DETERMINING A GRAPH'S CHROMATIC NUMBER FOR PART CONSOLIDATION IN AXIOMATIC DESIGN

A Thesis

Presented to

The Faculty of the Department of Mathematics

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Jeffery A. Cavallaro

December 2019

The Designated Thesis Committee Approves the Thesis Titled

DETERMINING A GRAPH'S CHROMATIC NUMBER FOR PART CONSOLIDATION IN AXIOMATIC DESIGN

by

Jeffery A. Cavallaro

APPROVED FOR THE DEPARTMENT OF MATHEMATICS

SAN JOSÉ STATE UNIVERSITY

December 2019

Sogol Jahanbekam, Ph.D.	Department of Mathematics
Wasin So, Ph.D.	Department of Mathematics
Jordan Schettler, Ph.D.	Department of Mathematics

ABSTRACT

DETERMINING A GRAPH'S CHROMATIC NUMBER FOR PART CONSOLIDATION IN AXIOMATIC DESIGN

by Jeffery A. Cavallaro

Mechanical engineering design practices are increasingly moving towards a framework called *axiomatic design* (AD), which starts with a set of independent *functional requirements* (FRs) for a manufactured product. A key tenet of AD is to decrease the *information content* of a design in order to increase the chance of manufacturing success. One important way to decrease information content is to fulfill multiple FRs by a single part: a process known as *part consolidation*. Thus, an important parameter when comparing two candidate designs is the minimum number of parts needed to satisfy all of the FRs. One possible method for determining the minimum number of parts is to represent the problem by a graph, where the vertices are the FRs and the edges represent the need to separate their endpoint FRs into separate parts. The answer then becomes the solution to a vertex coloring problem: finding the chromatic number of such a graph. Unfortunately, the chromatic number problem is known to be NP-hard. This research investigates a new algorithm that determines the chromatic number for a graph and compares the new algorithm's computer runtime performance to other well-known algorithms using random graph analysis.

ACKNOWLEDGMENTS

The author wishes to thank the faculty at SJSU for their time and support, in particular: graduate advisor Dr. Sogol Jahanbekam, undergraduate advisor Dr. Tim Hsu, committee members Dr. Wasin So and Dr. Jordan Schettler, and Mathematics department chairperson Dr. Bem Cayco. Special thanks are also extended to our mechanical engineering colleagues Dr. Sara Behdad and graduate student Praveen Gopalakrishnan at SUNY, Buffalo.

This material is based upon work supported by the National Science Foundation–USA under grants #CMMI-1727190 and CMMI-1727743. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 AXIOMATIC DESIGN

The axiomatic design (AD) framework was developed in the late 20^{th} century by Professor Nam P. Suh while at MIT and the NSF [1]. This was in response to concern in the engineering community that *design* was being practiced almost exclusively as an ad-hoc creative endeavor with very little in the way of scientific discipline. In the words of Professor Suh:

> It [design] might have preceded the development of natural sciences by scores of centuries. Yet, to this day, design is being done intuitively as an art. It is one of the few technical areas where experience is more important than formal education [1].

Professor Suh was not making these claims in an educational vacuum, but in the shadow of several recent major design failures such as the Union Carbide plant disaster in India, nuclear power plant accidents at Three Mile Island and Chernobyl, and the Challenger space shuttle O-ring failure. Furthermore, Professor Suh asserts that design-related issues resulting in production problems and operating failures were increasingly happening in everything from consumer products to big-ticket items. As a result, AD has been widely adopted by companies to promote efficiency and accuracy in the design process, resulting in more reliable products and reduced manufacturing costs [2].

The following sections provide an overview of axiomatic design as specified in detail by Professor Suh [1], [3], and summarized by Behdad, et al. [4], [5]. Following the overview is a description of how an algorithm like the algorithm proposed by this research can be a helpful tool to a designer using the AD framework.

1.1 Design

Design is defined as the process by which it is determined *what* needs to be achieved and then *how* to achieve it. Thus, the decisions on what to do are just as important as the decisions on how to do it. *Creativity* is the process by which experience and intuition are used to generate solutions to perceived needs. This includes pattern matching to and

adapting existing solutions and synthesizing new solutions. Thus, creativity plays a vital role in design. Different designers may approach the same problem differently, and their varying levels of creativity may lead to very different, yet still plausible, solutions. Therefore, there needs to be a design-agnostic method for comparing different designs with the goal of selecting the best one.

This discussion will sound familiar to mathematicians, since creativity is a very important part of solving math problems, particularly in the writing of proofs. Thus, the field of mathematics has established various tests on what constitutes a good proof. For example:

- Does every conclusion result by proper implication from existing definitions, axioms, and previously proved conclusions?
- Is direct proof, contrapositive proof, proof by contradiction, or proof by induction the best approach for a particular problem?
- Do proofs by induction contain clear basic, assumptive, and inductive steps?
- Are all subset and equality relationships properly proved via membership implication?
- Are all necessary cases included and stated in a mutually exclusive manner?
- Are degenerate cases sufficiently highlighted?
- Are all equivalences proved in a proper circular fashion?
- Are key and reused conclusions highlighted in lemmas?

In short, Professor Suh was looking for a similar framework for the more general concept of design.

1.2 The Axiomatic Design Framework

The *best* design among a set of candidates is the design that exactly satisfies a clearly defined set of needs and has the greatest probability of success in meeting those needs. In a desire not to hinder the creative element needed for design, yet provide some

methodology to distinguish bad designs from good designs from better designs, the diagram in Fig. 1 establishes the overall framework for axiomatic design.

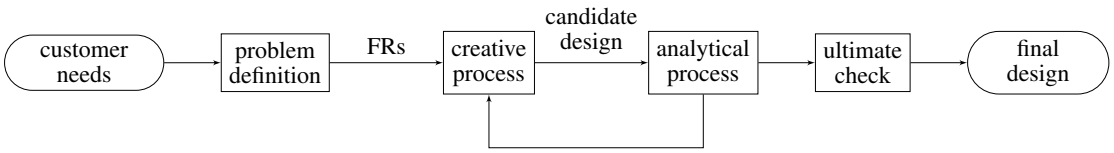


Fig. 1. The axiomatic design framework.

Design starts with the desire to satisfy a set of precisely stated *customer needs*. The term *customer* refers to any entity that expresses needs, and can be as varied as individuals, organizations, or society. In the *problem definition* phase, the designer determines how the customer needs will be met by generating a minimal list of *functional requirements* (FRs) that directly and exclusively fulfill the needs. It is this list of FRs that determines exactly *what* is to be accomplished.

Once the set of FRs has been determined, the designer begins the *creative process* by mapping the FRs into solutions that are embodied in so-called *design parameters* (DPs). The DPs contain all of the information concerning *how* the various FRs are to be satisfied: parts lists, drawings, specifications, etc. The FRs exist in a design-agnostic *functional space* and the DPs exist in a solution-specific *physical space*. It is the designer's job to provide the most efficient mapping between the two spaces. This process is represented by Fig. 2.

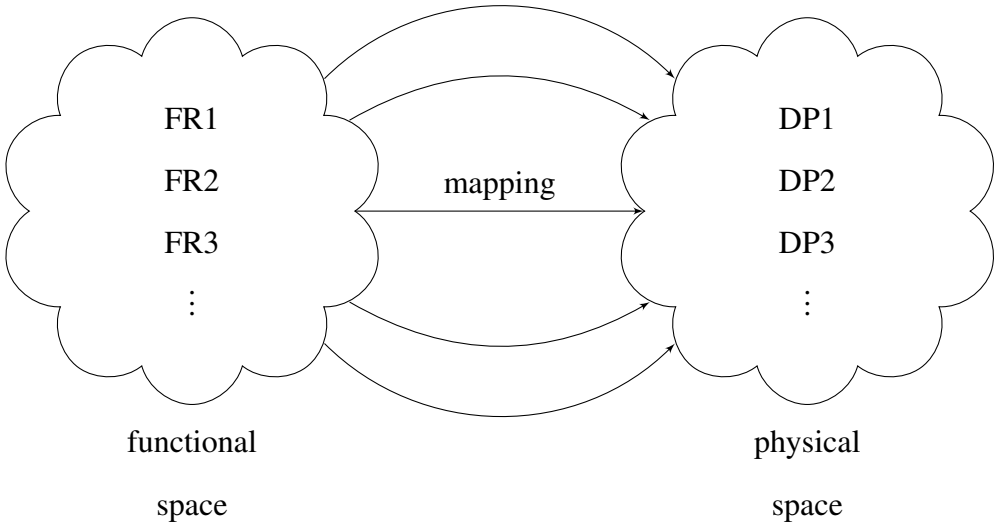


Fig. 2. Mapping FRs to DPs.

Simple problems may require only one level of FRs; however, more complicated designs may require a hierarchial structure of FRs from more general to more detailed requirements. This type of design is often referred to as *top-down* design. Each individual FR layer has its own DP mapping. In fact, the mapping process on one level should be completed prior to determining the FRs for the next level. This is because DP choices on one level may affect requirements on the next level. For example, consider a FR related to a moving part in a design. The DP for this FR could specify that the part be moved either manually or automatically. Each choice would result in different FRs for the actual mechanism selected by the DP.

The FR/DP mapping at each level in the design hierarchy is described by the *design equation*, which is shown in Equation 1.

$$[\mathrm{FR}] = [\mathrm{A}][\mathrm{DP}] \tag{1}$$

The design equation is a matrix equation that maps a vector of m FRs to a vector of n DPs via an $m \times n$ design matrix A. As will be shown later in this section, good designs require

$m = n$. A full discussion of the design matrix element values is well beyond the scope of this research. Instead, the following two summary values are used:

$$A_{ij} = \begin{cases} X, & \text{FR}_i \text{ depends on DP}_j \\ 0, & \text{FR}_i \text{ does not depend on DP}_j \end{cases}$$

Since the FR/DP mapping is non-unique, there needs to be a method to compare different plausible designs so that the best design can be selected as the final design. Thus, the framework in Fig. 1 includes an *analytical process*, where designs are judged by a set of axioms, corollaries, and theorems that specify the properties common to all good designs. Once the best design, according to this analysis, is selected, it undergoes an *ultimate check* to make sure that it exactly meets all of the customer's needs. If so, then that design is selected as the final design.

1.3 The Axioms

The analytical process is based on two main axioms: the independence axiom and the information axiom. This section describes these axioms and their related corollaries and theorems.

The independence axiom [1] imposes a restriction on the FR/DP mapping:

Axiom 1 (The Independence Axiom)**.** *An optimal design always maintains the independence of the FRs. This means that the FRs and DPs are related in such a way that a specific DP can be adjusted to satisfy its corresponding FR without affecting other FRs.*

The ideal case is when the design matrix is a diagonal matrix and so each FR is mapped to and is satisfied by exactly one DP. This is referred to as an *uncoupled* design, which is demonstrated in Fig. 3. Uncoupled designs completely adhere to the independence axiom.

$$\begin{bmatrix} \text{FR1} \\ \text{FR2} \\ \text{FR3} \end{bmatrix} = \begin{bmatrix} X & 0 & 0 \\ 0 & X & 0 \\ 0 & 0 & X \end{bmatrix} \begin{bmatrix} \text{DP1} \\ \text{DP2} \\ \text{DP3} \end{bmatrix}$$

Fig. 3. An example of an uncoupled design.

The next best situation is when the design matrix is a lower-triangular matrix. The idea is to finalize the first DPs before moving on to the later DPs. Thus, DP_i can be adjusted without affected FR_1 through FR_{i-1}. This is referred to as a *decoupled* design, which is demonstrated in Fig. 4. Although decoupled designs do not completely adhere to the independence axiom, they may be reasonable compromises in designs that address complex problems.

$$\begin{bmatrix} \text{FR1} \\ \text{FR2} \\ \text{FR3} \end{bmatrix} = \begin{bmatrix} X & 0 & 0 \\ X & X & 0 \\ X & X & X \end{bmatrix} \begin{bmatrix} \text{DP1} \\ \text{DP2} \\ \text{DP3} \end{bmatrix}$$

Fig. 4. An example of a decoupled design.

The worst solution is a non-triangular matrix, where every change in a DP affects multiple FRs in an unconstrained fashion. This is referred to as a *coupled* design, which is demonstrated in Fig. 5. Coupled designs are in complete violation of the independence axiom and generally should be decoupled by reworking the FRs or by adding additional DPs.

$$\begin{bmatrix} \text{FR1} \\ \text{FR2} \\ \text{FR3} \end{bmatrix} = \begin{bmatrix} X & X & X \\ X & X & X \\ X & X & X \end{bmatrix} \begin{bmatrix} \text{DP1} \\ \text{DP2} \\ \text{DP3} \end{bmatrix}$$

Fig. 5. An example of a coupled design.

Unfortunately, adding additional DPs runs counter to the second axiom: the information axiom [1].

Axiom 2 (The Information Axiom)**.** *The best design is a functionally uncoupled design that has the minimum information content.*

The amount of *information* contained in a particular DP is inversely related to the probability that the DP can successfully satisfy its corresponding FR(s) by Equation 2.

$$I = \log_2 \left(\frac{1}{p} \right) \tag{2}$$

where p is the probability of success and I is measured in bits. This probability must take into consideration such things as tolerances, ease of manufacture, and failure rates. The information content of a design is then the sum of the information content of its individual DPs.

From these two axioms come the following seven corollaries [1]:

Corollary 1. *Decouple or separate parts or aspects of a solution if the FRs are coupled or become interdependent in the designs proposed.*

Corollary 2. *Minimize the number of FRs.*

Corollary 3. *Integrate design features in a single physical part if FRs can be independently satisfied in the proposed solution.*

Corollary 4. *Use standardized or interchangeable parts if the use of these parts is consistent with the FRs.*

Corollary 5. *Use symmetrical shapes and/or arrangements if they are consistent with the FRs.*

Corollary 6. *Specify the largest allowable tolerance in stating FRs.*

Corollary 7. *Seek an uncoupled design that requires less information than coupled designs in satisfying a set of FRs.*

The theorems that arise from these axioms and corollaries are used to prove that an optimal design results from a square design matrix. In other words, the number of FRs should be equal to the number of DPs. First, consider the case where there are more FRs than DPs. This forces a single DP to be mapped to multiple FRs. Otherwise, some FRs cannot be satisfied by the DPs. This result is stated in Theorem 1 [1].

Theorem 1. *When the number of DPs is less that the number of FRs, either a coupled design results or the FRs cannot be satisfied.*

A possible solution to this problem is given by Theorem 2 [1].

Theorem 2. *A coupled design due to more FRs than DPs can be decoupled by adding new DPs if the additional DPs result in a lower triangular design matrix.*

An example is show in Fig. 6. Note that the addition of DP3 results in a decoupled design.

$$\begin{bmatrix}\text{FR1}\\ \text{FR2}\\ \text{FR3}\end{bmatrix} = \begin{bmatrix}X & 0\\ X & X\\ X & X\end{bmatrix}\begin{bmatrix}\text{DP1}\\ \text{DP2}\end{bmatrix} \Longrightarrow \begin{bmatrix}\text{FR1}\\ \text{FR2}\\ \text{FR3}\end{bmatrix} = \begin{bmatrix}X & 0 & 0\\ X & X & 0\\ X & X & X\end{bmatrix}\begin{bmatrix}\text{DP1}\\ \text{DP2}\\ \text{DP3}\end{bmatrix}$$

Fig. 6. Decoupling a design by adding DPs.

Next, consider the case where the number of FRs is less than the number of DPs. Assuming that the design is not coupled, this means that either a DP exists that does not address any FRs or multiple DPs exist that address a single FR and hence can be integrated into a single DP. Such a design is called a *redundant* design. This is addressed by Theorem 3 [1].

Theorem 3. *When there are less FRs than DPs then the design is either coupled or redundant.*

Finally, the previous three theorems lead to the conclusion in Theorem 4 [1].

Theorem 4. *In an ideal design, the number of FRs is equal to the number of DPs.*

1.4 Part Consolidation

One particularly important design parameter is the number of parts in a product design. According to Professor Suh:

> Poorly designed products often cost more because they use more materials or parts than do well-designed products. They are often difficult to manufacture and maintain [1].

Decreasing the number of parts in a design while maintaining the independence of the FRs is consistent with Corollary 3 and lowers the information content of the design. In fact, Tang, et al. [6] describe how part consolidation reduces the weight and complexity of a final product while boosting reliability and reducing cost.

An informative example of part consolidation is the combination can/bottle opener shown in Fig. 7. The design of this handy utensil has two FRs, shown in Table 1. Both FRs are consolidated into a single part, yet remain independent as long as there is no desire to open a can and a bottle simultaneously (although that would be a popular trick around a campfire).

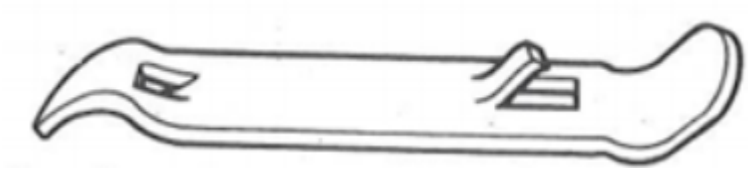


Fig. 7. A part consolidation example.

Table 1

Opener Functional Requirements

FR1	Open beverage cans
FR2	Open beverage bottles

1.5 Research Goals

The primary goal of this research is to provide designers with a tool that they can use to determine the minimum number of parts required to realize a particular design at a particular level in a FR/DP hierarchy. The designer is required to construct a graph whose vertices are the FRs and whose edges indicate that the endpoint FRs need to be realized by separate parts due to various design constraints. How these edges are actually determined is beyond the scope of this research. Adjacent FRs are candidates for part consolidation. The goal is to find the chromatic number of the resulting graph, which corresponds to the minimum number of parts required for the candidate design.

To be a viable tool, an algorithm running as a computer program must be able to deliver an answer in a reasonable amount of time. Unfortunately, finding the chromatic number of a graph is known to be an inherently intractable problem [7]. The nature of such problems is discussed in Section 3, but for now this is taken to mean that the time required to find a solution grows exponentially with the number of vertices in the graph. Furthermore, although not fully proven, it appears likely that there is no way to do any better than an exponential-time solution. Therefore, it is necessary to apply some solution parameters:

1) Maximum number of FRs in a design graph.
2) Target edge density in a design graph.
3) Acceptable runtime duration to obtain a solution.

Determining solid requirements for the maximum number of FRs and the target edge density would require a full case study by a qualified mechanical engineer; however, the examples submitted by our colleagues at SUNY, Buffalo all have under 20 FRs with low to average edge density. These values seem reasonable: designs with too many FRs may become untenable and hence broken up into multiple layers in the design hierarchy and

designs with too many edges may be too coupled. Although acceptable duration time is subjective, a limit of about 5 minutes will be selected as the goal.

Thus, the primary goal of this research is to provide AD designers with a tool that can determine the minimum number of parts needed to realize a particular design having about 20 FRs and less than 50% edge density in less than 5 minutes. Designs with different minimum part requirements can then be compared during the analytical process phase of the axiomatic design as part of the overall process of selecting the best design. Of course, this tool will be an algorithm that can be run on a computer.

This research compares the two most well-known existing algorithms: the Christofides algorithm [8] with improvements by Wang [9] and the Zykov algorithm [10]. It will be shown that although both algorithms satisfy the stated requirements, the Christofides/Wang algorithm is faster and can handle up to 30 vertices. A new algorithm is then proposed that is a modification of the Zykov algorithm in an attempt to match the advantages of the Christofides/Wang algorithm. It will be shown that this proposed algorithm can also handle up to 30 vertices and runs faster than the Christofides/Wang algorithm for problems in the target range.

2 GRAPH THEORY

This section presents the concepts, definitions, and theorems from the field of graph theory that are needed in the development of the proposed algorithm. This material is primarily taken from the undergraduate graph theory text by Chartrand and Zhang (2012) [11] and the graduate graph theory text by West (2001) [12].

2.1 Simple Graphs

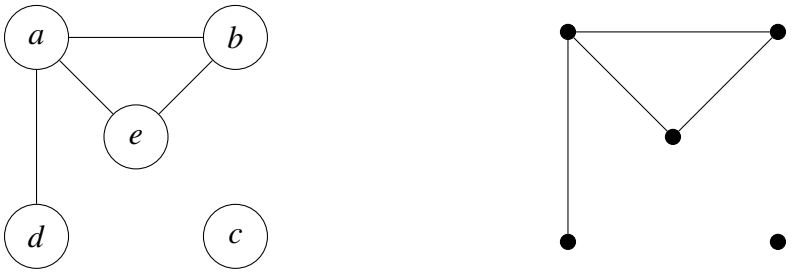
The problem of part consolidation is best served by a class of graphs called *simple graphs*. A *simple graph* is a mathematical object represented by an ordered pair $G = (V, E)$ consisting of a finite and non-empty set of *vertices* (also called *nodes*): $V(G)$, and a finite and possibly empty set of edges: $E(G)$. Each edge is represented by a two-element subset of $V(G)$ called the *endpoints* of the edge: $E(G) \subseteq \mathscr{P}_2(V(G))$. For the remainder of this work, the use of the term "graph" implies a "simple graph." Thus, a part consolidation problem can be represented by a graph whose vertices are the functional requirements (FRs) of the design and whose edges indicate which endpoint FRs should never be combined into a single part.

The choice of two-element subsets of $V(G)$ for the edges has certain ramifications that are indeed characteristics that differentiate a simple graph from other classes of graphs:

1) Every two vertices of a graph are the endpoints of at most one edge; there are no so-called *multiple* edges between two vertices.
2) The two endpoint vertices of an edge are always distinct; there are no so-called *loop* edges on a single vertex.
3) The two endpoint vertices are unordered, suggesting that an edge provides a bidirectional connection between its endpoint vertices.

When referring to the edges in a graph, the common notation of juxtaposition of the vertices will be used instead of the set syntax. Thus, edge $\{u, v\}$ is simply referred to as uv or vu.

Graphs are often portrayed visually using labeled or filled circles for the vertices and lines for the edges such that each edge line is drawn between its two endpoint vertices. An example graph is shown in Fig. 8.



$$V(G) = \{a, b, c, d, e\}$$

$$E(G) = \{ab, ad, ae, be\}$$

Fig. 8. An example graph (labeled and unlabeled).

When two vertices are the endpoints of the same edge, the vertices are said to be *adjacent* or are called *neighbors*, and the edge is said to *join* its two endpoint vertices. Furthermore, an edge is said to be *incident* to its endpoint vertices. In the example graph of Fig. 8, vertex a is adjacent to vertices b, d, and e; however, it is not adjacent to vertex c.

As demonstrated by vertex c in Fig. 8, there is no requirement that every vertex in a graph be an endpoint for some edge. In fact, a vertex that is not incident to any edge is called an *isolated* vertex.

We can also speak of adjacent edges, which are edges that share exactly one endpoint. Note that two edges cannot share both of their endpoints — otherwise they would be multiple edges, which are not allowed in simple graphs. In the example graph of Fig. 8, edge ab is adjacent to edges ad and ae via common vertex a, and be via common vertex b.

2.2 Order and Size

Two of the most important characteristics of a graph are its *order* and its *size*. The *order* of a graph G, denoted by $n(G)$ or just n when G is unambiguous, is the number of vertices in G: $n = |V(G)|$. The *size* of a graph G, denoted by $m(G)$ or just m when G is unambiguous, is the number of edges in G: $m = |E(G)|$. In the example graph of Fig. 8: $n = 5$ and $m = 4$.

Since every two vertices can have at most one edge between them, the number of edges has an upper bound:

Theorem 5. *Let G be a graph of order n and size m:*

$$m \leq \frac{n(n-1)}{2}$$

Proof. Since each pair of distinct vertices in $V(G)$ can have zero or one edges joining them, the maximum number of possible edges is $\binom{n}{2}$, and so:

$$m \leq \binom{n}{2} = \frac{n!}{2!(n-2)!} = \frac{n(n-1)}{2}$$

□

Some choices of graph order and size lead to certain degenerate cases that serve as important termination cases for the the proposed algorithm:

- The *null* graph is the non-graph with no vertices $(n = m = 0)$.
- The *trivial* graph is the graph with exactly one vertex and no edges $(n = 1, m = 0)$. Otherwise $(n > 1)$, a graph is called *non-trivial*.
- An *empty* graph is a graph containing no edges $(m = 0)$.
- A *complete* graph is a graph containing every possible edge $\left(m = \frac{n(n-1)}{2}\right)$.

Note that both the null and trivial graphs are empty.

2.3 Graph Relations

In addition to its vertices and edges, a graph may be associated with one or more relations. Each relation has $V(G)$ or $E(G)$ as its domain and is used to associate vertices or edges with problem-specific attributes such as labels or colors. Note that there are no particular limitations on the nature of such relations — everything from a basic relation to a bijective function are possible. Some authors include these relations and their codomains as part of the graph tuple; however, since these extra tuple elements don't affect the structure of a graph, we will not do so.

In practice, when a graph theory problem requires a particular vertex or edge attribute, the presence of some corresponding relation $\mathscr{R}$ is assumed and we say something like, "vertex v has attribute a," instead of the more formal, "vertex v has attribute $\mathscr{R}(v)$."

The following sections describe the two relations used by the proposed algorithm.

2.3.1 Labels

One possible relation associated with a graph G is a bijective function $\ell : V(G) \to L$ that assigns to each vertex a unique identifying label. The codomain L is the set of available labels. When such a function is present, the graph is said to be a *labeled* graph and the vertices are considered to be distinct. Otherwise, a graph is said to be *unlabeled* and the vertices are considered to be identical (only the structure of the graph matters).

The vertices in a labeled graph are typically drawn as open circles containing the corresponding labels, whereas the vertices in an unlabeled graph are typically drawn as filled circles. This is demonstrated in the example graph of Fig. 8: the graph on the left is labeled and the graph on the right is unlabeled.

Since the labeling function ℓ is bijective, a vertex $v \in V(G)$ with label "a" can be identified by v or $\ell^{-1}(a)$. In practice, the presence of a labeling function is assumed for a labeled graph and so a vertex is freely identified by its label. This is important to note when a proof includes a phrase such as, "let $v \in V(G)\ldots$" since v may be a reference to

any vertex in $V(G)$ or may call out a specific vertex by its label; the intention is usually clear from the context.

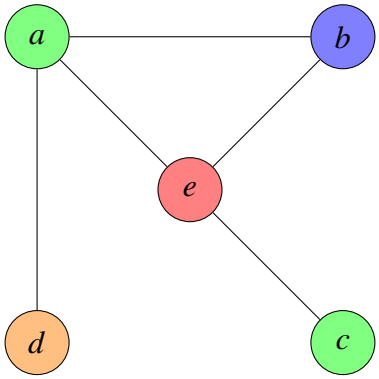
The design graphs that act as the inputs to the proposed algorithm are labeled graphs, where the labels represent the various functional requirements: $\mathrm{FR}_1, \mathrm{FR}_2, \mathrm{FR}_3, \ldots, \mathrm{FR}_n$.

2.3.2 Vertex Color

Other graph theory problems require that a graph's vertices be distributed into some number of sets based on some problem-specific criteria. Usually, this distribution is a true partition (no empty sets), but this is not required depending on the problem. One popular method of performing this distribution on a graph G is by using a *coloring* function $c : V(G) \to C$, where C is a set of *colors*. Vertices with the same color are assigned to the same set in the distribution. Although the elements of C are usually actual colors (red, green, blue, etc.), a graph coloring problem is free to select any value type for the color attribute. Note that there is no assumption that c is surjective, so the codomain C may contain unused colors, which correspond to empty sets in the distribution.

A coloring $c : V(G) \to C$ on a graph G is called *proper* when no two adjacent vertices in G are assigned the same color: for all $u, v \in V(G)$, if $uv \in E(G)$ then $c(u) \neq c(v)$. Otherwise, c is called *improper*. A proper coloring with $|C| = k$ is called a *k-coloring* of G and G is said to be *k-colorable*, meaning the actual coloring (range of c) uses *at most* k colors.

An example of a 4-coloring is shown in Fig. 9.



$$C = \{\text{green}, \text{blue}, \text{red}, \text{orange}\}$$

$$c(a) = \text{green}$$

$$c(b) = \text{blue}$$

$$c(c) = \text{green}$$

$$c(d) = \text{orange}$$

$$c(e) = \text{red}$$

Fig. 9. A graph with a 4-coloring.

Since there is no requirement that a coloring c be surjective, the codomain C may contain unused colors. For example, the coloring shown in Fig. 9 is surjective, but we can add an unused color to C:

$$C = \{\text{green}, \text{blue}, \text{red}, \text{orange}, \text{brown}\}$$

Now, c is no longer surjective, and according to the definition, G is 5-colorable — the coloring c uses at most 5 colors (actually only 4), which is the cardinality of the codomain. This fact is generalized by Theorem 6.

