Literature Review of Optimisation Techniques to the Graph Colouring Problem

Scott Whitemore & Brendon Veronese

October 30, 2015

Abstract

A review of papers concerning the current state-of-the-art for solving the GCP is presented. Algorithms from these papers are implemented and compared, results are shown in an unbiased way.

Chapter 1

Literature Review

1.1 Introduction

The graph coloring problem has widespread applications in areas such as scheduling and timetabling. Bondy and Murty describe graph coloring as follows:

> A $k - vertexcolouring$ of G is an assignment of k colours, $1, 2, \ldots, k$, to the vertices of G; the colouring is *proper* if no two distinct adjacent vertices have the same colour. Thus a proper k-vertex colouring of a loopless graph G is a partition $(V_1, V_2, \ldots, V_k)$ of V into k (possibly empty) independent sets. G is $k - vertex - colourable$ if G has a proper k-vertex colouring. It will be convenient to refer to a 'proper k-vertex colouring as a $k - colouring$'. The *chromatic number*, $\chi(G)$, of G is the minimum k for which G is k-colourable; if $\chi(G) = k$, G is said to be $k - chromatic$

The Graph Coloring Problem (GCP) is the task of trying to find the value for $\chi(G)$ for some given G, however, this is an NP-Hard problem of at least $O(N)$ complexity and so finding a k that is guaranteed to be $\chi(G)$ is often ($N >$ single digits) impossible. As such we hope to be able to develop heuristic algorithms that are able to find near-optimal colorings ($k \rightarrow \chi(G)$) quickly (seconds/minutes/hours depending on the size and complexity of the graph).
This work is laid out as follows: In chapter 1 we will introduce four different methods for solving the GCP that have been proposed by other authors: Simulated Annealing, Gravitational Swarm Intelligence, Genetic Algorithm and Flower Pollination Algorithm. The papers that describe these algorithms, as well as the algorithms themselves, vary in quality and we will be commenting on particularly bad or confusing aspects as they crop up. We attempted to implement the latter three methods in Java going off of just what was provided in the papers, to mixed success. A write up of these attempts, as well as some pseudocode, is presented in chapter 2. Finally, chapter 3 has the results of applying the three algorithms to some of the DIMACS benchmark graphs.

1.2 Johnson - Simulated Annealing

Johnson et al.'s three part series on Simulated Annealing published in 1991 is perhaps the seminal work on the subject. The second part of the series covers the well studied yet never satisfactorily solved *Graph Coloring Problem*. You didn't see that one coming did you?

The paper begins by introducing introducing the context, including what the GCP is, and presenting a general simulated annealing algorithm.
So what is simulated annealing? Simulated annealing is an adjustment to local optimisation that has the ability to escape local optima (to a point). By using a temperature parameter that reduces as the algorithm progresses to govern when an uphill move can be made, simulated annealing is able to move around the solution space easily at early iterations but becomes locked in at later iterations.
In order to approach the GCP from an optimisation stand point three components of the optimisation scheme have to be established: a neighbourhood graph that describes the solution space, a way to move through the solution space, and an initial solution.
Three types of neighbourhoods and movement strategies are proposed, leading to three different SA algorithms for solving the GCP. All algorithms use some sort of randomised initial solution suitable for their particular implementation.

1.2.1 Penalty Function approach

- A solution will be *any* partition of V into nonempty disjoint sets $C_1, C_2, \ldots, C_k$, whether the C_i are legal color classes or not.
- Two solutions are neighbours if one can be transformed to the other by moving a vertex from one color class to another.
- To generate a random neighbour by randomly picking a nonempty color class C_{OLD}, a vertex $v \in C_{OLD}$, and then an integer i, $1 \leq i \leq k+1$. The neighbour is obtained by moving v to C_i. If $i = k+1$ then v is moved to a new, previously empty class.
- Note that this biases the choice of v towards vertices in smaller color classes.
- Adopts the general philosophy of RLF, which constructs its colorings with the aid of a subroutine for generating large independent sets.
- Cost function has two components, the first favours large color classes, the second favours independent sets.
- Let $\Pi = (C_1, \ldots, C_k)$ be a solution and E_i, $1 \leq i \leq k$ be the set of edges from E both of whose endpoints are in C_i, i.e., the set of *bad* edges in C_i.
- Cost Function

$$cost(\Pi) = -\sum_{i=1}^{k} |C_i|^2 + \sum_{i=1}^{k} 2|C_i|\dot{|}E_i|$$

1.2.2 Kempe Chain approach

- Solutions are now restricted to be partitions $C_1, C_2, \ldots, C_k$ that are legal colorings.
- Cost function simplifies to

$$cost(\Pi) = -\sum_{i=1}^{k} |C_i|^2.$$

- Moves through the solution space are based on *Kempe Chains*
- Suppose that C and D are disjoint independent sets in a graph G.

- A Kempe chain for C and D is any connected component in the subgraph of G induced by $C \cup D$.
- Let $X \Delta Y$ denote the symmetric difference $(X - Y) \cup (Y - X)$.
- Let H be a Kempe chain of C and D, then $C \Delta H$ and $D \Delta H$ are themselves disjoint independent sets whose union is $C \cup D$.
- Randomly choose a nonempty color class C and a vertex $v \in C$.
- Randomly choose a nonempty color class D other than C
- Let H be the *Kempe chain* for C and D that contains v.
- Repeat until one obtains C, D, v and H s.t. $H \neq C \cup D$
- The next partition is obtained by replacing C by $C \Delta H$ and D by $D \Delta H$

1.2.3 Fixed-K approach

- Takes a different approach to the previous methods. As the name suggests, this algorithm runs for some fixed k.
- Solutions are any partitioning of V into k color classes.
- Attempts to minimise the number of *bad* edges.
- A partition Π_2 is a neighbour of a partition Π_1 if the two partitions differ only as to the position of a single vertex v and v is an endpoint of a bad edge in Π_1.
- A legal coloring has no neighbours since if a legal coloring is found the algorithm stops.

1.2.4 Testing

In order to form a comparison to the current methodologies of the time, three Successive Augmentation algorithms are also run: DSATUR, RLF and XRLF (an extension of the former). All algorithms are run on random graphs of various sizes and edge probabilities, some "cooked" graphs that have chromatic numbers approximately half the equivalent random graph, and some geometric graphs that have certain properties as well as their complements.

1.2.5 Author's Conclusions

- Experimental results did not allow the authors to identify a best graph coloring heuristic. Rather, they feel that their results reinforce the notion that there is no best heuristic.
- In general Kempe chain annealing starts to take over fixed-K annealing as density increases. This makes sense given their neighbourhood structure.
- Penalty function annealing, Kempe chain annealing and RLF are biased towards favouring unbalanced colorings, i.e. better a large and a small sized class than two equally sized classes. Fixed-K on the other hand is neutral, potentially giving it an advantage on graphs whose good colorings are balanced.

1.2.6 Wrap-up

Johnson et al. 1991 is a highly regarded and often referenced mainstay in the field of optimisation for good reason. The paper is well written, describes the methods in good detail and reports on extensive experimentation. On a personal note, I was very disappointed that the manuscript by Shapiro and Morgenstern that Johnson et al. cites for their Kempe Chain implementation is not available online and in fact only exists as a physical copy in three locaitons in the world: Centrum Wiskunde en Informatica - Amsterdam, NL 1098 XG Netherlands; University of New Mexico-Main Campus - Albuquerque, NM 87131 United States; and Stanford University Libraries - Stanford, CA 94305 United States.

1.3 Gravitational Swarm Intelligence

The natural inspiration of this algorithm does not come from living beings, such as ants or bees, but from the basic physical law of gravitational attraction between objects. We construct a world that agents navigate through, attracted by the gravitational pull of specific objects, the color goals, such that they may suffer specific repulsion forces, activated by the friend-or-foe nature of the relation between agents induced by the adjacency relation in the underlying graph.

