Project Definition Document

Liam Holland - 21386331

2024-2025

Contents

List of Figures

1 The Problem

In this section, I will give a general overview of the problem, including its background in the Graph Colouring Problem.

1.1 A Graph?

For the rest of this document, when I refer to a "graph", I mean a set of vertices (or nodes) V connected together by some set of edges E. For this project, I will almost always be dealing with an undirected graph, meaning that an edge connecting two nodes can be traversed in either direction. An example of a graph can be seen in (1). Graphs are a commonly used data structure in modelling complex problems in computer science.

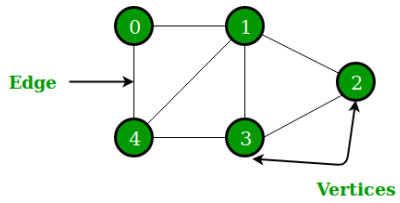


Figure 1: Example Graph

1.2 The Graph Colouring Problem

The Graph Colouring Problem (GCP) is a well-known problem in maths and computer science. It goes all the way back to at least 1852 when Francis Guthrie posed the four-colour conjecture when attempting to colour a map such that no neighbouring regions had the same colour [1].

Formally, it can be defined as:

Given a graph $G = (V, E)$, the GCP involves assigning each vertex $v \in V$ an integer $c(v) \in \{1, 2, \ldots, k\}$ such that

- $c(u) \neq c(v) \forall \{u, v\} \in E$, and
- k is minimal [2]

In other words, the goal is to assign a colour to each node so that you find a solution where every node has a colour, none of its neighbours have been assigned the same colour and you have used the minimum number of colours possible. This is called an optimal colouring. The number of colours used in an optimal colouring is called the *chromatic colour* of the graph, and is denoted by $\chi(G)$.

If you colour the graph and find you have two nodes which have been assigned the same colour, the colouring is *improper*. The case of two connected nodes being assigned the same colour can be called a clash or a conflict.

For this project, it is important to note that colours are represented as a set of numbers. As such, a vertex being assigned a "colour" will mean it has been assigned a number $i \in \{1, \ldots, k\}$ where k is the maximum possible colour. Representing colours as numbers makes implementation far simpler, but also means we can apply laws easier as well. For example, we know that the maximum number of colours possible is the number of nodes, i.e. the graph is connected so each node must have a unique colour. In most other graphs, we know that the maximum colour will be at most the largest degree in the graph.

The GCP can also be viewed as a partitioning problem, where a solution S is represented by a set of k colour classes. This kind of representation can be a useful way to think about the problem when looking at more probabilistic approaches.

1.3 NP-Hardness

A problem is in the set NP (non-deterministic polynomial) if there is no solution with a complexity in polynomial time. The GCP is known to be in the set NP and to be NP-complete. This means that every problem in NP can be reduced to a GCP in polynomial time. Take, for example, the common problem used to prove a problem is NP-complete, the Boolean Satisfiability problem (SAT). This problem attempts to find if there is an assignment of true and false to every variable in a Boolean expression such that the overall expression evaluates to true. If so, it is classified

as satisfiable. Immediately, we can see the similarities to finding a proper colouring of a graph. To represent this problem with graphs, we can use what are referred to as "gadget graphs" to represent Boolean operations, such as the OR-gadget in (2).

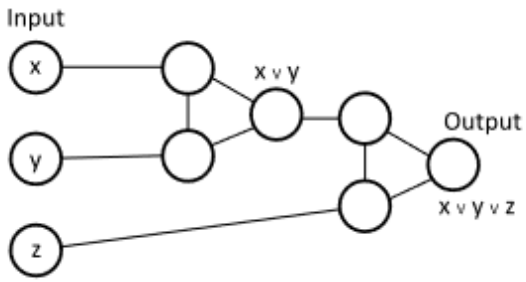


Figure 2: OR-gadget graph [2]

I believe that the GCP being an NP-complete problem makes it inherently interesting, but NP problems are also interesting since finding efficient ways to tackle these complex problems can often take you down unexpected routes of thinking.

1.4 Applications

So, if the problem is so hard, why have people spent so long trying to find a solution? The answer is that there are many useful applications of the Graph Colouring Problem. As mentioned above in section 1.2, map colouring is one such application. Others include scheduling, solving sudoku, channel frequency assignment, and various applications in scientific fields.