Theorem 6. *Let G be a graph and let $r \in \mathbb{N}$. If G is k-colorable then G is $(k+r)$-colorable.*

Proof. Although this conclusion is fairly intuitive, it is always best to construct a proper coloring function under the given conditions so that the result is based on the definition. So start by assuming that G is of order n and is k-colorable. This means that there exists a coloring function $c : V(G) \to C$ that is proper with $|C| = k$. Let $V(G) = \{v_1, \ldots, v_n\}$ and let $C = \{c_1, \cdots, c_k\}$. Now, let $C' = \{c_1, \cdots, c_{k+r}\}$ and define $c' : V(G) \to C'$ by:

$$c'(v) = c(v)$$

Assume that u and v are two nonadjacent vertices in G: $uv \notin E(G)$. Since c is proper:

$$c'(u) = c(u) \neq c(v) = c'(v)$$

and so c' is proper with $|C'| = k + r$.

Therefore, G is $k+r$-colorable. □

Furthermore, for a graph G of order n, if $n \leq k$ then we can conclude that G is k-colorable, since there are sufficient colors to assign each vertex its own unique color. This result is stated in Theorem 7, which will turn out to be an important termination case for the proposed algorithm.

Theorem 7. *Let G be a graph of order n and let $k \in \mathbb{N}$. If $n \leq k$ then G is k-colorable.*

Proof. Assume $n \leq k$. Let $V(G) = \{v_1, \ldots, v_n\}$ and let $C = \{c_1, \ldots, c_k\}$. Now, define $c : V(G) \to C$ by:

$$c(v_i) = c_i$$

which is possible since, by assumption, $n \leq k$. Finally, assume that v_i and v_j are two nonadjacent vertices in G: $v_i v_j \notin E(G)$. Since the c_i are distinct:

$$c(v_i) = c_i \neq c_j = c(v_j)$$

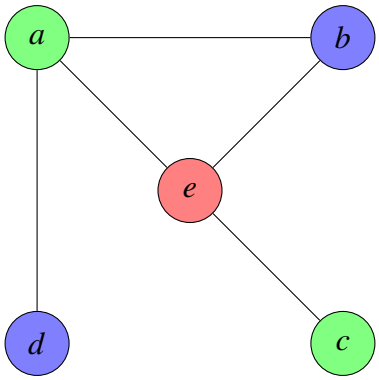
and so c is proper with $|C| = k$.

Therefore, G is k-colorable. □

Since $k \in \mathbb{N}$, by the well-ordering principle there exists some minimum k such that a graph G is k-colorable. This minimum k is called the *chromatic number* of G, denoted by $\chi(G)$. A k-coloring for a graph G where $k = \chi(G)$ is called a *k-chromatic* coloring of G.

Returning to the example 4-coloring of Fig. 9, note that vertex d can be colored blue and then orange can be excluded from the codomain, resulting in a 3-coloring. This is

shown in Fig. 10. Since there is no way to use less than 3 colors to obtain a proper coloring of the graph, the coloring is 3-chromatic. Note that when a coloring is chromatic, there are no unused colors (empty sets) and hence the distribution is a true partition.



$$C = \{\text{green}, \text{blue}, \text{red}\}$$

$$c(a) = \text{green}$$

$$c(b) = \text{blue}$$

$$c(c) = \text{green}$$

$$c(d) = \text{blue}$$

$$c(e) = \text{red}$$

Fig. 10. A Graph with a 3-chromatic coloring.

2.4 Subgraphs

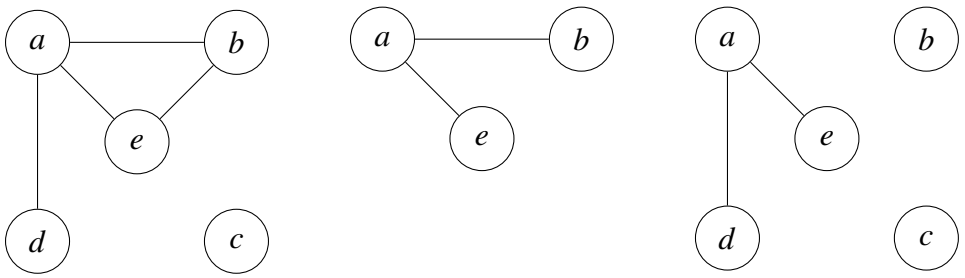
The basic strategy of the proposed algorithm is to arrive at a solution by mutating an input graph into simpler graphs such that a solution is more easily determined. The algorithm utilizes three particular mutators: vertex deletion, edge addition, and vertex contraction. Before describing these mutators, it will be helpful to describe what is meant by graph equality and a *subgraph* of a graph.

To say that graph G is *equal* to graph H, denoted by $G = H$, means that the *exact same graph* is given two names: G and H. It is specifically *not* a comparison between two different graphs. Two different graphs that have the same structure, meaning there exists an adjacency-preserving bijection between the vertices of the two graphs, are referred to as being *isomorphic*, denoted by $G \cong H$, and are not considered to be equal. Of course, if $G = H$ then $G \cong H$; however, the converse is usually not true. In fact, $G = H$ if and only if $V(G) = V(H)$ and $E(G) = E(H)$.

To say that H is a *subgraph* of a graph G, denoted by $H \subseteq G$, means that $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. Thus, H can be achieved by removing zero or more vertices and/or edges from G, and G can be achieved by adding zero or more vertices and/or edges to H. Once again, H is not a different graph. If H is a different graph then one can say that it is isomorphic to a subgraph of G, but not a subgraph of G itself. By definition, $G \subseteq G$ and the null graph is a subgraph of every graph.

When G and H differ by at least one vertex or edge then H is called a *proper subgraph* of G, denoted by $H \subset G$. In fact, $H \subset G$ if and only if $H \subseteq G$ but $H \neq G$, meaning $V(H) \subset V(G)$ or $E(H) \subset E(G)$. When H and G differ by edges only: $V(H) = V(G)$ and $E(H) \subseteq E(G)$, then H is called a *spanning subgraph* of G.

The concept of subgraphs is demonstrated by graphs G, H, and F in Fig. 11. H is a proper subgraph of G by removing vertices c and d and edges ad and be. F is a proper spanning subgraph of G because F contains all of the vertices in G but is missing edges ab and be.

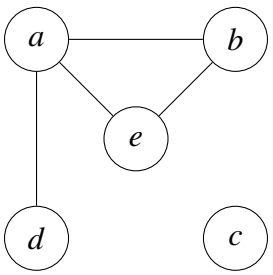


G $H \subset G$ (proper) $F \subset G$ (spanning)

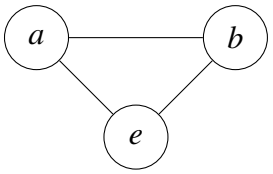
Fig. 11. Subgraph examples.

An *induced* subgraph is a special type of subgraph. Let G be a graph and let $S \subseteq V(G)$. The subgraph of G *induced* by S, denoted by $G[S]$, is a subgraph H such that $V(H) = S$ and for every $u, v \in S$, if u and v are adjacent in G then they are also adjacent in H. Such a subgraph H is called an *induced subgraph* of G.

In the examples of Fig. 11, H is not an induced subgraph of G because it is missing edge be. Likewise, a proper spanning subgraph like F can never be induced due to missing edges. In fact, the only induced spanning subgraph of a graph is the graph itself. Fig. 12 adds edge be so that H is now an induced subgraph of G.



G



$H = G[\{a,b,e\}]$

Fig. 12. An induced subgraph example.

2.5 Mutators

The following sections describe the graph mutators used by the proposed algorithm.

2.5.1 Vertex Removal

Let G be a graph and let $S \subseteq V(G)$. The induced subgraph obtained by removing all of the vertices in S (and their incident edges) is denoted by:

$$G - S = G[V(G) - S]$$

If $S \neq \emptyset$ then $G - S$ is a proper subgraph of G. If $S = V(G)$ then the result is the null graph.

Fig. 13 shows an example of vertex removal: vertices c and e are removed, along with their incident edges ae and be.

G $\qquad$ $G-\{c,e\}$

Fig. 13. A vertex removal example.

If S consists of a single vertex v then the alternate syntax $G-v$ is used instead of $G-\{v\}$.

The proposed algorithm uses vertex removal to simplify a graph that is assumed to be k-colorable into a smaller graph that is also k-colorable.

2.5.2 Edge Addition

Let G be a graph and let $u,v \in V(G)$ such that $uv \notin E(G)$. The graph $G+uv$ is the graph with the same vertices as G and with edge set $E(G) \cup \{uv\}$. Note that G is a proper spanning subgraph of $G+uv$.

Fig. 14 shows an example of edge addition: edge cd is added.

G $\qquad$ $G+cd$

Fig. 14. An edge addition example.

The proposed algorithm uses edge addition to prevent two non-adjacent FRs from being consolidated into the same part.

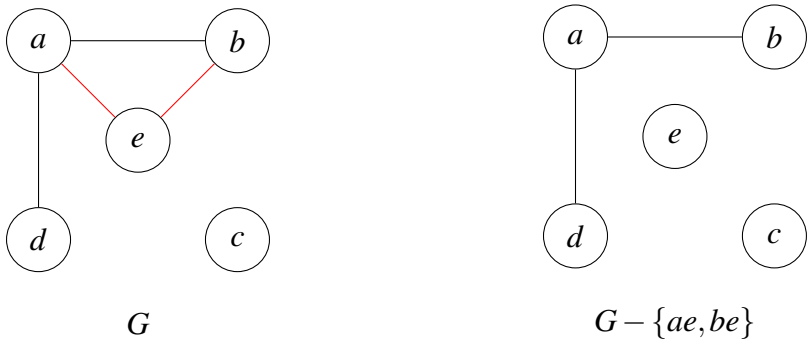
2.5.3 *Edge Removal*

The proposed algorithm does not use edge removal; however, a number of related algorithms do rely on this mutator so it is presented here. Let G be a graph and let $X \subseteq E(G)$. The spanning subgraph obtained by removing all of the edges in X is denoted by:

$$G - X = H(V(G), E(G) - X)$$

Thus, only edges are removed — no vertices are removed. If $X \neq \emptyset$ then $G - X$ is a proper subgraph of G. If $X = E(G)$ then the result is an empty graph.

Fig. 15 shows an example of edge removal: edges *ae* and *be* are removed.



G $\qquad$ $G - \{ae, be\}$

Fig. 15. An Edge removal example.

If X consists of a single edge e then the alternate syntax $G - e$ is used instead of $G - \{e\}$.

2.5.4 *Vertex Contraction*

Vertex contraction is a bit different because it does not involve subgraphs. Let G be a graph and let $u, v \in V(G)$. The graph $G \cdot uv$ is constructed by identifying u and v as one vertex (i.e., merging them). Any edge between the two vertices is discarded. Any other edges that were incident to the two vertices become incident to the new single vertex. Note that this may require supression of multiple edges to preserve the nature of a simple graph.

Fig. 16 shows an example of vertex contraction: vertices a and b are contracted into a single vertex. Since edges ae and be would result in multiple edges between a and e, one of the edges is discarded. Edges bc and bd also become incident to the single vertex.

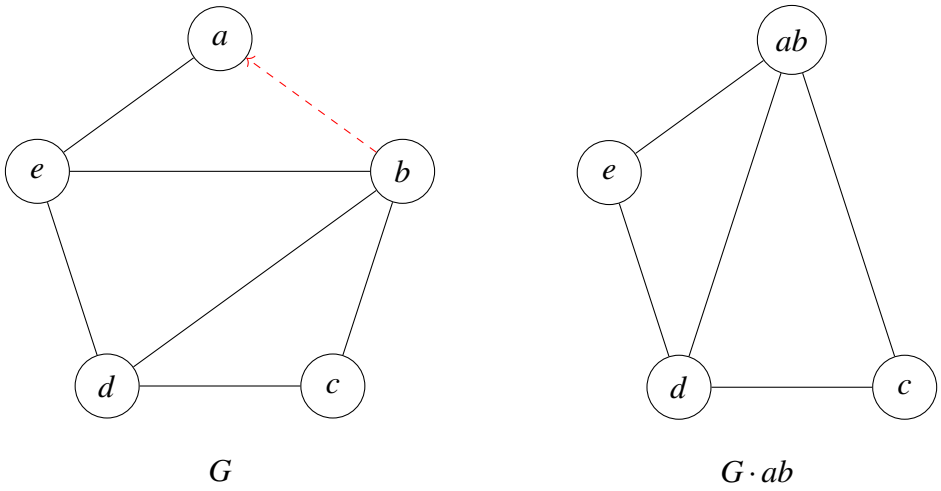


Fig. 16. A vertex contraction example.

For the operation $G \cdot uv$, if $uv \in E(G)$ then the operation is also referred to as *edge contraction*. If $uv \notin E(G)$ then the operation is also referred to as *vertex identification*. The proposed algorithm uses vertex identification to consolidate two non-adjacent FRs into the same part.

2.5.5 Graph Complement

One final important graph mutator is the *complement* of a graph. For a graph G, the *complement* of G, denoted by $\bar{G}$, is the graph with the same vertex set as G: $V(G) = V(\bar{G})$, and with edge set $E(\bar{G}) = \mathscr{P}_2(V(G)) - E(G)$; if $u, v \in V(G)$ are adjacent in G $(uv \in E(G))$ then they are not adjacent in $\bar{G}$ $\left(uv \notin E(\bar{G})\right)$.

An example of a graph complement operation is shown in Fig. 17.

G $\bar{G}$

Fig. 17. A graph complement example.

Some important properties of the complement of a graph are stated in Proposition 1.

Proposition 1. *Let G be a graph of order n and size m:*

1) $\bar{\bar{G}} = G$

2) *G is empty if and only if $\bar{G}$ is complete.*

3) $n(\bar{G}) = n(G)$

4) $m(\bar{G}) = \frac{n(n-1)}{2} - m(G)$

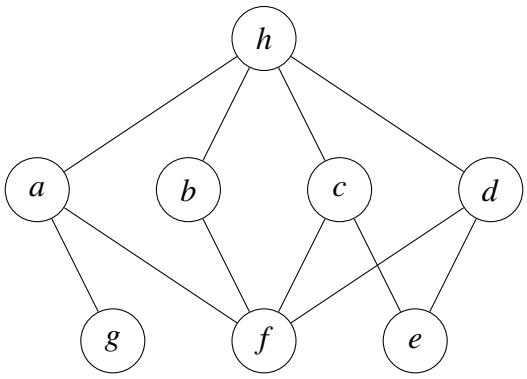
For example, in Fig. 17, since $n(G) = 4$ and $m(G) = 3$, it is the case that $n(\bar{G}) = 4$ and:

$$m(\bar{G}) = \frac{4(4-1)}{2} - 3 = 6 - 3 = 3$$

2.6 Independent Sets

The primary purpose of a k-coloring of a graph G is to distribute the vertices of G into k so-called *independent* (some possibly empty) sets. For a graph G, an *independent* set $S \subseteq V(G)$, sometimes called a *stable* set, is a set of pairwise non-adjacent vertices in G: for all $u, v \in S$, $uv \notin E(G)$. By definition, the empty set is an independent set of every graph G. A *maximal* independent set of a graph G, sometimes referred to as a *MIS* of G, is an independent set of G that cannot be extended by an additional vertex in $V(G)$; MISs of G are never proper subsets of other independent sets in G. The cardinality of the largest possible MIS in a graph G, denoted by $\alpha(G)$, is called the *independence number* for G.

Consider the example in Fig. 18. Although the graph contains independent sets of sizes 1 and 2, none of these are maximal. All of the maximal independent sets are of sizes 3 and 4, and so $\alpha(G) = 4$.



MIS	SIZE
$\{a,b,c,d\}$	4
$\{a,b,e\}$	3
$\{b,c,d,g\}$	4
$\{b,e,g\}$	3
$\{e,f,g,h\}$	4

$\alpha(G) = 4$

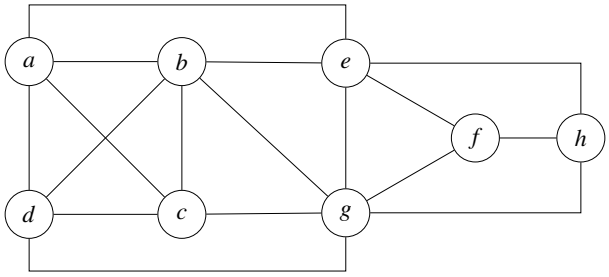
Fig. 18. The independent sets of an example graph.

Since a k-chromatic coloring of a graph G is surjective, there are no unused colors (empty sets) and so the coloring partitions the vertices of G into exactly k non-empty independent sets. The goal of the proposed algorithm is to find a chromatic coloring of a design graph so that the resulting independent sets indicate how to consolidate the FRs into a minimum number of parts: one part per independent set.

2.7 Cliques

A *clique* is a complete subgraph of a graph. A clique of order k in a graph G is called a k-clique of G. A *maximal* clique in a graph G is a clique in G that cannot be extended by an additional vertex in $V(G)$; maximal cliques in G are never proper subgraphs of other cliques in G. The order of the largest possible maximal clique in a graph G, denoted by $\omega(G)$, is called the *clique number* for G.

Consider the example in Fig. 19. Although the graph contains cliques of orders 1 and 2, none of these are maximal. All of the maximal cliques are of orders 3 and 4, and so $\omega(G) = 4$.



G

MAXIMAL CLIQUE	ORDER
$G[\{a,b,c,d\}]$	4
$G[\{a,b,e\}]$	3
$G[\{b,c,d,g\}]$	4
$G[\{b,e,g\}]$	3
$G[\{e,f,g,h\}]$	4

$\omega(G) = 4$

Fig. 19. The maximal cliques of an example graph.

Since it is true that two nonadjacent vertices in a graph must be adjacent in the graph's complement, there is an important relationship between the independent sets of a graph and the cliques in the graph's complement. This relationship is stated in Theorem 8.

Theorem 8. *Let G be a graph and let $S \subseteq V(G)$. S is an independent set in G if and only if $\bar{G}[S]$ is a clique in $\bar{G}$. Furthermore, S is maximal in G if and only if $\bar{G}[S]$ is maximal in $\bar{G}$ and so $\alpha(G) = \omega(\bar{G})$.*

Proof. By definition, for all $u, v \in V(G)$, u is not adjacent to v in G if and only if u and v are adjacent in $\bar{G}$, and so $G[S]$ is empty if and only if $\bar{G}[S]$ is complete. Therefore, S is an independent set in G if and only if $\bar{G}[S]$ is a clique in $\bar{G}$.

Furthermore, assume that S is maximal in G but assume by way of contradiction that $\bar{G}[S]$ is not maximal. Then there exists $v \in V(\bar{G})$ such that $v \notin S$ and $\bar{G}[S \cup \{v\}]$ is a clique in $\bar{G}$, and thus $S \cup \{v\}$ is an independent set in G. But $S \subset S \cup \{v\}$, violating the maximality of S. Therefore $\bar{G}[S]$ is maximal in $\bar{G}$.

Similarly, assume that $\bar{G}[S]$ is maximal in $\bar{G}$ but assume by way of contradiction that S is not maximal in G. Then there exists $v \in V(G)$ such that $v \notin S$ and $S \cup \{v\}$ is an independent set in G, and thus $\bar{G}[S \cup \{v\}]$ is a clique in $\bar{G}$. But $\bar{G}[S] \subset \bar{G}[S \cup \{v\}]$, violating the maximality of $\bar{G}[S]$. Therefore S is maximal in G. □

Indeed, the graphs in Fig. 18 and Fig. 19 are complements and, as expected, every MIS in Fig. 18 is a maximal clique in Fig. 19.

Since a k-clique of a graph G needs at least k colors in any proper coloring of G, the clique number of G provides a nice lower bound for the chromatic number of G. Unfortunately, the clique number problem is known to be inherently intractable as well [10]. Thus, there are many attempts in the literature to find a good lower bound for the clique number of a graph G, usually denoted by $\omega'(G)$. If such a lower bound is known then the conclusion of Proposition 2 holds:

Proposition 2. *Let G be a graph with clique number lower bound $\omega'(G)$:*

$$\omega'(G) \leq \omega(G) \leq \chi(G)$$

2.8 Connected Graphs

The edges of a graph suggest the ability to "walk" from one vertex to another along the edges. A graph where this is possible for any two vertices is called a *connected* graph. The concept of connectedness is an important topic in graph theory; however, an ideal coloring algorithm should work regardless of the connected nature of an input graph. The concept of connectedness and how it impacts coloring is described in this section.

2.8.1 Walks

The undirected edges in a simple graph suggest bidirectional connectivity between their endpoint vertices. This leads to the idea of "traveling" between two vertices in a graph by following the edges joining intermediate adjacent vertices. Such a journey is referred to as a *walk*.

A $u-v$ *walk* W in a graph G is a finite sequence of vertices $w_i \in V(G)$ starting with $u = w_0$ and ending with $v = w_k$:

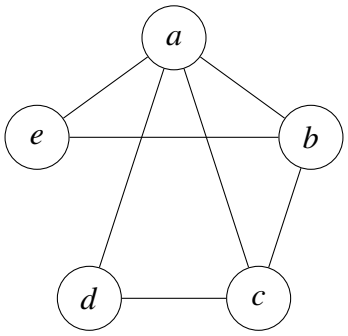
$$W = (u = w_0, w_1, \ldots, w_k = v)$$

such that $w_i w_{i+1} \in E(G)$ for $0 \leq i < k$. To say that W is *open* means that $u \neq v$. To say that W is *closed* means that $u = v$. The *length* k of W is the number of edges traversed: $k = |W|$. A *trivial* walk is a walk of zero length — i.e, a single vertex: $W = (u)$.

The bidirectional nature of the edges in a simple graph suggests the following proposition:

Proposition 3. *Let G be a graph and let $u - v$ be a walk of length k in G. G contains a $v - u$ walk of length k in G by traversing $u - v$ in the opposite direction.*

An example of two walks of length 4 is shown in Fig. 20. W_1 is an open walk because it starts and ends on distinct vertices, whereas W_2 is a closed walk because it starts and ends on the same vertex.



$W_1 = (a, b, e, a, c)$ is open

$W_2 = (a, e, b, c, a)$ is closed

$|W_1| = |W_2| = 4$

Fig. 20. Open and closed walks.

Vertices and edges are allowed to be repeated during a walk. Certain special walks can be defined by restricting such repeats:

trail	An open walk with no repeating edges	(a, b, c, a, e)
path	A trail with no repeating vertices	(a, e, b, c)
circuit	A closed trail	(a, b, e, a, c, d, a)
cycle	A closed path	(a, e, b, c, a)

The example special walks stated above refer to the graph in Fig. 20.

2.8.2 Paths

When discussing the connectedness of a graph, the main concern is the existence of paths between vertices. Let G be a graph and let $u, v \in V(G)$. To say that u and v are *connected* means that G contains a $u-v$ path.

But if there exists a $u-v$ walk in a graph G, does this also mean that there exists a $u-v$ path in G — i.e., a walk with no repeating edges or vertices? The answer is yes, as shown by the following theorem:

Theorem 9. *Let G be a graph and let $u, v \in V(G)$. If G contains a $u-v$ walk of length k then G contains a $u-v$ path of length $\ell \leq k$.*

Proof. Assume that G contains at least one $u-v$ walk of length k and consider the set of all possible $u-v$ walks in G; their lengths form a non-empty set of positive integers. By the well-ordering principle, there exists a $u-v$ walk P of minimum length $\ell \leq k$:

$$P = (u = w_0, \ldots, w_\ell = v)$$

We claim that P is a path.

Assume by way of contradiction that P is not a path, and thus P has at least one repeating vertex. Let $w_i = w_j$ for some $0 \leq i < j \leq \ell$ be such a repeating vertex. There are two possibilities:

Case 1: The walk ends on a repeated vertex ($j = \ell$). This is demonstrated in Fig. 21.

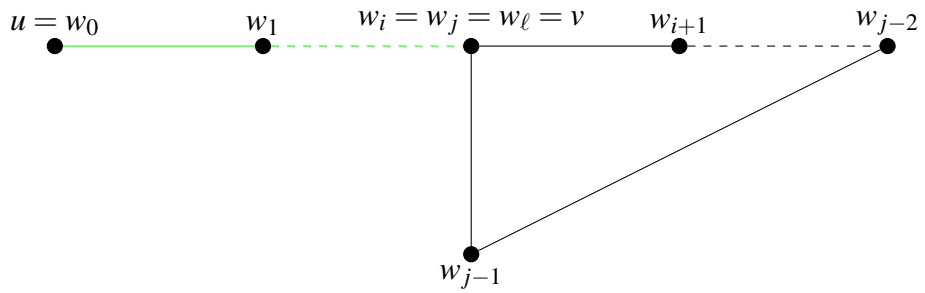


Fig. 21. Repeated vertex at end case.

Let $P' = (u = w_0, w_1, \ldots, w_i = v)$ be the walk shown in green in Fig. 21. P' is a $u - v$ walk of length $i < \ell$ in G.

Case 2: A repeated vertex occurs inside the walk ($j < \ell$). This is demonstrated in Fig. 22.

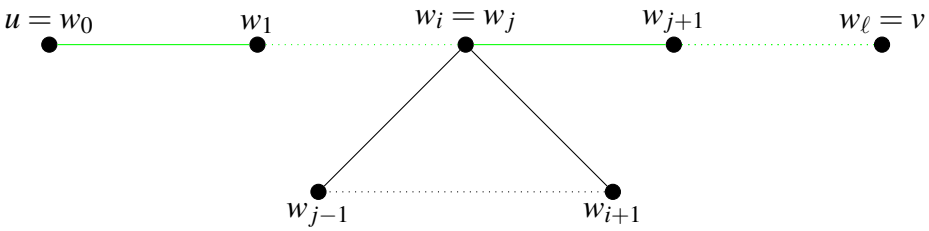


Fig. 22. Repeated vertex inside case.

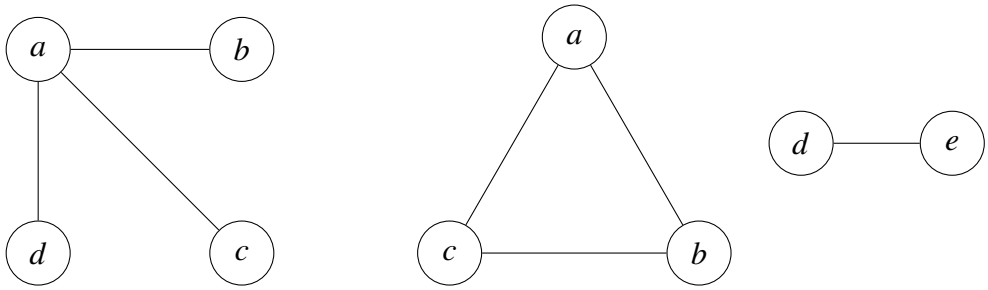
Let $P' = (u = w_0, w_1, \ldots, w_i, w_{j+1}, \ldots, w_\ell = v)$ be the walk shown in green in the Fig. 22. P' is a $u - v$ walk of length $\ell - (j - i) < \ell$ in G.

Both cases contradict the minimality of the length of P.

$\therefore P$ is a $u - v$ path of length $\ell \leq k$ in G. □

2.8.3 Connected

A *connected* graph G is a graph whose vertices are all connected: for all $u, v \in V(G)$ there exists a $u - v$ path. Otherwise, G is said to be *disconnected*. Examples of connected and disconnected graphs are shown in figure 23.



(a,b)
(a,c)
(a,d)
(b,a,c)
(b,a,d)
(c,a,d)

CONNECTED

No path from any of a,b,c to any of d,e

DISCONNECTED

Fig. 23. Connected and disconnected graphs.

By definition, the trivial graph is connected since the single vertex is connected to itself by a trivial path (of length 0).

2.8.4 Components

It would seem that a disconnected graph is composed of some number of connected subgraphs that partition the graph's vertex set under a connected equivalence relation. Each such subgraph is called a *component* of the graph.

Let G be a graph and let $\mathscr{G}$ be the set of all connected subgraphs of G. To say that a graph $H \in \mathscr{G}$ is a *component* of a G means that H is not a subgraph of any other connected subgraph of $\mathscr{G}$: for every $F \in \mathscr{G} - \{H\}$ it is the case that $H \not\subset F$. The number of distinct components in G is denoted by k(G), or just k if G is unambiguous. For a connected graph: $k(G) = 1$.