Initial definition: Let $G = (V, E)$ be a graph defined on a set of nodes $V = \{v_1, \ldots, v_N\}$ and edges $E \subseteq V \times V$. We define a group of GS-GC agents $B = \{b_1, b_2, \ldots, b_N\}$ each corresponding to a graph node. Each agent navigates inside a square planar toric world according to a speed vector $\overrightarrow{v_i}$. At any moment in time we know the position attribute of each agent $p_i(t) = (x_i, y_i)$ where x_i and y_i are the Cartesian coordinates in the space. When $t = 0$ we have the initial position of the agents $p_i(0) = (x_0, y_0)$. Suppose that we want to color the graph with K colors, denoting $C = \{1, 2, \ldots, K\}$ the set of colors, where K must not be lower than the chromatic number of the graph for the GS-GC to converge. We assign to these colors, K fixed points in space, the color goals $CG = \{g_1, \ldots, g_K\}$, endowed with a gravitational attraction resulting in a velocity component $\overrightarrow{v_{gc}}$ affecting the agents. The attraction force decreases with the distance, but affects all the agents in the space.

The problem collapses into the minimisation of a cost function:

$$min|\{b_i \textbf{ s.t. } b_i \in \{g_1, \ldots g_N\}\}|$$

We denote the set of agents whose position is in the region of the space near enough to a color neighbourhood of the color as:

$$\mathcal{N}(g_k) = \{b_i \textbf{ s.t. } \|p_i - g_k\| < nearenough\} \tag{1.1}$$

We denote the fact that the node has been assigned to the corresponding color assigning value to a the agent color attribute

$$b_i \in \mathcal{N}(g_k) \Rightarrow c_i = k \tag{1.2}$$

The initial value of the agent color attribute c_i is zero or null. Inside the spatial neighbourhood of a color goal there is no further gravitational attraction. However, there may be a repulsion force between agents that are connected with an edge in the graph G. This repulsion is only effective for agents inside the same color goal neighbourhood. To model this effect, we define function repulsion which has value 1 if a pair of GS-GC agents have an edge between them, and 0 otherwise. The

repulsive forces experimented by agent b_i from the agents in the color goal g_k are computed as follows:

$$R\left(b_i, g_k\right) = \sum_{\mathcal{N}(g_k)} repulsion\left(b_i, b_j\right) \tag{1.3}$$

The cost function defined on the global system spatial configuration is:

$$f\left(B, CG\right) = |\{b_i \textbf{ s.t. } c_i \in C \& R\left(b_i, g_{c_i}\right) = 0\}| \tag{1.4}$$

This cost function is the number of graph nodes which have a color assigned and no conflict inside the color goal. The agents outside the neighbourhood of any color goal can't be evaluated, so it can be a part of the solution of the problem. The dimension of the world and the definition of the *nearenough* threshold allows controlling the speed of convergence of the algorithm. If the world is big and the *nearenough* variable is small then the algorithm converges slowly but monotonically to the solution, if the world is small and the *nearenough* variable is big the algorithm is faster but convergence is jumpy because the algorithm falls in local minima and needs transitory energy increases to escape them. The reason of this behaviour is that the world is not normalized and the magnitude of the velocity vector can be larger than the radius of the color goal's spatial influence and this means an agent could potentially cross a goal without being captured by it.

Each color goal has an attraction well spanning the entire space, therefore the gravitational analogy. But in our approach the magnitude of the attraction drops proportionally with the Euclidean distance d between the goal and the GS-GC agent, but it never disappears. If $\|d\| < nearenough$ then we make $d = 0$, and the agent's velocity becomes 0, stopping it.
We now present a more formal definition of the algorithm.

Definition: A Gravitational Swarm (GS) is a collection of particles $P = \{p_1, \ldots, p_L\}$ moving in an space S subjected to attraction and repulsion forces. Attraction correspond to long range gravitational interactions. Repulsions correspond to short range electrical interactions. Particle attributes are: spatial localization $s_i \in S$, mass $m_i \in \mathbb{R}$, charge $\mu_i \in \mathbb{R}$, set of repelled particles $r_i \subseteq P$. The motion of the particle in the space is governed by equation:

$$s_i\left(t\right) = -m_i\left(t\right) A_i\left(t\right) + \mu_i\left(t\right) R_i\left(t\right) + \eta\left(t\right) \tag{1.5}$$

where $A_i(t)$ and $R_i(t)$ are the result of the attractive and repulsive forces, and $\eta(t)$ is a random (small) noise term. The attractive motion term is of the form:

$$A_i(t) = \sum_{p_j \in P - r_i} m_j(t)(s_i - s_j)\delta_{ij}^A \tag{1.6}$$

where

$$\delta_{ij}^A = \begin{cases} \|s_i - s_j\|^{-2} & \|s_i - s_j\|^2 > \theta^A \\ 0 & \|s_i - s_j\|^2 \leq \theta^A \end{cases} \tag{1.7}$$

The repulsive term is of the form

$$R_i(t) = \sum_{p_j \in r_i} \mu_j(t)(s_i - s - j)\delta_{ij}^R \tag{1.8}$$

where

$$\delta_{ij}^{R} = \begin{cases} \|s_i - s_j\|^{-2} & \|s_i - s_j\|^2 \leq \theta^R \\ 0 & \|s_i - s_j\|^2 > \theta^R \end{cases} \tag{1.9}$$

The two delta functions have different roles in the definition of the GS. The attractive δ^A corresponds to the inverse of the distance and is the strength of attraction. To avoid singular values when two particles are close to zero distance we set a threshold θ^A which determines the region around the particles where the motion due to attraction forces disappear. The repulsive δ^R defines for each ij, the maximum extension of the repulsive forces, which are short range forces. The threshold θ^R determines the region around the particles where the repulsive forces are active.
A vertex particle of a GS-GC reaches zero velocity if and only if it is at distance below θ^A of a color particle and no repulsive particle is in θ^R range.
A global state of the GS-GC is stationary if and only if all vertex particles are placed in the neighbourhood of some color particle without any repulsive particles located at the same color particle neighbourhood. If the graph's chromatic number $M^\star$ is smaller than or equal to the number of color particles $M^\star \leq M$, there will be a non-empty set of stationary states of the GS-GC.
Any stationary state of the GS-GC corresponds to a graph colouring. If the graph's chromatic number is greater than the number of color particles, there are no stationary states in the GS-GC.
These conditions mean that it is always possible to (given enough time) find the chromatic number for a graph. The algorithm to do this is outlined in the following sections.

1.4 Genetic Algorithm

1.4.1 Introduction

The genetic algorithm presented in this review is taken from this paper [2]. The paper presents a hybrid technique that applies a genetic algorithm followed by wisdom of crowds voting. The algorithm uses a variety of parent selection, child production and mutation steps. The methods of parent selection, child production and mutation vary according to the state of the fitness of the overall system. This results in an algorithm that is resistant to capture by local optima and allows for the solution space to 'jitter' around before falling upon a global solution.

1.4.2 Algorithm Discussion

The algorithm is presented in further detail in the Implementations section. The following discussion focuses on algorithm design and fine tuning for the GCP. The algorithm makes careful note that local optima are a concern to this algorithm, and the danger posed to any GCP solving heuristic algorithm is falling into a local optima and not being able to recover. The algorithm uses several approaches to avoid local optima and to that end, a number of different factors considered.
The crossover function, the method that produces a new chromosome from two previously existing chromosomes is the result of a tournament between parents, where 4 chromosomes are chosen at random and the best for the first pair becomes the first parent, and the best of the second pair becomes the second parent. The crossover function takes the chromosomes of the parents and cuts them both at the same point. The new chromosome is the first part of the first parent's chromosome with the second part of the second parents chromosome appended to the end. This results in a new chromosome of the same length as the original parents chromosomes, and a potentially vastly different 'bad edge' count. This new chromosome can then be subjected to mutation (at some rate,

given to be 0.7 here). This mutation method examines each vertex in the chromosome and tries to improve it if it is part of a 'bad edge'.
This process is shown to be very efficient at improving the solution space. It falls prone to the problem of local optima though, so an alternative parent selection and mutation method are employed. These new methods simply take the best solution and mutate it. The mutation allows the solution to become worse, instead of simply trying to improve the solution, each bad edge is allowed to become any colour. This results in the best solution having the possibility of improving, but most likely actually becoming worse. This is not a bad thing though, as the solution space is shown to improve over time.
If no valid solution to the colouring is found after a set number of generations (given to be 20,000 in this paper), then a wisdom of crowds approach is implemented, where the best portion (given to be half) of the chromosomes vote to create a new chromosome. This aggregated chromosome is checked to see if it contains a solution to the graph, if not, it becomes the first chromosome is the next round of generations, and all other chromosomes are discarded.

