Graph Coloring

Lucas Meijer[1]

lucas-meijer@kpnmail.nl
[1]Utrecht University, Department of Information and Computing Sciences

July 25, 2022

Abstract

We discuss many coloring problems. Most notably, we devise a modified algorithm for 3-COLORING that has a worst-case time bound of $\mathcal{O}^*(1.3236^n)$. We obtain this by adapting Beigel and Eppstein's time $\mathcal{O}^*(1.3289^n)$ algorithm. They found a structure in the graph that could be colored relatively easily called a maximal bushy forest, which we modify to limit the number of relatively difficult-to-color vertices by dividing the vertices in the graph using their relation to the bushy forest. Furthermore, we take a look at how a small vertex cover can be used to quickly find a k-coloring of a graph. Finally, we contribute new lower bounds for maximal induced k-colorable subgraphs, discuss the properties of these graphs and the upper bound on the number of maximal induced k-colorable subgraphs on graphs with these properties.

Contents

1 Introduction

Graph coloring: given a graph consisting of a set of vertices and a set of edges, each of which connects two vertices, assign a color to each vertex such that no edge connects two vertices with the same color. It is a fundamental problem within graph theory, allowing for a system of conflicts to be modeled. However, deciding the minimum number of colors such that a valid coloring exists for a graph is computationally difficult: GRAPH COLORING is one of Karp's [20] famous original NP-complete problems. This means that unless $P = NP$, it will require an exponential time algorithm to be solved. Furthermore, for any $k \geq 3$, it is NP-complete whether a given graph can be colored using k colors [26].

1.1 History of graph coloring

Graph coloring is a part of graph theory, which is thought to have originated in the city of Königsberg [3]: people wondered whether it was possible to cross the city's seven bridges in a single trip without crossing any bridge twice. Abstractly, this can be modeled by turning every landmass into a vertex of a graph, and every bridge into an edge. Then, the problem is whether there exists a path that visits every edge exactly once: the Eulerian path problem. Named after Leonard Euler, he was the first to show that this was not possible.

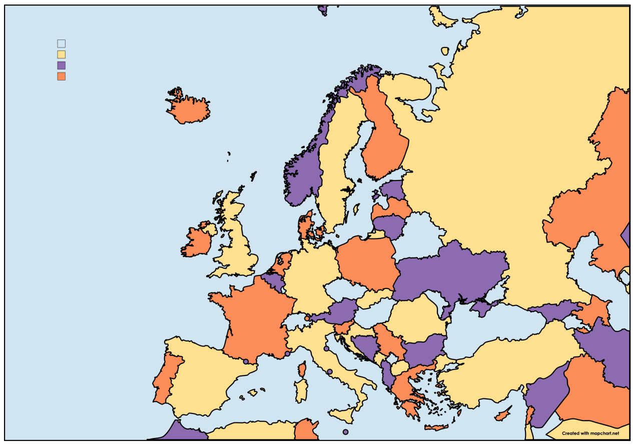


Figure 1: 4-colored map of Europe [15].

Graph coloring originated from a natural problem in cartography: given some map, how can we color each area to distinguish it, without two neighboring areas having the same color? In particular, we now know that any map can be colored using just four colors. As far as known, this was first observed by Francis Guthrie [8]. He had noticed that he could color the counties of England using four different colors. He told his brother Frederick, who was a mathematics student, and informed his professor Augustus De Morgan, which introduced graph coloring to academic circles.

This resulted in the four-color theorem: any planar graph can be colored using four colors. Here, a planar graph is any graph that can be drawn on a plane such that none of its edges intersect. While the conjecture was easy to encounter in practice, it demonstrated to be a difficult one to prove. The first supposed proof in 1890, by Alfred Bray Kempe, stood for ten years until it was disproved by Percy John Heawood. It took until 1976 for the theorem to finally be proven. This proof used over a thousand cases, for which it required computer assistance: no human was able to verify the correctness of the proof.

This four-color theorem sparked the first major research on graph coloring. From this, it was a logical step to extend planar graph coloring to coloring any given graph: determining the minimum number of colors such that a proper coloring of some graph exists with that number of colors.

1.2 Use of graph coloring

Abstractly, graph coloring can model a system of conflicts. Every element in the system turns into a vertex, while two vertices are connected by an edge if the combination of the two would result in a conflict. Then, every color class results in a set of vertices that do not conflict with each other. Furthermore, developing our knowledge of graph coloring may also improve our theoretical understanding of related fields in graph theory. Since we know that all NP-complete problems can be reduced to each other, perhaps improvements to graph coloring will affect other NP-complete problems, or simply inspire new improvements.

We will now describe some use cases of graph coloring in practice. One such application of graph coloring is when we are tasked with solving a scheduling problem. For example, when a school schedules its courses, some courses may not be scheduled at the same time due to room, student, or teacher availability [24]. As such, the chromatic number of the respective graph is the minimum number of time slots required. Similarly, the solution of a k-coloring algorithm shows whether k time slots can schedule all courses successfully, and if so, what this schedule looks like. Using this same method, we can avoid conflicts in any type of scheduling problem, such as assigning airplanes to flights [4].

A practical application outside scheduling problems is the problem of register allocation [24]. Compilers map each variable to one of the registers, but a register cannot hold variables that can be active at the same time. Hence, a coloring of the variable-conflict graph will determine how to assign the variables to the registers to avoid conflicts.

Graph coloring, as a model for finding sets that do not conflict, is rudimentary in a way that new applications can be found in emerging fields. For example, in the emerging field of bioinformatics, there has been research on applying graph coloring to protein-protein interaction networks. These networks that model cells would benefit from a low chromatic number, as it would allow the cells to perform many tasks at the same time [22].

A Latin square is a square with n rows and n columns, where each cell is represented by one of n symbols, without any row or column containing more than one of each symbol. While there exists a solution to empty Latin squares of any size, determining whether one exists for a partially filled square is shown to be NP-complete [9]. Some practical applications of Latin squares are their former use in cryptography, statistical experiments, and tournament scheduling [21]. These squares can be modeled as a graph coloring problem by denoting any cell as a vertex and by connecting any two cells that cannot hold the same symbol by an edge. Then, any solution may not have two cells in the same row or column holding the same symbol. One way to encode partially filled-in squares in the graph is by adding n dummy vertices representing the different symbols. These should all be connected, so they will all have different colors. Then, every filled-in cell should connect to all dummy vertices except the one that represents its symbol. This way, they must receive the same color as the vertex representing its symbol.

A notable example of a Latin square is the Sudoku, which has an additional restriction that for an n-by-n Sudoku, any two cells in the same block of size $\sqrt{n}$ by $\sqrt{n}$ may not hold the same number [11]. Like the previous ones, this new constraint indicates that specific cells cannot contain the same symbol. So, a Sudoku can also be solved using a graph coloring approach.

1.3 State of graph coloring

Within graph coloring, the minimum number of colors for which there exists a valid coloring of the graph is known as the chromatic number. Determining the chromatic number of a graph can be decided in time $\mathcal{O}^*(2^n)$ [5]. The algorithm works by iterating over all subsets of vertices: for each one, we find the number of maximal independent sets that do not intersect with the given subset. Further details on this problem will be given in Section 3.

A logical next step within graph coloring was to consider the decision variant of the chromatic number: given a value k and a graph G, can G be colored with only k colors. Evidently, we can always use the algorithm for the chromatic number for this. However, better algorithms are known for any $k \leq 6$. For $k \leq 2$, polynomial time algorithms are known:

1. 0-Coloring: possible if and only if no vertices exist.

2. 1-Coloring: possible if and only if no edges exist.

3. 2-COLORING: possible if and only if the graph is bipartite.

Clearly, 0-COLORING and 1-COLORING are possible in $O(1)$ time. Checking whether a graph is bipartite can also be done in polynomial time: greedily give some vertex one color and color all its neighbors with the other color. Keep coloring vertices with some neighbor colored, until no such vertices exist. Then, either the graph is completely colored, or the graph was disconnected. In the latter case, we can randomly assign one vertex in the remaining graph a color and repeat the algorithm. If this algorithm encounters a vertex that has two neighbors with different colors, the graph is not bipartite. Otherwise, it will find a valid 2-coloring of the graph.

The remaining problems are more complicated, as they require exponential time. In fact, it is known that even 3-COLORING is NP-complete: Lovász [26] showed that unless $P = NP$, there cannot exist a polynomial time algorithm for 3-COLORING.

The best algorithm for the 3-COLORING problem, previous to the improvements presented here, was a time $\mathcal{O}^*(1.3289^n)$ algorithm by Beigel and Eppstein [2]. This algorithm uses a 3-COLORING instance as a (3,2)-CONSTRAINT SATISFACTION PROBLEM instance, where the problem is turned into a set of possible color assignments for each vertex and constraints indicating that two adjacent vertices may not both receive the same color. This allows it to maintain the property that neighboring vertices cannot both be assigned the same color. We will explain this (3,2)-CONSTRAINT SATISFACTION PROBLEM algorithm in Section 5. Then, they find a set of vertices in the graph with many neighbors. When these vertices are given a color, their neighbors become easy to color, allowing them to be colored more quickly than by using the (3,2)-CONSTRAINT SATISFACTION PROBLEM algorithm. The combination of these two ideas leads to their final time $\mathcal{O}^*(1.3289^n)$ algorithm. We will improve upon the 3-COLORING algorithm in Part II, where we will find an improved set of vertices with many neighbors, allowing us to further reduce the time it takes to find the color of various types of vertices.

For 4-COLORING, the best algorithm known runs in time $\mathcal{O}^*(1.7272^n)$ [13]. This algorithm combines two techniques: the enumeration of independent sets and pathwidth. What these entail will be explained in Section 6. The idea of this is that any graph will have properties that will make either of these two methods favorable, leading to an improved runtime compared to only using one of the two concepts.

Finally, we know minor improvements for 5-COLORING and 6-COLORING over k-COLORING: they can be solved in time $\mathcal{O}^*((2-\epsilon)^n)$ [31]. The algorithm for these improvements finds either a large independent set with few vertices adjacent to multiple vertices in the independent set, or a small dominating set. For the independent set, they quickly colored the remainder of the graph and extended it to the independent set. On the other hand, a small dominating set can be colored to remove at least one color as an option for every other vertex in the graph, which can then be solved using $(k-1)$-COLORING.

Many more coloring problems exist. We will mention two noticeable problems: k-LIST COLORING and EDGE COLORING. For k-LIST COLORING, we still want to assign a color to each vertex, but there is not a global domain of k colors that every vertex can be assigned. Instead, every vertex has a list of at most k colors that can be assigned to it. Currently, this can be solved in time $\mathcal{O}^*((2-\epsilon)^n)$ for $k \leq 4$ by using the algorithms for (3,2)-CONSTRAINT SATISFACTION PROBLEM and (4,2)-CONSTRAINT SATISFACTION PROBLEM like k-COLORING [2]. The other problem, EDGE COLORING, involves coloring the edges of a graph instead of the vertices. As such, no vertex can be incident to multiple edges of the same color. A notable result in this field is Kowalik's paper showing that 3-EDGE COLORING, where every edge receives one of three colors, can be solved in time $\mathcal{O}^*(1.344^n)$. To do this, he employs maximal matchings, as he shows any valid solution can be represented as a special kind of maximal matching.

1.4 Structure

This thesis will consist of three parts. Firstly, in Part I, we will take a closer look at some problems closely related to 3-COLORING: calculating the CHROMATIC NUMBER (k-COLORING) in Section 3, calculating the largest possible amount of maximal independent sets in a graph in Section 4, solving the (3,2)-CONSTRAINT SATISFACTION PROBLEM and (4,2)-CONSTRAINT SATISFACTION PROBLEM (which are used as a black box for 3-COLORING) in Section 5 and (4,2)-CSP), and finally 4-COLORING in Section 6.

Then, in Part II, we show our improvement to 3-COLORING, which improves the runtime from time $\mathcal{O}^*(1.3289^n)$ to time $\mathcal{O}^*(1.3236^n)$. The new algorithm adapts the previous best algorithm for 3-COLORING, which is explained broadly in Section 7. We adapt some lemmas from the old algorithm to fit our purposes in Section 8. Then, in Section 9 we present our strategy to find a structure in any graph to assist the coloring process, whereas in Section 10 we partition the vertices to analyze their runtime separately using the structure we found. At last, in Section 11 we formulate a linear program to discover the worst-case runtime for the algorithm, while in Section 12 we summarize the new algorithm.

Lastly, in Part III, we present some smaller results and observations. In Section 13, we discuss how we can use a vertex cover to color a graph quickly. Here, we present a new concept to solve GRAPH COLORING more quickly given a vertex cover, when we have a large number of colors. In Section 14, we discuss maximal induced k-colorable subgraphs and their applications to graph coloring. For this problem, we present new lower bounds when $k > 2$ and discuss an upper bound for any k on graphs where every vertex is part of a group of k twins.

2 Preliminaries

In this section, we will introduce the concept of the work factor: a method of calculating the runtime of an exponential time algorithm, which we will use within this thesis. Furthermore, we will explain the notation that we use. Finally, we go over some rudimentary concepts in graph and complexity theory, such that it will be clear what they mean to those who do not know these ideas.

2.1 Notation

We will mostly be discussing algorithms on graphs: these algorithms will handle an instance graph G, which consists of a set of vertices V and a set of edges E. Furthermore, given some collection of vertices X, the graph $G[V - X]$ is the graph created when removing the vertices of X and all edges incident to at least one vertex in X from G.

We will denote the instance size of the instance of an algorithm as n: usually, n will equal the number of vertices $|V|$. If not, we will explicitly specify how n should be calculated instead. In some cases there are vertices that do not contribute to the instance size at all, we will avoid using n to show the instance size and will always explicitly show the instance size instead.

Many problems in graph theory require exponential time to solve. The big $\mathcal{O}$ notation denotes the asymptotic upper bound it can take for an algorithm to solve some problem. However, we will often use $\mathcal{O}^*$ instead, where the * indicates that polynomial factors in the number of vertices are ignored. For example, if an algorithm takes time at most $\mathcal{O}(n^2 \cdot 2^n)$, then this would turn into time at most $\mathcal{O}^*(2^n)$. We choose to ignore these terms, as the exponential term in these algorithms will cause the polynomial term to become negligible when the instance size becomes sufficiently large. Conversely, we will use Ω to denote an asymptotic lower bound.

2.2 Work factor

Most algorithms that we discuss will require exponential time to solve: they require worst-case time $\mathcal{O}^*((1+\epsilon)^n)$. On the contrary, some problems require polynomial time: algorithms that run in time $\mathcal{O}(n^c)$ for some constant c.

Oftentimes, we will formulate exponential algorithms through branching rules, which create multiple branches that each reduce the instance size n. A branching rule finds a set of smaller graphs such that if the original instance has a valid 3-coloring, at least one of the graphs in the branches will contain a 3-coloring as well.

For this, we need to be able to analyze the runtime of these branches to bound the runtime of the overall algorithm. For our algorithm, the instance size mentioned will refer to the number of vertices in the graph that can be colored any of the three colors.

Definition 2.1 (Work factor)**.** The work factor, $\lambda(r_1, r_2, \ldots, r_k)$, is a list of numbers representing a set of branches, where each number represents the reduction in instance size (the number of vertices) within that branch. The work factor is equivalent to a multiplicative cost per vertex: solving $c^n = \sum_{i=1}^{k} c^{n-r_i}$ for c.

An algorithm with branching rules and corresponding work factors $\lambda_1, \lambda_2, \ldots, \lambda_l$ runs in time $\mathcal{O}^*(c^n)$, if and only if every possible branching rule solves to at most a multiplicative cost per vertex c. So, this algorithm will take time $\mathcal{O}^*(\max_{i=1}^{l}(\lambda_i^n))$.

2.3 Concepts in graph theory

A graph G can contain a subgraph G': we can choose a subset of vertices $V' \subseteq V$ and a subset of edges $E' \subseteq E$, such that V' and E' form graph G'. Furthermore, we call a subgraph *induced* when every edge with both endpoints in V' is included in E': only edges are removed of which an endpoint has been removed. We call two vertices *neighbors* or *adjacent* when there exists an edge where the two vertices are the two endpoints. A vertex is *incident* to an edge when the vertex is one of the endpoints of the edge.

One important graph that we will find as a subgraph is a *tree*: a connected graph that does not contain any cycles. Such a graph will always contain $n-1$ edges, as any graph with $n-2$ edges will not be connected, while a graph with n edges must contain a cycle. As such, it must contain at least two vertices with only one neighbor. These vertices are called *leaves*. Furthermore, a *forest* is a disconnected graph consisting of only trees. So, it is any graph that does not contain any cycles.

Another special type of graph is the *complete* graph: a graph where there exists an edge between every pair of vertices. We denote a complete graph on x vertices as the graph K_x. Furthermore, we also have the *cycle* graph: any graph where the vertices form a cycle, so every vertex has exactly two neighbors. The cycle graphs are denoted by C_x, where x again denotes the number of vertices in the graph. Finally, there are the *cluster* graphs: a graph that consists of a disjoint number of complete graphs. In cluster graphs, the neighbors of any vertex will also be neighbors of each other.

Then, we have some sets of vertices with special properties in the graph. We say a set of vertices is an *independent set* when there does not exist any edge connecting two vertices in the independent set: none of the vertices are adjacent. Inversely, a set of vertices is a *clique* when there exists an edge between every two vertices in the clique: all of the vertices are adjacent. Notable, when coloring a graph, any color class in a valid solution must be an independent set, as no vertex may be adjacent to a vertex of the same color.

Another special set of vertices is the *dominating set*: for every vertex in the graph, it must either be included in the dominating set or have a neighbor in the dominating set.

For these sets of vertices, they can be *minimal* or *maximal*. When a set of vertices is minimal, no vertex can be removed from the set while maintaining its property. On the other hand, when it is maximal, no vertex can be added to the set while remaining its property.

Part I
Background Information

3 Chromatic Number

Determining the Chromatic Number entails finding the minimum number of colors, such that there exists a valid coloring for a given graph. As such, it is one of the most generic problems within the scope of graph coloring. Trivially, this would take $\mathcal{O}^*(\chi^n)$ time, where χ is the chromatic number of the graph: for every number $1 \leq i \leq \chi$ consider every possible coloring with i colors until a valid coloring is found. For every number of colors i, there exist i^n possible colorings of the graph, so this idea takes time $\mathcal{O}^*(\sum_{i=1}^{\chi}(i^n) \leq \chi \cdot \chi^n) = \mathcal{O}^*(\chi^n)$.

Currently, the fastest known algorithm for the chromatic number runs in time $\mathcal{O}^*(2^n)$. Surprisingly, this is the best algorithm for deciding whether a graph is k-colorable for any $k \geq 7$. Within graph problem, it is an important open problem whether k-coloring for some or all $k \geq 7$ can be decided within time $\mathcal{O}^*((2-\epsilon)^n)$. The $\mathcal{O}^*(2^n)$ time algorithm was created by Björklund et al. [5]. They partition the graph into independent sets through the principle of inclusion-exclusion: for every subset of vertices, they find how many maximal independent sets avoid this subset. Then, they can calculate how many combinations of k maximal independent sets avoid no vertices. Since there are 2^n subsets of vertices, and the values corresponding to them can be calculated in time $\mathcal{O}^*(2^n)$, the algorithm takes time $\mathcal{O}^*(2^n)$.

In this section, we will describe Björklund et al.'s algorithm: the combinatorial principle of inclusion-exclusion, how this is used to calculate the chromatic number, and how we use dynamic programming to find the values needed for this calculation.

3.1 The principle of inclusion-exclusion

The principle of inclusion-exclusion is based on one simple concept in combinatorics [1], for any sets A and B:

$$|A \cup B| = |A| + |B| - |A \cap B|.$$

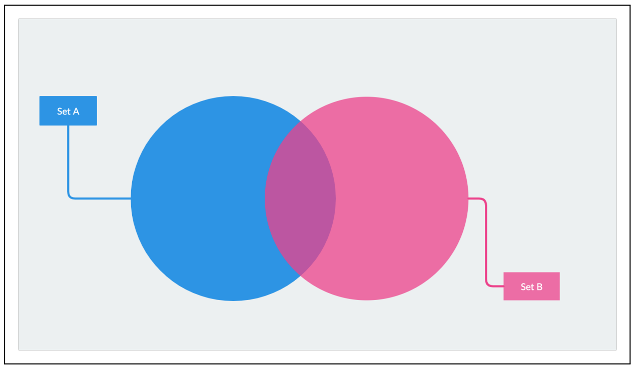


Figure 2: Simple Venn diagram.