Scheduling in particular is a common use of the GCP. Finding a timetable that works for every class, student, teacher and room is not a simple task when applied to something on the scale of a university. However, this is not the only scheduling that we can apply the problem to. It is also used for scheduling register allocation in compilers. A compiler can use GCP algorithms to find the minimum number of CPU registers it will need at any given time when running a program [3] [4].

In science, particularly computer science, the GCP can be used to tackle a diverse set of problems: Data mining, image segmentation, clustering, image capturing and networking, to name a few [5].

Hopefully this section has illustrated how the GCP is useful, due to the way we can use it to model and solve a vast range of problems. The more efficient our algorithms, the more complex problems we can tackle in a reasonable amount of time.

1.5 Centralised Approaches

Many of the approaches to finding an optimal colouring of a Graph G are what are referred to as *centralised* approaches. What this means is that you have knowledge of the entire graph when you are going about finding a solution for colouring it. One of the most common algorithms is the backtracking algorithm.

The backtracking algorithm attempts to apply the minimum possible colour to each node, given a maximum possible colour m. If it is found that a no colour in the set of colours $\{1, \ldots, m\}$, we backtrack and increase m. Implementations of this algorithm are often recursive when written in code. [3]

However, this approach is still very slow for large graphs. This and many other algorithms usually come with an exponential time complexity, which becomes prohibitively slow for large problems, even on modern hardware [6]. As such, there have been attempts to look at more heuristic-based algorithms which will find a solution that is good, or close to the optimal solution, in a reasonable amount of time. These include Greedy algorithms such as DSATUR [7] and RLF [8], along with other local search methods (see section 1.6).

1.6 Decentralised Approaches

The idea of a decentralised approach can vary greatly. In essence, a decentralised approach can be taken to be one where each vertex works with limited knowledge in order to find a solution to the

GCP for their locality in the graph. These local search algorithms can take many different forms. A common one is tabu search [6], a version of the hill climb algorithm which implements a "tabu list" that prevents the program from returning to previous solutions, to avoid local maxima in the solution space. The idea is based around iteratively adjusting the solution you have to the GCP in order to continually move towards an optimal colouring, or something close to it. A lot of these are based in the realm of genetic and evolutionary algorithms. I will discuss how this works in a practical sense in section 4.

Another approach that has picked up steam in recent years are ideas of Ant-Colony Optimisation and similar algorithms. These algorithms are fascinating, and take a great deal of inspiration from the way that nature has solved similar problems in real life. The idea is usually the same: use each vertex to move further and further towards an ideal solution. Ant-based algorithms tend to do this via the idea of a trail element and a greedy element. Essentially, this boils down to a probabilistic approach where each "ant" colours its vertex with the most likely best colour at a given iteration. [9] [10]

An important note is that heuristic-based algorithms like these simply end after a certain amount of time; when there are no changes for a certain number of iterations, for example. Our job is to find a means to reduce the amount of time they take to converge on a good solution. What this means is that decentralised approaches are often slower than centralised ones.

The benefits of using a decentralised approaches are greater than simply improved efficiency, however. They happen to do a much better job of representing real-world problems. For example, imagine we run a perfect algorithm which finds a solution to a scheduling problem in a university. What if what the algorithm says is the best solution does not work in a practical sense? Say the lecturer for a certain class needs to leave early on a particular day and is able to arrange a swap with a colleague. In a centralised approach, changing constraints like this means rerunning the algorithm, but decentralised approaches should allow us to change constraints whenever we want, with minimal impact on efficiency or to the unchanged portion of the solution.

Additionally, these approaches are ideal to be translated into a multithreading approach. It would be very straight forward to have a number of worker threads, which can be assigned a new node each time they are finished with their current one. Given the iterative approach of decentralised approaches, we can expect that the problems which often arise in parallel processing, such as race conditions, would have very limited impact in this context.

1.7 Problem Definition

Given the information in the preceding sections, I will define my final year project as follows:

To investigate the impact of changing constraints on a graph using decentralised approaches

In this project, I hope to find results which can show how we can modify constraints on the graph while attempting to find a solution, while investigating some novel approaches to implementing said approaches.

2 The Research Thus Far

In this section, I will give some insight into the results I have collected so far.

The logbook and notes I have on the project up to this point are available in the `fyp_logbook` folder in the GitHub repository.