1.4.3 Observations

This algorithm can become very stagnated when the best solution is very close to an optimal solution. To improve this part of the algorithm, the algorithm implementation was changed to instead choose the best parent and a random chromosome on some chance.

1.5 Flower Pollination

1.5.1 Introduction

The flower pollination algorithm[5] (FPA) was first described by Xin-She Yang[1]. It is a metaheuristic global optimisation algorithm inspired by the pollination process of flowering plants. Although originally formulated to solve continuous optimisation problems, Meriem Bensouyad[2] and DjamelEddine Saidouni[2] propose a discrete version of FPA for solving the graph coloring problem[1].

Both papers describe pollination as follows (with an attempt at English corrections):

> Pollination can take two major forms: abiotic and biotic. About 90% of flowers belong to the class of biotic pollinating flowers, that is, their pollen is transferred by pollinators such as insects, birds, bats and other animals. The remaining 10% of flowers use abiotic pollination, which does not require any pollinators. Wind and diffusion in water help pollination of such flowering plants, of which grass is a good example. Pollinators, or sometimes called pollen vectors, can be very diverse. It is estimated that there are at least 200,000 varieties of pollinators.

> A concept that comes up when talking about pollinators is *constancy*, which refers to the tendency of some pollinators to visit certain flower species while bypassing others. It has been conjectured that constancy has evolutionary advantages for both the flower species and the pollinator species.

[1]Department of Engineering, University of Cambridge
[2]MISC Laboratory Constantine 2 University

> Pollination can be achieved by self-pollination or cross-pollination. Cross-pollination, or allogamy, means pollination can occur from pollen of a flower of a different plant, while self-pollination is the fertilisation of one flower, such as peach flowers, from pollen of the same flower or different flowers of the same plant, which often occurs when there is no reliable pollinator available.

So we seem to have 5 different concepts here, keep note of that. The authors then go on to say:

> Biotic, cross-pollination may occur at long distances, and the pollinators such as bees, bats, birds and flies can fly a long distance, thus they can be considered as the global pollination. In addition, bees and birds may behave as Lẽvy flight behaviour, with jump or fly distance steps obey a Lẽvy distribution. Furthermore, flower constancy can be used an incremental step using a similarity or difference of two flowers.

At this point the authors appear to have equated biotic pollination with cross-pollination, which we believe to be illogical, and poorly introduces their concept of "global pollination" without preamble. The introduction of a Lẽvy flights is worth investigating. Yang references [3] here in such a way as the reader is lead to believe that there is justification for modelling bird and bee behaviour as a Lẽvy flight, however, the paper referenced includes no mention of bird or bee behaviour. The last sentence appears to be missing an "as", though we couldn't say for sure. If that is the case, the author provides no reasoning for why flower constancy can be modelled as a similarity of two flowers, nor what an incremental step even is in the context of this problem.

1.5.2 Flower Pollination Algorithm

The authors of the second paper do not deviate at this stage from the original, and both introduce the actual algorithm by arguing that the above characteristics can be idealised as follow:

1. Biotic and cross-pollination is considered as global pollination process with pollen-carrying pollinators performing Lévy flights.
2. Abiotic and self-pollination are considered to be local pollination.
3. Flower constancy can be considered as the reproduction probability is proportional to the similarity of two flowers involved.
4. Local pollination and global pollination is controlled by a switch probability $p \in [0, 1]$. Due to the physical proximity and other factors, such as wind, local pollination can have a significant fraction p in the overall pollination activities.

For simplicity's sake, the authors assume that each plant only has one flower and each flower only produces one pollen gamete[2]. In reality the number of flowers per plant and pollen gametes per flower can vary widely between species and individual plants. This means that a solution, which we denote x_i, is equivalent to a flower and/or a pollen gamete[3]. The original author believes that extending the

[2] sperm cells

[3] There is no distinction between a flower and the pollen it produces, all that matters is how the pollen is used.

algorithm to multiple pollen gametes and multiple flowers (for multiobjective optimisation problems) should be easy.
The FPA is described by breaking it down into 2 key steps: global pollination and local pollination. For global pollination it is argued that the possibility of large travel distances for pollens and flower constancy give rise to the following step rule:

$$x_i^{t+1} = x_i^t + L(x_i^t - g_*) \tag{1.10}$$

where x_i^t is the pollen i or solution vector x_i at iteration t and g_* is the current best solution found amongst all solutions at the current iteration.

At this point we have to mention that the argument for global pollination being based on the current best solution, "the fittest flower", is non-existent - it just appears in the middle of talking about travel distances and constancy without any justification. Similarly, we have no idea what part of this function is due to flower constancy.

The parameter L is supposedly the "strength" of the pollination, which is a step size drawn from a Lẽvy distribution

$$L \ \frac{\lambda\gamma\left(\lambda\right)\sin\left(\pi\lambda/2\right)}{\pi}\frac{1}{s^{1+\lambda}},\ (s \gg s_0 > 0) \tag{1.11}$$

Here $\gamma(\lambda)$ is the standard gamma function, and this distribution is valid for large steps $s > 0$. Yang reports using $\lambda = 1.5$ for the simulations in the original paper, the second paper's authors (for GCP) do not report what value they used[4].
The local pollination, which also includes an allowance for constancy apparently, can be represented as

$$x_i^{t+1} = x_i^t + \epsilon(x_j^t - x_k^t), \tag{1.12}$$

where x_j^t and x_k^t are pollens from the different flowers of the same plant species and ϵ is drawn from a uniform distribution in $[0, 1]$.

Pay careful attention to that last bit: "different flowers of the same plant species". There is not a single mention anywhere else of "same plant species" and there is no consideration for such a thing in the provided pseudocode or associated descriptions and comments. There is such a lack of reference to this that we didn't even notice and just assumed that x_j^t and x_k^t are pollens from different flowers. Note that even that is nonsensical since supposedly pollens and flowers are the same thing. Already it would be fair to say that these algorithms and the papers that describe them are confused and confusing. Despite this, we continue.

Most flower pollination activities can occur at both local and global scale. In practice, adjacent flower patches or flowers in the not-so-far-away neighbourhood are more likely to be pollinated by local flower pollens than those far away. For this, the authors use a switch probability p to switch between common global pollination and intensive local pollination. (we do not know why the authors felt the need to add the "common" and "intensive" descriptors; they are never explained as having any particular meaning.) In the original paper (Yang) the author describes using an initial

[4] we will talk about why this is unhelpful later

$p = 0.5$ and then performing a parametric study which found that $p = 0.8$ worked best for most applications. Again, the authors of the second paper did not discuss actual values.

Moving into the realm of the GCP, the second authors describe what they call an "integer representation scheme"

> an individual is a complete assignment of k colors to the graph vertices such that $S = \{C(1), C(2), \ldots, C(i), \ldots, C(n)\}$ where $C(i)$ represents the color of the vertex i.

Basically, they assign colors (represented by integers $1, \ldots, k$) to a "flower" vector whose indices correspond to vertices on the graph. This is opposed to other schemes that we have investigated in this work that assign vertices to color "buckets".
Since this is a combinatorial optimisation problem, we also need a cost function. The authors call this a fitness function and define it as follows:

Let $A(G)$ be a $(0, 1)$ adjacency matrix of a graph $G = (V, E)$ where (a_{ij}) defined as follows:

$$a_{ij} = \begin{cases} 1 & \text{if}(v_i, v_j) \in E \\ 0 & otherwise \end{cases} \tag{1.13}$$

Let the conflicting matrix *conflict* of a coloring C be given by:

$$\text{conflict}_{ij} = \begin{cases} 1 & \text{if} C(i) = C(j) \text{and} a_{ij} = 1 \\ 0 & otherwise \end{cases} \tag{1.14}$$

For a solution S, the fitness function $f(S)$ is given by

$$f(S) = \sum_{i=1}^{n} \sum_{j=1}^{n} \text{conflict}_i j \tag{1.15}$$

The aim is then to to minimise the number of conflicts until reaching $f^*(S) = 0$, for a fixed k. Thus a valid coloring is found.

Note that this is essentially the same cost function as used by Johnson et al. in their fixed-K algorithm.
Finally the authors describe a "swap strategy" that they argue helps to keep diversity in the population and avoid stagnation by swapping the color of the most conflicting vertex in a solution S with the color of a least conflicting vertex. We will discuss the swap strategy in more detail in the implementation section.