We can easily verify this with a simple Venn diagram, such as the one in Figure 2. When counting both set A and set B, we count the area of $A \cap B$ twice. So, it needs to be deducted from the sum to correctly calculate the size of $A \cup B$.

Now, we can extend this formula beyond two sets, where we instead count the number of elements that appear in none of the sets $A_1, \ldots, A_n \subseteq B$. Say we have n sets $A_1, A_2, \ldots, A_n$, where $\bigcap_{i \in \emptyset} A_i = B$:

$$|\bigcup_{i=1}^{n} \bar{A}_i| = \sum_{X \subseteq \{1,\ldots,n\}} (-1)^{|X|} \cdot |\bigcap_{i \in X} A_i|$$

We can confirm the validity of this formula by considering all elements that exist only in one set: they will be counted only by that one set. Then, all elements that appear in two sets will be counted twice by those singular sets, and removed once by the intersection of those sets. For elements that appear in three sets, we see they are counted thrice by singular sets, also counted three times by the intersections of two out of three sets, and added once more by the intersection of all three sets at once. This will always continue this way, as $\sum_{i=1}^{n}(-1)^{i+1}\binom{n}{i} = 1$: adding the number of sets with an uneven number of elements, and subtracting the number of sets of uneven elements always sums to one.

3.2 Computing a set cover of independent sets

We use the principle of inclusion-exclusion to compute the number of combinations of k maximal independent sets that cover the entire graph: any such cover will produce a valid coloring. We assign each vertex the color of some maximal independent set that covers it. If a vertex is covered by multiple maximal independent sets, both colors will lead to a valid coloring. Clearly, each color class will now be a subset of a maximal independent set: an independent set. So, if the number of combinations of k maximal independent sets that cover the entire graph is larger than zero, we know that the graph is k-colorable.

Given the set of maximal independent sets $|\mathcal{I}|^k$ of a graph, there will be $|\mathcal{I}|^k$ possible combinations of maximal independent sets, where we choose to use the same maximal independent set multiple times. We will also allow the same combination of maximal independent sets to be selected in different orders. This represents the independent sets receiving different colors. However, this will not increase the number of colorings when there does not exist any, so we can validate whether the graph is k-colorable when the number of colorings is zero, nonetheless.

We will count using the number of maximal independent sets that avoid all vertices in some set $X \subseteq V$. Indeed, the combination of k maximal independent sets avoids the vertices in X if and only if all k maximal independent sets avoid the vertices in X. First, consider $X = \emptyset$: every combination will count, so all $|\mathcal{I}|^k$ combinations of maximal independent sets will be counted. However, this will count all the combinations that exclude some vertex v: we remove all combinations that avoid the singletons. Then, this will exclude every combination avoiding some two vertices multiple times, so these need to be added again. As such, the principle of inclusion-exclusion emerges, and we can use the following calculation to determine whether some graph is k-colorable, where $a(X)$ is the number of maximal independent sets that avoid the vertices in X:

$$\sum_{X \subseteq V} (-1)^{|X|} \cdot a(X)^k.$$

Clearly, there are 2^n subsets of vertices in the graph. Given the values $a(X)$, it will take polynomial time per subset to calculate the result: this will take time $\mathcal{O}^*(2^n)$. However, calculating $a(X)$ naively would take time $\mathcal{O}^*(2^n)$ per set X and time $\mathcal{O}^*(4^n)$ over all 2^n possible sets X. However, these values of $a(X)$ can be calculated using dynamic programming using the following recurrence relation, where $v \nsubseteq X$, $N(v)$ is the set of neighbors of v, and $N(v)$ contains at least one vertex not in X:

$$a(X) = a(X \cup v) + a(X \cup N(v)), v \notin X, N(v) \nsubseteq X.$$

Notice that the maximal independent sets that avoid X must either avoid v or all neighbors of v: any maximal independent set containing v cannot include any neighbors, nor can one include v when it includes a neighbor. Furthermore, any maximal independent set avoiding all neighbors of v must include v. Otherwise, it would not be maximal.

Notice that this formula would refer to itself if X includes either v or all vertices in $N(v)$. As such, the base case will be any set of vertices X such that there does not exist any edge in the graph with both endpoints outside X. For these cases, we know that $V - X$ is an independent set, as there are no edges between any of the vertices. We verify whether it is also a maximal independent set: there will be one maximal independent set avoiding X. Otherwise, there will be zero. Then, we can calculate for every other subset of vertices how many maximal independent sets avoid that subset: we can iterate over all subsets from the largest cardinality to the smallest: when calculating $a(X)$, we will only use subsets with a larger cardinality than X.

As both the base cases and the recurrence take polynomial time, the runtime of this algorithm is bound by the number of subsets: it will take time $\mathcal{O}^*(2^n)$ as well.

4 Maximal Independent Sets

Remember that an independent set is a subset of the vertices of a graph, such that there does not exist an edge between any two vertices in this set. An independent set is maximal when no vertex can be added such that it remains an independent set. Notably, in any coloring of a graph, all vertices assigned the same color must form an independent set. Otherwise, two adjacent vertices would receive the same color. Many intuitive coloring algorithms use this property: we can iterate over maximal independent sets to determine which vertices belong to one vertex class, and then color the remaining graph with the remaining colors.

We will describe how we know that there exist at most $3^{n/3}$ maximal independent sets in any graph. It is known that we can iterate over the maximal independent sets of the graph in time polynomial to the number of maximal independent sets [19]: iterating over maximal independent sets will take at most time $\mathcal{O}^*(3^{n/3})$.

4.1 Moon & Moser's argument

In 1965, Moon and Moser [27] showed that the number of maximal cliques is at most $3^{n/3}$. Every (maximal) independent set in a graph G will become a (maximal) clique in the transpose graph G' where two vertices are adjacent if and only if they are not adjacent in G: no vertices in an independent set in G are adjacent, so they will all be adjacent in G'. For a graph G, let $c(G)$ be the number of maximal cliques in G. For some vertex v, let $\chi(v)$ be the number of cliques in G involving v. Finally, let $\alpha(v)$ be the number of cliques in G involving v that would still be maximal without v in $G[V-v]$. Let $\beta(v)$ be all cliques in G involving v that would not be maximal without v in $G[V-v]$. Evidently, we see that:

$$\alpha(v) + \beta(v) = \chi(v)$$

Furthermore, we can also calculate how many cliques exist in G after the removal of v by removing those cliques that would not be maximal anymore after the removal of v:

$$c(G-v) = c(G) - \beta(v)$$

Now, we will define an operation on any two non-adjacent vertices in a graph that can always be applied without decreasing the number of maximal cliques in the graph. Given a graph G and two non-adjacent vertices v and w, define $G(v;w)$ as the graph in which vertex v is removed and replaced by a vertex w' with the same set of neighbors as w. As the two vertices are non-adjacent, the number of cliques w is involved in will not change, while the removal of v removes $\beta(v)$ maximal cliques: the cliques it was involved in that are not maximal anymore. Meanwhile, w' has the same neighbor set as w, so it will be involved in the same number of maximal cliques: we add $\chi(w)$ new maximal cliques to the graph. Now, we can see that:

$$c(G(v;w)) = c(G-v) + \chi(w) = c(G) - \beta(v) + \chi(w)$$

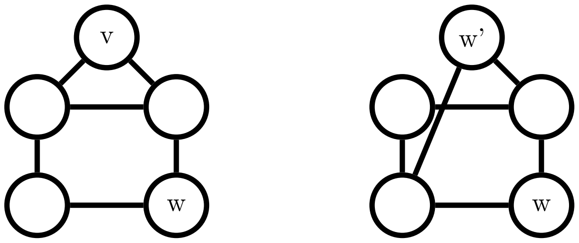


Figure 3: Replacing vertex v with a copy of vertex w which is involved in more cliques. Notice that the number of maximal cliques increases from four to six as $\beta(v) = 0$ and $\chi(w) = 2$.

Let $\chi(v) \leq \chi(w)$, as $\beta(v) \leq \chi(v)$, we can always perform operation $G(v;w)$ without decreasing the number of cliques in the graph (if $\chi(v) > \chi(w)$, then we can perform operation $G(w;v)$). Given two adjacent vertices, we can always perform this operation without decreasing the number of maximal cliques. So, we can perform this operation until the graph cannot change through this operation. This implies that any two vertices must either be adjacent, or they must have the same set of neighbors, such that the operation does not change the graph.

Suppose the resulting graph (where $G(v;w)$ never changes the graph) has t groups of vertices with the same neighbor set. Any vertex must be adjacent to all vertices in other groups, while not being adjacent to any vertex in its own group. We can create a maximal clique by picking exactly one vertex from each of these t groups. Consider that the t groups have $j_1, j_2, ..., j_t$ vertices contained in them: there will be $\prod_{i=1}^{t} j_i$ cliques, where $\sum_{i=1}^{t} j_i = n$. This formula is maximized when every group has exactly three vertices, meaning there are at most $3^{n/3}$ maximal cliques, as in all of the $3^{n/3}$ groups one of three vertices has to be chosen.

This argument also finds a lower bound on the number of maximal independent sets, as the graph described with $3^{n/3}$ groups of independent sets of size 3 will have $3^{1/3}$ maximal cliques: we know this upper bound is tight.

4.2 Branching argument

We can also use a simple branching rule, which is folklore, to find the same upper bound of at most $3^{n/3}$ maximal independent sets existing in any graph. Consider a vertex of minimum degree d in the graph. In any maximal independent set, we either include this vertex or include one of its neighbors. This leads to $1 + d$ branches, each of which removes at least $1 + d$ vertices from the instance: each instance removes the vertex included and all of its neighbors, which is at least $1 + d$ (as d is the minimum degree). We express this as a work factor of $\lambda((d+1)_1, (d+1)_2, ..., (d+1)_{d+1})$, which represents $d+1$ branches reducing the instance size by at least $d+1$: $\lambda((d+1)_1, (d+1)_2, ..., (d+1)_{d+1}) = (d+1)^{1/(d+1)}$. This is maximized at $d = 2$, showing the upper bound of maximal independent sets in any graph is $3^{\frac{n}{3}}$.

5 (3,2) and (4,2)-constraint satisfaction problem

The CONSTRAINT SATISFACTION PROBLEM (CSP): given a set of variables, assign to each variable some value (which we will refer to as colors), such that none of the given constraints are violated. Each of these variables has a list of colors it can be assigned: we will call these combinations of variables and colors *variable-color pairs*. A constraint contains some number of variable-color pairs: no valid assignment can contain all of those variable-color pairs. Specifically, the (k,d)-CONSTRAINT SATISFACTION PROBLEM (or (k,d)-CSP), considers a CONSTRAINT SATISFACTION PROBLEM where the number of possible colors per variable is at most k and every constraint contains at most d variable-color pairs.

Naively, the (k,d)-CONSTRAINT SATISFACTION PROBLEM can be solved by iterating over all possible variable-color pairs. The instance size n denotes the number of possible variables and each variable has at most k possible colors, so iterating over all possible combinations and verifying whether no constraints are violated takes time $\mathcal{O}^*(k^n)$. The fastest known algorithm for (k,d)-CSP adapts this idea: by randomly guessing a solution, we expect some variable-color pairs to be correct. When we explore similar solutions. By exploring local solutions, we improve the chance of finding a valid solution, if one exists, over randomly assigning colors. By repeatedly executing this randomized algorithm, we expect to solve (k,d)-CSP in time $\mathcal{O}^*((k - \epsilon_{d,k})^n)$ [17].

However, in this section we will discuss two cases for which a faster algorithm is known: the (3,2)-CSP and the (4,2)-CSP [2]. Notably, Beigel and Eppstein use this algorithm within their 3-COLORING algorithm, for which it provides important context.

Notice that the $(k,2)$-CSP algorithm can model k-COLORING without increasing the instance size: turn every vertex into a variable, consider each of the k colors a valid possible color for every variable. Per edge, create a constraint for each color, indicating that the two adjacent vertices may not both have said color. Notice that any solution to k-COLORING will correspond to a solution to $(k,2)$-CSP and vice versa: simply assign the same variables the same colors. Figure 4 shows an edge in a 3-COLORING instance and a (3,2)-CSP modeling this edge. Furthermore, $(k,2)$-CSP also models k-LIST-COLORING: $(k,2)$-CSP does not require that the colors of all variables are the same, so we can instead add constraints only between variable-color pairs of adjacent vertices when said color appears is a possibility for both vertices.

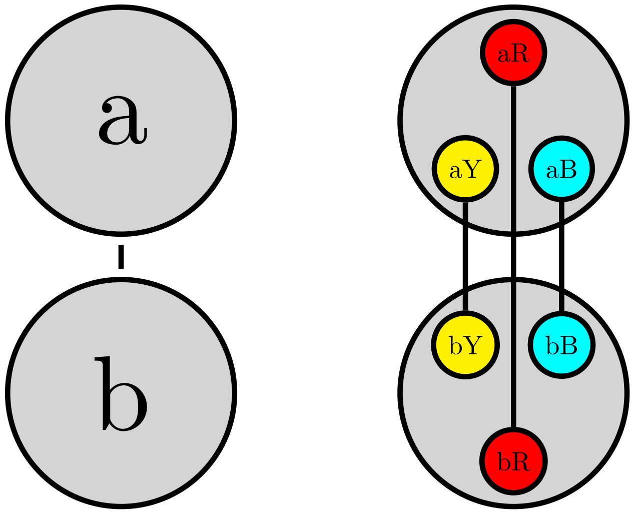


Figure 4: Modeling an edge from a 3-COLORING (left) instance to an edge in a (3,2)-CSP (right) instance. Notice that the edges in the (3,2)-CSP represent constraints: the two linked variable-color pairs may not both exist in any solution.

The algorithm consists of three main sections. First, it contains polynomial time reduction rules. These reduction rules take an instance containing a specific substructure and create a singular other instance, which contains a solution if and only if the original instance has a solution. Furthermore, there are reduction rules that require exponential time: these cases create multiple new instances that need to be explored. If and only if one of these instances contains a solution, the original instance will contain a solution. These cases taking exponential time can be analyzed using the work factor as described in Section 2, which can track the instance size reduction in each of the branches.

Finally, the algorithm will converge to a state where none of the previous reduction rules apply. In this case, we will be able to solve the remaining instance in polynomial time through a flow algorithm.

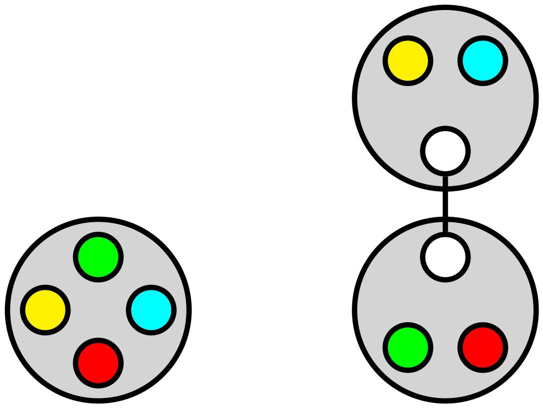


Figure 5: Transforming a variable with four possible colors (left) into two variables with three possible colors (right). The white vertices only appear in one constraint: we choose one white color and one of the other four colors. Notice that the other four colors can appear in any number of constraints with other variables, but those are not drawn here.

Now, we will discuss the relationship between (3,2)-CSP and (4,2)-CSP. First, notice that the (4,2)-CSP can solve any instance of the (3,2)-CSP as well: the only difference is the maximum number of possible colors per variable. Hence, this algorithm will solve (4,2)-CSP and solve (3,2)-CSP through it. However, variables with four possible colors will count more heavily in the instance size of the problem than ones with three possible colors. Let n_x be the number of vertices with x colors. We will calculate the instance size

n as follows: $n = n_3 + (2 - \epsilon) \cdot n_4$. Naturally, we could express a variable with four possible colors as two variables with three colors as shown in Figure 5, which indicates why we want to weigh variables with four possible colors approximately twice as heavily. However, the variable $\epsilon \approx 0.095543$ was added to optimize the time the various branching rules of the algorithm take.

5.1 Polynomial time reductions

Indeed, variables with one or two possible colors will not count towards the instance size. If there exists a single possible color, the variable must be assigned this color in any valid coloring. When a variable has two possible colors, we can find an instance that leads to the same solution that does not contain this variable. This is an example of a polynomial time reduction: when some pattern exists within the instance, another smaller instance can be found that contains a solution if and only if the original instance did. As this can be done in polynomial time, we only need to handle one such instance, or else it would take exponential time. There are five such polynomial time reductions, which we will quickly discuss in this section.

As for the variables with two possible colors, we know that either of the two possible variable-color pairs must be selected: we cannot choose variable-color pairs for other variables such that both of these become impossible. So, we can replace the variable with constraints between every combination of two variable-color pairs that would cause both possible assignments for this variable to be removed.

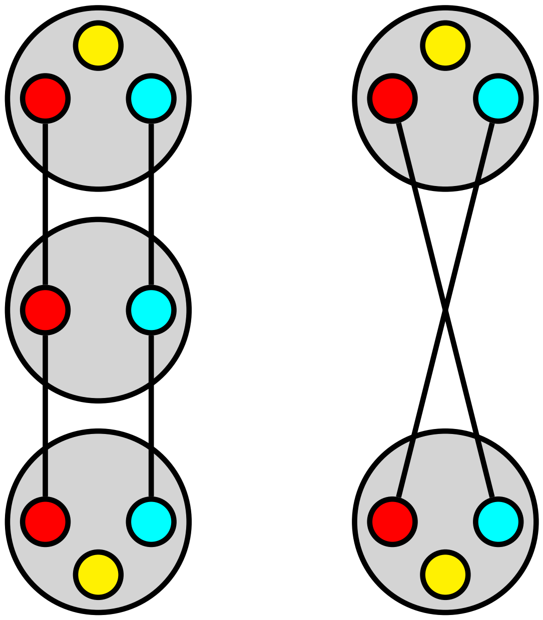


Figure 6: Removing a variable with two colors. Notice that there will not be a constraint created between two color-variable pairs within the same variable: a variable can only be assigned one color anyway.

The second polynomial time reduction is when two variables both have a variable-color pair that is only involved in constraints with variable-color pairs of the other variable, but not with each other. Then, we can assign both of the variables the color of said variable-color pair: it will never cause a violation of a constraint, as all the constraints they are involved with are with variable-color pairs that we decide not to use.

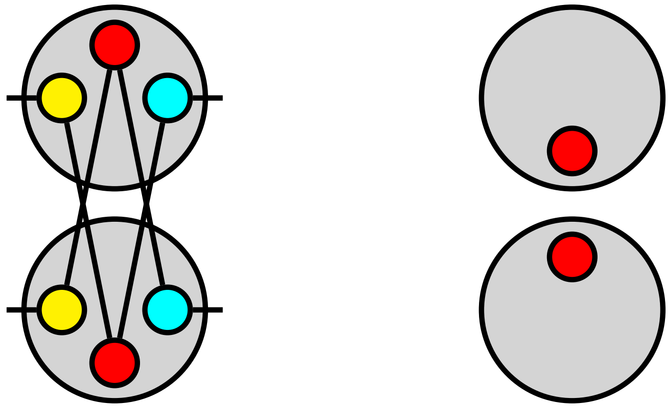


Figure 7: Assigning colors to two variables that both have a variable-color pair involved only in constraints with the other variable, but not with each other. Constraints that lack an endpoint represent that they connect to any other variable-color pair.

Consider a variable v with two possible colors R and G, such that for every constraint $((v, R), (w, C))$ for an arbitrary variable-color pair (w, C), there also exists a constraint $((v, G), (w, C))$. Then, we can remove color R as an option from variable v: when R is a valid color, G is always a valid color as well.