2.1 Types of Graphs

There are a great number of graphs that could be tested in this project. However, a lot of these are not too interesting from a standpoint of finding optimal solutions. For example:

Ring Graph

A ring graph is a graph in which every node has a degree 2, and is connected in a ring formation (3).

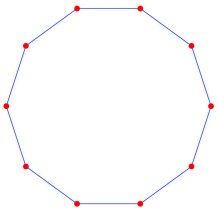


Figure 3: Example of a Ring Graph

The chromatic colour of a ring graph will always be 2 if there are an even number of nodes and will always be 3 if there is an odd number of nodes. As such, there is little reason in spending much time trying to find optimal solutions.

However, I did find something interesting to look out for in other types of graphs when running a random colouring algorithm on it (see section 4). Essentially, the algorithm might successfully return the chromatic colour of 3 on an odd numbered graph, but rather than being an optimal colouring where only one node has the colour 3, the colours are randomly distributed across the nodes.

Bipartite Graph

Again, this graph could be used for testing algorithms, but we already know that the chromatic colour will always be 2. The algorithms generally perform fine on this type of graph, as there is only every one other option if a conflict needs to be fixed. I implemented it in the program I used as more of a test of the code implementation than a means to find interesting results (see section 4).

Random Graphs

Random graphs are where I will be spending most of my time for this project. Randomly generated graphs are the ideal testing space because we never know exactly what the topology of the graph will look like in advance of running the program. In order to generate the graphs, I made use of the Erdős–Rényi model [11]. This model describes a random graph as a graph of v vertices where each edge in the would-be fully connected graph has p probability of existing. This model allows us to have a well defined means of generating graphs of varying density. As mentioned in section 1.1, I have used only undirected graphs thus far.

2.2 Random Colouring

When implementing a heuristic-based approach, then a random colouring is a nice, easy way to start. I began with a random algorithm which simply applied a random colour to each node until it managed to find a *feasible* colouring, then stop. This is not a particularly accurate or useful approach. Nevertheless, it helped give me a sense of the problem and allowed me to set up the code representation of a graph (see section 4).

You can see in (4) how the algorithm struggled more to even find a feasible solution if it was not given a large number of iterations to work over (number of iterations defined by `-m`).

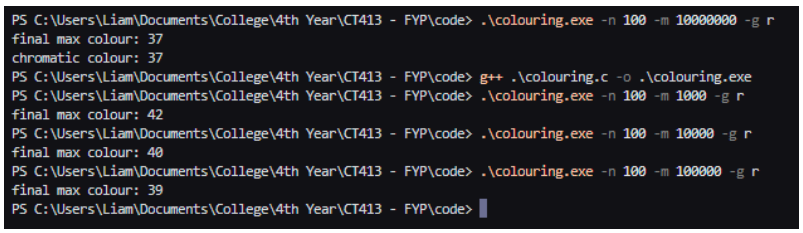


Figure 4: Results of initial random approach

2.3 What is an Agent?

In the following sections, there are going to be a lot of references to what I call "agents". These are essentially the same as an "ant" or "drop" or "mole" in other approaches, I just chose the most generic name for the generalised approach I was adopting. An agent can be thought of as an active node. If a node is active, it can change its colour based on some level of local information. The amount of information it has is down to each implementation (see section 2.4). An agent can also move, if that is relevant to a certain implementation (see section 2.7).

The important thing to remember is that the agents are not an implemented *thing* within the code. It is an abstract means of selecting a node to colour. This is discussed in greater detail in section 4.

2.4 The Limits of Local Information

The way agents behave will depend on the amount of information they have. The amount of information an agent has also determines how decentralised it is. For example, if you have an agent check the colour of each of its neighbours, and nothing else, that is entirely local information. On the other hand, if you give agents information about the most common colour being applied then that is information that agents can access which is global, so in that case you are moving closer to a centralised approach.

For the most part, I am keeping the implementations as decentralised as possible. One of the goals of the project is to implement agent algorithms with incrementally more global knowledge to compare how they perform in comparison to one another, but I want to stick to the decentralised aspect of the project as much as possible.