Each component of a graph G is denoted by G_i where $1 \leq i \leq k(G)$. We also use union notation to denote that G is composed of its component parts:

$$G = \bigcup_{0 \leq i \leq k(G)} G_i$$

Furthermore the G_i are induced by the vertex equivalence classes of the connectedness relation:

Theorem 10. *Let G be a graph with component G_i. G_i is an induced subgraph of G.*

Proof. By definition, G_i is a maximal connected subgraph of G. So assume by way of contradiction that G_i is not an induced subgraph of G. Thus, G_i is missing some edges that when added would result in a connected induced subgraph H of G. But then $G_i \subset H$, contradicting the maximality of G_i.

$\therefore G_i$ is an induced subgraph of G. □

2.8.5 Impact on Coloring

The impact of disconnectedness on coloring depends on the selected algorithm. One might assume that the selected algorithm should be run on each component individually in order to determine each $\chi(G_i)$ and then, as pointed out by Zykov (1949) [13], conclude that the maximum such value is sufficient for $\chi(G)$:

$$\chi(G) = \max_{1 \leq i \leq k(G)} \chi(G_i)$$

For example, consider the disconnected graph in Fig. 23. The graph contains two components, so number the components from left-to-right:

$$\chi(G_1) = 3$$

$$\chi(G_2) = 2$$

$$\chi(G) = \max\{3,2\} = 3$$

Using this technique requires application of an initial algorithm to partition the graph into components. Such an algorithm is well-known and is described by Hopcroft and Tarjan (1973) [14]. The algorithm is recursive. It starts by pushing a randomly selected

vertex on the stack and walking the vertex's incident edges, removing each edge as it is traversed. As each unmarked vertex is encountered, it is assigned to the current component. Vertices with incident edges are pushed onto the stack and newly isolated vertices are popped off the stack. Once the stack is empty, any previously unmarked vertex is selected to start the next component and the process continues until all vertices are marked. Given a graph G of order n and size m, this algorithm runs in $\max(n,m)$ steps.

Alternatively, an ideal coloring algorithm could be run on the entire graph at once regardless of the number of components in the graph. The proposed algorithm is such a solution, and therefore saves the needless work of partitioning the graph into components first.

2.9 Vertex Degree

Besides a graph's order and size, the next most important parameter is the so-called *degree* of each vertex. In order to define the degree of a vertex, we need to define what is meant by a vertex's *neighborhood* first. Let G be a graph and let $u \in V(G)$. If $v \in V(G)$ is adjacent to u then u and v are called *neighbors*. Note that for simple graphs, a vertex is never a neighbor of itself. The *neighborhood* of u, denoted by $N(u)$, is the set of all the neighbors of u in G:

$$N(u) = \{v \in V(G) \,|\, uv \in E(G)\}$$

The *degree* of u, denoted by $\deg_G(u)$ or just $\deg(u)$ if G is unambiguous, is then defined to be the cardinality of its neighborhood: $\deg(u) = |N(u)|$. Thus, the degree of a vertex can be viewed as the number of neighbor vertices or the number of incident edges.

When considering the degrees of all the vertices in a graph, the following limits are helpful:

$$\delta(G) = \min_{v \in V(G)} \deg(v)$$

$$\Delta(G) = \max_{v \in V(G)} \deg(v)$$

Therefore, we can state the conclusion of Proposition 4:

Proposition 4. *Let G be a graph of order n. For every vertex $v \in G$:*

$$0 \leq \delta(G) \leq \deg(v) \leq \Delta(G) \leq n-1$$

Intuitively, as $\delta(G)$ increases, a graph becomes denser (more edges) resulting in more adjacencies, making it harder to find a proper coloring at lower values of k.

Vertices can be classified based on their degree, as shown in Table 2.

Table 2

Classifying Vertex u in Graph G

$\deg(u)$	TYPE
0	isolated
1	pendant, end, leaf
$n-1$	universal
even	even
odd	odd

Isolated vertices have degree 0; they are not adjacent to any other vertex in G. *Pendant* (also called *end* or *leaf*) vertices have degree 1; they are adjacent to exactly one other vertex in G. *Universal* vertices are adjacent to every other vertex in G. *Even* vertices are adjacent to an even number of vertices in G and *odd* vertices are adjacent to an odd number of vertices in G. Note that if G has a universal vertex then it cannot have an isolated vertex, and vice-versa.

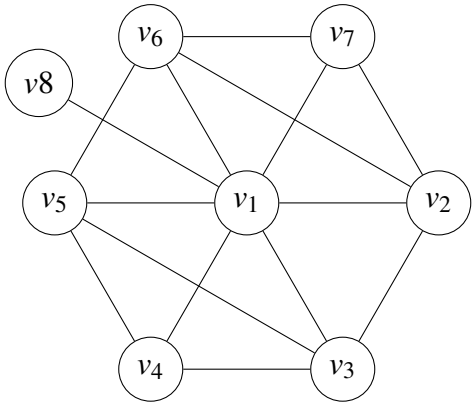
The degrees of the vertices in a graph and the number of edges in the graph are related by the so-called First Theorem of Graph Theory:

Theorem 11 (First Theorem of Graph Theory)**.** *Let G be a graph of size m:*

$$\sum_{v \in V(G)} \deg(v) = 2m$$

Proof. When summing all the degrees, each edge is counted twice: once for each endpoint. □

These concepts are demonstrated by the graph in Fig. 24.



vertex	degree	type
v_1	7	universal,odd
v_2	4	even
v_3	4	even
v_4	3	odd
v_5	4	even
v_6	4	even
v_7	3	odd
v_8	1	pendant,odd
total	30	

$n = 8 \qquad m = 15 = \frac{30}{2}$
$\delta(G) = 1 \quad \Delta(G) = 7$

Fig. 24. Vertex degrees and the first theorem of graph theory.

2.10 Special Graphs

The following sections described some special classes of graphs that are important to the execution of the proposed algorithm.

2.10.1 Empty Graphs

An *empty* graph of order n, denoted by E_n, is a graph with one or more vertices ($n > 1$) and no edges ($m = 0$). An empty graph is connected if and only if $n = 1$. Examples of empty graphs are shown in Fig. 25.

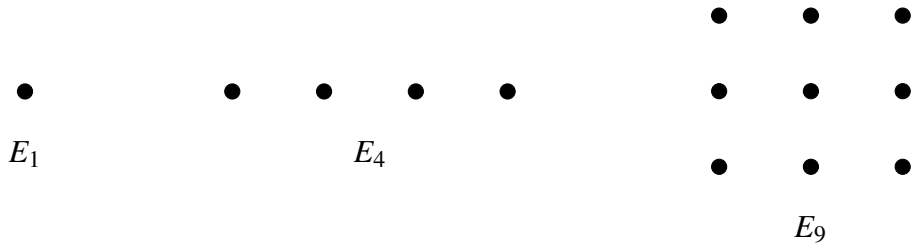


Fig. 25. Empty graphs.

The null graph ($n = 0$) is denoted by E_0 and is defined to be 0-chromatic. All other empty graphs are 1-chromatic and thus are important termination conditions for the proposed algorithm.

2.10.2 Paths

A *path* graph of order n and length $n-1$, denoted by P_n, is a connected graph consisting of a single open path. Examples of path graphs are shown in Fig. 26.

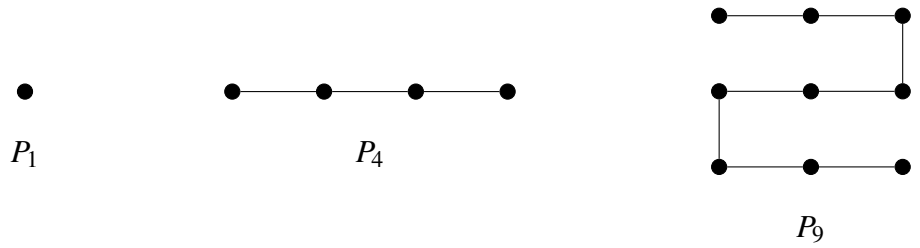


Fig. 26. Path graphs.

Note that $P_1 = E_1$ is 1-chromatic, whereas $P_{n>1}$ is 2-chromatic.

Paths are not particularly important to the proposed algorithm; however, they are used in the definition of cycles.

2.10.3 Cycles

A *cycle* graph of order n and length n for $n \geq 3$, denoted by C_n, is a connected graph consisting of a single closed path. When n is odd then C_n is called an *odd* cycle and when n is even then C_n is called an *even* cycle.

Examples of cycle graphs are shown in Fig. 27.

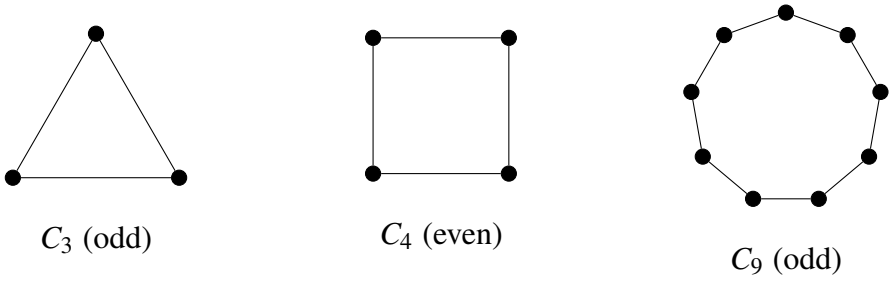


Fig. 27. Cycle graphs.

Note that even cycles are 2-chromatic; however, odd cycles are 3-chromatic.

Cycles are not particularly important to the proposed algorithm; however, they are used in the definition of trees, which are important to the later analysis of coloring algorithms.

2.10.4 Complete Graphs

A *complete* graph of order n and size $\frac{n(n-1)}{2}$, denoted by K_n, is a connected graph that contains every possible edge: $E(G) = \mathscr{P}_2(V(G))$. Thus, all of the vertices in a complete graph are universal.

Examples of complete graphs are shown in Fig. 28.

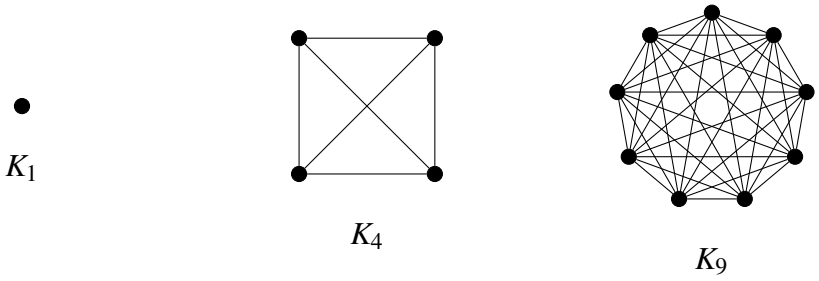


Fig. 28. Complete graphs.

Note that $K_1 = P_1 = E_1$.

Since all of the vertices in a complete graph are adjacent to each other, each vertex requires a separate color in order to achieve a proper coloring. Thus, K_n is n-chromatic and is also an important termination condition for the proposed algorithm.

2.10.5 Trees

A *tree* is a connected graph that contains no cycles as subgraphs. Typically, one vertex of the tree is selected as the *root* vertex and then the tree is depicted in layers that contain vertices that are equidistant from the root vertex. Thus, the bottom layer is composed entirely of pendant vertices, but pendant vertices can exist in the other layers as well. Such pendant vertices are usually referred to as *leaves* in this context.

An example tree is shown in Fig. 29. The root vertex r is shown in red and the leaf vertices b, e, g, h, i, j, k are shown in green.

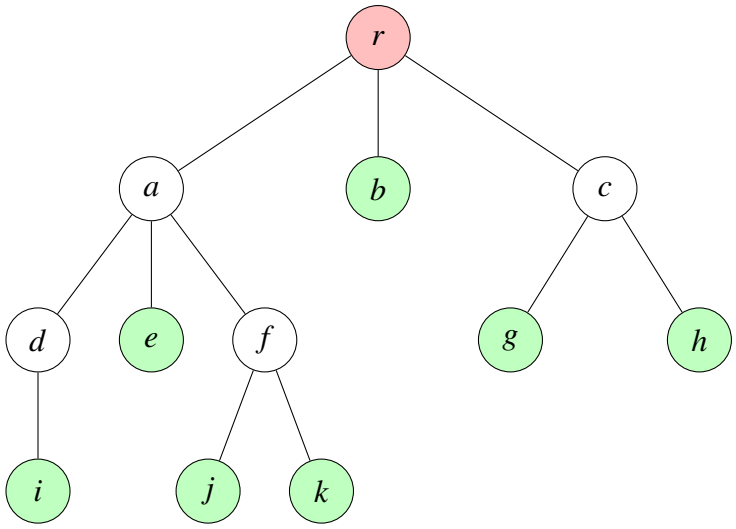


Fig. 29. A tree organized from root to leaves.

Trees are important because they can be used to track so-called "branch-and-bound" algorithms; each vertex represents a branch choice of the algorithm and thus a particular state of the problem. All such states can be visited using a so-called *depth-first* walk. In the example in Fig. 29, such a depth-first walk would be:

$$(r, a, d, i, d, a, e, a, f, j, f, k, f, a, r, b, r, c, g, c, h, c, r)$$

Note that this walk guarantees that each vertex is visited at least once.

When such a tree is applied to the problem of exhaustively finding the chromatic number of a graph via a sequence of vertex contraction and edge addition choices, the tree is called a *Zykov* tree and the algorithm is called a *Zykov* algorithm [15]. Zykov algorithms are described in detail in Section 4.5. In fact, the proposed algorithm is a variation of the standard Zykov algorithm.

2.11 The Adjacency Matrix

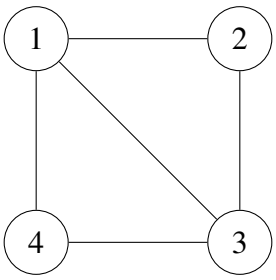
For a graph G of order n, the adjacency matrix A for G is the $n \times n$ matrix such that:

$$a_{ij} = \begin{cases} 0, & v_i v_j \notin E(G) \\ 1, & v_i v_j \in E(G) \end{cases}$$

In the case of a simple graph:

1) The a_{ij} values are limited to 0 and 1 in order to avoid multiple edges.
2) The diagonal values a_{ii} are always 0 in order to avoid loops.
3) A is symmetric due to the bidirectional nature of the edges.

An example graph and its adjacency matrix are shown in Fig. 30.



$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$

Fig. 30. A graph and its adjacency matrix.

The degree of vertex v_i can be calculated by summing the i^{th} row or the i^{th} column:

$$\deg(v_i) = \sum_{k=1}^{n} a_{ik} = \sum_{k=1}^{n} a_{ki}$$

The minimum ($\delta(G)$) and maximum ($\Delta(G)$) degree values can then be calculated by selecting the minimum and maximum calculated degree values.

The adjacency matrix is extremely important to graph algorithms since it provides an instant report of vertex adjacency. Furthermore, as the adjacency matrix is being constructed for a graph, it is easy to calculate each vertex degree as well as the minimum and maximum degree and cache these values for later use.

3 PROBLEMS AND ALGORITHMS

The chromatic number problem is *inherently intractable*. Informally, this means that finding a solution for a given input graph, regardless of the means, can take a very, very long time in the worst cases. Thus, trying to find an efficient *and* exact method of solution that satisfies all cases is a fool's errand; about the best that can be done is to perform better than other well-known methods in most cases.

Computation theory is the branch of computer science that is concerned with determining and comparing the runtime performance of algorithms. The history and specifics of computation theory, although interesting, are beyond the scope of this research. Instead, this section presents a brief overview of what is needed from the field of computation theory in order to characterize the chromatic number problem. Most of this material is based on the early yet still very influential text by Garey and Johnson (1979) [7] with some help from Spiser (2013) [16]. The material on incremental algorithm development is highly influenced by Johnston (1976) [17].

3.1 Problems

A *problem* consists of three parts:

1) A specific question to be answered.
2) A description of zero or more input *parameters*.
3) A statement of the properties that the *solution* is required to satisfy.

An *instance* of a problem is constructed by specifying particular values for each input parameter. The step-by-step procedure that translates the input parameters to a corresponding well-defined solution is called an *algorithm*. To say that an algorithm *solves* a problem means that the algorithm produces a valid solution for every possible instance of the problem.

The chromatic number problem accepts a graph G and uses an algorithm to obtain a number $k \in \mathbb{N}$ where k is the minimum value such that G is k-colorable. The proposed

algorithm is one such algorithm that can be used to solve the chromatic number problem, as are the well-known Christofides and Zykov algorithms.

3.2 Comparing Algorithms

Algorithms are compared using three parameters:

1) Runtime complexity
2) Space complexity
3) Runtime duration

These parameters are discussed in the following sections.

3.2.1 Runtime Complexity

Runtime complexity measures the number of *steps* required to obtain a solution for the worst possible input parameter and is a function of some *length* parameter of the problem. For graph algorithms, the length parameter is usually the order of the graph, although size and structure can also contribute to the worst case.

The runtime complexity of an algorithm is stated using the so-called *big-*$\mathcal{O}$ notation: to say that an algorithm has $\mathcal{O}(f(n))$ runtime complexity means that the maximum number of steps N required to obtain a solution for a given length parameter n has an upper bound of $cf(n)$ for some real number $c > 0$; N is asymptotic to $f(n)$ as $n \to \infty$. Roughly speaking, tractable problems are those problems with polynomial $\mathcal{O}(n^c)$ or better runtime complexity for some real number constant $c \geq 0$, and intractable problems are those problems with exponential $\mathcal{O}(c^n)$ or worse runtime complexity for some real number constant $c > 1$.

What constitutes a *step* in an algorithm is relative to the length parameter and the overall runtime complexity of the algorithm. One of the problems with big-*BO* notation is that it is geared towards large n where the effects of smaller steps are diminished.

Consider an algorithm that has an exponential number of steps 2^{an} and for each of those steps it must execute a step with n^c steps. The total number of steps would then be:

$$n^c 2^{an} = 2^{\log(n^c)} 2^{an} = 2^{an + c\log(n)}$$

For the types of algorithms examined in this research, a typical value for c would be no more than 3 and a typical value for a would be no more than 1. Table 3 lists results for various values of n. Note that for higher values of n the effect of the polynomial time steps diminishes; however, for low to moderate values of n the effect is significant. To put it bluntly, AD designers don't give a hoot about the runtime complexity at large n; they only care about how long it takes to get an answer for values of n in the stated range of the tool. Therefore, for the selected range of about 20 FRs, the effects of these steps can be significant.

Table 3

Comparing Runtime Complexity for an Exponential/Polynomial Mix

n	$n+3\log(n)$
10	20
15	27
20	33
25	39
30	45
100	120
1000	1030
10000	10040

Runtime complexity is used in two different ways to compare algorithms:

1) *Finding* a solution to a problem given a particular input parameter.
2) *Verifying* that a given solution is in fact a solution for a given input parameter.

These two comparisons can be very different. For example, finding a k-clique in a graph G for a particular value of k has exponential runtime complexity; however, verifying whether or not a given subgraph of a graph is a k-clique has polynomial runtime complexity. Because of these differences, algorithms are categorized into the computation classes shown in Table 4.

Table 4

Runtime Complexity Classes for Algorithms

P	Algorithms with polynomial or better runtime complexity to find or verify a solution.
NP	A superset of P with varying runtime complexity to find a solution but polynomial runtime complexity to verify a solution. It is an open question as to whether P=NP; however, it is conjectured that they are not equal.
NP-complete	A subset of NP problems that have been proven to have the same runtime complexity to find a solution.
NP-hard	Algorithms that have been proved to have the same runtime complexity as the NP-complete problems to find a solution but varying runtime complexity to verify a solution.

The relationships between these runtime complexity classes, assuming $P \neq NP$, is shown in Fig. 31.

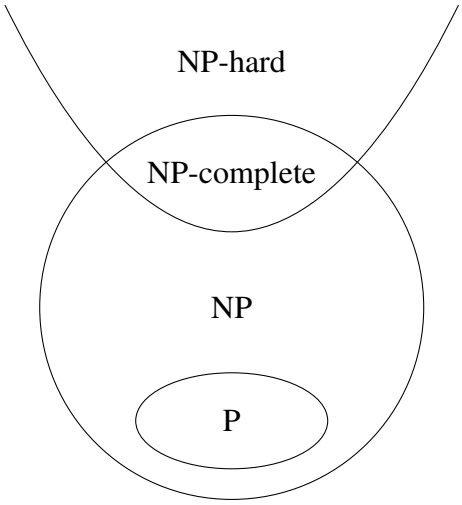


Fig. 31. The runtime complexity classes.

For the purposes of this research, the NP-complete problems are assumed to have exponential runtime complexity to find a solution and polynomial runtime complexity to verify a solution, and the NP-hard problems are assumed to have exponential runtime for both finding and verifying solutions.

The chromatic number problem is NP-hard [15]: it requires exponential time to exhaustively generate and check all possible independent set partitions to find a partition with the smallest number of independent sets k, and the same basic procedure must be used to verify that given a supposed k-chromatic coloring, there does not exist a proper coloring for smaller k.

3.2.2 Space Complexity

Space complexity measures the maximum amount of memory required at any point in time when an algorithm is run on a computer. The limited memory and CPU power in early computers forced algorithm designers and programmers to make careful tradeoffs between CPU cycles and the storage of intermediate results. With today's fast CPUs and practically unlimited virtual memory systems, such concerns are not as important. Thus,

space complexity will not be considered when comparing the three algorithms, except in a few limited cases where it may be an issue.

3.2.3 Runtime Duration

Runtime duration is an empirical measurement of how long an algorithm runs on a given computer, usually on best, average, and worst-case input parameter values. Determining the runtime complexity for some algorithms can be very complicated when the number of possible steps is dependent on the peculiarities of the input parameters. In the case of graph algorithms, such things as order, size, and edge density can all affect the number of steps. Furthermore, runtime complexity is geared towards theoretical comparisons between algorithms at large n, not actual runtime of an algorithm implementation on a given computer for a particular range of n. A comparison of runtime durations for the three algorithms for the selected parameter ranges is presented in Section 6.

3.3 Branch-and-Bound Algorithms

Exponential problems are usually associated with so-called *brute-force* algorithms that must examine all possibilities from an exponentially increasing set of candidate solutions in order to find the desired solution. The states of a brute-force algorithm can be represented by nodes in a tree. Each leaf node of the tree represents a candidate solution. Each non-leaf node represents a partial solution and serves as the root node of a subtree leading to a set of related candidate solutions. Such an algorithm is called a *branching* algorithm because each candidate solution can be found by walking a unique path through the tree starting at the root node and ending at the candidate solution leaf node.

For example, consider the problem of finding all maximal cliques in the graph shown in Fig.32.

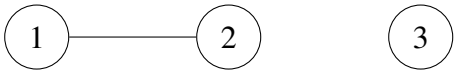


Fig. 32. Finding maximal cliques example graph.

First, consider the rather naïve solution of examining every subgraph in the graph. The resulting tree is shown in Fig.33. Each node has two branches: include the next vertex and exclude the next vertex. The results are summarized in the last row: green marks the desired maximal cliques, blue marks non-maximal cliques, and red marks non-cliques.

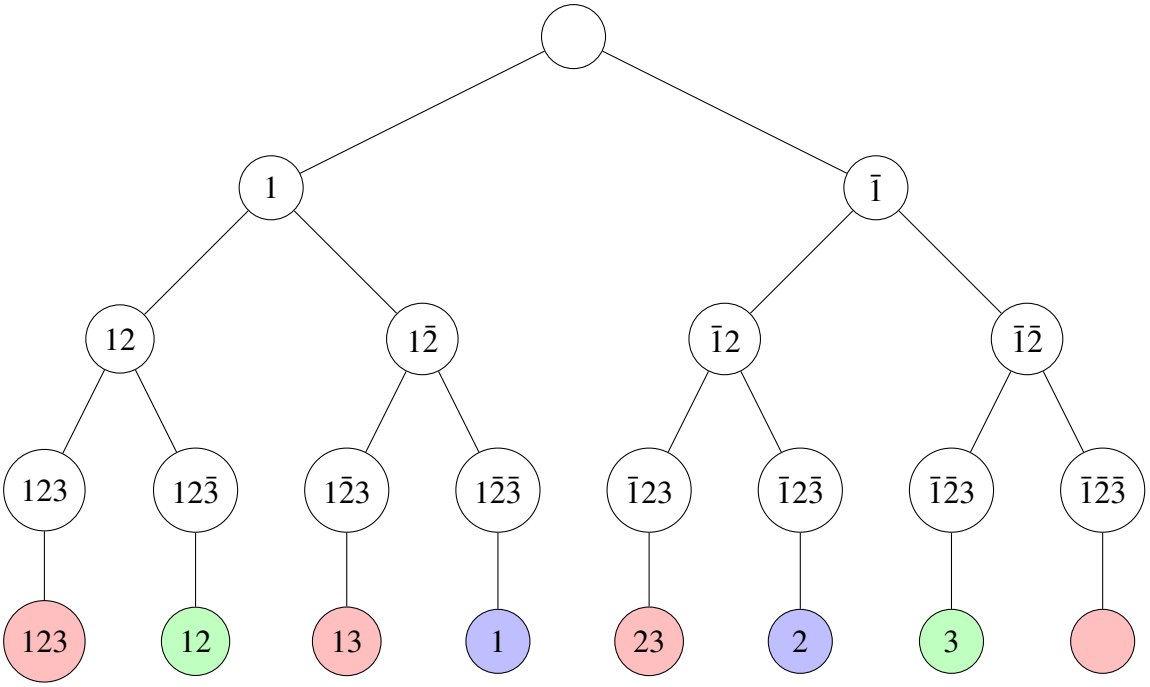


Fig. 33. Finding maximal cliques exhaustive tree example.

Exponential algorithms can walk their state trees in either depth-first or breadth-first fashion Breadth-first walks require that entire levels be maintained in memory, whereas depth-first walks only require that the current branch be maintained in memory. Since

state trees are generally much wider than they are deep, depth-first walks are almost always more desirable.

When the state tree is binary and balanced like this example, it is easy to calculate that the number of required steps (branches) is $2^{n+1} - 2$. Thus, there are 14 required steps to generate all of the subgraphs for this example. But that is not the entire story. Once all of the subgraphs have been generated, each one needs to be evaluated to see if it is a maximal clique. This requires an extra 11 steps as follows:

1) Eliminate 123 due to nonadjacent nodes.
2) Verify that 12 has all adjacent nodes.
3) Eliminate 13 due to nonadjacent nodes.
4) Verify that 1 has all adjacent nodes.
5) Eliminate 1 as a subset of 12.
6) Eliminate 23 due to nonadjacent nodes.
7) Verify that 2 has all adjacent nodes.
8) Eliminate 2 as a subset of 12.
9) Verify that 3 has all adjacent nodes.
10) Verify that 3 is not a subset of 12.
11) Eliminate the null graph.

In fact, each of the subset checks will take an addition number of steps, so the actual number of steps is greater than the 25 already mentioned.

It would be better to terminate subtrees as soon as a nonadjacent vertex is added to a subset of adjacent vertices. Such a test is called a *bounding* condition and subtrees that are terminated due to bounding conditions are said to be *pruned*. Branching algorithms that have bounding conditions are called *branch-and-bound* algorithms. The goal of any branch-and-bound algorithm is to prune as many subtrees as possible using bounding conditions. The new tree with the nonadjacent bounding condition applied is shown in

Fig. 34. Note that any subtree that attempts to combine vertex 3 with either vertex 1 or vertex 2 is pruned.

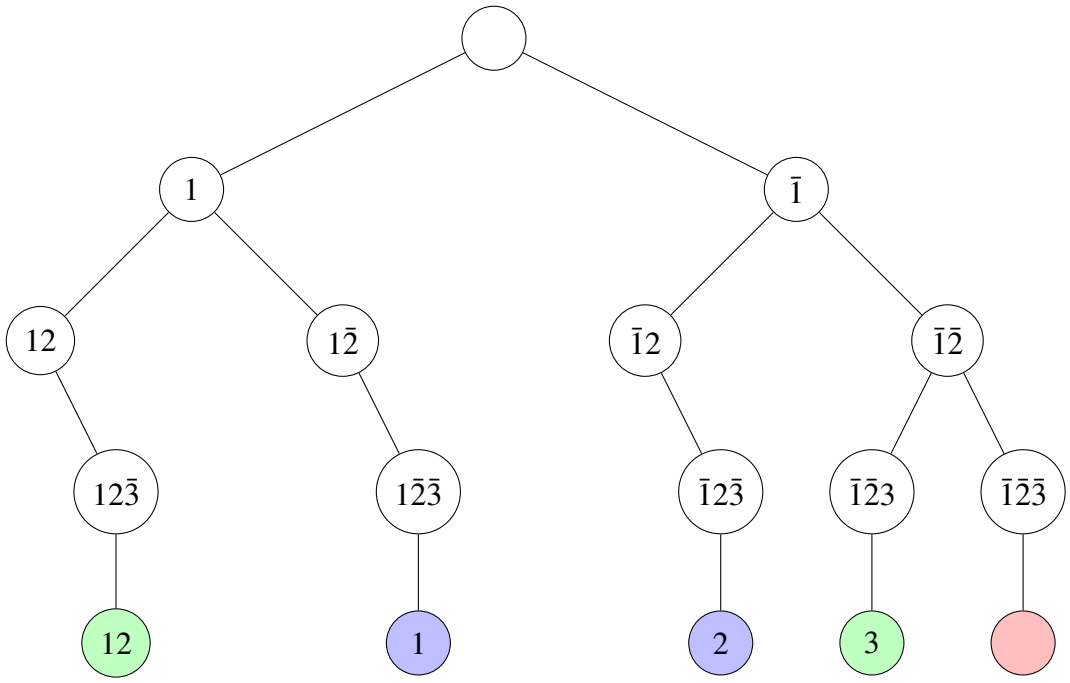


Fig. 34. Finding maximal cliques using nonadjacent bounding.