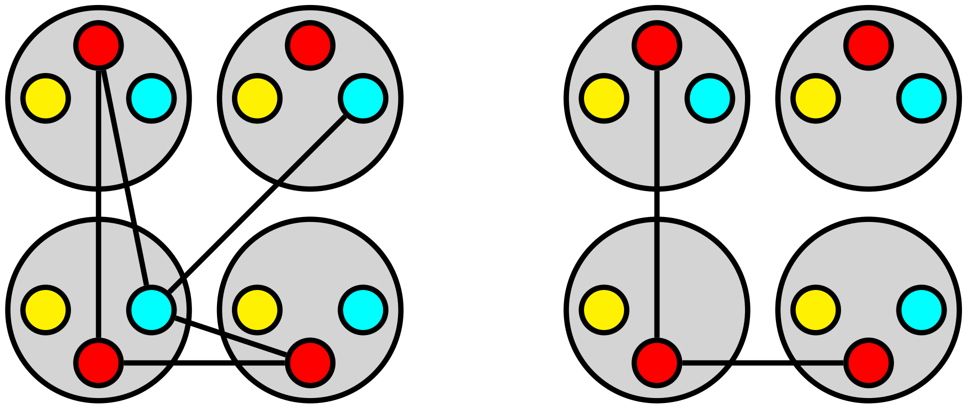


Figure 8: Removing a variable-color pair that appears in the same constraints as another variable-color pair of its variable.

Fourthly, if a variable-color pair does not appear in any constraints, we can safely assign this color to the variable. It will never violate any constraints.

Figure 9: Assigning a color to a variable, when said color-variable pair is not involved in any constraints. Constraints that lack an endpoint represent that they connect to any other variable-color pair.

Consider a variable-color pair (v, C) that is connected to every single variable-color pair of some other variable. Then, we can immediately remove C as a possible color for v: choosing to use variable-color pair (v, C) would always cause the other variable to be impossible to color.

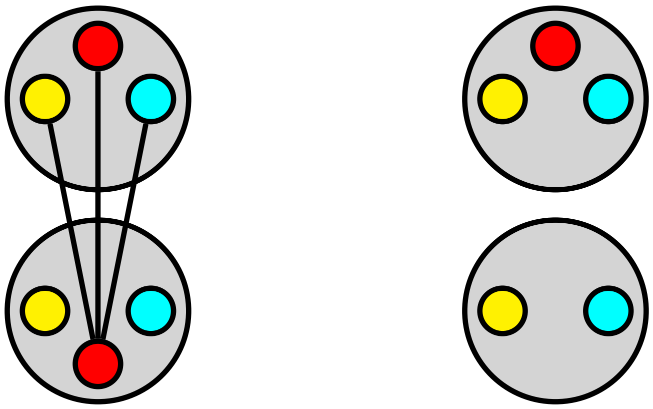


Figure 10: Removing a variable-color pair that connects to every possible variable-color pair for some other variable.

5.2 Exponential time reductions

We have seen how to use polynomial time reductions to find smaller instances for the (4,2)-CSP. However, we will need to use exponential time reductions to solve most instances: these create multiple smaller instances, which will all need to be solved to determine that no 3-coloring exists. Let a *clique of variable-color pairs* be a set of variable-color pairs, such that every two variable-color pairs in the clique appear in a constraint together. We know that (4,2)-CSP is solvable in polynomial time if every variable-color pair appears in a clique of variable-color pairs of size three or four, where no constraints exist outside these cliques. As such, we aim to find such a graph by eliminating all other configurations with exponential time reductions.

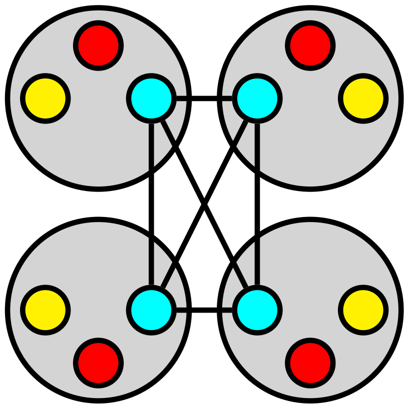


Figure 11: A clique of four variable-color pairs (the cyan variable-colors pairs).

We will express the cost it takes to reduce the instance size using the work factor, which we explained in Section 2. Remember that the weight of a variable with three possible colors is one in the instance size, whereas the weight of a variable with four possible colors is $2 - \epsilon$.

Consider two variable-color pairs that both appear only appear in one constraint, a constraint with each other. If both variables have three possible colors, we can combine them into a variable with four colors, reversing the transformation from Figure 5. Otherwise, we use the knowledge that we can always use one of the variable-color pairs. As such, we can create two new instances: one for each of the two variable-color pairs. The worst case happens when one of these is a four-color variable and one is a three-color variable. In the branch where we assign the color to the three-color variable, we remove the three-color variable and one color from the four-color variable: an instance size reduction of $1 + (2 - \epsilon - 1) = 2 - \epsilon$. In the second branch, we remove the four-color variable by assigning it the color and remove the paired color from the three-color variable as well. It will now only have two remaining possible colors, which allows for it to be removed from the instance as well, causing an instance size reduction of $2 - \epsilon + 1 = 3 - \epsilon$ and a work factor of $\lambda(2 - \epsilon, 3 - \epsilon) \approx 1.3401$.

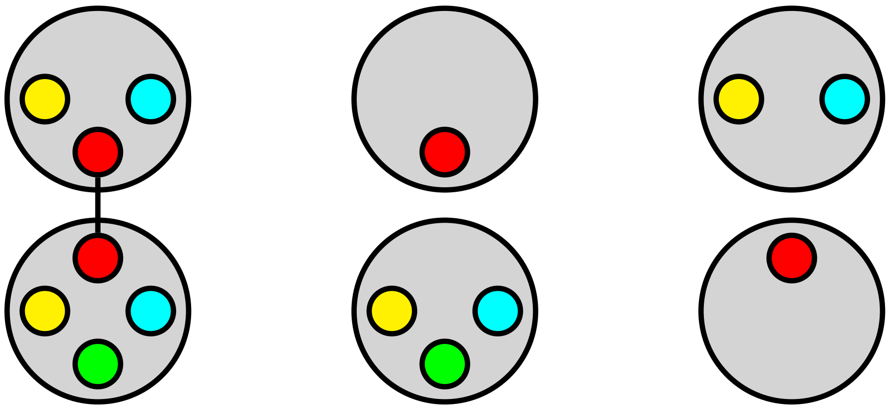


Figure 12: Branching on coloring either the upper or lower variable red. The original instance (left), with the resulting instance for each of its branches (middle and right). No constraints are shown beside the one between these two variables.

There are many exponential time reductions, and they are oftentimes more complicated than polynomial time reductions. As such, we will only summarize the remaining ones to show how they contribute to our goal of having all constraints appear in cliques of variable-color pairs of size three or four. As with the previous exponential time reduction, many of these branching rules have some branches in which we can apply another polynomial or exponential time reduction. In this case, we can include this operation in the exponential time reduction to reduce the work factor.

First, we reduce the remaining cases where some variable-color pair is involved in only a single constraint. Again, we choose to use either of the two involved variable-color pairs. In the worst case, the other variable-color pair in the constraint is involved in only one other constraint. By doing this, we can remove all cases from the instance where a variable-color pair is only involved in a single constraint.

Next, we reduce any instances of some variable-color pair being involved in constraints with multiple variable-color pairs of a different second variable. In this case, we choose to use either one of the two colors of the second variable linked with the same variable-color pair, or to use the variable-color pair involved with multiple pairs of the second variable. Both of these options remove the second variable from the instance: in both cases, it can have at most two remaining variable-color pairs. After this reduction, no variable-color pair can be involved in constraints with multiple variable-color pairs of the same variable.

Afterward, we instead reduce variable-color pairs involved in many constraints. These can be reduced easily because when they are selected, they will remove a color from many other variables. One noticeable case is when a variable-color pair involved in many constraints appears in some constraint with a variable-color pair that only appears in two. Whether we choose to use the highly constrained variable-color pair or not, this will remove one of the two constraints for the other variable. As it will only appear in one constraint afterward, we can then reduce this instance further.

Like this, we can reduce instances with a variable-color pair involved in at least four constraints and instances where one variable-color pair involved in two constraints and one involved in three constraints appear in a constraint together. As we already knew there could not be a variable-color pair involved in a single constraint, any constraint must be between two variable-color pairs involved in the same number of constraints, which must be either two or three.

Notice that if variable-color pairs are not involved in a clique of variable-color pairs, they must be part of a larger structure consisting of only variable-color pairs of the same degree. As such, we only need to remove these larger structures to find our objective instance. For variable-color pairs involved in two constraints, it must be a cycle of these variable-color pairs. As such, the reduction rules branch whether there are five consecutive variable-color pairs on the cycle over unique variables, whether a variable repeats after three variable-color pairs, or whether it loops over the same four variables. Once all of these cases have been eliminated, any variable-color pair involved in two constraints must be part of a clique of variable-color pairs.

Finally, we are left with variable-color pairs involved in three constraints. Then, the structure created by these constraints either contains five unique variables or contains eight or twelve variable-color pairs over the same four variables. In the latter case, ones over twelve variable-color pairs can never have constraints with variable-color pairs outside these four variables. As such, we can determine in $\mathcal{O}(1)$ time whether there exists a solution for these variables or not. If there are eight variable-color pairs, there exist three possible different solutions, each of which removes all four variables.

Finally, if the structure contains more than four unique variables, there must exist a configuration where one variable-color pair appears in constraints with three other variable-color pairs, at least one of which appears in a constraint with a variable-color pair of a fifth variable. Then, this configuration can be used to find the final exponential time reduction, eliminating all instances excluding those with only cliques of variable-color pairs of size two or three.

5.3 Solving the remaining graph

Every remaining variable in the graph will contain either three or four colors, with every variable-color pair appearing in a clique of three or four variable-color pairs. As such, choosing to assign a color to some variable will cause none of the other variable-color pairs in the clique of variable-color pairs to be possible. As the other variable-color pairs in the clique will not be involved in constraints with any variable-color pairs outside this clique, assigning a color to a variable will not have any impact outside its clique of variable-color pairs.

So, for every clique of variable-color pairs, we can select at most one variable-color pair for our solution. Additionally, we want to select a single variable-color pair for every variable. Then, we want to match variables to cliques of variable-color pairs, such that every clique is assigned a clique of variable-color pair that one of its variable-color pairs appear in.

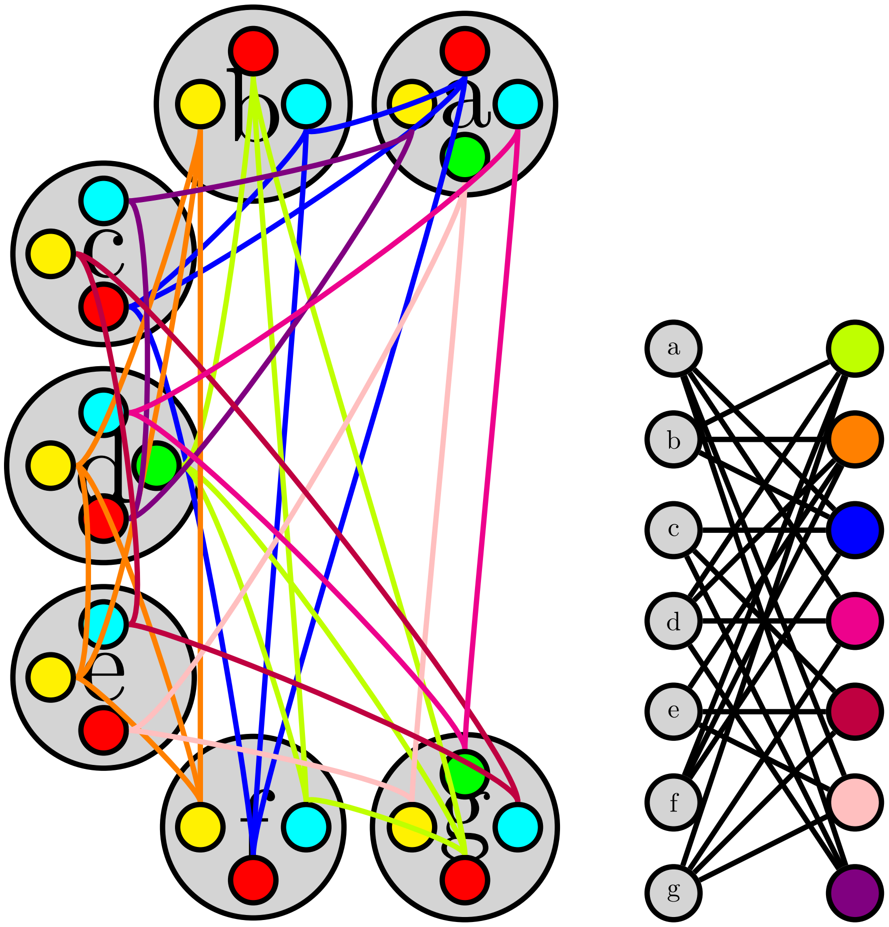


Figure 13: Transforming a (4,2)-CSP instance where all variable-color pairs are part of cliques of variable-color pairs of size three or four. The constraints corresponding to the cliques have been color coded for convenience, and the vertices representing these cliques have been given the same color in the graph used for matching.

We create a bipartite graph with the two sides representing the variables and the cliques of variable-color pairs respectively. We connect a variable with a clique of variable-color pairs when the variable has a variable-color pair that appears in this clique. Then, we find a maximum matching for this graph. If and only if every variable can be matched to a clique of variable-color pairs, there exists a solution for this instance: we color every vertex according to the clique of variable-color pairs it is matched with. It will get the color of its variable-color pair involved in this clique. If the maximum matching does not include all variables, we will not be able to color this instance, as any coloring for the unmatched variables will always cause a constraint to be violated.

There exists a linear number of cliques of variable-color pairs (compared to the instance size), so the number of vertices in the graph on which we perform a matching algorithm is also linear: the maximum matching can be found in polynomial time.

5.4 Runtime

We saw how we can solve the (4,2)-CSP, which also solves the (3,2)-CSP. As the transformation from 3-COLORING to (4,2)-CSP creates $|W|$ variables with three possible colors and none with four possible colors, the original instance size will be $|W|$. After optimizing ϵ, the largest work factors are $\lambda(4,4,5,5) = \lambda(3-\epsilon, 4-\epsilon, 4-\epsilon) = \lambda(1+\epsilon, 4) \approx 1.36443$. So, we can solve (3,2)-CSP in time $\mathcal{O}^*(1.36443^n)$.

6 4-Coloring

The best-known algorithm for 4-COLORING was created by Fomin, Gaspers, and Saurabh [13]. They combined a pathwidth approach with the enumeration of independent sets such that for any graph, one of the approaches will be relatively fast. We must ensure that algorithms exist in both of these manners for any problem we want to solve using this approach. We will describe these algorithms for 4-COLORING in Sections 6.1 and 6.2, respectively. Afterward, we will describe in Section 6.3 how we combine the two concepts to bound the runtime of the algorithm. Finally, this algorithm uses the 3-COLORING algorithm as a black box. Since we improve upon 3-COLORING in Part II, we will also quickly explain how this improves 4-COLORING.

6.1 Path decomposition algorithm

A *path decomposition* of a graph $G = (V, E)$ is a sequence of sets $S_1, S_2, ..., S_n$ of vertices such that the following properties hold [29]:

1. For every edge $(v, w) \in E$, there exists a set that contains both v and w.
2. For every three sets S_i, S_j, S_k such that $i \leq j \leq k$, all vertices in both S_i and S_k are also elements of S_j.

The *pathwidth* of a path decomposition is the number of elements in the largest set in the path decomposition.

Notice that, for some vertex v, S_i is the first set v is an element of and S_k is the final set v is an element of, v must also be an element of every set S_j where $i \leq j \leq k$. We say that vertex v is added to the path decomposition at set S_i and removed from the path decomposition at set S_{k+1}. Indeed, vertices can only be added from the path decomposition once and removed once, as well.

We can transform any path decomposition into a *nice path decomposition*: a path decomposition where every set either adds a single vertex or removes a single vertex. Given a path decomposition, any set that adds or removes multiple vertices can be split into multiple sets that either add or remove a single vertex. So, any nice path decomposition has $2 \cdot n + 1$ sets: one that adds every vertex, one that removes every vertex, plus an empty one at the start.

Given a nice path decomposition of pathwidth l, we can solve 4-COLORING in time $\mathcal{O}(4^l)$ through dynamic programming. Every set will differ from the previous set by exactly one vertex. As such, we can consider two different cases: *Forget* (removing a vertex) and *Introduce* (adding a vertex). All possible colorings of the set will be considered, so it takes at most 4^l iterations to consider all possible configurations per set. The first set will have all its configurations set to true. It will contain no vertices, so any configuration is valid. For any $1 < i \leq |V|$, we say the coloring is valid when no two adjacent vertices have the same color and:

1. Introduce: the same coloring is valid in the previous set, excluding the newly added vertex.
2. Forget: there exists a color for the removed vertex, such that the previous set has a valid coloring.

We verify whether the final set has a valid coloring to see if there exists a possible coloring for the entire graph. At this point, all vertices have been added and removed from the graph: for every vertex, it has been verified whether there exists a color that they could be. Furthermore, every edge has had both of its endpoints in the same set, so it is also known that no two neighbors have been given the same color.

As there are a polynomial number of sets, and it takes time $\mathcal{O}^*(4^l)$ to handle one set, it also takes time $\mathcal{O}^*(4^l)$ to complete the entire algorithm and solve 4-COLORING through a pathwidth approach.

This algorithm is likely optimal. Lokshatov et al. [25] found that if k-COLORING can be solved on bounded pathwidth graphs in time $\mathcal{O}((3 - \epsilon)^l)$, then it would contradict the Exponential Time Hypothesis. An acceleration from time $\mathcal{O}(4^l)$ to time $\mathcal{O}(3^l)$ would be possible while respecting this hypothesis, but this would have to be an improvement that cannot be applied to 3-COLORING: the same approach is used to solve 3-COLORING in time $\mathcal{O}^*(3^l)$, so any improvement that can be applied to 3-COLORING would violate the Exponential Time Hypothesis.

6.2 Enumeration algorithm

The second method that is used to solve 4-COLORING is the enumeration over maximal independent sets. Given all maximal independent sets in a graph, we can iterate over them and assign one color class to the maximal independent set. Then, we color the remaining graph using the 3-COLORING algorithm: which took time $\mathcal{O}^*(1.3289^n)$ at the time of Fomin et al.'s result, but we will update this to $\mathcal{O}^*(1.3236^n)$.

As we will already have colored one of the color classes, the 3-COLORING algorithm will receive less than the entire initial graph as its input: given an independent set I, the input graph for 3-COLORING will be $G[V - I]$, so the instance size will be $|V| - |I|$.

6.3 Combining the algorithms

We now know two methods to solve 4-COLORING. In this section, we will discuss how to decide which algorithm to use depending on the properties of the input graph. The general idea of combining these two algorithms is to enumerate the independent sets by branching on high-degree vertices to reduce the maximum degree of the graph. If the graph reaches a low maximum degree, we will be able to bound the pathwidth of the graph and find a path decomposition with pathwidth within this bound.

The algorithm will keep track of three different sets of vertices: V (the vertices in the graph), I (the vertices known to be in the independent set), and C (the vertices known to not be in the independent set). It branches on the vertex of the highest degree in $G[V - (I \cup C)]$. Let this vertex be v with degree $d(v)$. Then, we can either include it in I or add it to C. Notice that if v is added to I, all of its $d(v)$ neighbors will be added to C. This results in the following recurrence, where n denotes the number of vertices in $G[V - (I \cup C)]$.

$$T(n) \leq T(n - d(v) - 1) + T(n - 1)$$

We continue with this algorithm until $d(v) < a$. For 4-COLORING, we set $a = 5$. Depending on the number of vertices in C, the algorithm decides whether to continue branching on the highest degree vertex or to use the pathwidth algorithm. For every $2 \leq i < a$, we fix a number α_i between zero and one, such that the pathwidth approach is chosen when $|C| \leq \alpha_i \cdot |V|$. When $d(v) = 2$, it will iterate over the remaining maximal independent sets. Otherwise, it repeats the branching rule and reconsiders which algorithm to use again afterward. Furthermore, we will consider α_a to be zero when used in any calculations: the algorithm will never use the pathwidth algorithm when the maximum degree is a or larger.

