Algorithms for maximum colourful subgraphs in vertex-coloured graphs

Student Name: Rohan Lad

Supervisor Name: Dr Eleni Akrida

Submitted as part of the degree of BSc Computer Science to the

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

Abstract — Vertex-coloured graphs can be used to represent connections and interactions between groups of entities of a network where different groups have different colours. Given a vertex-coloured graph, one can ask the question of finding a cycle with the most diverse population, i.e. containing the maximum number of colours in its vertices, amongst all other cycles. This is the 'maximum colourful cycle' problem. The aim of this project is to implement an existing algorithm for finding the maximum colourful cycle in vertex-coloured threshold graphs and compare its experimental performance with the theoretically predicted one. Our implementation is in the form of a Python program, which can randomly generate representations of threshold graphs and compute the maximum colourful cycles in them, visually displaying the result thereafter. We iteratively ran the program with varying input sizes and measured its corresponding run times. We observed similarities when comparing the trends in our resultant data to the theoretically predicted run time behaviour.

Index Terms— Graph algorithms, Graphs and networks, Graph Theory, Path and circuit problems

1 Introduction

In graph theory, vertex-coloured graphs refer to graphs having the additional property that each vertex of the graph is 'coloured' - essentially assigned a particular colour. Graphs of this nature are typically used to model real-life scenarios and environments. Various problems can be established for these vertex-coloured graphs which can then be used to find solutions to the real-life scenarios they are modelling. Problems that have been studied in recent literature, which we shall discuss in further detail, are typically categorised twofold: those focusing on vertex-coloured graphs whereby no two adjacent vertices may have the same colour, and those where this restriction does not apply. In this paper, we shall be considering graphs of the latter type – namely, graphs whereby two adjacent vertices may be assigned the same colour.

One subject in which graphs of this nature are particularly applicable is bioinformatics. Corel et. al. show equivalence to and subsequently perform operations on vertex-coloured graphs within their research on the consistency problem in multiple sequence alignment [2]. Fellows et. al. study the problem of finding occurrences of motifs in vertex-coloured graphs, which asks: given a vertex-coloured graph and a set of colours M, does the graph have a connected subset of vertices whose set of colours equals M [27]? This problem has an important application in metabolic network analysis. Bodlaender et. al. discuss how the problem of determining whether a vertex-coloured graph is a subgraph of a coloured interval graph has an application in DNA physical mapping [28]. Kannan et. al. investigate the problem of determining whether a vertex-coloured graph can be triangulated without introducing edges between vertices of the same colour [13]. They outline how this problem is polynomially equivalent to the perfect phylogeny problem, a fundamental problem in bioinformatics which is concerned with the inference of evolutionary history. Vertex-recolouring problems are also relevant in this field. Chor et. al. discuss how the concept of connected colouring problems, where an r-component connected colouring of a graph is a vertex colouring such that each colour class induces a subgraph having at most r connected components, has several applications within bioinformatics when applied to general graphs [6]. These include the analysis of protein-protein interaction networks, protein structure graphs, and phylogenetic relationships modelled by split trees.

Vertex-coloured graphs can also be used to represent internet web pages and the links between them. The colour assigned to a vertex could represent the content or theme of a page, and an edge between two vertices represents, say, a hyperlink that can navigate the user from one page to another. Research has been undertaken to apply this in the context of Wikipedia pages which contain links to represent topics in multiple languages [1]. It has been shown that there are often errors in relation to the fact that links to pages in other languages tend to be incorrect or missing. A partitioning problem on a vertex-coloured graph model has been shown to be effective in identifying and removing the incorrect links.

One such problem that can be asked of these vertex-coloured graphs is the 'Maximum Colourful Subgraph' problem [4]. This asks the question of finding the subgraph of a vertex-coloured graph that contains the maximum number of colours amongst its vertices. Typically, this

question is asked for subgraphs of a particular structure, such as cycles, with the goal to then find the subgraph containing the maximum numbers of colours amongst its vertices across all other subgraphs of that particular structure. Vertex-coloured graphs can be used to model connections and interactions between groups of entities of a network. For example, they could be used to model species in a biological habitat where each unique species is represented by a different colour. The Maximum Colourful Subgraph problem may then provide insights towards the area of the network with the greatest diversity. Solving this problem specifically for subgraphs of various particular structures, as well as input graphs of various particular structures, will provide tailored insights of their own – for example, such solutions will be especially applicable in situations where the environment that is being modelled is best represented specifically by a graph of the given structure. It may also be useful to find a subgraph of a particular given structure with the most diverse population, if that structure best represents the format of the desired solution to whatever the problem may be.

With the motivation behind solving the Maximum Colourful Subgraph problem now clear, the primary aim of our project was to produce a working program that would be able to solve this problem on given input graphs. This was to be achieved via the implementation of Algorithm 1 presented by Italiano et. al., a novel algorithm that computes the maximum colourful cycle in vertex-coloured threshold graphs [4]. This essentially solves the Maximum Colourful Subgraph problem for subgraphs of a particular structure – in this case, cycles, and for input graphs of a particular structure – in this case, threshold graphs. This algorithm, however, makes use of a few sub-algorithms, referenced from different papers, all of which must be implemented individually as pre-requisites in order to complete the construction of Italiano et. al.'s algorithm. Thus, the implementation of each of these sub-algorithms represented key objectives of our project. Upon successful completetion of the implementation, the secondary key aim of our project was to perform an evaluation of our implementation, by comparing its experimental performance to the theoretically predicted one, as per the theoretical time complexity of the algorithm provided by the authors in the paper. These two aims together form the basis of our research question: can we successfully implement an existing algorithm for computing maximum colourful cycles, alongside querying whether or not its theoretically predicted run time is reflected in a practical setting of the algorithm.

As mentioned, vertex-coloured graphs can be very useful for modelling real-life environments. In order to run theoretical algorithms (such as those proposed by Italiano et. al.) on these models, however, a practical implementation of the algorithm needs to have been established. Such an implementation will also allow the algorithm to be tested and run across a large number of inputs, which can vary in terms of the nature of their parameters. In implementational environments, it is typically very easy to generate many representations of input graphs whose parameters can be tuned to one's liking, and iteratively call the algorithm on these inputs in an automated manner. This is not something that can be feasibly achieved to the same extent in a purely theoretical environment. Thus, applying the algorithm in a practical setting has the potential to raise insights or concerns that might have been difficult for the algorithm's authors to identify when they were designing the algorithm in a theoretical context.

One of the key reasons why a particular algorithm might be appropriate to use in certain practical settings could be due to its resource complexity. In many practical settings where vertex-coloured graphs may be used for modelling purposes, there is likely to be variation in terms of the make-up of these graphs; it might be the case that the size and nature of the graphs are highly unpredictable. There may be certain requirements with regards to resource usage, such as time: there may be limitations on how long programs running on these graphs may take. These factors highlight why a particular algorithm with a certain time complexity would be desirable for practical use ahead of similar algorithms with worse such complexities. But it is important to evaluate any implementation of the algorithm to ensure that it does indeed behave according to its theoretically predicted complexity in a practical and unpredictable environment. Otherwise, the benefit of using that particular algorithm is lost. This highlights the importance of the evaluative work presented in this paper.

We successfully achieved our main objective: a working implementation of the aforementioned algorithm by Italiano et. al. for maximum colourful cycles in vertex-coloured threshold graphs. This consisted of a program, written in Python 3.10.1, capable of computing and visually displaying thereafter the maximum colourful cycle of an input threshold graph. The program can randomly generate such a threshold graph whose underlying properties (such as its number of vertices) can be given by the operating user. As mentioned previously, this algorithm could not be complete without the prior implementation of a few sub-algorithms that are called within the algorithm. A successful implementation of each of these algorithms represented an objective for this project, and these objectives were achieved. The first of these was the implementation of an algorithm for computing the maximum matching of a graph. This was achieved via the implementation of Jack Edmonds' 'Blossom' algorithm [14]. Upon successful completion of this, the next objective was the implementation of an algorithm for computing the maximum colourful matching of a graph. This was successfully achieved via the implementation of an algorithm presented by Cohen et. al. [9]. The implementation of an algorithm presented by Harary et. al. to find Hamiltonian cycles in threshold graphs, another requirement from within the main algorithm, was also a successfully achieved objective [15]. Once we had put together and verified our implementation of Italiano et. al.'s algorithm, we then also successfully met our objective of carrying out an evaluation of its performance. We were able to measure the

total program runtime of our implementation and did this iteratively for inputs of varying sizes. This was done in the context of three separate experiments, where each of the number of vertices, edges and colours (the three variables that appear in the function representing the theoretical time complexity of the algorithm) were varied whilst the other two were kept constant. We plotted graphs that showed how the program run time varied as the given input variable changed. We analysed the patterns and behaviour of these graphs, comparing them to our expectations given the theoretical time complexity of the algorithm as presented by Italiano et. al.. We note that overall, there is strong evidence that our implementation does indeed behave as expected with regards to the predicted time complexity.

2 Related Work

A great deal of research has been conducted on vertex-coloured graphs. As mentioned previously, such research is typically based within two contexts: vertex-coloured graphs whereby no two adjacent vertices may have the same colour, and vertex-coloured graphs where this restriction does not apply. The first context is often used in various scheduling and assignment problems, where vertex-coloured graphs under the restriction that adjacent vertices must have different colours prove to be highly effective at modelling the relevant scenario. A classic example might be the job scheduling problem, whereby vertices of a graph represent the jobs; two vertices are connected if they cannot be executed simultaneously. Colours are used to represent time slots; every job requires one time slot. The graph has a k-colouring (i.e. it can be coloured using k unique colours) if and only if the jobs can be executed in k time slots, such that clashing jobs are not executed simultaneously. The number of colours of the graph then tells us the minimum time required to execute all the jobs [3]. Vertex-coloured graphs of this nature continue to be a heavy area of research, much of it focused on their properties and results, and their applications towards solving logical and computational problems.

Our project, however, considers vertex-coloured graphs that are not under the condition that adjacent vertices must have differing colours. The reader may assume that any future references to 'vertex-coloured graphs' refer to graphs coloured in this manner. Italiano et. al. present algorithms for constructing maximum colourful cycles in each of bipartite chain graphs and threshold graphs. A bipartite graph is defined as a graph whose vertices can be divided into two disjoint and independent sets such that every edge connects a vertex in one set to a vertex in the other. A bipartite chain graph is a bipartite graph in which the vertices in one of its independent sets can be linearly ordered such that $N(x_1) \supseteq N(x_2) \supseteq \ldots \supseteq N(x_{|X|})$, where $N(x)$ denotes the neighbourhood of vertex x; that is, the subgraph of the original graph composed of all the vertices adjacent to x and all edges connecting vertices adjacent to x. Yannakakis first introduced the concept of bipartite chain graphs in his research on vertex-deletion problems, which ask what the minimum number of vertex deletions of a graph result in a subgraph satisfying a particular set of properties [5]. His work involved the characterisation of certain properties of graphs for which a bipartite restriction of the vertex-deletion problem is polynomial and those for which it remains NP-complete.