There is still the problem of eliminating non-maximal cliques and the null subgraph. This can be accomplished by maintaining a list of "used" vertices for each branch edge. When transitioning to the right from an "include vertex" branch to an "exclude vertex" branch, the excluded vertex is added to the branch's used list. When transitioning down an "include vertex" branch, used vertices that are not adjacent to the newly included vertex are removed from the branch's used list. Thus, a leaf node resulting from a branch with a non-empty used list is a subset of some previously found clique and hence is not maximal. The new tree with the non-maximal bounding condition applied is shown in Fig. 35.

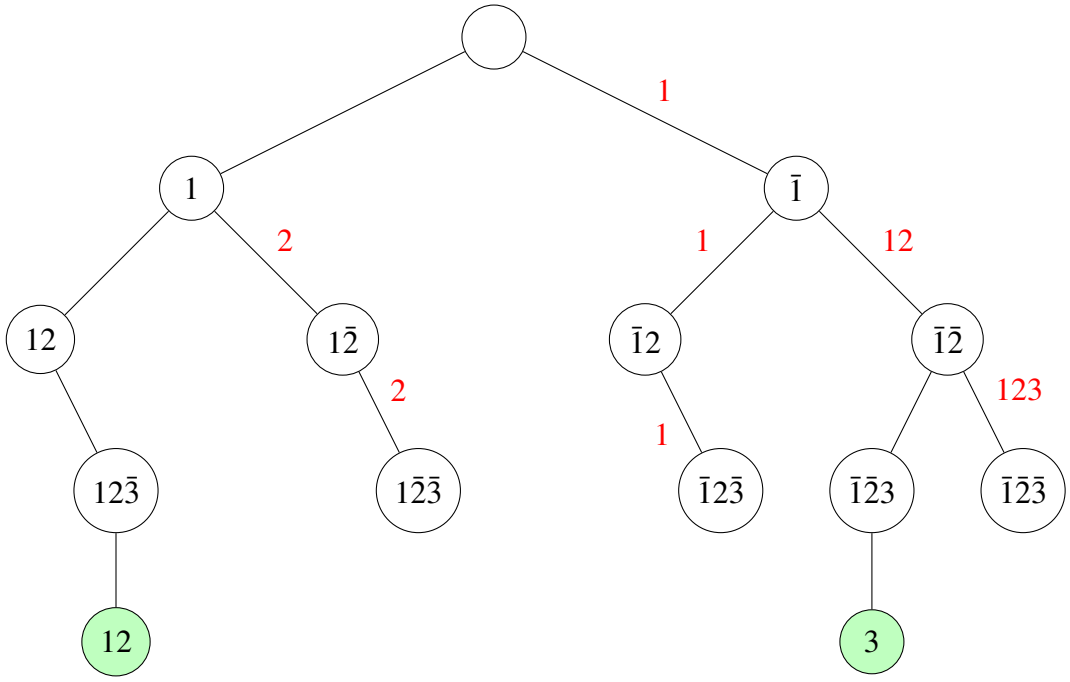


Fig. 35. Finding maximal cliques using non-maximal bounding.

Note that leaf node $1\bar{2}\bar{3}$ is not maximal because vertex 2 remains on the used list. Also note that the branch from $\bar{1}\bar{2}$ to $\bar{1}\bar{2}3$ eliminates vertices 1 and 2 from the used list because they are not adjacent to vertex 3. Thus, $\bar{1}\bar{2}3$ is a desired maximal clique. The null subgraph is eliminated because all of the vertices are on the used list.

Since bounding conditions are often very dependent on graph structure, it can be very hard to determine the theoretical runtime complexity for a specific branch-and-bound algorithm. This is especially true when multiple bounding conditions interact such that it is unclear what constitutes a worst case for the algorithm. In the previous maximal clique example, it is clear that the worst case has an upper bound of $\mathscr{O}(2^n)$; however, this bound is not very tight. In these cases, runtime complexity values gleaned from empirical data are convenient substitutes for truly theoretical answers. In fact, it will be shown in Section 4 using empirical data that the actual runtime complexity of this maximal clique algorithm is about $\mathscr{O}(1.25^n)$.

Furthermore, runtime complexities for exponential algorithms tend to address large n cases where the exponential nature of the whole algorithm far exceeds the effect of any P-time steps. As was shown in Section 3.2.1, these P-time steps are more significant at moderate values of n. Thus, worst case runtime complexity values for large n may be of little use if a problem domain is adequately addressed by lower values of n. For these cases, runtime duration of algorithms applied to real problems may be much more useful.

4 THE CHROMATIC NUMBER PROBLEM

This section investigates the various well-known methods to either estimate or find the exact chromatic number for a graph. Since the chromatic number problem is NP-hard, an alternative to finding an exact answer is finding lower and upper bounds for the actual value. If these bounds happen to match then they provide the actual chromatic number. Algorithms that find the exact chromatic number are of the branch-and-bound variety. The two most well-known algorithms are one proposed by Christofides 1971 [8] with modifications by Wang (1974) [9] and the so-called Zykov algorithms, with a particular implementation by Corneil and Graham (1973) [10].

The specifics of the random graph analysis used throughout this research are described in detail in Section 6. In short, a binomial edge probabity model was used. Trials were run for edge probabilities from $p = 10\%$ to $p = 90\%$ in steps of 10%. For P-time algorithms, 1000 trials were run for each edge probability and for each order from $n = 5$ to $n = 50$. For non-P-time algorithms, due to increased runtime duration, the maximum order was reduced to 30 and the number of trials was reduced to 100 for $n \geq 20$.

4.1 Finding a Lower Bound

The most popular strategy for estimating a lower bound for the chromatic number of a graph is based on the statement of Proposition 2. For a graph G:

$$\omega'(G) \leq \omega(G) \leq \chi(G)$$

where $\omega'(G)$ is a lower bound estimate for the clique number of G. Another less popular bound is given by Theorem 12 [11].

Theorem 12. *Let G be a graph of order n.* $\chi(G) \geq \dfrac{n}{\alpha(G)}$.

Proof. Assume that G is k-chromatic. This means that $V(G)$ can be partitioned into k non-empty independent sets $A_1, \ldots, A_k$, where each $|A_i| \leq \alpha(G)$.

$$n = \left| \bigcup_{1 \leq i \leq k} A_i \right| = \sum_{i=1}^{k} |A_i| \leq \sum_{i=1}^{k} \alpha(G) = k\alpha(G)$$

Therefore, $k \geq \dfrac{n}{\alpha(G)}$. □

Note that both of the above lower bounds are tight when G is empty or complete and are related by the statement of Theorem 8:

$$\omega(G) = \alpha(\bar{G})$$

4.1.1 The Mycielski Construction

It is well-known that certain triangle-free graphs with $\omega(G) = 2$ can have arbitrarily high $\chi(G)$. Examples are the graphs created using the so-called *Mycielski* construction [12]:

1) Start with $G = P_2$ $(\chi(G) = 2)$.
2) For the vertices in $v \in V(G)$, create new vertices $U = \{u_1, \ldots, u_n\}$ such that $N(u_i) = N(v_i)$. The new vertices form what is referred to as a *shadow* graph.
3) Add an additional vertex w such that $N(w) = U$ and call this new graph G', which has $\chi(G') = \chi(G) + 1$.
4) Let $G = G'$ and go to step 2.

The first three graphs resulting from the Mycielski construction are shown in Fig. 36.

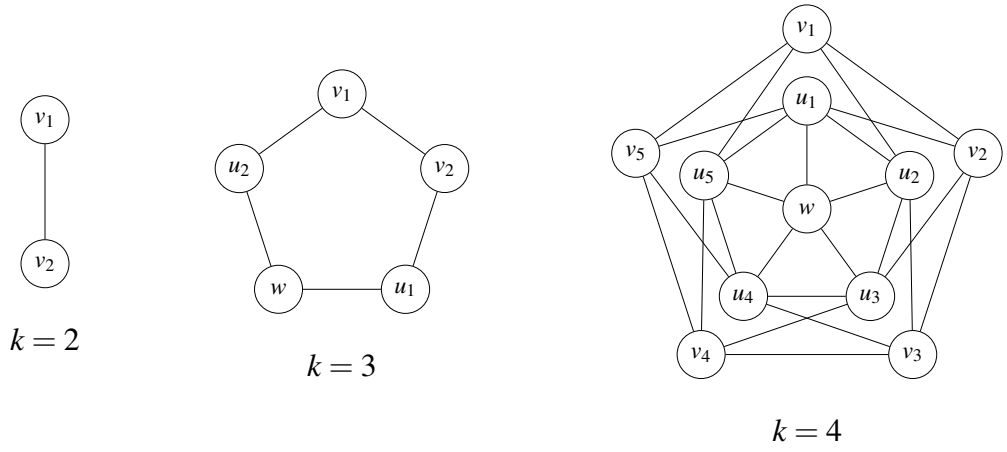


Fig. 36. The first three graphs from the Mycielski construction.

The third graph in Fig. 36 is called the *Grötzsch* graph. For another example of triangle free graphs with arbitrarily high chromatic number, see Zhang [18]. Nevertheless, for the general case the clique number is a suitable lower bound for the chromatic number.

4.1.2 The Edwards Elphick Algorithm

Unfortunately, the clique number problem for a graph G is also NP-hard, so the next best step is to use a P-time calculation for a $\omega'(G)$ that is as tight as possible to the actual $\omega(G)$. Edwards and Elphick (1982) [19] investigated several such methods and concluded that the best method was a simple calculation based on the adjacency matrix of G:

1) Select (either lowest index or at random) a vertex $v \in V(G)$ of maximum degree in G $(\deg(v) = \Delta(G))$ and let $S = \{v\}$.
2) Select the vertex $v_i \in V(G)$ such that $v_i \notin S$ and with the minimum index value i that is adjacent to all of the vertices in S. If no such vertex exists then go to step 4.
3) Add v_i to S and go to step 2.
4) $G[S]$ is a complete subgraph of G so conclude that $\omega'(G) = |S| \leq \omega(G)$.

The results of a random graph analysis of the Edwards Elphick algorithm measuring the mean of $\omega(G) - \omega'(G)$ are shown in Fig. 37. The error generally increases with both edge probability and order; however, there appears to be a small hitch in the curves

between $p = 80\%$ and $p = 90\%$ for orders $n \leq 40$. This may be because of the increased probability that the next selected vertex is in fact universal.

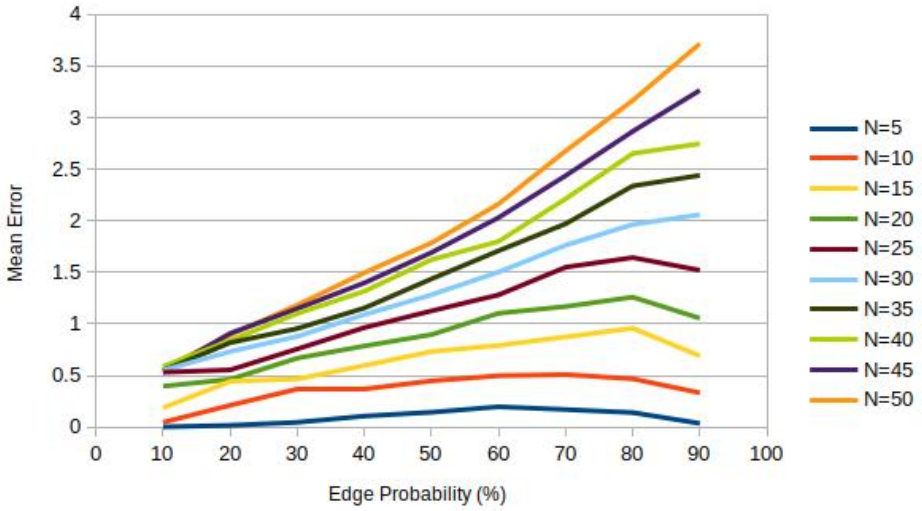


Fig. 37. Edwards Elphick algorithm mean error.

The mean number of steps is shown in Fig. 38. The number of steps increases with both edge probability and order, so the worst case for each order is assumed to be at $P = 90\%$.

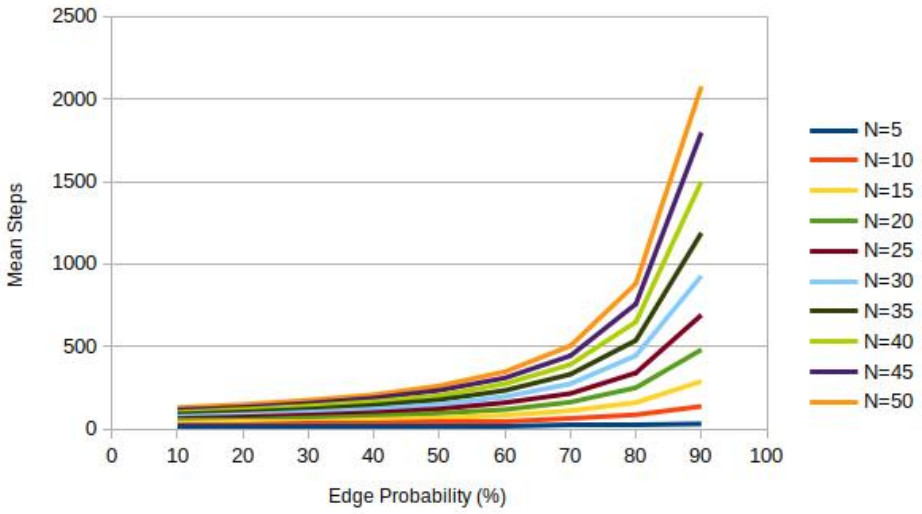


Fig. 38. Edwards Elphick algorithm mean number of steps.

A graph of the $P = 90\%$ values for each order is shown in Fig. 39. Note that the runtime complexity is $\mathcal{O}(n^2)$ as expected. Thus, the Edwards Elphick algorithm would be suitable for use as a lower bound approximator step in a branch-and-bound algorithm.

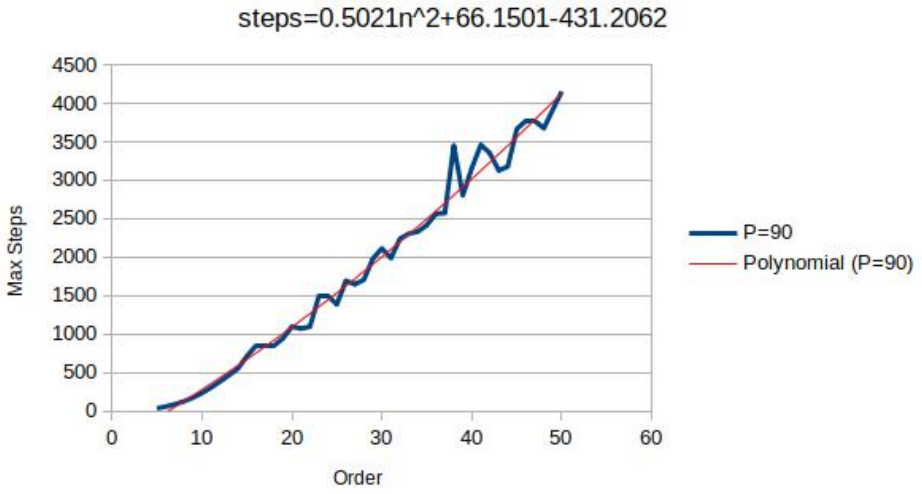


Fig. 39. Edwards Elphick algorithm runtime complexity.

Step 2 of the Edwards Elphick algorithm selects the next vertex of lowest index that is adjacent to all previously selected vertices. An improvement would be to select a vertex with the highest degree that is adjacent to all previously selected vertices. This would of course increase the average runtime complexity to the worst case of the unimproved algorithm, but it should still be P-time. The results of this improved algorithm are shown in Fig. 40. Note that the improved algorithm cuts the mean error in half. Also note that the hitch at $p = 80\%$ remains and is more pronounced, probably due to the increased probability of finding a high degree vertex that is more likely to be part of a clique.

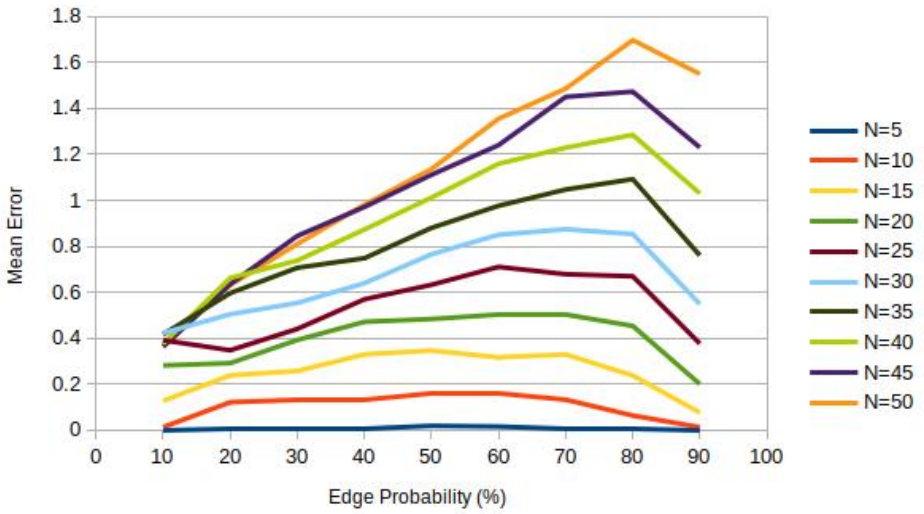


Fig. 40. Improved Edwards Elphick algorithm mean error.

The increase in the number of steps is demonstrated in Fig. 41.

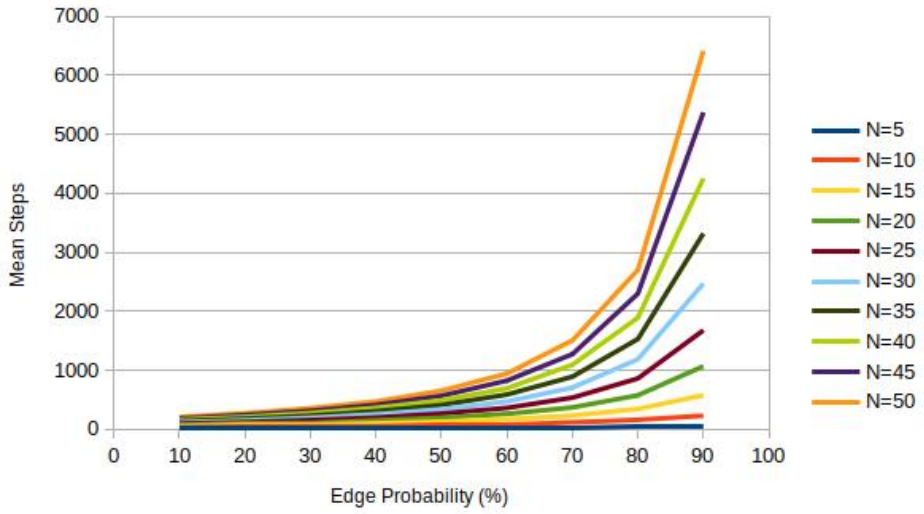


Fig. 41. Improved Edwards Elphick algorithm mean number of steps.

And the new runtime complexity approximation is shown in Fig. 42. The improved algorithm is still $\mathcal{O}(n^2$.

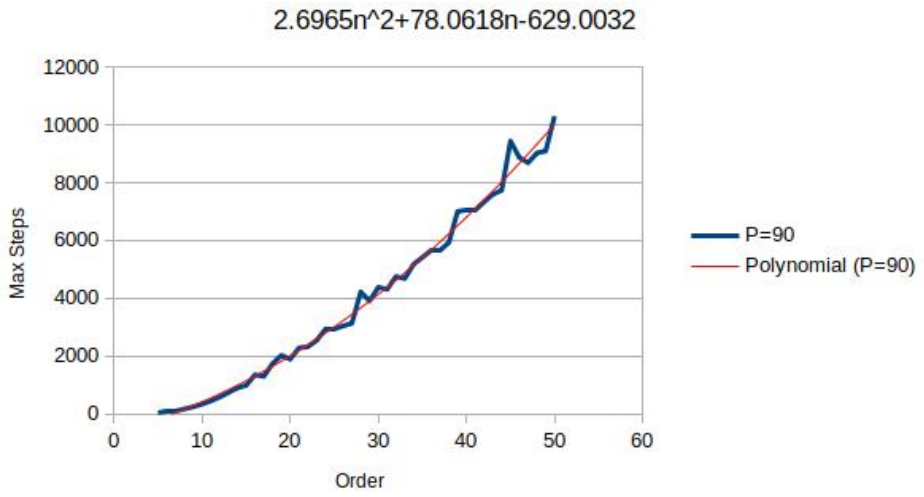


Fig. 42. Improved Edwards Elphick algorithm runtime complexity.

4.1.3 The Bron Kerbosch Algorithm

Although an estimate of the clique number is nice, an exact value is better. Unfortunately, the clique number problem is also NP-hard. A nice summary of well-known exact clique number algorithms is given by Xiao and Nagamouchi (2017) [20]. They claim that the known algorithms tend to converge on a runtime complexity of $\mathcal{O}(1.2^n)$. These algorithms tend be somewhat complex and geared towards larger n.

A simpler yet efficient alternative to these exact algorithms for more modest values of n is the Bron Kerbosch (BK) algorithm (1973) [21]. In fact, the algorithm that was incrementally developed in Section 3.3 is essentially the BK algorithm. The advantage of BK is that it finds all possible cliques in a graph, and hence can be used to find $\alpha(G) = \omega(\bar{G})$. Moon and Moser (1965) [22] show that every graph G of order n has at most $3^{\frac{n}{3}}$ maximal cliques, so the runtime complexity of the Bron Kerbosch algorithm is expected to be about $\mathcal{O}(1.44^n)$.

The heart of BK is a recursive subroutine called *extend* that implements the breadth of a level in the corresponding state tree, and recursively calls itself in order to implement the branches in the state tree. At each node in the state tree, three vertex lists are maintained:

compsub The current maximal clique accumulator.

candidates A set of vertices that can be added to *compsub*.

used A set of vertices that already have been used in previous branches.

The initial call is seeded with an empty *compsub* and all of the graph's vertices in *candidates*. Each call to *extend* performs the following steps:

1) If *used* contains a vertex that is adjacent to everything in *candidates* then any generated cliques in the current subtree will never be maximal, so return. This implements the non-maximal bounding condition.

2) The next vertex is selected from *candidates* and is added to *compsub*. This implements the "include vertex" subtree.
3) New versions of *candidates* and *used* are created by removing vertices from the old lists that are not adjacent to the selected vertex. This bounding condition prunes branches that might mix adjacent and nonadjacent vertices.
4) A recursive call with the new *candidates* and *used* lists is made to continue the current subtree.
5) The selected vertex is removed from *compsub* and is added to *used*. This implements the "exclude vertex" subtree.
6) If *candidates* is not empty then go to step 1.
7) If *used* is empty then *compsub* contains the vertices for a maximal clique.
8) Return to the previous level in the state tree.

A small improvement added to BK by this research is to abandon the current branch when the desire is to only find $\alpha(G)$ (and hence $\omega(\bar{G})$) and the number of vertices in *compsub* and *candidates* are not enough to build a maximal clique larger than all previously found maximal cliques.

Bron and Kerbosch actually proposed two versions of their algorithm that differ by how the next vertex is selected from the *candidates* list in step 2. In the basic mode, the first (or any) vertex in the list is selected. Fig. 43 shows the average number of calls to the *extend* method. The number of calls increases with both graph order and edge probability, except for the appearance of the mysterious hitch again at $p = 80\%$. The worst case for each order is thus assumed to occur at 90% edge probability.

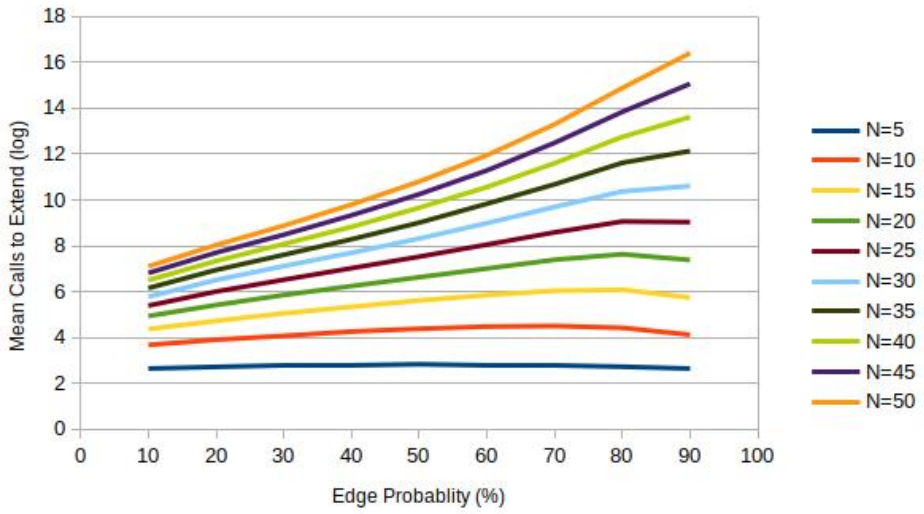


Fig. 43. Basic Bron Kerbosch algorithm calls to extend.

A graph of the $P = 90\%$ values for each order is shown in Fig. 44. This time, the graph appears to have a log effect and indeed the best curve fit is a mix of n and $log(n)$. This is expected based on the discussion in Section 3.2.1. The fit indicates that the runtime complexity is about $\mathcal{O}(2^{0.2259n}) \approx \mathcal{O}(1.17^n)$.

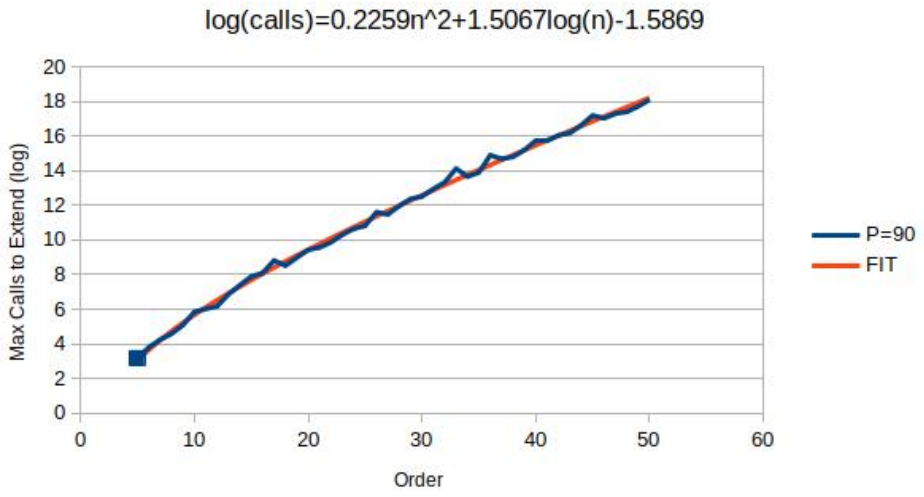


Fig. 44. Basic Bron Kerbosch algorithm runtime complexity.

In smart mode, a particular vertex in the *used* list with the smallest number of nonadjacencies to vertices in the *candidates* list is identified. The next selected vertex is then a vertex that is not adjacent to the identified vertex. This causes the non-maximal bounding condition to occur as soon as possible. Fig. 45 shows the average number of calls to the *extend* method for the smart version of the algorithm.