For the enumeration algorithm, we will first analyze how long it takes to solve 4-COLORING given sets G, I, and C: we enumerate over all independent sets of at least size $|V|/4$ and check whether the remainder is 3-colorable. We know that there exist at most $3^{4 \cdot l - |V|} \cdot 4^{|V| - 3 \cdot l}$ maximal independent sets of size l [12]. As we know there will always exist a color class containing at least $|V|/4$ vertices, we iterate over all possible independent sets $\lceil n/4 - |I| \rceil$ to $|V| - |I| - |C|$ in the remaining graph, add them to I and verify whether the remaining vertices are 3-colorable: this takes time $\sum_{l=\lceil n/4-|I|\rceil}^{|V|-|I|-|C|} 3^{4l-|V|+|I|+|C|} \cdot 4^{|V|-|I|-|C|-3l} \cdot 1.3289^{|V|-|I|-l}$ (where we use Beigel and Eppstein's [2] 3-COLORING algorithm, we update this reasoning in Section 6.4). Knowing this, we express the runtime separately in terms for each set of vertices $t_n^{|V|}$, $t_i^{|I|}$, and $t_c^{|C|}$, where $t_n \cdot t_i \cdot t_c$ is the time it takes to calculate whether there exists a solution for sets V, I, and C. For 4-COLORING, we find $t_n = 4^{1/4} \cdot 1.3289^{3/4}$, $t_i = 16/27$, and $t_c = 3/4$.

Fomin et al. [14] showed that for any ε, there exists an integer n_ε, such that for any graph with more than n_ε vertices, the following equality bounding the pathwidth of a graph holds:

$$pw(G) \leq \frac{1}{6}n_3 + \frac{1}{3}n_4 + \frac{13}{30}n_5 + \frac{23}{45}n_6 + n_{\geq 7} + \varepsilon n$$

When the maximum degree in the graph is lower than seven, we can use this equation to determine that the graph has limited pathwidth. Let β_d be the pathwidth of a graph with maximum degree d compared to the total number of vertices. We bound this by using the equation above and state that $\beta_3 = \frac{1}{6}, \beta_4 = \frac{1}{3}, \beta_5 = \frac{13}{30}, \beta_6 = \frac{23}{45}$. Then, when the maximum degree of a graph is d, a path decomposition of width $\beta_i \cdot |(|V| - (|I| + |C|)) + |C|$ can be found by using the path decomposition of $G[V - (I \cup C)]$ and adding C to every set.

We will now analyze the runtime of the algorithm. To do this, we will bound the runtime by analyzing the circumstances under which the algorithm will be executed. The pathwidth will only be executed when the graph is known to have a limited pathwidth: below $\beta_d + (1 - \beta_d) \cdot \alpha_d$. So, the pathwidth algorithm can either take time:

1. $\mathcal{O}^*(4^{\alpha_2 \cdot n})$
2. $\mathcal{O}^*(4^{(1/6+5\cdot\alpha_3/6)} \cdot n)$
3. $\mathcal{O}^*(4^{(1/3+5\cdot\alpha_4/3)} \cdot n)$
4. $\mathcal{O}^*(4^{(13/30+17\cdot\alpha_4/30)} \cdot n)$
5. $\mathcal{O}^*(4^{(23/45+22\cdot\alpha_4/45)} \cdot n)$

Where we can ignore any algorithm depending on an α_d for $d \geq a$, as we will only ever use the pathwidth algorithms when $d < a$.

The runtime of the enumeration algorithm depends on the degrees of the vertices it branches on: a higher degree vertex v has more neighbors, which will never be in the independent set if v is added to the independent set. Consider when the algorithm branches on a vertex of degree $d \in \{2, 3, \ldots, a-1, \geq a\}$. It takes more time to branch on vertices of a lower degree. So, in the worst case, the highest degree of the graph decreases to d the moment there are more than $\alpha_d \cdot n$ vertices in $|C|$: then $|C|$ increases by at most $\alpha_{d-1} - \alpha_d$ by branching on vertices of degree d.

As $|C|$ will increase by d when a vertex is added to the independent set and by one when it is not, we can use this upper bound on how much $|C|$ increases to limit the time it takes to branch on vertices of degree d. The details of the calculation are too complicated to mention here, but are given in Fomin et al.ś paper [13]. However, they conclude that it takes time $\mathcal{O}^*(t_d^{(\alpha_{d-1}-\alpha_d)\cdot n})$ to branch on vertices of degree d, where $t_d = 1 + r_d$ and r_d the minimum positive root of $(1+r)^{-(d-1)} \cdot r^{-1} \cdot t_i - 1$. Remember that t_i is the multiplicative cost per vertex in the independent set, which they contribute to the overall runtime when solving 4-coloring given sets G, I, and C. Finally, we can multiply the time it takes to branch on each vertex degree with the time it takes to solve the remaining problem when $d = 2$ and $|C| > \alpha_2 \cdot n$: $\mathcal{O}^*(t_n^{|V|} \cdot t_c^{\alpha_2 \cdot n} \cdot \prod_{d=3}^{a} t_d^{(\alpha_{d-1}-\alpha_d)\cdot n})$. Notice that the term t_i does not appear in this equation directly, as we do not have a lower bound on the number of vertices in I at the time we solve the problem directly.

For 4-coloring, we use the following parameters to minimize the runtime of both the pathwidth and the enumeration algorithms: $t_n = 4^{\frac{1}{4}} \cdot 1.3289^{\frac{3}{4}}, t_i = \frac{16}{27}, t_c = \frac{3}{4}, a = 5, \alpha_2 = 0.39418, \alpha_3 = 0.27302, \alpha_4 = 0.09127, t_3 = \frac{4}{3}, t_4 \approx 1.28154, t_5 \approx 1.24592$. So, 4-coloring can be solved in time $\mathcal{O}(t_n^{|V|} \cdot t_c^{\alpha_2 \cdot |V|} \cdot \prod_{d=3}^{a} t_d^{(\alpha_d - \alpha_{d-1})\cdot|V|} + \sum_{d=2}^{a-1} 4^{(\beta_d + (1-\beta_d)\cdot\alpha_d)\cdot|V|}) \approx \mathcal{O}(1.7272^{|V|})$.

6.4 Updating 4-coloring

For reproducibility, we give the parameters used to update 4-coloring from time $\mathcal{O}^*(1.7272^n)$ to time $\mathcal{O}^*(1.7247^n)$ using the improvement of 3-coloring as discussed in Part II: 3-coloring can now be solved in time $\mathcal{O}^*(1.3236^n)$ instead of time $\mathcal{O}^*(1.3289^n)$. We use $a = 5$, which means that we get parameters α_2, α_3, and α_4.

t_n	$4^{1/4} \cdot 1.3236^{3/4}$	t_3	$4/3$	α_2	0.392148
t_i	$16/27$	t_4	1.28154	α_3	0.270578
t_c	$3/4$	t_5	1.24592	α_4	0.0882225

Table 1: Parameters used to update 4-coloring with the improved runtime for 3-coloring.

Part II
3-coloring in Time $\mathcal{O}^*(1.3236^n)$

We will discuss our major contribution in this part: improving Beigel and Eppstein's [2] algorithm for 3-coloring to improve the time from $\mathcal{O}^*(1.3289^n)$ to $\mathcal{O}^*(1.3236^n)$. As k-coloring is solvable in polynomial time for $k \leq 2$, 3-coloring is the easiest version of the NP-complete k-coloring problems. In fact, Lovász [26] showed that 3-coloring itself is NP-complete. Due to this quality, it has become one of the most studied problems in graph theory, with an interesting history of improvements.

Lawler [23] presented one of the first algorithms for 3-coloring. He found that 3-coloring can be solved by iterating over all maximal independent sets. In each iteration, all vertices in the maximal independent set are given one color. Then, he verified whether the remainder of the graph is 2-colorable in polynomial time. Moon and Moser [27] found that there exist at most $\mathcal{O}^*(3^{3/n})$ maximal cliques on n vertices. As (maximal) independent sets become (maximal) cliques in the complement graph, the same bound applies to maximal independent sets. It is also possible to iterate independent sets in time $\mathcal{O}^*(|\mathcal{I}|)$, where $\mathcal{I}$ denotes the set of all maximal independent sets in a graph [19], which has size at most $\mathcal{O}^*(3^{3/n})$. So, this also bounds Lawler's 3-coloring algorithm.

Later, Schiermeyer [30] created an algorithm for 3-coloring that runs in time $\mathcal{O}^*(1.415^n)$. This algorithm finds vertices of the same color and contracts them into a single vertex. A critical case involved finding a 2-coloring for the neighbors of some vertex, for which he validated whether this 2-coloring would be possible within the rest of the graph. Eventually, the contraction of vertices would either find an impossible case or be reduced to a small size for which it would be trivial to find a 3-coloring. In the latter case, all vertices could be expanded again to determine their color.

Theorem 6.1. *There is an algorithm for* 3-coloring *running in time $\mathcal{O}^*(1.3236^n)$ on n-vertex graphs.*

The 3-coloring problem is used as a black box in the fastest algorithm for 4-coloring [13], which runs in time $\mathcal{O}^*(1.7272^n)$. Depending on the input graph, they determine whether it is optimal to use a pathwidth approach with dynamic programming or one enumerating the number of independent sets. In the latter case, they validate for each independent set whether the remainder of the graph can be colored with three colors using the optimal 3-coloring algorithm. Due to the improvements to 3-coloring we present, 4-coloring can be solved in time $\mathcal{O}^*(1.7247^n)$, as elaborated upon in Section 6.

This section will go over Beigel and Eppstein's algorithm in Section 7, where we will briefly discuss how they solved any 3-coloring instance in time $\mathcal{O}^*(1.36443^n)$, which we will use as a black box algorithm. Then, we will explain in-depth how they further improved upon this algorithm by finding a set of vertices that can be colored more quickly, as this paper will improve upon this analysis. We will alter two lemmas from Beigel and Eppstein to fit our new analysis in Section 8. Subsequently, in Section 9 we describe and find a graph structure that enables us to color a large number of vertices relatively easily. Afterward, Section 10 analyzes the structures found in Section 9 to partition the set of vertices and uses this partition to formulate constraints between the number of difficult-to-color vertices and easy-to-color vertices. Finally, Section 11 presents a new linear program which uses the analysis of Section 10 and our newfound algorithm is summarized in Section 12.

7 Summary of Beigel and Eppstein's $\mathcal{O}^*(1.3289^n)$ algorithm

The 3-coloring problem can be solved in time $\mathcal{O}^*(1.3289^n)$ time [2]. This algorithm uses the (3,2)-constraint satisfaction problem ((3,2)-CSP) as a generalization of 3-coloring. In the (3,2)-CSP, there are n vertices that can have at most 3 different colors. Here, a vertex-color pair is the combination of a vertex and one of its possible colors. There are constraints in the (3,2)-CSP consisting of two vertex-color pairs, which prevent both pairs from existing in a valid solution. As such, the idea that no two adjacent vertices may be assigned the same color can be formulated by creating a constraint for each edge and color specifying that the adjacent vertices may not both have that color. More details on solving the (3,2)-CSP are given in Section 5.

As the (3,2)-CSP problem allows any semi-colored graph as an input, Beigel and Eppstein also analyze the remainder of the graph to color some vertices more quickly than by using the (3,2)-CSP algorithm.

In this algorithm, Beigel and Eppstein used many branching rules that reduced the size of the problem over multiple branches. A branching rule finds a set of smaller graphs such that if the original instance has a valid 3-coloring, at least one of the graphs in the branches will admit a 3-coloring as well.

For this, we need to be able to analyze the runtime of these branches to bound the runtime of the overall algorithm. The instance size mentioned will refer to the number of vertices in the graph that can be colored any of the three colors, and will be denoted by the variable n. We will now reiterate the definition of the work factor from the preliminaries for clarity.

Definition 7.1 (Work factor). The work factor, $\lambda(r_1, r_2, \ldots, r_k)$, is a list of numbers representing a set of branches, where each number represents the reduction in instance size (the number of vertices) within that branch. The work factor is equivalent to a multiplicative cost per vertex: solving $c^n = \sum_{i=1}^{k} c^{n-r_i}$ for c.

An algorithm with branching rules and corresponding work factors $\lambda_1, \lambda_2, \ldots, \lambda_l$ runs in time $\mathcal{O}^*(c^n)$, if and only if every possible branching rule solve to a multiplicative cost per vertex c. So, this algorithm will take time $\mathcal{O}^*(\max_{i=1}^{l}(\lambda_i^n))$.

7.1 Solving the (3,2)-constraint satisfaction problem

Beigel and Eppstein showed that the (3,2)-CONSTRAINT SATISFACTION PROBLEM can be solved within time $\mathcal{O}^*(1.36443^n)$ through a branch and reduce approach. As the (3,2)-CSP models the vertices with a list of possible colors, it can also represent partially colored graphs, where the colors for some vertices are restricted. In this case, the problem size is the number of vertices that have not been assigned a color, regardless of the number of remaining possible colors. An important case for upcoming improvements is that if there exists some vertex with two possible colors, then there exists a smaller instance that leads to a solution if and only if the initial instance did, so these vertices can be removed in polynomial time.

Theorem 7.1 ([2, Theorem 1]). *The* (3,2)-CONSTRAINT SATISFACTION PROBLEM *can be solved with a work factor at most* $\lambda(4,4,5,5) \approx 1.36443$.

Details on how we solve the (3,2)-CSP are given in Section 5. As our algorithm for 3-COLORING does not adapt the algorithm for (3,2)-CSP and only uses it as a black box, the theorem is listed here for completeness.

7.2 Removing vertices of degree three or less

Later, we will find a small subset of vertices in the graph such that many vertices neighbor this subset. We will be able to color this subset, causing all neighbors to have at most two possible colors remaining, which will allow them to be reduced from the instance. Vertices of low degree have few neighbors, which makes it harder to find such a subset. Vertices of degree one or degree two can always be removed from the instance without changing the 3-colorability: for a vertex of degree one or two, there will exist a color that none of its neighbors are assigned.

Large connected subgraphs of degree-three vertices can also be removed relatively easily. Any cycle consisting of degree-three vertices can be removed from the instance with a worst-case work factor of $\lambda(5,6,7,8) \approx 1.2433$. The vertices in a cycle of degree-three vertices will have one neighbor outside the cycle. The various cases they presented branch on whether these neighbors have the same color or not.

Lemma 7.2 ([2, Lemma 20]). *Let G be a* 3-COLORING *instance in which some cycle consists of only degree-three vertices. Then we can replace G with smaller instances with a work factor at most* $\lambda(5,6,7,8) \approx 1.2433$.

After removing cycles of degree-three vertices, any connected subgraph of degree-three vertices will be a tree. However, we can remove any subgraph of degree-three vertices with eight or more vertices. If possible, a vertex within the subgraph with three neighbors of degree three is chosen. Otherwise, one is selected in the middle of the subgraph. Then, we know that the three neighbors of the vertex need to be colored using two colors. There will be three branches, one for each pair of neighbors that get the same color. Afterward, in some branches, there will be vertices of degree two that can be removed as well. This leads to a work factor of $\lambda(2,5,6)$.

Lemma 7.3 ([2, Lemma 21]). *Let G be a* 3-COLORING *instance containing a connected subgraph of eight or more degree-three vertices. Then we can replace G with smaller instances with a work factor at most $\lambda(2,5,6) \approx 1.3247$.*

7.3 Defining easy-to-color vertices

Recall that the (3,2)-CSP algorithm has a mechanism to remove vertices that can only be assigned two colors trivially. As such, it can be worthwhile to branch on the color of a vertex with a high degree. Any of its neighbors will only have two remaining possible colors, which allows us to remove them from the instance. Due to this, if we iterate over all possible colors of some vertex, then the neighbors can be removed from the instance in polynomial time. So, we can try to find a set of vertices with many neighbors, such that we can also easily color a lot of vertices.

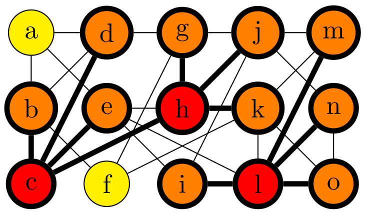


Figure 14: A maximal bushy forest in a graph. Red vertices are internal nodes, orange vertices are leaves and yellow vertices are not covered by the bushy forest. The bold edges indicate the edges included in the forest, showing that the forest does not need to be induced.

Let a *bushy forest* be a forest where every internal vertex has degree four or more in the forest. A bushy forest is *maximal* when no vertex can be added to obtain another bushy forest. Or equivalently, there does not exist any vertex outside the bushy forest with four vertices outside the bushy forest, nor any leaf of the bushy forest with three neighbors outside the bushy forest. The internal nodes will be colored by brute force, which will allow us to remove the leaves from the instance in polynomial time.

We can efficiently color a bushy forest as follows: in each tree, we arbitrarily select one vertex to be the root of the tree. Let the set of the roots of the trees be R. When coloring the bushy forest, we branch on all possible colors of the root nodes. So, coloring all root nodes takes $3^{|R|}$ time.

Intuitively, we saw in Lemma 7.3 that there cannot exist any large subgraphs consisting of only degree-three vertices. Then, there must exist many vertices of degree four, which will become the internal vertices of our bushy forest.

Let I be the set of all internal nodes of the bushy forest, excluding the roots. After coloring the vertices in R, we color the vertices in I adjacent to R with one of the two remaining colors. Subsequently, we keep coloring vertices in I adjacent to at least one colored vertex, until all vertices in I are colored. This means that we can color the vertices in I by iterating over the two possible colors for each vertex. This means that these vertices can be colored in $2^{|I|}$ time.

Let L be the set of leaves of the bushy forest. As any leaf will have two remaining possible colors, they will be removed from the instance by the (3,2)-CSP algorithm. As such, they do not contribute to the eventual runtime of the algorithm.

The remainder of the graph, any vertex not covered by the bushy forest, will be partitioned into two different sets. Let N be the set containing all the neighbors of the bushy forest and U be the set of all other vertices. Notice that any vertex in U must have degree three. All its neighbors must not be in the bushy forest, so if it had a degree of four or larger, it could be added to the bushy forest.

Beigel and Eppstein next showed that any vertex in U can be covered in a forest of trees with three children and at most five grandchildren. This forest is created by finding a large forest of $K_{1,3}$ trees in the graph: trees with a root node and three children. These $K_{1,3}$ trees are augmented by a flow algorithm that assigns all remaining vertices in U as a grandchild to one of these trees, such that no tree has over five grandchildren.

Lemma 7.4 ([2, Lemma 24])**.** *Let T be a tree with three children and at most five grandchildren. Then T can be colored with multiplicative cost per degree-three vertex at most 1.3366 in the runtime.*

The degree-three vertices in these trees can be colored in time $1.3366^{|U|}$ by brute-force coloring some vertices in the trees. The vertices in N remain and are colored by the (3,2)-CSP algorithm, so it takes $1.36443^{|N|}$ time to color those.

7.4 Finding easy-to-color vertices.

Any maximal bushy forest will always cover a constant fraction of the graph. We will establish some rules on how many vertices the aforementioned sets can contain, depending on the number of vertices in other sets.

