parent_2$	$\{1\}$	$\emptyset$	$\{7\}$
$child$	$\{\mathbf{3456}\}$	$\{\mathbf{289}\}$	$\emptyset$

Step 4: Since $2, 8$ and 9 are present in $child$, they are removed from $parent_1$ and $parent_2$.

$parent_1$	$\{\mathbf{1}\}$	$\emptyset$	$\{7\}$
$parent_2$	$\{\mathbf{1}\}$	$\emptyset$	$\{7\}$
$child$	$\{\mathbf{3456}\}$	$\{\mathbf{289}\}$	$\emptyset$

Step 5 & 6: Since $\{1\}$ & $\{7\}$ are both the most used colour in $parent_1$ $\{1\}$ is selected **randomly** to be added to $child$. Then 1 is removed from $parent_1$ and $parent_2$

$parent_1$	$\emptyset$	$\emptyset$	$\{\mathbf{7}\}$
$parent_2$	$\emptyset$	$\emptyset$	$\{\mathbf{7}\}$
$child$	$\{\mathbf{3456}\}$	$\{\mathbf{289}\}$	$\{\mathbf{1}\}$

Step 7 & 8: Since $\{7\}$ is the only remaining colour in $parent_2$, it is added to a random colour class in $child$ and removed from $parent_1$ and $parent_2$.

Tabu Search. The purpose of a tabu search operator is to improve a colouring produced by the crossover for a number of iterations before inserting it into the population. Here, we describe our tabu-search operator developed in fulfillment of deliverable **A3**. A tabu-search, like any local search method, requires a neighbourhood function. Starting with an initial colouring, a typical tabu-search procedure proceeds iteratively to visit a series of neighbouring colourings. At each iteration, a best neighbour is chosen to replace the current colouring, even if the former does not improve the current one. This iterative process often suffers from cycling and gets trapped in a local optima. To avoid this, the tabu-search introduces the notion of tabu lists. The purpose of the tabu list is to store the properties of recently visited configurations, and to forbid re-visiting such configurations during the next iterations. The number of iterations for which we forbid this is called the tabu tenure.The tabu-search operator used in our solution works in the following way.

Tabu Search operator

input: $colouring_{initial}$, *number_of_iterations*
output: $colouring_{best\,neighbour}$

1: $colouring_{current} = colouring_{initial}$
2: **while** $i <$ *number of iterations* **do**
3: $\quad colouring_{neighbour} =$ **get_neighbour**$(colouring_{current})$
4: $\quad$ **if conflicts**$(colouring_{current}) \leq$ **conflicts**$(colouring_{neighbour})$ **then**
5: $\quad\quad colouring_{current} = colouring_{neighbour}$
6: $\quad$ **end if**
7: $\quad i = i + 1$
8: **end while**

Here a neighbour of a given configuration is obtained by moving a single vertex v from its current colour class to another colour class. In order to make the search more efficient, the algorithm uses a simple heuristic: the vertex v to be moved must be conflicting with at least another vertex in its original class. Thus a neighbouring colouring is characterised by an action defined by the couple $(v, i) \in V \times \{1, \ldots, k\}$. When we perform such an action, we introduce (v, i) to the tabu list and as a result, it is classified tabu for a number of iterations. This ensures that v cannot be reassigned the colour i during this period, and helps to prevent cycling. Despite this, a tabu move leading to a colouring with fewer conflicts than the best colouring found so far is always accepted - aspiration criterion (Galinier & Hao 1999). The number of iterations for which an action is classified tabu is referred to as the tabu tenure, the tabu tenure is variable and depends on the number of conflicting vertices in the current colouring:

$$\textit{tabu tenure} = Random(0, k) + \alpha \cdot \textit{number of conflicts}$$

where our parameter α is dictated by the graph we are applying our algorithm to. In order to implement the tabu list, it is sufficient to use $V \times \{1 \ldots k\}$ array. The algorithm starts with the current colouring $colouring_{current}$, and after each iteration it replaces $colouring_{current}$ with the most recent $colouring_{recent}$ if $conflicts(colouring_{recent}) \leq conflicts(colouring_{current})$. The rational behind returning the most recent 'best' colouring is that we hope that the tabu-search produces a solution of sufficient quality whilst also being as far away as possible from the initial solution, in order to better preserve the diversity in the population.