1.5.3 Wrap-up

So what can we take away from this? Flower pollination has five concepts that provide a basis for a heuristic optimisation algorithm. Said algorithm has two movement mechanisms based on these five concepts: global pollination and local pollination. There is insufficient detail as to how the five concepts give rise to the two mechanisms. To put simply, local pollination is a move through the solution space that is constrained to the nearby neighbourhood which is defined, rather oddly, by a

(uniform) random perturbation of the other (at this iteration tick) solution vectors in the population (we call it a flowerbed). On the other hand, global pollination is a move through a solution space that is defined by perturbing the difference between the current solution vector and the current best solution vector by a Lévy step. Supposedly the potential for the Lévy step to be very large means that global pollination is a move that is able to escape local optima. This sounds all well and good, but the solution space is so ill-defined that in reality we have no idea where either of these moves are taking us.

In practise we noticed that local pollination (as we possibly incorrectly implemented it) tended to make the set of solution vectors become more and more homogeneous as we iterated. This is predictable behaviour given the action of the local pollination, but we're not sure *why* we would want to make the solution vectors homogeneous. Since the local pollination step doesn't depend on the *best* solution, this continuous move to homogeneity doesn't seem to serve the purpose of optimisation at all, rather, it inhibits that ability to explore the solution space - the exact opposite of what we want in a non-convex optimisation problem!

The link between the observed pollination properties and these steps as we've described them is so tenuous that one questions why the authors cite flower pollination as the inspiration for the algorithm.

Beyond the supposed pollination steps, the discrete FPA for GCP as described by the second authors has several other steps that are performed that are even less well justified. In particular, they describe a "swap strategy" that simply swaps the most conflicting vertex's assigned color with one of the least conflicting vertices' assigned color. The only justification for using this is that it "is frequently used when dealing with discrete problems". We have no doubt that this is the case, even though no reference is given, but it seems to have just been tacked on as way to improve the performance of the algorithm. In fact we suspect that the swap strategy might be the only reason the algorithm gets anywhere at all..

I hate flowers, I hate this paper and I hate these authors. What a god damned waste of time.

Chapter 2

Implementations & Original Contributions

2.1 Introduction

This section outlines specific implementations of some of the algorithms described above. Where possible pseudocode is provided. In most cases it is *not* the complete algorithm and further analysis of the original paper will be required to reconstruct the algorithm. Results for comparison of the algorithms is given in the Results section.

2.2 Random Brute Buckets

The Random Brute Buckets (RBB) algorithm presented here is a class of Successive Augmentation Technique. It is at its core an implementation of an insertion sort algorithm which samples vertices from the graph in a random order. This algorithm is guaranteed to find a valid colouring for graphs that are directed and undirected. It is however not bounded, and there is no guarantee that the algorithm will converge upon the best colouring.

2.2.1 Implementation and Problems

The RBB algorithm is presented in Algorithm 1 below.

Algorithm 1 Random Brute Buckets

```
1: procedure SOLVE(Graph g, long iterationLimit)                    ▷ g is predefined
2:     currentIteration ← 0
3:     currentBestColouring ← ∞
4:     while currentIteration < iterationLimit do
5:         Create empty list of buckets bList
6:         Populate vertex set V
7:         while V ≠ ∅ && |bList| < currentBestColouring do
8:             v ←random vertex from V                              ▷ Remove v from V
9:             vertexPlaced ← FALSE
10:            for all bucket ∈ bList do
11:                if bucket contains no conflicts with v then
12:                    add v to bucket
13:                    vertexPlaced ← TRUE
14:                    break for
15:                else continue
16:                end if
17:            end for
18:            if !vertexPlaced then                                ▷ create a place for the vertex
19:                create new bucket b_0
20:                add v to b_0
21:                add b_0 to bList
22:            end if
23:        end while
24:        if |b| < currentBestColouring then
25:            currentBestColouring = |bList|
26:        end if
27:        currentIteration++
28:    end while
29:    return currentBestColouring
30: end procedure
```

This algorithm always improves upon the best colouring (which is initially set to ∞ on line 3) in the first iteration, and upon subsequent iterations, if a better solution is stumbled upon randomly, that solution is set as the current best solution. The algorithm returns the best solution found after a set number of iterations.

2.2.2 Workarounds

This algorithm is very fast at finding initial solutions. This algorithm does not try to improve upon solutions that it finds, so local minima offer no resistance to finding the solution. This algorithm is not guaranteed to find the best solution. Indeed even after infinitely many attempts, there is no guarantee that the best solution will be found. That being said, the algorithm is generally able to find solutions close to the optimal colouring for most graphs extremely quickly and with a minimum use of resources (compared to other algorithms presented in this paper).

2.3 Gravitational Swarm Intelligence

The Gravitational Swarm Intelligence Algorithm presented in this paper is a modified version of the algorithm presented in the paper "Gravitational Swarm for Graph Coloring" by Israel Carlos Rebollo Ruiz [4]. Swarm Intelligence is a model where the emergent collective behavior is the outcome of a process of self-organization, where the agents evolve autonomously following a set of internal rules for their motion, interaction with the environment and the other agents. In this algorithm, the agents are subjected to an environment subject to a version of Newtonian Gravity. Centres of attraction (wells) are placed in the environment and the agents move based upon their attraction to these wells. Each well represents a distinct grouping or colouring of the underlaying graph. Agents experience a repulsive force if they try and enter a well already containing agents that prohibit a valid colouring. If an agent enters a well that is not prohibited, then it stops moving and takes on that well's colour. The algorithm ends once a set iteration limit is reached, or all agents settle into the wells, and hence a valid colouring is found.

2.3.1 Implementation and Problems

The action of each agent on every iteration is presented below:

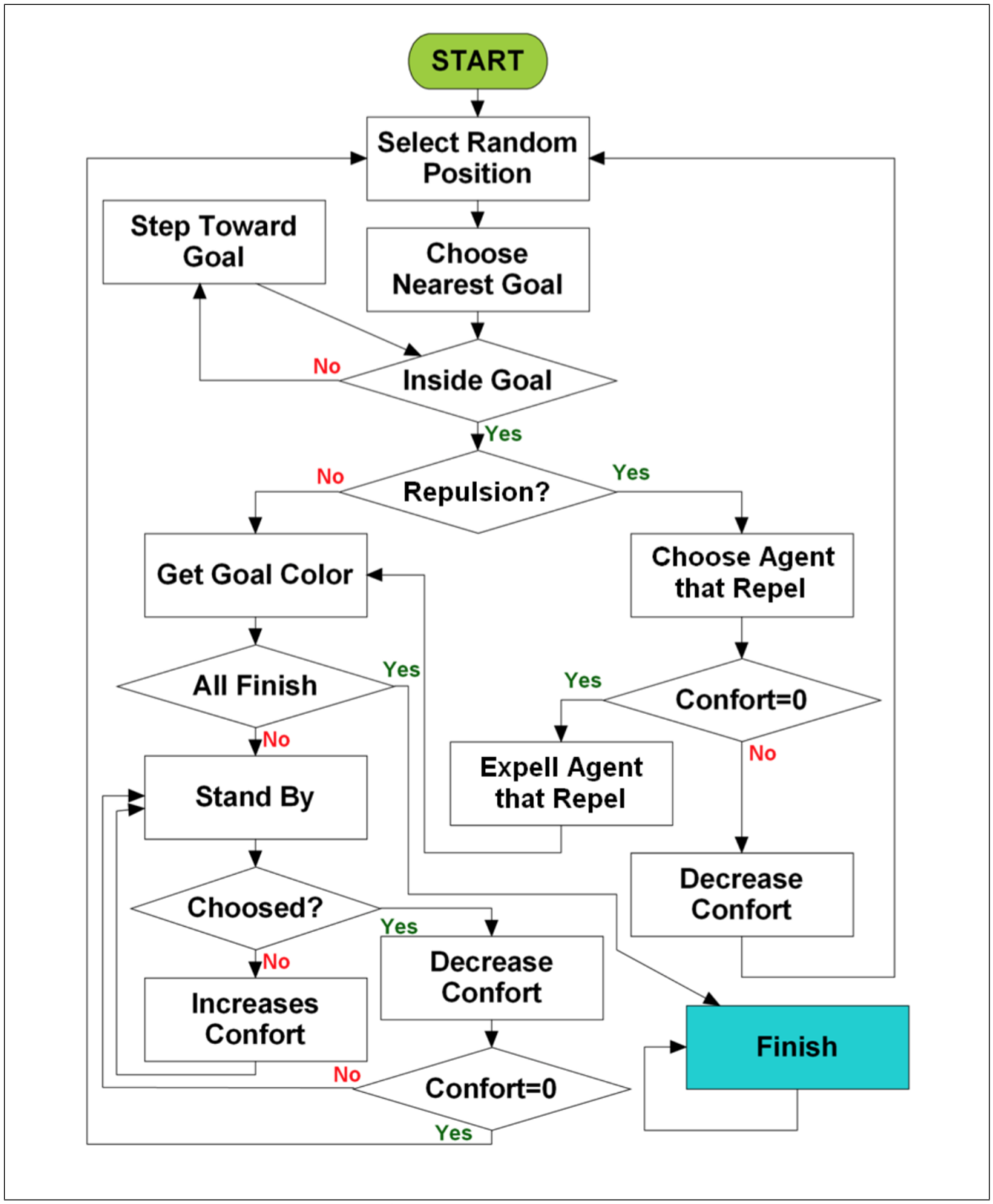

Figure 2.1: Agent logic for Gravitational Swarm

Each agent moves in the toric world until all agents fall into the "Stand By" State. Then once the last

agent triggers the "All Finished" State, the current colouring is accepted as valid. This algorithm presented some issues when implemented for testing. The comfort statistic that each agent tracks can grow unboundedly if the agents repeatedly fall into wells that they are subsequently repulsed from. This causes all captured agents to grow in comfort, making the local minuma harder to improve with every iteration and the chance of jumping out and finding a valid solution fall dramatically.
The distance functions that were implemented were found to contain an error, resulting in a toroidal universe but gravitational attraction only in the plane. This meant that all results presented here use the non-toroidal distance. This does not present an issue, as the algorithm is still shown to converge to the solution as long as the distance functions work partially.
The figure below shows the location of wells within the universe, with distance functions permitting a toroidal universe. The colouring is a result of starting an agent at every possible start location and logging which well captures it.

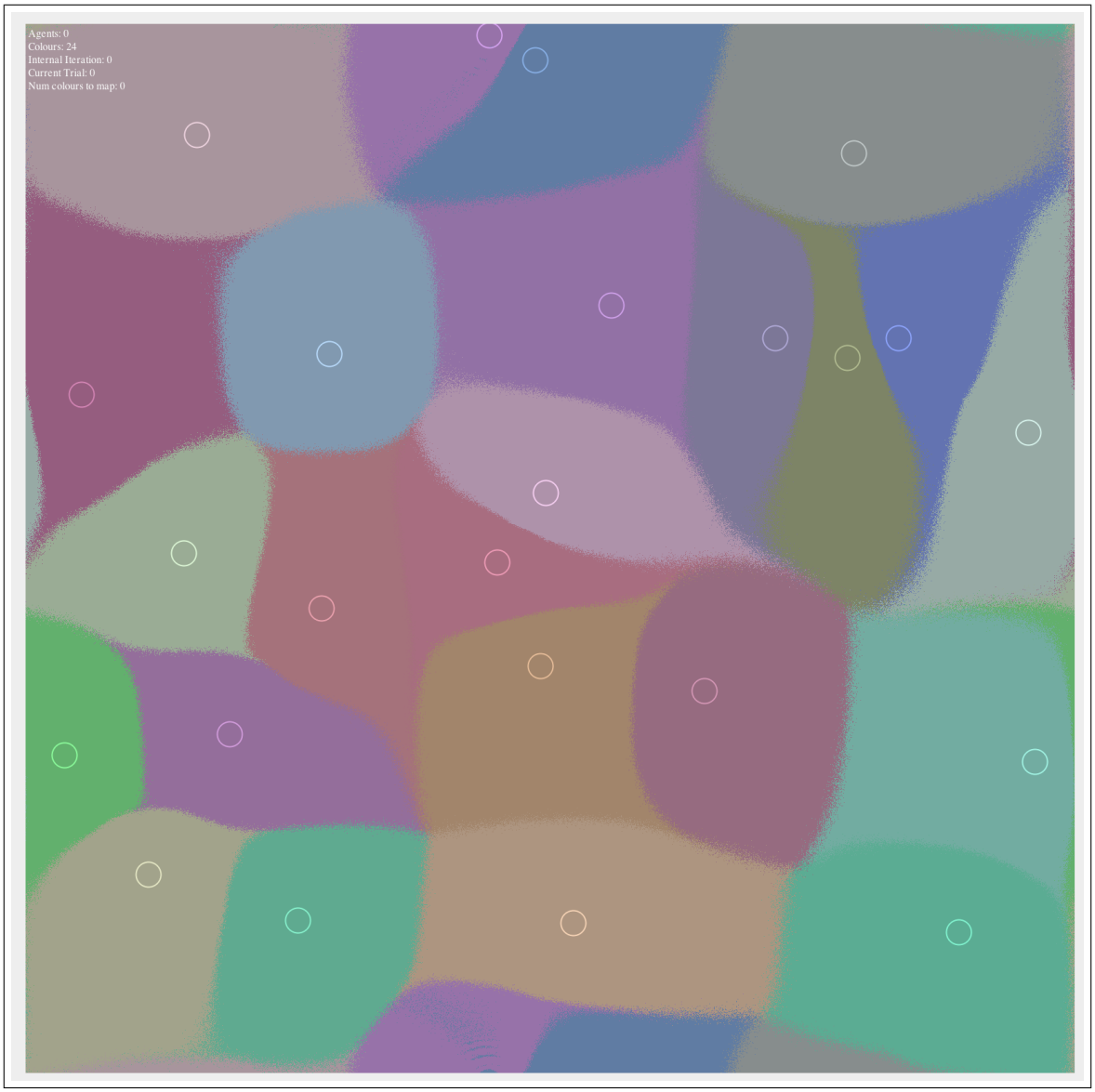

Figure 2.2: Gravity mapping with gravity well locations overlayed

2.3.2 Workarounds

This algorithm was relatively easy to implement, however much tweaking was required to ensure that the algorithm converges in a reasonable time and iteration count. The strength of the "gravity" had to be varied, and different radii of capture were compared, as these were the factors that caused the algorithm's runtime to be most affected.
This algorithm was implemented from the psuedocode provided in [4], however the sample code omitted some details of the algorithm and instead presented these details the "Results and Observations" section of the paper. This lead to some aspects of the algorithm not functioning as intended.

Interestingly, once the algorithm was tweaked to function correctly, the algorithm reflected the same observations that the original author had made.

2.3.3 Additions

Substantial time and effort has been committed to the algorithm, with many aspects and details changing and growing over time to better solve the problems presented. The algorithm was trialled with the option for gravity wells to be able to take some "mass" from captured agents and have their own velocity in the universe. This addition did not increase the convergence rate of the algorithm but it did expose some flaws with the distance functions used to allow agents to track the wells.
Agent velocity and direction calculations have been changed and tweaked numerous times in the attempt to have agents move in the toroidal universe and react correctly to having multiple sources of strong attraction. To this end, a function was created that mapped agent start locations to final end points. Any "inconsistencies" are easily visually identified. An example of inconsistent behaviour is shown here:

Figure 2.3: Toroidal Gravity Map with inconsistent behaviour - top left, top right exhibit "water-drop" pattern

The large amounts of variability around disputed boundaries is a result of the random noise added to the velocity on every iteration. It was found that if no noise was added, when there was very low gravity values (an agent was a long distance from *any* well), the agent would stall, taking many iterations to gain speed and be captured. The random noise helps to speed up the process, and can reduce the amount of iterations in a stall by "randomly walking" around the region until the gravity values become larger than the random noise.

2.4 Genetic Algorithm

The Genetic Algorithm presented here is a variant of the algorithm described by [2]. The algorithm was implemented as described, then tweaked to improve efficiency and to experiment with parameter tweaking and the effect of changing certain flows within the algorithm. These changes are discussed further in the Additions (2.4.3) section.

