Distributed Graph Colouring

A final year project completed for the BSc in Computer Science and Information Technology

Authored By

Liam Holland

21386331

Supervised By

Dr. Colm O'Riordan

School of Computer Science
College of Science and Engineering

Acknowledgments

I want to acknowledge my supervisor Dr. Colm O'Riordan, who not only convinced me to do this project in the first place, but provided invaluable assistance, guidance and advice throughout. When working on such a broad topic, it can be sometimes be difficult to know which direction to go and which ideas are worth pursuing. Having someone to turn to that can help you decide which ideas are worthwhile, provide new suggestions and engage with you in heated discussion with was endlessly worthwhile.

Links to Additional Material

Project Repository: https://github.com/liamholland/fyp-code
Video Demo: https://youtu.be/Zcn6IQtaDM

Contents

List of Algorithms

List of Figures

List of Tables

1 Introduction

The graph colouring problem is a well-known problem in computer science. The problem deals with the goal of attempting to colour a set of vertices connected by edges. The model of the problem can be used in order to represent real-world problems; scheduling, for example. However, real-world problems are prone to change, which gives rise to the need for a reliable means to solve dynamic problems. This can be done using distributed approaches. In these approaches, nodes act individually using agents in order to move towards a solution in the society.

In this project, we investigate the ability of agents which have as little centralised knowledge as possible in their ability to solve the GCP on randomly generated graphs. We then expand the problem to consider cases where the graph can change (dynamic graphs) as well as more niche experiments, where nodes become immutable during the colouring process, for instance. We also attempt to use the distributed agents to solve problems that arise in real-world and abstract situations, such as identifying poor constraints in the graph and investigating how the agents behave when there is a bad actor among them. Finally, we investigate the behaviour of the agents when they are mobile, capable of moving between nodes on each iteration of the colouring process.

In order to perform these investigations, a command-line tool was written in a low-level language that facilitates extensive control over every aspect of the implementation. This tool facilitates extensive customisation and repetition of experiments. The code repository for this tool, scripts to run experiments discussed in this report and related results files can all be found in the GitHub repository.

1.1 Core Objectives

In this project, we aim to

- design and implement colouring and utility algorithms
- investigate and compare the performance of different local colouring algorithms
- incorporate changing constraints into the graphs
- relate distributed graph colouring problems to real-world problems
- design more advanced experiments to investigate the abilities of distributed systems

1.2 Additional Objectives

In this project, we also aim to

- investigate different graph types
- visualise the colouring process
- research previous work on distributed systems and compare these approaches with ours
- implement both agent-based sequential colouring from an initial colouring and a truly distributed, path-based approach for finding optimisations in pre-coloured graphs

2 Related Work

This section provides an overview of the Graph Colouring Problem (GCP), the problem which defines this project.

2.1 Graph Definition

A "graph", is a set of vertices (or nodes) V connected together by some set of edges E. For this project, we are dealing exclusively with undirected graphs, meaning that an edge connecting two nodes can be traversed in either direction. An example of a graph can be seen in Figure 1. Graphs are a commonly used data structure in modelling complex problems in computer science.

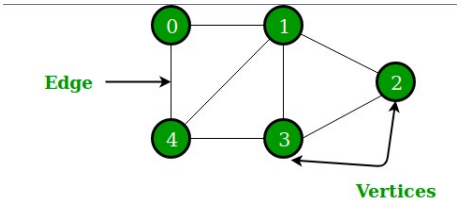


Figure 1: Example Graph

2.2 Formal Definition and Terminology

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.

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.

Finally, in this project we will be considering the concept of sub-optimal colouring in order to better represent real-world problems. For example, we can colour the graph with the maximum degree plus one, but we might consider allowing more colours in order to obtain a solution faster, even if we know that it is sub-optimal in terms of achieving $\chi(G)$. Additionally, we will consider cases where we wish to find a balance between the number of colours in the graph and the number of conflicts remaining after we finish the colouring. These concepts are discussed further in Section 2.8.

2.3 Types of Graphs

Different arrangements of vertices and edges have different names that are assigned to describe a general category that they fall into. For example, bipartite graphs are graphs that have their vertices divided into two distinct sets [3], a ring graph is a graph where each node has a degree of two and the rank is equivalent to the free rank of the graph [4].

In this project, we will deal primarily with randomly generated graphs based on the Erdős–Rényi model [5]. The details of this model are discussed in Section 4.5.

2.4 NP-Hardness

A problem is in the set NP (non-deterministic polynomial) if there is no known 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 [6]. 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 Figure 2.

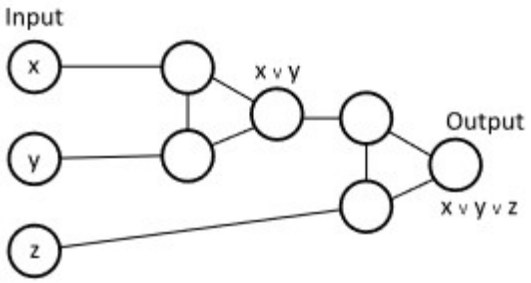


Figure 2: OR-gadget graph [2]

NP-Hard problems are interesting since finding efficient ways to tackle these complex problems can often take you down unexpected routes of thinking.

2.5 Applications

There are many useful applications of the Graph Colouring Problem. As mentioned above in section 2.2, 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 application 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 [7] [8] [9] [10].

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 [11].

This section has laid out how the GCP is useful in a practical sense, 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.

2.6 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 famous is the simple greedy algorithm, which attempts to assign the smallest possible colour to each vertex [12].

Another of the most well-known 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 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 [7]. This algorithm essentially works by extending partial solutions [13]. In terms of graph colouring, this generally means revisiting nodes to see if they can be modified to use a different colour to improve the colouring.

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 [14]. 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 [15] and RLF [16], along with other local search methods (see section 2.7).

2.7 Decentralised Approaches

The idea of a decentralised or distributed 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. Decentralised approaches are types of local search algorithms. They can take many different forms, but they are usually based in the realm of evolutionary and genetic algorithms. An important note is that the algorithms we are

using are different to construction heuristic algorithms. These algorithms are similar in that they end after a certain number of iterations or generations; when there are no changes for a certain number of iterations, for example. In our algorithms, we do not maintain a *pool* of solutions in these algorithms, instead we have the local agents work towards a solution iteratively [17]. 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.

There has been related work done with developing decentralised approaches to finding colourings for graphs. Galán proposed a simple approach which is similar to the goals proposed in this project [18]. The algorithm focuses on reducing the number of conflicts in the locality. It also makes no changes if the node is not in conflict.

Other approaches are more complex, but still attempt to remain as decentralised as possible. Checco, Alessandro, Leith and Douglas proposed an algorithm that has no message-passing between nodes, but instead infers satisfaction with decisions from its surroundings [19]. This kind of algorithm is particularly useful in the wireless networking space, where you have no idea what your neighbour might be doing. In these kinds of situations, it is difficult to even determine if your locality is fixed, and different nodes and constraints could ephemerally enter and exit the equation.

Another approach that has picked up steam in recent years are ideas of agent-based local search algorithms such as 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 concepts 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 [20] [21]. In the context of the GCP, this leads to agents evolving a near-optimal solution from a base initial state.

While decentralised approaches tend to be less efficient than centralised counterparts, their benefits outweigh the cost in efficiency. 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. There may also be cases where it is impossible to know all of the conflicts, due to the division of departments in a university, to continue the analogy. In these cases, distributed approaches can again grant us the ability to identify and fix these subtle conflicts.

Additionally, these approaches are ideal to be translated into a multithreading approach. It should be very straight forward to initialise 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.

2.8 What are we solving for?

The goal when searching for a solution to the GCP is often an optimal solution, but there are also instances where a sub-optimal solution is acceptable. There will always be a trade-off between reducing the number of colours, compute time and conflicts. In some cases, we will be aiming to solve as quickly as possible, which may limit our ability to achieve the optimal results. In others, we may not care if there are certain amount of conflicts, but restrict certain nodes to be certain colours. Take our class scheduling example from Section 2.7. We may have classes that are only on sometimes, or part of the year, or encounter cases where a certain class must be in a certain room. These are cases where we may be willing to accept sub-optimal or even improper colourings (a colouring that has conflicts) as solutions.

Similarly, we might consider achieving a colouring with more colours than necessary in order to find a feasible solution faster. Alternatively, we can design algorithms which take this notion into account, allowing them to adapt to being able to find optimal solutions when dealing with a sub-optimal limit on the number of conflicts. We might also come to a point where we want to increase the number of colours in the graph, even if it is not quite optimal, but makes the problem easier to solve. Continuing our class scheduling analogy once again, adding new colours might be equivalent to building a new room that classes can be taken in, or hiring a new teacher.

These examples show how relaxations in the goals of the problem allow us to more accurately map the GCP to real-world problems. In Section 5, we will discuss methods of dealing with these

trade-offs further.

2.9 Graph Representations

There are generally two different ways of storing graphs in computer programs: an adjacency matrix or an adjacency list. There are advantages and disadvantages to both approaches. For example, while the adjacency matrix provides an O(1) means of looking up edges, it has masses of redundant data when storing sparse graphs. On the other hand, an adjacency list might be more compact, but it is more complicated to check if two nodes are connected, and becomes more expensive as the list grows [22]. Neither representation is ideal for this project, however, given that both are a form of centralised information and neither are especially suited to dynamic graphs.

Instead, this project implements a custom structure based around individual nodes, which is discussed in Section 4.4.

2.10 Dynamic Graphs

There has been some previous work done on investigating the continued colouring that must be done in order to recolour a graph. These are the *other* types of distributed approaches, where the distributed aspect comes from only considering a small number of vertices in order to achieve a proper colouring of the entire graph. In general, there is a trade-off between the number of nodes that you need to consider and how fast the recolouring can be completed [23]. Some algorithms focus exclusively on dynamic edges in the graph [24]. Other approaches have allowed the algorithms to have some knowledge of what changes will be made to the graph in the future, which makes it possible for the current algorithm to adapt in order to reduce the number of conflicts when these changes occur [25].

3 Development

This section gives an overview of the development process relating to the construction of the command line tool used to run the experiments for this project.

3.1 Choice of Language

The language chosen for this assignment was the C programming language. The reasons this was chosen was largely for practical reasons, but there are benefits beyond the practical also.

First and foremost, the language is fast. This is ideal for a project which revolves around a large amount of data processing. In Section 5, some results show how it can take a significant amount of time to run certain colouring experiments with a consequential set of parameters. Using C meant that the code will be executed relatively quickly, even when running the code on large graphs.

Another reason to use C is the fact that from an algorithmic perspective, it is very easy to read code written in C. While pointers do have some mental overhead when initially looking at the code, consistent, verbose and clear naming of variables can alleviate a lot of this difficulty. Similarly, separating out the code into different source and header files with descriptive and clear documentation is a simple step that can be taken to make the code base far simpler to traverse. Take `graphutil.c`, for example. While this is a large source file, the header file lays the function prototypes and their related documentation out clearly. Maintainers can quickly get a handle of what this part of the code base is for, as well as what each function does, without needing to worry about the finer implementation details. This is a core principle of good API design and crucial in the design of a program that is meant to be easily extensible.

Finally, the benefits that writing a project such as this beyond its conclusion are innumerable. Understanding memory manipulation, algorithmic implementations of basic operations, as well as just the language itself is of immense benefit to any programmer, no matter the language they are writing in. Using a low level language such as C provides a great programming challenge, as well as many learning opportunities which can be taken forward into any programming problem.

3.2 Version Control

This program was maintained and tracked with the Git version control system. The repository was hosted on GitHub remotely. There are a couple of reasons, not least of which was the requirement to back up the code and share it between multiple devices (see Section 3.4).

However, the primary benefit of version control is just that: controlling the version. I was sure to restrict pushing to the master branch right from the beginning of the project, to avoid a situation where a functional version of the application was broken. This means that all additions and changes had to be made in a branch, which could then be merged with the master branch via a pull request.

I would encourage readers to explore the pull request history of the repository, as each one contains an in depth description of that change, what it does, why it was done and how it was done. It also gives a sense of the development process of the project, problems that came up and how they were solved. Figure 4 is an example of how detailed these pull requests are.

Finally, the GitHub repository includes in depth instructions of how to compile and use the program yourself, which allows for experiments discussed in this document to be replicated on your own machine (See Section 5).

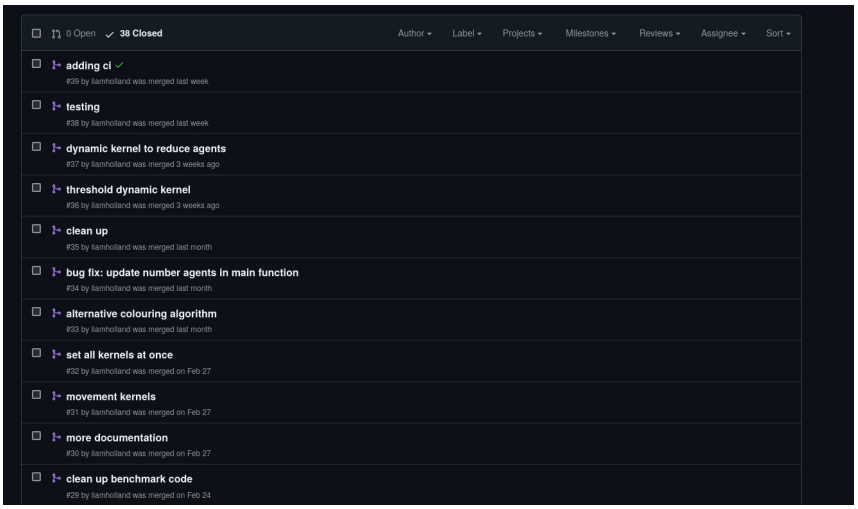


Figure 3: Pull Requests in the GitHub Repository

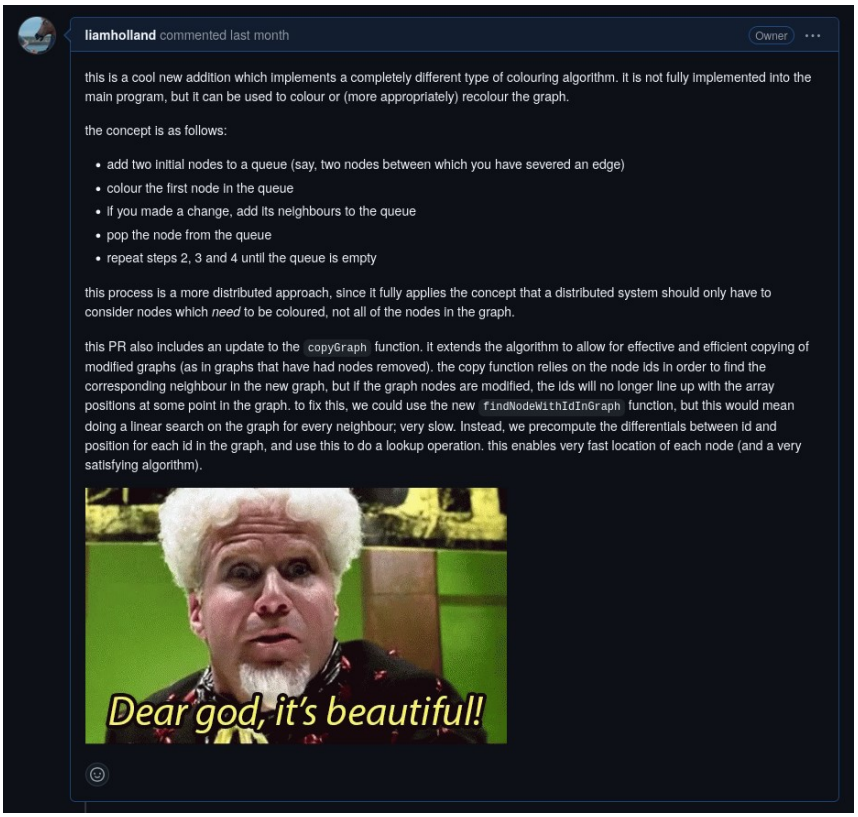


Figure 4: Example of the level of detail in PRs

3.3 Tooling

This project saw the use of a number of different common tools which allow for smoother development of projects in the C programming language.

3.3.1 Code Editor

The VSCode/VSCodium editor was used for this project. This is a lightweight, open-source editor that is highly extensible and practical to use. Extensions can include anything from additional syntax highlighting for the C language, to git repo graphing which allows for easier visualisation of changes being made to the code base.

3.3.2 Make

Make is a vital tool when writing a project in C. Make allows you to define macros for compilation commands which makes the process of iterating on changes far less tedious. Rather than entering the full compilation command every time we want to compile, we can instead define it in the `Makefile` and simply enter `make` into the command line. You can also define multiple different commands which can be executed individually. For example, this project implements a `debug` command that will compile an executable that can be debugged with gdb and a `compile_test` command that will produce the test executable.

3.4 Development Environment

This project was developed across two different machines. The first is a custom PC with a Ryzen 5 2600x CPU @ 3.6GHz and 16GB of RAM running Windows 10, the other is a DELL Inspiron 15 5000 laptop with an 11th Gen Intel i5 CPU @ 4.2GHz and 8GB of RAM running a Debian-based distribution of Linux, Linux Mint.

The ability to develop and run the program across two different machine, running two different CPU architectures and two different operating systems was a very useful boon to avail of when writing such a project in C. It allowed for various different issues to be caught and fixed very quickly and likely improved the correctness of the code overall.

That being said, the use of two different machines was purely a pragmatic one, and there is no investigation or discussion of the performance on the different machines in this project.

3.5 Testing

The code base has a completely custom testing framework that was implemented in order to verify the logic of some of the core functions of the program. Given the dual development environments and the nature of the C language, a lot of the code is easy to verify naturally, given that most mistakes will cause a crash. However, unit tests still have value, as they can show in a provable manner that the actual logic of the code works as intended, not simply that it does not run.

The tests are also run automatically as part of a continuous integration pipeline in the GitHub repository whenever code is pushed to a branch, or a pull request is made, enabling constant and automatic verification of changes.

The implementation of the testing framework is discussed in Section 4.

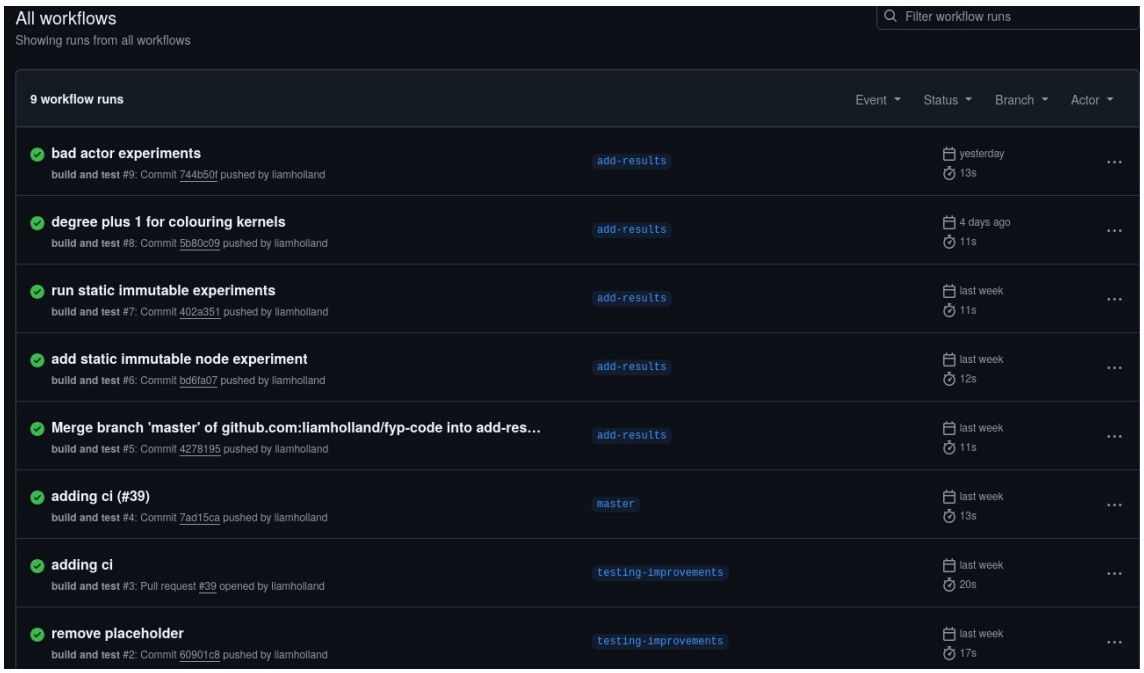


Figure 5: The CI results board in the GitHub Repository

```
13:39:44 fyp-code master » ./test_colouring

/// 28 tests have been registered ///

TESTING testPass... PASSED
TESTING testFail... PASSED
TESTING testInitialiseGraphNoNeighbours... PASSED
TESTING testInitialiseGraphWithNeighbours... PASSED
TESTING testCopyUnmodifiedGraph... PASSED
TESTING testCopyModifiedGraph... PASSED
TESTING testFreeGraph... PASSED
TESTING testFindNumColoursUsed... PASSED
TESTING testCountNumConflictsAll... PASSED
TESTING testCountNumConflictsNone... PASSED
TESTING testNumUncolouredNodes... PASSED
TESTING testFetchNodes... PASSED
TESTING testFetchNodesMoreThanGraph... PASSED
TESTING testNodeInConflict... PASSED
TESTING testWhichColoursInGraphOnlyOne... PASSED
TESTING testWhichColoursInGraphAll... PASSED
TESTING testFindConflictingNeighbours... PASSED
TESTING testFindConflictingNeighboursNoConflicts... PASSED
TESTING testCountNumberOfConflictsForNode... PASSED
TESTING testRemoveEdge... PASSED
TESTING testMakeNodeOrphan... PASSED
TESTING testRemoveNode... PASSED
TESTING testAddEdge... PASSED
TESTING testFindHighestDegree... PASSED
TESTING testFindLowestDegree... PASSED
TESTING testRemoveNodePointerFromList... PASSED
TESTING testFindMostCommonColour... PASSED
TESTING testFindNodeWithID... PASSED

/// ALL TESTS PASSED ///

13:39:46 fyp-code master »
```

Figure 6: Running the tests locally

4 Implementation and Design Details

This section provides an in-depth explanation of the implementation details of the command line tool developed to run experiments for this project, and the details of the algorithms within it.