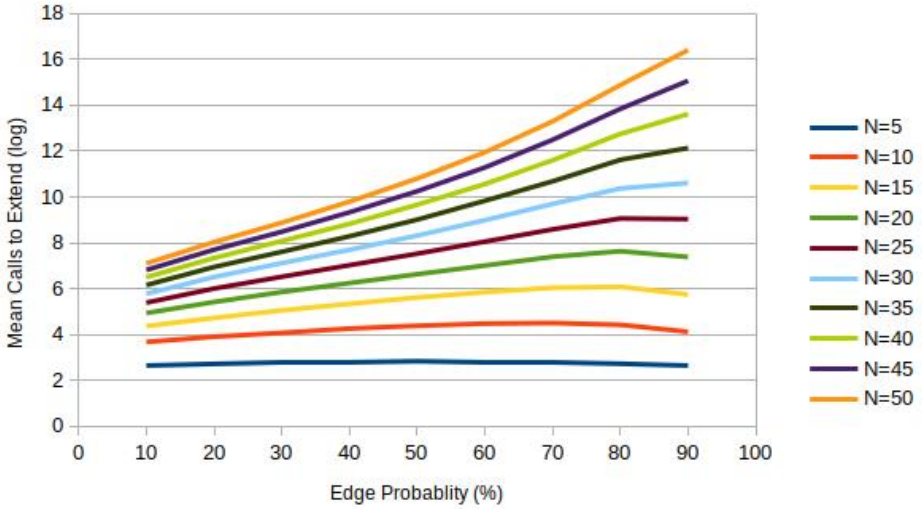


Fig. 45. Smart Bron Kerbosch algorithm calls to extend.

The runtime complexity estimate for the smart version of the algorithm is shown in Fig. 46. Note that the runtime complexity is improved to $\mathcal{O}(2^{0.1867n}) \approx \mathcal{O}(1.14^n)$.

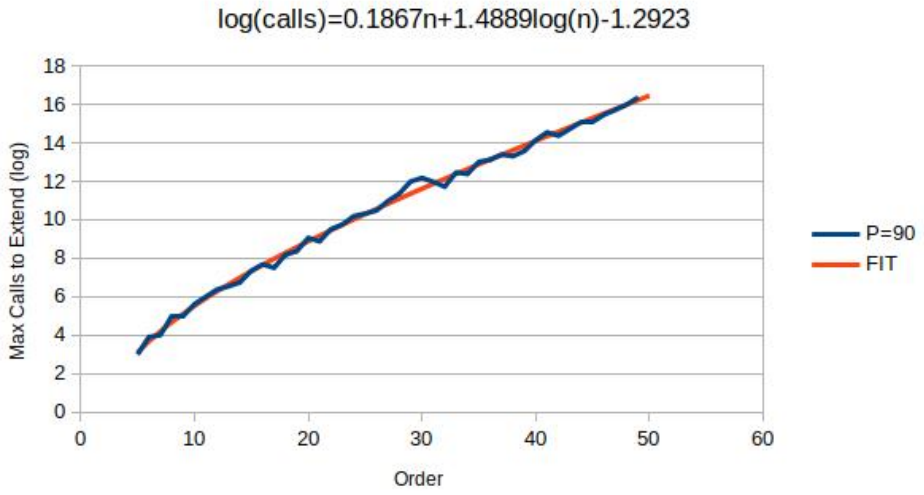


Fig. 46. Smart Bron Kerbosch algorithm runtime complexity.

Therefore, BK in smart mode decreases the runtime complexity over the target range from $\mathcal{O}(1.17^n)$ to $\mathcal{O}(1.14^n)$ and so the improved bounding condition is effective.

4.2 Finding an Upper Bound

The most popular technique for finding an upper bound for the chromatic number of a graph is to construct a proper coloring for the graph using a so-called *sequential* algorithm, often referred to as a *greedy* algorithm. The algorithms are sequential because the vertices are ordered in some fashion and are then colored according to that order. The algorithms are greedy because a new color is selected whenever one is needed. The result is that too many colors may be used; however, such algorithms are P-time and the number of colors used is suitable as an upper bound for the chromatic number since the graph is at least colorable using that many colors.

The specific steps of a greedy algorithm for a graph G of order n are as follows:

1) Order the vertices in some fashion: $V = \{v_1, \ldots, v_n\}$.
2) Start with $C = \emptyset$ and assume some coloring function: $c : V \to C$.
3) Let $i = 1$.

4) Let $k = |C|$.

5) If $i = n$ then done and k is an upper bound for the chromatic number.

6) Determine all of the already colored vertices that are adjacent to v_i:

$$S = \left\{ v_j \in V \,\middle|\, j < i \text{ and } v_i v_j \in E(G) \right\}$$

7) Determine all of the colors used by the vertices in S: $c[S]$.

8) If $c[S] = C$ then an additional color is needed for v_i, so add c_{k+1} to C, let $c_j = c_{k+1}$, and go to step 10.

9) Otherwise, an existing color can be reused for v_i so select c_j from $C - c[S]$ with the smallest j.

10) Color v_i with c_j by extending c: $c(v_i) = c_j$.

11) Let $i = i + 1$.

12) Go to step 4.

The two most popular theorems used to quickly estimate the chromatic number upper bound of a graph follow from the worst case results from the greedy algorithm. The first is based on a random ordering of the vertices [11].

Theorem 13. *Let G be a graph.* $\chi(G) \leq 1 + \Delta(G)$

The second, from Welsh and Powell (1967), is based on ordering by non-increasing vertex degree [23].

Theorem 14. *Let G be a graph:*

$$\chi(G) \leq \max_i \min \{1 + \deg(v_i), i\}$$

But does there exist an ordering of the vertices such that the greedy algorithm gives the correct exact answer? This question is answered by Theorem 15.

Theorem 15. *Let G be a graph. There exists an ordering of the vertices in G such that the greedy algorithm produces an exact result for* $\chi(G)$.

Proof. Assume that $\chi(G) = k$. This means that $V(G)$ can be partitions into k independent sets $\{A_1, \ldots, A_k\}$. Order the vertices starting with the vertices in A_1, and then A_2 and so on, finishing with the vertices in A_k. Use this ordering as an input to the greedy algorithm.

Consider the following proof by induction on $1 \leq i \leq n(G)$ where i selects the i^{th} vertex in the ordered list. Color v_1 using color c_1, which is trivially proper using one color. Assume that the coloring of the first v_i is proper and uses at most k colors. Now consider v_{i+1}. Assume that $v_{i+1} \in A_j$ for $1 \leq j \leq k$. If there exists some $1 \leq r < j$ such that v_{i+1} is not adjacent to any of the vertices in A_r then color v_{i+1} with color c_r using the smallest such r. Otherwise, since v_{i+1} is not adjacent to any of the vertices in A_j, color v_{i+1} with color c_j. In both cases the coloring is proper using at most k colors. Therefore, the greedy coloring algorithm produces an exact result. □

It will be shown in Section 5.3 that one of the outputs of the proposed algorithm is such an ordering. In fact, the final step of the proposed algorithm uses this fact to construct a final chromatic coloring.

Matula, et al. (1967) [24] performed a study on the various well-known chromatic number greedy algorithms and concluded that ordering the vertices by non-increasing degree order, as in the Welsh Powell bound, worked best. Matula referred to this algorithm as the *last-first* algorithm. The results of a random graph analysis of the last-first greedy algorithm are shown in Fig. 51. The graph shows the mean difference between the value found by the greedy algorithm and the actual chromatic number. Note that the former is always greater than or equal to the latter. Due to runtime duration considerations, only 100 trials were run per edge probability and order was limited to 20, which is the target range. The algorithm is exact for an empty or complete graphs and indeed Fig. 51 indicates that the algorithm performs better at lower and higher edge probabilities.

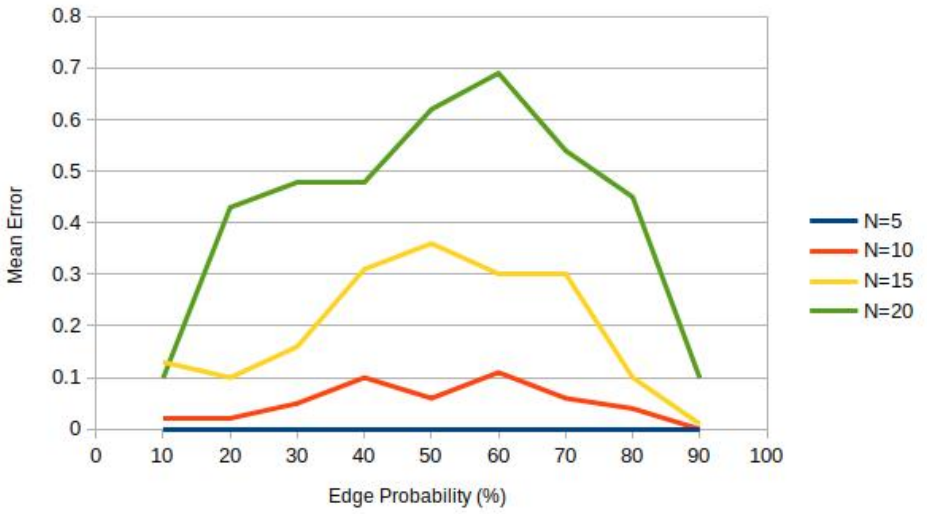


Fig. 47. Last-first greedy algorithm error.

The mean number of steps is shown in Fig. 48. The number of steps increases with both edge probability and order, so the worst case for each order is assumed to be at $P = 90\%$.

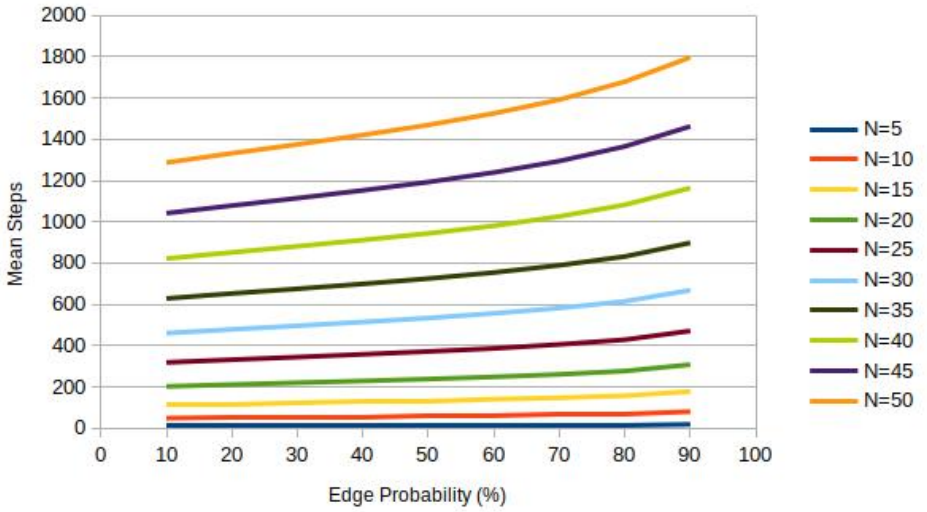


Fig. 48. Last-first greedy algorithm steps.

A graph of the $P = 90\%$ values for each order is shown in Fig. 49. Note that the runtime complexity is $\mathcal{O}(n^2)$ as expected.

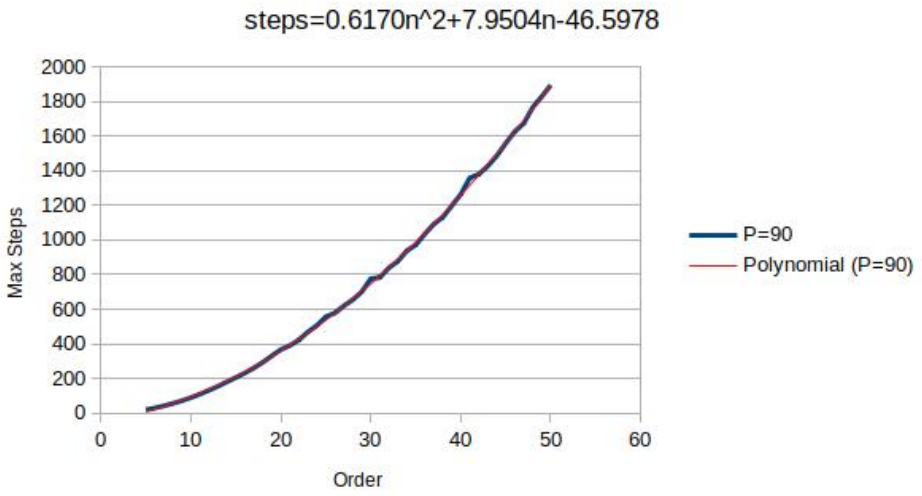
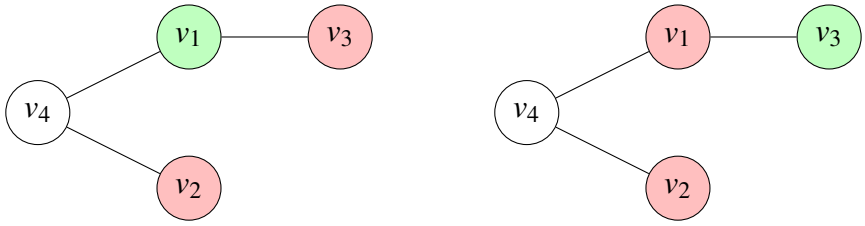


Fig. 49. Last-first greedy algorithm runtime complexity.

Matula proposed an improvement to greedy algorithms known as *color interchange*. Color interchange is based on the situation summarized by the example in Fig. 50. When it is time to color v_4, the normal greedy algorithm is forced to select a new color. However, vertices v_1 and v_3 can swap colors and thus v_4 can use an existing color.



Before Swap After Swap

Fig. 50. An example that allows color interchange.

The specific steps for color interchange when attempting to color vertex v_i are as follows:

1) Determine all of the colors that are used for already colored vertices that are adjacent to v_i.
2) Select those colors that occur only once in the neighborhood of v_i.
3) Select all vertices that are already colored with the used-once colors.
4) Construct a subgraph using the selected vertices.
5) Partition the subgraph into components.
6) Find a component that includes one vertex and excludes one vertex that is adjacent to v_i in the original graph. If no such component is found then interchange is not possible and a new color must be used for v_i.
7) Let c_1 be the color of the included vertex and let c_2 be the color of the excluded vertex.
8) Interchange colors c_1 and c_2 for all such colored vertices in the selected component.
9) Color c_1 is now available for v_i.

The mean error when color interchange is added to the last-first greedy algorithm is shown in Fig. **??**. The Hopcroft Tarjan algorithm [14] introduced in Section 2.8.5 was used to partition the subgraph in step 5. The algorithm still tends to do better at lower and higher edge densities. It appears that color interchange has a slight advantage at higher orders.

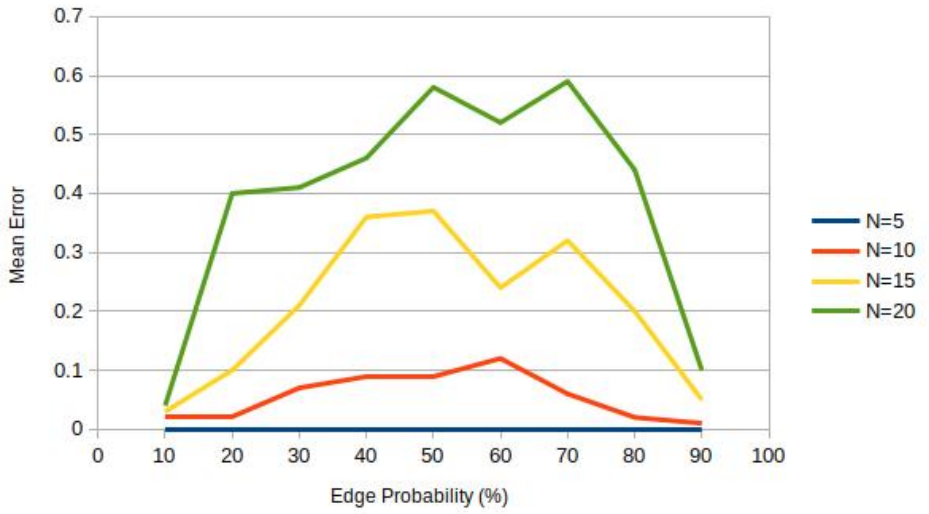


Fig. 51. Last-first greedy algorithm with color interchange error.

The mean number of steps with interchange is shown in Fig. 52. Once again, the number of steps increases with both edge probability and order, so the worst case for each order is assumed to be at $P = 90\%$.

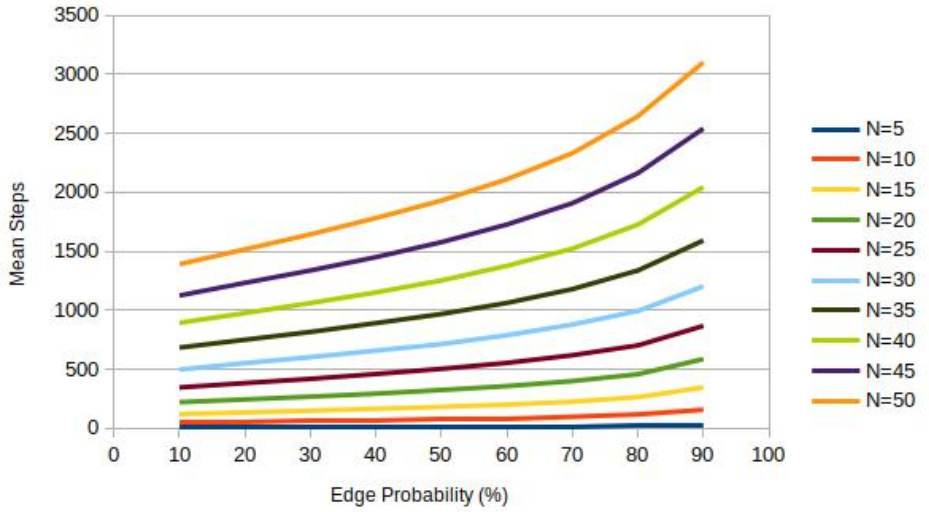


Fig. 52. Last-first greedy algorithm with color interchange steps.

A graph of the $P = 90\%$ values for each order is shown in Fig. 53. Note that color interchange is a bit expensive; however, the runtime complexity is still $\mathcal{O}(n^2)$.

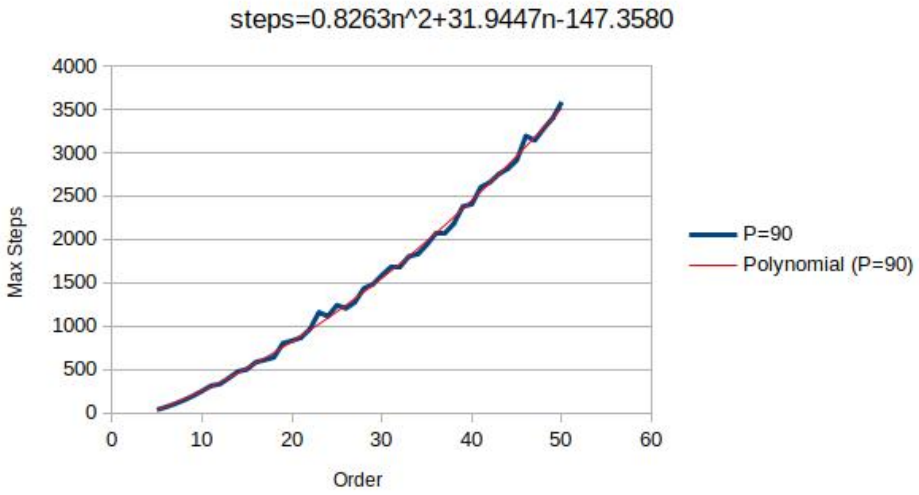


Fig. 53. Last-first greedy algorithm with color interchange runtime complexity.

4.3 The Christofides Algorithm

The first exhaustive algorithm that will be examined was proposed by Cypriot mathematician Nicos Christofides (1971) [8]. The Christofides algorithm is a breadth-first algorithm that assembles maximal independent sets from a graph until the first combination that uses all of the vertices is found. Thus, the Bron Kerbosch algorithm is a vital part of the Christofides algorithm.

The Christofides algorithm starts by decomposing a graph G into all of its maximal independent sets. This constitutes the first level of the state tree. Then, for each maximal independent set with vertices S, the subgraph $G - S$ is constructed and all of its maximal independent sets are found. Each of these sets is combined with the previous maximal independent sets to form the next layer of the state tree. This process continues until the first time that all of the vertices in G are used. The number of maximal independent sets used to form the final combination is the chromatic number.

The Christofides does have bounding conditions. If the vertices in any new combination of maximal independent sets is a subset of a previous combination then the new subtree is pruned. If the new is a superset of a previous then the previous is pruned by replacing it with the new.

For example, consider the graph in Fig. 54.

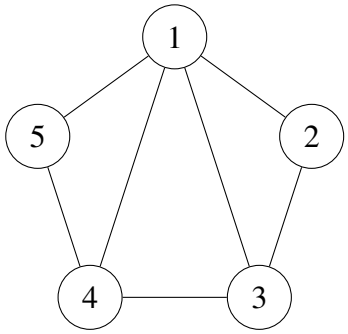


Fig. 54. A Christofides algorithm example.

The Bron Kerbosch algorithm is used to decompose Fig. 54 into the maximal independent sets $\{1\}$, $\{2,4\}$, $\{2,5\}$, and $\{3,5\}$. This first level is shown in Fig. 55, along with the resulting subgraphs.

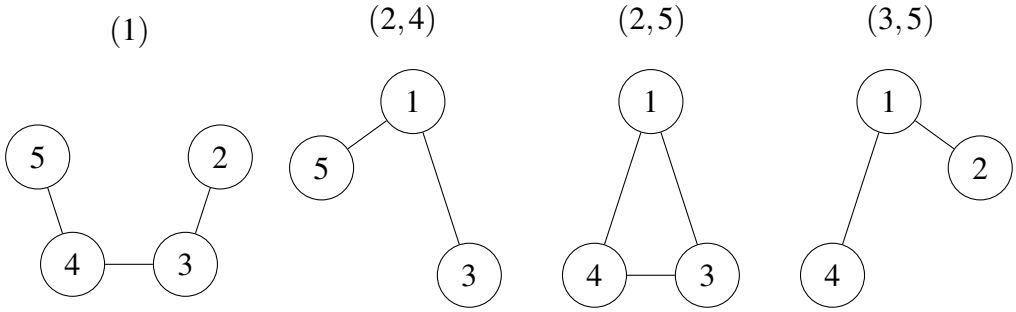


Fig. 55. Level 1 of a Christofides algorithm example.

The next level starts with the leftmost subgraph in Fig. 55. It has the maximal independent sets $\{2,4\}$ and $\{3,5\}$. These are combined with the parent state to form the next states $\{\{1\},\{2,4\}\}$ and $\{\{1\},\{3,5\}\}$. Likewise, the second graph in Fig. 55

contains maximal independent sets $\{1\}$ and $\{3,5\}$. This yields the next states $\{\{2,4\},\{1\}\}$ and $\{\{2,4\},\{3,5\}\}$; however, the vertices in $\{\{2,4\},\{1\}\}$ are a subset of the previous state $\{\{1\},\{2,4\}\}$ and so the new subtree is pruned. This process continues for the third and fourth subgraph in Fig. 55. The second level results are as follows:

$(1|24)$ $(1|35)$ ~~$(24|1)$~~ $(24|35)$ $(25|1)$ ~~$(25|3)$~~ ~~$(25|4)$~~ ~~$(35|1)$~~ ~~$(35|24)$~~

Next, the first state in the second level contains a single maximal independent set $\{3,5\}$, which when combined with the parent state uses all of the vertices. The final coloring is thus $(1|24|35)$ and the graph is 3-chromatic as shown in Fig. 56.

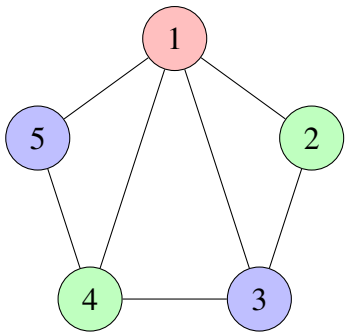


Fig. 56. Christofides algorithm example results.

Fig. 57 shows the results of a random graph analysis of the Christofides algorithm. It measures the mean number of calls to the routine that processes each found MIS and applies the bounding conditions. Thus, the number of calls is essentially the number of states in the state tree.

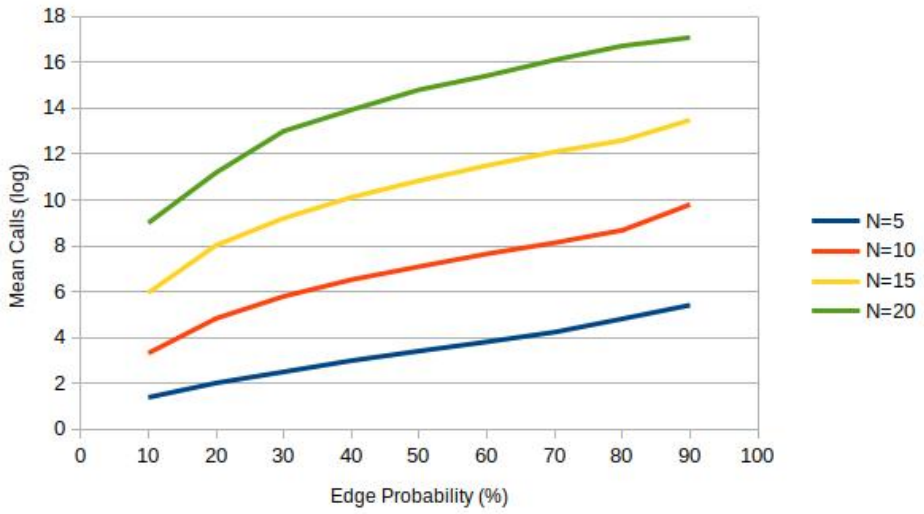


Fig. 57. Christofides algorithm mean number of calls.

The number of calls increases with both graph order and edge probability. The worst case for each order is thus assumed to occur at 90% edge probability. A log (base 2) plot of the maximum number of calls for each order at 90% edge probability is shown in Fig. 58. Note that due to excessive runtime duration, the test had to be stopped at $n = 20$. A linear curve fit indicates that the runtime complexity for the Christofides algorithm is about $\mathcal{O}(2^{0.7607n}) \approx \mathcal{O}(1.69^n)$.

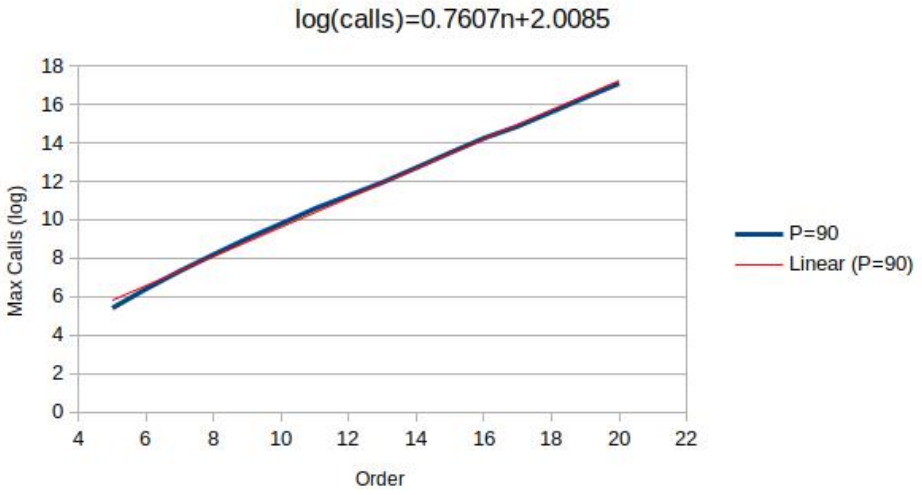


Fig. 58. Christofides algorithm runtime complexity.

4.4 Wang Improvements to Christofides

One drawback of the Christofides algorithm is the fact that each level must be maintained entirely in memory before moving on to the next level. As has been show, the breadth of state graphs grows exponentially, so depth-first algorithms are generally preferred. Wang (1974) [9] proposes two improvements to the Christofides algorithm to combat this memory usage. The first improvement prunes a large number of subtrees and the second improvement converts the algorithm to a depth-first search.