Our main focus in this project is the algorithm constructing the maximum colourful cycle in threshold graphs [4]. Threshold graphs are defined as graphs that can be constructed from a one-vertex graph by repeated applications of either of the following two operations:

(1) Addition of a single 'isolated' vertex to the graph; upon addition it is not connected to any other vertices.
(2) Addition of a single 'dominating' vertex to the graph; upon addition it is connected to all other vertices currently in the graph.

The concept of threshold graphs was first introduced in 1977 by Chvátal et. al. [16]. Given an m x n zero-one matrix A, they investigated the possibility of the existence of a single linear inequality $ax < b$ whose zero-one solutions are precisely the zero-one solutions of $Ax < e$. They interpreted their results in terms of graph theory and the logic of threshold graphs. Golumbic's 1980 book included a chapter which presented the first detailed treatment of threshold graphs [30]. He explored some results, such as the fact that any induced subgraph of a threshold graph is also a threshold graph, and how any graph that contains an induced subgraph isomorphic to one of those is not threshold. In 1995, Mahadev and Peled published a book devoted to threshold graphs, which to this day is widely considered to be the most complete reference on the topic [31]. It includes nine different unique characterisations of threshold graphs, as well as many proofs on them. Research continues to be undertaken to explore properties and behaviour of threshold graphs, such as work by Herggernes et. al. providing a linear running time algorithm to recognise threshold graphs [29].

Italiano et. al. devised an algorithm to be performed on threshold graphs (as well as a similar algorithm for bipartite chain graphs) because they proved that the goal of finding a cycle containing the maximum number of colours is actually NP-hard for the majority of types of input graphs [4]. However, they showed it is possible to solve this problem in polynomial time for threshold and bipartite chain graphs (the reason for this is outlinned later in this section) and thus devised corresponding algorithms for each thereafter. A key aspect of the algorithms is the the computation of a *maximum colourful matching* of the input graph (because the authors identify connections between maximum colourful matchings and maximum colourful cycles). A *matching* of a graph is a subset of the edges of the graph such that any two edges of the matching have no common vertices. A *maximum matching* is a matching that covers as many vertices as possible. A *maximum colourful matching* refers to a matching in a vertex-coloured graph with the largest number of unique colours amongst its vertices.

Cohen et. al. present polynomial time algorithms for finding both maximum and minimum *tropical matchings* (should they exist) in vertex-coloured graphs [9]. They define a *tropical matching* of a graph to be a matching in the graph in which each colour of the graph appears once. This is a specific example of a *tropical subgraph*: a subgraph where each colour of the graph appears once. Establishing tropical subgraphs is a problem that has been investigated in the context of various other types of subgraphs too. For example, Manoussakis et. al. have investigated tropical paths, in the context of shortest tropical paths (tropical paths with the minimum total weight) and maximum colourful paths (paths with the maximum number of colours) [26]. They showed that both problems are NP-hard for some types of input graphs (such as interval graphs) but provide a fixed parameter algorithm and a polynomial time algorithm for the former and latter problems respectively, for some other specific input graphs. In fact, threshold graphs were an example type of input graph for which they established that a maximum colourful path can be computed in polynomial time. Foucaud et. al. have also investigated this problem in the context of tropical connected subgraphs and tropical independent sets [25]. They were successful in classifying the complexity of some classes of this problem.

A tropical matching can be considered to be maximum or minimum if it contains the largest or smallest possible number of edges, respectively. Note the distinction here between a tropical matching and a maximum colourful matching – a tropical matching contains every single colour of the graph, whereas a maximum colourful matching (required for Italiano et. al.'s algorithm) contains as many colours as possible – which may not be all of them. It is therefore a generalisation of a tropical matching. Computing it is perhaps more relevant in practical situations where a tropical matching will often not exist. For example, rather than finding the area of a habitat with the greatest possible diversity, it is often more desirable to locate the area with the greatest diversity across the entire habitat, especially if no area exists with the greatest possible degree of diversity, in which case, attempting to find a tropical matching would yield no result. Of course, a maximum colourful matching containg all possible colours is also tropical.

The nature of Cohen et. al.'s algorithm for computing maximum tropical matchings is such that it can also be used to find maximum colourful matchings. The algorithm's logic is based on the Lemma that any maximum tropical matching in a graph is also a maximum (uncoloured) matching in the graph, since it possesses the largest possible number of edges a matching can have. Thus, the algorithm seeks to use a maximum (uncoloured) matching and construct a tropical matching from this by iteratively increasing the number of colours in the matching without decreasing the number of its edges, until such an operation is no longer possible. Therefore, on termination, the matching in its final state is either tropical, or, if not, has the maximum number of colours possible, because the algorithm was not able to induce any further colours into it. This means the output of this algorithm is a maximum colourful matching, whether that be tropical or not, and is therefore appropriate for our use case.

Clearly, this algorithm requires the need to find a maximum (uncoloured) matching in the graph as a pre-requisite. Several algorithms have been established in literature to solve this problem, with varying theoretical running times. The first ever polynomial time algorithm for this problem was Jack Edmonds' 'Blossom' algorithm, published in 1965, which runs in $O(|E||V|^2)$, where $|E|$ denotes the number of edges of the graph, and $|V|$ denotes the number of vertices of the graph [14]. The algorithm constructs a matching by iteratively increasing the size of an initially empty matching along augmenting paths in the graph. Odd-length cycles in the graph are contracted into single vertices, with the search then continuing iteratively in the contracted graph. We describe this algorithm in further detail in section 3.1. Since Jack Edmonds released his algorithm, a faster algorithm devised by Micali et. al. is currently accepted as the fastest known algorithm for solving this problem [17], capable of finding the maximum matching in $O(\sqrt{|V|}|E|)$ (although since then, a few more algorithms have been developed, of equal time complexity). This algorithm increases an existing matching along maximal sets of disjoint minimum length augmenting paths. It involves a novel method of handling odd-length cycles, via the growth of breadth-first-search trees at all vertices not in the matching.

A Hamiltonian path in an undirected graph is a path that visits every vertex exactly once. A Hamiltonian cycle is a Hamiltonian path such that there is an edge connecting the last and first vertices of the Hamiltonian path. The 'Hamiltonian Cycle Problem' seeks to determine whether a Hamiltonian cycle exists in a given graph. It is widely established that this problem is NP-complete, even for some special types of input graphs: for example, Garey et. al. showed that even if the domain of the problem is restricted to planar graphs or graphs with low vertex degrees, the problem remains NP-complete [23]. The length of the longest cycle of a graph, often referred to as the graph's circumference, can be used to solve the Hamiltonian Cycle Problem and is therefore NP-hard to determine. However, previous research has provided sufficient evidence that finding the longest cycle in each of bipartite chain graphs and threshold graphs is achievable in polynomial time: Uehara et. al. presented an algorithm for finding the longest path in a bipartite permuation graph of linear running time [8]. Mahadev et. al. showed that the length of a longest cycle in a threshold graph is obtained in terms of a largest matching in a specially structured bipartite graph, and the cycle can be computed in linear time [7]. It is this property of bipartite chain graphs and threshold graphs that Italiano et. al. take advantage of; their algorithms construct a Hamiltonian cycle over a given set of candidate vertices which are meaningfully selected such that any such constructed cycle will correspond to a maximum colourful cycle of the input graph [4]. Thus, Italiano et. al. restrict the nature of the input graphs to being threshold and bipartite chain in each of their respective maximum colourful cycle algorithms to ensure that they have polynomial running times.

A key aim in our project was to establish the time complexity of our implementation and compare this to the predicted time complexity of the algorithm stated by Italiano et. al. in their paper [4]. However, it is far from trivial to simply retrieve the time complexity of any program analytically (without execution) - essentially due to the Halting Problem; Alan Turing proved that determining whether or not a program will even terminate successfully is not decidable, let alone how efficient it is [10]. Instead, a more resourceful approach would be required in order to gain an understanding of the program's time complexity. In previous research, experimentation has been carried out with graph-theoretic complexity measures based on analysis of program control flow - calculating the number of basic paths within a program and graphically modelling this thereafter [11]. Further research, however, has shown that methods such as this will not always return fully reliable results, because there are certain limitations - such as the inability to measure the complexity of individual blocks within a program, which can alter the program's overall complexity [12]. We ultimately settled on a numerical method to estimate the program time complexity, which we shall describe further in the next section.

3 Solution/Methodology

This section provides a high-level overview of the steps involved in each of the key sub-algorithms that we implemented, and how we were able to put these together to complete the implementation of the main maximum colourful cycle algorithm presented by Italiano et. al. [4]. Implementing each algorithm involved reading and analysing each algorithm's associated research paper(s), in which the authors would typically present the steps of their algorithms alongside proofs for any expected results. The nature of the algorithmic presentation typically varied between the papers in terms of clarity and format, ranging from highly technical descriptions that weren't straightforward to follow, to clear lines of pseudocode. A major aspect of this project therefore involved understanding the logic behind each of the algorithms as well as the proofs interlaced within them, as this was necessary in order to gain confidence that our corresponding implementation was doing the right thing. This section also describes the technical and practical details of our implementation, alongside a usage description. An overview of the methods we employed in order to analyse the implementation's performance is also outlined.

3.1 Maximum Matchings

As previously mentioned, a necessary aspect of Italiano et. al.'s algorithm is the prior establishment of the maximum matching of the input graph. Ideally, we would have implemented Micali et. al's proposed algorithm for this problem, given that it is one of the fastest known solutions [17]. Its implementation would have therefore been most conducive towards our program being as efficient as possible. However, we noted that our ultimate aim here is to compare the run time of our final program to the theoretically predicted one. Italiano et. al. state the time complexity of their maximum colourful cycle in threshold graphs algorithm to be $O(\max\{|C| \cdot M(m, n) , n(n + m)\})$, where $|C|$ denotes the number of colours in the input graph, and $M(m, n)$ denotes the time for finding a maximum matching in a general graph with m edges and n vertices. So essentially, this time complexity has been stated in terms of the run time of whatever algorithm is used to find a maximum matching. This means that the choice of algorithm that we use to find this will not have any adverse effect with regards to making a fair comparison during our evaluation. We therefore opted to implement the Blossom algorithm to find maximum matchings [14]. This algorithm was published in 1965, and since then has been widely analysed. As a result, plenty of resources have been published and are available to help understand the algorithm and ensure the correct steps are followed in implementing it correctly. The same cannot be said of Micali et. al.'s algorithm, which, being a considerbaly more complex algorithm, is significantly more challenging to implement, compounded by the fact that the paper itself is far from simple to follow, and additional supplementary resources to help with this are not widely available. It was therefore decided that the slightly worse run time of the Blossom algorithm was a worthwhile trade-off, especially given that implementing it would not undermine the robustness of our evaluation.