4.1 Program Structure

The program is structured in such a way that means that the code is easy to digest and expand. The code base is not that large, and simply looking at the header files will give a reader all of the understanding they need in order to properly understand the algorithms listed below. The project structure is laid out in Figure 7.

The flow diagram illustrates the way the operational components of the program fit together. There are other components which are not included in the diagram, since they are used throughout, such as the `node.h` file, which provides the `node` type (Section 4.4) and `graphutil.c`, which as the name suggests provides utility functions used through the entire program.

The flow diagram is a high-level view of the programs operation, but its behaviour is determined by the options that it is run with.

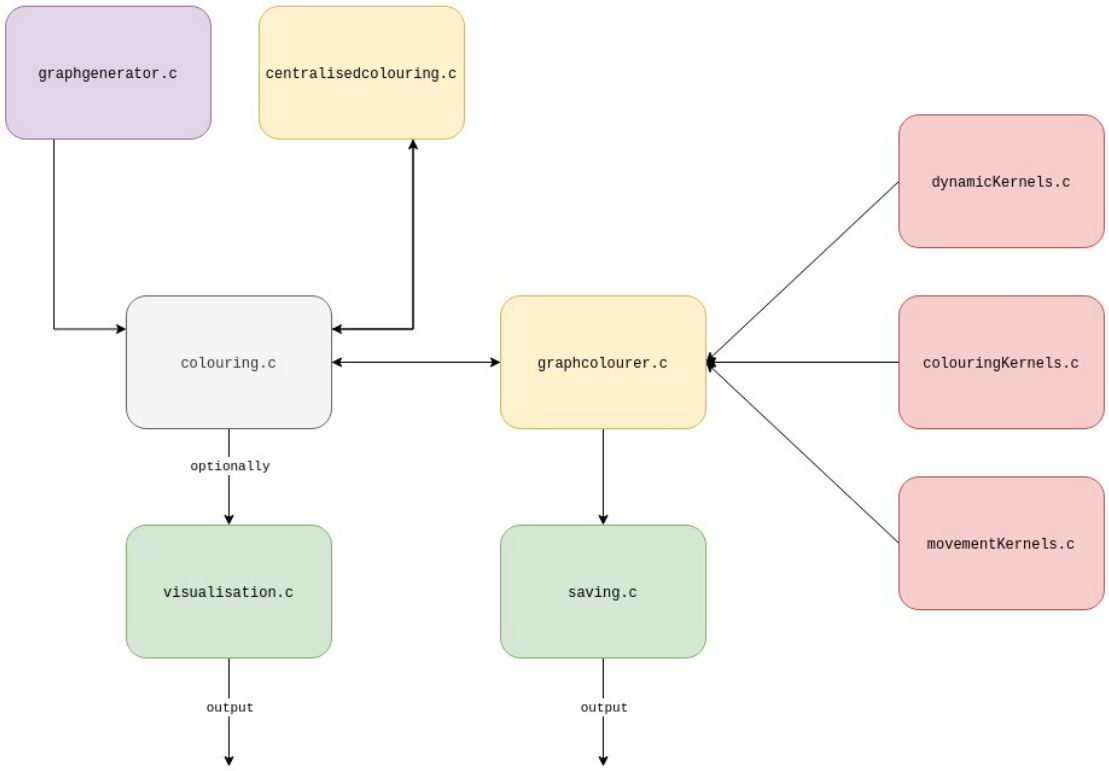


Figure 7: Program Structure

4.2 Command Line Options

The defined options provide the ability to the user to define the behaviour of the program on any given run. In essence, this is what the user uses to determine the experiment they are conducting. The options are thoroughly documented and this documentation is available both in the repository and in the program itself by running the `-h` option. For the readers convenience, the documentation is provided in Figure 8, although it is not necessary to know or understand the options in order to understand the experiments in this report.

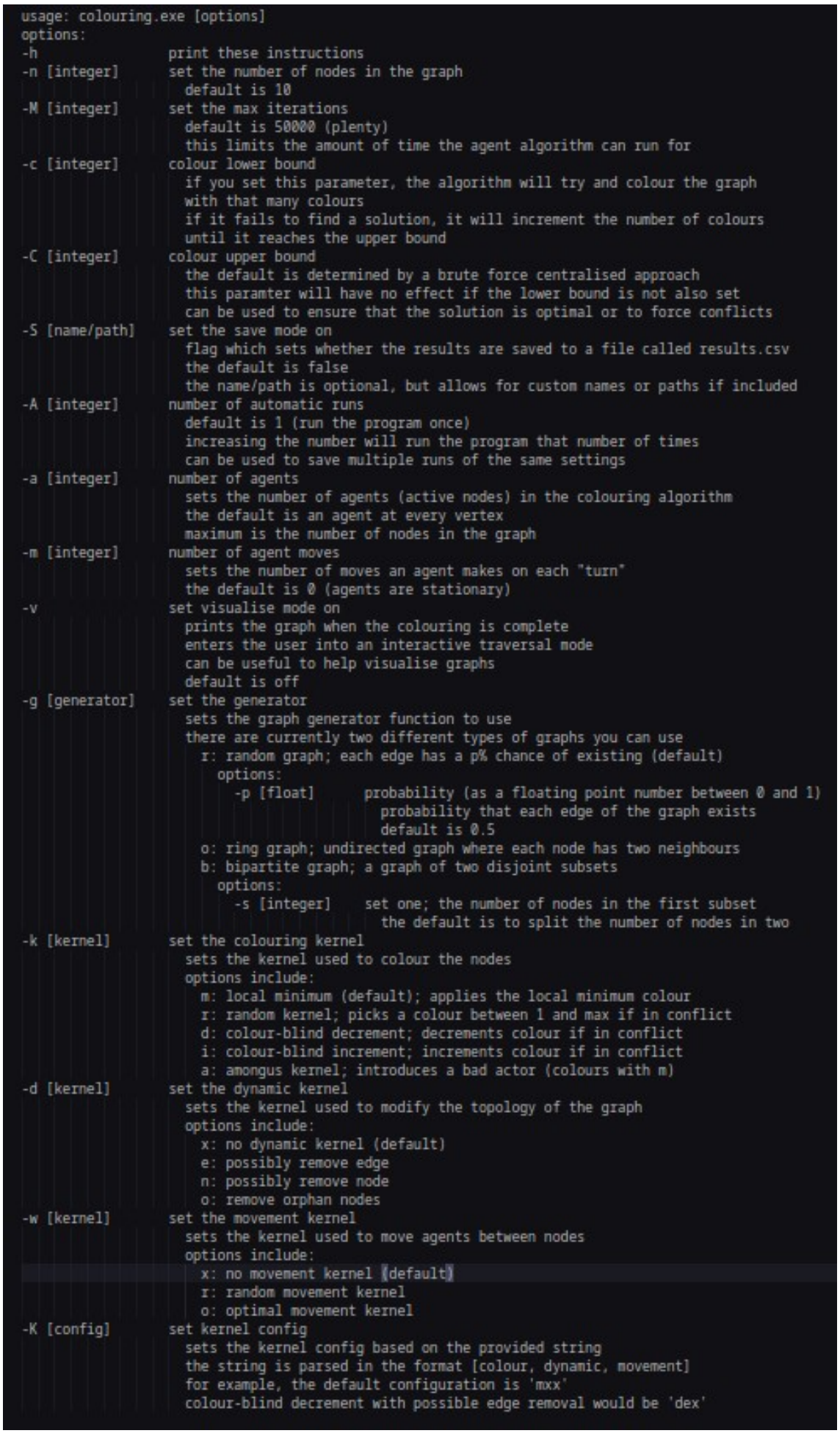


Figure 8: Command Line Options

4.3 Colour Representation

Colours are represented as a set of numbers. As such, a vertex being assigned a "colour" will mean it has been assigned a number $c \in \{1, \ldots, k\}$ 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 fully 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 plus one.

4.4 Graph Representation

Graphs are represented as arrays of `node` structs, which contain the important information about each node, and allow us to find information about the entire graph or the locality of a single node. The node struct is as follows:

```
typedef struct nodeStruct {
    int id;
    int colour;
    int degree;
    struct nodeStruct** neighbours;
} node;
```

Coming up with a robust means by which graphs could be represented was one of the earliest and most crucial challenges in the whole development process. Ensuring that the representation was robust is an excellent example of a conceptual challenge. It also required considerable practical knowledge, as it was important to understand not only how to implement the graph representation, but *why* the conceived solution would work.

The recursive address-based means of keeping track of neighbours means we avoid duplication of structs, reducing the amount of memory required to store the graph. There is an alternative approach of maintaining an edge matrix, where we indicate which nodes are connected to each other via a 1 or 0. While a structure like this would have the memory required to store a fully connected graph, it will also use much more memory than the above approach for less dense graphs. On top of that, we would be keeping track of which nodes are *not* connected, as well as which ones are, which is redundant data, on top of making it more difficult to iterate over neighbours. The above implementation also has the advantage of allowing for very neat code based around pointers which allows us to use the same utility functions for the entire graph, or a list of neighbours (sub-graph), as well as being far more suitable to dynamic approaches; we only have to modify the affected nodes rather than a whole centralised structure.

4.5 Types of Graphs

The tool implements three different types of graphs that can be used in colouring experiments. These three types are ring graphs, bipartite graphs and Erdős–Rényi random graphs [5]. Ring and bipartite graphs are straight-forward to colour, and were primarily added in order to test the implementation and structure of the project.

The latter, random graphs, are the type is the one used to generate graphs for experiments in this project. The implementation follows the model, giving each potential edge a $p\%$ chance of existing. This can be achieved quickly by only iterating over half of the edge matrix, and connecting both relevant nodes if the chance determines that the edge should be connected. As such, p determines the *density* of the graph used in the colouring algorithm. This is illustrated in Algorithm 1.

4.6 Centralised Benchmark

In order to get a sense of the performance of each colouring algorithm, a simple greedy benchmark was added. This benchmark is categorised as a centralised benchmark, as it simply iterates over every node in the graph directly, and applies the smallest colour it can based on the nodes neighbours. This is similar to a single iteration of the colouring algorithm, but here we keep a centralised metric of the number of colours in the graph, which can be returned at the end. Pseudocode is included in Algorithm 2.

Input: the number of nodes N and density percentage p
Output: a graph G

$G \leftarrow N$ unconnected nodes
for $n \leftarrow 0$ **to** $N - 1$ **do**
  for $nb \leftarrow n + 1$ **to** N **do**
    if *random chance* p **then**
      $G[n]_{neighbours} \leftarrow G[nb]$
      $G[nb]_{neighbours} \leftarrow G[n]$
    end
  end
end

Algorithm 1: Erdős–Rényi random graph generation

Input: a graph G and the number of nodes N
Output: An estimate of $\chi(G)$, m

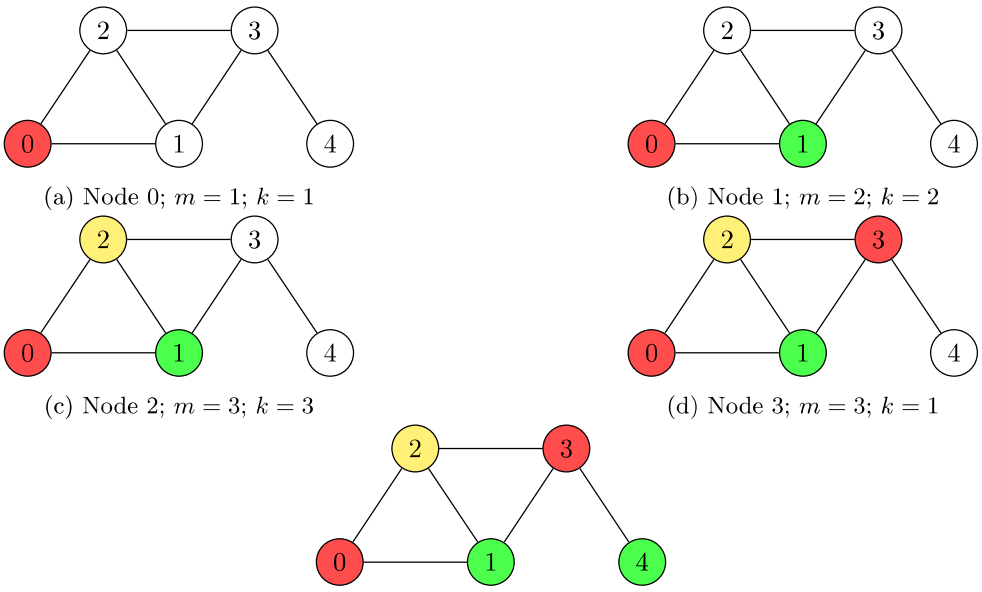
$m \leftarrow 1$
for $n \leftarrow 0$ **to** N **do**
  for $c \leftarrow 1$ **to** m **do**
    $G[n] \leftarrow c$
    if $G[n]$ *not in conflict with neighbours* **then**
      break
    else if $k = m$ **then**
      $m \leftarrow m + 1$
    end
  end
end

Algorithm 2: Centralised Benchmark

The algorithm begins with an estimate of 1, meaning 1 colour is needed to colour the graph. It then visits each node in the graph one-by-one in an arbitrary order. At each node, it will iteratively cycle through each colour from 1 to m, and check if that colour can be assigned to the node without causing a conflict. If it reaches such a colour, it assigns the colour, stops and moves on to the next node. If it reaches m, it will increment the value and assign the new colour, then move on. The final value of m is returned as the estimate. No nodes are revisited in this algorithm, and it follows no particular optimal path (e.g. starting with the node of highest degree). In other words, the value returned is highly unlikely to be $\chi(G)$ in complex graphs. On the other hand, applying the minimum possible colour to each node means that we effectively limit the number of colours in the graph to a small number and the colouring will always be feasible (conflict free). This makes it perfect to provide an estimate to measure our decentralised approaches against.

The benchmark is also useful in the sense that it can provide us with a maximum value to colour to. If the user sets a minimum number of colours, but not a maximum, the program will use the benchmark as the maximum. This way, we can test incrementing the number of colours in the graph without it the need to wait an extremely long time for poor performance algorithms to finish and quickly obtain estimates as to whether a decentralised algorithm is capable of effectively colouring the graph.

A visualisation of the benchmark algorithm's process is shown in Figure 9.



(a) Node 0; $m = 1$; $k = 1$

(b) Node 1; $m = 2$; $k = 2$

(c) Node 2; $m = 3$; $k = 3$

(d) Node 3; $m = 3$; $k = 1$

(e) Node 4; $m = 3$; $k = 2$

Figure 9: Centralised benchmark visualisation

4.7 Kernels

Kernels are an important part of this project. The kernels are a means of defining the behaviour of agents in the graph colouring algorithm. By structuring the program like this, it makes it easy to quickly change the way the agents will behave and combine different algorithms with different parameters.

There are three different types of kernels in the program. These are the colouring kernels, movement kernels and dynamic kernels. Each is a set of algorithms that can be combined together in any way. The details of these algorithms are explained in later sections.

The kernels are combined in the main program loop, where they are called on each of the agents currently operating on the graph. This is shown in Algorithm 3.

Input: a graph G
⋮
for *agent a operating over* G **do**
  colouringKernel(a)
  movementKernel(a)
end
for *agent a operating over* G **do**
  dynamicKernel(a)
end
⋮

Algorithm 3: Excerpt: calling the kernels

Note how the dynamic kernel is applied separately to the others. This is done so that the modification of the data structure does not affect the colouring process. In other words, the graph goes through a colouring iteration, and is modified after every agent has taken a colouring action.

One other important note is the fact that the only kernel which must be called is the colouring algorithm. The other kernels are optional parameters to the program that allow for more advanced and complex experiments, but the program will operate fine if neither are defined functions.

4.8 Colouring Algorithms

There are a number of different colouring algorithms that have been implemented in the program. This section will give an overview of the main three algorithms used in experiments: random local colour, colour-blind local colour and minimum local colour.

All three algorithms take the same input of an agent and a max colour k that imposes a limit on the number of colours the algorithm is allowed to use.

4.8.1 Random Local Search

This algorithm is illustrated in Algorithm 4. If the agent's node is in conflict with its neighbours, it will pick a random colour between 1 and k until it is no longer in conflict. This is the simplest algorithm in the program and provides a baseline to compare other algorithms to more than anything else.

Input: an agent a and a max colour k
Output: the number of changes made to the graph, x

$x \leftarrow 0$
while *nodeInConflict(a)* **do**
  $a \leftarrow \text{rand}() \bmod k$
  $x \leftarrow 1$
end

Algorithm 4: Random Local Colour

4.8.2 Colour Blind Local Colour

This algorithm is depicted in Algorithm 5. The major difference is the inclusion of m, which allows the node to limit itself to its own degree plus one (the maximum number of colours that could possibly be required to colour it).

There are also two different implemented versions of this algorithm. The one depicted in Algorithm 5 is colour-blind local colour (decrement). There is an increment variant which is identical apart from the fact that the algorithm increments the agent's colour rather than decrementing it if it is in conflict with its neighbours.

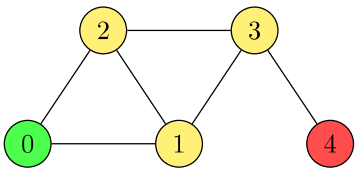
This algorithm does not extend the knowledge of random colour, but makes the colouring process slightly more directed. The intent of this algorithm is to see if we can implement an algorithm which achieves a more effective colouring than random local colour, without increasing the local knowledge. As such, the algorithm is effectively colour-blind; it has no knowledge of which of its neighbours it is in conflict with, or what colour is causing the conflict. However, the agent *does* have more knowledge about itself, taking its degree into account.

Input: an agent a and a max colour k
Output: the number of changes made to the graph, x

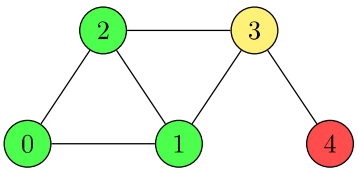
$x \leftarrow 0$
$m \leftarrow k < deg(a)\ ?\ k : \deg(a)$
if *nodeInConflict(a)* **then**
  $a \leftarrow colour(a) - 1 \bmod m + 1$
  $x \leftarrow 1$
end

Algorithm 5: Colour-Blind Local Colour

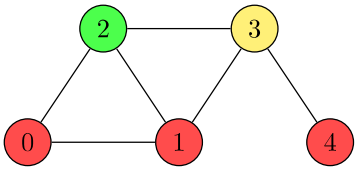
In Figure 10, we can see a visualisation of the colouring process. For the purpose of the visualisation, the nodes are visited in order. In the actual algorithm, the order the nodes are visited in is still sequential, but randomised. We use a maximum number of colours of five (the number of nodes) for the colouring. The following mapping of integers to colours is used in the diagram: 1 - red, 2 - green, 3 - yellow, 4 - orange, 5 - pink.