Wang's first proposal is based on Lemma 1

Lemma 1. *Let G be a graph, let $v \in V(G)$, and let $\{M_1, \ldots, M_r\}$ be all of the maximal independent sets in G containing v. There exists a chromatic coloring of G containing one of the M_i.*

Proof. Assume that G is k-chromatic and let $\{A_1, \ldots, A_k\}$ be the independent sets of a chromatic coloring of G. Assume without loss of generality (AWLOG) that $v \in A_1$. It must be the case that $A_1 \subseteq M_i$ for some $1 \leq i \leq r$, since all of the M_i are maximal. Now,

let $X = M_i - A_1$ and let $X_j = A_j - X$ for $2 \leq j \leq k$. Next, construct the coloring $\{M_i, X_2, \ldots, X_k\}$. This is a k-chromatic coloring of G containing M_i. □

Basically, Lemma 1 says that any independent set in a chromatic coloring of a graph G containing a vertex v can be extended to a maximal independent set containing v by snatching vertices from the other independent sets in the coloring.

Next consider the fact that each state in the Christofides algorithm state tree represents a subgraph and the goal is to (recursively) find a chromatic coloring for that subgraph. Therefore, a particular vertex can be selected and only MISs containing that vertex need be considered for the next level. So, at each state, select the vertex that appears in the fewest MISs of the subgraph; the subtrees corresponding to the MISs that do not contain the selected vertex are pruned.

Referring back to the example in Fig. 54, recall that the first level MISs were: $\{1\}$, $\{2,4\}$, $\{2,5\}$, and $\{3,5\}$. Vertices 1, 3, and 4 occur in only one MIS each. So if 1 is selected, only the leftmost subgraph in Fig. 55 need be considered.

Wang's second improvement is to convert the search to a depth-first search, keeping track of the minimum length branch from the root state to a leaf state. Branches that equal or exceed the current minimum are pruned. Branches that are smaller than the current minimum become the new current minimum. Although this does require that the entire pruned tree be traversed, the hope is that the first improvement has pruned enough subtrees so that the depth-first search is now economical.

The results of a random graph analysis of Wang's improvements to the Christofides algorithm are shown in Fig. 59.

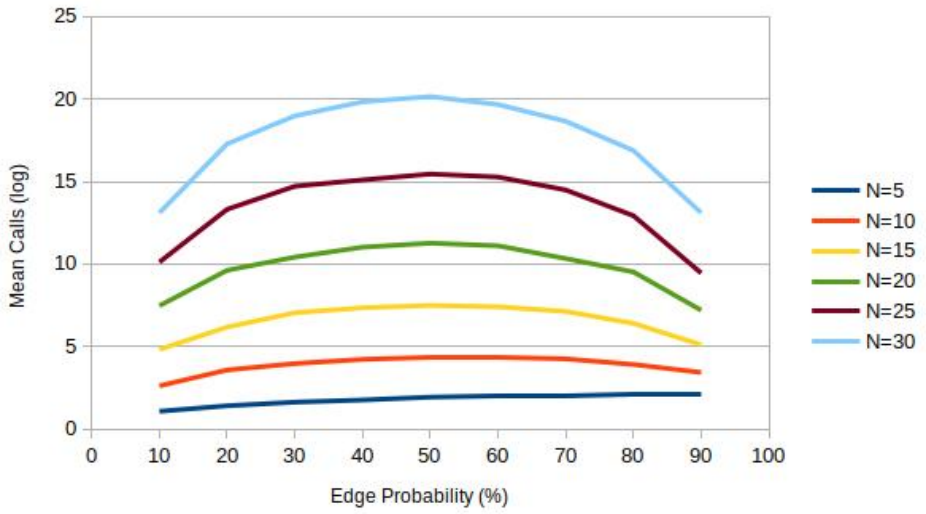


Fig. 59. Christofides algorithm with Wang improvements mean number of calls.

The results are quite surprising. Not only is there a dramatic improvement, it now appears that the worst case occurs at moderate edge density. Intuition suggests that lower edge density graphs will have fewer and larger MISs. For the high density case, it may be that each vertex is in fewer MISs. The worst case for each order is thus assumed to occur at 50% edge probability. The runtime complexity calculation is shown in Fig. 60. This time, a polynomial curve fit is required, resulting in a runtime complexity of about $\mathcal{O}(2^{0.0065n^2}) \approx \mathcal{O}(1.0045^{n^2})$.

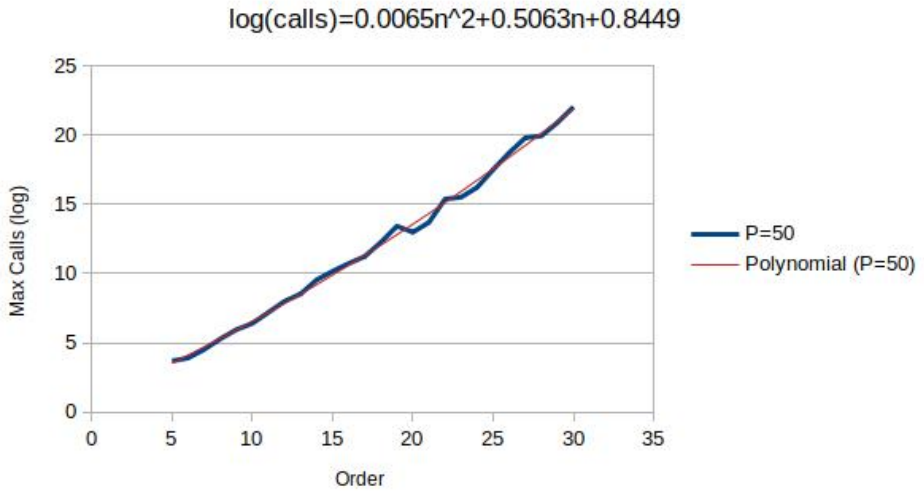


Fig. 60. Christofides algorithm with Wang improvements runtime complexity.

A comparison of the Christofides and Wang runtime complexities demonstrates some of the pitfalls of big-$\mathcal{O}$ notation and improvements to exhaustive algorithms. The Wang improvements clearly have an advantage; however, the big-$\mathcal{O}$ analysis indicates that plain Christofides is $\mathcal{O}(a^n)$ and Wang is $BO(b^{n^2})$. The difference is due to the fact that $a = 1.69 > 1.0045 = b$ and so for lower values of n, Wang wins. In fact, the runtime complexity analysis predicts that Wang loses its advantage at about $n = 43$. Unfortunately, the runtime durations at that value of n are too long to test the threshold.

4.5 Zykov Algorithms

The second exhaustive algorithm that will be examined is based on a branching technique attributed to Ukranian mathematician Alexandre A. Zykov. In his 1949 paper (translated by the AMS in 1952) [13], Zykov addresses the question: given a graph G and a number $k \in \mathbb{N}$, how many ways are there to properly color G using at most k colors? In fact, he is not particularly concerned about the chromatic number, which he calls the *rank* of a graph.

To solve this problem, Zykov notes that in any proper coloring of a graph:

1) Nonadjacent vertices have either the same color or different colors.

2) Adjacent vertices always have different colors.

If nonadjacent vertices have the same color then they can be contracted and the resulting graph retains the same k-coloring as the original graph. This is demonstrated in Fig. 61.

G $\qquad\qquad$ $G \cdot bd$

Fig. 61. Same colors with vertex contraction.

If nonadjacent vertices have different colors then they can be joined by an edge and the resulting graph retains the same k-coloring as the original graph. This is demonstrated in Fig. 62.

G $\qquad\qquad$ $G + bc$

Fig. 62. Different colors with edge addition.

By applying these steps recursively, all of the possible partitions of the nonadjacent vertices to independent sets are generated. The termination condition for each recursive path is a complete graph of some varying order k. Each node in the complete graph represents an independent set of nonadjacent nodes in the original graph that have been combined via vertex contraction. Thus, each complete graph of order k represents a

possible k-coloring of the original graph. The complete graphs of smallest order represent chromatic colorings and their order is the chromatic number of the original graph.

Zykov uses a graph equation syntax to record the recursive processing of a graph, where each line in the equation represents the next recursive layer. Isomorphic graphs are combined with a frequency multiplier at each layer. This is demonstrated in Fig. 63.

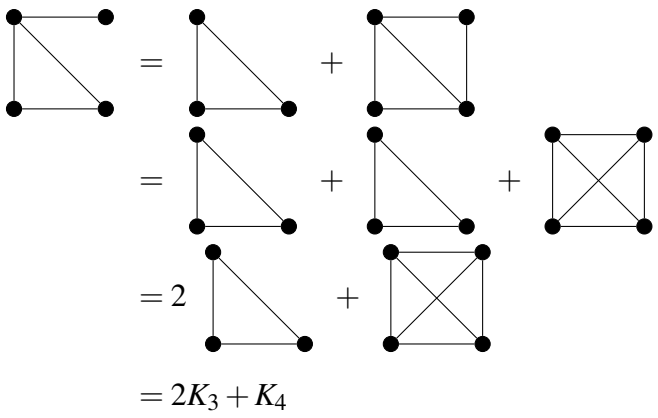


Fig. 63. A Zykov graph equation example.

Determining whether two graphs are isomorphic is hard, so combining isomorphic graphs in all but the very simple cases should be skipped; the complete graphs resulting from the further processing of two isomorphic graphs will eventually be combined anyway by the end.

Zykov was trying to determine the number of k-colorings of a graph without color indifference: each permutation of colors for a particular distribution is considered unique. Thus, Zykov multiplied each complete graph coefficient in the final line of a graph equation by the number of permutations resulting from selecting the order n of the

particular complete graph from k colors:

$$k^{(n)} = k(k-1)(k-2)\cdots(k-n+1)$$

So the total number of unique colorings for the example shown in Fig. 63 using k colors would be:

$$M(G,k) = 2k^{(3)} + k^{(4)} \tag{3}$$

Equation 3 is known as the *factorial form* of the *chromatic polynomial* for the graph. The corresponding *expanded form* is shown in Equation 4.

$$M(G,k) = k^4 - 4k^3 + 5k^2 - 2k \tag{4}$$

Read (1968) [25] expands on the construction of the factorial form of the chromatic polynomial for a graph and proves several theorems regarding the expanded form. Some examples are:

1) $M(G,k) = M(G \cdot uv) + M(G + uv)$, where u and v are any two nonadjacent vertices in the current recursive step.
2) The degree of M(G,k) is the order of G.
3) The highest order coefficient is 1.
4) There is no constant term.
5) The terms alternate in sign.

In fact, Read shows that the expanded form is actually an inclusion-exclusion equation resulting from starting with all possible proper and improper colorings k^n and then subtracting the improper colorings.

Corneil and Graham (1973) extend Zykov's work with Theorem 16 [10]:

Theorem 16. *Let G be a graph and let u and v be two nonadjacent vertices in G:*

$$\chi(G) = \min\{\chi(G \cdot uv), \chi(G + uv)\}$$

Zykov's method combined with Theorem 16 can be used to construct a depth-first branching algorithm for finding the chromatic number and a chromatic coloring for a graph G. Each state in the state tree for such an algorithm is represented by a graph whose nodes are sets of contracted vertices from G and hence represent independent sets in a candidate coloring, and whose edges are the edges remaining after the vertex contractions. The leaves of the state tree are complete graphs that represent proper colorings. The leaf state graphs with the smallest order are chromatic colorings. Such a state tree is called a *Zykov tree* and a branch (and bound) algorithm that uses such trees is called a *Zykov algorithm* [10].

The Zykov tree for the example in Fig. 63 is shown in Fig. 64. Note that the three leaf states are complete graphs of order 3 or 4. Therefore, the example in Fig. 63 is 3-chromatic and the two K_3 leaves represent chromatic colorings.

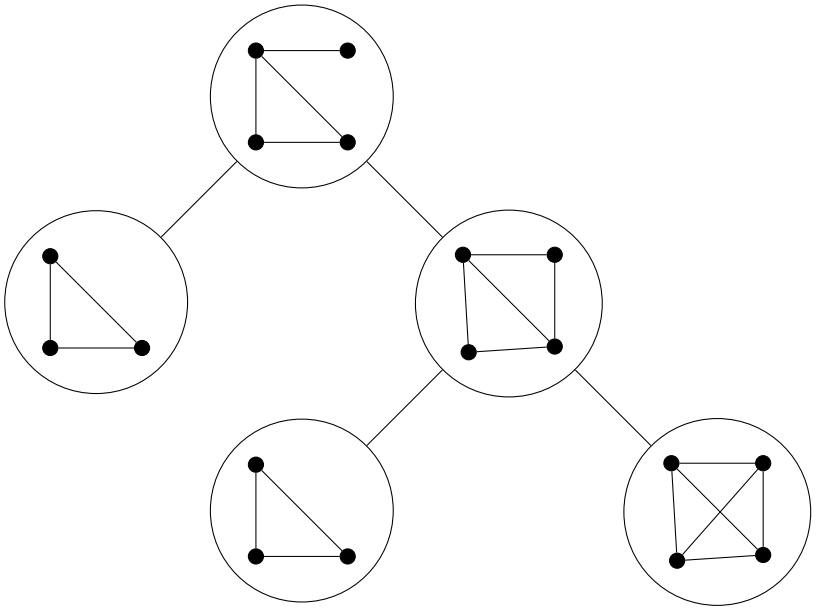


Fig. 64. A Zykov tree example.

The worst case for a Zykov algorithm with no bounding is an empty graph, where the number of leaf nodes is equivalent to the number of partitions of the set of vertices. This is known to be the so-called Bell number [26]:

$$B_n = \sum_{k=1}^{n} S_{n,k}$$

where the $S_{n,k}$ are the so-called Stirling numbers of the second kind:

$$S_{n,k} = \frac{1}{k!} \sum_{i=0}^{k} (-1)^i \binom{k}{i} (k-i)^n$$

The first 20 Bell numbers are provided in Table 5 [27]. The best case is a complete graph, which terminates immediately.

Table 5

Bell Numbers

1
1
2
5
15
52
203
877
4,140
21,147
115,975
678,570
4,213,597
27,644,437
190,899,322
1,382,958,545
10,480,142,147
82,864,869,804
682,076,806,159
5,832,742,205,057

Branch-and-bound Zykov algorithms require the following components:

1) A main routine that establishes G as the root of the state tree.
2) A global variable X that records the state corresponding to the current smallest k-coloring.
3) A global variable b that records the current upper bound for the chromatic number of G.
4) A method to determine a lower bound for the chromatic number of a graph.
5) A method to determine an upper bound k for the chromatic number of a graph G and a corresponding k-coloring of G.
6) A recursive subroutine that performs the depth-first search of the state tree and updates X and b as necessary.

The lower bound method is typically one of the clique number lower bound algorithms (e.g. Edwards Elphick). The upper bound method is typically a greedy coloring algorithm (e.g., last-first).

The steps of the main routine are as follows:

1) Construct a graph G' that is isomorphic to G and where each vertex in G' is a set of contracted vertices initialized to a one element set containing the corresponding vertex in G.
2) Run the greedy algorithm and set X and b based on the results.
3) Call the recursive subroutine with G'.
4) Return b or $n(X)$ as the found chromatic number for G and X representing a chromatic coloring of G.

The recursive subroutine is called with a graph H and has access to X and b. The steps are as follows:

1) If $n(H) < b$ then $b = n(H)$.
2) If H is not complete then go to step 6.
3) If $n(H) < n(X)$ then $X = H$.
4) Go to step 11.
5) Determine the chromatic number lower bound for H. If it is greater than or equal to the current upper bound then go to step 11.
6) Select any two nonadjacent vertices u and v in H.
7) Construct $H' = H \cdot uv$, where the vertex set for the new contracted vertex is the union of the vertex sets for u and v.
8) Recursively call this subroutine with H'.
9) Construct $H'' = H + uv$.
10) Recursively call this subroutine with H''.
11) Return.

McDiarmid (1978) [15] used a binomial edge probability module to predict that even with bounding, Zykov algorithms have a runtime complexity of $\mathcal{O}(c^{n\sqrt{\log(n)}})$ for some real number constant $c > 1$, which is worse than exponential. If this is true, then Zykov algorithms will generally perform worse than Christofides-type algorithms.

Corneil and Graham (1973) [10] described and tested a Zykov algorithm that uses a custom lower bound estimation technique for step 5 that they referred to as α-clusters. The full algorithm can be found in Graham (1972) [26]. This technique is rather complex and it isn't clear that it performs any better than the Edwards Elphick algorithm, so Edwards Elphick is used for this research.

A random graph analysis for the a Zykov algorithm using Edwards Elphick is shown in Fig. 65. It measures the mean number of calls to the recursive routine that processes each state and applies the bounding conditions. Thus, the number of calls is essentially the number of states in the state tree.

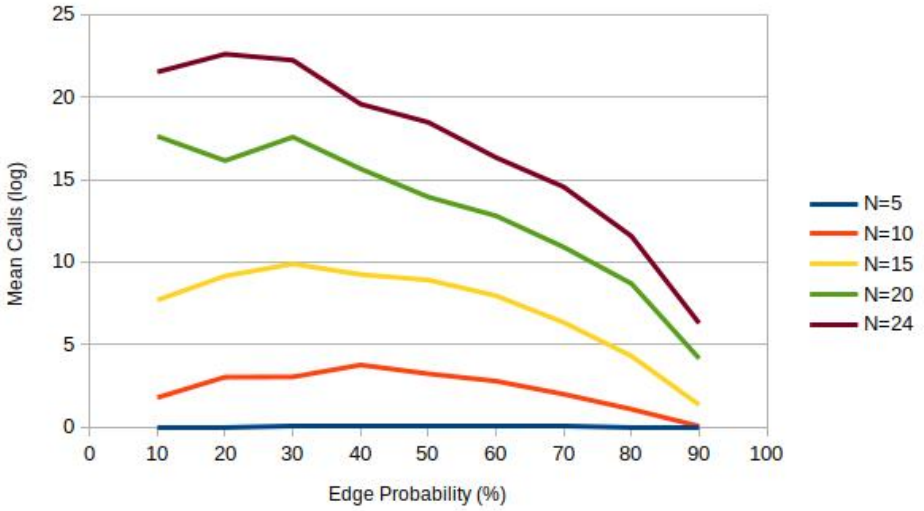


Fig. 65. Zykov algorithm mean number of calls.

The number of calls increases with both graph order but decreases with probability. This is consistent with the facts that the worst case is an empty graph and the best case is

a complete graph. The worst case for each order is assumed to occur at 30% edge probability. A log (base 2) plot of the maximum number of calls for each order at 30% edge probability is shown in Fig. 66. Note that due to excessive runtime duration, the test had to be stopped at $n = 24$. A linear curve fit indicates that the runtime complexity for the Zykov algorithm is about $\mathcal{O}(2^{1.7244n}) \approx \mathcal{O}(3.3^n)$.

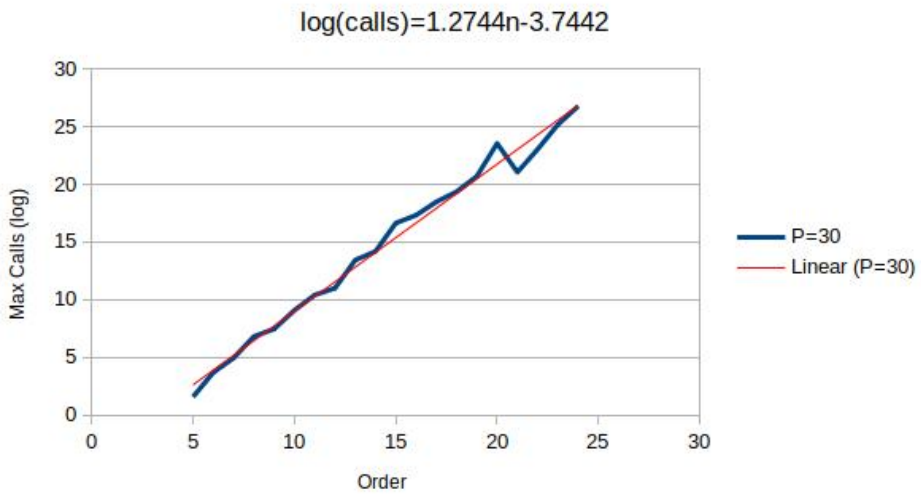


Fig. 66. Zykov algorithm runtime complexity.

Clearly, the Christofides/Wang algorithm outperforms the Zykov algorithm.

5 THE PROPOSED ALGORITHM

It was shown in Section 4 that the Christofides algorithm with the Wang improvements outperforms the Zykov algorithm. But can some additional bounding conditions be added to the Zykov algorithm to close the performance gap? This section proposes a new Zykov-like algorithm that attempts to do just that. An early version of this new algorithm was first introduced by the author and his advisor in collaboration with a team of mechanical engineering researchers from SUNY Buffalo [4].

The major advantages of the Christofides and the Zykov algorithms are that they don't depend on the connectedness of a graph, an example of a chromatic coloring is readily available, and the fact that the algorithms can be coded rather easily to run on a computer. Their major disadvantage is their high runtime complexity, which is inherent to the chromatic number problem.

Thus, the goals of the proposed algorithm are as follows:

1) It should not depend on whether the graph is connected or not.
2) An example of a chromatic coloring should be readily available.
3) It can be easily coded for execution on a computer.
4) It has better runtime performance than the well-known algorithms over the target range of less than 20 nodes with less than 50% edge density.

To accomplish these goals, the proposed algorithm loops on successively higher values of k. For each candidate k value, a graph G is assumed to be k-colorable and a modified version of a Zykov algorithm is executed on G to either prove or disprove this assumption. Since a candidate k value is known, certain reversible steps can be applied to mutate G into simpler graphs with equivalent colorability and test for early termination of the Zykov tree. The first k for which G (or one of its simplifications) is found to be k-colorable is the chromatic number of G.

One slight disadvantage of the proposed algorithm is that whereas the other algorithms readily provide examples of actual chromatic colorings, the proposed algorithm requires an additional step to construct a chromatic coloring: a greedy algorithm with a particular sorting of the vertices. The coloring step is discussed in detail in Section 5.3. Albeit additional work, this extra coloring step is P-time so it does not have a significant impact on the performance of the proposed algorithm.

The proposed algorithm accepts a graph G as input and provides $\chi(G)$ and a chromatic coloring as output and is composed of the following components:

1) A main routine that loops on increasing values of k.
2) The Bron Kerbosch algorithm used by the main routine to determine a lower bound k_{min} for the chromatic number of G (Section 4.1.3).
3) The last-first with color interchange greedy coloring algorithm used by the main routine to determine an upper bound k_{max} for the chromatic number of G and a corresponding k_{max}-coloring of G (Section 4.2).
4) A recursive subroutine that runs a modified Zykov algorithm with additional pruning to determine if G is k-colorable.
5) The Edwards Elphick algorithm used by the recursive subroutine to determine a lower bound for the chromatic number of a graph (Section 4.1.2);
6) A coloring routine called by the main routine to construct a chromatic coloring of G based on the results of the algorithm.

The main routine, recursive subroutine, and coloring routine are summarized in the following sections. A complete description of the theorems that support the various steps in the algorithm and the application of the algorithm to a sample graph then follow.

5.1 The Main Routine

The main routine accepts a graph G as input and returns $\chi(G)$ and a chromatic coloring for G. It initially computes a chromatic number lower bound using Bron

Kerbosch and a chromatic number upper bound using greedy last-first with color interchange. The latter also provides a default coloring. If the lower and upper bounds match then the default coloring is accepted. If not, then the main routine loops on increasing values of k, starting with the lower bound and going no further than the upper bound. For each value of k, the recursive subroutine is called to execute a modified Zykov algorithm in order to determine if G is k-colorable. If k reaches the upper bound then the default coloring is accepted as chromatic.

Instead of walking a single Zykov tree for the whole graph G, the Wang technique (see Section 4.4) is used to mutate G into smaller graphs G_i by selecting a vertex that occurs in the least number of MISs in G. Each of these MISs implies a set of vertex contractions and edge additions that are applied to G to construct the corresponding G_i. Thus, the modified Zykov algorithm is applied in sequence to these separate G_i for each k value. The first successful return identifies $\chi(G)$ and then the coloring routine is called to construct the final coloring based on the information from the successful tree.

The steps of the main routine are as follows:

1) Use the Bron Kerbosch algorithm to compute a chromatic lower bound k_{min} for the input G.
2) Use the greedy last-first with color interchange algorithm to compute a chromatic number upper bound k_{max} and a default k_{max}-coloring for G.
3) Initialize k to k_{min}.
4) If $k_{min} = k_{max}$ then accept the default coloring found in step 2 and go to step 10.
5) Use the Bron algorithm on $\bar{G}$ to find all the MISs in G. Select a vertex that occurs in the smallest number of MISs and then use those MISs to construct n graphs that are the roots of n Zykov trees. Each tree is constructed from G by contracting the vertices in the MIS and adding edges between the contracted vertex and all vertices not in the MIS. Each tree is also associated with an initially empty list of removed

vertices S. The trees are sorted by decreasing MIS length since graphs resulting from larger MISs have fewer choices for branching and thus can be walked more expediently.

6) If $k = k_{max}$ then accept the default coloring found in step 2 and go to step 10.
7) Call the recursive subroutine on each tree to determine if its G is k-colorable. The recursive subroutine accepts the current tree's graph G and removed vertex list S as input and returns a Boolean result R. The called routine may simplify G and may append removed vertices to S. If a tree results in a solution (R =true) then go to step 9.
8) Increment k and go to step 6.
9) Call the coloring routine to construct the final coloring based on the successful tree's final state of G and S.
10) Return the current k as the chromatic number and the found chromatic coloring.

A flowchart of these steps is shown in Fig. 67.

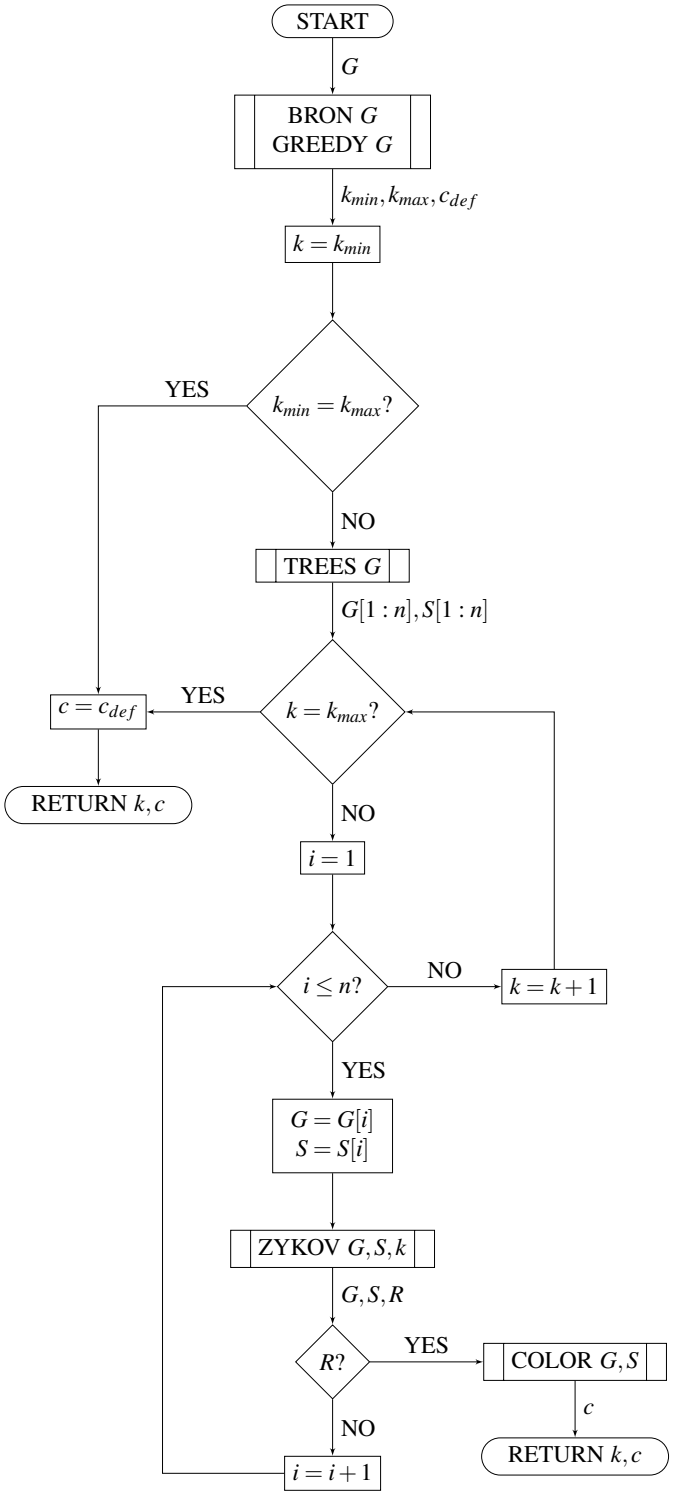


Fig. 67. The proposed algorithm main routine.