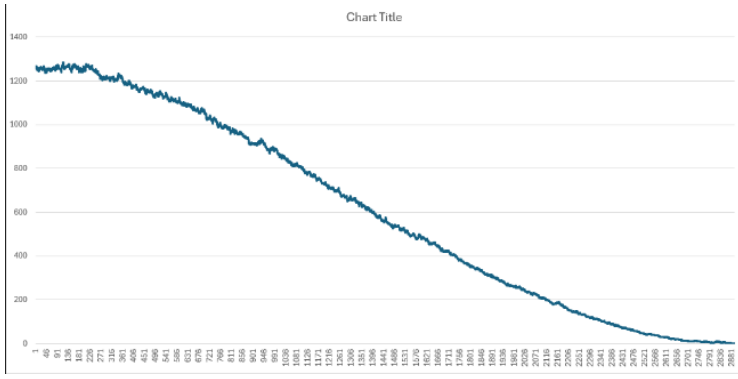
One of the challenges of this project has been and will continue to be to define what is and is not a decentralised approach. For example, is it a decentralised approach if we limit the number of colours used in the process to the maximum degree in the graph, given that we would have to have knowledge of the entire graph to obtain that information? Is the degree of neighbour nodes part of the local information a node should be able to access? These are the kind of questions that can, should and will be explored here.

2.5 Colour-blind Algorithm

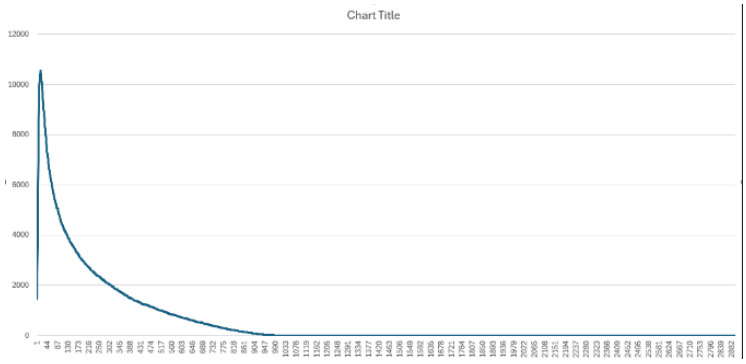
One of the first I tested was an agent with as little local knowledge as possible. All the agent knows is the colour of the node it is on and whether it is conflicting, which limits the amount it can do in order to make an informed change. It does not know the colour of its neighbour nodes (hence the colour-blindness) This agent colours the graph based on a few conditions:

- If the node does not have a colour, apply its degree as the colour
- If the node is **not** in conflict, do nothing
- If the node is in conflict, increment/decrement its colour by 1 (mod the degree of the node)

With these rules, I expected to quickly reduce the number of conflicts in the graph, but also did not have high hopes for it finding an optimal colouring. The changes it makes are not a far cry from essentially picking a random colour, as the incremented/decremented colour could still have a conflict with any of the neighbour nodes. The number of conflicts over time are graphed in (5).



(a) Decrementing the colour



(b) Incrementing the colour

Figure 5: Colour-blind Algorithm Results

It is interesting to see how the approaches, which differ only slightly, produce very different results. In (5a), we can see how decrementing the colour means it takes far longer to reduce the number of conflicts, as if everything is conflicting, they would then all be reduced by one. On the other hand, (5b) shows how incrementing the colour is far more efficient. Initially, the number of conflicts spikes sharply, as since every node has its degree as its colour, incrementing it gives any node which was conflicting the colour 1, resulting in even more conflicts initially. However, this very quickly smooths out, and we get a lower number of conflicts much faster.

Something to keep in mind for both approaches, however, is that neither produce anything close to an optimal colouring. Both only find a feasible colouring, but as I mentioned earlier, this algorithm is little better than simply picking a random colour.

2.6 An Agent on Every Node

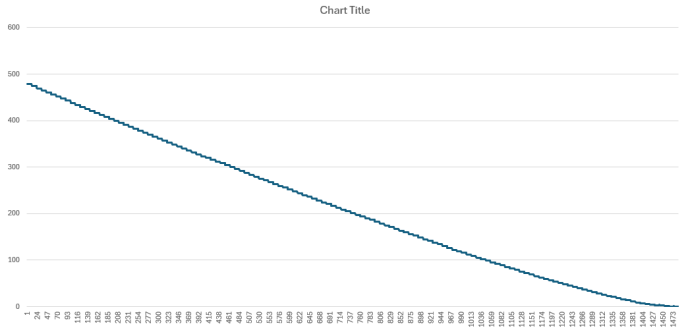
One approach is to have an agent placed on every node. With this setup, the idea is that every node is working in tandem to find the best solution. Unsurprisingly, this is a far more efficient solution, given a kernel (see section 4.3) such as the minimum agent, which simply finds the minimum possible colour that can be applied to a node, given the colours of its neighbours.