When creating a bushy forest, any newly added tree must consist of an internal node and at least four leaves. Furthermore, if any of those leaves are turned into a new internal node, then it must have had at least three neighbors outside the bushy forest. One leaf is changed into a new internal node, but at least three leaves are added, so there is a surplus of at least two leaves:

$$4 \cdot |R| + 2 \cdot |I| \leq |L|. \tag{1}$$

Any vertex in N must have a neighbor in the maximal bushy forest. Of all vertices in the bushy forest, only its leaves can have neighbors outside the maximal bushy forest. Additionally, the leaves may only have two neighbors outside the forest. Otherwise, we could increase the size of the forest by turning the leaf into an internal node. So, there can only be two vertices adjacent to the bushy forest for each leaf in the bushy forest:

$$|N| \leq 2 \cdot |L|. \tag{2}$$

Finally, any vertex in U must be a degree-three vertex, as it is not connected to the bushy forest. Any connected subgraph consisting only of vertices in U may have at most seven vertices, due to Lemma 7.4. As there cannot be any cycles in the subgraph, it must have exactly two more edges connecting it to vertices in N than the number of vertices in the connected subgraph. In the worst case, there are seven vertices in the subgraph and nine edges go to N, so amortized, there would be at most $\frac{9}{7}$ edges per vertex in U. Vertices in N can have at most three neighbors outside the bushy forest:

$$\frac{9}{7} \cdot |U| \leq 3 \cdot |N|. \tag{3}$$

Beigel and Eppstein used these to formulate a linear program. As R, I, L, N, and U form a partition of all vertices, the runtime of the algorithm can be calculated by multiplying the time it takes to color all the sets of vertices. However, this would not be a linear formula, so the logarithm is maximized instead: whenever the original function increases, its logarithmic will too, so the same optimum will be found.

$$\log(3^{|R|} \cdot 2^{|I|} \cdot 1.36443^{|N|} \cdot 1.3366^{|U|}) = \log(3) \cdot |R| + \log(2) \cdot |I| + \log(1.36443) \cdot |N| + \log(1.3366) \cdot |U|. \tag{4}$$

Their linear program found a worst-case graph with as many vertices in $|I|$, $|N|$, and $|U|$ as possible, leading to a runtime of $\mathcal{O}^*(1.3289^n)$. Before the improvement we presented, this was the fastest known way to solve 3-COLORING. We will refine the bushy forest to be able to further partition the sets of vertices and refine these rules to improve the linear program.

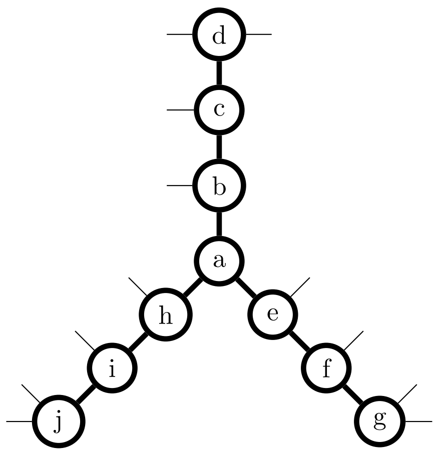
```
1  If the input graph $G$ contains any vertex $v$ with degree less than two, recursively
   color $G[V - v]$ and assign $v$ a color different from its neighbors.
2  If the input graph contains a cycle or large tree of degree-three vertices, split the
   problem into smaller instances according to Lemmas 7.2 and 7.3, recursively attempt
   to color each smaller instance, and return the first successful coloring found by these
   recursive calls.
3  Find a maximal bushy forest $F'$ in $G$.
4  Find a maximal set $T$ of $K_{1,3}$ subgraphs in $G[V - V(F)]$.
5  While it is possible to increase the size of $T$ by removing one $K_{1,3}$ subgraph and
   using the vertices in $G[V - V(F \cup T)]$ to form two more $K_{1,3}$ subgraphs, do so.
6  Use the network flow algorithm of Lemma  [2, Lemma 23] to assign the vertices of
   $G[V - V(F \cup N(F) \cup T)]$ to trees in $T$, forming a forest $H$ of height-two trees.
7  Recursively search through all consistent combinations of colors for the bushy forest
   roots and internal nodes, and for selected vertices in $H$.
   For each coloring of these vertices, form a (3,2)-CSP instance describing the possible
   colorings of the uncolored vertices and use the (3,2)-CSP algorithm to attempt to
   solve this instance. If one of the (3,2)-CSP instances is solvable, return the resulting
   coloring. If no (3,2)-CSP instance is solvable, return a flag value indicating that no
   coloring exists.
```

Table 2: Overview of Beigel and Eppstein's algorithm [2, Table 2].

8 Redefining subgraphs of degree-three vertices

We will improve upon Beigel and Eppstein's algorithm by optimizing the bushy forest to have few adjacent vertices with a high degree outside the bushy forest: this will restrict the number of vertices that can exist outside the bushy forest. However, the algorithm presented in this paper runs in time $\mathcal{O}^*(1.3236^n)$. The original reduction from Lemma 7.3 presented to remove large subgraphs of eight or more degree-three vertices runs in time $\mathcal{O}^*(1.3247^n)$, and will be modified to remove connected components of degree-three vertices of nine or more vertices instead to run in time $\mathcal{O}^*(1.3236^n)$.

This modification will also change the analysis of the time it takes to color a forest of trees with three children and at most five grandchildren: Beigel and Eppstein used the idea that a connected subgraph of degree-three vertices could consist of at most seven vertices. We will adapt this to accept connected subgraphs of at most eight degree-three vertices. Furthermore, we will prove later that some vertices of degree four or more will always be covered by this vertex and express the runtime in multiplicative cost per vertex, instead of improving only the multiplicative cost per degree-three vertex (where all other vertices assumed a multiplicative cost per vertex of 1.36443).



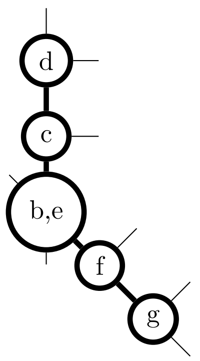


Figure 15: A large degree-three subgraph and a branch after replacing it with smaller instances.

Lemma 8.1. *Let G be a* 3-COLORING *instance containing a connected subgraph of nine or more degree-three vertices. Then we can replace G with smaller instances with a work factor at most* $\lambda(2, 6, 6)$.

Proof. We select a vertex in the connected subgraph with three neighbors of degree three if possible. Otherwise, the subgraph must be a path of degree-three vertices, in which case we select the vertex in the middle of the path. Let the selected vertex be v.

Regardless of the color of the chosen vertex, it will remove one color as an option from its neighbors. So, the three neighbors need to be colored using two colors, meaning at least two will be assigned the same color. We branch on all three ways to pick two out of the three vertices and merge the two vertices that will be assigned the same color. This reduces the instance size in any branch by two, as the chosen vertex is removed and two other vertices are merged into one.

Let the neighbor not chosen to be merged be w. This vertex will now have at most degree two and can be removed from the instance. However, if there were any other degree-three vertices adjacent to w these can be removed after w is removed. As such, any vertices in the connected subgraph of degree-three vertices that are connected to v through w will be able to be removed.

Every vertex in the subgraph of degree-three vertices will be connected to v through one of its neighbors, so every vertex besides v will be removed in one of these branches in this manner. As such, the work factor needs to be $\lambda(2 + \alpha_1, 2 + \alpha_2, 2 + \alpha_3)$ with $\alpha_1 + \alpha_2 + \alpha_3 = |V_3| - 1$, where $|V_3|$ is the number of vertices in the connected subgraph of degree-three vertices. Notice that if the connected subgraph of degree-three vertices was a path, then one α will be zero and the other two will differ by at most one. The worst case happens when there are nine vertices in the subgraph and the selected vertex has two neighbors of degree three, which forces it to be in the middle of the subgraph. This leads to a work factor of $\lambda(2, 6, 6) \approx 1.3022$. $\square$

This reduces the worst-case runtime of the algorithm when there are many connected subgraphs of degree-three vertices from $\mathcal{O}^*(1.3247^n)$ to $\mathcal{O}^*(1.3022^n)$. However, Lemma 7.4 relies on the idea that a connected subgraph of degree-three vertices can contain at most seven vertices. We will redefine this lemma to avoid the reliance on small subgraphs of degree-three vertices and also involve vertices of a higher degree in the analysis.

Lemma 8.2. *Let T be a tree with three children and at most five grandchildren. Then T can be colored with multiplicative cost per vertex at most* 1.3400.

Proof. Presume T has at most four grandchildren. In this case, we can give the root one of the three possible colors. This removes one color as a possibility for its three children, allowing those to be solved trivially by the (3,2)-CSP algorithm. The grandchildren remain and will have to be solved by the $\mathcal{O}^*(1.36443^n)$ algorithm. This algorithm becomes worse with more grandchildren, so the worst case happens when there are four grandchildren. Then, the time it takes per vertex of the tree is $(3 \cdot 1.36443^4)^{\frac{1}{8}} \approx 1.3400$.

Now, let T be a tree with exactly five grandchildren. Beigel and Eppstein showed that each child will have at most two children, so there have to be two children with two grandchildren. We iterate over all colors these two children may have. If they receive the same color, the root and their combined four grandchildren will have two remaining possible colors, leaving only the other child and grandchild for the (3,2)-CSP algorithm. If they receive different colors, the color of the root is implied to be the third color, resulting in only the fifth grandchild being left for the (3,2)-CSP algorithm. This results in a runtime per vertex of $(3 \cdot 1.36443^2 + 6 \cdot 1.36443)^{\frac{1}{9}} \approx 1.3383$.

Now we see that the worst case is a tree with four grandchildren, which leads to the claim of the lemma that the worst case is a multiplicative cost per vertex of 1.3400. $\square$

9 Finding an optimal bushy forest

Remember that a maximal bushy forest is a forest consisting of trees where each internal node has at least four neighbors in the bushy forest, and no vertex can be added to increase the size of the bushy forest.

Definition 9.1 (High-Magnitude Vertex)**.** Let a *high-magnitude vertex* be a vertex adjacent to the bushy forest, with three neighbors also outside the bushy forest.

These high-magnitude vertices allow for many vertices to exist outside the bushy forest, as they may have a lot of neighbors outside the bushy forest.

Definition 9.2 (Low-Magnitude Maximal Bushy Forest)**.** Let a *low-magnitude maximal bushy forest* be a bushy forest be any maximal bushy forest in a graph such that every high-magnitude vertex is adjacent to a tree with one internal node, four leaves, and such that every two high-magnitude vertices adjacent to the same tree must be adjacent to the same leaf of said tree, or another vertex outside the bushy forest.

In this section, we will transform a maximal bushy forest into a low-magnitude maximal bushy forest. In Section 10, we will analyze how we can improve the analysis of the algorithm by using a low-magnitude maximal bushy forest to color any graph.

Lemma 9.1. *Let G be a graph in which all vertex degrees are three or more, in which there is no cycle of degree-three vertices nor any connected subgraph of nine or more degree-three vertices. Then there exists a low-magnitude maximal bushy forest.*

Proof. First, we find any maximal bushy forest F. Then, we will increase the number of trees by showing that if specific substructures exist in the maximal bushy forest, there will exist a maximal bushy forest with more trees. As the number of trees in a graph cannot be infinite, there will be a point at which these structures cannot exist.

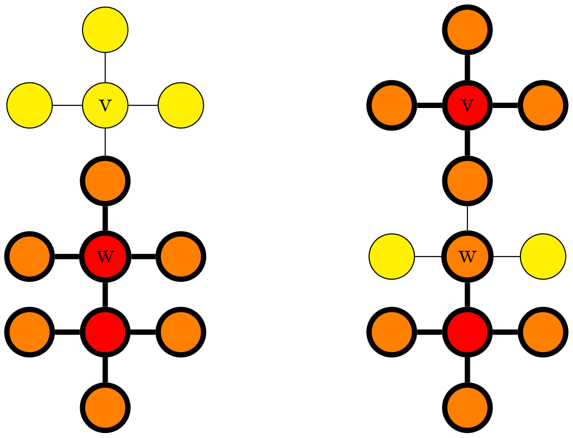


Figure 16: Removing high-magnitude vertices next to large trees. Red vertices are internal nodes, orange vertices are leaves and yellow vertices are not covered by the bushy forest. The bold edges indicate the edges included in the forest.

Let T be a tree in F with multiple internal vertices and an adjacent high-magnitude vertex v, like the graph in Figure 16 and let the internal node of T with a leaf adjacent to v be w. Remove w and the leaves adjacent to it from T and create a new tree rooted at v, which now has at least four neighbors outside F. The removal of w can cause T to be split into any number of trees, which may not be valid according to the definition of the bushy forest. However, w can become a leaf to one of the trees, with any others that do not correspond with the definition being removed as well. So, at least one tree of these trees remains valid. While any tree with multiple internal vertices and an adjacent high-magnitude vertex exists, we can perform this operation to increase the number of trees in the forest. So, there will exist a point at which no such tree can exist anymore.

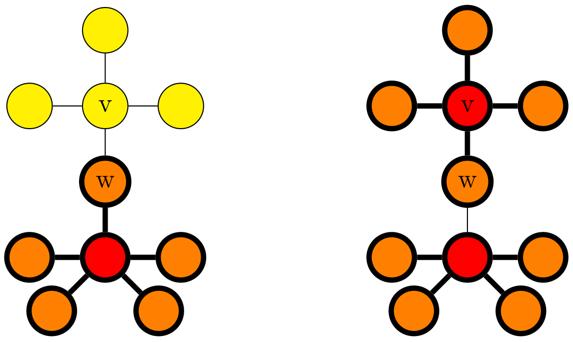


Figure 17: Removing high-magnitude vertices next to trees with many leaves. Red vertices are internal nodes, orange vertices are leaves and yellow vertices are not covered by the bushy forest. The bold edges indicate the edges included in the forest.

We can find a maximal bushy forest without any high-magnitude vertices adjacent to trees with multiple internal nodes. Next, let T be a tree in F with one internal node, more than four leaves, and an adjacent high-magnitude vertex v, like the bushy forest shown in Figure 17. Let the leaf adjacent to v be w. Remove w from T and create a new tree rooted at v, which now has four neighbors outside F. Also, T is still a valid tree, as it has one internal node and four leaves.

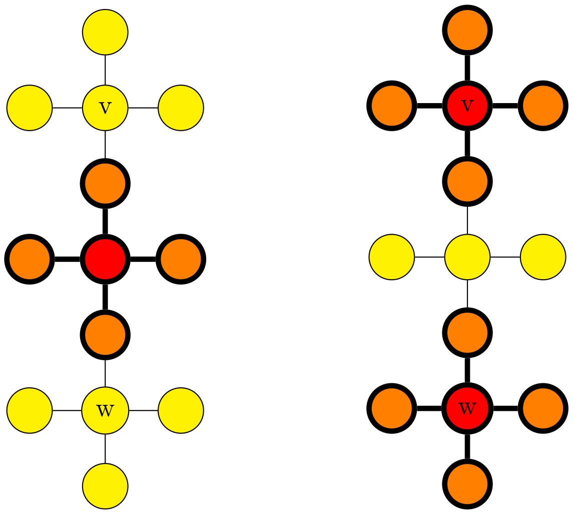


Figure 18: Removing high-magnitude vertices without shared neighbors. Red vertices are internal nodes, orange vertices are leaves and yellow vertices are not covered by the bushy forest. The bold edges indicate the edges included in the forest.

Finally, let T be a tree with two adjacent high-magnitude vertices, v and w, that do not share a leaf or a common neighbor outside F, like the one in Figure 18. Then there exists a maximal bushy forest where every two high-magnitude vertices adjacent to the same tree share a common neighbor.

Remove T from F. Now, create two new trees centered at v and w respectively. Both will have four unique neighbors outside F, so they both are valid new trees, increasing the number of trees in the forest. $\square$

Notice that any high-magnitude vertex adjacent will be adjacent to a tree with one internal node and four leaves. Each leaf may have at most two neighbors outside the bushy forest, so there can be at most eight high-magnitude vertices adjacent to any tree.

10 Partitioning vertices

Given a low-magnitude maximal bushy forest F, we know that there can exist a limited number of high-magnitude vertices. Recall that R is the set of roots of the bushy forest, I all other internal nodes, L the leaves of the bushy forest, N the neighbors of the bushy forest, and U all remaining vertices. Now, we will further partition N and U to improve the analysis of the relationship between them.

Let N_1 be the set of vertices adjacent to F that have degree three and one neighbor in F, let N_2 be the set of vertices with two neighbors in F, and let N_3 be the set of high-magnitude vertices (which have exactly one neighbor in F).

1. N_1: Vertices in N of degree three (and one neighbor in F).
2. N_2: Vertices in N with at least two neighbors in F.
3. N_3: High-magnitude vertices in N.

In this section, we will show that all of N_1, N_2, and N_3 have disadvantages that limit the number of possible vertices in U. For N_1, they further limit the number of vertices in U due to also having degree three. For N_2, they must have two neighbors in L and at most two in U, whereas the number of vertices in N_3 is limited, and they must exist in the structures described in Section 9. Finally, we will also show that if there are a lot of vertices in N_3, we can ensure that some of these vertices are covered by the trees from Lemma 8.2, meaning they can be solved more quickly.

First off, we will show that the number of vertices in N is limited based on the number of leaves L in the bushy forest.

Lemma 10.1. *Let G be a graph in which all vertex degrees are three or more, in which there is no cycle of degree-three vertices nor any connected subgraph of nine or more degree-three vertices. Let F be a low-magnitude maximal bushy forest in G. Then $|N_1| + 2 \cdot |N_2| + |N_3| \leq 2 \cdot |L|$.*

Proof. Vertices in N_1 or N_3 must have at least one edge connecting them to a vertex in L. Likewise, vertices in N_2 must have at least two edges connecting them to vertices in L. Following the definition of a maximal bushy forest, vertices in $|L|$ can have at most two edges connecting them to vertices in N. So, $|N_1| + 2 \cdot |N_2| + |N_3| \leq 2 \cdot |L|$.

$\square$

Define U' as the set of vertices in U that cannot have any neighbors in U due to all neighbors being high-magnitude vertices as described in Lemma 9.1. Next, let $G[(U - U') \cup N_1]$ be the graph induced by the vertices in U and N_1 on G, excluding those in U'. Indeed, these vertices all have degree three, so any connected component in $G[(U - U') \cup N_1]$ must consist of at most eight vertices. Let U_j be the set of vertices $U - U'$ that are part of a connected component in $G[(U - U') \cup N_1]$ with exactly j vertices in N_1: we know that $0 \leq j \leq 7$, as the component has at most eight vertices, of which at least one is the vertex in U that is assigned to a set U_j. This will partition the vertices in $U - U'$, as each of these vertices will be part of exactly one connected component in $G[(U - U') \cup N_1]$.

We will now count the edges connecting vertices in N_2 or N_3 and vertices in $(U - U') \cup N_1$. Let S be the set of edges between vertices in N_2 or N_3 and vertices in $(U - U') \cup N_1$. Let $N_{3,i}$ be the set of high-magnitude vertices such that the tree to which they are connected has i different adjacent high-magnitude vertices: $1 \leq i \leq 8$, since the vertex itself will cause it to be at least one and Lemma 9.1 shows there are at most eight high-magnitude vertices adjacent to a single tree.

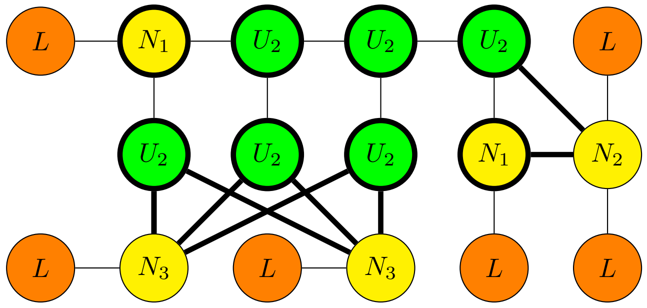


Figure 19: A graph displaying the partitions of N and U (with one connected component in $G[(U-U')\cup N_1]$): vertices in L are orange, those in N are yellow, and those in U are green. The bold vertices have degree-three and the bold edges display the edges in S (notice there are eight). The bushy forest was left out of this image, but every leaf would need to be adjacent to some internal vertex.