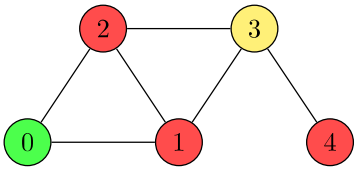
(a) Iteration 1; nodes are initialised with their degree



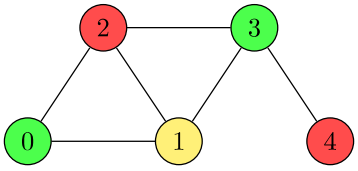
(b) Iteration 2; conflicting nodes are decremented



(c) Iteration 3



(d) Iteration 4



(e) Iteration 5; no conflicting nodes remain - done!

Figure 10: Colour blind local colour visualisation (decrementing variant)

Because the algorithm does not know the colour of its neighbours, changing your colour when not in conflict risks increasing the number of conflicts in the graph, so the agent does nothing.

4.8.3 Minimum Local Colour

In this algorithm, we are expanding the local knowledge to include the colour of each neighbour. This means that the algorithm is capable of picking the "lowest" colour that it possibly can in order to colour the node and is also capable of avoiding picking a new colour which will cause new conflicts.

The algorithm is depicted in Algorithm 6. The major new inclusion is an array of length k which consists of boolean values (1 or 0) indicating whether the corresponding colour is present in the graph. We can call this array the vector $\vec{C}$, which represents a truth vector indicating whether the colour represented by each index is in the graph. In this case, the "graph" is limited to the neighbours of the agent. The algorithm then sets its own colour to true (1) and iterates over the vector to see if there is a colour "lower" than its current colour that it could assign to itself.

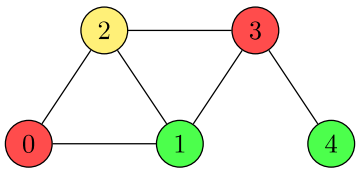
The conditions of this algorithm ensure that there are never any conflicts in the graph. There is a variant which is not used in the experiments, in which you assign the node the colour m if it fails to find a lower colour, hence either assigning it the max colour, k or its current colour.

Input: an agent a and a max colour k
Output: the number of changes made to the graph, x

$x \leftarrow 0$
$\vec{C} \leftarrow \text{coloursInGraph}(\text{neighbours}(a))$
$\vec{C}[\text{colour}(a)] \leftarrow 1$
$m \leftarrow \text{colour}(a) > 0 \ ? \ \text{colour}(a) : k$
for $c \leftarrow 1$ **to** m **do**
  if $\vec{C}[c] = 0$ **then**
    $a \leftarrow c$
    $x \leftarrow 1$
  end
end

Algorithm 6: Minimum Local Colour

A visualisation of the algorithm is shown in Figure 11. The same setup is used as the visualisation of the colour-blind algorithm in Figure 10.



(a) Iteration 1; nodes are coloured with an initial, lowest possible colouring - done!

Figure 11: Minimum local colour visualisation

This illustration depicts the excellent efficiency of the colouring algorithm in finding a feasible colouring; it takes only a single iteration to achieve the same results as the centralised benchmark. In the actual implementation, there are a number of subsequent iterations where agents will check if they can improve the colour on their node. In this case, that will not happen, given that reducing the colour on any node would cause conflicts.

4.9 Movement Algorithms

Movement algorithms allow the program to allow the agents to move around in the graph. This obviously greatly increases the complexity of the colouring process; not every node requires an agent, yet the entire graph will eventually be coloured.

The concept for these kernels is that by reducing the number of workers in the graph, you can improve the efficiency of the overall algorithm, as each iteration will do less work, but the work will be more targeted. This of course depends on the initialisation of the agents, which is done randomly. A targeted means of selecting nodes on which to place agents could potentially result in improved performance.

Currently, there are two different simple movement algorithms which have significantly different behaviours. These are the random movement algorithm and optimal movement algorithm. The term *optimal* is used loosely here, but the concept is that the move is optimal given the distributed nature of the system and the limited amount of knowledge the agents have about their surroundings.

The algorithms are laid out in Algorithms 7 and 8 respectively. All movement algorithms take the input of an agent and the number of moves the agent will make per call of the kernel. They return the new location of the agent in the graph.

Input: an agent a and number of moves m
Output: the new location of the agent a'

for $i \leftarrow 0$ **to** m **do**
| $a' \leftarrow \text{neighbours}(a)[\text{rand}() \bmod \deg(a)]$
end

Algorithm 7: Random Move

4.10 Dynamic Graph Algorithms

Dynamic kernels allow for the implementation of algorithms which modify the graph's topology in some way. This allows for the constraints represented in the graph to be changed and can be combined with colouring algorithms in order to produce more complex societal behaviour. The main difference between any of the algorithms is the condition by which the algorithm makes a change to the graph. Two detailed examples are listed in Algorithms 9 and 10. These are the possibly remove edge and possibly remove node algorithms respectively. These serve as good examples, since the only major difference between these algorithms and any other dynamic algorithms are the conditions by which the modification is made.

Other dynamic kernels include removing orphan nodes from the graph, removing nodes with more than two thirds of their neighbours conflicting and incrementally reducing the number of agents operating on the graph.

Modifying data structures can be a tricky thing to correctly implement in any programming situation. In this instance, it was a major challenge to ensure that it was done properly and correctly. There are a number of pull requests recorded in the repository relating to implementing dynamic graphs and fixing issues that arose later.

The implementation required two core functions: removing and adding edges, and removing and adding nodes. Edge operations are generally quite straight forward, as these are quite similar

Input: an agent a and number of moves m
Output: the new location of the agent a'

```
for i ← 0 to m do
    if deg(a) = 0 then
        break
    else if numUncolouredNodes(neighbours(a)) > 0 then
        for nb ← 0 to deg(a) do
            if colour(neighbours(a)[nb] = 0) then
                a' ← neighbours(a)[nb]
            end
        end
    else
        n ← neighbours(a)[0]
        for nb ← 0 to deg(a) do
            if colour(neighbours(a)[nb] > colour(n) then
                n ← neighbours(a)[nb]
            end
        end
        a' ← n
    end
end
```

Algorithm 8: Optimal Move

Input: a graph G, an agent a

```
if deg(a) > 0 AND rand() mod 1000 == 0 then
    removeEdge(a, neighbours(a)[rand() mod deg(a)])
end
```

Algorithm 9: Possibly Remove Edge

Input: a graph G, number of nodes n_G, an agent a, the list of agents $\vec{A}$, the number of agents n_A

```
if rand() mod 1000 == 0 then
    removeNode(G, a)
    n_G ← n_G − 1
    removeAgent(A, a)
    n_A ← n_A − 1
end
```

Algorithm 10: Possibly Remove Node

to the initialisation of the initial graph before the colouring process even begins. However, one instance of this being not so straight forward was the implementation of a function to make a node an orphan. Initially, this was done by simply calling the `removeEdge` function on each neighbour the node has. However, this was incorrect, due to the fact that it essentially resulted in a process of looping over a list that was being modified. Instead, the program iterates over each neighbour and removes the soon-to-be-orphan node from its list of neighbours. When this operation is complete, only then is the orphan node's neighbour list wiped.

Nodes, on the other hand, presented a number of programming challenges, as often the graph structure had to be modified in the middle of the colouring process; a risky procedure. However, due to the sequential nature of the colouring process over the set of agents, the structure can be safely modified. The implementation of this ultimately used references to the graph to modify the data structure directly. There was an alternative to using this process in the utility functions, but it was decided that the code was actually much cleaner when the structure was directly modified. All the algorithm has to do is make the node an orphan (as discussed above) and then free the node's memory and remove it from the graph data structure which is a relatively simple operation.

The implementation is ultimately very solid and allows the programmer to implement any dynamic kernel in a straightforward manor and surprisingly simple code through the use of the utility APIs.

4.11 Bad Actor Algorithm

The concept of the bad actor algorithm is to implement a system in which the nodes must colour the graph while simultaneously combatting an agent which is acting with the opposite intent; to colour the nodes with the colour which will cause the greatest number of conflicts.

There is the basis for an entire separate project in this concept, and it is discussed further in Section 7. Nevertheless, the algorithm to implement this system within the confines of a program intended for distributed and cooperative nodes is included (at a high level) in Algorithm 11. The kernel is actually a colouring kernel, but adds some additional global knowledge on the implementation side in order to allow the kernel to keep track of certain important information, such as which agent is the bad actor. The reason the number of kernel calls is measured against the maximum colour is because this value is generally the same as the number of agents (one per node).

Input: an agent a and a max colour k
Output: the number of changes made to the graph, x

if *bad actor not selected* **then**
  possibly choose a as the bad actor
end

if a *is the bad actor* **then**
  $a \leftarrow$ most common colour in locality
else
  a votes for the neighbour it thinks is the bad actor
  $a \leftarrow$ colouring by minimum colour
end

if *number of kernel calls* $=$ *max colour* **then**
  tally the votes
  if *the node with the most votes has more than* 1 *vote* **then**
    make it an orphan node
  end
end

Algorithm 11: Bad Actor Algorithm

4.12 Saving Experiment Results

Experiment results are saved via outputting the results to two different `csv` files. One of the files records the conflicts at iteration i for the colouring process, the other file saves the output data from the program including information such as the number of colours in the graph, the number of iterations, run time, etc.

The graphs and results in Section 5 are all computed via the Microsoft Excel/LibreOffice Calc software. Saving the data in columns is therefore preferable, and the code used to repeatedly

calculate the number of problems (conflicts and uncoloured nodes) at a certain iteration and then write the data to the file was all custom written for this application. Due the complexity and computational time required to calculate the number of conflicts in the graph, the time taken to do this is actually subtracted from the overall running time before it is output by the program. This design choice was made in order to get a sense of the actual *distributed* colouring time, rather than including centralised data collection in the reported computational time.

Outputting an array of integers into a column of `csv` data, which needs to be appended to the file in a way which allows for multiple runs to be included in the same file, requires a number of specific algorithmic steps. A high level version of the algorithm is listed in Algorithm 12.

Input: a file f
Output: f with the new data appended

read existing line from f

strip the newline

append the new data

rewrite the line to f

fill in empty columns in the previous columns

append the remaining data to the end of the empty columns

Algorithm 12: High level saving algorithm

Essentially, this algorithm will write the new data to the `csv` file, while accounting for the cases where the new data column is longer or shorter than the data previously written to the file.

4.13 Visualising the Graph

Visualising the graph colouring process can be a helpful means to understanding and interpreting the results from the colouring process. However, this is a complex domain in and of itself. Finding a way to properly visualise a graph of 1000 nodes, or indeed anything beyond 20 nodes, can be difficult when any given node could be connected to hundreds of others. Additionally, it can add much complexity at development time, which must be committed to specifically visualisation, which is complex when writing the program in a low-level language such as C.

The solution in this project, inspired by the distributed basis of the research area, was to simply visualise one node at a time in the terminal window. The program enters an interactive mode in which the user can execute commands in order to explore and view different parts of the graph and see for themselves how the graph was coloured by the program.

By default, the program will start by showing the user the node with the highest degree. The user will see the node itself, and three of its neighbours; the neighbours with the lowest and highest ids, and the node with the lowest id plus one. The user can then traverse the neighbours of this node, or alternatively jump to any other node in the graph.

This mode also grants the user the ability to sever any connections between nodes, and then rerun the colouring algorithm to see if removing that constraint has reduced the number of colours required to colour the graph. When rerunning the program, it will use a more distributed approach described in Section 4.14.

4.14 Traversing the Graph

When colouring the graph in a distributed way, it is conceptually assumed that each node is coloured independently, however, this is not entirely true, as only in an environment which is perfectly parallelised, where every node is processed on its own CPU core at the same time, could this actually happen. This is understood to not be the case in this program. Each node is processed sequentially, so the order in which this happens is inevitably going to have an effect on the colouring that the graph receives.

The primary process for colouring the graph simply involves looping over each node in the graph, applying the kernels discussed above over the course of a number of generations. The order in which the nodes are processed is randomised in two different ways. Firstly the graph is randomly generated, so the way in which the first processed node sits within the context of the topology of the graph is entirely unknown. Additionally, the graph is not directly interacted with, instead agents are placed on the graph and the positions of these agents is randomised on every run of

the program, such that the order in which the randomly connected nodes are processed is itself randomised.

In experimentation, the results are taken over the course of multiple runs on the same parameters. This should mean that any outlier topologies or outsized effects of an unusually impactful ordering of nodes has on the graph is alleviated through repeated randomisation. As such, we can be confident that we are considering results related to the *parameters* we are running the program with, not a specific graph.

However, while this method is technically a distributed process, one of the main advantages of using a distributed approach should be a reduced amount of processing. Say we colour the graph, finish the process and *then* we change the constraints in the graph. We should not have to iterate over all of the nodes in order to investigate whether the constraint change has significantly impacted the graph. Instead, we want to first consider the nodes which have been directly affected, then incrementally expand out if we made a change to the colour of that node. The algorithm for this is listed in Algorithm 13, but essentially this can be achieved by adding the affected nodes to a queue, then repeatedly adding node neighbours to the queue if we make changes to any node. The algorithm can then simply be run until the queue is empty.

Input: a graph G, two initial nodes, s_1 and s_2, a colouring kernel K
Output: the coloured graph G'

$Q \leftarrow$ empty queue
$Q \leftarrow s_1, s_2$
$|Q| \leftarrow 2$
while $|Q| > 0$ **do**
  $x \leftarrow K(Q[0])$
  if $x > 0$ **then**
    pushToQueue(neighbours($Q[0]$))
    $|Q| \leftarrow |Q| + \deg(Q[0])$
  end
  popFromQueue($Q[0]$)
  $|Q| \leftarrow |Q| - 1$
end

Algorithm 13: Path Colour

4.15 Utility Functions

This sub-section gives an overview of some of the more notable utility functions that are implemented in the program.

4.15.1 Copy the Graph

Copying the graph is an important operation, given that it is preferable to modify a copy of parameter values but we are passing a reference to the graph between most functions. As such, the graph is copied at certain important checkpoints in the program, such as at the beginning of any colouring function.

The process of copying the graph, given its unique recursive structure, is a simple but specific process. The algorithm is illustrated in Algorithm 14. At a high level, the nodes can be copied in a very straight-forward manner, as you might expect. The neighbour arrays of each node must also be updated, however, which is done by utilising the id fields of the nodes to jump between the indexes of each graph structure. This is possible due to the fact that the ids are assigned at initialisation based on the position in the array. This also gives rise to the property that the graph structure is sorted in ascending order by id (recall that the graph is coloured randomly, but by a separate list of pointers; the graph is never rearranged). However, because we could be dealing with a dynamic graph from which any number of nodes may have been removed, there are some steps we have to take to be sure we are indexing to the correct node in the original graph.

The utility API provides a function to locate a node with a certain id in a graph, but in order to keep the copy function fast, this is not used in the copy function. This is because this function is a linear search, and is intended for cases such as path colour (Algorithm 13) where we are given the node and need to find its location in a graph that has already been copied. In this case, this is too slow.

Consider the case where the last node in a graph of n nodes is removed. If we were to check the position of each node in the graph given the id, we would be looping over the graph $n-1$ times, completing $\sum_{i=1}^{n-1} i$ checks. Instead, when we find the graph has been modified, we can loop over the graph once, calculating the differential between node id and position for each id and save this data to an array. Now, we can simply look up the pre-calculated differential and subtract this value from the neighbour id in order to locate the new neighbour pointer in the copied graph.

Input: a graph G and the number of nodes n
Output: the copied graph G'

$G' \leftarrow$ empty array of node pointers
$D? \leftarrow n - ((id(G[n-1])) + 1) \neq 0$
for $i \leftarrow 0$ **to** n **do**
  $G'[i] \leftarrow G[n]$
  if $D?$ **then**
    $\vec{d}[id(G[i])] \leftarrow id(G[i]) - i$
  end
end
for $i \leftarrow 0$ **to** n **do**
  for $j \leftarrow 0$ **to** $\deg(G'[i])$ **do**
    $id_j \leftarrow \text{id}(\text{neighbours}(G[i])[j])$
    $\text{neighbours}(G'[i])[j] \leftarrow G'[id_j - \vec{d}[id_j]]$
  end
end

Algorithm 14: Copy Graph

In Figure 12, we can see what these differentials look like in practice when five nodes have been removed from a graph which originally had 20 nodes. The zeroes in between the increasing numbers are the nodes that have been removed. We can tell from this data that the original list which nodes have been removed, and see how this information could be used in order to find the new positions of the nodes.

4.15.2 Free Graph Memory

Freeing the graph memory is a relatively straightforward operation, but requires a custom function in order to do it correctly. The algorithm is illustrated in Algorithm 15.

Input: a graph G, the number of nodes n

for $i \leftarrow 0$ **to** n **do**
  $\text{free}(\text{neighbours}(G[i])$
  $\text{free}(G[i])$
end
$\text{free}(G)$

Algorithm 15: Free Graph

4.15.3 Find Colours in Graph

Finding the number of colours in the graph is an important operation. Recall that one of the advantages of the structure of the program and `node` data type is that we can use the same functions on the whole graph or any individual locality. As such, this function is used primarily in the minimum local colour algorithm to find which colours can be used to improve the state of the node.

The algorithm is relatively simple, but the important element is the way the colours are communicated. The chosen data structure was a truth vector, i.e., a vector of boolean values where each value indicates whether the colour relating to index i is in the graph or not. This is similar to the way the differentials are stored in the copy graph function (Algorithm 14).

The algorithm is illustrated in Algorithm 16.

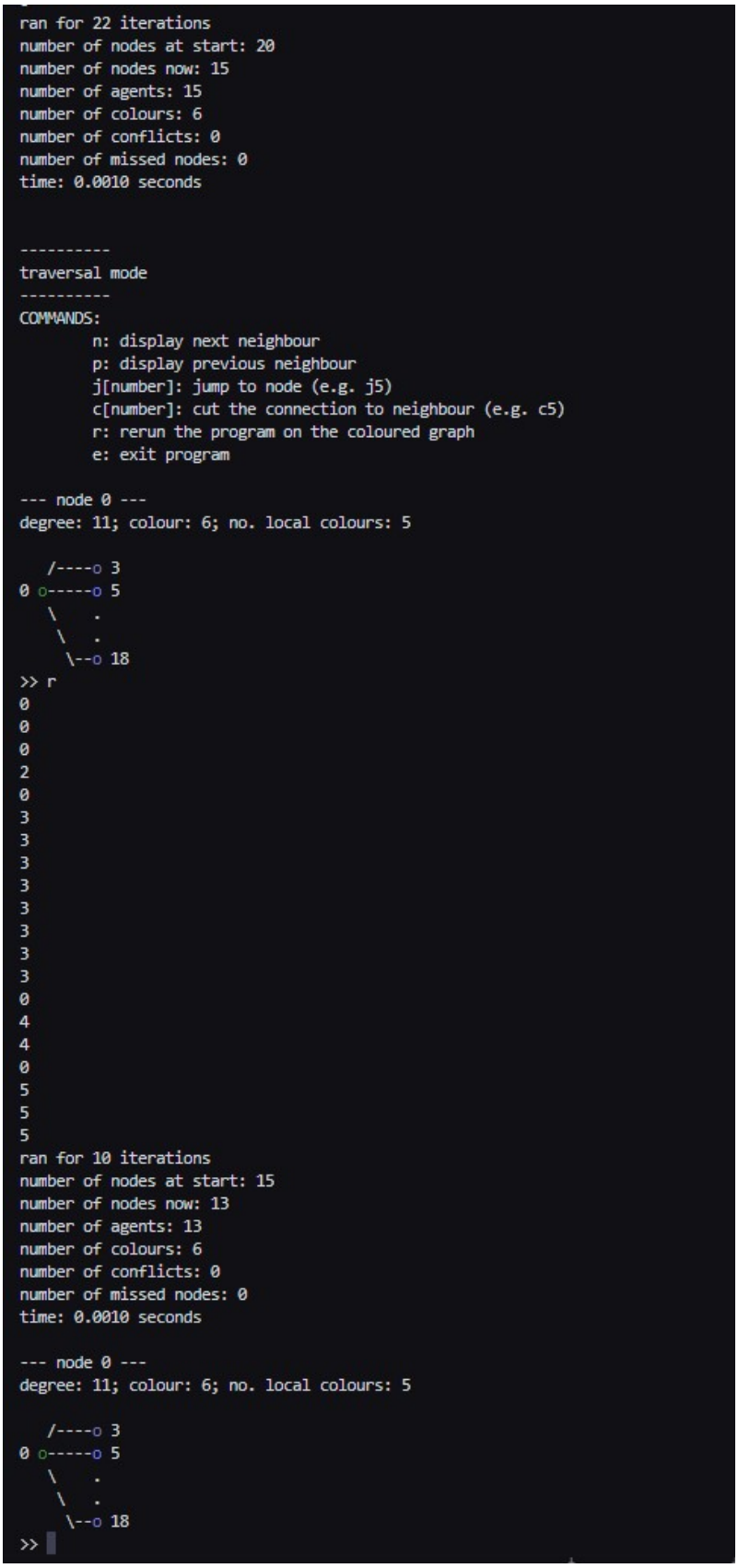


Figure 12: Index differentials after nodes have been removed

Input: a graph G, the number of nodes n and a max colour k
Output: a colour truth vector, $\vec{c}$

$\vec{c} \leftarrow \vec{0}$ of length k
for $i \leftarrow 0$ **to** n **do**
  $\vec{c}[\text{colour}(G[i])] \leftarrow 1$
end

Algorithm 16: Find colours in graph

4.15.4 Find the Number of Colours in the Graph

A similarly centrally important operation, finding the number of colours in the graph allows us to report on the performance of the algorithm. It is arguable that this is the most important metric we have in the program, given that the agents are meant to find the best colouring they can over a number of iterations.