Not only is this approach (minimum colour agent on every node) very efficient, it is also very good at finding colourings. In (6), you can see some example output from the program.

```
PS C:\Users\Liam\Documents\fyp> .\colouring.exe -n 500 -p 0.8 -A 10
centralised colour: 137
ran for 376 iterations
number of agents: 500; number of colours: 135; number of conflicts: 0; number of missed nodes: 0
centralised colour: 139
ran for 376 iterations
number of agents: 500; number of colours: 136; number of conflicts: 0; number of missed nodes: 0
centralised colour: 134
ran for 372 iterations
number of agents: 500; number of colours: 135; number of conflicts: 0; number of missed nodes: 0
centralised colour: 134
ran for 376 iterations
number of agents: 500; number of colours: 134; number of conflicts: 0; number of missed nodes: 0
centralised colour: 135
ran for 378 iterations
number of agents: 500; number of colours: 134; number of conflicts: 0; number of missed nodes: 0
centralised colour: 134
ran for 376 iterations
number of agents: 500; number of colours: 133; number of conflicts: 0; number of missed nodes: 0
centralised colour: 137
ran for 372 iterations
number of agents: 500; number of colours: 139; number of conflicts: 0; number of missed nodes: 0
centralised colour: 141
ran for 377 iterations
number of agents: 500; number of colours: 134; number of conflicts: 0; number of missed nodes: 0
centralised colour: 133
ran for 377 iterations
number of agents: 500; number of colours: 135; number of conflicts: 0; number of missed nodes: 0
centralised colour: 134
ran for 373 iterations
number of agents: 500; number of colours: 137; number of conflicts: 0; number of missed nodes: 0
PS C:\Users\Liam\Documents\fyp>
```

Figure 6: Output from running the minimum agent on every node

The only problem I have with this approach is that it is difficult to graph its performance. As every node can only have the minimum colour applied based on what is around it, there are never any conflicts. One work around is to limit the number of colours the algorithm can use. This means that it will not be possible to colour every node in the graph if the limit is lower than $\chi(G)$. If you start with a low number of colours, then gradually increment the limit until you have a graph with no conflicts, you get the graph pictured in (7a).



(a) Graphed results of limiting the number of colours

```
PS C:\Users\Liam\Documents\fyp> .\colouring.exe -M 50000 -n 500 -p 0.8 -c 5 -A 10 -S
centralised colour: 134
max colour: 134
no changes after 1429 iterations
number of agents: 500; number of colours: 134; number of conflicts: 0; number of missed nodes: 0
centralised colour: 139
max colour: 139
no changes after 1484 iterations
number of agents: 500; number of colours: 139; number of conflicts: 0; number of missed nodes: 0
centralised colour: 135
max colour: 135
no changes after 1440 iterations
number of agents: 500; number of colours: 135; number of conflicts: 0; number of missed nodes: 0
centralised colour: 138
max colour: 138
no changes after 1474 iterations
number of agents: 500; number of colours: 138; number of conflicts: 0; number of missed nodes: 0
centralised colour: 138
max colour: 138
no changes after 1473 iterations
number of agents: 500; number of colours: 138; number of conflicts: 0; number of missed nodes: 0
centralised colour: 137
max colour: 137
no changes after 1462 iterations
number of agents: 500; number of colours: 137; number of conflicts: 0; number of missed nodes: 0
centralised colour: 135
max colour: 135
no changes after 1440 iterations
number of agents: 500; number of colours: 135; number of conflicts: 0; number of missed nodes: 0
centralised colour: 135
max colour: 135
no changes after 1440 iterations
number of agents: 500; number of colours: 135; number of conflicts: 0; number of missed nodes: 0
centralised colour: 138
max colour: 138
no changes after 1473 iterations
number of agents: 500; number of colours: 138; number of conflicts: 0; number of missed nodes: 0
centralised colour: 137
max colour: 137
no changes after 1462 iterations
number of agents: 500; number of colours: 137; number of conflicts: 0; number of missed nodes: 0
PS C:\Users\Liam\Documents\fyp>
```