Lemma 10.2. *Let G be a graph in which all vertex degrees are three or more, in which there is no cycle of degree-three vertices nor any connected subgraph of nine or more degree-three vertices. Let F be a low-magnitude maximal bushy forest in G. Then,*

$$|S| \leq 2 \cdot |N_2| + 3 \cdot \sum_{i=1}^{4} |N_{3,i}| + \frac{12}{5}|N_{3,5}| + 2 \cdot |N_{3,6}| + \frac{6}{7}|N_{3,7}| \tag{5}$$

$$|U'| \geq \frac{1}{5}|N_{3,5}| + \frac{2}{6}|N_{3,6}| + \frac{5}{7}|N_{3,7}| + |N_{3,8}| \tag{6}$$

Proof. Notice that the two equations together describe the maximum number of edges between vertices in N_2 and N_3 to other vertices outside the bushy forest: N_2 has at most two neighbors outside the bushy forest, whereas N_3 has exactly three. For every vertex in U', three edges connecting vertices in N_3 to another vertex outside the bushy forest must connect it to a vertex U' instead of $(U - U') \cup N_1$. Now, we can see that Equation 5 can be rephrased as $|S| \leq 2 \cdot |N_2| + 3 \cdot |N_3| - 3 \cdot |U'|$. So, we want to prove Equation 6 to imply both.

For vertices in $N_{3,5}$, in the best case, there are two pairs of vertices in $N_{3,5}$ connected through a leaf, while one vertex does not share a leaf as a common neighbor with any vertex in $N_{3,5}$. As per Lemma 9.1, this vertex needs to have a common neighbor with all four other high-magnitude vertices adjacent to this tree. It has at most three neighbors, so one of its neighbors must neighbor two of the high-magnitude vertices: there has to be at least one vertex in U' for every five vertices in $N_{3,5}$.

Similarly, for every six vertices in $N_{3,6}$, there need to be at least two vertices in U'. For every seven vertices in $N_{3,7}$, there are at least five vertices in U'. For $N_{3,8}$ all neighbors must be in U', meaning that vertices in $N_{3,8}$ cannot have any neighbors in either $(U - U') \cup N_1$. In total, there need to be eight vertices in U' for every eight vertices in $N_{3,8}$.

To conclude, every five vertices in $N_{3,5}$ forces one vertex to exist in U', every six vertices in $U_{3,6}$ forces two vertices in U', every seven vertices in $N_{3,7}$ force five vertices in U', and every eight vertices in $U = N_{3,8}$ force eight vertices in U'. This shows that $|U'| = \frac{1}{5} \cdot |N_{3,5}| + \frac{2}{6} \cdot |N_{3,6}| + \frac{5}{7} \cdot |N_{3,7}| + |N_{3,8}|$. □

Now, we can bound the maximum number of vertices in $U - U'$ that may exist per edge in S.

Lemma 10.3. *Let G be a graph in which all vertex degrees are three or more, in which there is no cycle of degree-three vertices nor any connected subgraph of nine or more degree-three vertices. Let F be a low-magnitude maximal bushy forest in G. Then,*

$$\sum_{j=0}^{7} \left(\frac{10-j}{8-j} \cdot |U_j|\right) \leq |S| \tag{7}$$

Proof. First, we will upper bound the number of vertices in U_j based on the number of edges between connected components in $G[(U-U')\cup N_1]$ and vertices in N_2 or N_3. We know that any connected component in $G[(U-U')\cup N_1]$ may have at most eight vertices through Lemma 8.1 and cannot have any cycles as per Lemma 7.2. Furthermore, all vertices in U must have degree three, so all these vertices will exist in $G[(U-U')\cup N_1]$.

Consider a connected component in $G[(U-U')\cup N_1]$ with n total vertices and j vertices in N_1 within the connected component. Then, there will be $n-j$ vertices in U in this component. Notice that the sum of degrees of the vertices in the component (in G) is $3\cdot n$. There will be exactly $n-1$ edges within the component, each of which causes two vertices to be each other's neighbors. So, the component will have $3\cdot n-2\cdot(n-1)=n+2$ neighbors. The ratio of neighbors outside the vertex to vertices inside the component is maximized when n is maximized: eight. Each vertex in N_1 will have one neighbor in L and two outside F: otherwise, we are allowed to count it as a vertex in N_2 instead, as it would have at least two neighbors in L and at most two outside F. As each vertex in N_1 has one neighbor in F, the component requires one less neighbor in N_2 or N_3, but the maximum number of vertices in U_j decreases by the same amount. So, per vertex in U_j, at least $\frac{10-j}{8-j}$ edges in S need to exist. □

Recall the $\mathcal{O}^*(1.3400^n)$ algorithm from Lemma 8.2 and that all vertices in U will be covered by a forest of trees with three children and at most five grandchildren. For this, they first found a maximal forest in $G[V-V(F)]$, such that no tree could be removed while adding multiple new trees. We will now prove that we can find a forest of $K_{1,3}$ trees in $G[V-V(F)]$, such that all vertices in U' must have at least one neighbor covered by the forest and two vertices if its neighbors are either in $N_{3,5}$ or $N_{3,6}$.

Lemma 10.4. *Let G be a graph in which all vertex degrees are three or more, in which there is no cycle of degree-three vertices nor any connected subgraph of nine or more degree-three vertices. Let F be a low-magnitude maximal bushy forest in G. Let F' be a maximal forest of $K_{1,3}$ trees in $G[V-V(F)]$, where we cannot remove one $K_{1,3}$ to add multiple. Then, we can find a forest of $K_{1,3}$ trees where at least $\frac{2}{5}$ of the vertices in $N_{3,5}$ and $\frac{2}{3}$ of the vertices in $N_{3,6}$ are covered by F'.*

Proof. Consider a vertex v in U' adjacent to a tree with five or six high-magnitude vertices. If v has no neighbors covered by F', then v cannot be included either. This contradicts that F' is maximal, as we could add a new tree rooted at v: at least one neighbor of v will be included in F'.

Now, consider that v has one neighbor in F'. Let this neighbor be part of tree T'. We can remove T' and place a new tree at v including all three of its neighbors. T' contained at most three vertices in $N_{3,5}$ or $N_{3,6}$, as either the root of the tree was in $N_{3,5}$ or $N_{3,6}$ and one of its leaves was not, or the root of the tree was not in $N_{3,5}$ or $N_{3,6}$. So, this operation never decreases the number of vertices in $N_{3,5}$ or $N_{3,6}$ covered by forest F'.

As vertices in $N_{3,5}$ and $N_{3,6}$ border one vertex in U', the roots of the trees removed by this operation must not be in U'. The new tree will have its root in U', so this operation will never remove any trees that were created by this operation.

Whenever this operation creates a forest where one $K_{1,3}$ can be removed to add two new ones, do so. As there cannot be more trees than vertices in the graph, this will happen a polynomial amount of times.

There are at most a linear number of vertices in U'. Every step requires moving a $K_{1,3}$ tree to be rooted at a vertex in U' and the operation will never remove a tree rooted at a vertex in $K_{1,3}$. So, the number of iterations of this algorithm is bounded by the number of vertices in U' and the maximum number of $K_{1,3}$ trees in the graph (as this can remove at most one $K_{1,3}$ tree rooted at a vertex in U'): this algorithm will terminate in polynomial time.

We know that any vertex in U' adjacent to vertices in $N_{3,5}$ or $N_{3,6}$ must have at least two neighbors covered by the $K_{1,3}$ trees. For every five vertices in $N_{3,5}$, three will be adjacent to a vertex in U', of which at least two will be covered by F'. Similarly, for every six vertices in $N_{3,6}$, there will be two sets of three vertices that share a common neighbor in U': at least four out of every six will be covered by F'. □

The forest F' of $K_{1,3}$ trees will be augmented with at most five grandchildren per tree, as per Beigel and Eppstein [2]. This forest can then be colored with time per vertex 1.3400 algorithm, including some vertices in $N_{3,5}$ and $N_{3,6}$, which is faster than the time per vertex 1.36443 algorithm used otherwise.

11 Formulating a Linear Program

We define a new linear program using the bounds on the partitions of the vertices. This linear program leads to the new time bound of $\mathcal{O}^*(1.3236^n)$. The linear program consists of the objective function and thirteen types of constraints.

$$\max \quad \log(3) \cdot |R| + \log(2) \cdot |I| + \log(1.36443) \cdot |N'| + \log(1.3400) \cdot |U'| \tag{8}$$

The objective function (8) calculates the worst-case graph for our algorithm. We maintain the runtime for vertices in R, I, and N from Equation (4). It takes time $\mathcal{O}^*(1.3400^{|U|})$ to color the vertices in U due to Lemma 8.2. Furthermore, we define N^* and U^* as the vertices colored by the multiplicative time per vertex 1.36443 and 1.3400 algorithms respectively, which allows us to move vertices from N to U' according to Lemma 10.4.

$$|N| - \frac{2}{5} \cdot |N_{3,5}| - \frac{2}{3} \cdot |N_{3,6}| = |N^*| \tag{9}$$

$$|U| + \frac{2}{5} \cdot |N_{3,5}| + \frac{2}{3} \cdot |N_{3,6}| = |U^*| \tag{10}$$

Constraints (9) and (10) determine the number of vertices that can be colored more easily per vertex in U' when there are a lot of high-magnitude vertices. This follows from Lemma 10.4.

$$|R| + |I| + |L| + |N| + |U| = n \tag{11}$$

Constraint (11) shows that R, I, L, N, and U partition all vertices in the graph.

$$4 \cdot |R| + 2 \cdot |I| \leq |L| \tag{12}$$

Constraint (12) follows from the definition of a bushy forest. Every internal node must have degree four in the bushy forest. So, the node creating the tree adds four leaves, while every leaf turned into an internal node adds three leaves and removes one.

$$|N_1| + 2 \cdot |N_2| + |N_3| \leq 2 \cdot |L| \tag{13}$$

Constraint (13) follows from Lemma 10.1 and represents that the vertices in N must be adjacent to either one or multiple vertices in the bushy forest.

$$\sum_{i=1}^{8} (\frac{8}{i} \cdot |N_{3,i}|) \leq 8 \cdot |R| \tag{14}$$

Constraint (14) follows from the fact that at most high-magnitude vertices can exist per tree in the bushy forest, as shown by Lemma 9.1. The number of vertices per tree follows from the definition of $N_{3,i}$.

$$\sum_{j=0}^{7} (\frac{10-j}{8-j} \cdot |U_j|) \leq |S| \tag{15}$$

$$|S| \leq 2 \cdot |s_2| + 3 \cdot \sum_{i=1}^{4} |s_{3,i}| + \frac{12}{5} \cdot |s_{3,5}| + 2 \cdot |s_{3,6}| + \frac{6}{7} \cdot |s_{3,7}| \tag{16}$$

Constraints (15) and (16) represent the number of edges between the vertices in N_2 and N_3 and the vertices in $(U - U') \cup N_1$ and how these impact the number of vertices in U_j that can exist, as determined in Lemmas 10.2 and 10.3.

$$\sum_{j=1}^{7}(\frac{j}{8-j} \cdot |U_j|) \leq |N_1| \tag{17}$$

By definition of U_j, constraint (17) enforces that in each connected component in $G[(U-U') \cup N_1]$, there must exist some number of vertices in N_1 when $j > 0$, following from the definition of U_j.

$$\sum_{k=1}^{3} |N_k| = |N| \tag{18}$$

$$\sum_{i=1}^{8} |N_{3,i}| = |N_3| \tag{19}$$

$$\sum_{j=0}^{7} |U_j| + \sum_{i=5}^{8}(\frac{i-4}{i} \cdot |N_{3,i}|) = |U| \tag{20}$$

Finally, constraints (18) through (20) follow from the partitions of the respective sets. The partitions need to equal the number of vertices in their parent set. Notable, here the vertices in U' are counted and added to U: there exists at least one for every five vertices in $N_{3,5}$, two for every six vertices in $N_{3,6}$, three for every seven vertices in $N_{3,7}$, and four for every eight vertices in $N_{3,8}$.

The linear program is the maximization of objective function (11) with respect to constraints (12) through (20)

$\lvert R\rvert$	0.0396825	$\lvert N^*\rvert$	0.1190476	$\lvert U^*\rvert$	0.6825397
$\lvert I\rvert$	0	$\lvert N_{3,1}\rvert$	0	$\lvert U_0\rvert$	0.4444444
$\lvert L\rvert$	0.1587302	$\lvert N_{3,2}\rvert$	0	$\lvert U_1\rvert$	0
$\lvert N\rvert$	0.2777778	$\lvert N_{3,3}\rvert$	0	$\lvert U_2\rvert$	0
$\lvert U\rvert$	0.5238095	$\lvert N_{3,4}\rvert$	0	$\lvert U_3\rvert$	0
$\lvert N_1\rvert$	0	$\lvert N_{3,5}\rvert$	0	$\lvert U_4\rvert$	0
$\lvert N_2\rvert$	0.0396825	$\lvert N_{3,6}\rvert$	0.2380952	$\lvert U_5\rvert$	0
$\lvert N_3\rvert$	0.2380952	$\lvert N_{3,7}\rvert$	0	$\lvert U_6\rvert$	0
S	0.5555556	$\lvert N_{3,8}\rvert$	0	$\lvert U_7\rvert$	0

Table 3: Results of the linear program.

Table 3 shows the values found by the linear program to maximize the runtime of the algorithm when divided by the total number of vertices n. Notice that most values are zero. Most neighbors of the bushy forest are in $N_{3,6}$ with the remainder being in N_2. There do not exist any vertices in N_1, so all vertices in U are either in connected subgraphs of degree-three vertices with eight vertices in U or vertices in U' due to the large number of high-magnitude vertices.

In the new worst-case graph, the maximal bushy forest consists of many trees with a singular internal node and four leaves each. There will not exist any vertices of degree three adjacent to the bushy forest. Instead, three-quarters are high-magnitude vertices, whereas the others are vertices with at most two neighbors outside the bushy forest and at least two neighbors in the bushy forest. As there are six high-magnitude vertices adjacent to each tree, there are two vertices in U' for every tree in the bushy forest. One such tree, along with the structure it enforces, is shown in Figure 20.

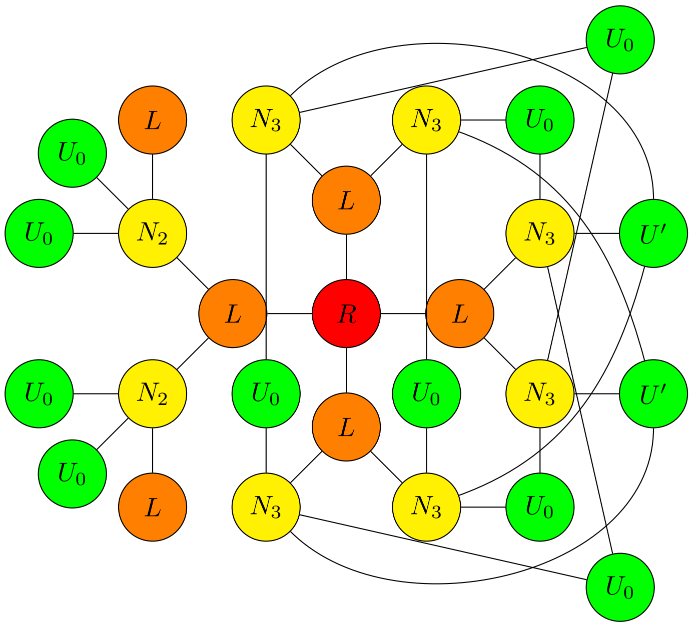


Figure 20: Worst case scenario for 3-COLORING (single tree). Red vertices are internal nodes, orange vertices are leaves, yellow vertices are adjacent to the bushy forest, and all remaining vertices are green.

The number of other vertices that are not adjacent to the bushy forest is then maximized. All of them are degree-three vertices and there do not exist any degree-three vertices adjacent to the bushy forest, so they are all U_0 vertices.

Specifically, for every vertex in R, there will be no vertices in I, and four vertices in L. There will be one vertex in N_2 and six vertices in $N_{3,6}$. This means there will be $11\frac{1}{5}$ nodes in U_0 and two in U' for every vertex in R. Finally, out of every six vertices in $N_{3,6}$, four will be solved by the more efficient $\mathcal{O}^*(1.3400^n)$ algorithm. This leads to a final runtime of $\mathcal{O}^*((3^5 \cdot 1.36443^{15} \cdot 1.3400^{86})^{\frac{n}{126}}) = \mathcal{O}^*(1.3236^n)$.

12 Overview

1	If the input graph G contains any vertex v with degree less than two, recursively color $G[V - v]$ and assign v a color different from its neighbors.
2	If the input graph contains a cycle or large tree of degree-three vertices, split the problem into smaller instances according to Lemmas 7.2 and 8.1, recursively attempt to color each smaller instance, and return the first successful coloring found by these recursive calls.
3	Find a maximal bushy forest F in G ([2, Lemma 22]).
4	Modify F' to find a low-magnitude maximal bushy forest F (Lemma 9.1).
5	Find a maximal set T of $K_{1,3}$ subgraphs in $G[V - V(F)]$.
6	While it is possible to increase the size of T by removing one $K_{1,3}$ subgraph and using the vertices in $G[V - V(F \cup T)]$ to form two more $K_{1,3}$ subgraphs, do so.
7	While Lemma 10.4 does not apply, move $K_{1,3}$ subgraphs to increase the number of vertices in T adjacent to trees in the low-magnitude bushy forest with five or six adjacent high-magnitude vertices.
8	Use the network flow algorithm of Lemma [2, Lemma 23] to assign the vertices of $G[V - V(F \cup N(F) \cup T)]$ to trees in T, forming a forest H of height-two trees.
9	Recursively search through all consistent combinations of colors for the bushy forest roots and internal nodes, and for selected vertices in H as described in Lemma 8.2. For each coloring of these vertices, form a (3,2)-CSP instance describing the possible colorings of the uncolored vertices and use the (3,2)-CSP algorithm from Theorem 7.1 to attempt to solve this instance. If one of the (3,2)-CSP instances is solvable, return the resulting coloring. If no (3,2)-CSP instance is solvable, return a flag value indicating that no coloring exists.

Table 4: Updated algorithm. White steps are unaltered from Beigel and Eppstein's algorithm, light blue ones have been updated, and indigo steps are completely new.

We have presented a modification to Beigel and Eppstein's $\mathcal{O}^*(1.3289^n)$ time algorithm for 3-COLORING to improve the runtime to $\mathcal{O}^*(1.3236^n)$. The summary of the algorithm is shown in Table 4.

Our contributions to the algorithm are the modification of the branching rule removing connected subgraphs of eight degree-three vertices to connected subgraphs with nine vertices, which would otherwise have been a bottleneck with its work factor of $\mathcal{O}^*(1.3247^n)$. The analysis of the coloring of trees with three children and at most five grandchildren also relied on connected subgraphs of degree-three vertices consisting of at most seven vertices. Hence, we changed the analysis of these to permit connected subgraphs of eight degree-three vertices and also included the vertices of degree larger than three in the analysis.

Furthermore, we added algorithmic steps to find a low-magnitude maximal bushy forest instead of a normal bushy forest, which limits the number of high-magnitude vertices, the worst-case vertices used in the original analysis of Beigel and Eppstein. We also described a method to cover high-magnitude vertices when there exist many of them by a maximal forest of $K_{1,3}$ trees. This allows us, in combination with the new analysis of the trees with three children and at most five grandchildren, to color these vertices more efficiently. All in all, this algorithm will run in time $\mathcal{O}^*(1.3236^n)$.

Part III
Further Advancements and Observations

In this part, we will discuss some other coloring problems for which we have found minor improvements or observations. The k-COLORING problem given a vertex cover will be discussed in Section 13 and the problem of determining the maximum number of maximal induced k-colorable subgraphs a graph can have is discussed in Section 14.

13 Vertex cover coloring

Parameterized k-COLORING: solving k-COLORING by using some parameter of the graph, such as a set of vertices with a specific property. For example, given a dominating set D of a graph G, we can reduce k-COLORING to $(k-1)$-LIST-COLORING in time $\mathcal{O}^*(k^{|D|})$ by considering all combinations of colors for the dominating set, causing each remaining vertex to have at least one neighbor colored [31]. Notably, this solves 3-COLORING in time $\mathcal{O}^*(3^{|D|})$ as 2-LIST-COLORING is solvable in polynomial time [16].