Update Population. After the tabu-search has been performed on the *child* colouring, we must add the colouring to the population. In order to maintain a manageable *population size*, in terms of memory and running time costs, we must also remove a colouring from the population. To ensure that the quality of our population increases over time, we remove the *parent* colouring with most conflicts from the population, and replace it with the *child* colouring it helped to produce.

Genetic-SA algorithm

Simulated Annealing. Alongside the algorithm **Genetic-tabu** algorithm described in Algorithm 1, we also developed a genetic algorithm hybridised with a simulated annealing meta-heuristic - in order to satisfy deliverables **I2** and **I5**. This follows the same general structure of Algorithm 1, however Line 6 is replaced with a simulated annealing operator [*child* = **sim_ann**(*child*)].

The simulated annealing operator is initially provided with a colouring, and aims to improve the colouring over a series of iterations without getting caught in a local minima. The simulated annealing utilises a temperature variable which is gradually decreased using the cooling schedule $current\_temperature = current\_temperature/(1 + \log(1 + i))$ (Geman & Geman 1984), until eventually the temperature reaches a stopping temperature at which point the local search terminates and the current colouring is returned. In order to traverse the search space and consider neighbouring colourings the operator incorporates a neighbourhood function, which returns a colouring which neighbours the current colouring. Here, we say that two colourings neighbour each other if the we can obtain the neighbouring colouring by assigning one vertex a different colour, and the vertex currently conflicts (has the same colour as) with at least one of its neighbours. If the neighbouring colouring has fewer conflicts that the current colouring we make it the current colouring, otherwise we make it the colouring with probability $swapping\_prob = \exp((difference\_in\_conflicts)/\log(initial\_temperature))$. The reasoning behind introducing some probability of moving to a 'worse colouring is to explore more of the search space and ensure we escape local minima. We see that the 'swapping probability is proportional to the current temperature, as the temperature decreases so does the probability of moving to a worse colouring. The rationale behind this is that as we have performed more and more iterations of the simulated annealing we expect to hone in the global optima, and that moving to a worse quality colouring near the end of the local searches execution is likely to hinder the algorithms performance. The swapping probability is impacted significantly by the $difference\_in\_conflicts$, since the swapping probability is only utilised if $difference\_in\_conflicts < 0$ the more conflicts the neighbouring colouring has the less likely we are to move to it.

Simulated Annealing operator

input: $colouring_{initial}$
output: $colouring_{current}$

$colouring_{current} = colouring_{initial}$
$initial\_temperature = number\_of\_vertices$
$stopping\_temperature = number\_of\_vertices/4$
$i = 0$
while $current\_temperature < stopping\_temperature$ **do**
    $colouring_{neighbour} =$ **get_neighbour**$(colouring_{current})$
    $difference\_in\_conflicts =$ **conflicts**$(colouring_{current}) -$ **conflicts**$(colouring_{adjacent})$
    $swapping\_prob = \exp(conflicts\_difference/\log(initial\_temperature))$
    if $conflicts\_difference \geq 0$ **then**
        $colouring_{current} = colouring_{neighbour}$
    else if $random[0, 1] <= swapping\_prob$ **then**
        $colouring_{current} = colouring_{neighbour}$
    end if
    $current\_temperature = current\_temperature/(1 + \log(1 + i))$
    $i = i + 1$
end while

Our remaining deliverables **I3** and **I4**, concerning the testing and experimentation of algorithms. As such they are investigated in the Results section.

IV RESULTS

Throughout this section, we present the results obtained by our 'genetic-tabu' and 'genetic-SA', compared with the *DSATUR* greedy algorithm and best-known upper bounds on the chromatic number.

Test instances

We perform our algorithms on 17 well-known benchmark graph instances, produced as part of the reputable 'DIMACS Implementation Challenge' (Johnson & Trick 1993). In order to assess our algorithms efficiently, we utilise a number of graphs with varying properties. The benchmark graphs utilised are stated below, as well as the graph class each graph instance belongs to:

- *Register Allocation Graphs*: We tested our algorithms on a total of 6 graphs based register allocation for variables in real codes, the chromatic number is known for each of these graphs (Johnson & Trick 1993). These were '*mulsol.i.1*', '*mulsol.i.3*', '*fpsol2.i.1*', '*fpsol2.i.3*', '*inithx.i.1*' and '*inithx.i.2*' respectively.
- *Leighton Graphs*: Leighton graphs are random graphs with a fixed number of edges and a predetermined chromatic number. The following graphs produced by *Frank Thomas Leighton* will display our algorithms proficiency in obtaining increasingly difficult chromatic numbers: '*le450_25a*', '*le450_15b*' and '*le450_5a*'. Each of these graphs having 450 vertices and 8260, 8169 and 5714 edges respectively.
- *Random Geometric Graphs*: We also utilise two random geometric graphs, these are graphs such that vertices are randomly positioned in a space with edges added to connect pairs of vertices which are close to each other (Penrose 2003). The two random geometric graphs we use are '*DSJR500.1*' and *DSJR500.5*, both having 500 vertices. With '*DSJR500.1*' having 3555 edges and '*DSJR500.5*' having 58862 edges.
- (n, p) *Random Graphs*: A total of 6 (n, p) random graphs provided by *David Johnson* where n is the number of vertices and p is the probability their exists an edge between any pair of vertices. We first consider '*DSJC125.1*', '*DSJC125.5*' and '*DSJC125.9*'. Each of these graphs have 125 vertices, whilst their 'edge probabilities' and consequently densities differ significantly. Each graph instance having approximate densities of 0.1, 0.5 and 0.9 respectively. We also assess '*DSJC125.1*' alongside another three such graphs '*DSJC250.1*', '*DSJC500.1*' and '*DSJC1000.1*' - in this case each of the graphs have approximately the same density of 0.1, whilst the number of vertices as 125, 250, 500 and 1000.

The range of graph-class, number of vertices and densities of the graphs in this set of benchmark graphs allowed us to gain an in-depth insight into our algorithms performance and work on deliverables **I4** and **A2**.

Table 2, presents the best performance of our algorithms and *DSATUR* applied to the graph instances described above.

Table 2: The best performance of our algorithms and *DSATUR* applied to a sample of 17 benchmark graphs, chosen in fulfilment of deliverables **I3**, **I4** and **A2**.

		Genetic-tabu hybrid			*Genetic-SA hybrid*			*DSATUR*	
Graph	χ	k	*Generations*	*Time* (s)	k	*Generations*	*Time* (s)	k	*Time* (s)
mulsol.i.1	49	49	1	1.14	49	1	0.77	49	0.01
mulsol.i.3	31	31	1	0.99	31	1	0.81	31	0.07
fpsol2.i.1	65	65	1	6.93	65	1	6.48	65	0.64
fpsol2.i.3	30	30	1	5.26	30	1	5.08	30	0.37
inithx.i.1	54	54	1	20.76	54	1	21.16	54	1.80
inithx.i.2	31	31	1	12.41	31	1	12.33	31	1.43
le450_25a	25	25	1	1.23	25	1	7.50	25	1.14
le450_15b	15	15	92	1131.45	17	1	7.54	17	0.28
le450_5a	5	5	313	4527.22	9	667	15010.34	10	0.36
dsjr500.1	12	12	2	32.45	13	1	25.79	13	0.33
dsjr500.5	122	123	1249	7419.06	126	1	37.88	130	0.56
dsjc125.1	5	5	137	1835.32	5	635	7408.33	6	0.02
dsjc125.5	17	17	365	12186.91	23	1	0.58	22	0.09
dsjc125.9	44	44	773	20804.67	51	42	490.46	51	0.11
dsjc250.1	8	8	563	647.12	10	1	1.87	10	0.08
dsjc500.1	12	12	1416	4611.92	16	1	8.18	16	0.33
dsjc1000.1	20	22	2634	83111.54	27	1	50.26	27	1.40

Algorithm Comparison

In order to determine the proficiency of our solution we compare the performance of our algorithms with the standard *DSATUR* algorithm and the best-known upper bounds on the chromatic number for the benchmark graphs. When analysing performance we focus on two main factors, solution quality and computational effort required in order to obtain a solution. Due to the operational differences of the algorithms, solution quality is measured by simply observing the smallest number of colours used in a feasible solution. Whilst the computational effort is measured by the number of generations (crossovers) and real-time required. We allow each of the algorithms to run for a maximum of 24 hours, or until a solution deemed sufficient is attained. The amount of time is deliberately set to be high in order to draw attention to the excess time required by some methods.