To count the number of colours, we can simply check the colour on each node, while ensuring we do not count duplicates, as shown in Algorithm 17.

Input: a graph G, the number of nodes n and a max colour k
Output: the number of colours, x

$x \leftarrow 0$
$\vec{c} \leftarrow \vec{0}$ of length k
for $i \leftarrow 0$ **to** n **do**
  $c \leftarrow \text{colour}(G[i])$
  if $\vec{c}[c] = 0$ **then**
    $x \leftarrow x + 1$
    $\vec{c}[c] \leftarrow 1$
  end
end

Algorithm 17: Count the number of colours in the graph

4.15.5 Find the number of conflicts in the Graph

The opposite metric in the graph is the number of conflicts. This metric can essentially be considered to be the distance from a feasible solution. There are two important considerations to be taken into account in this algorithm:

- the colour 0 (uncoloured) does not count as a conflict
- the graph is undirected

We can solve these problems by firstly checking if the node's colour is greater than 0. The other can be solved by dividing the number of conflicts we count by two, since every conflict will be counted twice. This is clear from the listing in Algorithm 18.

Although integer truncation means that odd numbers will be reduced to the next even number. This is only a problem when the number of conflicts is 1, as it will be registered as 0, but given our experiments involve hundreds of thousands of conflicts, this inaccuracy is negligible. The number of conflicts is also not taken into account by the colouring algorithm, meaning that the inaccuracy will not result in an incomplete algorithm.

4.15.6 Select N Random Nodes from Graph

This is the operation used to select the agents with which we colour the graph. The structure of the algorithm means that we can use this to select any number of agents, which allows for both the selection of a number of agents less than the number of nodes, or the randomisation of the order of nodes when we place a static agent on every node.

It is a simple operation, the only consideration we have to make being that we must ensure we do not place two agents on the same node. This can be achieved easily by keeping track of the "used" node indexes.

The algorithm is illustrated in Algorithm 19.

Input: a graph G, the number of nodes n
Output: the number of conflicts, x

$x \leftarrow 0$
for $i \leftarrow 0$ **to** n **do**
  for $j \leftarrow 0$ **to** $\deg(G[i])$ **do**
    if *colour(G[i]) = colour(neighbours(G[i])[j]) AND colour(G[i]) > 0* **then**
      $x \leftarrow x + 1$
    end
  end
end
$x \leftarrow \frac{x}{2}$

Algorithm 18: Count the number of conflicts in the graph

Input: a graph G, the number of nodes n and the number of nodes to select n'
Output: a list of selected nodes g

$\vec{s} \leftarrow \vec{0}$ of length n
for $i \leftarrow 0$ **to** n' **do**
  while $\vec{s}[s] = 1$ **do**
    $s \leftarrow \text{rand}() \bmod n$
  end
  $\vec{s}[s] = 1$
  $g[i] \leftarrow G[s]$
end

Algorithm 19: Select N nodes from the graph

4.16 Testing Implementation

The design of the testing framework is such that the testing code will not be included in the main executable. The MakeFile will compile a new executable which makes use of a specific main function to run a predefined set of testing functions. While this is not the most extensible testing framework ever created, it works perfectly for the purposes of this application.

The tests are designed to either succeed or fail based on logic tests. There is a certain level of assumption that the code that is being tested is correctly implemented in such a way that it will not crash. The main target for the tests is the core logic of the application, as these are the parts of the application that are used to simplify the code in more important parts of the application, such as the actual colouring algorithms.

5 Experimental Results

All of the experiments are also recorded as shell scripts in the repository. All results are also saved as csv files.

5.1 Static Graphs

The experiments on static graphs consisted of running the colouring process on graphs generated with the Erdős–Rényi model of varying densities. All of the following experiments were run on graphs of 1000 nodes. Each algorithm had five runs each associated with a limit on k, the maximum number of colours: 2, 50, 500, 800 and an unlimited number of colours (practically 1000, the same as the number of nodes).

5.1.1 Random Local Colour

Using an unlimited number of colours with the random local colour algorithm sees the algorithm colour the graph with a very large number of colours, which does still increase over time. This is shown in Table 1. The benefit of this is that the conflicts are resolved very quickly (Figures 15 and 16), but obviously with a far from optimal colouring (Figure 13). In Figure 14, we can see how the algorithm converges quickly for all densities.

density	mean iter.	mean colours	mean time
0.1	11.7	651.6	0.00384
0.2	12.4	676.7	0.00741
0.3	13.3	700.7	0.01209
0.4	14.4	728.8	0.01613
0.5	15.3	754.5	0.02095
0.6	16.5	786.1	0.02466
0.7	17.9	819.1	0.03179
0.8	21.4	865.5	0.04104
0.9	29.8	921.7	0.05858
1.0	351.5	1000.0	0.70133

Table 1: Performance results for random local search

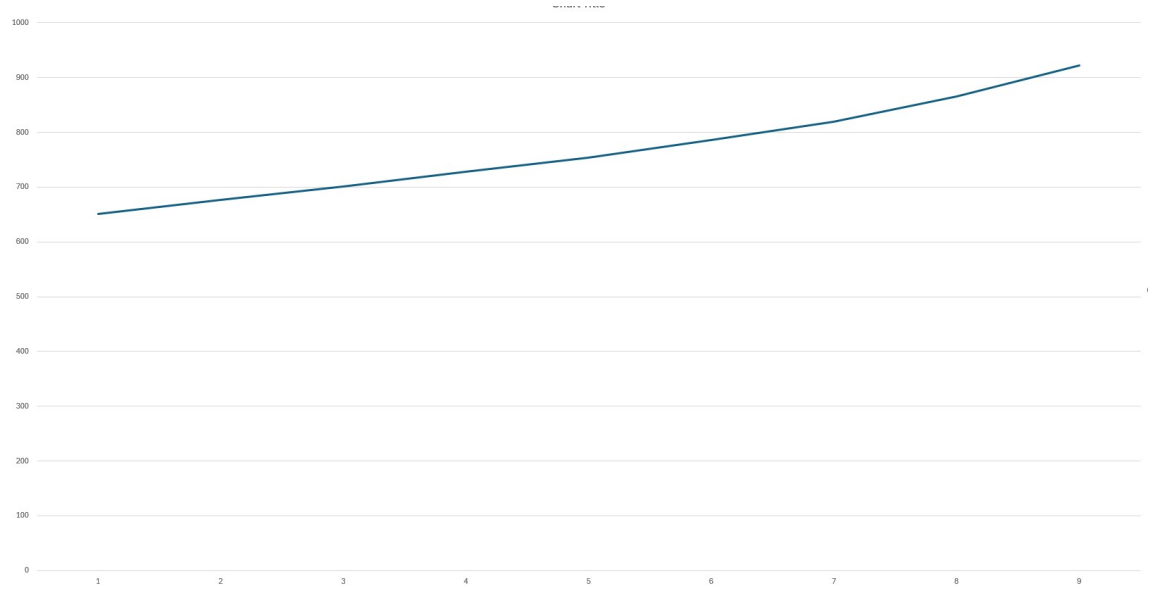


Figure 13: Number of colours used as density increases (no limit on colours)

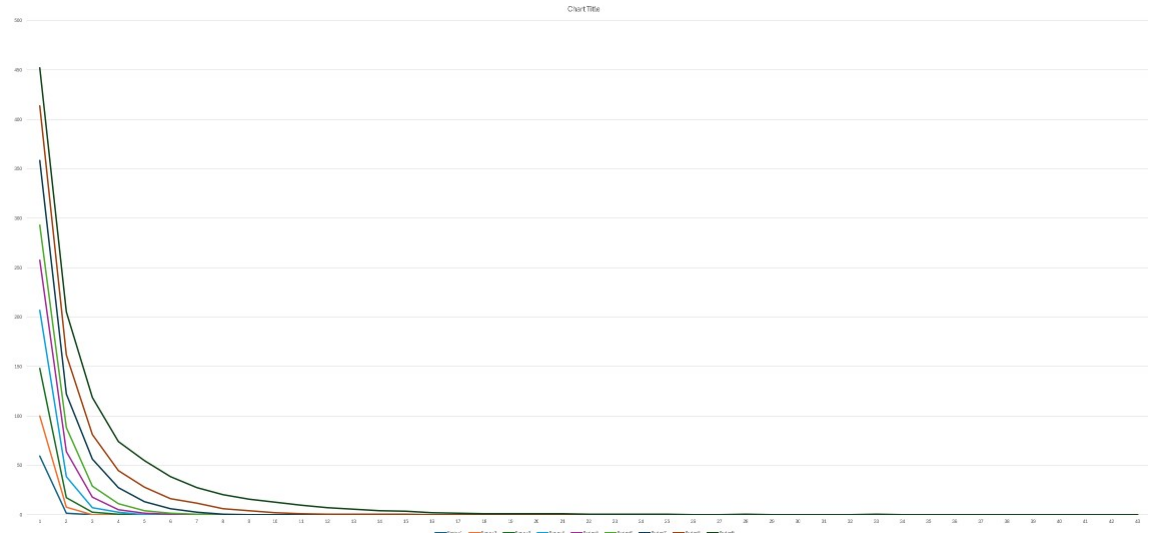


Figure 14: Conflicts over iterations for random colour

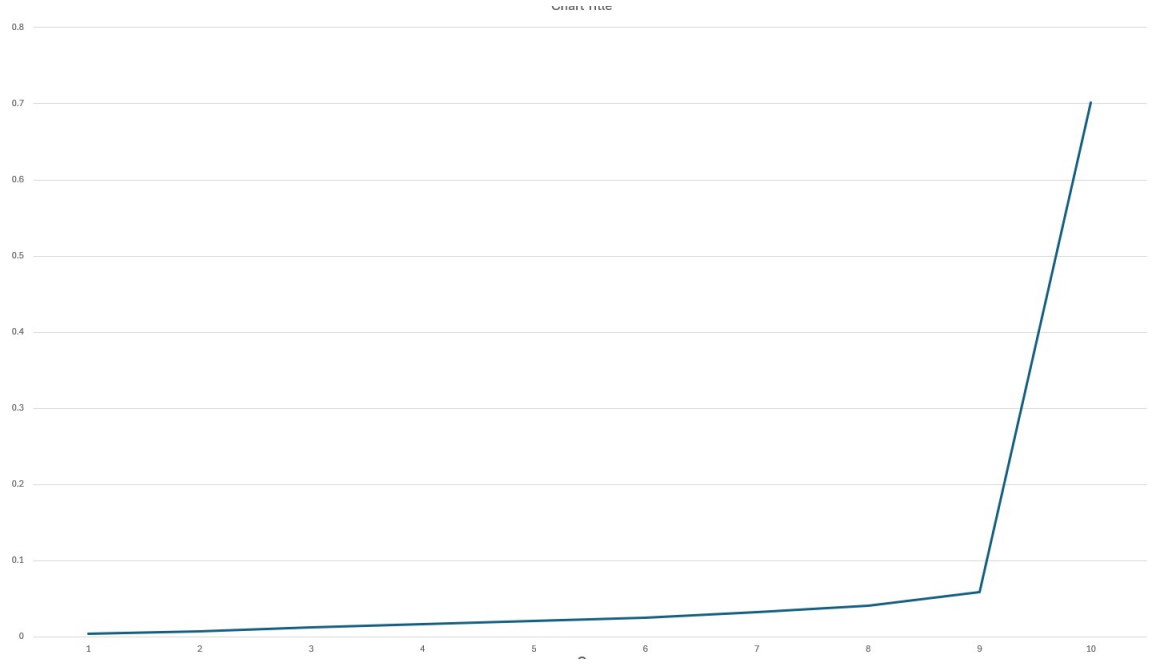


Figure 15: Time to colour graph as density increases

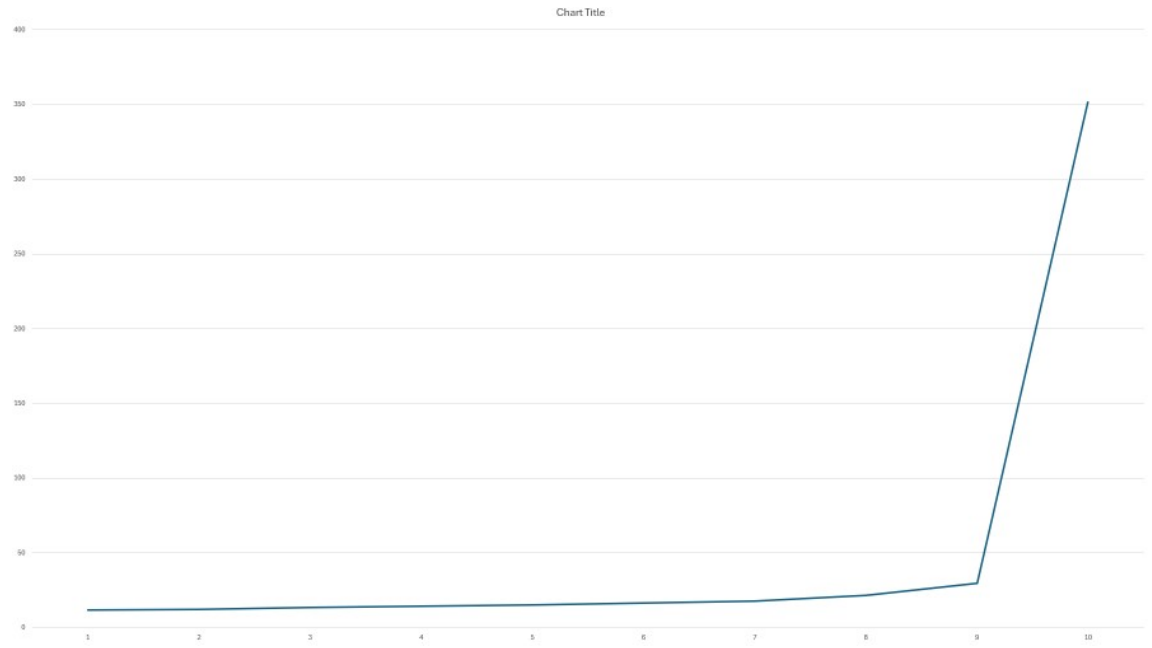


Figure 16: Number of iterations required as density increases

When limiting the colours used in the graph, we see that the algorithm was unable to colour the graph successfully at all before the number of colours begins to align with the average degree of the graph. The average degree in these experiments is roughly density times the number of nodes. For example, Figure 17 shows the results when colouring with 2 colours, which you would expect to fail. When we do get to the point where the number of colours is sufficient to easily colour the graph by randomly selecting colours, we can see the algorithm begin to succeed. It will then again begin to fail with more and more conflicts as the density surpasses this point. This is illustrated by visualising how points where the number of colours begins to level out (hit the maximum limit) and the number of conflicts becomes greater than zero are at the same density. In Figures 18 and 19, the blue lines represent the number of colours and the orange lines represent the number of conflicts as density increases. The data visualised in Figure 18 is listed in Table 2.

density	mean colours	mean conflicts
0.1	446.8	0
0.2	462.9	0
0.3	478.3	0
0.4	492.4	0
0.5	500	0
0.6	500	195.8
0.7	500	359.6
0.8	500	511.6
0.9	500	658.0
1.0	500	790.3

Table 2: Performance results for different probabilities when limited to 500 colours

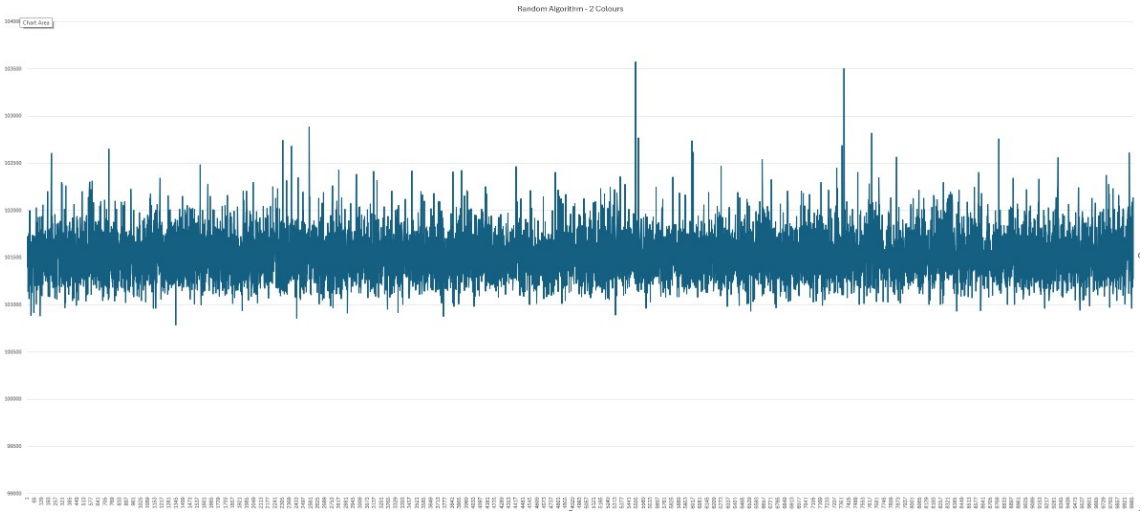


Figure 17: Random Kernel with Two Colours

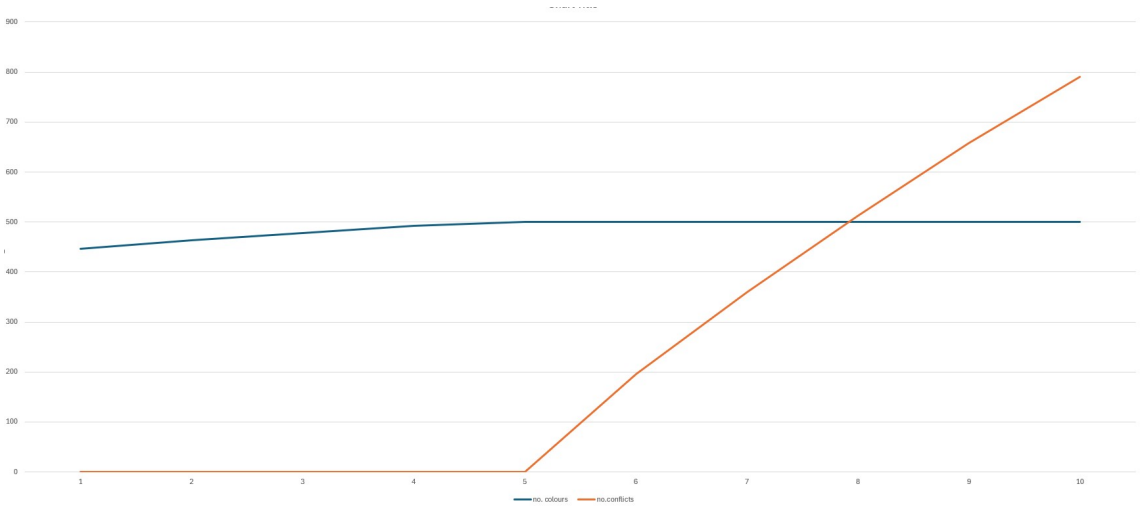


Figure 18: Colours (blue) and conflicts (orange) over density when limited to 500 colours

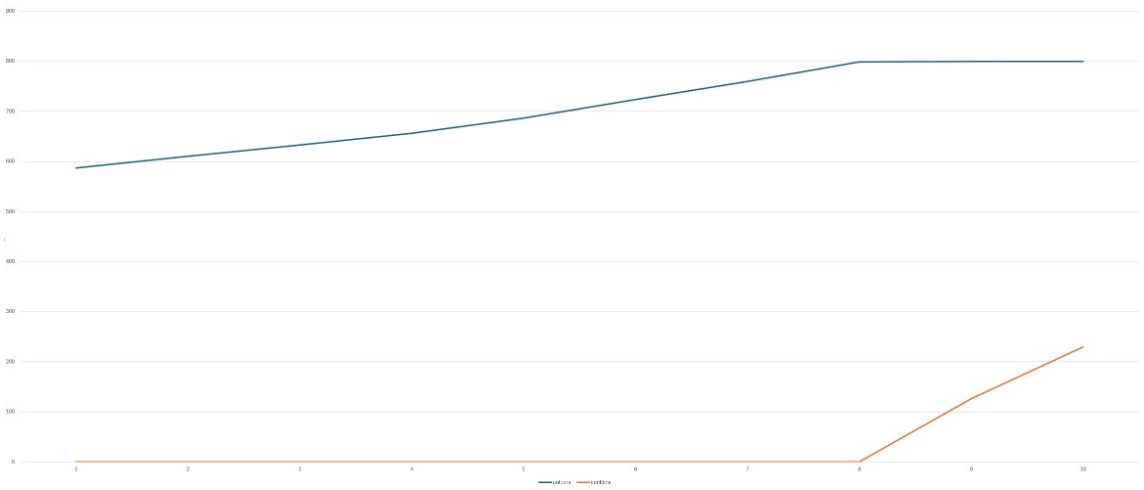


Figure 19: Colours (blue) and conflicts (orange) over density when limited to 800 colours

5.1.2 Colour Blind Algorithms

The intent of the colour blind algorithms is to investigate the ability of the program to colour the graph given the absolute minimum knowledge about the current state of the colouring process. As such, the agent knows only whether it is in conflict or not, but nothing further about the state of its locality. The only feasible change to make then, which has some more intent than to simply pick a random colour as above, is to "increment" or "decrement" the colour modulo k.