In this section, we will take a detailed look at k-COLORING given a vertex cover: given a graph G with a vertex cover W, we can solve k-COLORING in time $\mathcal{O}^*((k-1.11)^{|W|})$ [18], which we will elaborate in Section 13.1. As such, this can be faster than the standard algorithm for k-COLORING for sufficiently small vertex covers. However, we found that for large values of k there exists a better algorithm for k-COLORING given a vertex cover by using a reduction to (k,k)-CSP: this will be discussed in Section 13.2.

13.1 Graph coloring in time $\mathcal{O}^*((k-1.11)^{|W|})$

We will discuss Jaffke and Jansen[18]'s algorithm to solve k-COLORING given a vertex cover W in time $\mathcal{O}^*((k-1.11)^{|W|})$. first fix the number of colors $k=3$ and solve 3-COLORING by using a vertex cover. Given a vertex cover W, we know that the remainder of the graph, $G[V-W]$, will be an independent set: no edges can have both endpoints outside the vertex cover. When we find a coloring for the vertices in W, we can verify whether this coloring can be extended to the independent set in polynomial time: for each vertex in the independent set, check whether there exists a color assigned to none of its neighbors. No vertices in the independent set can be adjacent, so coloring some vertex in the independent set will never affect whether a different vertex in the independent set can be colored.

Coloring all vertices in the vertex cover takes time $\mathcal{O}^*(3^{|W|})$, but we can speed it up to time $\mathcal{O}^*(1.89^{|W|})$ by only assigning vertices in the vertex cover to one of the color classes. Then, we can add all vertices in the independent set that are not adjacent to any colored vertices to this color class. If the uncolored part of the graph could be colored by the remaining two colors, it must be bipartite, which we can verify in polynomial time. So, we need to iterate over all independent sets in the vertex cover: if the graph is 3-colorable, then there must exist some independent set of the vertex cover that is used in a valid coloring. In the worst case, every subset of the W vertices is an independent set, meaning this takes time $\mathcal{O}^*(2^{|W|})$.

However, in any valid 3-coloring, there exists some color class that covers at most a third of the vertex cover: we only need to iterate over independent sets of size at most $\frac{|W|}{3}$. We calculate the number of subsets of at most size $\lfloor\frac{|W|}{3}\rfloor$ using the binary entropy function $H(x) = -x\log_2(x) - (1-x)\log_2(1-x)$ to show that there are at most $1.89^{|W|}$ independent sets of size at most $\frac{|W|}{3}$:

$$\sum_{i=0}^{\lfloor\frac{|W|}{3}\rfloor} \binom{|W|}{i} \leq 2^{H(1/3)\cdot|W|} \leq 1.89^{|W|}. \tag{21}$$

As such, we can solve 3-COLORING given a vertex cover W in time $\mathcal{O}^*(1.89^{|W|})$.

We will now extend this result to k-COLORING: for any $k>3$, we solve k-COLORING given a vertex cover by reducing it to $(k-1)$-COLORING. Again, we color every independent set in the vertex cover of size at most $\frac{|W|}{k}$ and find the vertices in $G[V-W]$ that can be added to this independent set: we have found a color class and can solve $(k-1)$-COLORING instead. Using this concept, we can use induction and the fact that $\sum_{i=0}^{n} \binom{n}{i} \cdot a^i \cdot b^{n-i} = (a+b)^n$ to show that we can solve k-COLORING in time $\mathcal{O}^*((k-1.11)^{|W|})$ given a vertex cover W:

$$\sum_{i=0}^{\lfloor \frac{|W|}{k} \rfloor} \binom{|W|}{i} ((k-1)-1.11)^{|W|-i} \leq \sum_{i=0}^{|W|} \binom{|W|}{i} (k-2.11)^{|W|-i} = (k-1.11)^{|W|}. \tag{22}$$

As we know we can solve 3-COLORING in time $\mathcal{O}^*((3-1.11)^{|W|})$, this implies we can solve k-COLORING for any $k \geq 3$ in time $\mathcal{O}^*((k-1.11)^{|W|})$. Furthermore, k-COLORING is solvable in polynomial time for $k < 3$, so the statement holds for any k.

13.2 Improvement for large values of k

In this section, we will discuss our contribution, where we improve k-COLORING given a vertex cover W for $k \geq 12$. We can reduce the k-COLORING problem given a vertex cover W to the (k,k)-CONSTRAINT SATISFACTION PROBLEM: turn every vertex in the vertex cover W into a variable and allow each variable to be assigned any of the k colors. To ensure that the solution to the (k,k)-CSP is a valid solution to k-COLORING, we now have to enforce constraints that ensure no two adjacent vertices receive the same color. Between any two adjacent vertices in W, we simply create constraints for every color, such that not both vertices can be assigned this color. Then, we need to guarantee vertices outside W can be colored. Indeed, any vertex outside W can be colored when there exists some color that is assigned to none of its neighbors. As such, we can create a constraint between every combination of k neighbors of a vertex outside W and every permutation of the colors, such that no selection of k neighbors may cover all k colors. Then, there will always be some color assigned to none of its neighbors, which allows the vertices outside W to have a valid color in any solution of the (k,k)-CSP: if we can solve the (k,k)-CSP, we can solve k-COLORING. Furthermore, every solution to k-COLORING given a vertex cover W will correspond to a solution in the (k,k)-CSP: assign the colors assigned to vertices in W to their corresponding variables. As all solutions to k-COLORING correspond to a solution to the (k,k)-CSP and vice versa, we can use this reduction to solve k-COLORING.

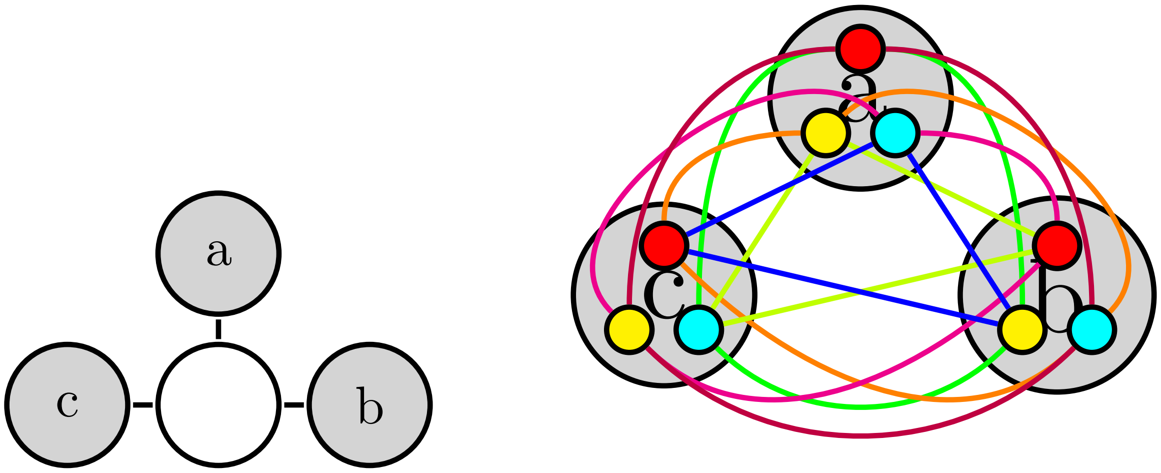


Figure 21: Left: A small instance of 3-COLORING with a vertex cover in grey. Right: the corresponding instance of a (3,3)-CSP instance, where edges of the same color form a triangle, indicating the variable-color pairs involved in one constraint. The white node is removed, and its neighbors are instead involved in constraints indicating they cannot cover all possible colors.

The most efficient algorithm to solve the (k,k)-CSP, by Hertli et al. [17], is a generalization of the algorithm for k-SAT by Paturi et al. [28]. However, they use a complex calculation to determine the runtime of their algorithm and the code presented indicates that the runtime of (k,k)-CSP would peak at $k = 34$ and decline afterward. As such, this would indicate that (k,k)-CSP for $k > 34$ is easier than for

$k = 34$, which is contradictory. Furthermore, it also descends below a runtime of $\mathcal{O}^*(2^n)$ for very large values of k: if this trend would continue infinitely, this also contradicts the strong exponential time hypothesis as it would imply we can solve k-SAT in time $\mathcal{O}^*(2^{o(n)})$.

Instead, Hertli et al. mention a simple algorithm to reduce any (k, k)-CSP into $(2, k)$-CSP by randomly selecting two colors for every vertex. This has a $(k/2)^n$ chance of not removing the valid color for every vertex. After $(k/2)^n$ iterations, we will have a constant chance of success. Given an algorithm for $(2, k)$-CSP running in $\mathcal{O}^*(g_k(n))$ time, we can solve (k, k)-CSP in time $\mathcal{O}^*((k/2)^n \cdot g_k(n))$ using this randomized algorithm that runs for $(k/2)^n$ iterations.

Paturi et al. showed that $(2, k)$-CSP can be solved in time $\mathcal{O}^*(2^{(1-\frac{\sum_{j=1}^{\infty} \frac{1}{j\cdot(j+1/(k+1))}}{k-1})\cdot n})$. Using this formula, we find that this method can improve k-COLORING given a vertex cover W for $k \geq 12$: for $k = 12$ we find that the algorithm runs in time $\mathcal{O}^*((2^{1-\frac{1.543885672}{12-1}} \cdot 6)^n) = \mathcal{O}^*(10.888^n)$, which is an improvement over the time $\mathcal{O}((12 - 1.110)^n) = \mathcal{O}(10.890^n)$ algorithm.

However, one thing to notice with these algorithms is that for higher values of k, the vertex cover has to be increasingly small for it to be useful. For example, consider the time $(k - 1.11)^{|W|}$ algorithm for k-COLORING given a vertex cover W: as k-COLORING can be solved in time $\mathcal{O}^*(2^n)$, we need $(k - 1.11)^{|W|} < 2^n \Rightarrow |W| < \frac{n}{\log_2(k-1.11)}$. So, $|W|$ would need to be increasingly smaller in comparison to n to make this algorithm worth it.

14 Maximal Induced k-Colorable Subgraphs

A *maximal induced k-colorable subgraph*: a subset of the vertices of a graph, such that the graph induced by these vertices is k-colorable and no vertex can be added while maintaining this property. Indeed, maximal independent sets are maximal induced 1-colorable subgraphs. Like maximal independent sets, other maximal induced k-colorable subgraphs can be useful in coloring problems: if we can iterate over the maximal induced 2-colorable subgraphs in time $\mathcal{O}^*(f(n))$, then we can also solve 4-COLORING in time $\mathcal{O}^*(f(n))$. To do this, simply iterate over all maximal induced 2-colorable subgraphs and check whether the rest of the graph is 2-colorable. Similarly, if we can iterate over all maximal induced k-colorable subgraphs in $\mathcal{O}^*(f(n))$ time, we can solve $(k + 2)$-COLORING in time $\mathcal{O}^*(f(n))$.

In this section, we will discuss maximal induced 2-colorable subgraphs in Section 14.1. Then, in Section 14.2, we will discuss improvements on the lower bounds of the number of maximal induced k-colorable subgraphs that can exist in any graph by presenting a family of graphs that establish non-trivial lower bounds. Finally, we will briefly discuss one property of the graphs presented in Section 14.2 and present an algorithm that provides a better upper bound for any graphs with this property.

14.1 Maximal induced 2-colorable subgraphs

An unpublished result states that there exist at most $\mathcal{O}^*(1.7724^n)$ maximal induced 2-colorable subgraphs in a graph on n vertices [6]. For any $k > 2$, there are no known upper bounds on the number of maximal induced k-colorable subgraphs, besides the trivial limit of 2^n induced subgraphs of any graph. The $\mathcal{O}^*(1.7724^n)$ bound was found through a complex branching algorithm. However, the best published upper bound proves that there exist at most $\mathcal{O}^*(12^{n/4}) = \mathcal{O}^*(1.8612^n)$ maximal induced 2-colorable subgraphs [7].

Theorem 14.1 ([7, Theorem 2]). *Any graph contains at most $\mathcal{O}^*(12^{n/4}) = \mathcal{O}^*(1.8613^n)$ maximal induced 2-colorable subgraphs. Moreover, there is an algorithm that takes as input a graph and outputs all its maximal induced 2-colorable subgraphs in time $\mathcal{O}^*(1.8613^n)$.*

In short, this algorithm finds two independent sets and combines them to find a maximal induced 2-colorable subgraph. Naively, this would involve combining every combination of two independent sets: there are at most $3^{n/3}$ maximal independent sets, or $3^{2n/3} = 2.0801^n$ maximal induced 2-colorable subgraphs. However, Byskov's analysis improves upon this with two novel ideas: ensuring that all induced 2-colorable subgraphs are maximal and bounding the number of maximal independent sets that can exist based on the size of the set.

For each possible size of the independent set, Byskov enumerates all independent sets of that size: if the size of the independent set $|I|$ is smaller than $n/4$, then there are at most $\mathcal{O}^*(3^{4\cdot|I|-n} \cdot 4^{n-3\cdot|I|})$ maximal independent sets. Otherwise, there are at most $\mathcal{O}^*(3^{n/3})$. For each maximal independent set I, Byskov iterates over all maximal independent sets in the graph $G[V-I]$: this will take time $\mathcal{O}^*(3^{(n-|I|)/3})$ or $\mathcal{O}^*(3^{5\cdot|I|-n}\cdot 4^{n-4\cdot|I|})$ depending on the whether $|I|$ is smaller than $\frac{n}{4}$ or not. Notice that the second maximal independent set will be found in $G[V-I]$ which contains $n-|I|$ vertices and that all maximal 2-colorable subgraphs found will be maximal, as both independent sets were maximal and will not overlap.

$$\sum_{|I|=1}^{\lfloor n/4\rfloor}(3^{4\cdot|I|-n}\cdot 4^{n-3\cdot|I|}\cdot 3^{5\cdot|I|-n}\cdot 4^{n-4\cdot|I|})+\sum_{|I|=\lfloor n/4+1\rfloor}^{n}(3^{4\cdot|I|-n}\cdot 4^{n-3\cdot|I|}\cdot 3^{(n-|I|)/3})=\mathcal{O}^*(1.8613^n). \quad (23)$$

Solving this equation shows that this approach of enumerating over the maximal independent sets twice runs in time $\mathcal{O}^*(1.8613^n)$. However, if a maximal induced 2-colorable subgraph has l connected components, then it will count this subgraph 2^l times: the colors of each connected component can be flipped while resulting in the same subgraph.

We have seen that we can enumerate maximal induced 2-colorable subgraphs in time $\mathcal{O}^*(1.8613^n)$. However, we would like to be able to enumerate them with *polynomial delay*: each maximal induced 2-colorable subgraph is output in polynomial time after the previous maximal induced 2-colorable subgraph. Then, if a graph contains $\mathcal{O}^*(f(n))$ maximal induced 2-colorable subgraphs, we can enumerate them in time $\mathcal{O}^*(f(n))$.

Johnson et al. [19] proved that one can enumerate overall maximal independent sets with polynomial delay. Specifically, they generate the maximal independent sets in lexicographical order: we assign every vertex a label in $\{1,\ldots,n\}$ on which we impose an alphabetical order. This is notable, as it is NP-Hard to generate the next maximal independent set in lexicographical order, given a graph G and an independent set I. Recently, Conte and Uno [10] showed that we can also iterate over maximal induced 2-colorable subgraphs with polynomial delay by using a new concept called proximity search.

To enumerate all maximal induced 2-colorable subgraphs, we start by finding the first one in lexicographical order by greedily adding the lowest labeled vertex. Notably, every maximal induced 2-colorable subgraph with multiple connected components will have a canonical coloring: only this coloring will be counted.

For every maximal induced 2-colorable subgraph found, they then check whether the maximal induced 2-colorable subgraphs have been found within its proximity. To find the maximal induced 2-colorable subgraphs in the proximity of a given maximal induced 2-colorable subgraph, they consider every vertex v not in the given subgraph and add this to the subgraph. From this, two subgraphs are created: one for removing the neighbors of each color of v. While possible, this subgraph is augmented by greedily adding the first vertex in lexicographic order that maintains the 2-colorability.

As this algorithm happens once for every maximal induced 2-colorable subgraph and none of the operations take exponential time, the maximal induced 2-colorable subgraphs will be enumerated with polynomial delay. Furthermore, Conte and Uno also prove that this method will enumerate through all maximal induced 2-colorable subgraphs, but we will not explain this due to its relative complexity.

However, we cannot enumerate over all maximal induced k-colorable subgraphs with polynomial delay for $k>2$ unless $P=NP$: if a graph is k-colorable, then the only maximal induced k-colorable subgraph will be the entire graph. However, if we could list this with polynomial delay, it would solve k-COLORING in polynomial time, as we would need to list only a single maximal induced k-colorable subgraph. As k-COLORING is NP-complete for $k>2$, enumerating over maximal induced k-colorable subgraphs with polynomial delay is impossible unless $P=NP$.

14.2 Lower bounds

We will now discuss our newfound lower bounds for maximal induced k-colorable subgraphs. These bounds indicate the asymptotic lower bound on how many maximal induced k-colorable subgraphs can exist in any graph. Interestingly, for any $k>1$, we do not know a tight lower bound, meaning there could exist graphs with more maximal induced k-colorable subgraphs than the currently known lower bounds. We can prove that the lower bound for the number of maximal induced k-colorable subgraphs is $\Omega(c^n)$ by finding some graph G with c^n maximal induced k-colorable subgraphs. Using graph G, we can create an infinite family of graphs that have c^n maximal induced k-colorable subgraphs by creating larger graphs with many disjoint copies of G. Consider a graph G_l with l copies of G: every combination of the maximal induced k-colorable subgraph for the l copies of G will form a maximal induced k-colorable subgraph of the larger graph. Then, there are $(c^{n/l})^l = c^n$ maximal induced k-colorable subgraphs in any graph G_l.

Let a *t-blow-up graph*, given some graph G, be a new graph G', where every vertex in G has been replaced with a clique of t vertices. If two vertices in G were adjacent, all vertices in the corresponding cliques in G' will also be adjacent. Let a *twin group* be a clique of t vertices in G' that correspond to a single vertex in G. Furthermore, let two vertices be *twins* when they are both vertices in the same twin group. Finally, let a *blow-up cycle* $B_{x,y}$ be a y-blow-up graph of the cycle C_x.

The current best lower bound for maximal induced 2-colorable subgraphs is $\mathcal{O}^*(1.5926^n)$. This lower bound came from blow-up cycle $B_{5,2}$ as shown in Figure 14.2.

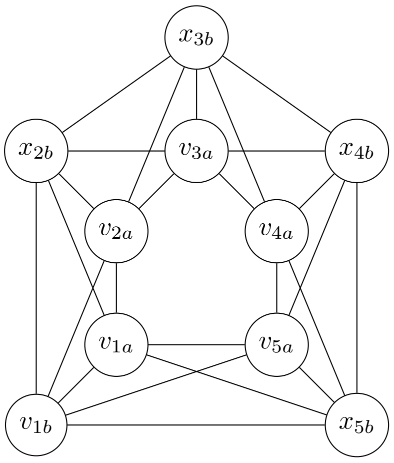


Figure 22: Graph on 10 vertices with 105 maximal induced 2-colorable subgraphs.

If this lower bound would be tight, then this would have big implications for coloring algorithms. Clearly, it would imply that 4-COLORING could be solved in time $\mathcal{O}^*(1.5926^n)$ through enumerating all maximal induced 2-colorable subgraphs and checking whether the remainder of the graph is 2-colorable. Furthermore, it would allow 5-COLORING to be solved in time $\mathcal{O}^*(1.5926 \cdot 1.3236^{3n/5}) = \mathcal{O}^*(1.8843^n)$: enumerate over all maximal 2-colorable subgraphs of at least size $2n/5$ and check whether the remainder of the graph is 3-colorable. Any valid 5-coloring of a graph will have two color classes that together cover at least two-fifth of the graph: we remove these vertices from the graph and validate whether the remaining part of the graph (which covers at most three-fifths of the graph) is 3-colorable.