To understand how the Blossom algorithm is able to find the maximum matching in a graph, it is important to first understand the concept of an augmenting path. Given a graph and any matching of that graph, an augmenting path is a path in the graph (i.e. a sequence of connected vertices) whose endpoints are unmatched (i.e. not in the matching, whereas vertices in the matching are described as 'matched'); the length of this path is odd. The path's edges alternate such that one edge is in the matching, the next one is not in the matching, and so on and so forth. The main logic behind the Blossom algorithm is that given a graph and a matching, it finds augmenting paths in the graph so that it can increase the size of the matching. Since the endpoints of the augmenting path are unmatched, and the path is of odd length, the augmenting path will contain x edges currently in the matching, and $x + 1$ edges not currently in the matching. Therefore, locating such a path can lead to an increase in the size of the matching by 1, via the removal of any edges from the matching that are in the augmenting path, and the addition to the matching of all the other edges in the augmenting path. This process is iteratively repeated until an augmenting path can no longer be found, which indicates that the current matching is maximum.

However, this process alone can run into issues on input graphs that contain odd-length cycles. The algorithm may be unable to find the desired augmenting path, because as it begins to build up the augmenting path, it may traverse a cycle resulting in an augmenting path that is longer than the shortest path between the two endpoints. This problem is caused by the existence of an odd-length cycle in the graph (referred to as a 'blossom'), which is connected (via

one of the vertices in the cycle) to an alternating path that ends in an unmatched vertex (referred to as the 'stem'). To avoid this problem, the algorithm follows the stem, and when it reaches the blossom, it contracts it into a single vertex, creating a new, condensed graph. An augmenting path is then found in this new graph, which is used to improve the matching in the traditional manner. Note that this is achieved via the algorithm recursively calling an instance of itself, so as it then attempts to find an augmenting path in this new graph, further recursive calls may be required if further blossoms are identified in the new graph. When an augmenting path has been found, the algorithm steps out of its current recursion, and the previously contracted blossom is 'lifted' back, with its edges induced into the new matching accordingly. This works via the reliance on the fact that the graph contains an augmenting path if and only if the contracted graph has an augmenting path, a theorem which Edmonds proves in his paper [14].

3.2 Maximum Colourful Matchings

The next element of our implementation was Cohen et. al.'s proposed algorithm to find a maximum colourful matching of a graph, that is, the matching of a graph that contains the maximum number of colours amongst its vertices [9]. This algorithm works via iterative improvement upon a maximum (uncoloured) matching to increase the number of its colours. The output of the Blossom algorithm as described previously is therefore used as the input to this algorithm.

Given a vertex-coloured graph G, and a maximum (uncoloured) matching, M, of the graph, the algorithm constructs a new graph, G'. This graph consists of all vertices of G whose colour does not appear once in M. It also has all the vertices of M whose colour appears once in M. Finally, it consists of an extra vertex z. The edges of G' are those edges of G that are induced in G' as a result of its aforementioned defined vertex set. There is also an edge from z to all images of matched vertices in G, whose colour does not appear once in M.

Cohen et. al. prove that the construction of G' in this precise manner ensures that the following steps will guarantee convergence towards a solution: if there exists an augmenting path in G', then use it to augment the matching by one edge. Then, based on the new (resultant) matching, construct G' again and continue repeating these steps until an augmenting path can no longer be found in the latest construction of G' (with respect to the current matching). The current matching then represents the maximum colourful matching of the graph. On each iteration, the matching is updated such that the number of colours amongst its vertices increases. This is despite the fact that during the process of locating an augmenting path in G', the colours of the vertices do not play a role in the algorithm; the outcome with regards to the increase in the number of colours occurs purely due to the manner in which G' is constructed. But the fact that the process of finding an augmenting path in G' and using it to update the matching M is generic was significant. We had already implemented this aspect of the algorithm as part of our implementation of the Blossom algorithm, which includes the exact same logic. Thus, we were able to create a re-usable function for finding augmenting paths and updating the matching accordingly which was shared between both this algorithm and the Blossom algorithm.

3.3 Hamiltonian Cycles in Threshold Graphs

Throughout Italiano et. al.'s algorithm, a set of 'candidate vertices' is carefully selected from the input graph such that if there exists a Hamiltonian cycle amongst these vertices, the cycle would represent the maximum colourful cycle in the original input graph and therefore the final output of the algorithm. Harary et. al. present an algorithm for constructing such a Hamiltonian cycle – a problem which is typically NP-complete for the majority of input graphs [15]. But the authors present, crucially, a polynomial time algorithm for threshold graphs, which the aforementioned 'candidate vertices' constitute.

Recall from section 2 the constitution of a threshold graph, and how each of its vertices are either 'isolated' or 'dominating'. The input to Harary et. al.'s algorithm is a threshold graph, and knowledge of the nature of each of the vertices with regards to whether they are dominating or isolated is assumed, as well as the order in which the vertices were added to the graph (which is not an issue, because this metadata is included as part of the graph's definition when the graph is initially passed in as input to our program). It is worth noting at this stage that, by definition, the set of dominating vertices of a threshold graph is a clique – a subset of vertices such that all members of the subset are connected to one another. It is straightforward to construct a Hamiltonian cycle amongst vertices in a clique because one can simply traverse through each of the vertices individually without the need to consider a particular order. This property is used several times in Harary et. al.'s algorithm, which we shall now provide an overview of.

Recall that the input to this algorithm is a threshold graph, which in our case is a carefully constructed subgraph of the original input (threshold) graph. Recall also that a Hamiltonian cycle of a graph must traverse every single one of its vertices exactly once.

If there are no isolated vertices, and at least three dominating vertices (the minimum number required for the graph to contain a cycle) in the graph, then the whole graph is a clique (i.e. every pair of vertices is adjacent) and so a simple traversal through each of the vertices is returned as the Hamiltonian cycle. If there is just one isolated vertex, and its degree is at least two (the minimum amount such that this isolated vertex can be part of a cycle), then the graph again contains a clique between all the dominating vertices, as well as an isolated vertex which is connected to at least two members of this clique. Thus, the returned Hamiltonian Cycle is constructed on the clique as before, with an edge to and from the isolated vertex inserted in. If there are at least two isolated vertices, and the number of dominating vertices is greater or equal to this quantity (if this wasn't the case, any cycle would be forced to traverse through a vertex twice which would therefore render it not Hamiltonian) then a new graph is

constructed from the input graph. This graph is constructed via the removal of the first s - r dominating vertices from the input graph, where s denotes the number of dominating vertices in the graph, and r denotes the number of isolated vertices in the graph (thus r dominating vertices are ultimately included in the constructed graph). All edges between dominating vertices are also removed. The resultant graph is a bipartite graph. Recall bipartite graphs can be defined as a graph whose vertices can be divided into two disjoint and independent sets such that every edge connects a vertex in one set to a vertex in the other. In this case, there are r vertices in each of these sets; one set contains the r isolated vertices, and the other contains the r remaining dominating vertices. If a Hamiltonian cycle exists and can be constructed in this bipartite graph (we shall refer to this cycle as A), then A can be extended to construct a Hamiltonian Cycle in the original graph. Recall that the bipartite graph, and therefore also A, contain all the vertices of the original threshold graph except for the first s - r dominating vertices. Thus, the desired Hamiltonian Cycle of the original threshold graph must append these s - r dominating vertices to A. The s - r dominating vertices represent a clique, so it is straightforward to construct a Hamiltonian path between these vertices. To then connect this path to A, we simply need to locate a vertex from each of the disjoint sets (with regards to the aforementioned composition of a bipartite graph) to act as the connection points. From the set containing the isolated vertices, any of them which neighbours of one of the s - r dominating vertices acts as the connection point from that set. Since all dominating vertices are connected to one another, the connection point from the set containing the dominating vertices can be any of those vertices. This will successfully allow A to be extended to include the s - r dominating vertices that were not part of the bipartite graph, which then represents a Hamiltonain Cycle of the original threshold graph.

Of course, constructing A, a Hamiltonian Cycle of the constructed bipartite graph, is not a trivial task. To achieve this, the algorithm first checks that the degree (in the original threshold graph) of the jth latest isolated vertex added to the graph is at least $j + 1$. Similarly, it checks that the degree (in the original threshold graph) of the jth earliest added dominating vertex added to the graph is at least $j + 1$. If any vertices fail to meet this condition, no Hamiltonian Cycle will exist. This check is adapted (to fit our specific scenario) from Chvátal's condition [18]. This states that in order for a general graph with degree sequence $d_1 \leq d_2 \leq \ldots \leq d_n$ to be Hamiltonian, $d_j \geq j + 1$ or $d_{n-j} \geq n - j$ for $j = 1, 2, \ldots, [n/2]$.

If all vertices successfully pass this check, the algorithm constructs a Hamiltonian cycle based on whether the size of the disjoint sets of the bipartite graph (i.e. the quantity of isolated vertices, which is also equivalent to the quantity of dominating vertices) is even or odd. Let $y_1, y_2 \ldots y_r$ denote the dominating vertices of the bipartite graph, in the order in which they were added to the original threshold graph i.e. y_1 denotes the first dominating vertex added and y_r the last. Similarly, let $x_1, x_2 \ldots x_r$ denote the isolated vertices, but in reverse order; i.e. x_1 denotes the last added isolated vertex and x_r the first. If r is even, the constructed Hamiltonian cycle in the bipartite graph is x_1 -> y_r -> x_2 -> y_{r-2} -> x_4 -> y_{r-4} -> … -> x_{r-2} -> y_2 -> x_r -> y_1 -> x_{r-1} -> y_3 -> … -> y_{r-1} -> x_1. If r is odd, the cycle is x_1 -> y_r -> x_2 -> y_{r-2} -> x_4 -> y_{r-4} -> … -> x_{r-1} -> y_1 -> x_r -> y_2 -> x_{r-2} -> y_4 -> … -> y_{r-1} -> x_1. These cycles are constructed by taking advantage of the bipartite nature of the graph; each disjoint set contains an equal number of vertices, and so a traversal can be established via a 'top down' approach which visits each vertex exactly once, alternating between the two sets as it travels along.