Note that in the case of the null or an empty graph, the upper and lower bounds for k will match and the main routine will terminate immediately with the default greedy coloring. Thus, the recursive subroutine is always called with $k \geq 2$. Also note that the main routine is guaranteed to terminate because k will eventually reach k_{max} or the recursive subroutine will return true when a simplification occurs such that $n(G) \leq k$.

5.2 The Recursive Subroutine

The recursive subroutine executes a modified version of the Zykov algorithm to determine whether a graph is k-colorable. It accepts a graph G of order n and size m, a list of already removed vertices S (in removed order), and the target value of $k \geq 2$ as inputs. It returns a possibly simplified version of G, a possibly extended list of removed vertices S, and a boolean value R indicating whether or not G is k-colorable. Internally, various tests are applied to prune the corresponding Zykov tree or abandon it all together based on the current value of k.

The steps of the recursive subroutine and references to their associated theorems are as follows:

1) If $n \leq k$ set R to true and go to step 17 (Theorem 7).
2) Calculate a maximum edge threshold:

$$a = \frac{n^2(k-1)}{2k}$$

3) If $m > a$ then set R to false and go to step 17 (Corollary 8).
4) Construct the set X of all vertices with degree less than k:

$$X = \{v \in V(G) \,|\, \deg(v) < k\}$$

5) If $X \neq \emptyset$ then replace G with $G - X$, append X to S, and go to step 1 (Corollary 9).
6) Calculate the common number of neighbors between each pair of vertices in G, stopping if one vertex's neighborhood is found to be a subset of another.

7) If G has vertices u and v such that $N(u) \subseteq N(v)$ then replace G with $G \cdot uv$ and go to step 1 (Theorem 10).

8) Let b be the smallest number of common neighbors between any pair of vertices in G based on the calculations in step 6:

$$b = \min_{u,v \in V(G)} |N(u) \cap N(v)|$$

9) Calculate an upper bound for the minimum number of common neighbors between any pair of vertices in G:

$$c = n - 2 - \frac{n-2}{k-1}$$

10) If $b > c$ then set R to false and go to step 17. (Corollary 11).

11) Use the Edwards Elphick algorithm to calculate a chromatic number lower bound ℓ for the current state of G.

12) If $\ell > k$ then set R to false and go to step 17.

13) Select two non-adjacent vertices $u, v \in V(G)$ with the smallest number of common neighbors based on the calculations in step 6. It will be shown below that such a pair of vertices is guaranteed to exist.

14) Assume that u and v are assigned the same color by letting $G' = G \cdot uv$. Also make a copy of the removed vertices list $R' = R$. Recursively call this routine using G', S', and k as inputs to see if G' is k-colorable. Note that G' and S' may be modified. If G' is k-colorable then replace $G = G'$ and $S = S'$, set R to true, and go to step 17 (Theorem 21).

15) Assume that u and v are assigned different colors by letting $G' = G + uv$. Also make a copy of the removed vertices list $R' = R$. Recursively call this routine using G', S', and k as inputs to see if G' is k-colorable. Note that G' and S' may be modified. If G'

is k-colorable then replace $G = G'$ and $S = S'$, set R to true, and go to step 17 (Theorem 21).

16) Conclude that G is not k-colorable and set R to false.

17) Return the determine result R, the possibly simplified G, and the possibly extended list of removed vertices S.

A flowchart of these steps is shown in Fig. 68.

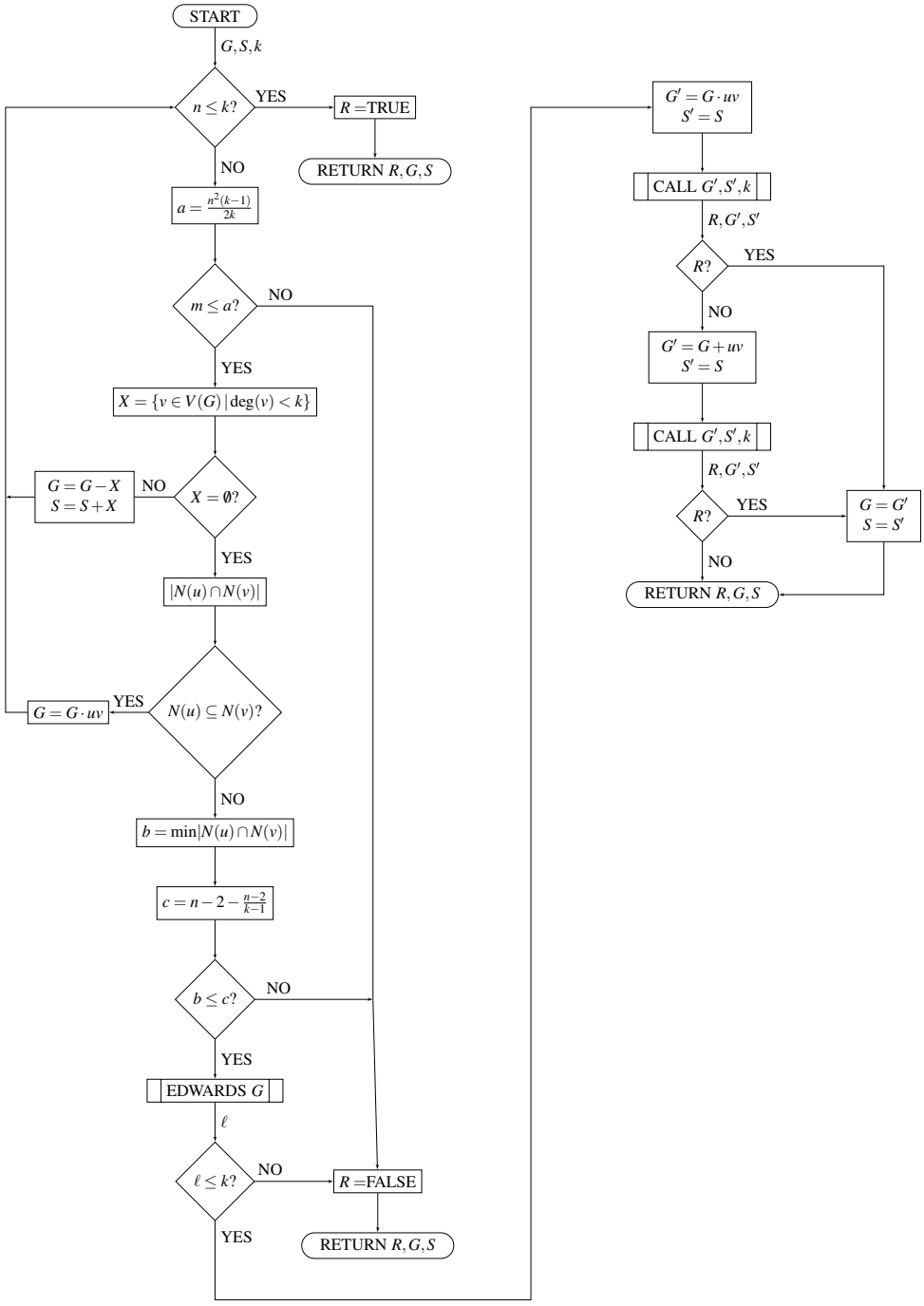


Fig. 68. The proposed algorithm recursive subroutine.

Step 1 is the success condition. Success occurs when G is simplified by removing or contracting sufficent vertices (sub steps 4–7) and the main routine has sufficiently incremented k (main step 8) such that $n \leq k$.

Steps 4–7 attempt to simplify G using vertex removal and contraction in order to achieve a simpler graph that is equivalently k-colorable. Each time vertices are removed or contracted the associated branches in the corresponding Zykov tree are pruned. Since these same steps would just be repeated for $k+1$, the subroutine saves the simplified G as a starting point for the next candidate value of k.

Steps 2–3 and 8–12 apply tests that attempt to disprove that the current state of G is k-colorable for the current value of k. If so, then the current Zykov tree is abandoned and the subroutine returns false.

Steps 13–16 constitute the recursive portion of the modified Zykov algorithm. The recursive calls are guaranteed to terminate because either there will be sufficient vertex contractions such that $n \leq k$, resulting in a true return, or sufficient edge additions such that the graph becomes complete and (as will be shown) is rejected by step 3, resulting in a false return. Note that in the event of a false return, any modifications to the current states of G and S resulting from the recursive calls are not returned to the main routine.

5.3 The Coloring Routine

The recursive subroutine will eventually return true when applied to a particular Zykov tree using a particular value of k. The final state of G, which should be a complete graph of single and/or contracted vertices, and the final state of S, which is a list of removed single and/or contracted vertices in the order removed are used to construct the final chromatic coloring.

According to Theorem 15, there exists some ordering of the vertices such that the greedy algorithm will produce an exact result for the chromatic number of a graph. The removed vertices in reverse order removed is such an ordering. Consider the example

shown in Fig. 69. If $k = 4$ and vertex v is to be removed since $\deg(v) = 3$, then in the reverse direction when v is added the fourth color is available for v. In general, given a graph $G - v$ such that $deg(v) < k$ in G, there will always be an available color for v regardless of how $N(v)$ is colored. This result is formalized in Theorem **??**.

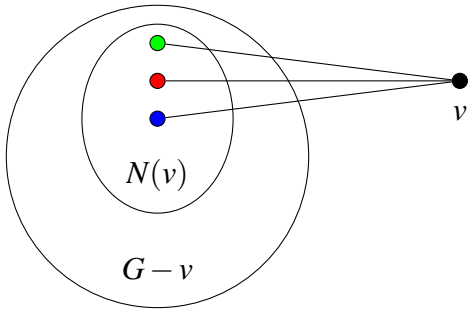
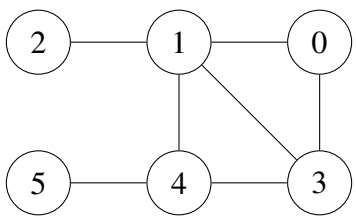


Fig. 69. Coloring a removed vertex example.

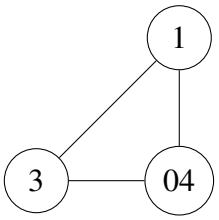
The coloring routine accepts a graph G that is k-chromatic, a complete graph G_i of order k that results from running the modified Zykov algorithm on G, and a removed vertex list R sorted by the order removed as inputs. It outputs a k-chromatic coloring of G. The steps of the coloring routine are as follows:

1) Start with k empty color classes $\{c_1, \ldots, c_k\}$.
2) Order the vertices in G_i into a list. Note that these vertices represent single and/or contracted vertices from G.
3) Append the vertices in S to the list in reverse order. Note that these vertices also represent single and/or contracted vertices from G.
4) Assign all of the vertices from G that are represented by $v_j \in V(G_i)$ to color c_j.
5) Greedy color the remaining vertices in the list. No additional colors should be required. Note that when determining color use by adjacent vertices, all of the single and/or contracted vertices from G represented by a vertex in the sorted list must be checked against all the single and/or contracted vertices from G represented by the previously colored vertices in the sorted list.

Consider the example graph G and its reduced form G_1 in Fig. 70.



G



G_1

S
2
5

Fig. 70. A coloring routine example.

Since $\chi(G) = 3$, start with three empty color classes: $\{c_1, c_2, c_3\}$. Next, list the vertices in G_1, followed by the removed vertices in reverse order: $04, 1, 3, 5, 2$. Now, assign the vertices from G_1 to the color classes:

c_1	$0,4$
c_2	1
c_3	3

Next, color the removed vertices. Since 5 is adjacent to 4 but not to 1, 5 is assigned to c_2. Finally, since 2 is adjacent to 1 but to neither 0 nor 4, 2 is assigned to c_1:

c_1	$0,2,4$
c_2	$1,5$
c_3	3

5.4 Supporting Theorems

This section contains the theorems that support the steps in the recursive subroutine. Remember that the success check of step 1 is already supported by Theorem 7.

5.4.1 Maximum Edge Threshold

The maximum edge threshold test of steps 2 and 3 is supported by Theorem 17.

Theorem 17 (Maximum Edge Threshold)**.** *Let G be a graph of order n and size m and let $k \in \mathbb{N}$. If G is k-colorable then:*

$$m \leq \frac{n^2(k-1)}{2k}$$

Proof. Assume that G is k-colorable. This means that $V(G)$ can be distributed into k independent (some possibly empty) subsets. Call these subsets $A_1, \ldots A_k$ and let $a_i = |A_i|$. Thus, each $v \in A_i$ can be adjacent to at most $n - a_i$ other vertices in G, and hence the maximum number of edges incident to vertices in A_i is given by: $a_i(n-a_i) = na_i - a_i^2$. Now, using Theorem 11, the maximum number of edges in G is given by:

$$m \leq \frac{1}{2}\sum_{i=1}^{k}(na_i - a_i^2)$$

with the constraint:

$$\sum_{i=1}^{k} a_i = n$$

This problem can be solved using the Lagrange multiplier technique. We start by defining:

$$\begin{aligned} F(a_1,\ldots,a_k) &= f(a_1,\ldots,a_k) - \lambda g(a_1,\ldots,a_k) \\ &= \frac{1}{2}\sum_{i=1}^{k}(na_i - a_i^2) - \lambda\sum_{i=1}^{k} a_i \\ &= \sum_{i=1}^{k}\left(\frac{1}{2}na_i - \frac{1}{2}a_i^2 - \lambda a_i\right) \end{aligned}$$

Now, optimize by taking the gradient and setting the resulting vector equation equal to the zero vector:

$$\nabla F = \sum_{i=1}^{k}(\frac{n}{2} - a_i - \lambda)\hat{a}_i = \mathbf{0}$$

This results in a system of k equations of the form:

$$\frac{n}{2} - a_i - \lambda = 0$$

And so:

$$a_i = \frac{n}{2} - \lambda$$

Plugging this result back into the contraint:

$$\sum_{i=1}^{k} a_i = \sum_{i=1}^{k} \left(\frac{n}{2} - \lambda\right) = k\left(\frac{n}{2} - \lambda\right) = n$$

Solving for λ yields:

$$\lambda = \frac{n}{2} - \frac{n}{k}$$

And finally, to get a_i in terms of n and k:

$$a_i = \frac{n}{2} - \left(\frac{n}{2} - \frac{n}{k}\right) = \frac{n}{k}$$

Therefore:

$$m \leq \frac{1}{2}\sum_{i=1}^{k}\left[n\left(\frac{n}{k}\right) - \left(\frac{n}{k}\right)^2\right] = \frac{k}{2}\left(\frac{n^2k - n^2}{k^2}\right) = \frac{n^2(k-1)}{2k}$$

□

The recursive subroutine actually uses the contrapositive of this result, as stated in Corollary 8.

Corollary 8. *Let G be a graph of order n and size m and let $k \in \mathbb{N}$. If:*

$$m > \frac{n^2(k-1)}{2k}$$

then G is not k-colorable.

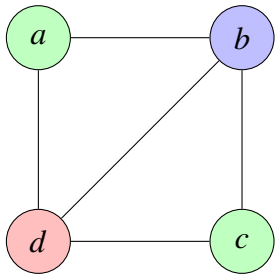
Corollary 8 is demonstrated by Fig. 71. The shown graph G has $n = 4$, $m = 5$, and $\chi(G) = 3$. Testing for $k = 2$:

$$a = \frac{4^2(2-1)}{2 \cdot 2} = 4$$

But $m = 5 > 4 = a$ and so we can conclude that G is not 2-colorable. However, testing for $k = 3$;

$$a = \frac{4^2(3-1)}{2 \cdot 3} = 5.3$$

So $m = 5 \not> 5.3 = a$ and thus G *may* be 3-colorable, since this test only provides a necessary and not a sufficient condition.



G

Fig. 71. Corollary 8 example.

In fact, the the test of Corollary 8 will always fail for a complete graph when $k < n$. Since $k, n > 0$:

$$\begin{aligned}
\frac{n(n-1)}{2} - \frac{n^2(k-1)}{2k} &= \frac{kn(n-1) - n^2(k-1)}{2k} \\
&= \frac{kn^2 - kn - kn^2 + n^2}{2k} \\
&= \frac{n^2 - kn}{2k} \\
&= \frac{n(n-k)}{2k} \\
&> 0 \qquad (n > k)
\end{aligned}$$

5.4.2 Vertex Removal

Steps 4 and 5 remove vertices with degrees less than k. The idea is that since a vertex's neighbors only use less than k colors there will always be an available color for the vertex without adding a new color, regardless of how the vertex's neighbors are actually colored. In order words, these small degree vertices do not affect the overall colorability of their graph. This fact is demonstrated by Fig. 72; no matter which vertex is removed, the resulting subgraph is still properly colored using at most four (in fact, three) colors.

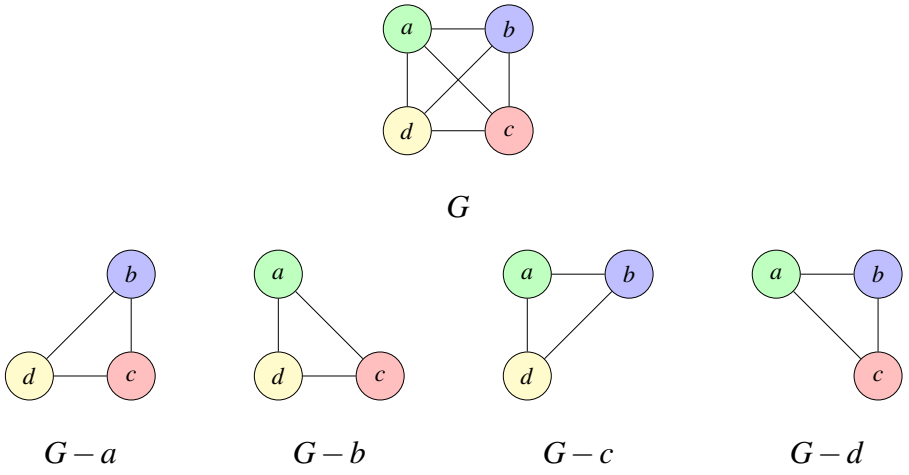


Fig. 72. Vertex removal example.

Vertex removal is supported by Theorem 18:

Theorem 18. *Let G be a graph and let $v \in V(G)$ such that* $\deg(v) < k$ *for some $k \in \mathbb{N}$. G is k-colorable if and only if $G - v$ is k-colorable.*

Proof. Assume that G is k-colorable. By definition, there exists some coloring function $c : V(G) \to C$ where $|C| = k$. Consider the restricted coloring function $c' = c|_{V(G-v)}$ and assume $uw \in E(G-v)$ and hence $u \neq w$. Since c is proper:

$$c'(u) = c(u) \neq c(w) = c'(w)$$

This means that $c' : V(G-v) \to C$ is a proper coloring of $G - v$ with $|C| = k$.

Therefore $G - v$ is k-colorable.

For the converse, assume that $G - v$ is k-colorable. By definition, there exists some coloring function $c : V(G-v) \to C$ where $|C| = k$. By assumption, $\deg(v) < k$, so v has at most $k-1$ neighbors in G, using at most $k-1$ colors. This means that there is an additional color that can be assigned to v in G such that the coloring remains proper (see Fig. 69). So let $N(v) = \{v_1, \ldots, v_r\} \subseteq V(G-v)$ for some $r < k$, and let $c[N(v)] = \{c_1, \ldots, c_s\} \subset C$ for some $s \leq r < k$. Since $c[N(v)]$ is a proper subset of C,

select $c_k \in C - c[N(v)]$ and define a coloring function $c' : V(G) \to C$ as follows:

$$c'(u) = \begin{cases} c(u), & u \neq v \\ c_k, & u = v \end{cases}$$

Now, assume that $uw \in E(G)$ and hence $u \neq w$.

Case 1: $v \notin uw$

So $u, w \in E(G - v)$ and since c is proper:

$$c'(u) = c(u) \neq c(w) = c'(w)$$

Case 2: $v \in uw$

Assume without loss of generality (AWLOG) that $v = u$. Thus, $w \in V(G - v)$ and since c is proper:

$$c'(v) = c_k \neq c(u) = c'(u)$$

This means that $c' : V(G) \to C$ is a proper coloring of G with $|C| = k$.

Therefore $G - v$ is k-colorable. □

Theorem 18 is demonstrated in Fig. 73 for $k = 4$ and $\deg(v) = 3$.

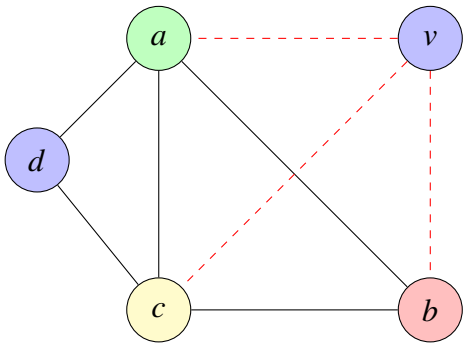


Fig. 73. Theorem 18 example.

The recursive subroutine actually removes all such small degree vertices at once, which is supported by the inductive proof in Corollary 9.

Corollary 9. *Let G be a graph of order n and let $X = \{v \in V(G) \,|\, \deg(v) < k\}$ for some $k \in \mathbb{N}$. G is k-colorable if and only if $G - X$ is k-colorable.*

Proof. (by induction on $|X|$)

Base Case: Let $|X| = 0$.

But $G - X = G$ (trivial case).

Inductive Assumption: Let $|X| = r$.

Assume that G is k-colorable if and only if $G - X$ is k-colorable.

Inductive Step: Consider $|X| = r + 1$.

Since $|X| = r + 1 > 0$, there exists $v \in X$ such that $\deg(v) < k$. Let $Y = X - \{v\}$ and note that $|Y| = |X| - 1 = (r+1) - 1 = r$. So, G is k-colorable if and only if $G - v$ is k-colorable (Theorem 18) if and only if $(G - v) - Y$ is k-colorable (inductive assumption).

Therefore, by the principle of induction, G is k-colorable if and only if $G - X$ is k-colorable. □

Returning to the example in Fig. 73, note that $X = \{v, b, d\}$ is the set of all vertices with degree less than 4 and so all three can be removed at once in accordance with Corollary 9.

5.4.3 Neighborhood Subsets

Step 7 contracts vertices whose neighborhoods are subsets of other vertices. Lemma 2 states that in any k-coloring such vertices can be colored using the same color.

Lemma 2. *Let G be a graph such that:*

1) $n(G) \geq 2$

2) $u, v \in V(G)$ and $u \neq v$

3) $N(u) \subseteq N(v)$*, and hence* $uv \notin E(G)$

and let $k \in \mathbb{N}$*. In any k-coloring of G, u and v can be assigned the same color.*

Proof. Assume that the coloring function $c : V(G) \to C$ is proper where $|C| = k$. Since $N(u) \subseteq N(v)$, it must be the case that $c[N(u)] \subseteq c[N(v)]$. But $c(v) \notin c[N(v)]$ and hence $c(v) \notin N(u)$. Therefore, $c(v)$ is available for u. □

Theorem 19 states that any two vertices using the same color can be contracted without affecting colorability.

Theorem 19. *Let G be a graph such that:*

1) $n(G) \geq 2$

2) $u, v \in V(G)$ *and* $u \neq v$

3) u and v are assigned the same color in any proper coloring of G

and let $k \in \mathbb{N}$*. G is k-colorable if and only if* $G \cdot uv$ *is k-colorable.*

Proof. Assume that G is k-colorable. By definition, there exists some coloring function $c : V(G) \to C$ where $c(u) = c(v)$ and $|C| = k$. Let $w \in V(G \cdot uv)$ be the identified vertex and define a coloring function $c' : V(G \cdot uv) \to C$ as follows:

$$c'(z) = \begin{cases} c(z), & z \neq w \\ c(u) = c(v), & z = w \end{cases}$$

Now, assume that $xy \in E(G \cdot uv)$ and hence $x \neq y$.

Case 1: $w \notin xy$

So $x, y \in E(G)$ and since c is proper:

$$c'(x) = c(x) \neq c(y) = c'(y)$$

Case 2: $w \in xy$

Assume without loss of generality (AWLOG) that $w = x$ and so $w \neq y$. Thus, $y \in V(G)$, $y \notin \{u, v\}$, and since c is proper:

$$c'(w) = c(u) \neq c(y) = c'(y)$$

This means that $c' : V(G \cdot uv) \to C$ is a proper coloring of $G \cdot uv$ with $|C| = k$.

Therefore $G \cdot v$ is k-colorable.

For the converse, assume that $G \cdot uv$ is k-colorable and let $w \in V(G \cdot uv$ be the identified vertex. By definition, there exists some coloring function $c : V(G \cdot uv) \to C$ where $|C| = k$. Define the coloring function $c' : V(G) \to C$ as follows:

$$c'(z) = \begin{cases} c(z), & z \notin \{u, v\} \\ c(w), & z \in \{u, v\} \end{cases}$$

Now, assume that $xy \in E(G)$ and hence $x \neq y$.

Case 1: $u, v \notin xy$

So $x, y \in E(G \cdot uv)$ and since c is proper:

$$c'(x) = c(x) \neq c(y) = c'(y)$$

Case 2: $u \in xy$ and $v \notin xy$ or $u \notin xy$ and $v \in xy$

Assume without loss of generality (AWLOG) that $u = x$ and $v \neq y$. Thus, $y \in V(G \cdot uv)$, $y \neq w$, and since c is proper:

$$c'(u) = c(w) \neq c(y) = c'(y)$$

This means that $c' : V(G) \to C$ is a proper coloring of G with $|C| = k$.

Therefore G is k-colorable. □

Finally, Corollary 10 combines the previous two results.

Corollary 10. *Let G be a graph such that:*

1) $n(G) \geq 2$

2) $u, v \in V(G)$ *and* $u \neq v$

3) $N(u) \subseteq N(v)$*, and hence* $uv \notin E(G)$

and let $k \in \mathbb{N}$*. G is k-colorable if and only if $G \cdot uv$ is k-colorable.*

Proof. By Lemma 2, u and v can be assigned the same color. Therefore, by Theorem 19, G is k-colorable if and only if $G \cdot uv$ is k-colorable. □

Corollary 10 is demonstrated in Fig. 74.

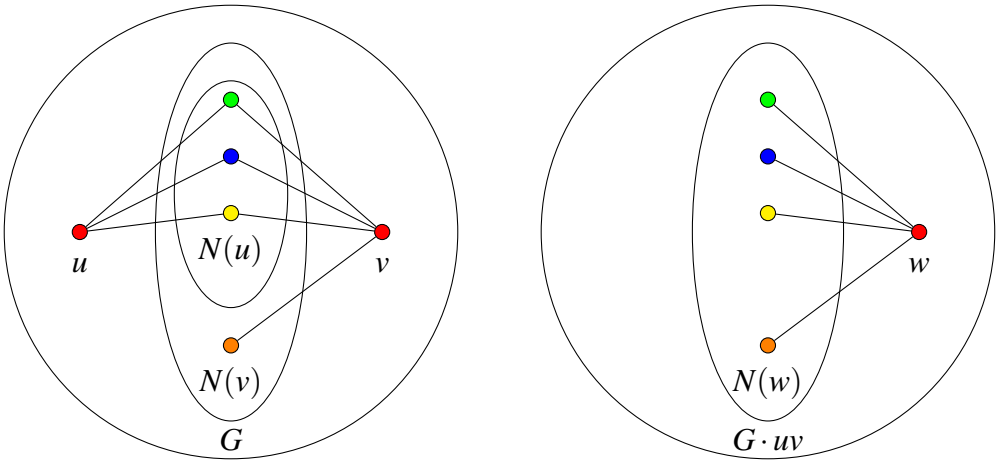


Fig. 74. Demonstration of Theorem 10.

In the original version of the proposed algorithm, if $N(u) \subset N(v)$ then G was replaced by $G - u$. Although technically correct, simply removing u loses the important information that u and v are assigned the same color, which would make the coloring routine more complicated. Special thanks to Graham [26] for pointing out that contraction is sufficient.

5.4.4 Minimum Common Neighbor Upper Bound

Steps 8–10 establish an upper bound for the minimum common neighbor count between any two vertices in a graph that is assumed to be k-colorable. This limit is dependent on the following facts that are guaranteed by previous steps:

1) $2 \leq k < n$
2) There are no $u, v \in V(G)$ such that $N(u) \subseteq N(v)$

The supporting theorem uses these facts along with Lemma 3 in its proof.

Lemma 3. *Let G be a graph and let S be a non-empty independent subset of $V(G)$. If there exists a vertex $v \in S$ such that v is adjacent to all vertices in $V(G) - S$ (i.e., $N(v) = V(G) - S$) then for all vertices $u \in S$ it is the case that $N(u) \subseteq N(v)$.*

Proof. Assume that such a v exists and then assume that $u \in S$. If $u = v$ then (trivially) $N(v) = N(v)$, so assume $u \neq v$. Furthermore, since $u, v \in S$ and S is independent (by assumption), it must be the case that u and v are not neighbors.