Table 2 present upper bounds of the chromatic number for the benchmark graphs stated above, obtained by various algorithms from related works (Johnson & Trick 1993). We include the time taken to obtain the upper bound on the chromatic number and the number of generations required for our population based approaches. Due to the stochastic nature of the two approaches, the time values and generations required are average values. The chromatic number χ and best upper bounds are sourced from (Resende & Sousa 2004) and (Johnson & Trick 1993) respectively.

In general for the genetic-tabu approach, when defining *tabu tenure* $= Random(0, k) + \alpha \cdot$ *number of conflicts*, we chose $\alpha = 0.6$ and this was a robust choice for all the graph instances we tested against. Whilst altering α on a case-by-case basis can lead to marginally improved results for some graph instances the value $\alpha = 0.6$ offered a reasonable compromise for the range of graph instances investigated, and in interest of generality we decided to fix this value for all runs

of the algorithm. For similar reasoning, we also decided to fix the number of iterations of the tabu algorithm to 2000, as not only could this number of iterations be performed in reasonable time but lesser number of iterations seemed to hinder the algorithm in obtaining optimal colourings.

From Table 2, we see that the in the case of the leighton graphs since they all have the same number of vertices and a number of edges of the same order, the graphs with lower chromatic number are more difficult to colour optimally. In each case, as we reduce the chromatic number the number of generations required by our tabu-genetic algorithm increases dramatically. In contrast to this, despite obtaining an optimal colouring for the first graph instance with relative ease, the 'DSATUR' algorithm (Brélaz 1979) and genetic-SA quickly yield colourings which are far from optimal. With the upper bound obtained for the chromatic number of 'le450_5a' being 100% and 80% from the actual chromatic number.

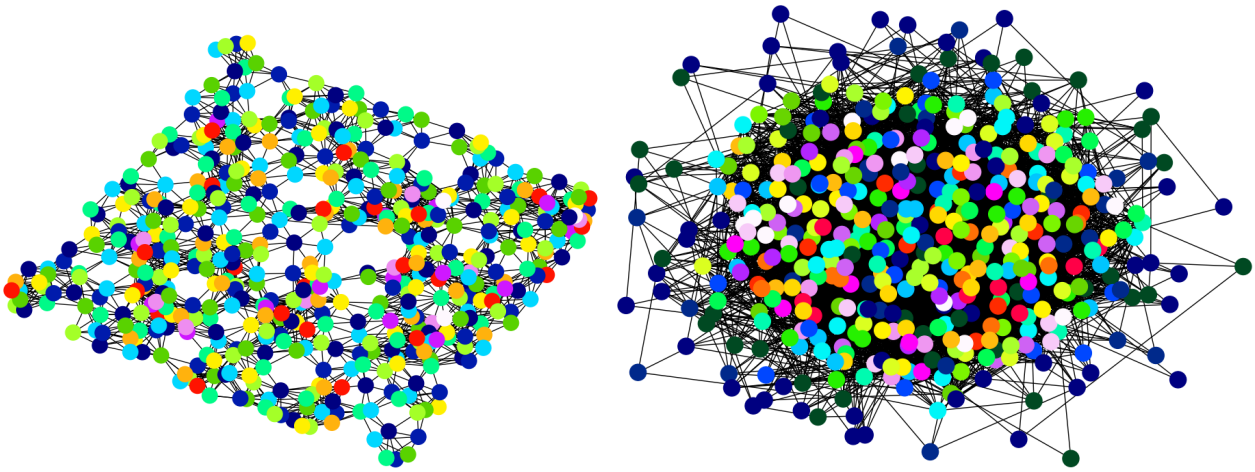

Figure 3: Optimal colourings for **dsjr500.1** and **le450_25a** respectively, visualised using our implementation, in contribution to **A1**.

In Table 2, we see the same trends across all sets of graphs, and our genetic-tabu consistently outperforms the less advanced methods. Whilst we see no significant difference for the more simple graph instances for larger graphs our algorithms strategy of exploring the space of infeasible solutions using the tabu-search search operator proves most successful. The '*DSATUR*' algorithms insistence on preserving feasibility implies a lower level of connectivity in its underlying solution space, resulting in noticeably inferior solutions.

In Figure 3 we see the colourings provided by our algorithms for the '*DSJR500.1*' and '*le450_25a*' instances, these are presented here not for analytical purposes but to give an impression of the complexity of the networks and as a result the difficulty in obtaining optimal colourings for them.