We would expect that this algorithm would have a similar effectiveness as that of the random kernel, as picking the next sequential colour provides no guarantee whatsoever that the new colour will not actually cause more conflicts than the last. Indeed, we can see in Figures 20, 22 and 23 how the algorithm does indeed manage to find a feasible colouring, but that it is far from optimal. It is evident that the number of colours tends to be around the same as the average degree of the graph, regardless of whether the algorithm increments or decrements the colour when in conflict.

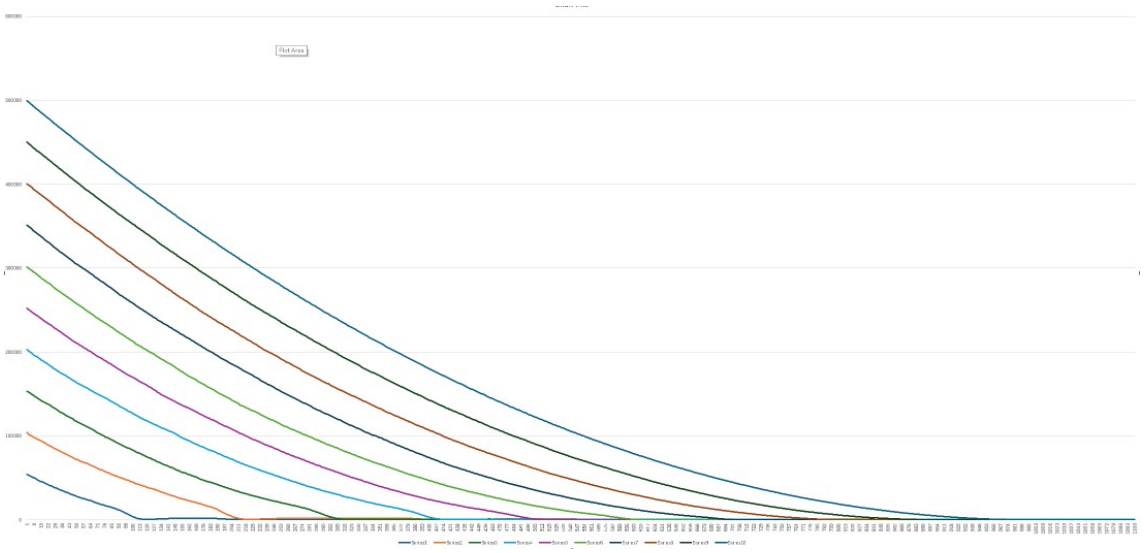


Figure 20: Conflicts over iterations (incrementing)

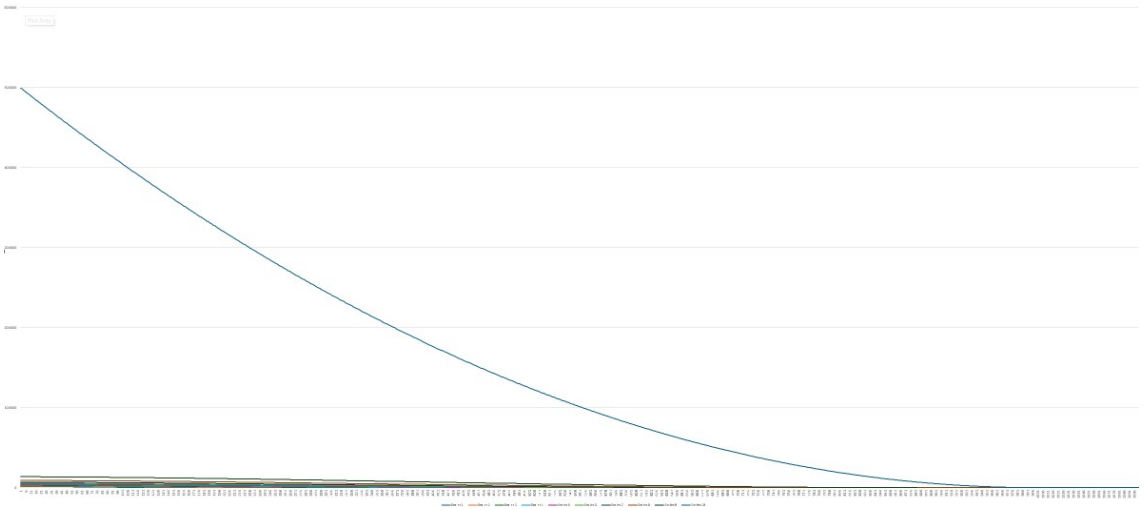


Figure 21: Conflicts over iterations (decrementing)

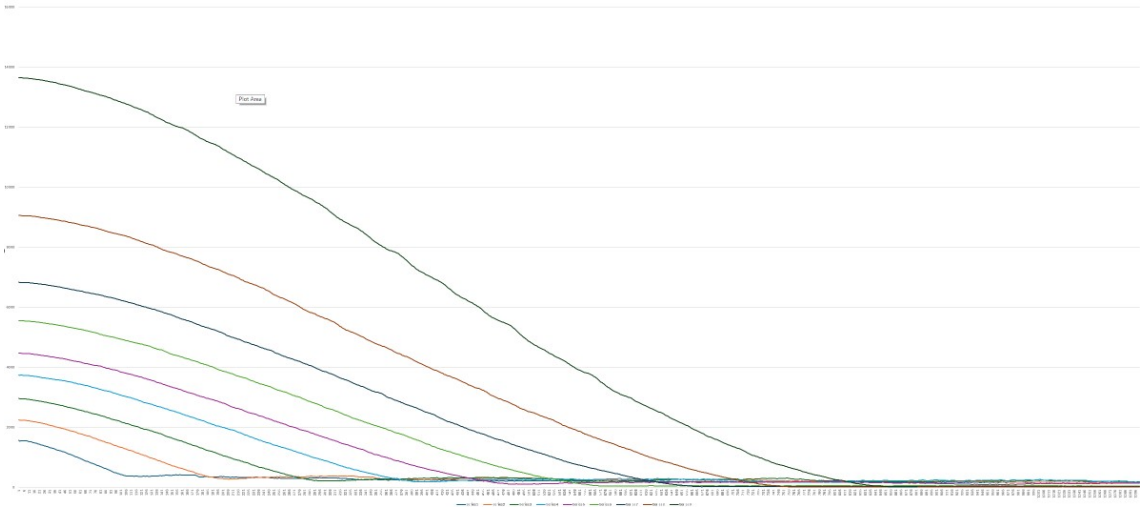


Figure 22: Conflicts over time excluding $p = 1$ (decrementing)

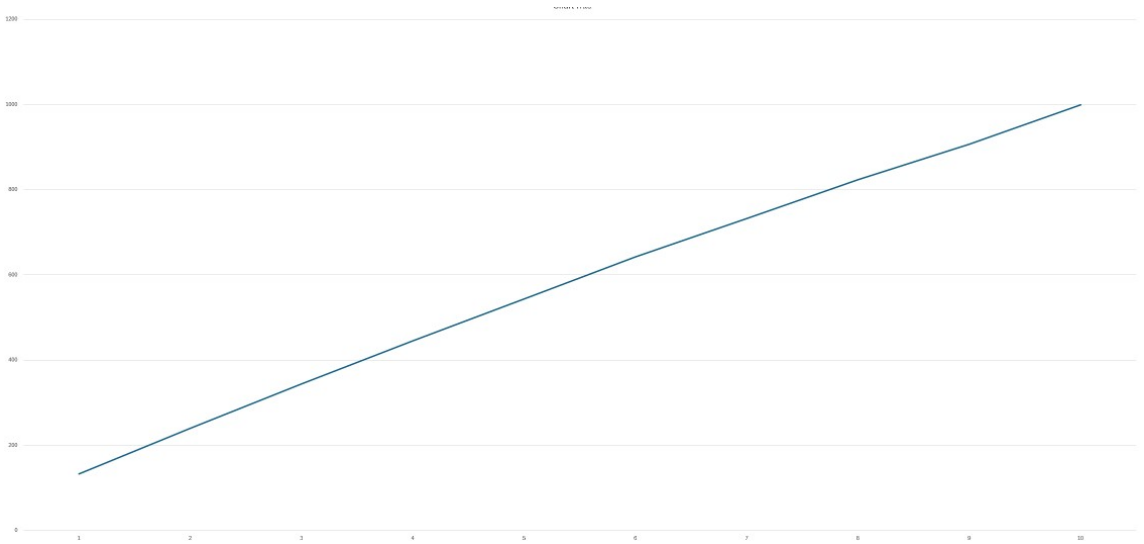


Figure 23: Number of colours over increasing density (incrementing)

The amount of time required to colour the graph is not similar to the random colouring at all. Indeed, Figure 24 presents what could be considered a strange set of results. The answer to this

puzzle is that when the density is low, it is easy for the algorithm to find a solution, when it is very dense, it also becomes easier, as it is easier for the algorithm to not find a solution. The challenge comes in the middle, where these is not enough density to effectively force many nodes to change at once, but also too much density to quickly converge on a solution.

The performance of the algorithms with an unlimited number of colours is listed in Table 3.

	incrementing			decrementing		
density	**mean iter.**	**mean colours**	**mean time**	**mean iter.**	**mean colours**	**mean time**
0.1	50000	132.8	9.51518	14576.2	137.5	2.9039
0.2	50000	239.3	18.05619	11104.8	243.7	4.32286
0.3	41964.6	343.7	23.82568	19963.6	347.9	11.60537
0.4	32237.8	445	24.42498	28540.4	448.7	22.14793
0.5	19883	543.5	18.4763	38658	547.5	38.50411
0.6	28363	641.7	32.08181	40452.6	646.7	46.7927
0.7	22244.7	731.9	31.53111	26482.9	740.6	37.21365
0.8	28647.8	824.1	45.58321	5690.9	824.9	8.86594
0.9	9181.6	907.2	15.71696	866.5	908.5	0.82399
1.0	50000	999	102.88701	50000	999	98.5688

Table 3: Performance results for different densities (incrementing vs decrementing)

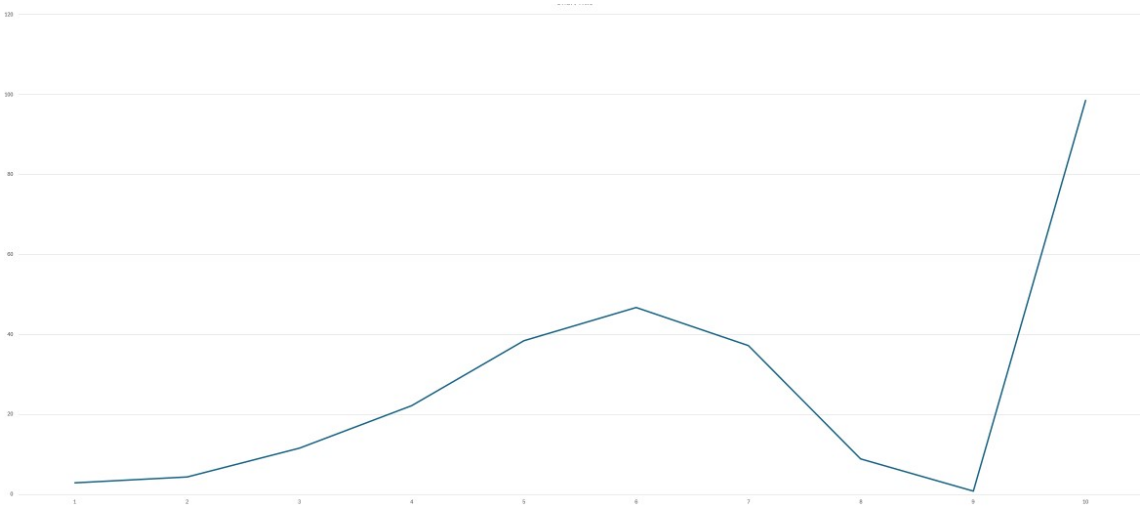


Figure 24: Time to colour graph as density increases (decrementing)

Far more interesting results emerge when we limit the number of colours the algorithm can use to colour the graph. The patterns that emerge here are strange, but consistent and relate directly to the density of the graph and the value assigned to k.

Figures 25, 26, 27, 28, 29, and 30 illustrate the process of colouring a graph with a density (node-neighbour edge connection probability) of 0.5 with increasing limits on the number of colours that can be used. From the results of Table 3 and Figure 23, we can expect that we will require roughly 500 colours to find a feasible solution for the graph. In other words, the first five limits should result in the algorithm being incapable of finding a feasible colouring.

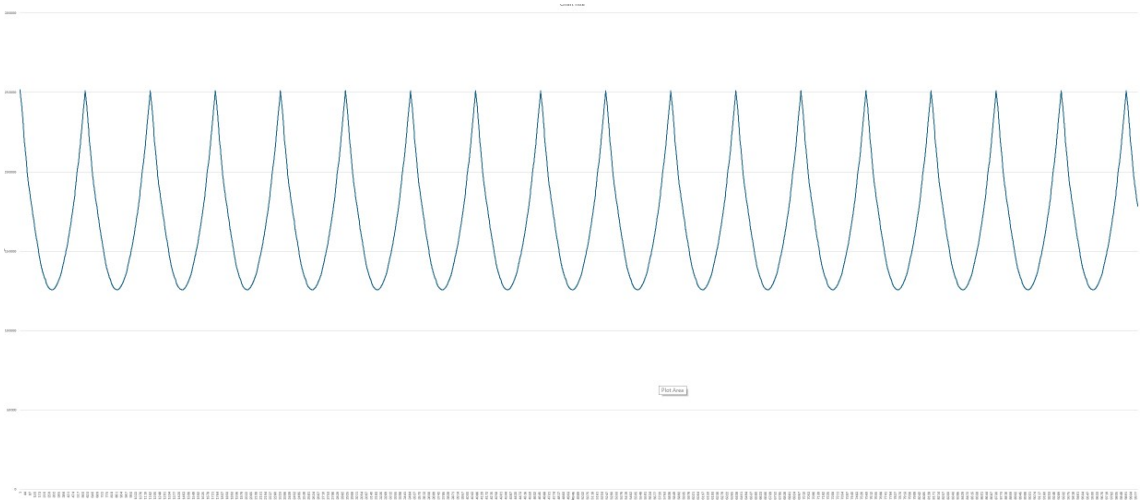


Figure 25: Conflicts over time when limited to two colours (incrementing)

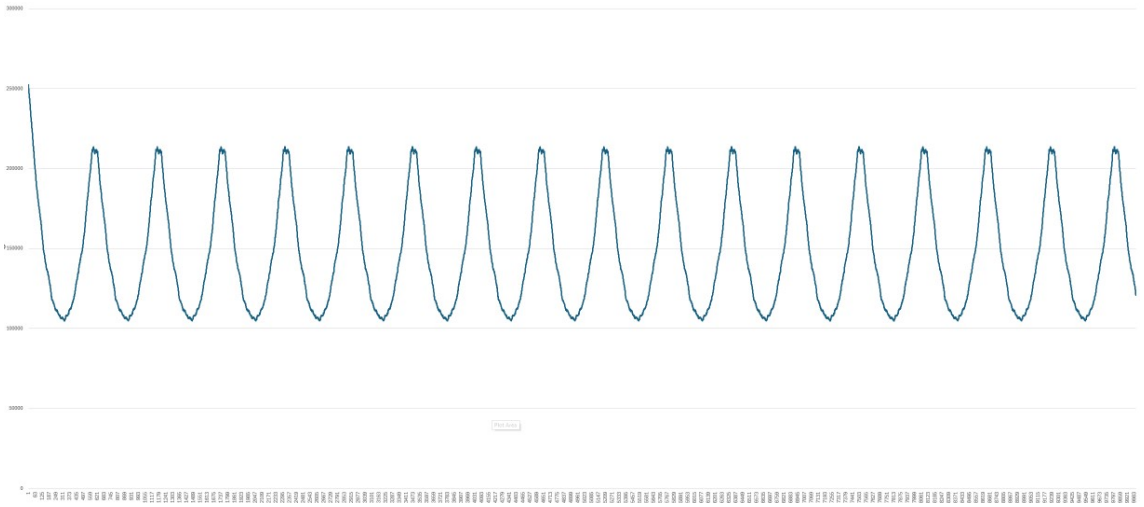


Figure 26: Conflicts over time when limited to 50 colours (incrementing)

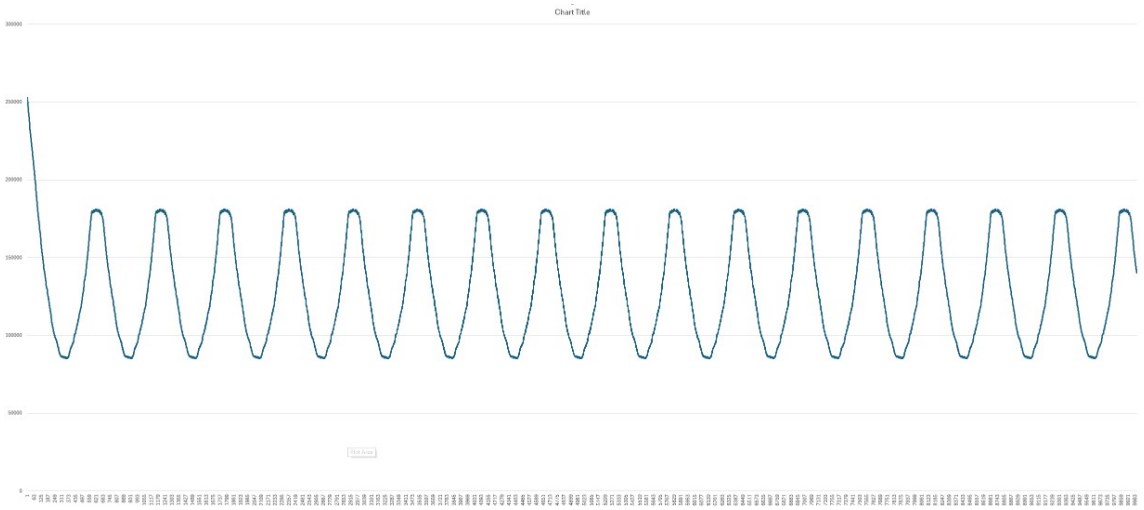


Figure 27: Conflicts over time when limited to 100 colours (incrementing)