3.4 Maximum Colourful Cycles in Threshold Graphs

We shall now describe the logic behind Italiano et. al.'s algorithm, the central aspect of our implementation, to construct the maximum colourful cycle in an input threshold graph [4]. It begins by computing the maximum colourful matching in the graph, using the algorithm described in section 3.2. The number of colours in this matching is established as a constant, C_m, that is referenced several times throughout the algorithm. The authors establish that there are a various 'cases' – referring to distinct properties of the input graph with respect to the value of C_m – which dictate how a maximum colourful cycle would be constructed, should one exist. We shall describe the nature of each of these cases in further detail. In each case, a set of candidate vertices is established. If there exists a Hamiltonian cycle in the subgraph induced by those candidate vertices (constructed using the algorithm described in section 3.3, which will return appropriately if no Hamiltonian cycle can be constructed) then this cycle represents the maximum colourful cycle in the input graph and is returned as the algorithm's output. Otherwise, the algorithm rules out that particular case and moves onto the possibility of the input graph meeting the conditions for the next case, establishing the corresponding candidate vertices again.

Let C_c denote the number of colours of any maximum colourful cycle. Let X denote the set of dominating vertices of the graph, and Y the set of isolated vertices. Let C_1 denote the set of colours in Y but not in X, and C_2 the set of colours in X. For any set of vertices Z, let $C(Z)$ denote the number of unique colours amongst the vertices of Z. Let $c(v)$ denote the colour of a vertex v. The graph falls into case 1 if $C_c = C_m + 1$. In this situation, a maximum colourful cycle may exist with some edge connecting two dominating vertices. The authors prove that if this is indeed true, then another maximum colourful cycle will exist whose vertices is given by $X \cup A$, where, for each colour in C_1, the set A contains the first isolated vertex (with regards to the order of being added to the threshold graph) of that colour. Thus, the algorithm checks for the existence of a Hamiltonian cycle on these candidate vertices, and constructs and returns it if it exists. Note that this proposal of the existence of a maximum colourful cycle is conditional on the graph falling under case 1.

If the Hamiltonian cycle doesn't exist, this implies that a maximum colourful cycle with some edge connecting

two dominating vertices may not exist. Still under the proposal that the graph falls into case 1, that is where $C_c = C_m + 1$, the opposite scenario is then assumed: for any maximum colourful cycle, there is no edge of this cycle that connects two dominating vertices. The authors prove that if this is indeed true, then another maximum colourful cycle will exist (again, conditional on the graph meeting the case 1 condition). Let $X^+(v)$ denote the set of dominating vertices added to the graph after vertex v. Let v^* denote the unique vertex satisfying $|X^+(v^*)| + |C(X^+(v^*))| = C_m + 1$, the existence of which is proved by the authors. Note its identification can be computed efficiently by simply checking if this equation holds over all vertices. The vertices of a maximum colourful cycle would then be given by $X^+(v^*) \cup B$, where, for each colour in the set $C(Y) \setminus C(X^+(v^*))$, the set B contains the first isolated vertex of that colour. As before, the algorithm then attempts to construct a Hamiltonian cycle on these candidate vertices.

If no Hamiltonian cycle exists over these candidate vertices, then it is safe to conclude that the input graph does not fall into case 1, where $C_c = C_m + 1$. We now assume case 2; $C_c = C_m$. In this situation, a maximum colourful cycle may exist which does not contain a particular dominating vertex, v^{**}, nor its colour. The authors prove that if this is indeed true, then another maximum colourful cycle exists. Let the integer l be equal to $C_m - |C(X)| + 1$. The vertices of the maximum colourful cycle would be given by the union of $X \setminus \{v^{**}\}$ and the set of vertices D, where, for the first l colours of $C_1 \cup \{c(v^{**})\}$, the set D contains the first isolated vertex of that colour. When we refer to the 'first' l colours, an ordering of the colours can be considered in terms of when the first isolated vertex of each colour was added to the graph – a colour whose associated isolated vertex was added earlier than another colour's associated isolated vertex is considered to be 'before' that colour. Thus, the algorithm will iterate over every dominating vertex of the graph, and on each iteration, set the given vertex as v^{**} and establish the candidate vertices as above. If, at any point, a Hamiltonian cycle can be constructed on these candidate vertices, the algorithm returns this as the maximum colourful cycle.

It may be the case that after having iterated over all the dominating vertices, a Hamiltonian cycle was never successfully constructed on the candidate vertices. This implies that no maximum colourful cycle exists which omits a dominating vertex and its colour. Still under the proposal that the graph may fall into case 2, that is where $C_c = C_m$, the opposite scenario is then assumed: any maximum colourful cycle contains all the dominating vertices (and therefore also all the colours of the dominating vertices). Let l be equal to $C_m - |C(X)|$. Consider the set E, where, for the first l colours, E contains the first isolated vertex of that colour. Also consider the set F, where for every colour in C_2, F contains the last dominating vertex with that colour. X_t denotes the t last vertices in $X \setminus F$. The authors prove that the vertices of a maximum colourful cycle would then be given by $X_t \cup F \cup E$. The algorithm varies t from 0 to $|X \setminus F|$ inclusive, in increments of 1. On each iteration, it produces the candidate vertices as per the above, and seeks to construct a Hamiltonian cycle accordingly, returning it as the output if it is ever successful in doing so.

It may be the case that after having iterated over every t value, a Hamiltonian cycle was never successfully constructed on the candidate vertices: at this stage, it is safe to conclude that the input graph does not fall into case 2. This implies that case 3 must hold for this input graph; $C_c = C_m - 1$. The authors proved that $C_m - 1 \leq C_c \leq C_m + 1$. Thus, now we have been able to safely rule out cases 1 and 2, case 3 must hold. It is fairly straightforward to obtain a cycle with $C_m - 1$ colours. In a maximum colourful matching M (computed at the beginning of the algorithm), every edge must contain at least one dominating vertex (by the definition of threshold graphs). If one dominating vertex is selected from each edge of M, these selected vertices can be denoted $x_1, x_2, \ldots, x_{|M|}$ in order of when they were added to the graph (x_1 being added earliest). Let z_j denote the neighbour of x_j in M. (x_j, z_{j-1}) must be an edge in the input graph, again by definition of threshold graphs with respect to how dominating vertices are connected to all currently existent vertices upon their insertion. Therefore, $(x_1, z_1, x_2, z_2, \ldots, x_{|M|-1}, z_{|M|-1}, x_{|M|}, x_1)$ represents a cycle that contains all vertices in M except for $z_{|M|}$. This cycle must have at least $C_m - 1$ colours, since every single vertex of the maximum colourful matching has been included, except for $z_{|M|}$. Thus, $C_c \geq C_m - 1$ holds for this constructed cycle. Since we already exhausted the potential of C_c being greater than $C_m - 1$, we know that $C_c = C_m - 1$. Thus, this constructed cycle is the maximum colourful cycle.

3.5 Implementational Details

The program representing our implementation was written in Python 3.10.1. Each of the four algorithms described in this section were implemented in individual Python files. Each of these files contain a main function that takes a graph as input and returns the algorithm's output with regards to this input graph. Each file typically contains other helper functions too, which are called within the main function. For example, a helper function was written which takes a graph and a matching as input and returns a corresponding augmenting path (this particular helper function is actually called from the main functions of both the Blossom algorithm and the maximum colourful cycle algorithm). The file representing the algorithm for computing the maximum colourful cycle then imports the other three files, so that it can call the main functions from these files as necessary.

The Python module 'Networkx' was used to represent the composition of any graphs (most importantly the input graph, but also any additional graphs generated throughout the algorithm) and store them in memory [19]. The module provides a 'dictionary of dictionaries of dictionaries' data structure to represent the graphs. The keys of the parent dictionary are the vertices of a graph; the values associated to each key are an adjacency dictionary keyed by neighbor to the edge attribute dictionary. This particular data structure allows for efficient lookup and removal of edges, a simple way for iterating over the graph's vertices, and a simple way for iterating over a

vertex's neighbours. These are common operations that have to be carried out throughout the algorithm. It was therefore desirable to use a data structure for which such operations would not incur a significant performance cost, which could interfere with our evaluation when it comes to timing the program run-time and comparing it to the theoretical time complexity. Another advantage of NetworkX is that it allows us to store the entire graph within a single variable; methods and objects on this variable are used to perform particular actions, such as adding an edge to the graph. This also means that metadata and additional custom properties of the graph (such as its colouring) can also be stored within this variable and associated to the graph accordingly. This is useful from a programmatic standpoint: in our algorithm, the graph ends up having several associated properties such as a matching, a colouring etc. The ability to store all this data holistically within one variable ensures simplicity when passing the graph from one function to another, as we just have to pass over one variable. It also ensures important data such as a graph's colouring is always 'attached' to the graph and won't be lost when say, the graph is being passed across different functions.

Ultimately, the input graph consists of some vertices and edges which are stored in the aforementioned data structure afforded by NetworkX. We chose to 'label' the vertices by their index with regards to the order in which they were added to the graph. This allows for programmatic ease. It means that whenever another data structure is required to store values associated to the vertices, such as their colours, a list can be used whose value at index x represents the value associated to the vertex labelled x. Two lists are created in this fashion, one containing the colour of each vertex, and one containing the identity of each vertex in terms of whether it is dominating or isolated. Both of these lists are stored in the graph variable. This then concludes the make-up of the graph variable that represents the initial input to the algorithm.

In order to create such a graph, a simplistic user interface has been designed. This asks the user to enter at the command line the number of dominating and isolated vertices they wish the threshold graph to contain, as well as the number of unique colours they wish the graph to contain. We wrote a function that randomly generates a threshold graph that adheres to these specifications. Recall the list that stores whether each vertex is dominating or isolated; its value at index x determines the nature of the xth vertex to be added to the graph. This list is initialised with length equal to the number of vertices that will be in the graph (given by the combined number of dominating and isolated vertices initially requested by the user). The value at every index of this list is initially set to 'False', implying that every vertex is isolated. Say the operating user requested there to be y dominating vertices. The function will then randomly select y indices of this list and set their values to 'True' – meaning those vertices will be dominating. It then iterates over this list in order to construct the threshold graph in a traditional fashion: when it encounters a 'False' value in the list, it will add an isolated vertex to the graph. When a 'True' value is encountered, a dominating vertex is added to the graph, along with the corresponding edges induced as a result of this. An alternative command line option allows the user to provide an explicit breakdown of exactly when the dominating vertices should be inserted into the graph, should they wish - but by default, the function decides this randomly. This favours ease of use when we wish to generate graphs quickly without always wanting to define them to such precision. Each vertex's colour is randomly assigned to be one of a certain set of colours – the size of this set being equal to the number of colours requested by the user. This graph is then passed as input to the main function of the file representing the algorithm for the maximum colourful cycle. Upon termination of the algorithm, the maximum colourful cycle will be printed to the terminal. This is a sequence of comma-seperated edges, each of the form (u, v), that comprise the cycle. In addition, a visualisation of the graph, along with its associated maximum colourful cycle, is produced and displayed to the user. This is achieved via the 'draw' function of Networkx, which uses the Python library Matplotlib [20]. This library is designed for the creation of visualisations in Python; we also use it for producing and displaying graphs in our evaluation of the implementation's performance. It supports the ability to individually set the colours of each of the vertices and edges in the graph. Thus, we set the colours of those edges that appear in the maximum colourful cycle to red, whilst all the other edges are grey. This makes it visually clear which edges comprise the maximum colourful cycle. The vertices themselves are coloured circles, their colours reflecting the colouring of the vertices as per the definition of the graph. This allows the operating user to easily observe which vertices and therefore which colours are part of the maximum colourful cycle, as well as which vertices and therefore colours (if any) are not part of the cycle. We also used this visualisation to manually verify the correctness of the output, as part of our general testing and validation - which is described in further detail in section 5. An example visualisation is provided in the next section.