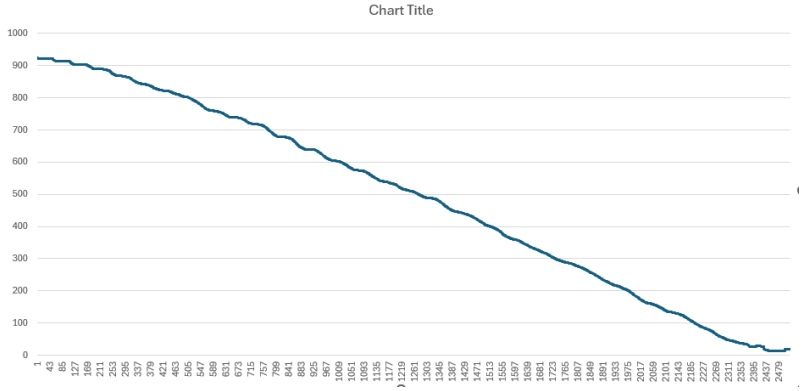
(b) Input and Output

Figure 7: Minimum Agent with Colour Limit

2.7 Mobile Agents

One avenue of research that I have explored is that of making the agents mobile. The basic premise of this is to give the agents a means to decide which node in their vicinity is the best to colour next. This can be repeated a number of times to have each agent take a number of moves on each iteration, after they have coloured a node. The other aspect to this approach is to have far less agents than nodes, reducing the number of nodes that are active on any given iteration.

The motivation behind this was to see how efficiently a smaller number of agents can colour the graph, and how they would perform depending on how many moves they can make on an iteration. The results collected from the colour-blind algorithm in section 2.5 were achieved when using mobile agents that can move twice on each iteration, for example. In figure (8), we can see what happens when we reduce the number of agents to 10% of the number of nodes in a random graph of 1000 nodes with 10% chance each edge exists where each agent can move twice (Note the size of the graph, these results are not directly comparable to 7a; more testing is required to compare the approaches, but so far I have seen results such as a 10:1 node to agent ratio only taking twice as many iterations).



(a) Moving agents with minimum algorithm

```
PS C:\Users\Liam\Documents\fyp> .\colouring.exe -M 50000 -n 1000 -p 0.1 -a 100 -m 2 -c 5 -S -A 10
centralised colour: 34
no changes after 2499 iterations
number of agents: 100; number of colours: 34; number of conflicts: 0; number of missed nodes: 8
centralised colour: 33
no changes after 2400 iterations
number of agents: 100; number of colours: 32; number of conflicts: 0; number of missed nodes: 33
centralised colour: 33
no changes after 2401 iterations
number of agents: 100; number of colours: 33; number of conflicts: 0; number of missed nodes: 20
centralised colour: 33
no changes after 2397 iterations
number of agents: 100; number of colours: 33; number of conflicts: 0; number of missed nodes: 24
centralised colour: 35
no changes after 2515 iterations
number of agents: 100; number of colours: 32; number of conflicts: 0; number of missed nodes: 18
centralised colour: 33
no changes after 2372 iterations
number of agents: 100; number of colours: 33; number of conflicts: 0; number of missed nodes: 61
centralised colour: 34
no changes after 2427 iterations
number of agents: 100; number of colours: 32; number of conflicts: 0; number of missed nodes: 51
centralised colour: 33
no changes after 2402 iterations
number of agents: 100; number of colours: 33; number of conflicts: 0; number of missed nodes: 23
centralised colour: 33
no changes after 2378 iterations
number of agents: 100; number of colours: 33; number of conflicts: 0; number of missed nodes: 63
centralised colour: 34
no changes after 2469 iterations
number of agents: 100; number of colours: 34; number of conflicts: 0; number of missed nodes: 11
```

(b) Input and Output

Figure 8: Minimum Agent which Moves

I believe that this approach could potentially provide significant performance improvements if implemented properly. If only a certain number of nodes are active at any given time, we have less work to do on each iteration. It would be interesting to see if increasing the amount of global knowledge the agents have about the graph could improve their movement so that more dynamic graphs can be solved without the need to perform many iterations where we make no changes (see section 3 for further discussion).

3 The Research To Come

In this section, I will discuss some of the planned outcomes for the project, as well as some of the outcomes I hope to achieve by the time the project is complete.

3.1 Reducing Agents

Similar to the concept presented in section 2.7, the idea of reducing the number of agents in the graph could be an interesting avenue of research to explore. The idea here is to start with an agent on every node, then incrementally decrease the agents present in the graph and see how it affects the colouring process.