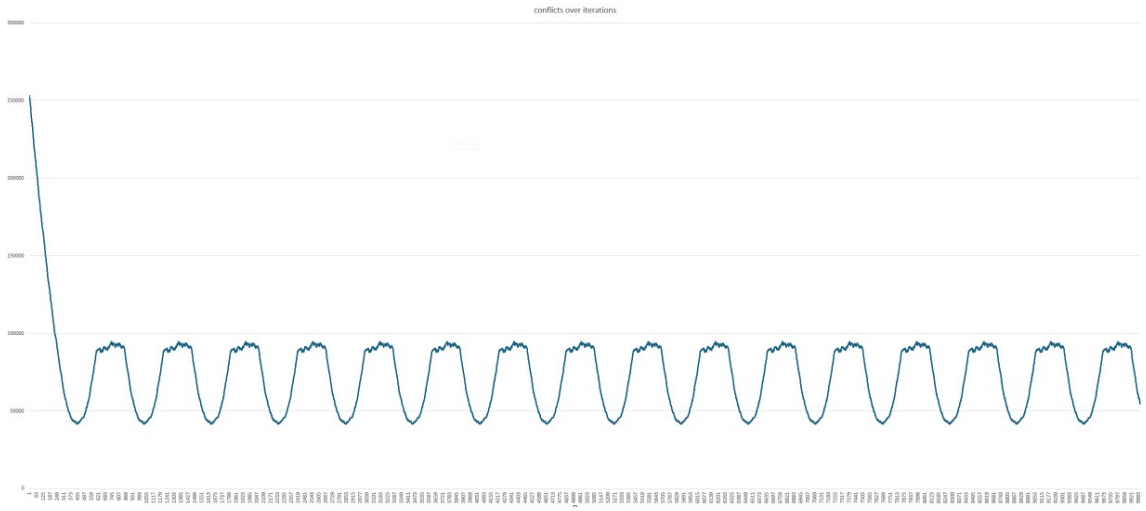


Figure 28: Conflicts over time when limited to 250 colours (incrementing)

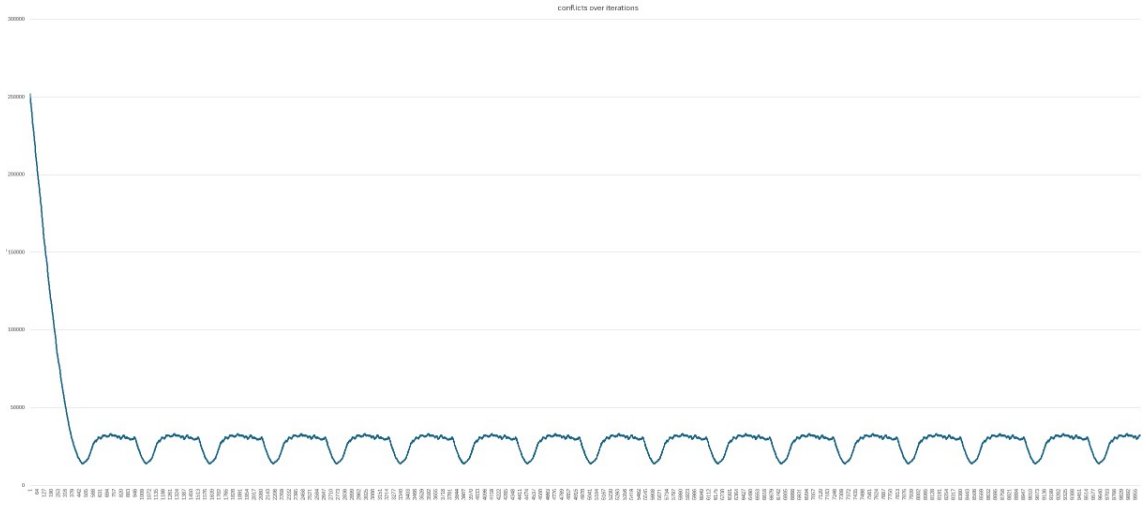


Figure 29: Conflicts over time when limited to 375 colours (incrementing)

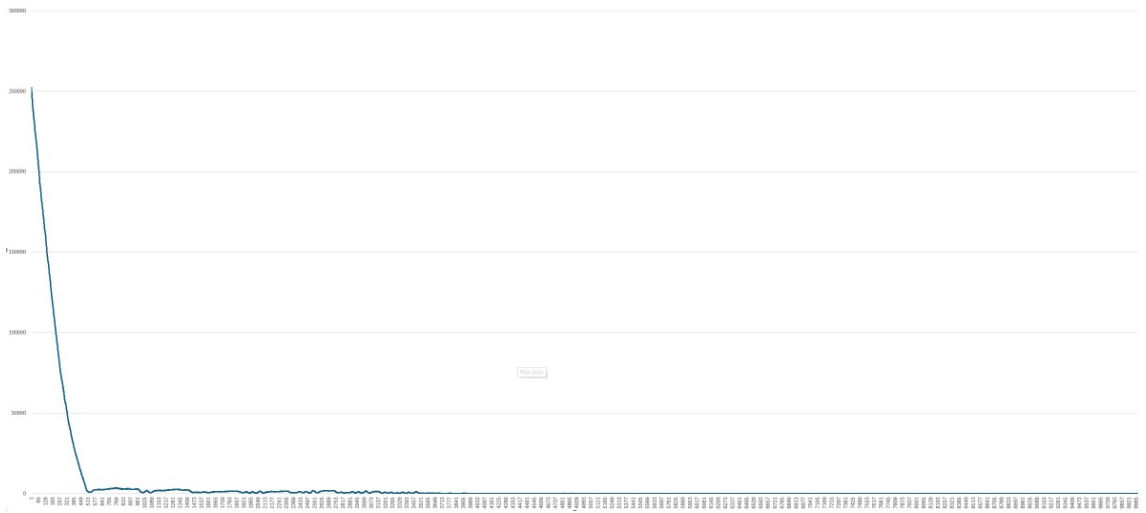


Figure 30: Conflicts over time when limited to 500 colours (incrementing)

This is what we see in the aforementioned set of graphs, as well as 31, 32 and 33. Interestingly, the results of 34 contradicts this hypothesis. We would expect decrementing the number of colours to result in more optimal colourings than when incrementing the number of colours, but this is

not the case. One potential explanation for this is the fact that the number of colours is reducing. Recall the centralised benchmark algorithm (Algorithm 2). In that algorithm, we use the smallest number of colours possible, but increase the limit whenever we hit it. The decrementing kernel could be described as doing the opposite. Where the incrementing kernel increases the limit for each node when a conflict is encountered, the decrementing kernel reduces it, implicitly making the problem harder to solve. This is evident in Figure 22, which shows how at most densities, the decrementing kernel struggles to converge on a solution, even with no limit on the number of colours.

Finally, an explanation for the way the flat-top pattern emerges in Figure 33 is required. This is a fully connected graph, which follows the wave patterns which previously arose, only now with a consistent number of conflicts between nodes for period of time. The confusing aspect of these results is the fact that the number of conflicts remains the same for some period of time, then goes back into a wave pattern. Curiously, we can also see this pattern emerge over time as density increases from 0.5 (Figure 32). The best explanation for this emerging pattern is that the progressively more connected graph serves to cause fixing conflicts in some nodes to be offset by other conflicts appearing in others. This reaches the apex of its effect when the graph is completely connected. In this case (1000 nodes and 500 colours), the number of conflicts starts at 499500, then reduces and returns to 125250, which is about one quarter of this value.

Figure 31: Conflicts over time for a graph limited to 500 colours and a density of 0.5 (decrementing)

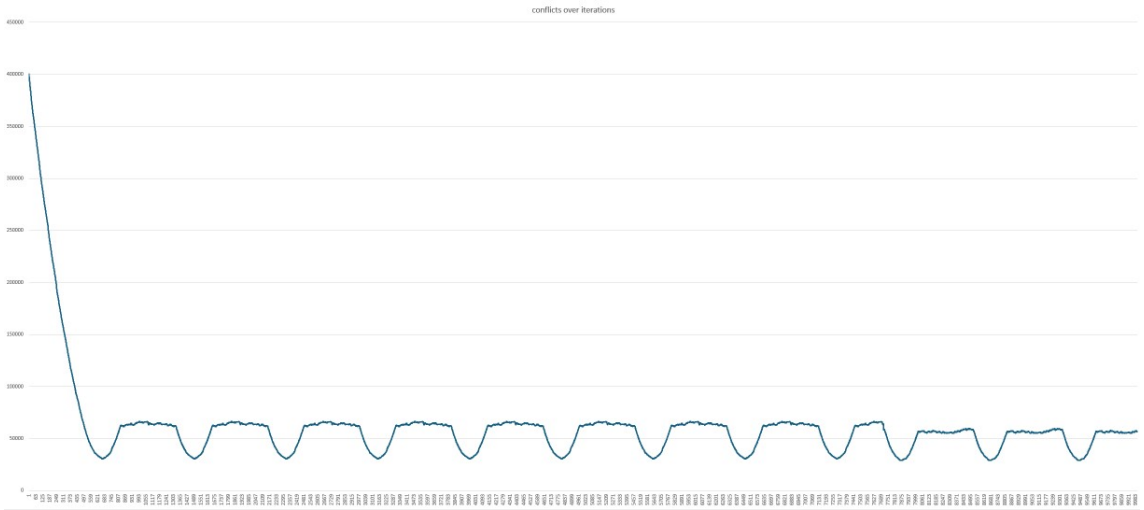


Figure 32: Conflicts over time for a graph with density 0.7 limited to 500 colours (decrementing)

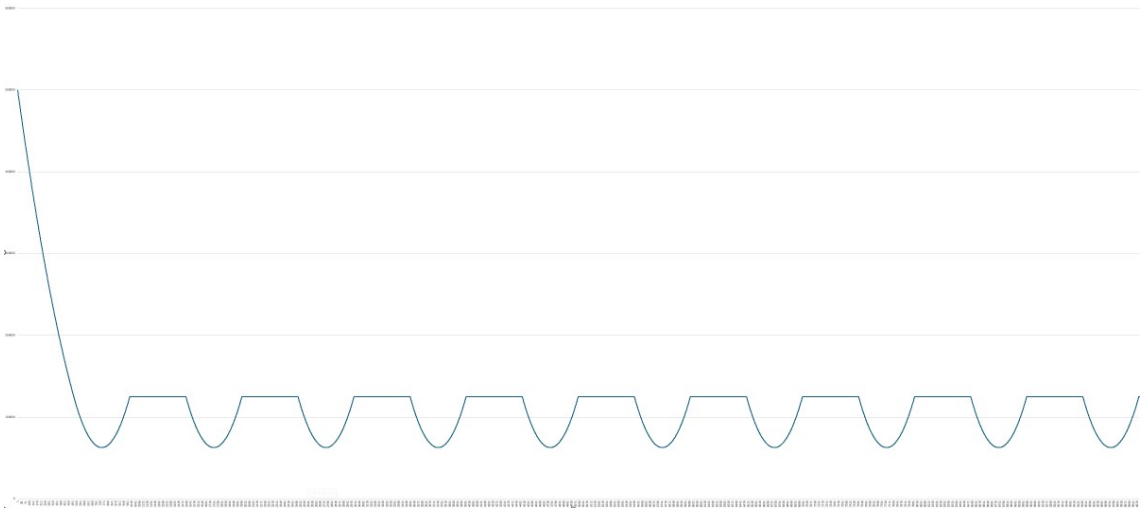


Figure 33: Conflicts over time for a fully connected graph limited to 500 colours (decrementing)

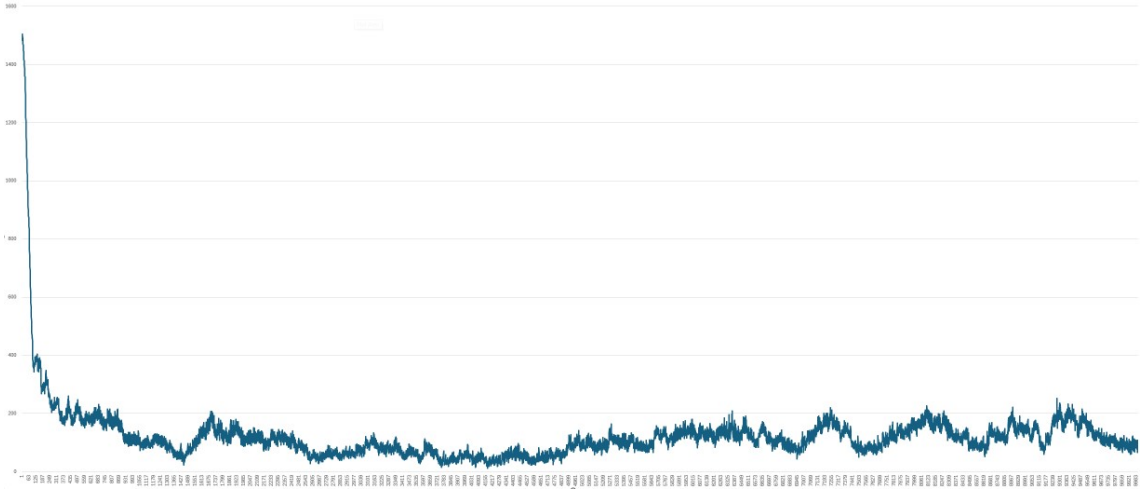


Figure 34: Low density graph limited to 500 colours with decrementing

If we consider that there are only 500 colours in the graph, then about half of the nodes should be in conflict at any time in this fully connected graph. This is $\frac{500 \times 500}{2} = 125000$, roughly the same as the number we see in the data. The decrementing algorithm will reduce the colour of a node if it is in conflict. This can explain the increasing wave pattern, since as the nodes which *must* conflict reach colour 1, they will all loop back around to 500, which other nodes will also have assigned as their colour already. Essentially, a certain portion of the nodes will always be in conflict, which will lead to the repeating wave pattern as this portion must forever cycle colours, causing a knock-on effect for other, currently non-conflicting nodes.

We can then extrapolate this explanation to account for the point where the maximum number of nodes are conflicting. As the conflicting nodes wrap back around to the limit, the "colour-space" reduces and more nodes conflict. This leads to a period of time where the colours align in the conflicting nodes. In other words, there is a portion of nodes that are conflicting, and there are a portion that are not, half of the conflicting nodes will reduce their colour (since this will fix the conflict for some other nodes as well), but this will mean a new set of nodes that already had the new "fix" colour applied have now joined the conflicting set. This will continue until the colour-space is loops back around and the reduction of a colour no longer means causing more conflicts. This is the reducing period. This continues until the minimum is reached and at this point the number of conflicts begins to increase again, until the conflicting set is maximised again.

This explains how the wave pattern emerges in a fully connected graph, but can be back-projected to other densities to explain how it emerges there, as it is clear to see how this would take effect in a more complex manner when changing the colour of each node has a varying impact.

In summary, the wave pattern emerges from periods of alignment in the colour-space between nodes and the flat-top emerges from this alignment being maximised and fixing conflicts leading

to more conflicts.

	incrementing		decrementing	
density	**mean colours**	**mean conflicts**	**mean colours**	**mean conflicts**
0.1	132.8	201.8	136.9	91.7
0.2	239.6	148.6	244.6	24.4
0.3	342.6	98.2	347.2	35
0.4	442.3	60.2	447.7	39.4
0.5	500	85.3	500	116.2
0.6	500	15951.3	500	16701.3
0.7	500	37176.5	500	38212.3
0.8	500	57461.2	500	59453
0.9	500	85414.5	500	85352.4
1.0	500	125250	500	125250

Table 4: Performance results for different densities with 500 colour limit (incrementing vs decrementing)

5.1.3 Minimum Local Colour

Unlike the other algorithms investigated in this project, minimum local search is capable of effectively and efficiently colouring the graph with a limited amount of local knowledge. Recall from Section 4 how an agent running minimum local search uses the colours in its locality in order to select improvements for itself, without the requirement to know which nodes have which colours. As such, there is no negotiation required, the node simply needs to look at its neighbours.

A side effect of the algorithm is that there are never any conflicts in the graph. This presents a challenge in visualising the graphs performance. In order to get around this, we can limit the number of colours in the graph, then increase the limit incrementally when the agents stop making changes for some number of iterations. This gives us Figure 35, where the stepped improvements in the reducing number of conflicts in the graph is evident for all graph densities.

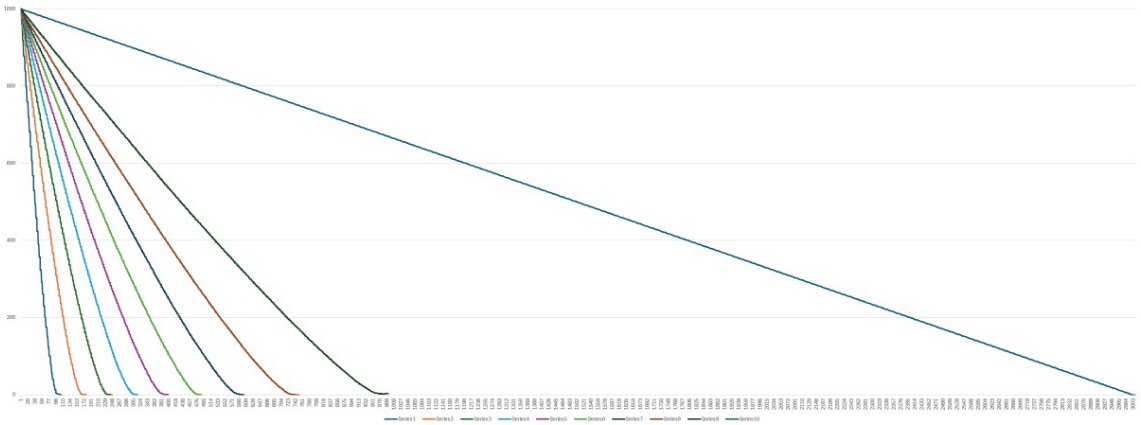


Figure 35: Conflicts over time for minimum local search with an increasing colour limit

In terms of performance, we have Figure 36, which shows the time taken to achieve the colouring of the graph at each density in a graph of 1000 nodes. The complete performance data is listed in Tables 5 and 6. A fully connected graph takes the minimum number of iterations, as 10 iterations is the convergence limit in the algorithm.

density	mean iter.	mean time
0.1	981.7	1.16685
0.2	959	1.35415
0.3	936.3	1.49299
0.4	911.5	1.52402
0.5	885.6	1.59061
0.6	854.7	1.58289
0.7	816.8	1.778
0.8	766	1.56813
0.9	691.5	1.55231
1.0	10	0.03414

Table 5: Performance results for running time at different densities (minimum local colour)

density	mean bm.	mean colours	mean diff. (%)
0.1	34.4	32.4	-5.81%
0.2	56.7	54.2	-4.41%
0.3	79	76.5	-3.16%
0.4	102.3	100.5	-1.76%
0.5	128.8	126.2	-2.02%
0.6	160.1	156.8	-2.06%
0.7	196	193.5	-1.28%
0.8	244.4	243.8	-0.25%
0.9	323.5	319	-1.39%
1.0	1000	1000	0%
Overall			**-2.22%**

Table 6: Performance results for colouring at different densities (minimum local colour)

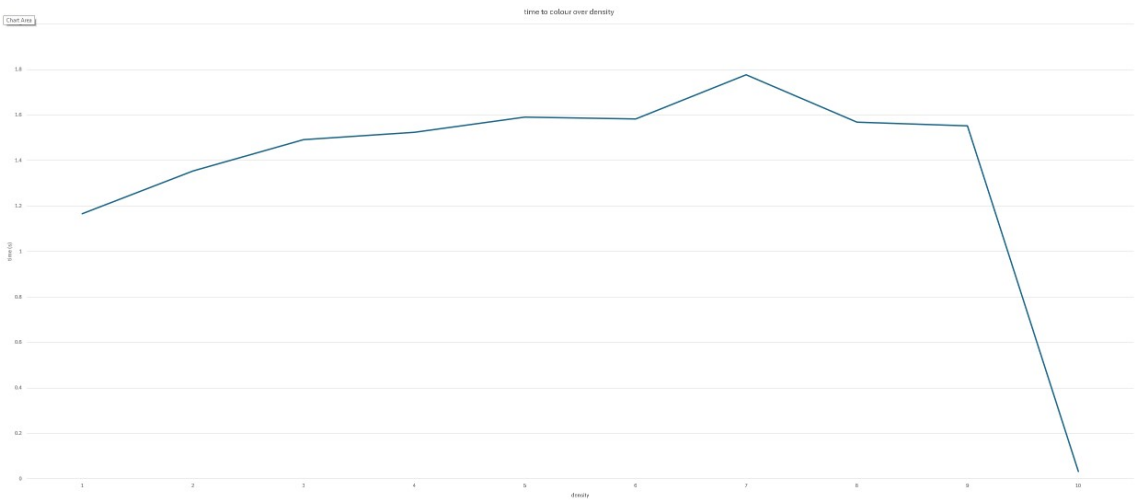


Figure 36: Time to colour graph over density

5.1.4 Summary of Static Graphs

To summarise, in this section we have measured the performance and analysed the results of four different algorithms: random local colour, minimum local colour and colour-blind local colour (incrementing and decrementing variants).