3.6 Performance Evaluation Details

Our secondary key objective was to perform an evaluation of the performance of our implementation. The aim here was to establish the time complexity of our implementation and compare this to the predicted time complexity of the algorithm stated by Italiano et. al. in their paper [4], which is dependent on the number of edges, vertices and colours of the input graph. Thus, to evaluate our program's performance, we iteratively ran it with varying values for one of these three variables, whilst fixing the other two to be constant (and did this three times so as to independently vary each of the three variables). On each execution, the raw program run time was measured; the aggregate data was fitted using a curve-fitting library (we used Matplotlib [20]), with the value of the varying term (e.g. number of vertices) on the x-axis, and the raw program run time on the y-axis. In theory, the general trend/shape of the fitted function should imply the time

complexity of the implementation. This can then be compared to the theoretical time complexity of the algorithm, in the context of the two fixed variables being constants. This method requires us to be very mindful of the fact that the experimental run times are very rough estimates; it is likely that the procured data will be quite sparse and probably contain several anomalies. Thus, it is important for us to focus our interest in the general trend of the data, if any, as the size of the input that we are varying starts to get much larger, rather than the actual individual values of each data point.

We firstly generated a graph that adhered to the required conditions with regards to the two fixed variables – so for example, if we were analysing the variation in the number of vertices, we set the number of edges and colours to selected constant values; the generated graphs' number of edges and colours then needed to be equal to these constants. We then ran the program using this generated graph as the input and measured the program's raw run time using the Python time module [22]. This provides functions capable of capturing the current timestamp. Thus, calling such functions at the start and end of the program meant a simple subtraction of the two values yielded us the program's run time. We then kept track of this data point and repeated the entire process with a new graph, whose value corresponding to the variable we were varying was incremented. For each data point, the values corresponding to the fixed variables remained the same as they were before, even if the graph was different. Eventually, all the data points were plotted onto a scatter plot using Matplotlib (examples of which are shown in the next section), allowing us to analyse the effect of varying the variable of choice on the program run time [20]. We opted to use scatter plots because the most important aspect of our evaluation is the analysis of the relationship/correlation between the variable of choice and the program run time. We thought scatter plots would be most conducive towards such an analysis. We know that there is not too much significance in each individual data point, given that each one represents a raw program run time and is therefore a very rough estimate in the grand scheme of things. There are also likely to be many anomalies and values that don't fit the supposed trend. However, scatter plots containing a large number of data points are useful to portray a general trend amongst the data points should one exist and are not significantly discredited by anomalies.

The first element of our evaluation involved varying the number of vertices in the graph, for a fixed number of edges and colours. In order to carry out this evaluation, we needed to iteratively call the algorithm; on each iteration, the input graph needed to have the same number of edges (and colours), but the number of vertices had to vary, ideally in a structured manner. Fixing the number of colours in each input graph was straightforward; if we wanted there to be x colours in the graph, we would randomly assign each vertex to be one of these x colours. Fixing the number of edges for a varying number of vertices, however, was a much greater challenge. The function we originally wrote to generate threshold graphs takes the number of vertices as input and randomly generates a threshold graph with that many vertices; thus, the number of edges of this graph is random. Working in the opposite direction – that is, fixing the number of edges beforehand and subsequently generating a threshold graph from this is far from straightforward. We could not identify a numerical method that can compute the exact constitution (with regards to the quantities and ordering of dominating and isolated vertices, which of course dictate how many edges the graph will have) of a threshold graph containing a given number of edges. Although it isn't too difficult to come up with one threshold graph containing a given number of edges, we have a need to generate many such graphs, varying the number of vertices each time. Thus, we came up with a different solution to generate our data points for this case. Knowing the number of edges that the threshold graph contains also tells us the minimum and maximum number of vertices it can contain (by considering the cases of having as many and as little dominating vertices as possible, respectively). This firstly tells us the range by which we could possibly vary the number of vertices over. We also use this to randomly generate a threshold graph (using the same function described in section 3.5), by choosing random quanitites of isolated and dominating vertices in the graph, such that their sum falls in this range of possible vertex counts. The idea here is to iteratively generate graphs in this manner until a generated graph's number of edges is equal to the desired number of edges. We then have a valid graph which we can run through the algorithm, yielding us a data point. This process is then repeated until we have enough data points. Although this method does work, it is fundamentally very inefficient, primarily due to the repeated generation of graphs which do not have the target number of edges. Thus, there was a significant focus during development of this function on efficiency optimisation and improvement in the generated data. We found that a random ratio of isolated to dominating vertices meant that the number of edges in the vast majority of the generated graphs were above the desired value. This made sense, because the 'average' threshold graph containing a given number of edges will have considerably more isolated vertices than dominating vertices, since dominating vertices contribute significantly more edges in the graph compared to isolated vertices. We therefore experimented with this ratio to improve efficiency, employing strategies such as dynamically adjusting it when trying to target a specific number of vertices.

The second element of our evaluation involved varying the number of edges in the graph, for a fixed number of vertices and colours. As in the previous experiment, fixing the number of colours in each input graph was straightforward; if we wanted there to be x colours in the graph, we would randomly assign each vertex to be one of these x colours. In terms of varying the edges, one of the primary characteristics of a threshold graph that determines how many edges it will have is the ratio between its number of dominating and isolated vertices. Thus, we iteratively generated graphs with a

given number of isolated and dominating vertices (such that the sum of their respective quantities was always equal to the fixed number of vertices), and on each iteration, incremented the number of dominating vertices and decremented the number of isolated vertices. This ultimately meant we had a large data number of data points effectively covering the entire range of possible number of edges a graph of the given number of vertices could contain. This was the most efficient function of the three experiments, and thus the easiest for generating large amounts of data for large input sizes.

The third element of our evaluation involved varying the number of colours in the graph, for a fixed number of vertices and edges. Unlike the other two experiments, it was not necessary to generate a new graph (as far as the vertex and edge constitution is concerned) for every single data point. Since the number of vertices and edges are fixed, we simply generated a graph adhering to these fixed values at the beginning of the function. We then set a colour assignment on the graph, such that it would have x colours (again by randomly assigning each vertex in the graph to be one of these x colours), and iteratively did this whilst incrementing x for each data point until it reached its maximum value - the number of vertices in the graph (which ultimately acted as a ceiling value for how many data points we could have).

4 Results

We successfully met our objective of implementing Italiano et. al.'s algorithm for computing the maximum colourful cycle of any input threshold graph [4]. A visualisation corresponding to an example output from our program is shown in Figure 1.

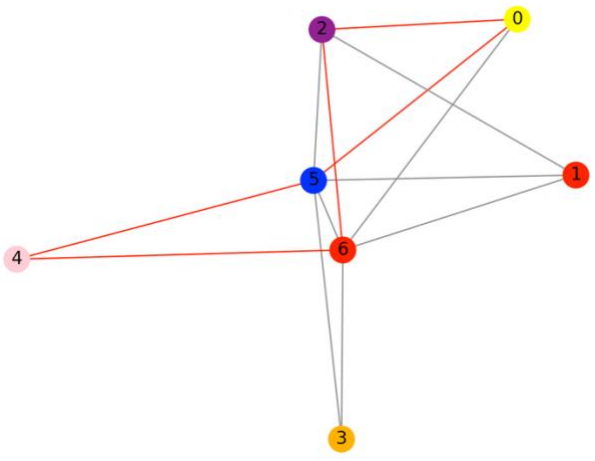

Fig 1. A maximum colourful cycle of an input threshold graph. Each time it is run, our implementation produces a visualisation of the output like this. The red edges denote the cycle: in this case, it is 4 -> 5 -> 0 -> 2 -> 6 -> 4, which contains 5 colours. The graph contains 6 colours, but it is not possible to construct a cycle containing all 6 colours.

4.1 Validity

Testing and validation represented an extremely important aspect of this project. It was vital that we were able to have confidence in the accuracy of the output of our implementation. Our objective of implementing the algorithm could not really be considered to have been met if our implementation was not correct. Also, if our outputs were inaccurate, this would seriously undermine the significance of any conclusions made during our evaluation, when we measure our program's run time and compare it to the theoretically predicted time complexity: if an inaccurate output is returned, it is highly possible that the algorithm ran through a different section than it would have run through when generating the correct output, which may have been quicker or slower. Therefore, from the very beginning, a test-driven development approach was adopted throughout, as if we were operating in a software engineering context [21]. This worked very effectively due to the nature of what we were implementing. Our work was already partitioned into a couple of different algorithms, and typically, each algorithm was partitioned into several different functions, stages or 'cases'. Therefore, we were able to assign each of these functions/stages as individual units to be tested. We then would not begin implementing the next unit until we were absolutely certain that the output of the previous unit was correct. This process then scaled continuously as we began to introduce new algorithms and develop our implementation further. Considering a few high-level examples of how this worked in practice: the algorithm to identify an augmenting path in a graph was thoroughly tested and exhausted across as many different possible inputs types as possible. Only once it was safe to conclude the output of this algorithm was correct was the remainder of the Blossom algorithm developed. The implementation of the maximum colourful matching algorithm did not begin until we were absolutely certain that we were generating the correct maximum matching from the Blossom algorithm, and so on and so forth. For the main maximum colourful cycle algorithm, there are various 'cases'; every input graph falls under one of these cases, which subsequently determines how the output cycle is constructed. This represented another example of how we split up the implementation into individual tested units. We would not begin implementing one case until we were certain that the output of the previous case was correct should the input graph fall under that case. This approach allowed constant retention of confidence in the accuracy of our implementation throughout the development process.