Since no graphs are known that would contradict the lower bound of $\mathcal{O}^*(1.5926^n)$ maximal induced 2-colorable subgraphs per graph, it is likely that this algorithm could be faster than the current theoretical best known algorithms running in time $\mathcal{O}^*(1.7272^n)$ for 4-COLORING and time $\mathcal{O}^*((2-\epsilon)^n)$ for 5-COLORING. All analyzed graphs contain $\mathcal{O}^*(1.5926^n)$ maximal induced k-colorable subgraphs, so we can iterate over them relatively efficiently in these graphs.

We are not aware of any prior research on the lower bounds of maximal induced k-colorable subgraphs for any $k > 2$. However, it is simple to optimize the number of maximal induced k-colorable subgraphs on some cluster graph: maximize $\binom{n}{k}^{1/n}$.

We have found new lower bounds that improve upon these cliques for every $k \geq 3$ by generalizing the result found for maximal induced 2-colorable subgraphs. These use a blow-up cycle, where every vertex is blown up to become a clique of k vertices. We have tested different blow-up amounts than k for the cycles, but k appears to be optimal. The comparison between our results and these cliques is shown in Table 5.

Lemma 14.2. *Given a blow-up cycle $B_{x,k}$ where x is odd, every subgraph of $(x-1) \cdot k/2$ vertices where no two adjacent twin groups contain more than k vertices is a maximal induced k-colorable subgraph.*

Proof. Notice that two adjacent twin groups also form a clique: no more than k vertices in any clique can be colored in any valid k-coloring. We want to show that any graph respecting this constraint with $(x-1) \cdot k/2$ vertices is k-colorable and maximal.

Let b_i be the ith twin group and let u_i be the number of colors not used in b_i nor b_{i+1} (wrapping around to one when $i = x$: as there are an odd number of twin groups, no color can appear in more than $\lfloor x/2 \rfloor$ twin groups: two adjacent twin groups would both include this color. As such, for every color, there exist at least two adjacent twin groups where the color is not used. So, $\sum_{i=1}^{x} u_i \geq k$.

Now, we show that for every possible combination of u-values such that $\sum_{i=1}^{x} u_i = k$, the subgraph is k-colorable. Since there are exactly k cases where two adjacent twin groups both avoid the same color, we can assign each color to one of those cases. Then, we can greedily assign twin groups that will have this color: there already exist two that will not have this color and since no two other adjacent twin groups will avoid the color, their neighbors will include it. That way, we will find a coloring, given a set of u_i values, that will produce a valid coloring. Every color will occur exactly $\frac{x-1}{2}$ times, so there are in total $\frac{(x-1)\cdot k}{2}$ vertices in the subgraph.

For a given configuration of u-values, where $\sum_{i=1}^{x} u_i = k$, we can find the corresponding b-values. Due to the method described above never assigning a color to two adjacent twin groups, it will always produce a valid k-coloring and as every twin group will either contain a color or have an adjacent twin group that contains the color, the induced k-colorable subgraph is maximal as well. $\square$

We can use the set of subgraphs of a blow-up cycle for which $\sum_{i=1}^{x} u_i = k$ and $\forall i \in 1, \ldots, x : s_i + s_{i+1} \leq k$ (such that $i+1$ wraps to one when $i = x$) as a lower bound of the total number of maximal induced k-colorable subgraphs. Notice that it is a lower bound, as there might be more of these subgraphs, such as in Figure 14.2, where $\sum_{i=1}^{x} u_i > k$.

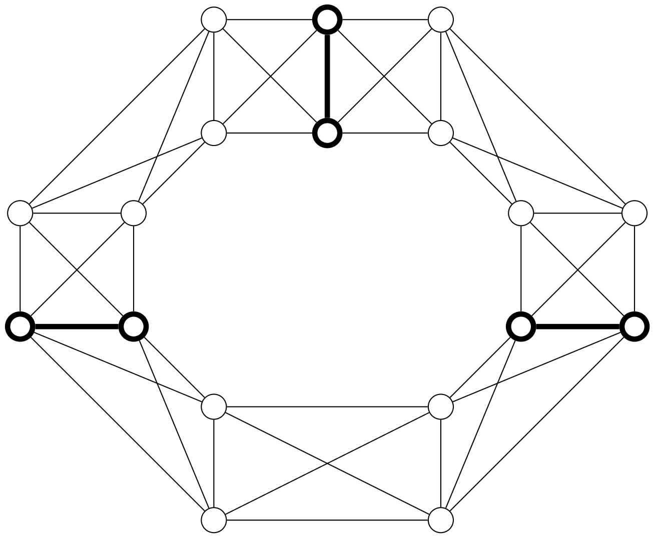


Figure 23: Graph $B_{9,2}$ with a highlighted maximal induced 2-colorable subgraph.

Lemma 14.3. *Given a blow-up cycle $B_{x,k}$ where x is odd, there exist at least $\sum_{\{b_1,\ldots,b_x\}\in\mathcal{B}} \prod_{i=1}^{k} \binom{k}{b_i}$ maximal induced k-colorable subgraphs, where $\mathcal{B}$ is the set of all b-values such that $\sum_{b=1}^{x} = (x-1)\cdot k/2$ and for all values $1 \leq i \leq x : b_i + b_{i+1} \leq k$ (where $i + 1 = 1$ when $i = x$).*

Proof. For each possible configuration of b-values, we can include any b_i out of k vertices per twin group. Then, the sum of all possible b-configurations finds all possible maximal induced k-colorable subgraphs where $\sum_{i=1}^{k} u_i = k$ and for any $1 \leq i \leq k : u_i + u_{i+1} \leq k$. As such, this is a lower bound to the total number of maximal induced k-colorable subgraphs in $B_{x,k}$. $\square$

k-coloring	Clique K_x	Lower bound (Clique)	Blown-up cycle C_x	t-blow-up	Lower bound (Blow-up cycle)
1	3	1.44224957	3	1	1.44224957
2	5	1.58489319	5	2	1.59264481
3	7	1.66180916	5	3	1.68584416
4	9	1.71149055	5	4	1.74216951
5	11	1.74678792	7	5	1.78434967
6	13	1.77340946	7	6	1.814545
7	15	1.79433379	7	7	1.8370999
8	17	1.81128734	7	8	1.85453805
9	19	1.82534816	7	9	1.86838589
10	21	1.8372282	9	10	1.87968758
11	23	1.84741852	9	11	1.88996704
12	25	1.85626996	9	12	1.89860994
13	27	1.86404047	9	13	1.90596655
14	30	1.8710601	9	14	1.91229507
15	32	1.87723595	9	15	1.91778983
16	34	1.88278235	9	16	1.92259978
17	36	1.88779477	9	17	1.92684092
18	38	1.89234991	9	18	1.93060487
19	40	1.89651012	9	19	1.93396489
20	42	1.90032666	11	20	1.93723983

Table 5: Maximal induced k-colorable subgraphs in cliques and cycle blow-up graphs

Now, we have optimized the number of vertices in the cycle by trying many cycles $B_{x,k}$: both lower and higher values of x return worse lower bounds. Changing k to a different value never seems to increase the number of maximal induced k-colorable subgraphs. The lower bounds found, formulated as the c in $\Omega(c^n)$ maximal induced k-colorable subgraphs, are shown in Table 5.

14.3 Twin groups

Blow-up graphs appear useful to increase the number of maximal induced 2-colorable subgraphs of a graph: when creating a blow-up graph of any graph, every maximal induced 2-colorable subgraph consisting of n_i vertices will cause 2^{n_i} maximal induced 2-colorable subgraphs to exist in the blow-up graph. For every vertex in the original maximal induced 2-colorable subgraph, we can choose one of the two corresponding vertices in the blow-up graph. More maximal induced 2-colorable subgraphs will exist due to every independent set turning into a maximal induced 2-colorable subgraph by selecting both vertices in the corresponding twin group. Furthermore, any combination between the two creates a maximal induced 2-colorable subgraph as well: where some vertices have to be independent, while others form a maximal induced 2-colorable subgraph. This analysis can also be extended to any maximal induced k-colorable subgraph, but with an increasing number of cases. This method models what happens when a vertex can be given multiple colors, as the twin groups in the blow-up graphs can be seen to model a single vertex.

If using blow-up graphs is indeed optimal due to this property, we would be able to find a better upper bound on the number of maximal induced k-colorable subgraphs for all $k > 2$:

Lemma 14.4. *Given a k-blow-up graph G on n vertices, it can contain at most $(2^k - 1)^{n/k} = (2 - \epsilon_k)^n$ maximal induced k-colorable subgraphs.*

Proof. We can describe a k-blow-up graph as a collection of n/k twin groups. We will now describe all possible k-colorable subgraphs by coloring each twin group in one of $2^k - 1$ ways: we consider for every vertex whether it is in the subgraph, but will never include all vertices in the twin group.

Clearly, we find all maximal induced k-colorable subgraphs that do not include all vertices in any twin group. Now, consider a maximal induced k-colorable subgraph that does include all vertices in some twin group: then no vertices in adjacent twin groups may be included. For any subgraph we find with a twin group with at least one included vertex for which no adjacent twin groups have any vertices included, we know it is not maximal: all vertices in this twin group could be included, but that is never the case as we did not consider this possibility. We can now augment all twin groups with no neighboring vertices included to include all vertices to also list all maximal independent k-colorable subgraphs that include all vertices in some twin group. Through this operation, we will list all maximal induced k-colorable subgraphs:

any maximal induced k-colorable subgraphs that include all of some twin group will be represented by an induced k-colorable subgraph that is not maximal, where we augment all non-empty twin groups for which no neighbors are included.

As each twin group can be colored in $2^k - 1$ ways and there are n/k twin groups, we discover $(2^k - 1)^{n/k} = (2 - \epsilon_k)^{n/k}$ possible maximal induced k-colorable subgraphs in k-blow-up graphs on n vertices. □

Indeed, we can use this concept as an algorithm to find all maximal induced k-colorable subgraphs, by verifying for each subset of vertices whether it is k-colorable. We can check whether a graph is 2-colorable in polynomial time, so it will run in $\mathcal{O}^*(3^{n/2}) = \mathcal{O}^*(1.7321^n)$ time for maximal induced 2-colorable subgraphs, which is an improvement over the generic time $\mathcal{O}^*(1.7724^n)$ algorithm to enumerate maximal induced 2-colorable subgraphs on these blow-up graphs. For any $k > 3$, no previous algorithm was known that will never iterate over exponentially fewer than $\mathcal{O}^*(2^n)$ subgraphs, so this algorithm provides an improvement as well. However, verifying whether the subgraph is k-colorable for $k > 3$ will take exponential time per subgraph checked: we will take more than time $\mathcal{O}^*(2^n)$ to iterate over all maximal induced k-colorable subgraphs for $k > 3$.

15 Conclusion

We have presented our research on various topics in graph theory. Most notably, we have made strides in solving 3-COLORING, where we improved the runtime from time $\mathcal{O}^*(1.3289^n)$ to time $\mathcal{O}^*(1.3236^n)$. To do this, we adapted Beigel and Eppstein's time $\mathcal{O}^*(1.3289^n)$ algorithm. We introduced the maximal low-magnitude bushy forest, which limited the number of difficult-to-color vertices: the vertices outside the bushy forest. Furthermore, we partitioned the vertices adjacent to the bushy forest:

1. Degree-three vertices: a limited number of these can exist.
2. Two adjacent vertices in the bushy forest: fewer vertices can be adjacent to the bushy forest.
3. Other "high-magnitude" vertices: large numbers of these force graph structures that are easier to solve.

Using these ideas, we formulated constraints on the number of vertices that could appear in each set of vertices relative to other sets of vertices and used a linear program to calculate how many vertices in each set the graph that would take the longest to solve would have. Through this, we reached the new result that we can solve 3-COLORING in time $\mathcal{O}^*(1.3236^n)$. Additionally, this also allows us to solve 4-COLORING in time $\mathcal{O}^*(1.7247^n)$.

Furthermore, we made an observation that we can solve k-COLORING given a vertex cover W using the best known algorithm for (k, k)-CSP. For large values of k, this allows us to solve this problem faster than the known time $\mathcal{O}^*((k - 1.11)^{|W|})$ algorithm for this problem.

Finally, we talked about maximal induced k-colorable subgraphs. Here, we discussed that we can generate all maximal induced 2-colorable subgraphs with polynomial delay, and mentioned that this is not possible for maximal induced k-colorable subgraphs for $k > 3$ unless $P = NP$. The known lower bound on the number of maximal induced 2-colorable subgraphs is $\mathcal{O}^*(1.5926^n)$. If future research could prove this bound is tight, or show a new upper bound of $\mathcal{O}((1.7247 - \epsilon)^n)$ maximal induced 2-colorable subgraphs, it would imply improvements for 4-COLORING. If the bound were to be tight, or close to tight, we could even improve 5-COLORING.

We also found non-trivial lower bounds for the maximum number of maximal induced k-colorable subgraphs in a graph for $3 \leq k \leq 20$, which can likely be extended to $k > 20$. Within this problem, we used the blow-up graphs of cycles, where we replaced every vertex in the cycle with a clique of k vertices. Furthermore, on these blow-up graphs, we have shown that there also exists an upper bound on the number of maximal induced k-colorable subgraphs. Future research could center around the properties of the graphs with the most maximal induced 2-colorable subgraphs: similar to the fact that the graph with the most maximal independent sets must be a cluster graph. In particular, if it were to be proven that the graph with the most maximal induced k-colorable subgraphs is a k-blow-up graph, we showed that this would imply that there exist $\mathcal{O}((2 - \epsilon_k)^n)$ maximal induced k-colorable subgraphs for any k.

16 Acknowledgements

We thank Carla Groenland and Jesper Nederlof for their detailed feedback on earlier drafts of this paper and helpful discussions about graph coloring.

References

[1] Robert A. Beeler. *How to count an introduction to combinatorics and its applications.* Springer International Publishing, 2015.

[2] Richard Beigel and David Eppstein. "3-coloring in time O(1.3289^n)". In: *Journal of Algorithms* 54.2 (2005), pp. 168–204. DOI: 10.1016/j.jalgor.2004.06.008.

[3] Norman Biggs, E. Keith Lloyd, and Robin J. Wilson. *Graph theory 1736-1936: By Norman L. Biggs, E. Keith Lloyd, Robin J. Wilson.* Clarendon Press, 1977.

[4] A. K. Bincy and B. Jeba Presitha. "Graph Coloring and its Real Time Applications an Overview Research". In: 2017.

[5] Andreas Björklund, Thore Husfeldt, and Mikko Koivisto. "Set partitioning via inclusion-exclusion". In: *SIAM Journal on Computing* 39.2 (July 2009), pp. 546–563. DOI: 10.1137/070683933.

[6] Jesper Byskov and David Eppstein. "An Algorithm for Enumerating Maximal Bipartite Subgraphs". unpublished. 2004.

[7] Jesper Makholm Byskov. "Enumerating maximal independent sets with applications to graph colouring". In: *Operations Research Letters* 32.6 (2004), pp. 547–556. DOI: 10.1016/j.orl.2004.03.002.

[8] Gary Chartrand and Ping Zhang. *Chromatic graph theory.* CRC Press/Balkema, 2009.

[9] Charles J. Colbourn. "The complexity of completing partial Latin squares". In: *Discrete Applied Mathematics* 8.1 (1984), pp. 25–30. DOI: 10.1016/0166-218x(84)90075-1.

[10] Alessio Conte and Takeaki Uno. "New polynomial delay bounds for maximal subgraph enumeration by proximity search". In: *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing* (2019). DOI: 10.1145/3313276.3316402.

[11] Jean-Paul Delahaye. "The science behind Sudoku". In: *Scientific American* 294.6 (June 2006), pp. 80–87. DOI: 10.1038/scientificamerican0606-80.

[12] David Eppstein. "Small maximal independent sets and faster exact graph coloring". In: *Graph Algorithms and Applications 4* (2006), pp. 131–140. DOI: 10.1142/9789812773296_0006.

[13] Fedor V. Fomin, Serge Gaspers, and Saket Saurabh. "Improved exact algorithms for counting 3- and 4-colorings". In: *Lecture Notes in Computer Science* (2007), pp. 65–74. DOI: 10.1007/978-3-540-73545-8_9.

[14] Fedor V. Fomin et al. "On two techniques of combining branching and treewidth". In: *Algorithmica* 54.2 (2007), pp. 181–207. DOI: 10.1007/s00453-007-9133-3.

[15] Minas Giannekas. *Europe: Create a custom map.* URL: https://www.mapchart.net/europe.html.

[16] Sylvain Gravier, Daniel Kobler, and Wieslaw Kubiak. "Complexity of list coloring problems with a fixed total number of colors". In: *Discrete Applied Mathematics* 117.1-3 (2002), pp. 65–79. DOI: 10.1016/s0166-218x(01)00179-2.

[17] Timon Hertli et al. "The PPSZ algorithm for constraint satisfaction problems on more than two colors". In: *Lecture Notes in Computer Science* (2016), pp. 421–437. DOI: 10.1007/978-3-319-44953-1_27.

[18] Lars Jaffke and Bart M. Jansen. "Fine-grained parameterized complexity analysis of graph coloring problems". In: *Lecture Notes in Computer Science* (2017), pp. 345–356. DOI: 10.1007/978-3-319-57586-5_29.

[19] David S. Johnson, Mihalis Yannakakis, and Christos H. Papadimitriou. "On generating all maximal independent sets". In: *Information Processing Letters* 27.3 (1988), pp. 119–123. DOI: 10.1016/0020-0190(88)90065-8.

[20] Richard M. Karp. "Reducibility among combinatorial problems". In: *Complexity of Computer Computations* (1972), pp. 85–103. DOI: 10.1007/978-1-4684-2001-2_9.

[21] Anthony Donald Keedwell and József Dénes. *Latin squares and their applications.* Elsevier, 2015.

[22] S. Khor. "Application of graph colouring to biological networks". In: *IET Systems Biology* 4.3 (2010), pp. 185–192. DOI: 10.1049/iet-syb.2009.0038.

[23] E.L. Lawler. "A note on the complexity of the chromatic number problem". In: *Information Processing Letters* 5.3 (1976), pp. 66–67. DOI: 10.1016/0020-0190(76)90065-x.

[24] Gary Lewandowski and Anne Condon. "Experiments with parallel graph coloring heuristics and applications of graph coloring". In: *Cliques, Coloring, and Satisfiability* (1996), pp. 309–334. DOI: 10.1090/dimacs/026/15.

[25] Daniel Lokshtanov, Dániel Marx, and Saket Saurabh. "Known algorithms on graphs of bounded treewidth are probably optimal". In: *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms* (2011). DOI: 10.1137/1.9781611973082.61.

[26] László Lovász. "Coverings and coloring of hypergraphs". In: *Proceedings of the Fourth Southeastern Conference on Combinatorics, Graph Theory, and Computing.* Utilitas Math. Winnipeg, Man., 1973, pp. 3–12.

[27] J. W. Moon and L. Moser. "On cliques in graphs". In: *Israel Journal of Mathematics* 3.1 (1965), pp. 23–28. DOI: 10.1007/bf02760024.

[28] R. Paturi et al. "An improved exponential-time algorithm for K-Sat". In: *Proceedings 39th Annual Symposium on Foundations of Computer Science (Cat. No.98CB36280)* (Nov. 1998). DOI: 10.1109/sfcs.1998.743513.

[29] Neil Robertson and P.D. Seymour. "Graph minors. I. Excluding a forest". In: *Journal of Combinatorial Theory, Series B* 35.1 (1983), pp. 39–61. DOI: 10.1016/0095-8956(83)90079-5.

[30] Ingo Schiermeyer. "Deciding 3-colourability in less than $\mathrm{O}(1.415^n) steps$". In: *Graph-Theoretic Concepts in Computer Science* (1994), pp. 177–188. DOI: 10.1007/3-540-57899-4_51.

[31] Or Zamir. "Breaking the 2^n barrier for 5-coloring and 6-coloring". In: *CoRR* abs/2007.10790 (2020). arXiv: 2007.10790. URL: https://arxiv.org/abs/2007.10790.