We saw how the difference in colouring performance between the random and colour blind algorithms was not that different, but the ability of the algorithms to colour the graph differs vastly when limiting the number of colours in the graph. In particular, the colour-blind algorithms produced wave patterns in the number of conflicts over iterations, displaying how properties of the topology of the graph can impact the graph colouring process and how individual agents in

distributed systems impact each other when making localised changes, even (or perhaps especially) with such limited local knowledge.

In contrast, we saw how minimum local colour was capable of colouring the graph with less colours than the greedy centralised benchmark consistently and much faster than either of the naive algorithms.

In section 6, we will go into more detail on the differences between the algorithms and what we can learn from them.

5.2 Dynamic Graphs

Dynamic graphs are a type of graph which can change during the colouring process. The change can be adding or removing nodes or edges in the graph.

5.2.1 Demonstration of Distributed Robustness

Simply removing edges and nodes from the graph could happen for any number of reasons. It could be due to the constraints changing in the problem, for example. As discussed in Section 2.7, one advantage of decentralised approaches is the robustness they provide when these changes occur. In this experiment, edges and nodes were removed due to simple random chance. In both cases, this was a 1 in 1000 probability that the relevant element was removed from the graph. In the case of edges, a random edge in the neighbour list of the node was selected to be removed.

Tables 7 and 8 show that the algorithms can adapt to the changing constraints and produce a colouring with no conflicts without the process failing.

density	mean bm.	mean colours	mean diff. (%)	mean conflicts
0.2	56.1	53.7	-4.28%	0
0.5	129.2	126.2	-2.32%	0
0.8	246.5	242.1	-1.78%	0

Table 7: Removing edges from the graph does not inhibit colouring

density	mean bm.	mean colours	mean diff. (%)	mean conflicts
0.2	56.3	25.9	-54%	0
0.5	129.1	59.1	-54.22%	0
0.8	246.1	106.6	-56.68%	0

Table 8: Removing nodes from the graph does not inhibit colouring

The results show how removing nodes has a far greater impact on the number of colours, but this is simply because removing one node is equivalent to removing many hundreds of edges. In each of the three density experiments for nodes, the number of nodes in the graph was reduced from 1000 to about 360 over roughly 900 iterations. This is obviously going to result in a much reduced number of colours compared to the benchmark.

5.2.2 Identifying Poor Constraints

Identifying poor constraints in a decentralised system is not straight-forward. The approach taken in this project was to set a threshold for conflicts in the locality of the node. If this threshold was met, then the node is simply removed from the system. The intent with this approach is to emulate the balancing of conflicts and constraints in a centralised system without the ability to share the general balance in the system between the nodes. In other words, the agents must adopt a means to reduce the constraints in some way that will balance conflicts and removal of constraints in the society.

In this experiment, a threshold of two thirds of neighbours must be met in order to meet the criteria for the removal of the node. The experiment was run on different densities; low (0.2), medium (0.5) and high (0.8). In Figures 37 and 38, we can see how the algorithm initially removes many conflicts, then a second run where it removes some more, then finally levels out. This flat portion can be considered the balance that the society has decided on. Of course, more constraints

are considered "poor" in a denser graph given our criteria, so the reduction in conflicts is greater in the denser graph. Table 9 illustrates the reduction in nodes at each tested density.

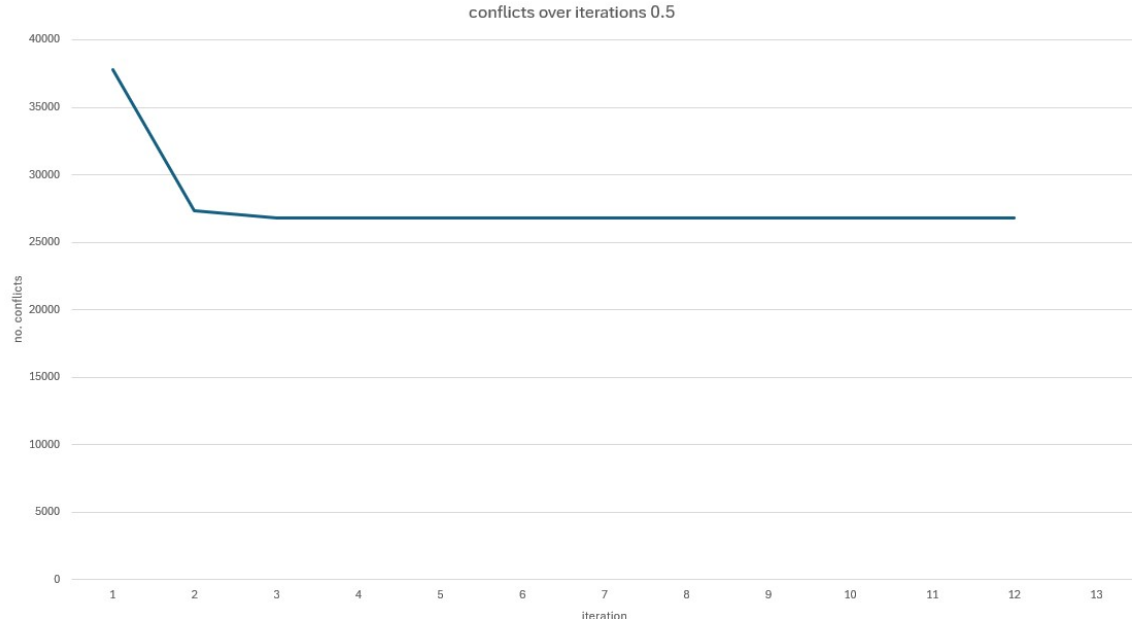


Figure 37: Reducing the conflicts by identifying poor constraints at density 0.5

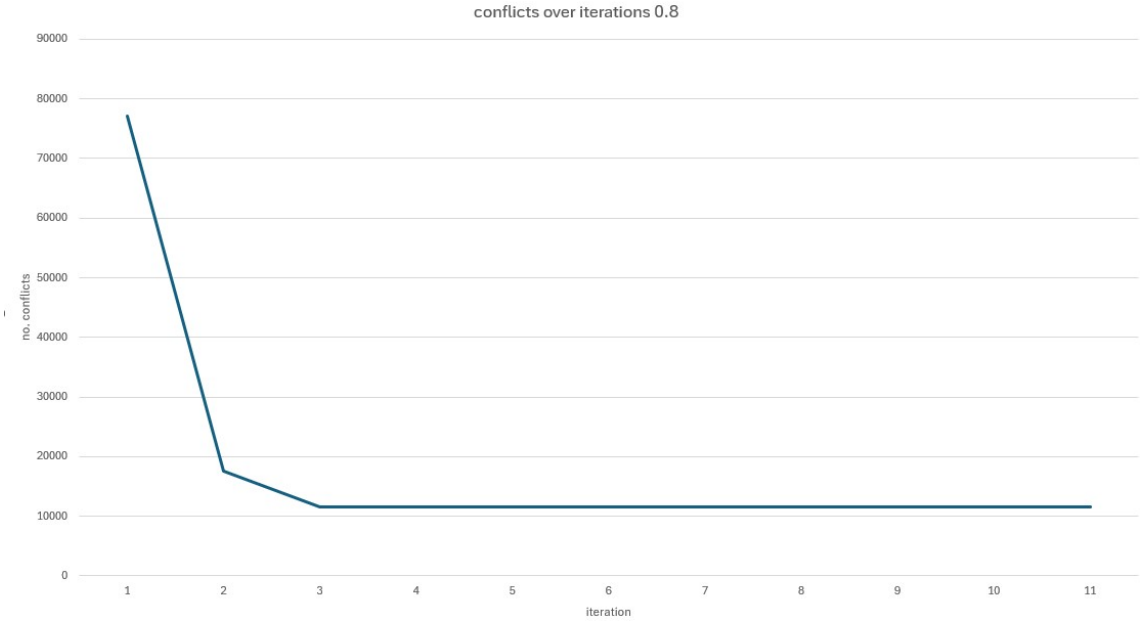


Figure 38: Reducing conflicts by identifying poor constraints at density 0.8

density	mean final nodes	mean diff. (%)
0.2	1000	0%
0.5	513.6	-48.64%
0.8	268.4	-73.16%

Table 9: Higher densities yield more poor constraints

5.3 Immutable Nodes

For these experiments, we sought to explore the impact of immutable nodes on the colouring process. The minimum local colour algorithm was used for both experiments.

5.3.1 Static Immutability

For this experiment, we run the colouring process with less agents than nodes in the graph. A certain percentage of nodes are therefore immutable and cannot change their colour or respond to changes in their locality in any way. In order to make the experiment a little more interesting, the code was modified so that random graphs were created with an initial colouring. This initial colouring assigns the node's degree plus one to each node. I.e., the maximum colour it could possibly need to not be in conflict with its neighbours. The job of the algorithm is then the same as usual: to reduce its colour if possible and avoid conflicts.

We aimed to find how much immutability we could withstand. To achieve this, we introduced a number of immutable nodes covering a 0.5%, 2.5% and 5% proportion of the total nodes in the graph (1000). Table 10 shows the results of these experiments.

		mean colours			mean diff. (%)			mean conflicts		
density	**mean bm.**	**0.5%**	**2.5%**	**5%**	**0.5%**	**2.5%**	**5%**	**0.5%**	**2.5%**	**5%**
0.2	56.3	60.2	73.6	84.4	7.12%	29.58%	50.71%	0	1	6.6
0.5	129.1	134.6	146.8	157	4.02%	13.98%	21.71%	0	2.2	13.6
0.8	245.9	249.4	262.8	264.8	1.22%	7.7%	7.12%	1	2.4	23.2

Table 10: Colouring with a static proportion of immutable nodes

The results show that higher density graphs are relatively more robust to immutable nodes than lower densities. This can be seen in the way that the relative increase in colours was lower, despite the raw number of conflicts being higher. Note that due to the nature of the algorithm, no new conflicts will be introduced; conflicts can only be fixed. In other words, the conflicts are more of a measure of the connectedness of the immutable portions of the graph than of any sense of lack of ability on the part of the algorithm.

5.3.2 Dynamic Immutability

Making nodes immutable should make the problem more difficult to solve. Nodes can be made immutable by removing the agent from that node in a system with immobile agents. When the agent is removed, no further changes can be made to the node, resulting in its colour being set in stone and no further modifications or improvements being made.

Table 11 shows how making nodes immutable makes the problem harder to solve. Roughly the same proportion of agents was removed (nodes made immutable) in the experiment at each density, namely around 500-600. Crucially, these nodes were removed over the course of the colouring process. In other words, the problem became harder the longer it continued. It follows then, that smaller graphs, or algorithms which can optimise a node's colour relative to its locality fastest will be less impacted by the increasing immutability.

density	**mean bm.**	**mean colours**	**mean diff. (%)**
0.2	56.3	381.2	577.09%
0.5	129.5	336.5	159.85%
0.8	244	354.7	45.37%

Table 11: The problem becomes harder when nodes are made immutable

In Table 11, we see how higher densities are less impacted by the increasing immutability. This could come from the hypothesised effect that being able to "solve" your locality faster will reduce the impact of immutability or from the fact that a more connected graph will need more colours anyway, and so a node becoming immutable will have less impact on the end result of the colouring.

5.4 Mobile Agents

In experiments with mobile agents, the number of moves that the agents can make, the number of agents in the graph and the density of the graph was modified. The results of the experiment can be seen in the tables below. Our main objective with these experiments is to measure the performance of an algorithm that uses less agents than there are nodes in the graph if these agents are capable of moving around in the graph.

density	mean iter. 1	2	5	mean time 1	2	5
0.2	11406.6	1545.4	11257.8	0.28	0.06	0.77
0.5	5352.4	2011.4	6302.4	0.25	0.15	0.79
0.8	2995.8	1636.8	3015.8	0.23	0.2	0.74

Table 12: Performance results for running time with 10 agents

density	mean bm.	mean colours 1	2	5	mean diff. (%) 1	2	5
0.2	56.2	71.8	102	74.8	27.3%	82.8%	33.09%
0.5	128.8	165	258.4	171.4	28.3%	98.77%	32.87%
0.8	244.4	322.2	314.8	313	31.51%	29.02%	27.65%

Table 13: Performance results for colouring with 10 agents

density	mean iter. 1	2	5	mean time 1	2	5
0.2	5434.2	739.8	5358	1.25	0.25	3.46
0.5	3554.8	1433.2	3548.8	1.63	1.15	5.25
0.8	2297.8	1505.4	2340.6	1.52	1.88	5.35

Table 14: Performance results for running time with 100 agents

density	mean bm.	mean colours 1	2	5	mean diff. (%) 1	2	5
0.2	56.2	63.4	123.4	63.6	13.62%	120.36%	10.42%
0.5	128.8	160.8	250.4	156	24.27%	93.2%	20.74%
0.8	244.4	324.4	292	321.8	31.76%	18.5%	31.03%

Table 15: Performance results for colouring with 100 agents

density	mean iter. 1	2	5	mean time 1	2	5
0.2	2014.2	1030.2	2008.8	2.19	1.74	6.19
0.5	2010	770.4	2008.6	4.02	3.05	13.14
0.8	1174.6	1077.8	1134.2	3.87	6.8	12.34

Table 16: Performance results for running time with 500 agents

density	mean bm.	mean colours 1	2	5	mean diff. (%) 1	2	5
0.2	56.2	57.2	93.4	58.8	0.7%	66.19%	4.26%
0.5	128.8	128	191	128.2	-1.39%	48.75%	-1.38%
0.8	244.4	315.4	275	322.6	28.12%	11.88%	31.57%

Table 17: Performance results for colouring with 500 agents

These results provide some interesting data. Most notably, the performance of having the agents move twice versus once or five times on every iteration. Apparently, this configuration is much worse at colouring the graph than others. Interestingly, in other informal experiments, other

even numbers for moves tend to produce similar results. However, this detrimental performance appears to even out with other configurations at higher densities. In fact, it performs better than other configurations with 500 agents.

The data shows that increasing the number of moves (at least, in small increments) does not improve the ability of the mobile agents to colour the graph. It is actually the opposite. As the number of moves increases, the performance of the algorithm decreases. However, in terms of running time, there is actually very little difference between one and five moves in terms of iterations. Using two moves appears to reduce the required number of iterations, but this is at the cost of significantly worse colouring results.

We can also see that the ability of the algorithm to colour the graph relative to a benchmark improves as the number of agents increases. This is in line with expectations. In Section 6, we will compare the performance of the original algorithm with its performance with a lower number of agents.

5.5 Bad Actor

In this experiment, we created a more complex algorithm which allows for the simulation of a situation in which the nodes must collectively identify and remove a bad actor from the graph. A bad actor in this case is an agent which uses its action to colour the node it is on with the colour that will cause the most conflicts in its locality. Recall that this algorithm is described in detail in Algorithm 11.

This experiment is quite different to any of the previous experiments conducted in this project. Instead of measuring performance, we simply want to investigate whether it is possible for the nodes in the graph (society) to coordinate effectively in order to identify a bad actor node. While implementing the experiment, the scope and number of variables quickly grew. This will be discussed further in Sections 6 and 7.

This experiment is not about how well we can colour the graph, but rather about how the agents can adapt to problems within the graph itself; similar, but more extreme than, the dynamic graph and immutable node experiments discussed earlier. The colouring algorithm used for the non-bad-actor nodes was the minimum local colour algorithm. This is due to the fact that it was a previously implemented algorithm which is effective and fast. Additionally, we want the nodes to be able to vote for the neighbour they believe to be the bad actor. This is done in a manner that aligns with the principles of the previous experiments: with as little centralised knowledge as possible. As such, the agents vote based on the neighbours which are conflicting.

In order to emphasise the cooperation of the nodes in this case, it was determined that a node had to have at least two votes in order to be "voted out" of the graph. This means that the distributed system must reach a consensus. The experiments were run on a small graph of just 30 nodes since stress-testing the algorithms is not relevant in this experiment. The experiment was repeated for low, medium and high density graphs. One important note is the means by which the votes are collected. Agents submit votes to a centralised collection, where the votes are counted after each iteration. The agents themselves receive no feedback from the tallying process, and as such do not know whether the bad actor has been voted out or not.

The results are visualised in Figures 39, 40 and 41, which show the points where the bad actor takes effect and is identified by the rest of the nodes. Figure 42 shows the output of running the program with the bad actor algorithm.