With regards to validating the output of each individual 'unit', this was generally achieved via a combination of automated and manual testing, across a wide variety of input types. Automated testing typically involved ensuring that the output was of the expected structure and met any expected guidelines or conditions. For example, if the output of a unit was expected to be a path in the graph, it was simple enough to write a function that checks that an edge exists in the graph between every consecutive vertex of the given path. We were then able to check these conditions held when running the function iteratively in an automated manner, across as many different types of inputs as possible (e.g. varying the ratio of dominating to isolated vertices in the graph, varying the number of

colours etc.) A large degree of manual validation was also carried out throughout the implementation. This typically involved inspecting the output of a unit to ensure that a correctly structured output that had passed simple automated testing actually seemed correct in its value. This was especially applicable when validating the final output (the maximum colourful cycle) of the implementation; the generated visualisation (such as that in Figure 1) of the cycle proved very useful for this. We were able to inspect the graph and ensure it was indeed covering as many colours as possible. We spent time attempting to spot another cycle in the graph that might have a larger number of colours. Again, a key aspect of this was the requirement to carry it out for a range of different input graphs, to account for potential unforeseen behavior under some conditions, and to ensure maximum code coverage, as some 'cases' of the algorithm are only run through on certain types of inputs. Thus, this manual checking process was repeatedly carried out on as many different inputs as possible. With the implementation in its final form, as far as we could tell, the output appeared to indeed represent the maximum colourful cycle on every single run. For aspects of the implementation which represented more commonly known problems, such as the maximum matching of a graph, we were also able to carry out further validation by comparing our ouputs to outputs from open-source implementations of these algorithms written by other people. Observing the outputs from these other implementations being identical to ours gave us further confidence in the accuracy of our implementation. With that said, we could only do this for sub-sections of the algorithm; as far as we could see nobody else has (publicly) implemented Italiano et. al.'s algorithm, so we were not able to validate the absolute final output of our implementation in that respect.

As a result of the extensive testing and validation that we carried out, we felt confident in the accuracy of our implementation. Furthermore, the program is complete in terms of its expected capabilities. It is able to take a threshold graph, and, regardless of its nature in terms of its vertex constitution (i.e. ratio of dominating to isolated vertices, positioning of these vertices) or colouring (the number of unique colours with respect to how many vertices there are, the distribution of the colours in terms of sparsity and biasing unique colours amongst isolated/dominating vertices etc.), is able to compute the maximum colourful cycle in that graph. This was a key condition for meeting this objective successfully – that the implementation would be able to correctly compute the maximum colourful cycle, regardless of the nature of the input threshold graph. Even for cases where the implementation runs very slowly because the input graph is very large or complex, from our testing, the algorithm still eventually terminates in these scenarios.

4.2 Performance

We shall now provide an overview of the outcome of our performance evaluation, the methods for which were described in section 3.6. Recall our objective to compare the experimental performance of our implementation of Italiano et. al.'s algorithm with the theoretically predicted one, as outlined in their paper. The authors claim the algorithm computes a maximum colourful cycle in the input graph in time $O(\max\{|C| \cdot M(m, n), n(n + m)\})$, where $|C|$ denotes the number of colours in the graph, m denotes the number of edges in the graph, n denotes the number of vertices in the graph, and $M(m, n)$ denotes the time for finding a maximum matching in a general graph [4]. Since we used the Blossom algorithm to compute the maximum matching, we can substitute the time complexity of the Blossom algorithm into the $M(m, n)$ to yield the time complexity of the algorithm in our case. Edmonds states that the time complexity of the Blossom algorithm is $O(|E||V|^2)$, where $|E|$ denotes the number of edges of the graph, and $|V|$ denotes the number of vertices of the graph [14]. Subsituting this in yields us a total time complexity of $O(\max\{|C| \cdot m \cdot n^2, n(n + m)\})$. In each experiment, as the two fixed variables are treated as constants, a new time complexity function is established in terms of the variable being varied. This gives us an idea of how our program run time should behave as that particular variable's value is varied. For each experiment, we also plotted this time complexity function itself, on a seperate graph, substituting in the values for the number of colours, number of edges and number of vertices into $|C|$, m and n respectively (two of which are constant across every data point). Of course, the resultant output values themselves from this function are entirely arbitrary and cannot be compared on face-value to the corresponding program run time for matching input values. However, the general shape that this graph exhibits (which should reflect the order of the time complexity function relative to which variables are being treated as constant) is of interest to us. We wish to examine any similarity between the trend/shape of the scatter plot containing the program run times (as the variable in question is varied) with the shape of the plot of this time complexity function.

The first aspect of our evaluation was the analysis of variation in the number of vertices whilst keeping the number of edges and colours the same. Treating $|C|$ and m as constants simplifies the time complexity function to $O(n^2)$. This is perhaps the most signifiant and interesting aspect of our evaluation, since the time complexity merely simplifies to being linear in each of the other two experiments. But here, the theoretically predicted time complexity predicts a quadratic relationship between the run time and the number of vertices (whilst the number of edges and colours remain constant). We can therefore analyse our data to specifically see if the program run time gets increasingly larger as the number of vertices increases, and if there is a polynomial relationship in this regard. Figure 2 shows an example plot corresponding to a variation in the number of vertices when the number of edges was fixed to 2000 and the number of colours was fixed to 15. Figure 3 shows a plot of the theoretical run time function across the relevant input values. Figure 4 shows a superposition of the curve in Figure 3 over the data in Figure 2.

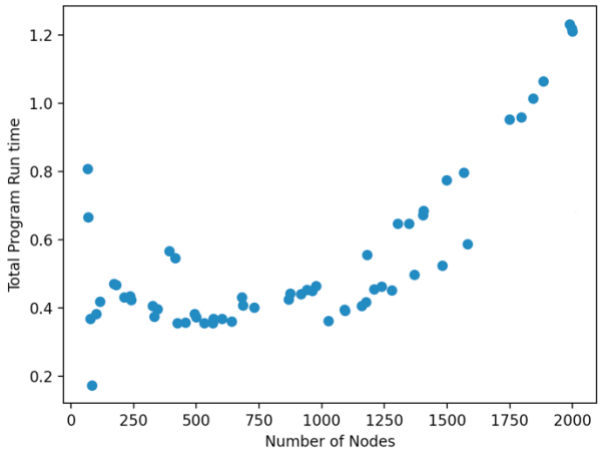

Fig 2. A scatter plot where the data points represent the total program run time (in seconds) of the implementation for input graphs with varying number of vertices.

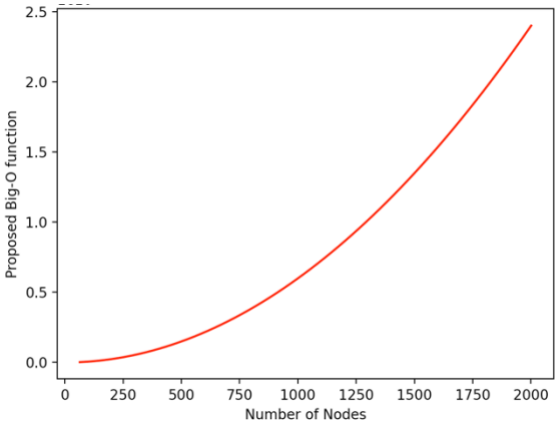

Fig 3. A plot of the function $\max\{|C| \cdot m \cdot n^2 , n(n+m)\}$, for fixed $|C|$ and m.

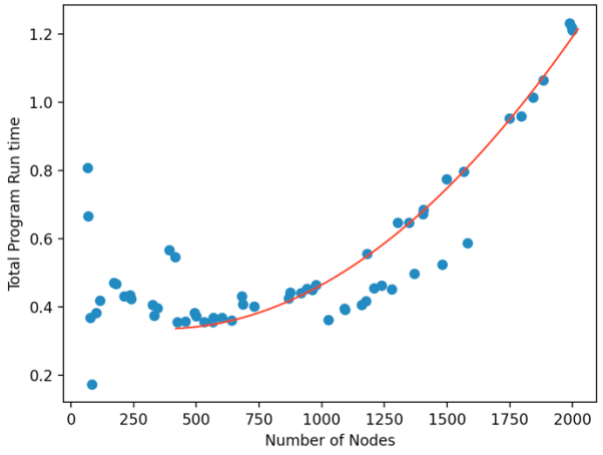

Fig 4. A superposition of the curve in Figure 3 over Figure 2, for visualising a comparison between the respective shapes of the two plots.

The second aspect of our evaluation was the analysis of variation in the number of edges whilst keeping the number of vertices and colours the same. Treating n and $|C|$ as constants simplifies the time complexity function to $O(m)$. Thus, the theoretically predicted time complexity predicts a linear relationship between the run time and the number of edges (whilst the number of vertices and colours remain constant). Figure 5 shows an example plot corresponding to a variation in the number of edges when the number of vertices was fixed to 20 and the number of colours was fixed to 15. Data representing the theoretical run time function with the relevant input values is superimposed onto the data. In this case, we thought the theoretically predicted run time function would be better represented in a scatter plot fashion. Recall its output value is the maximum between two values; in this case, there was no consistency across the data points between which of these is maximum, so unlike the previous case, the function would be poorly represented by a smooth curve. We also stress again that the superimposition here is done purely to compare the shapes of the two plots - the raw output values themselves from the two plots derive from 2 entirely different spectra.

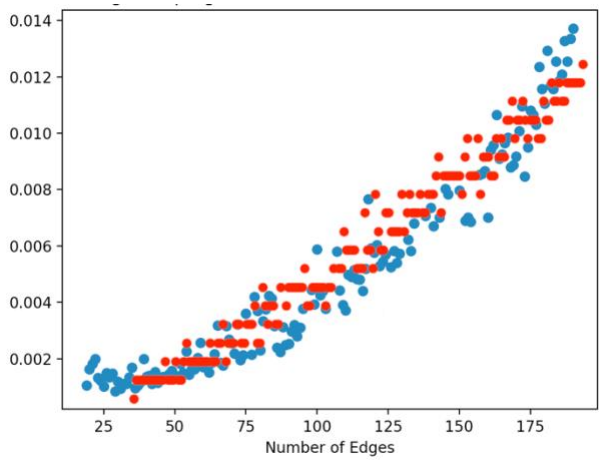

Fig 5. A scatter plot where the blue data points represent the total program run time (in seconds) of the implementation for input graphs with varying number of edges. The superposition of the plotted theoretically predicted run time function (in red) allows for visualising a comparison between the respective shapes that the two plots exhibit.