2.4.1 Implementation and Problems

The algorithm is implemented as follows:

Algorithm 2 Genetic Algorithm with Wisdom of Crowds

```
1:  procedure SOLVE(Graph g, iterationLimit, numChromosomes)        ▷ g is predefined
2:      currentAttempt ← 0
3:      currentBestColouring ← Δ(g) + 1
4:      aggregateChromosome ← chromosome of randomly assigned colours.
5:      while currentIteration < iterationLimit do
6:          population ← set of chromosomes with randomly assigned colours. (up to numColours)
7:          if solvePop() then
8:              aggregateChromosome ← chromosome of randomly assigned colours.
9:              currentAttempt ← 0
10:         else
11:             currentAttempt ← currentAttempt + 1
12:         end if
13:     end while
14:     return currentBestColouring
15: end procedure
```

Algorithm 3 Genetic Algorithm with Wisdom of Crowds - Tick Generation

```
1: procedure SOLVEPOP(Graph g, iterationLimit, numChromosomes)        ▷ g is predefined
2:     currentIteration ← 0
3:     while currentIteration < iterationLimit and best solution has cost > 0 do
4:         currentIteration ← currentIteration + 1
5:         if best chromosome has cost ≥ altMethodThreshold then
6:             parents ← getParentsA()
7:         else
8:             parents ← getParentsB()
9:         end if
10:        child ← crossOver(parents)
11:        if rand < mutChance then
12:            if best chromosome has cost ≥ altMethodThreshold then
13:                child ← mutateA()
14:            else
15:                child ← mutateB()
16:            end if
17:        end if
18:        add child to population
19:        remove bottom performing half of population
20:        repopulate up to numChromosomes
21:    end while
22:    if currentIteration ≥ iterationLimit then
23:        perform wisdomOfCrowds()          ▷ generate aggregateChromosome by voting
24:        add aggregateChromosome to population
25:    end if
26:    if best solution has cost 0 then
27:        currentBestSolution ← currentBestSolution − 1
28:        return true
29:    else
30:        return false
31:    end if
32: end procedure
```

Algorithm 4 Genetic Algorithm with Wisdom of Crowds - Parent Selection

```
1: procedure GETPARENTSA
2:     tempParents ← choose two random chromosomes from population.
3:     parent1 ← fitter of tempParents
4:     tempParents ← choose two random chromosomes from population.
5:     parent2 ← fitter of tempParents
6:     return parent1, parent2
7: end procedure
8:
1: procedure GETPARENTSB
2:     return top performing chromosome, top performing chromosome
3: end procedure
```

Algorithm 5 Genetic Algorithm with Wisdom of Crowds - Crossover

```
1: procedure crossOver
2:     child ← colours up to and including a random point from parent1, followed by the colours
       from parent2 from that point on in the chromosome.
3:     return child
4: end procedure
```

Algorithm 6 Genetic Algorithm with Wisdom of Crowds - Child Mutation

```
1: procedure MutateA
2:     for all vertex in chromosome do
3:         if vertex has a conflict then
4:             adjacentColours ← all adjacent colours to vertex
5:             validColours ← allColours − adjacentColours
6:             newColour ← random colour from validColours
7:             set chromosome colour at vertex to be newColour
8:         end if
9:     end for
10: end procedure
11:
1: procedure MutateB
2:     for all vertex in chromosome do
3:         if vertex has a conflict then
4:             newColour ← random colour from allColours
5:             set chromosome colour at vertex to be newColour
6:         end if
7:     end for
8: end procedure
```

Algorithm 7 Genetic Algorithm with Wisdom of Crowds - Wisdom Of Artificial Crowds

```
1: procedure WisdomOfCrowds
2:     expertChromosomes ← best half of final population
3:     aggregateChromosome ← best performing chromosome
4:     for all vertex dog
5:         if vertex is part of a bad edge then
6:             newColour ← the most used colour for vertex among expertChromosomes
7:             set colour at vertex of aggregateChromosome to be newColour
8:         end if
9:     end for
10: end procedure
```

2.4.2 Workarounds

The algorithm provides some parameters to tailor the solving method to be able to deal with local optima in several ways. Firstly there are 2 different parent selection methods that are used depending upon how close the best chromosome is to a valid colouring. Secondly there are two different mutation

methods that are used depending upon how close the best chromosome is to a valid colouring. It was found that while these methods allowed for some level of control over the algorithm, it was by no means as configurable as the other algorithms that were implemented. To remedy this and to produce our own variant of the Genetic Algorithm for solving the GCP, alterations were made to the flow to allow more control over the algorithm.

2.4.3 Additions

The first deviation to the documented algorithm was to the parent selection criterion. It was found that having the best solution be the only parent that is mutated when it is sufficiently close to a valid colouring slowed the convergence rate of the other chromosomes to zero. To remedy this, we implemented a switch such that a certain percent of the time, the parents chosen were the best chromosome and itself. But a small percentage of the time, the parents would be the best chromosome and a random chromosome from the population. This prevented the stagnation of the problem, and allowed for other changes to be made to improve the algorithm. The second part of the algorithm identified for change was the crossover method. This method originally selected the first portion of $parent1$ and mixed it with the complement of that section of $parent2$. This process was changed to a simple 50/50, such that the chance of the first section coming from $parent1$ was equally weighted to the first section coming from $parent2$. This prevented possible bad colourings from being propagated simply because the occurred early in the chromosome. The final change to the algorithm was in response to results collected from the algorithm. It was found the wisdom of crowds voting mechanism was finding the solution far more successfully than the Genetic Algorithm itself. This prompted a change to the algorithm whereby a vote could not take place before certain conditions were met in the agreement of the chromosomes. This lead to a much higher runtime for the algorithm, but once a vote could commence, the algorithm was much more likely to either improve the colouring or produce a chromosome with many fewer conflicts. This method was settled upon after several other methods were trialed and discarded. As the algorithm searches for a tighter colouring, it takes many more generations for the chromosomes to converge. This method was settled upon since it allowed the algorithm to be applied to any graph with a minimum of tweaking, yet a valid vote will always take place, since the algorithm will always reach a point of agreement, even if the solution is not valid for the given graph. Setting the level of agreement that each chromosome must exhibit allows the algorithm to be sped up or slowed down. If the limit is relaxed, each trial takes less time to complete, yet the wisdom of crowds vote has a lower chance of improving the solution. Tightening the bounds has the effect of dramatically increasing the runtime of the algorithm but increases the chance that the wisdom of crowds vote will yield a better solution. Setting the bound too tight however will result in the algorithm running for an infinite amount of time (if the level of agreement specified is not possible). The last change that was made to this algorithm was to create a variable parameter for how much of the population was discarded and repopulated at random on each generation. This parameter is responsible for setting how much of the solution space is tried. Each iteration, the random nature of the generation process has the possibility to generate a better solution to the problem. However, the larger the proportion of the population that is discarded upon each iteration, the lower probability that the wisdom of crowds vote that occurs at the end of each trial will improve upon the solution. For this reason, the proportion of the population that is allowed to participate in the vote is set to always be less than (or equal to) the proportion that is kept in each iteration, so that random noise is not added to the voted data when a vote takes place.