V EVALUATION

As we can see the results of our project yield a complicated picture when determining which algorithms will perform well on the *Graph Colouring Problem*, and this is expected due to the longevity of the problem and the lack of success experienced by many algorithms when applied to it. Despite this, our 'genetic-tabu' algorithm produces optimal colourings for many of the graphs provided and we see a vast improvement on some of the basic and even some of the more advanced methods mentioned in the related work section.

Strengths of approach

During this section we shall analyse the main factors that the 'genetic-tabu' algorithms success can be attributed to and also some limitations and reliance of the algorithm that ensure that there exist graphs for which the algorithm will not provide an optimal colouring.

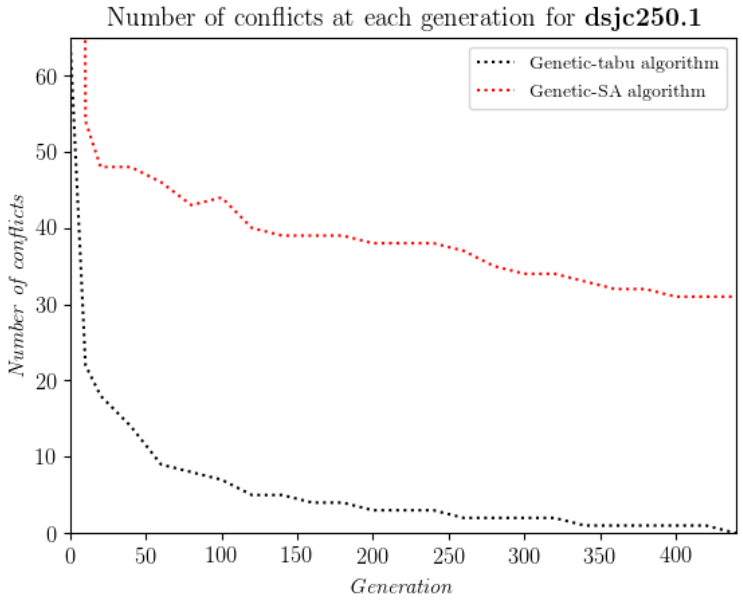

Figure 4: A graph displaying the decline in number of conflicts in a colouring over the course of the execution of our algorithm, for the graph *dsjc250.1*.

The algorithm operates for the most part in the space of infeasible solutions, meaning that when we are trying to obtain a k-colouring for a given graph, at any given point the colourings in the population are likely to contain conflicts (i.e. numerous pairs of adjacent vertices may have the same colour). At first glance it may not be clear why this improves the performance of the algorithm but the space of infeasible solutions seem to offer higher level of connectivity, and thus less restriction of movement. This property allows our algorithm to navigate through infeasible solutions towards proper (feasible) k-colourings more easily, and also does not incorporate many traditional 'navigation techniques' that can often lead to the algorithm being stuck in a local optima. In Figure 4 we see that the algorithm consistently improves upon the number of existing conflicts and that the exploration of the non-feasible space allows the algorithm to navigate to a feasible solution with relative ease (for this particular graph).

Unlike the aforementioned 'traditional genetic algorithm' or other such methods including the 'ant colony' metaheuristic the global search operators (i.e. our new crossover) are robust. In contrast to traditional crossovers which have experienced little to no success when applied to the graph colouring problem, our crossover operator performed its purpose well. This is due to it striking an appropriate balance between maintaining useful substructures from colourings formed in earlier iterations, i.e. ensuring a *child colouring* inherited necessary characteristics from parents; and also altering the colourings sufficiently in order to ensure the algorithm navigated towards an optimal colouring.

Our solution also makes use of one an efficient local search operator in the form of the tabu

search, however when applied to graph colouring in isolation the tabu search algorithm performs poorly. This is due to the fact that it can regularly become trapped in local optima. In contrast, our solution alleviates this problem (except in extreme cases) due to its population-based approach and the aforementioned global search operators. The tendency for local search algorithms to become stuck in local optima is the main reason that a population-based algorithm is necessary for such an algorithm, and as a result our algorithm yields much higher quality colourings. From Figure 4 we see that the the tabu operator implemented is far superior to the simulated annealing. This is due to the aggressive approach adopted by the simulated annealing of always moving to any colouring which improves the current one and this very quickly leads to a local optima. In contrast, the aspiration criteria employed by the tabu search to always move to colourings improving the 'current best colouring' which is far less aggressive and more suitable to the *graph colouring problem*.