The third aspect of our evaluation was the analysis of variation in the number of colours whilst keeping the number of vertices and edges the same. Treating n and m as constants simplifies the time complexity function to $O(|C|)$. Thus, the theoretically predicted time complexity predicts a linear relationship between the run time and the number of colours (whilst the number of vertices and edges remain constant). Figure 6 shows an example plot corresponding to a variation in the number of colours on an input graph consisting of 100 vertices and 194 edges (the number of colours was therefore varied between 2 and 100).

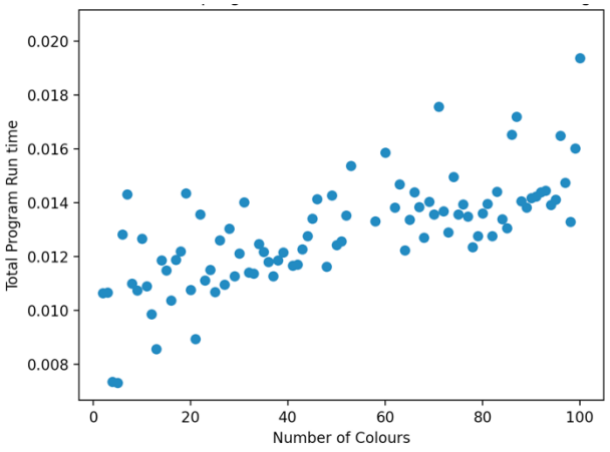

Fig 6. A scatter plot where the data points represent the total program run time (in seconds) of the implementation on a graph whose number of colours is varied.

5 Evaluation

For varying the number of vertices and keeping the numbers of edges and colours constant, the behavior of the program run time appears to roughly match its expected behavior as per the theoretically predicted run time. The run time increases at a greater rate as the number of vertices continues to increase. This is further backed up in Figure 4, where it appears the shape of the data representing the variation in program run time can be interpreted to closely match the shape of the theoretically predicted run time function. There are limitations in these results though, which is ultimately due to the difficulty in generating substantial data for this case (with regards to the method used, described in section 3.6). A large portion of the data points in Figure 4 are clumped together within a particular region, and there aren't too many data points representing vertex counts towards the higher end of the range. This somewhat weakens the conclusions that can be drawn from the data. Since we were running our function iteratively until we had reached a certain number of data points and plotting them thereafter; the number of vertices associated to each data point was random and not pre-defined. Thus, the ratio of isolated to dominating vertices in the generated graphs was probably such that there was a bias towards the generation of graphs whose number of vertices fell within this particular region. We could have possibly adjusted the function such that it would divide the range of possible vertex counts into incremental ranges, iteratively consider each range, and only accept generated graphs whose number of vertices fell within that range. Alternatively, we could have simply ran the function for even longer to generate more data points (although the majority of these probably would have fallen inside the same range too). But neither of these solutions would have been straightforward to apply; the data in Figure 2 alone took well over 24 hours to generate, and it didn't seem as if running it for any longer would generate many, if any, further data points. We could have changed the function such that it would accept generated graphs whose edges were within a particular range, such as between 1990 and 2010 instead of exactly 2000. This would have brought about efficiency improvements, but at the cost of accuracy in the results.

For varying the number of edges and keeping the number of vertices and colours constant, the behavior of the program run time appears to roughly match its expected behavior as per the theoretically predicted run time. Aside from the initial values where the number of edges is small, for which the variation in the program run time is minimal (understandably so: as at these values, a change in the number of edges is not causing the graph to become sufficiently more complex such that there would be a meaningful impact on the algorithm running time), the run time increases at a fairly linear rate as the number of vertices increases. This is further backed up in Figure 5, where it appears the shape of the data representing the variation in program run time can be interpreted to closely match the shape of the theoretically predicted run time function.

For varying the number of colours and keeping the numbers of vertices and edges constant, the behavior of the program run time is quite sporadic. The theoretically predicted time complexity in this case suggests that the run time should increase linearly as the number of colours increases. Although a general upwards trend can be seen in the data, it is not consistent enough to draw any concrete conclusions from. Looking at the data in Figure 6, it seems logical that this experiment should be ran again, but across a much larger range of colours. This is because, if there was indeed a linear trend in the run time as the number of colours increases, the impact of the data being sporadic as it is in Figure 6 would be outweighed by the general trend shown by the data. However, this would be very difficult to achieve. The challenge lies with the fact that in order to vary the number of colours between, say, 1 and 1000, the graph which the algorithm runs on would need to have 1000 vertices, in order to support the possibility of having 1000 colours. However, as we must keep the number of vertices and edges constant across every data point, this same 1000 vertex graph would be used to measure the program run time on every single data point, including those where the number of colours is very small. With this being such a large and complex graph, the algorithm would take a substantial amount of time to compute the maximum colourful cycle, and this length of time would be repeated on every single data point, thus making such an experiment infeasible. The graph could be made less complex by setting the number of dominating vertices to be very small, but this would not be particularly representatitve of a typical threshold graph of the given size, which in turn could undermine any potential results. We would conclude from Figure 6 that that the data possibly follows a linear trend, but we just cannot know in the absence of data corresponding to the number of colours getting considerably greater. The general variation in the program run time as seen in the data in Figure 6 is most likely attributed to the fact that the actual distribution of the colours is completely random from one data point to the next, even though the number of colours has only increased by one; the addition of the new colour is not done in a structured manner in that sense. Therefore, there

will be variation in terms of how the algorithms go about their computation: for example, there might be significantly more iterations in the maximum colourful cycle algorithm from one data point to the next. Such a variation would usually be insignificant when the input values get large enough, as the predicted time complexity function is described in terms of its limiting behavior, but we imagine that this variation is apparent in the data in Figure 6.

Considering all three experiments holistically, there is strong evidence that our implementation's performance behaves according to the algorithm's theoretical time complexity. However, in the absence of an analytical method to determine the program time complexity (which, as described previously, is impossible), the numerical method we employed for our evaluation has its limitations. Ultimately, the general trend in the data representing the variation in the program run time is only a very rough approximation of the nature of the program's time complexity. This is especially prevalent in our specific case where there are multiple independent variables which contribute to the time complexity. Even though we fix two of them constant, it is possible that the actual selected values for these constants could have a large influence on the generated data. For example, the program run time may increase at a faster rate when increasing the number of vertices if the number of colours in the graph is set to a particularly large constant. Although we were able to experiment with various permutations of parameters in this sense, which yielded some interesting results, it highlights the underlying restriction of this method in that it can only provide approximate insights towards the complexity. In each of the experiments, in order to determine whether the program run time behaves according to the algorithm's predicted time complexity function, we compared their respective shapes via a superposition, as seen in Figures 4 and 5. Future work could carry out further analysis to verify the behavior of the data. This could be achieved, perhaps, by running the data through a mathematical function; this might have been particularly applicable in the case when we were varying the vertices, as we could employ such a method to verify that the data is indeed behaving polynomially. Such a method could be designed such that it can verify a property of (in our case) quadratic functions, such as checking the gradient is linear, to ensure that the original function is indeed quadratic.

Overall, we are pleased with how this project went from an organisation and management perspective. At the beginning of the project, we identified a list of objectives, which were categorised into basic, intermediate and advanced deliverables. Where appropriate, larger, more complex objectives were broken down into multiple individual deliverables, whose collective completion would equate to the objective in question being met. We also considered objectives outside of the project implementation scope i.e. more administrative tasks required within the project, such as the preparation for the oral presentation in December. The 'basic' deliverables were based on the foundational objectives, which were considered mandatory towards a successful completion of the project, in line with the key project aims. In our case, the fundamental aim of this project was to implement Italiano et. al.'s algorithm for maximum colourful cycles in threshold graphs. Thus, the majority of the basic deliverables were tasks building up to the implementation, such as the implementation of the Blossom algorithm for computing a maximum matching in a graph. The 'intermediate' deliverables were based on objectives considered highly desirable. These typically required relevant basic objectives to be met as pre-requisites. An example was to perform an evaluation on our implementation, which would require a correctly working implementation as pre-requisite. The 'advanced' deliverables were based on objectives considered to be extension material; they typically required relevant intermediate objectives to be met as pre-requisite. They were planned to be worked on only if we had sufficient time to do so. An example was the investigation of any possible improvements to Italiano et. al.'s algorithm.

For each deliverable, we assigned an estimated length of time for completion. We then used a GANTT chart to plan a schedule of when these deliverables would be worked on across the year [24]. The 'basic' deliverables were heavily scheduled within term 1, as they were of the highest priority. The 'intermediate' deliverables were heavily scheduled within term 2, given their general requirement of various basic deliverables being met in order to begin working on. Advanced deliverables were not set aside their own time in the schedule; they were intended to be worked should other deliverables be completed ahead of their deadlines, thus creating additional time.

Ultimately, the project planned out well. All the basic and intermediate deliverables were met to their full extent. However, there was not sufficient time to work on any of the advanced deliverables. The organisation and management techniques adopted within the project, in terms of the breakdown into deliverables and subsequent organisation onto the Gantt chart, proved to be effective. It ensured our work fell within a rigid structure. It also maximised productivity, because each deliverable had its own deadline that ideally had to be adhered to in order to follow the structure of the project. It was unfortunate that we were not able to work on any of the advanced deliverables. This was primarily due to many of the basic and intermediate deliverables presenting unforeseen issues and challenges, often in the form of persistent bugs in our code. In some cases, for deliverables for which a successful outcome could not be explicity defined, we opted to continue working on that deliverable rather than moving on to, say, one of the advanced deliverables, even though the deliverable could be considered to have already been met. For example, we opted to spend further time on the performance evaluation of our implementation, experimenting with various different permuations of parameters and investigating results further. We would therefore say that the organisation and management techniques we used would be most effective when the deliverables' outcomes are as tangible and well-

defined as possible. We did allocate more time than expected for each deliverable, to account for unexpected issues, and this proved to be a good decision: even where on some deliverables, this extra time was not used, it was then carried over to another deliverable for which further time was required. If we were to do the project again, we would spend more time initially investigating each deliverable before assigning it a time period for working on. In our case, this would involve thoroughly reading through the relevant papers associated with each algorithm and forming a coherent plan of how it would be implemented. This way, we would have been better informed as to the possible challenges that might be associated to each deliverable: on a few deliverables, we only discovered the issue when we began working on it which slightly threw us off schedule (for example, a paper's description of an algorithm being unexpectedly complex to follow). If we had a larger schedule, we would have also liked to work on some of the advanced deliverables, which we were not able to find time for.