2.5 Flower Pollination

2.5.1 Implementation and Problems

Ok so I have to say that I think that this paper[1] SUCKS. I was drawn to it initially because it was a new (2015) optimisation algorithm and we wanted something to compare to the gravitational swarm intelligence[4], plus it was flowers which is cute. Anyway, the first warning sign should have been that it's from Algeria, but that would have been racist. Second warning sign: THEY WORD FOR WORD COPIED from Yang's paper. Like, full blown plagiarism. Damn, how does that pass peer review??? Of course, I hadn't actually read Yang's paper at the start so I didn't know that.. Third warning sign: The algorithm doesn't quite match up with the pollination properties it supposedly takes inspiration from. Ok, not just these guys fault, Yang does the same thing. So I tried to implement it. And I failed. Some parts of the pseudocode I just didn't know what they were talking about, and when I managed to work around that to hobble something together that ran, it just didn't do anything. So I hacked and hacked and made something that started to work! But it was still rubbish.
The original algorithm is as follows

Algorithm 8 FPA for GCP

procedure SOLVE(Graph g, *iterationLimit*, *numFlowers*) ▷ g is predefined
  Initialize a population of N flowers/pollen gametes with random solutions
  Find the best solution g_* in the initial population
  Define a switch probability $p \in [0, 1]$
  $t = 0$
  while $t <$ *iterationLimit* **do**
    for $i = 1 : N$ **do**
      if $rand < p$ **then** ▷ $rand \backsim$ Unif(0,1)
        Draw a (d-dimensional) step vector L which obeys a Lèvy distribution.
        Global pollination via $x_i^{t+1} = x_i^t + L(x_i^t - g_*)$
      else
        Draw U from a uniform distribution $[0, 1]$
        Do local pollination via $x_i^{t+1} = x_i^t + \epsilon(x_j^t - x_k^t)$
      end if
      //Discretisation and correction step
      Apply round function to obtain integer values
      Correction Step() ▷ To ensure that no invalid colors are used.
      if all k colors are assigned **then**
        Swap the most conflicting node with a random vertex of the lowest used color
        Evaluate new solutions x^{t+1} using global pollination
        if $f(x^{t+1}) < f(x^t)$ **then** ▷ If new solutions are better
          $x^t = x^{t+1}$
        end if
      elsecontinue
      end if
    end for
    Update the current best solution g_*
    if $f^* == 0$ **then** ▷ Legal coloring found
      $k = k - 1$
      Break
    end if
    $t = t + 1$
  end while
  Output the best solution found
end procedure

The Correction Step() is also described, but it's so simple I'm not sure why they bothered. If a color class is assigned that is greater than k instead set it to k, if a color class is assigned that is less than zero instead set it to zero. Note that the **Break** when a legal coloring for a given k is found should probably be a **Return** since it is implied (poorly) that this solve procedure exists within a larger framework for descending k.

2.5.2 The modified FPA for GCP

Since the paper does such a poor job of describing the process, I was unable to implement the algorithm in such a way that it was capable of finding even a $k = \Delta$ coloring in an arbitrary *reasonable* time (couple of hours). Frustrated, I made some rather drastic changes without much

justification. I will now describe the final algorithm that *sometimes* beats the genetic algorithm (see results).
First let's look at the algorithm

Algorithm 9 modified FPA for GCP

```
1: procedure SOLVE(Graph g, iterationLimit, numFlowers)                ▷ g is predefined
2:     Initialize the flowerbed
3:     Update the current best solution g_*
4:     Define a switch probability p ∈ [0, 1]
5:     t = 0
6:     while t < iterationLimit do
7:         for flowerNum = 0; flowerNum < numColours; flowerNum++ do
8:             fixColors(flowerNum)
9:             if rand < p then                                        ▷ rand ∽ Unif(0,1)
10:                Draw a (d-dimensional) step vector L which obeys a Lèvy distribution.
11:                Global pollination via x_i^{t+1} = x_i^t + L(x_i^t − g_*)
12:            else
13:                Draw U from a uniform distribution [0, 1]
14:                Do local pollination via x_i^{t+1} = x_i^t + ε(x_j^t − x_k^t)
15:            end if
16:            fixColors(flowerNum)
17:            Apply round function to obtain integer values
18:            Correction Step                  ▷ To ensure that no invalid colors are used.
19:            for col = 0 : α do               ▷ α could be anything, currently set to k
20:                swap(flowerNum)
21:            end for
22:        end for
23:        Update the current best solution g_*
24:        if f* == 0 then
25:            k = k − 1
26:            Return
27:        end if
28:        doNaturalSelection()
29:        t = t + 1
30:    end while
31:    Output the best solution found
32: end procedure
```

Ok so as mentioned before, we call the set of flowers the *flowerbed*. The main differences are as follows:
fixColors() In the original algorithm, each flower is checked to see if every k color is assigned. Only if that returns true is the swap strategy and global pollination perturbation performed. I found this to be odd, but rather than remove it I went the other direction and wrote a function that ensured that all colors are present. Simply put, fixColor searches the flower for each color in turn and if a color is not present it assigns that color to a random vertex.
fixColors is run before and after pollination to ensure that every flower adheres to the current k.
Rather than performing swap once, under the suspicion that swap is the main mover in the algorithm,

I perform swap an arbitrary α number of times. For the purpose of experimentation, of which we report the results on later, $\alpha = k$ without justification. Swap is directly reducing the cost and so moves the solution vector very quickly through the solution space towards a, presumably, local optima.

doNaturalSelection() Despite regularly referring to, and comparing against, the genetic algorithm, the authors chose not to implement natural selection or wisdom of crowds. I didn't have the motivation to work out how wisdom of crowds would work for the flowerbed, however implementing natural selection was incredibly straightforward. The cost of each flower in the flowerbed is assessed and the median value calculate. Every flower with a cost higher than the median value is replaced with a randomly generated flower. Doing this is a response to the tending towards homogeneity that we've remarked upon before. Every iteration the available genetic material gets a massive shake-up, providing potentially new color partitions that might lead to a legal coloring.

That's pretty much it for flower pollination.

Chapter 3

Results & Discussion

The algorithms were run in an unmanaged Ubuntu 14.04 cloud server over the course of several weeks. It was found that the algorithms would (somewhat predictably) grow in size as the algorithm narrowed the solution space. This meant that the algorithm's runtime would grow very long. To prevent unreasonably long runtimes, the algorithms were bounded at 6 hours total runtime per graph. Each algorithm was set to test up to 20 times over per graph in the allotted period. The meta-heuristic algorithms are started at their known upper bound, the maximal degree + 1, shown here as $\Delta + 1$, and the number of colours is successively reduced, until the algorithm fails to find a valid colouring. The successive augmentation algorithm (RBB) does not have a set start position, since it simply tries to find a better valid colouring upon every iteration. Its initial best colouring uses *infinite* colours, and therefore must be lowered upon the first iteration.

3.1 Algorithms

The algorithm are run on 5 graphs of varying difficulty. Sample graphs[1] were run successively for each algorithm up to 20 times, the average time to the given colouring, and the success rate for each algorithm to attain that colouring is shown in Table 3.1. The total runtimes for each algorithm are shown in Table 3.2, signifying the amount of time that the algorithm required to not find a better solution. Note that due to the hard limitation of 6 hours (21600 seconds) on each algorithm, better colourings may have been attained had the algorithm been allowed to run for a longer amount of time.

[1] provided for the 1992-1992 DIMACS Implementation Challange for NP-Hard Problems: Max Clique, Graph Coloring, and SAT.

Graph	χ	$\Delta + 1$	Solver	k	avg time to k (sec)	% success
myciel4.col	5	12	RBB	5	0	100
myciel4.col	5	12	GS	5	0.05	100
myciel4.col	5	12	FPA	5	0.9	100
myciel4.col	5	12	GA	5	24.7	100
queen5_5.col	5	17	RBB	5	0.001	100
queen5_5.col	5	17	GS	5	1.27	100
queen5_5.col	5	17	FPA	9	113.4	80
queen5_5.col	5	17	FPA	10	36.2	100
queen5_5.col	5	17	GA	5	3984.5	12
queen5_5.col	5	17	GA	6	218.4	63
myciel5.col	6	24	RBB	6	0.001	100
myciel5.col	6	24	GS	6	0.8	100
myciel5.col	6	24	FPA	8	1918.5	13
myciel5.col	6	24	FPA	8	464.4	93
myciel5.col	6	24	GA	6	12814.1	100
queen7_7.col	7	25	RBB	7	0.7	5
queen7_7.col	7	25	RBB	8	0.5	75
queen7_7.col	7	25	GS	9	678.5	5
queen7_7.col	7	25	GS	19	42.2	100
queen7_7.col	7	25	FPA	20	2483.2	60
queen7_7.col	7	25	FPA	21	887.1	100
queen7_7.col	7	25	GA	14	5089.4	100
mulsol.i.3.col	31	158	RBB	31	0.001	100
mulsol.i.3.col	31	158	GS	31	5397	10
mulsol.i.3.col	31	158	GS	32	2815.5	50
mulsol.i.3.col	31	158	GS	33	1944.5	100
mulsol.i.3.col	31	158	FPA	81	21083.8	50
mulsol.i.3.col	31	158	FPA	82	3585.5	100
mulsol.i.3.col	31	158	GA	82	19960.3	100

Table 3.1: Results showing the time taken for the algorithm to find a valid colouring (k colours), optimal or not