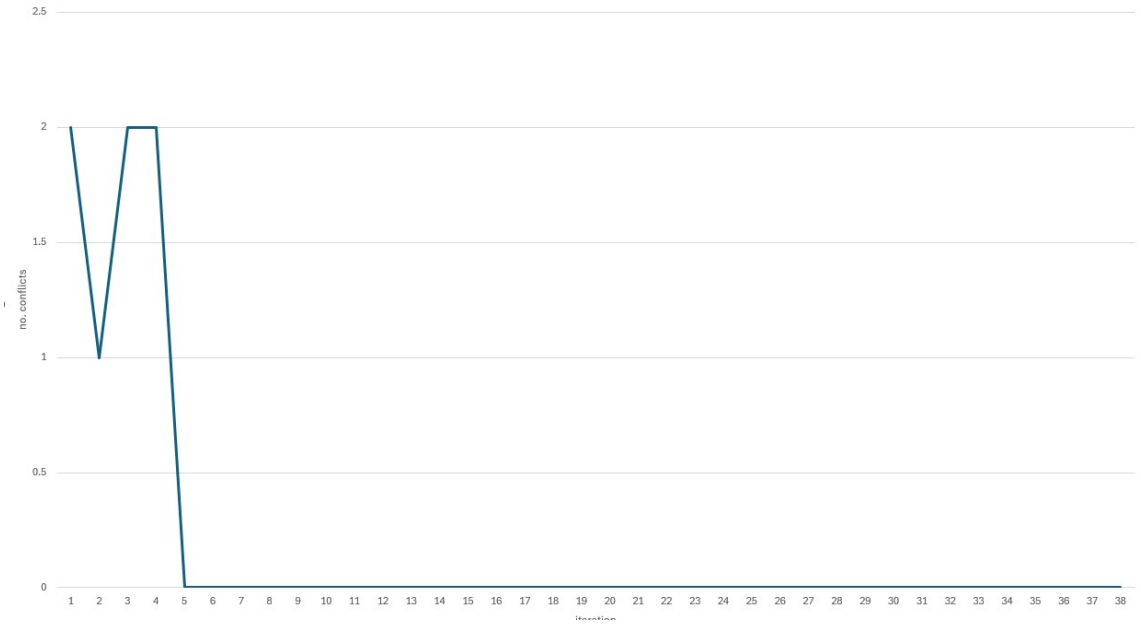


Figure 39: Low density graph with a bad actor

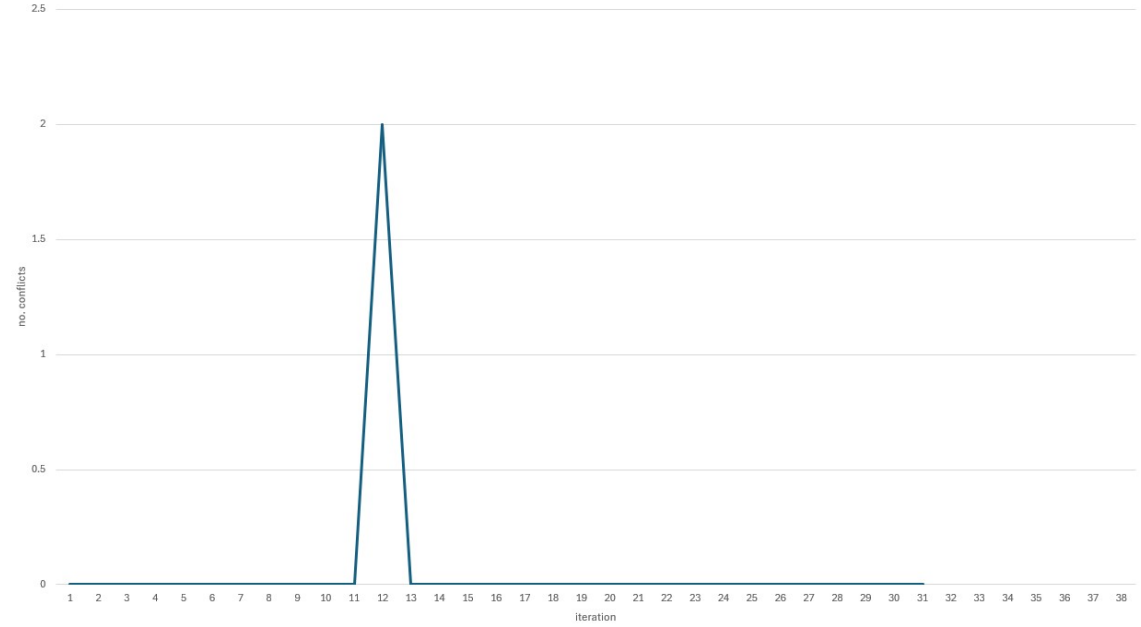


Figure 40: Medium density graph with a bad actor

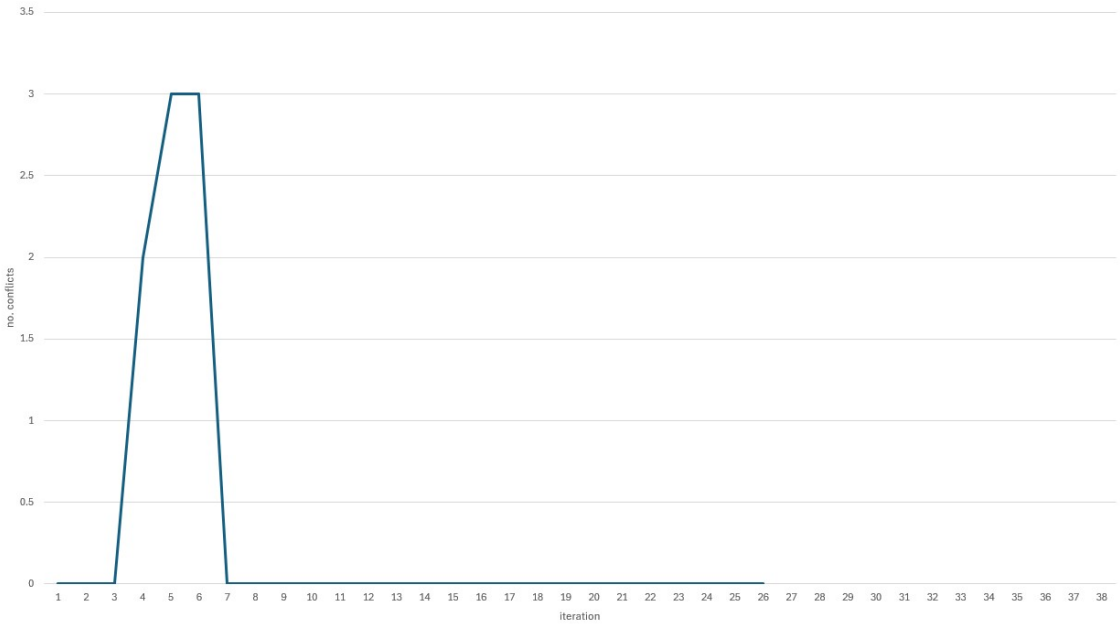


Figure 41: High density graph with a bad actor

```
PS C:\Users\Liam\Documents\fyp> .\colouring.exe -n 30 -p 0.5 -k a
highest degree: 20
lowest degree: 20
centralised colour: 9
selected most votes: 1
selected most votes: 4
the collective identified the imposter
ran for 31 iterations
number of nodes at start: 30
number of nodes now: 30
number of agents: 30
number of colours: 8
number of conflicts: 0
number of missed nodes: 0
time: 0.0050 seconds
```

Figure 42: Output of the bad actor algorithm

6 Analysis

6.1 Comparison of Static Algorithms

The algorithm with the best performance across all of the static algorithms we implemented for this project was the minimum local colour algorithm. This is in terms of both colouring performance and running time. In particular, the ability for it to colour the graph in the minimum number of iterations when the graph is fully connected is a good endorsement of its ability to effectively recognise when the graph is fully optimised in a distributed system.

Compare the colouring performance relative to the benchmark to the ability of the other algorithms (colour-blind and random). The other algorithms were capable only of achieving feasible colourings. It is surprising how quickly the random algorithm can do this. When running with an unlimited number of colours, it never takes more than one second, and is usually less than 0.1 seconds. Compared to this, even the minimum local colour algorithm looks slow. While this shows us that it is actually not too difficult for a distributed system to organise themselves into a working pattern, the solution is far from optimal.

The colour-blind algorithms take this to the extreme by being both poor at achieving optimal colourings and also very slow. In fact, much of the time the algorithm fails to even converge. This is most obvious when limiting the number of colours in the graph.

Limiting the number of colours in the graph is an interesting way to measure the relative performance of algorithms to find the most optimal solution *possible* in the graph given a maximum number of colours. When this is the situation, minimum local colour is once again the best choice, given it minimises the number of colours in the graph as much as it can. On the other hand, the random algorithm shows that it cannot find a colouring, but the graphed conflicts show that there

is no means by which it could minimise it in any way. We see interesting results in the colour-blind algorithm also, with the wave patterns it can produce.

6.2 The Wave Patterns

The wave patterns that emerge from the limiting of the colours in the graph when using the colour-blind algorithms exhibit fascinating properties of the graph. Essentially what we see is a wave pattern which reflects the topology of the graph. The most interesting aspect of this is that it is purely an artefact of the distributed, agent-based approach of the colouring process. What we see is the agents reacting to each others' decisions. This is what happens in each and every one of the distributed algorithms, including those discussed in Section 2, where we looked at the common implementations of these kinds of systems.

The fact that the waves produced are essentially configurable by adjusting properties of the graphs (density, number of colours, etc.) shows us how this is a new means by which we can visualise the graph; information about the structure of the graph problem is encoded into the graph of conflicts.

6.3 Identifying Poor Constraints

The ability of the algorithm to identify poor constraints is entirely dependent on the definition of what constitutes a poor constraint. In our experiments, it is clear how the conditions and implementation could relate to a real-world problem; you have a limited number of resources and have a certain tolerance for conflicts. Applying a tolerance to the problem will result in the program identifying aspects of the problem that do not meet the specified criteria.

This experiment was surprisingly successful. It shows how even simple criteria, combined with primitive algorithms that have only immediate information relating to their locale are still capable of finding a balance between colours and conflicts that satisfies the conditions specified by the problem definition.

It is also clear that this would work on any type of graph. Adjusting the threshold, the number of vertices in the problem, the density of the graph, etc. will change the rate at which nodes are removed. Ultimately, the conclusion is relatively straightforward: higher thresholds will identify more "poor" constraints in the graph in higher density nodes.

6.4 Comparison of Mobile Agents to Static Agents

When implementing the mobile agents, the hope was to find a configuration through which we could see perhaps a minor decrease in colouring performance as a trade-off for equivalent colouring performance. In this way, we could find an alternative to the path colour approach which is capable of colouring the initial, uncoloured graph. Unfortunately, this was not the result of the experiment. The closest we came was with 500 agents (half the number of the previous static experiment), which produced colourings lower than the benchmark. At the same time, this configuration took about 2.5x the time to run, not a true improvement at all. In general, we could not achieve anywhere close to the -2.22% improvement on the benchmark that we could with the standard agent-per-node approach.

What is really strange about the results of the mobile agent experiments is the emergence of the pattern where an even number of moves has entirely different performance to an odd number. This likely emerges from the fact that moving twice will skip nodes. Imagine a case where there are three connected nodes in a line. In this case, a single agent starting on either the left or right nodes will *never* be able to reach the middle node. The same would be true for any even number of moves. On the other hand, an odd number of nodes would be able to move between all three of the nodes.

This explanation clearly outlines how we achieve the results we see in the tables. Larger scale versions of the above example will inevitably occur in the graphs. This will make it harder for the agents to reach, and therefore optimise, the colours on particular nodes, leading to a case where the algorithm *thinks* there are no more changes to make when it actually simply cannot reach the nodes that need to be optimised. It is also how we end up with more conflicting/uncoloured nodes when the colouring process is terminated.

From these results, we can learn something about the mobile configurations. An even number of moves will result in faster colourings, but further from the optimal colouring. An odd number of moves will result in better colourings, but will take longer. We can also see that there is little

benefit to using more moves. In fact, it is better to use either one or two moves depending on what you want to achieve.

There is definitely potential for certain problems to be tackled with less agents. For example, identifying poor constraints in a dynamic graph is one case where using less agents may result in similar results with less work on each iteration.

6.5 Bad Actor

The experiments with the bad actor involved having the agents in the graph use the colouring information to identify the agent that was acting against the society. The demonstrations provided show how this is possible in (small) graphs of varying densities. Interestingly, a similar pattern to the colour-blind colouring performance appears to emerge, where the agents are capable of identifying the bad actor faster at low and high densities, with medium density being the toughest for the algorithm to work with.

The way the graph identifies the bad actor depends heavily on the algorithm and the order that the agents colour in. It could be the case that the bad actor is the first agent in the order. In this case, it would always colour to cause conflicts, but its neighbours would then fix the conflicts. The nodes vote before they colour in order to avoid this case, but it is harder to avoid cases similar to this which could come up. Essentially, the bad actor is not always identified, but this is in cases where it has no measurable detrimental effect on the colouring process. We can then conclude there are two ways in which the society can counteract the bad actor: they colour around its sub-optimal colouring or they identify the bad actor and remove its ability to interfere. In both scenarios, the agents in the graph have adapted to the bad actor in the graph and successfully coloured the graph, all with a very limited level of knowledge about the graph and no real means to communicate with each other.

However, as mentioned, there are certain factors not explored or adjusted in this experiment that do have a major impact on the way in which the algorithm behaves. These aspects, and how they could be approached in specialised experiments, are discussed in Section 7.

7 Future Work

In this section, we will discuss some potential directions that this research could go in the future.

7.1 Mixing colouring algorithms

The current implementation applies the same kernel to every agent. This could be easily modified to store the function pointer as part of the node struct instead of as a parameter to the algorithm. In this version, the kernel is part of the agent. This would allow for different nodes to have different behaviours in the same colouring process. This then poses a number of questions that could be investigated: What portion of agents need to be intelligent (random versus minimum colour, for example) in order to achieve acceptable results in the distributed system? Can certain algorithms *overpower* others? Do certain algorithms perform better when they are on the highest degree node in a cluster, or similar? Are there certain configurations that could have a leader node (one intelligent combined with simple random nodes as neighbours)? How would including and potentially also mixing movement algorithms impact the this?

7.2 More or less limited approaches

The motivation behind much of this research was to find how well agents with extremely limited knowledge can colour a graph. The project has implemented a number of algorithms with limited intelligence, but are there others that have not been considered? Consider the colour-blind algorithms. What else could a node do which is intentional and directed, but with no knowledge about what is causing the conflict?

On the other hand, while they were discussed in Section 2 and a centralised benchmark is built in to the program, it would be interesting to see how these more limited algorithms compare to more sophisticated and centralised algorithms in formal experiments. Indeed, the centralised benchmark could be extended to become yet another modifiable aspect of the program via parameters.

7.3 Dynamic immutability extended

The concept of nodes becoming immutable over time was an interesting extension of the static implementation that made the problem harder to solve over time. There is almost certainly a whole range of different experiments that could be done on this concept alone, but one that immediately jumps out as an obvious extension is a system where the agents are "woken up" with some probability after they have become immutable. This kind of experiment would delve even further into investigating the behaviour of agents and how they react to each other's states. It would be interesting to compare a system like this to one such as the mobile agents. Consider a situation where an immutable node wakes up in the same iteration that its neighbour *becomes* immutable. This is equivalent to the case in mobile systems where an agent moves once in an iteration. Is there some rate of immutability that is equivalent to a mobile system? Would the performance of the algorithm change depending on whether the kernel makes changes to the node before or after it becomes immutable?

7.4 Bad actor behaviour

Earlier, it was discussed how we could see the bad actor algorithm showed that the agents in a distributed system were capable of adapting to a bad actor in their midst. It was also discussed how the observed behaviour depended heavily on particular aspects of the algorithm. For example, the fact that minimum local colour does not naturally produce conflicts means that if conflicts arise, the neighbour nodes will vote for it, so the algorithm depends somewhat on the order in which the agents colour nodes in order to identify the bad actor.

There are many different aspects of this concept that could be explored, modified and adjusted in ways that the structure of the program developed for this project does not allow. For example, the modification of the node struct to hold the colouring kernel itself would simplify the implementation of the bad actor experiment greatly, as the kernel would not have to manually keep track of which agent was the bad actor.

A specialised program (or a modified version of the current one) would allow for experiments which investigated the effect of different voting methods, ordering of the agents (bad actor first or last), different colouring algorithms, multiple bad actors, etc. There is much which could be explored in terms of future research related to this concept.

7.5 More advanced movement algorithms

The movement algorithms implemented in this project are designed with the same set of principles as the colouring algorithms; the algorithm should operate with the most limited level of centralised information that it can manage. However, there are many different approaches that could be applied with more centralised information. Would more intelligent movement algorithms lead to better performance against agent-per-node approaches? Or is the mobile agent approach simply incapable of performing as well as approaches with more agents? What level of "intelligence" is required in a system with less agents to match a system with an agent on every node, but with less "intelligence"?

8 Conclusion

In this project, we designed, implemented and investigated a number of different distributed colouring algorithms, before expanding the scope of the graph colouring problem and modifying, relaxing and and adjusting its constraints to allow for our experiments to more closely relate to real-world problems. Our experiments included measuring the performance of the algorithms on static graphs, making the graphs dynamic in order to see how the distributed system withstood unpredictable modifications to the problem, identifying poor constraints in the most distributed manner possible, making nodes immutable both statically and dynamically, having agents become mobile in order to traverse the graph and comparing the performance of this troop of agents to a static setup and introducing a bad actor into the graph.

In addition to running these experiments, we provided an in depth analysis and explanation for the most interesting results, including the differences between the static colouring algorithms, the emergence of wave patterns in the colour-blind algorithms and the effect of various aspects of the bad actor algorithm on its ability to identify the bad actor in the society.

We also researched previous methods and approaches to distributed graph colouring and designed algorithms with this related work in mind, while also setting our own agenda for the path

of research. The project then went on to identify and suggest many different directions in which the varied and original concepts that this project has explored could be taken.

This was all done through the medium of a command line tool written in the C programming language that allows for both flexible use of the currently implemented algorithms, as well as a simple, well-documented and approachable means by which the program can be extended to make it capable of any other kind of distributed graph colouring experiment, making it a success on both a functional and a software-engineering level.

Overall, this project was a fascinating, educational and successful investigation and expansion of distributed approaches to the graph colouring problem.

9 Reflections

In this final section, I will reflect on the project in a personal capacity, including areas that presented specific challenges and personal reasons for certain decisions taken throughout the project.

9.1 Choice of Language

In Section 3, the practical reasons for choosing the C language were discussed. There are also a number of reasons I personally wanted to use the C language for this project. In particular, the opportunity to work on a complex project in a low-level language appealed to me. It is rare to have the opportunity to work in a low-level language such as this that is not simply a personal project or some form of systems engineering. The added complexity does add a certain additional challenge to the project, but after finishing the project, I feel I have a much better understanding of many different algorithms, as well as a better understanding of memory management, which I believe can improve the code you write, no matter the language.

Upon reflection, I believe it was entirely worth working on the project in C, given all of the reasons I have listed, but also due to the amount I have learned from doing it. However, I believe it would have been good to take the opportunity to work in a completely new language that I had never used before. When beginning the project, it seemed like it would have been a lot of work to learn an entirely new language, but given the amount I have learned about writing code in the C language, I do not think the overhead of learning the syntax of a new language such as Rust or Go, or alternatively a high-level language in a different paradigm, such as Clojure (Lisp), would have been that significant.

9.2 Distributed Implementation Design

The design of the distributed implementation is something that I am particularly happy with in terms of implementation. It finds an extremely satisfactory middle ground between the adjacency list and adjacency matrix that allows for suitable memory usage and simple implementation. This implementation also feels much more in the spirit of a distributed approach than other approaches for storing the graph.

In particular, the way in which this approach allows for the implementation of algorithms that can be run on local clusters of neighbours or the entire graph makes it very satisfying.

Additionally, the design is extremely extensible. It would be very straightforward to extend the node struct with new properties (attaching the colouring kernel as discussed in Section 7, for example) which could be used in new algorithms and approaches without the need for the modification of any of the existing code.

9.3 New Research

I feel that this project has explored areas of research that were previously under-explored, or outright not considered. In particular, I believe that the approach to the colouring problem as a means to an end to represent real-world problems allows for us to find new ways in which we can explore approaches to the problem. By not outright rejecting things like conflicts or sub-optimal colourings, we can more readily accept and find new interesting ways in which to work the distributed system to our advantage.

Additionally, I believe that this project has opened up many more questions which I believe will lead to new and interesting answers. Just the bad actor algorithm on its own is something I find fascinating in the way in which it could be expanded and explored to reveal all sorts of things about the interdependence and cooperation of agents in a distributed system. Some questions raised led to more interesting and promising results than others (immutable nodes versus mobile

agents, for example), but ultimately it feels very satisfying to have designed such a flexible piece of software which allows for so many of these different ideas to be explored.

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] Armen S Asratian, Tristan MJ Denley, and Roland Häggkvist. *Bipartite graphs and their applications*. Vol. 131. Cambridge university press, 1998.

[4] Isidoro Gitler, Enrique Reyes, and Rafael H Villarreal. "Ring graphs and complete intersection toric ideals". In: *Discrete mathematics* 310.3 (2010), pp. 430–441.

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

[6] Donald E Knuth. "Postscript about NP-hard problems". In: *ACM SIGACT News* 6.2 (1974), pp. 15–16.

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

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

[9] Dániel Marx. "Graph colouring problems and their applications in scheduling". In: *Periodica Polytechnica Electrical Engineering (Archives)* 48.1-2 (2004), pp. 11–16.

[10] Ulrich Hirnschrott, Andreas Krall, and Bernhard Scholz. "Graph coloring vs. optimal register allocation for optimizing compilers". In: *Joint Modular Languages Conference*. Springer. 2003, pp. 202–213.

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

[12] Thore Husfeldt. "Graph colouring algorithms". In: *arXiv preprint arXiv:1505.05825* (2015).

[13] Grzegorz Kondrak and Peter Van Beek. "A theoretical evaluation of selected backtracking algorithms". In: *Artificial Intelligence* 89.1-2 (1997), pp. 365–387.

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

[15] 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.

[16] 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.

[17] Marco Chiarandini, Thomas Stützle, et al. "An application of iterated local search to graph coloring problem". In: *Proceedings of the computational symposium on graph coloring and its generalizations*. Ithaca New York (USA). 2002, pp. 112–125.

[18] Severino F. Galán. "Simple decentralized graph coloring". In: *Springer Nature* (2016).

[19] Alessandro Checco and Douglas Leith. "Fast, Responsive Decentralized Graph Coloring". In: *IEEE/ACM Transactions on Networking* PP (Sept. 2017), pp. 1–13. DOI: 10.1109/TNET.2017.2751544.

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

[21] 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.

[22] Harmanjit Singh and Richa Sharma. "Role of adjacency matrix & adjacency list in graph theory". In: *International Journal of Computers & Technology* 3.1 (2012), pp. 179–183.

[23] Luis Barba et al. "Dynamic graph coloring". In: *Algorithmica* 81 (2019), pp. 1319–1341.

[24] Aleksander Bjørn Grodt Christiansen, Krzysztof Nowicki, and Eva Rotenberg. "Improved dynamic colouring of sparse graphs". In: *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*. 2023, pp. 1201–1214.

[25] Bradley Hardy, Rhyd Lewis, and Jonathan Thompson. “Tackling the edge dynamic graph colouring problem with and without future adjacency information”. In: *Journal of Heuristics* 24 (2018), pp. 321–343.