6 Conclusions

In summary, we have provided some background around vertex-coloured graphs and their typical applications within various fields, such as bioinformatics. We have described the motivation behind algorithms for the maximum colourful cycle problem, which can be asked of these graphs. We highlighted the importance of developing such theoretical algorithms into successful implementations, such that they can be applied in practical situations and environments; but equal to this, the importance of such implementations performing as expected with regards to their theoretically predicted time complexity, for which there are often requirements in real-world environments where inputs can get very large and unpredictable. An overview of key topics and terminology associated to problems that this project addresses and involves, including a survey of existing work and research in these areas, was provided. Each of the algorithms that our implementation uses are described in detail, followed by an overview of the technical and administrative details of our implementation. We discussed how our program is able to generate a threshold graph whose parameters (such as the ratio of dominating to isolated vertices, the overall number of vertices and the number of colours in the graph) can be easily fine-tuned to the operating user's liking, meaning that the algorithm can be run on any type of input required. A description of the methodology behind how we carried out an evaluation of our implementation's experimental performance is outlined. We then provided an overview of the results. The first aspect of this involved critiquing the validity and correctness of our implementation. We provided strong evidence pointing towards the view that our program does successfully output the correct maximum colourful cycle of an input threshold graph, regardless of the make-up of the graph or the nature of its parameters. This represented the first major element of our contribution. However, it is worth noting that ultimately, there is no absolute concrete method of verifying the correctness of the output, aside from the use of a computational brute force check of every single possible cycle on the graph to check if it is possible to construct one containing more colours. Regardless of the huge inefficiency of such a check, this method itself would also need to be verified for correctness if it were to be implemented. Our program does begin to run very slowly as the input graph gets considerably larger and more complex (which ended up being a limiting factor in our analysis of the program run time variation when varying the number of colours). However, this is ultimately to be expected; the resultant slowness in the program run time is due to the algorithm's intrinsic make-up and structure. The only improvement that could be made in this regard is the utilisiation of Micali et. al.'s algorithm for computing maximum matchings instead of the Blossom algorithm, given its superior time complexity [17]; this is something that could be implemented in future work. But ultimately, we wouldn't expect measures such as running the implementation on more sophisticated hardware, or the utilisation of more efficient technical implementation methods, to make a significant difference in this regard when the algorithm is run on increasingly larger inputs.

We then presented graphs showcasing our implementation's experimental performance in three different contexts: the variation in each of the number of vertices, edges and colours, and the subsequent effect on the program running time. For each of these three variables, we compared the behaviour of the run time to the theoretically predicted algorithm time complexity. Overall, these comparisons showed convincing signs that our implementation does appear to behave according to the predicted time complexity. This is the necessary result that one would wish to see should they intend to apply Italiano et. al.'s algorithm in a practical context, where their primary motive for selecting this particular algorithm might be its advantageous time complexity over other similar algorithms (such as a brute-force approach). Evidence that an implementation of the algorithm does indeed exhibit its proposed time complexity would then be crucial. As our experiments point towards this likely being the case for our implementation, our program itself could then theoretically be used in these practical contexts. This represents the second major element of our contribution.

Where applicable, the limitations of the results in each of our evaluative experiments were outlined. Our principal experiment involved varying the number of vertices whilst keeping the number of edges and colours constant. The largest limitation in our results for this experiment lied in the quantity of data we could generate. We faced a unique challenge of needing to iteratively generate threshold graphs where each one had the same number of edges, but the number of vertices varied. Our method to achieve this iteratively generated 'candidate' graphs until one with the correct number of edges was generated. Ultimately, this method became too inefficient as the target number of edges became very large (required in order to vary the number of vertices over a large enough range) because too much time was expended generating candidate graphs possessing the incorrect number of

edges. Future work could investigate the possibility of a numerical method in order to structurally increase the number of vertices in a threshold graph, iteratively, whilst keeping the number of edges constant throughout. This would ideally need to work on any threshold graph, regardless of the number of or positioning of its dominating and isolated vertices, so that fully representative data can be acquired. Since a threshold graph is defined by its chronological vertex make-up, which in turn dictates the edges of the graph, working in the opposite direction by initially fixing the edges is far from straightforward. However, if such a numerical algorithmic method could be identified, it could replace the 'trial and error' method that we used. This would bring about drastic improvement in the efficiency of data generation for this particular experiment. This would ultimately mean that significantly more data could be produced in this experiment, across a potentially larger range of vertices, which in turn would allow for more conclusive analysis of the behavior of the program run time as the number of vertices increases.

Other future work could address some of the advanced objectives that we established for this project, for which we didn't end up having time to work on. For example, further investigation into the implemented algorithm by Italiano et. al. could be undertaken to analyse whether any possible improvements could be made to improve its time complexity. In addition to an analysis of the algorithm from a theoretical standpoint to identify any such areas of improvement, it might be useful to use our implementation itself for a practical analysis. This could involve analysing which elements of the algorithm tend to consume the most time, considering why this is the case and examining how these specific elements could be improved. A practical implementation of the algorithm provides the opportunity to run the algorithm on a very large number of different input types; this could be taken advantage of in such a practical analysis; perhaps it could be identified that the algorithm could be improved for certain types of threshold graphs, for example.

Similar to our implementation of the algorithm for maximum colourful cycles on threshold graphs, the corresponding algorithm for bipartite chain graphs could also be implemented in future work (this algorithm was also presented by Italiano et. al. in the same paper [4]). Further from that, similar algorithms established in research that can run on other types of input graphs could be implemented. Perhaps even new algorithms could be designed (and subsequently implemented) to be run on further types of input graphs. Such implementations could be used in conjunction with the implementation we presented in this project for threshold graphs; holistically, this would represent a more versatile solution for practical scenarios where the input data might be modelled in various different graph formats.

REFERENCES

[1] Bruckner, S., Hu ̈ffner, F., Komusiewicz, C., Niedermeier, R.: Evaluation of ILP-based approaches for partitioning into colorful components. In: Inter- national Symposium on Experimental Algorithms. pp. 176–187 (2013)

[2] Corel, E., Pitschi, F., Morgenstern, B.: A min-cut algorithm for the consistency problem in multiple sequence alignment. Bioinformatics 26(8), 1015– 1021 (2010)

[3] Marx, D.: Graph colouring problems and their applications in scheduling. Periodica Polytechnica Electrical Engineering 48(1-2), 11–16 (2004)

[4] Italiano, G., Manoussakis, Y., Thang, N., Pham, H.: Maximum colorful cycles in vertex-colored graphs (2018)

[5] M. Yannakakis. Node deletion problems on bipartite graphs. SIAM J. Comput., 10:310- 327, 1981

[6] B. Chor, M.R. Fellows, M.A. Ragan, F.A. Rosamond, S. Snir, Connected coloring completion for general graphs: Algorithms and complexity, in: Proceedings of the 13th Conference on Computing and Combinatorics (COCOON), 2007, pp. 75–85.

[7] Mahadev, Peled: Longest cycles in threshold graphs. Discrete Mathematics 135(1-3), 169–176 (1994)

[8] Uehara, R., Valiente, G.: Linear structure of bipartite permutation graphs and the longest path problem. Information Processing Letters 103(2), 71–77 (2007)

[9] Cohen, J., Manoussakis, Y., Pham, H., Tuza, Z.: Tropical matchings in vertex-colored graphs. In: Latin and American Algorithms, Graphs and Optimization Symposium (2017)

[10] Turing, A.: ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEDIUNGSPROBLEM (1936)

[11] McCabe, T.: A Complexity Measure. IEEE Transactions of Software Engineering, Vol. SE-2, No. 4 (Dec 1976)

[12] Harrison, W., Magel, K.: A Complexity Measure Based On Nesting Level

[13] Kannan, S.K. and Warnow, T.J., 1992. Triangulating 3-colored graphs. SIAM journal on discrete mathematics, 5(2), pp.249-258.

[14] Edmonds, Jack. "Paths, trees, and flowers." Canadian Journal of mathematics 17.3 (1965): 449-467

[15] F. Harary, U. Peled, Hamiltonian threshold graphs, Discrete Appl. Math., 16 (1987), 11–15.

[16] Václav Chvátal and Peter L. Hammer. Aggregation of inequalities in integer programming. Annals of Discrete Mathematics, 1:145–162, 1977.

[17] Micali, S., Vazirani, V.V.: An O(√|V||E|) algorithm for finding maximum matching in general graphs. In: Proc. 21st Symposium on Foundations of Computer Science. pp. 17–27 (1980)

[18] V. Chvatal, On Hamilton's ideals, J. Combin. Theory (B) 12 (1972) 163-168.

[19] Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart, "Exploring network structure, dynamics, and function using NetworkX", in Proceedings of the 7th Python in Science Conference (SciPy2008), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11–15, Aug 2008

[20] J. D. Hunter, "Matplotlib: A 2D Graphics Environment", Computing in Science & Engineering, vol. 9, no. 3, pp. 90-95, 2007

[21] Beck. 2002. Test Driven Development: By Example. Addison-Wesley Longman Publishing Co., Inc., USA.

[22] Van Rossum, G. & Drake, F.L., 2009. Python 3 Reference Manual, Scotts Valley, CA: CreateSpace.

[23] Garey, M.R., Johnson, D.S., Stockmeyer, L.: Some simplified NP-complete problems. In: Proc. 6th Symposium on Theory of Computing. pp. 47–63 (1974)

[24] Clark, W., Polakov, W. N., & Trabold, F. W. (1923). The Gantt chart: A working tool of management. New York: Ronald Press.

[25] Foucaud, F., Harutyunyan, A., Hell, P., Legay, S., Manoussakis, Y. and Naserasr, R., 2017. The complexity of tropical graph homomorphisms. Discrete Applied Mathematics, 229, pp.64-81.

[26] Johanne Cohen & Giuseppe F. Italiano & Yannis Manoussakis & Nguyen Kim Thang & Hong Phong Pham, 2021. "Tropical paths in vertex-colored graphs," Journal of Combinatorial Optimization, Springer, vol. 42(3), pages 476-498, October.

[27] Fellows, M.R., Fertin, G., Hermelin, D. and Vialette, S., 2011. Upper and lower bounds for finding connected motifs in vertex-colored graphs. Journal of Computer and System Sciences, 77(4), pp.799-811.

[28] H.L. Bodlaender, L.E. de Fluiter, Intervalizing k-colored graphs, in: Proceedings of the 22nd International Colloquium on Automata, Languages and Programming (ICALP), 1995, pp. 87–98

[29] Heggernes, P. and Kratsch, D., 2007. Linear-time certifying recognition algorithms and forbidden induced subgraphs. Nord. J. Comput., 14(1-2), pp.87-108.

[30] Golumbic, M.C., 2004. Algorithmic graph theory and perfect graphs. Elsevier.

[31] Mahadev, N.V. and Peled, U.N., 1995. Threshold graphs and related topics. Elsevier.