Limitations and further improvements

Maintaining Diversity

As prior mentioned an important factor behind the behaviour of any evolutionary/genetic algorithm is maintaining diversity and many methods of doing so have been proposed (Gupta & Ghafir 2012). Typically, during earlier iterations of our algorithm the diversity of a population will be high, allowing the algorithm to consider more of the solution. As the population evolves over time, we hope that the amount of diversity falls as the algorithms 'homes in' on promising regions of the solution space and seeks to search these areas more thoroughly. In (Lewis 2015) these are described as the exploration and exploitation phases of the algorithm.

To analyse the fluctuation of diversity throughout the execution of our algorithm we use the following definition of diversity between two solutions (partitions) C_i and C_j - this is the *Jacard distance* between two sets:

$$D(C_i, C_j) = \frac{|C_i \cup C_j| - |C_i \cap C_j|}{|C_i \cap C_j|}.$$

In order to calculate the diversity for the entire population P we calculate the weighted sum:

$$\textit{Diversity}(P) = \frac{1}{\binom{|P|}{2}} \sum_{\forall C_i, C_j \in P : i < j} D(C_i, C_j).$$

Through calculating the diversity in the population after every 20 iterations, and plotting the result for an execution both of our population-based algorithms on the graph **dsjc250.1**, we obtain Figure 5 which is a somewhat surprising plot. Here we see that the diversity in the population is maintained for several generations but falls relatively quickly, this is referred to as *premature convergence*. This is an issue, particularly for the 'genetic-SA' implementation, as the majority of the run-time is spent examining very small areas of the search space. Hence why we see in Table 2 that the genetic-SA tends to obtain a solution quickly or not at all. Numerous methods for combating premature convergence have been presented including: *Multiple parents*, *Restricted Mating* and *Kempe chain interchanges*. However, there is considerable debate over how well each of them aid in managing diversity. This is due to them being unable to arrive at an appropriate balance between population diversity and selective pressure (i.e. enforcing that poor colourings leave the population).

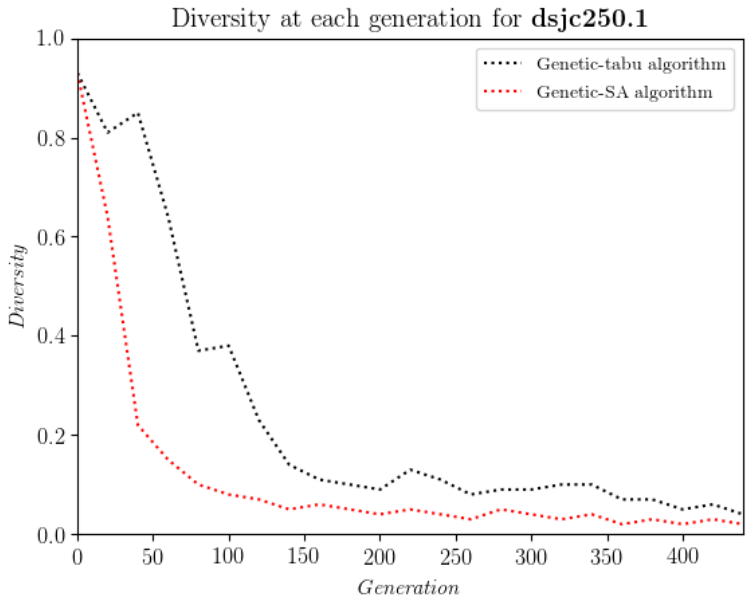

Figure 5: A graph displaying the fluctuation of diversity in a population of colourings over the course of the execution of our population-based algorithms (genetic-SA terminated without finding a proper colouring), for the graph *dsjc250.1*.

VI CONCLUSIONS

During the course of this project we implemented and experimented with a tabu-genetic algorithm which was applied to the *Graph Colouring Problem* and yielded colourings which were optimal or equalled the best-known k-colouring in most cases. Our algorithm was built on two fundamental principles that pose a stark contrast to many of the current well-established graph colouring algorithms. In our implementation, a colouring is a partition of vertices and not an assignment/mapping of colours to vertices, and the crossover implemented should transmit subset of colour classes from parents to offsprings (Galinier & Hao 1999). Our algorithm significantly improves on the tabu search methods proposed by (Hertz & de Werra 1987), suggesting that the crossover operator and the use of a population of solutions significantly improve the colourings obtained which is somewhat surprising due to the lack of success that 'traditional Genetic Algorithms' experience (Davis 2003).