Reducing the number of agents in the graph is a paramount to preventing a node from being modified again (provided the agents are stationary). This is an interesting idea, and could be tackled from a number of different directions. Making a node immutable could be done randomly, simply relying on random chance to determine when a node should remain the way it is. We could alternatively define some heuristic or specific condition that determines when a node should become immutable. Maybe we want to make the nodes with the highest degree immutable first (this is veering into global territory, but we could bake a degree threshold into a heuristic), making the activity of a node directly tied to its position in the broader topology of the graph. These approaches should produce different results, and could give us interesting information and data that could be fed back into mobile approaches.

3.2 Visualisation

Currently, the program prints out only the final results of the operation by default. There is also the option to print out the graph in a text-based format, but once you exceed 20 nodes in the graph, this becomes very difficult to decipher and understand. As a result, the visualisation of the topology of a graph would make understanding results far easier. Ideally, this would also allow for the colouring process to be replayed, although this could vastly increase the memory requirements of the program.

Going even further, the program would find the optimal angle for the graph to be viewed at (its "good side"), or perhaps allow for user interaction, although these are not priorities of the project at this point.

3.3 Dynamic Graphs

Of course, the driving force behind this project is the concept of dynamic graphs, and how they can be tackled using decentralised approaches. For example, imagine a situation in which the graph is expanded, doubled in size even, but is connected to this new portion of the graph by a single edge. In this case, we should expect nearly the entirety of the original graph to remain how it is, but the new portion should still be coloured. Could we create an algorithm in which the agents in the graph can move to the new portion quickly, and exclusively colour this section, but with confidence that we are maintaining an optimal colouring? This is one of the questions that dynamic graphs pose.

Another, more general question is what effect the use of dynamic graphs has on the colouring methods we already use on static ones. I.e., how does removing or adding an edge/node effect the processes we use.

Finally, something which is definitely worth investigating is whether we can use decentralised colouring approaches to *identify* places in the graph where removing or adding a constraint could improve the colouring process. This could be returned as a result to the user, or it could be fed back into the algorithm, to provide not only the optimal colouring of the original graph, but how significantly it could potentially improved with the fewest number of changes to the topology.

3.4 Bad Actor

Up to now, we have had every agent working towards the common goal of finding an optimal colouring for the graph. However, what might happen if were to include an agent whose goal did not align with the others? Consider an agent which works not to reduce conflicts, but instead to increase them. How should we run the algorithm while we are actively being prevented from finding an optimal solution? Could the other agents working on the graph identify this bad actor in a decentralised system? What if the bad actor is mobile, while the other agents are stationary? There are a number of interesting research questions that this concept raises.

4 The Means

In this section, I will discuss some aspects of the code implementation I have used and will continue to develop in order to obtain results for this project.

You can also browse the source code build the tool yourself in the GitHub repository.

4.1 The Tool

The approach I took to this project was to create a command line tool which could be dynamically used and modified in order to produce repeatable results of various experiments of different types. In order to do this, I employed the C language, which I find to be intuitive when implementing algorithms. Importantly, it also allows for fast execution, which saves time when running large experiments.

I achieved this via including a large number of command line options the user can include in order to customise each run of the program. It is well documented in a number of places, including the ReadMe of the repository linked at the beginning of section 4, but the user can also print the instructions to their screen by running the program with the `-h` option.

One small implementation-side improvement I hope to be able to make in this project is to possibly find a cleaner way of reading the command line options, as there are a lot of them at this point.

4.2 Graph Implementation

The approach I took to implement graphs in code form was to model them as a linked-list type structure of `node` structs. Each struct has an `id` (correlating to order of creation), a `colour` code, a `degree` and an array of `node` pointers, representing its neighbours (see figure (9)).

```
typedef struct nodeStruct {
    int id;     // i included this to make copying easier
    int colour;
    int degree;
    struct nodeStruct** neighbours;
} node;
```

Figure 9: node Struct

There are many benefits to modelling the graph this way. One of the major ones is the ease with which it can be traversed, which is vital for a system which needs to allow for mobile agents. Since I am using C, I can simply represent agents as pointers to a particular node. Moving an agent then becomes as simple as updating a pointer.