Graph	χ	$\Delta + 1$	Solver	min k reached	total runtime (sec)
myciel4.col	5	12	RBB	5	0.47
myciel4.col	5	12	GS	5	161.9
myciel4.col	5	12	FPA	5	225.7
myciel4.col	5	12	GA	5	173.5
queen5_5.col	5	17	RBB	5	0.3
queen5_5.col	5	17	GS	5	388.5
queen5_5.col	5	17	FPA	9	496.6
queen5_5.col	5	17	GA	5	21600*
myciel5.col	6	24	RBB	6	1.3
myciel5.col	6	24	GS	6	512.6
myciel5.col	6	24	FPA	8	2929.4
myciel5.col	6	24	GA	8	21600*
queen7_7.col	7	25	RBB	7	1.1
queen7_7.col	7	25	GS	9	1795.3
queen7_7.col	7	25	FPA	20	4806.5
queen7_7.col	7	25	GA	15	21600*
mulsol.i.3.col	31	158	RBB	31	19.05
mulsol.i.3.col	31	158	GS	31	7972.9
mulsol.i.3.col	31	158	FPA	81	21600*
mulsol.i.3.col	31	158	GA	82	21600*

Table 3.2: Results of algorithms attempts at finding a better solution (and failing)

It is interesting to note that the successive augmentation algorithm RBB does not exhibit a blowup in time when it gets close to a solution, indeed it actually *speeds up* as it narrows its scope. It exhibits a polynomial dependence on the number of vertices a graph contains. However it is not guaranteed to converge to the optimal solution, even after infinitely many iterations. The meta-heuristic algorithms are seen to get dramatically slower with the addition of more vertices, and are also affected by the narrowing of the search space. The FPA and GA were especially affected by the narrowing of the search space, each hitting the time limit upon their iterations well before reaching the optimal solution. The GS does not suffer from this problem to such an extreme degree, but the time taken does increase dramatically as the solution space narrows.
Each solver was seen to struggle when faced with graphs with a complex geometry. For these graphs, like the hypercube-class graphs which are bipartite by design, all algorithms were seen to struggle obtaining a solution due to the highly ordered structure of the underlying problem.
Note that the runtimes marked (*) were terminated at maximal runtime, and a better solution may have been found had the algorithm been allowed to continue.

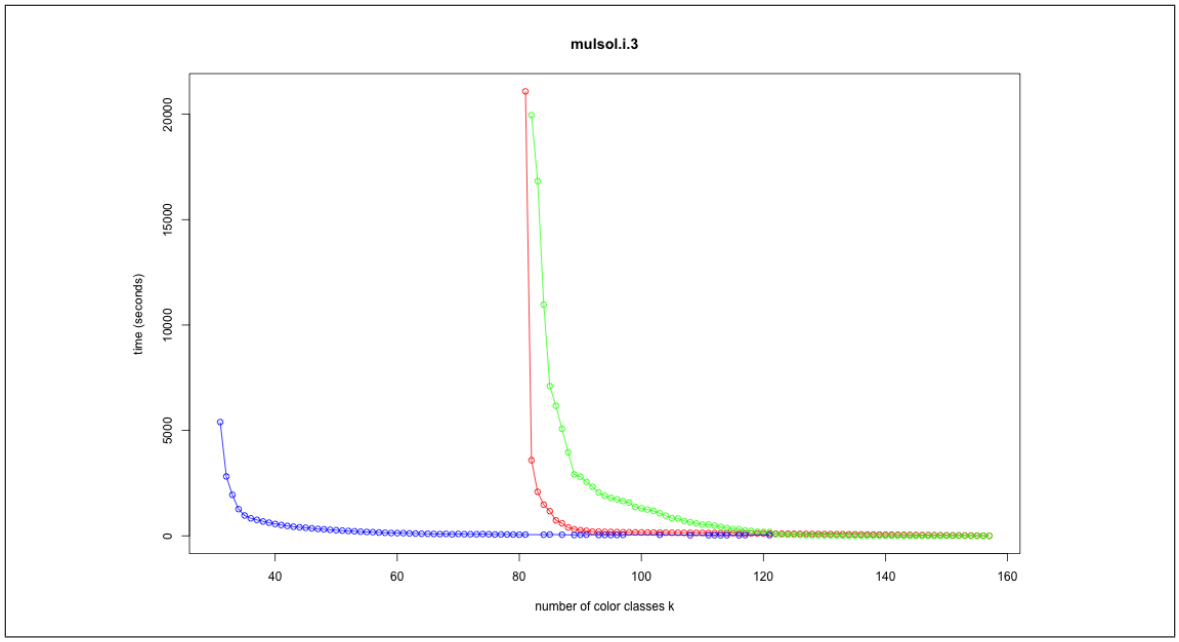

Figure 3.1: Asymptotic runtime for graph mulsol.i.3. Blue: GS, Red: FPA, Green: GA

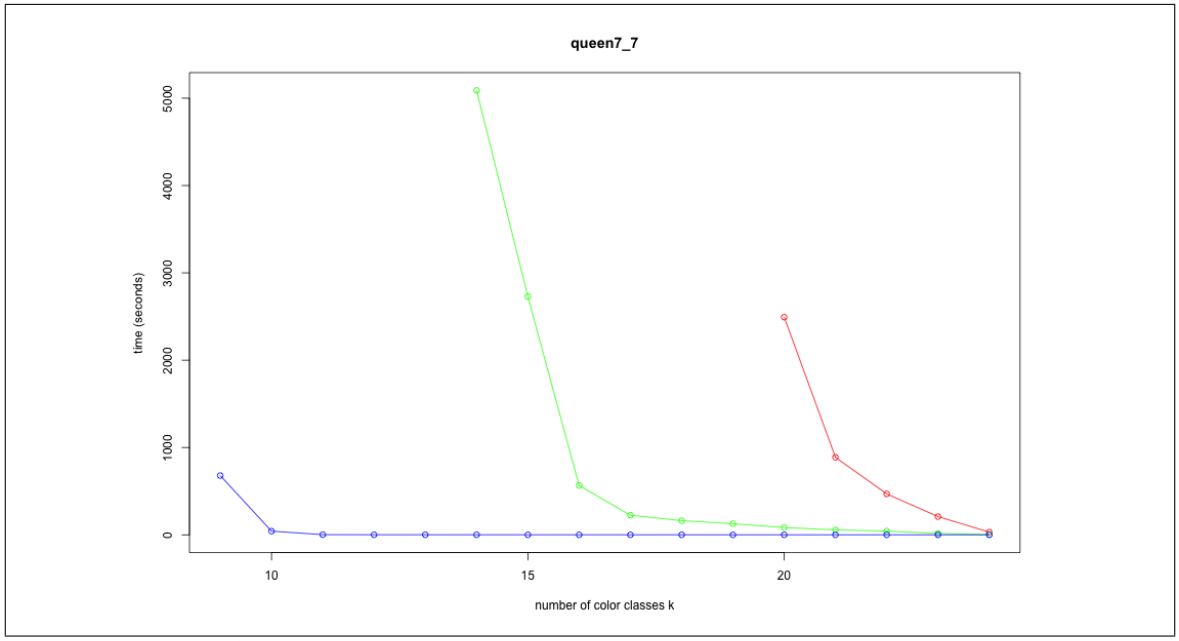

Figure 3.2: Asymptotic runtime for graph mulsol.i.3. Blue: GS, Red: FPA, Green: GA

So pretty!

Bibliography

[1] M. Bensouyad and D. Saidouni. A discrete flower pollination algorithm for graph coloring problem. In *Cybernetics (CYBCONF), 2015 IEEE 2nd International Conference on*, pages 151–155, June 2015.

[2] Musa M Hindi and Roman V Yampolskiy. Genetic algorithm applied to the graph coloring problem. In *Proc. 23rd Midwest Artificial Intelligence and Cognitive Science Conf*, pages 61–66. Citeseer, 2012.

[3] Ilya Pavlyukevich. Lévy flights, non-local search and simulated annealing. *Journal of Computational Physics*, 226(2):1830–1844, 2007.

[4] Israel Carlos Rebollo Ruiz. *Gravitational Swarm for Graph Coloring*. PhD thesis, The University of the Basque Country, 2012.

[5] Xin-She Yang. Flower pollination algorithm for global optimization. In *Unconventional computation and natural computation*, pages 240–249. Springer, 2012.