We saw that the solution proposed consistently produce near-optimal results for each of the graph instances we applied it to, and this can be attributed to the algorithms ability to quickly identify promising whilst maintaining sufficient diversity during the early stages of each execution. This contrasted significantly to our 'genetic-SA' algorithm, which was unable to maintain diversity optimally, and this resulted in relatively poor performance as seen in Table 2. Further investigation revealed that as we increase the number of vertices and density of the graph instances the time required by the algorithm increases exponentially, and often the final colouring was of lesser quality than the best-known upper bounds. This is to be expected since the search space we need to consider also increases in size exponentially.

However, as we studied larger and more complex graphs the algorithm struggled more and more to find an optimal colouring. Without the development of new technology algorithms will

struggle to analyse the massive search space as graph instances become larger and more complex. The *Graph Colouring Problem* remains a very difficult problem and the performance of a given algorithm is clearly dependent on the underlying structure of the graph to which the algorithm is applied, and for this reason a variety of methods experience varied success when applied to the problem.

Future work may involve investigating more recent advances in local search heuristics for graph colouring such as the tabu based algorithm proposed by (Porumbel 2008), which utilises an alternative evaluation function which is dynamically adjusted throughout the execution of an algorithm. This was notably implemented with a simple 'steepest descent algorithm' and it yielded results far superior to traditional evaluation functions (based only the current number of conflicts) and preliminary results suggest it could improve the performance of algorithms such as ours.

References

Appel, K. & Haken, W. (1976), 'Every planar map is four colorable'.

Brélaz, D. (1979), 'New methods to color the vertices of a graph', p. 252.

Chaitin, G. (1982), 'Register allocation & spilling via graph colouring'.

Chartrand, G. & Zhang, P. (2009), *Chromatic graph theory*, Chapman & Hall/CRC/Taylor & Francis Group.

Davis, L. (2003), *Handbook of genetic algorithms*, Van Nostrand Reinhold: New York.

Galinier, P. & Hao, J. (1999), 'Hybrid evolutionary algorithms for graph coloring', pp. 384–387.

Geman, D. & Geman, S. (1984), 'Stochastic relaxation, gibbs distributions, and the bayesian restoration of images', pp. 721–741.

Gupta, D. & Ghafir, S. (2012), 'An overview of methods maintaining diversity in genetic algorithms', **2**.

Halldórsson, M. M. (1993), 'A still better performance guarantee for approximate graph coloring'.

Hertz, A. & de Werra, D. (1987), 'Using tabu search techniques for graph coloring'.

Johnson, D. & Trick, M. (1993), *Cliques, Coloring, and Satisfiability*, American Mathematical Society.

Leighton, F. T. (1979), 'A graph coloring algorithm for large scheduling problems', pp. 490–491.

Lewis, R. (2015), *A Guide to Graph Colouring: Algorithms and Applications*, Springer.

Malaguti, E., M. M. & Toth, P. (2010), 'An exact approach for the vertex coloring problem.'.

Marappan, R. & Sethumadhavan, G. (2017), 'Solution to graph coloring using genetic and tabu search procedures.', pp. 525–528.

Marco Chiarandini, I. D. & Stützle, T. (2003), 'Local search for the colouring graph problem. a computational study'.

Mitchem, J. (1973), 'On various algorithms for estimating the chromatic number of a graph', pp. 182–183.

Penrose, M. (2003), *On various algorithms for estimating the chromatic number of a graph*, Oxford University Press.

Porumbel, D., H. J. K. P. (2008), *A study of evaluation functions for the graph K-coloring problem.*, Springer.

Resende, M. & Sousa, J. (2004), *Metaheuristics*, Springer.

Tarjan, R. E. & Trojanowski, A. E. (1976), 'Finding a maximum independent set'.

Trudeau, R. (1993), 'Introduction to graph theory', p. 64.

Vose, M. D. (1999), 'The simple genetic algorithm: Foundations and theory', pp. 21–36.

Wikipedia (2007), 'Four color theorem'.

Zuckerman, D. (2007), 'Linear degree extractors and the inapproximability of max clique and chromatic number'.