Another major benefit is the ease with which the graph can be modified, ideal for a project focusing on dynamic graphs. For example, look at the function used to create a bipartite graph. In section 2.1, I eluded to the fact that this function was mainly added in order to test the implementation, and it does indeed showcase how easily two completely separate graphs can be combined into one (see figure (10)).

```
//add g2 to g1
for(int n = setOne; n < setOne + setTwo; n++) {
    g1[n] = g2[n - setOne];
    g1[n]->id = n;
}
```

Figure 10: Combing the two parts of a bipartite graph

4.3 Kernels

Another major benefit to using the C programming language is the ability to use higher order functions. This allowed me to implement a kernel-like system where the algorithm the agents use to change the colour of the graph and make moves can be dynamically altered. The general

evolutionary algorithm sits around this, providing the same base functionality, but allowing me to produce entirely different results.

4.4 Utility Functions

Personally, I found the `graphutil.c` file to be the most fun to write. While each function is small, they all perform very different tasks, and put my knowledge of algorithms and ability to solve coding problems to the test. For me, these functions provided fun side-track rabbit holes to think about, giving me a break from graphs to think about more abstract coding challenges, while also directly benefitting the project. Any one of these functions would be a good example, as all are remarkably simple, yet entirely unique. There are no stand out approaches to any particular problems here, but I believe it is worth mentioning them as a whole.

4.5 Centralised Benchmark

When implementing various decentralised approaches, I quickly realised it was impossible for me to tell whether an algorithm was performing well on a graph of 1000 nodes. To resolve this, I implemented a very rudimentary benchmark, which is based on a greedy colouring approach. This centralised approach simply iterates over each node and applies the minimum possible colour to each one. It provides a decent benchmark to immediately give feedback on how a given algorithm has performed in relation to a simple brute force approach.

4.6 Other Languages and Paradigms

One goal I think could be an interesting one to explore is the exploration of other languages and paradigms through re-implementing this tool in other languages. A currently popular low-level language is the Rust programming language, which sees itself as a safer version of C. Given I have not taken much care to prevent use-after-free vulnerabilities in this code, it would be interesting to see what changes might be necessary in order to implement this tool in Rust.

While I do like the procedural paradigm, I have also long been interested in trying to get into the functional paradigm more, as I see it as the paradigm that probably makes the most logical sense in theory. In particular, LISP is very appealing to me. There is an implementation in the JVM, called Clojure, which could be a good place to start. Implementing this tool in a functional paradigm would be an interesting challenge, but doing it in a high-level language such as Clojure would also force me to rethink some of the approaches I have taken in the current implementation.

This is not a high priority of the project, but it is worth mentioning as a long-term goal.

References

[1] Carl Johan Casselgren. "On Some Graph Colouring Problems". PhD thesis. Umea University, 2011.

[2] R.M.R. Lewis. *Guide to Graph Colouring; Second Edition*. Springer, 2021.

[3] Geeks for Geeks. *Instroduction to Graph Colouring*. 2024. URL: https://www.geeksforgeeks.org/graph-coloring-applications/.

[4] Heon Lee. *Coloring Code: How Compilers Use Graph Theory*. 2023. URL: https://www.youtube.com/watch?v=K3mi2m7ccDQ.

[5] Besjana Tosuni. "Graph coloring problems in modern computer science". In: *European Journal of Interdisciplinary Studies* 1.2 (2015), pp. 87–95.

[6] Wen Sun. "Heuristic algorithms for graph coloring problems". PhD thesis. Université d'Angers, 2018.

[7] Daniel Brélaz. "New methods to color the vertices of a graph". In: *Commun. ACM* 22.4 (Apr. 1979), pp. 251–256. ISSN: 0001-0782. DOI: 10.1145/359094.359101. URL: https://doi.org/10.1145/359094.359101.

[8] Frank Thomson Leighton. "A graph coloring algorithm for large scheduling problems". In: *Journal of research of the national bureau of standards* 84.6 (1979), p. 489.

[9] Thang N Bui et al. "An ant-based algorithm for coloring graphs". In: *Discrete Applied Mathematics* 156.2 (2008), pp. 190–200.

[10] Matthieu Plumettaz, David Schindl, and Nicolas Zufferey. "Ant local search and its efficient adaptation to graph colouring". In: *Journal of the Operational Research Society* 61.5 (2010), pp. 819–826.

[11] P ERDdS and A R&wi. "On random graphs I". In: *Publ. math. debrecen* 6.290-297 (1959), p. 18.