Case 1: $N(u) = \emptyset$.

Therefore, by definition, $N(u) = \emptyset \subseteq N(v)$.

Case 2: $N(u) \neq \emptyset$.

Assume that $w \in N(u)$. This means that w is adjacent to u and hence $w \notin S$, since S is an independent set. So $w \in V(G) - S$ and thus, by assumption, v is adjacent to w and we can conclude that $w \in N(v)$. Therefore $N(u) \subseteq N(v)$.

Therefore, for all $u \in S$, $N(u) \subseteq N(v)$. □

Lemma 3 is demonstrated in Fig. 75. Note that since $v \in S$ is adjacent to every vertex in $V(G) - S$, vertex $u \in S$ can't help but be adjacent to some subset of $N(v)$.

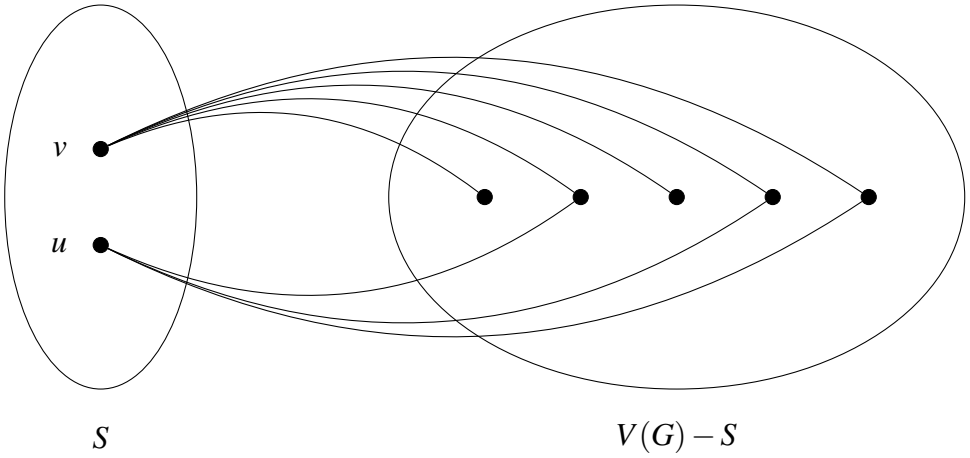


Fig. 75. Lemma 3 example.

Theorem 20 establishes the desired upper bound.

Theorem 20. *Let G be a graph of order n and size m such that there are no $u, v \in V(G)$ where $N(u) \subseteq N(v)$, and let $k \in \mathbb{N}$ such that $2 \leq k < n$. If G is k-colorable then there exists two vertices $w, z \in V(G)$ such that:*

$$|N(w) \cap N(z)| \leq n - 2 - \frac{n-2}{k-1}$$

Proof. Assume that G is k-colorable. This means that $V(G)$ can be distributed into k independent (some possibly empty) subsets $A_1, \ldots, A_k$ such that $a_i = |A_i|$ and $a_1 \geq a_2 \geq \cdots \geq a_k$. Since $n > k$, by the pigeonhole principle, it must be the case that $a_1 \geq 2$. Assume that $v \in A_1$.

First, assume by way of contradiction (ABC) that v is adjacent to all other vertices in $V(G) - A_1$. Since $a_1 \geq 2$, there exists $u \in A_1$ such that $u \neq v$ and u is not adjacent to v. Thus, by Lemma 3, $N(u) \subseteq N(v)$, contradicting the assumption. Note that this contradiction also eliminates the degenerate case where $A_1 = V(G)$; however, this case does not occur here because the graph would be an empty graph and would have been

eliminated by previous steps. Therefore, there exists some $v' \in V(G) - A_1$ such that v is not adjacent to v'. Assume that $v' \in A_i$ for some i such that $1 < i \leq k$:

Case 1: $a_i = 1$

By the pigeonhole principle:

$$a_1 \geq \left\lceil \frac{n-1}{k-1} \right\rceil \geq \frac{n-1}{k-1}$$

Now, assume by way of contradiction (ABC) that v' is adjacent to all vertices in $V(G) - A_1 - A_i$ and assume $u \in N(v)$. Then it must be the case that $u \in V(G) - A_1 - A_i$, and so u is adjacent to v', and thus $u \in N(v')$. Therefore $N(v) \subseteq N(v')$, which contradicts the assumption. This situation is demonstrated by Fig. 76.

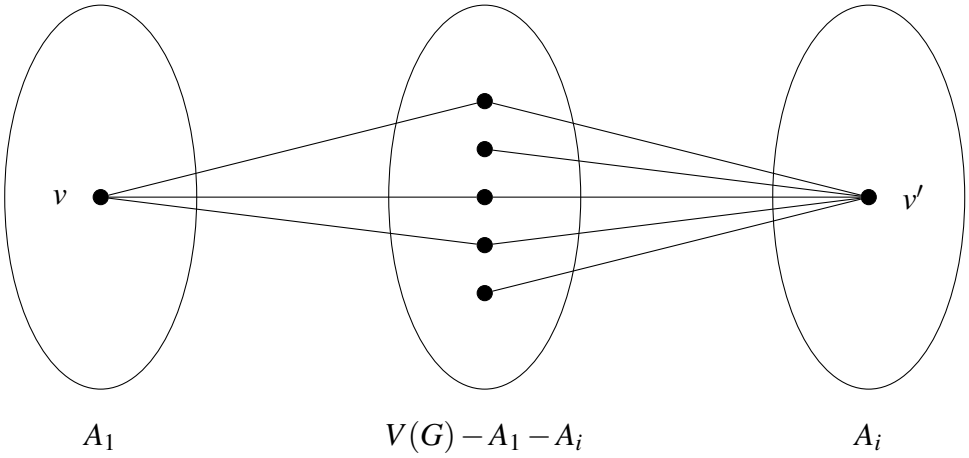


Fig. 76. Case $a_i = 1$ contradiction.

So there must exist some $u \in V(G) - A_1 - A_i$ such that u is not adjacent to v'. This results in the upper bound:

$$|N(v) \cap N(v')| \leq n - |\{u, v'\}| - a_1 \leq n - 2 - \frac{n-1}{k-1}$$

Note that since $v \in A_1$, it is already counted in a_1. Comparing this bound to the desired bound:

$$\left(n-2-\frac{n-2}{k-1}\right)-\left(n-2-\frac{n-1}{k-1}\right)=\frac{(n-1)-(n-2)}{k-1}=\frac{1}{k-1}>0$$

for $k \geq 2$. Thus the new bound is tighter and so:

$$|N(v)\cap N(v')| \leq n-2-\frac{n-1}{k-1} \leq n-2-\frac{n-2}{k-1}$$

Case 2: $a_i = 2$

By the pigeonhole principle:

$$a_1 \geq \left\lceil \frac{n-2}{k-1} \right\rceil \geq \frac{n-2}{k-1}$$

This results in the upper bound:

$$|N(v)\cap N(v')| \leq n-a_i-a_1 \leq n-2-\frac{n-2}{k-1}$$

Case 3: $a_i \geq 3$

By the pigeonhole principle:

$$a_1 \geq \left\lceil \frac{n-3}{k-1} \right\rceil \geq \frac{n-3}{k-1}$$

This results in the upper bound:

$$|N(v)\cap N(v')| \leq n-a_i-a_1 \leq n-3-\frac{n-3}{k-1}$$

Comparing this to the desired bound:

$$\left(n-2-\frac{n-2}{k-1}\right)-\left(n-3-\frac{n-3}{k-1}\right)=1-\frac{(n-3)-(n-2)}{k-1}=1-\frac{1}{k-1}>0$$

for $k \geq 2$. Thus the new bound is tighter and so:

$$|N(v) \cap N(v')| \leq n-3-\frac{n-3}{k-1} \leq n-2-\frac{n-2}{k-1}$$

Therefore, there exists $w, v \in V(G)$ such that:

$$|N(w) \cap N(z)| \leq n-2-\frac{n-2}{k-1}$$

□

The recursive subroutine actually uses the contrapositive of this result, as stated in Corollary 11.

Corollary 11. *Let G be a graph of order n and size m such that there are no $u, v \in V(G)$ where $N(u) \subseteq N(v)$, and let $k \in \mathbb{N}$ such that $2 \leq k < n$. If for all $w, z \in V(G)$ it is the case that:*

$$|N(w) \cap N(z)| > n-2-\frac{n-2}{k-1}$$

then G is not k-colorable.

Corollary 11 is demonstrated in Fig. 77. The shown graph has $n = 5$, is 3-chromatic, and has:

$$\min_{u,v \in V(G)} |N(u) \cap N(v)| = 1$$

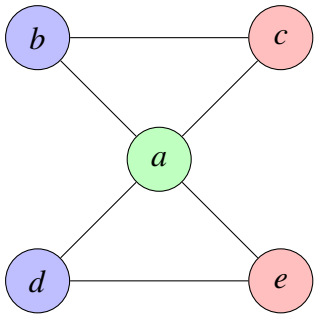
Testing for $k = 2$:

$$5-2-\frac{5-2}{2-1} = 0$$

But $1 > 0$ and so we can conclude that G is not 2-colorable. However, testing for $k = 3$:

$$5-2-\frac{5-2}{3-1} = \frac{3}{2}$$

So $1 \not> \frac{3}{2}$ and thus G *may* be 3-colorable, since this test only provides a necessary and not a sufficient condition.



G

Fig. 77. Corollary 11 example.

5.4.5 Recursive Steps

If nothing more can be done in the preceding steps then steps 13–16 revert to branching. Step 13 selects two non-adjacent vertices with the smallest number of common neighbors. Such a pair must exist. Otherwise, the current state of G is complete, which would have been eliminated by step 3. The first recursive call (step 14) assumes that the two selected vertices have the same color, so they are contracted. The second recursive call (step 15) assumes that the two selected vertices have different colors, so they are joined by an added edge. Each call starts a new branch of the Zykov tree corresponding to the current value of k. If either call returns true then it can be concluded that the input graph was indeed k-colorable. Otherwise, it can be concluded that the input graph is not k-colorable and the recursive subroutine returns the state of G prior to the recursive calls to the main routine.

These steps are supported by Theorem 21.

Theorem 21. *Let G be a graph of order $n \geq 2$ and let $u, v \in G$ such that u and v are not adjacent. G is k-colorable if and only if $G \cdot uv$ or $G + uv$ is k-colorable.*

Proof. Assume that G is k-colorable. By definition, there exists some coloring function $c : V(G) \to C$ where $|C| = k$. There are two possibilities, corresponding to the two recursive choices: $c(u) = c(v)$ (same color) or $c(u) \neq c(v)$ (different colors). The case $c(u) = c(v)$ has already been proven by Theorem 19, so it remains to be proven that G is k-colorable if and only if $G + uv$ is k-colorable.

Assume $c(u) \neq c(v)$. By adding edge uv, u and v become adjacent and thus must have different colors. Thus, u and v can retain their same colors, the coloring is unchanged and remains proper, and therefore $G + uv$ is k-colorable.

For the converse, assume that $G + uv$ is k-colorable. Since u and v are adjacent in $G + uv$, they must have different colors. Once uv is removed in G, u and v are no longer adjacent and so there are no requirements on their colors. Thus, u and v can retain their same colors, the coloring is unchanged and remains proper, and therefore G is k-colorable. □

5.5 An Example

In this section, the proposed algorithm will be applied to the Grötzsch graph G with $n = 11$, $m = 20$, and $\chi(M) = 4$ (see Section 4.1.1) shown in Fig. 78.

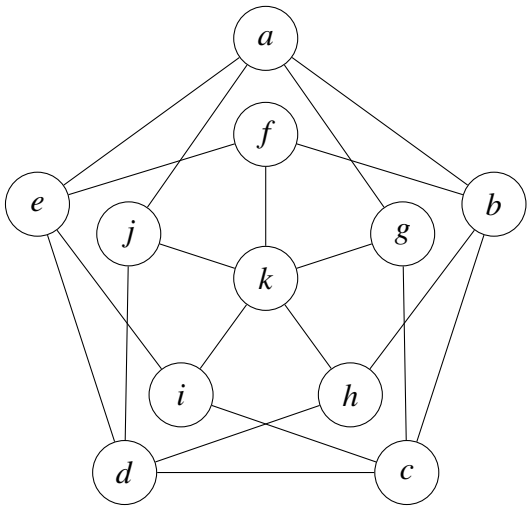


Fig. 78. The Grötzsch example: input graph.

The main routine first executes the Bron Kerbosch algorithm and finds that the chromatic number lower bound is $k_{min} = 2$. It then executes the greedy algorithm and finds that the chromatic number upper bound is $k_{max} = 4$ with the following default coloring:

$\{0, 2, 10\}$
$\{1, 3, 6, 8\}$
$\{4, 5, 9\}$
$\{7\}$

Since the lower and upper bounds do not match, the main routine continues. The Bron Kerbosch algorithm is executed on the complement of G to discover than G has 16 MISs. Vertex a is found to occur in the least number of these MISs: 5 times. These 5 selected MISs, sorted by decreasing length, are as follows:

$\{0, 2, 5, 7\}$
$\{0, 4, 5, 9\}$
$\{0, 5, 7, 9\}$
$\{0, 2, 10\}$
$\{0, 4, 10\}$

The main routine creates 5 Zykov trees, one per selected MIS. For brevity, the remainder of this example will focus on the first tree; assume that the other 4 trees are processed similarly and that none of them produces a solution. The outer loop initializes $k = 2$ and since $k = 2 < 4 = k_{max}$, the recursive subroutine is called on the first tree with $k = 2$.

The starting graph for the first tree is constructed by contracting all of the vertices in the first MIS and then adding edges between the contracted vertex and all other vertices. The result is shown in Fig. 79.

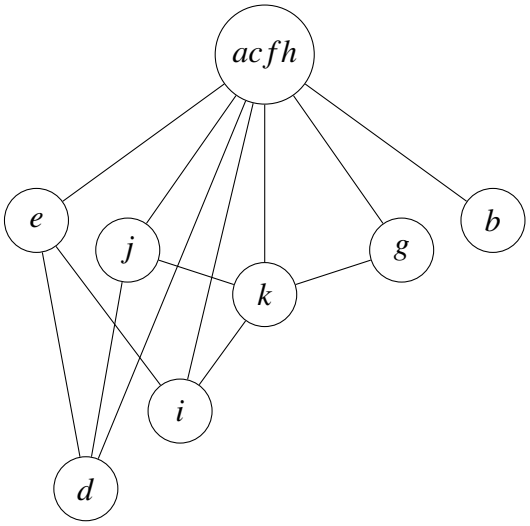


Fig. 79. The Grötzsch example: first tree initial graph.

Since $n = 8 \geq 2 = 4$, the called routine continues and calculates a maximum edge threshold of $a = 16$. Since $m = 13 <= 16 = a$ the called routine continues and finds that $\deg(b) = 1 < 2 = k$ and so vertex b is removed and added to the removed vertex list R. The resulting graph is shown in Fig. 80.

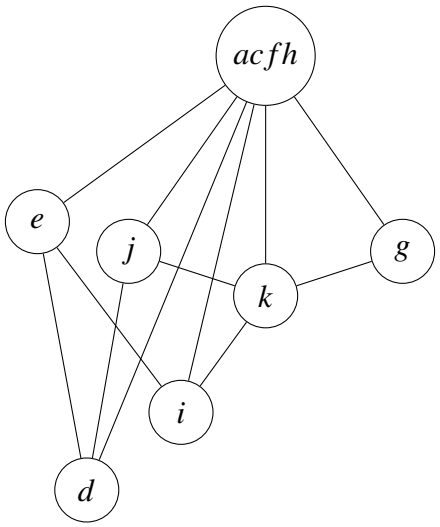


Fig. 80. The Grötzsch example: vertex b removed.

Now, $n = 7 \geq 2 = k$ so the called routine continues. The new maximum edge threshold is $a = 12.25$ and since $m = 12 \leq 12.25 = a$ the called routine continues. There are no more small degree vertices; however it is found that $N(g) \subseteq N(j)$ and so these two vertices are contracted. The resulting graph is shown in Fig. 81.

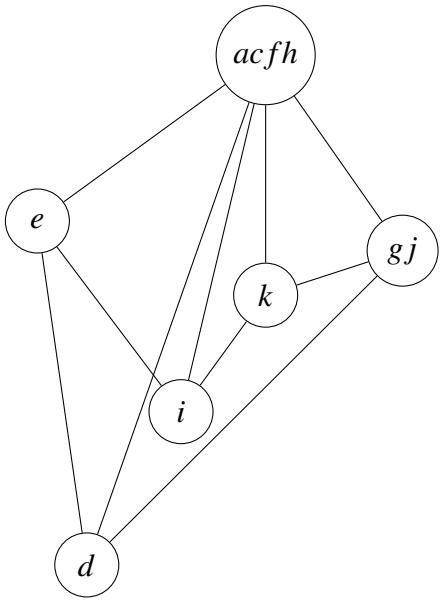


Fig. 81. The Grötzsch example: vertices g and j contracted.

Now, $n = 6 \geq 2 = k$ so the called routine continues. The new maximum edge threshold is $a = 9$ and since $m = 10 > 9 = a$ the maximum edge check fails. The called routine returns with false, the current graph, and a removed vertex list of $R = (b)$. The remaining 4 trees similarly fail, so the main routine increments $k = 3$ and since $k = 3 < 4 = k_{max}$ the recursive subroutine is recalled on the first tree.

Since $n = 6 > 3 = k$ the called routine continues. The new maximum edge threshold is $a = 12$ and since $m = 10 <= 12 = a$ the called routine continues. No small degree vertices or neighborhood subsets are found, so vertices d and e are selected as having the smallest number of common neighbors: $b = 1$. The minimum common neighbors upper bound is calculated to be $c = 2$ and since $b = 1 < 2 = c$ the called routine continues. The

Edwards Elphick algorithm is executed to determine that the new chromatic number lower bound is 3, and since $3 <= 3 = k$, the called routine continues.

At this point all bounding efforts have failed so it is time to start branching. Since all nonadjacent vertices share 2 common neigbors, vertices e and k are selected and are contracted. The resulting graph is shown in Fig. 82. Copies are made of this graph and the removed vertex list and a recursive call is made with these values.

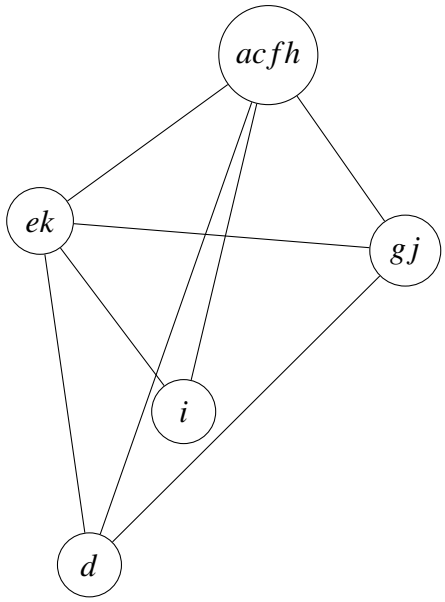


Fig. 82. The Grötzsch example: vertices e and k contracted.

Now, $n = 5 \geq 3 = k$ so the called routine continues. The new maximum edge threshold is $a = 8.33$ and since $m = 8 \leq 8.33 = a$ the called routine continues. Since $deg(i) = 2 < 3 = k$, vertex i is removed and the removed vertex list is now $R = (b, i)$. The resulting graph is shown in Fig. 83.

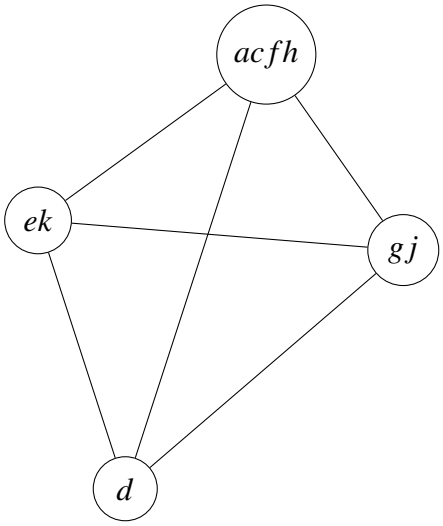


Fig. 83. The Grötzsch example: vertex i removed.

The current state is now a leaf node with a complete graph of order 4. Since $n = 4 > 3 = k$ the called routine continues; however, the new maximum edge threshold is $a = 5.33$ and since $m = 6 > 5.33$ the edge threshold test fails as expected and the recursive call returns false. Modifications to the graph and removed vertex list resulting from the contraction are discarded and the graph of Fig. 81 remains the current graph for the first tree with a removed vertex list of just $R = (b)$.

For the next branch, the edge *ek* is added. The resulting graph is shown in Fig. 84. Copies are made of this graph and the removed vertex list and a recursive call is made with these values.

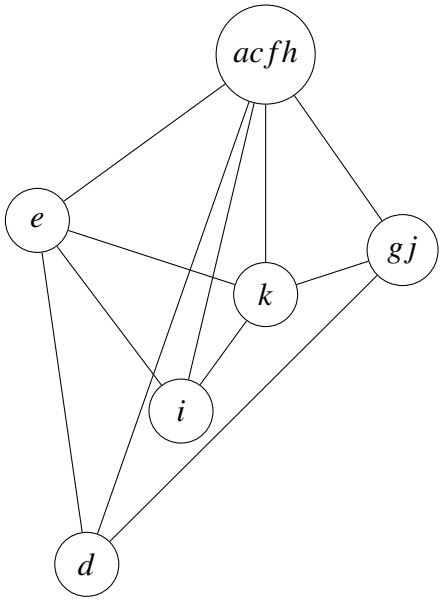


Fig. 84. The Grötzsch example: edge *ek* added.

Now, $n = 6 > 3 = k$ so the called routine continues and calculates the new maximum edge threshold $a = 12$. Since $m = 11 < 12 = a$ the called routine continues. No small degree vertices are found; however, it is found that $N(gj) \subseteq N(e)$, so these vertices are contracted. The resulting graph is shown in Fig. 85.

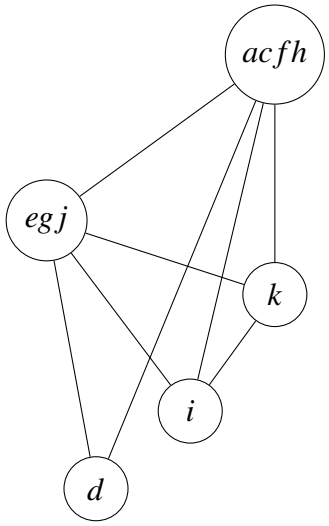


Fig. 85. The Grötzsch example: vertices gj and e contracted.

Now, $n = 5 > 3 = k$ so the called routine continues and calculates the new edge threshold $a = 8.33$. Since $m = 8 < 8.33 = a$ the called routine continues and finds that $\deg(d) = 2 < 3 = k$ so vertex d is removed. The resulting graph is shown in Fig. 86 and the new removed vertex list is now $R = (b, d)$.

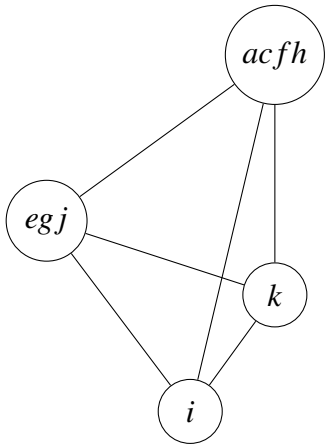


Fig. 86. The Grötzsch example: vertex d removed.

Once again, a leaf node with a complete graph of order 4 is achieved. Since $n = 4 > 3 = k$ the called routine continues. The new maximum edge threshold is $a = 5.33$ and since $m = 6 > 5.33 = a$ the edge threshold test fails as expected. All of the changes to the graph and removed vertex list applied by this branch are discarded and the recursive call returns false.

At this point, both branches have failed and so the called routine returns false to the main routine with the graph shown in Fig. 81 and a removed vertex list of just $R = (b)$. The remaining 4 trees similarly fail, so the main routine increments $k = 4$ and since $k = 4 = 4 = k_{max}$ the default 4-coloring is accepted as the chromatic coloring for the input graph.

6 RANDOM GRAPH ANALYSIS

This section describes the random graph analysis used in this research. The testbed, custom graph software, and the results obtained for the proposed algorithm are discussed. It finishes with a runtime duration comparison between the Christofides/Wang, the Zykov, and the proposed algorithm in the target range of 20 vertices and moderate edge density.

6.1 The Testbed

All of the random graph analysis in this research was performed on an Acer Aspire running 64-bit Ubuntu Linux. The system contains 8 i7-7780 3.6GHz cores and 20Gb of memory. The graphing software was custom written in C++. Although many present day researchers may choose Java or Python, the author feels that the interpretive nature of those languages combined with unpredictable garbage collection leads to overly inflated and inconsistent results.

The representation of a graph in memory needs to convey the list of vertices, the list of edges, and the connection matrix. Graph mutations that change the number of vertices are complex because the connection matrix must be rebuilt from the altered vertex and edge lists. Thus, the testbed software assumes that graphs are invariant with respect to vertex removals and contractions; if vertex removal or contraction is desired then a new graph instance must be constructed. Edge additions are innocuous: the new edge can be marked in the connection matrix and appended to the edge list.

The general layout of a graph instance in memory is shown in Fig. 87. The vertex and edge lists are vectors of instances that contain the needed vertex and edge attibutes. In a particular graph, a vertex or edge is identified by its position in its list, called its number: vertex numbers are from 0 to $n-1$ and edge numbers are from 0 to $m-1$. The connection matrix is a two-dimensional matrix indexed by vertex numbers: entry $M[i,j]$ returns a list of edge numbers that can be used to index the edge list to locate the edges. If the edge list

is empty then the two vertices are not adjacent. Note that for simple graphs, each edge list has at most one entry and the connection matrix is symmetric: $M[i, j] = M[j, i]$.

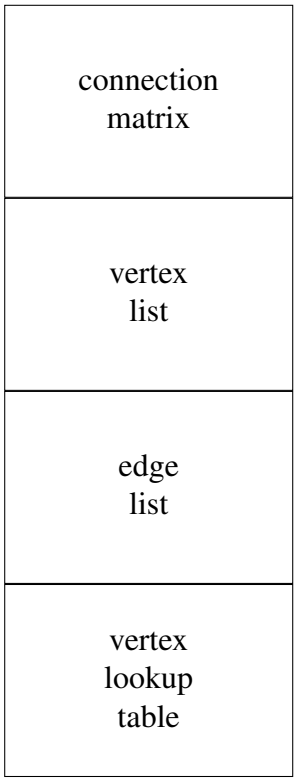


Fig. 87. Graph layout in memory.

One of the complications of graph mutations is that the vertex and edge numbers change when a new graph is constructed. This is why graph mutations result in new instances. In order to be able to identify vertices across graph mutations, each vertex is assigned a unique vertex ID when a new graph is created. Vertices keep their original IDs across graph mutations, although their numbers may change. Thus, edges in the edge list refer to their endpoint vertices by ID and a vertex ID to vertex number lookup table is included as part of the graph schema.

Contracted vertices are assigned new vertex IDs when created; however, graph algorithms like the proposed algorithm may need to remember the original vertices that

were contracted. To support this, each vertex instance includes a list of contracted vertex IDs. When two non-contracted vertices are contracted, the new contracted vertex has a contracted vertex ID list consisting of the original two vertex IDs. When contracted vertices are are contracted, their contracted vertex ID lists are concatenated. Thus, contracted vertex ID lists contain only vertex IDs from the original graph.

One of the most important parameters used by graph algorithms is vertex degree. As was explained in Section 2.11, the degree of vertex v_i can be calculated by summing the i^{th} row or column in the connection matrix. So that degrees do not have to be recalculated each time they are used, the connection matrix precalculates vertex degrees upon graph creation: each time an edge is added the degrees of the endpoint vertices are incremented. The minimum and maximum degrees for a graph still need to be recalculated each time, so algorithms should cache those values once fetched.

Since new graphs may be created many times during the execution of a graph algorithm, graph construction must be as efficient as possible. The testbed software creates a new graph using the following steps:

1) Each of the n vertices are added to the new vertex list and a corresponding entry is added to the vertex ID to number lookup table.
2) An empty $n \times n$ connection matrix is allocated with all vertex degrees set to 0.
3) Each edge is added to the edge list. The endpoint vertex IDs are found in the lookup table and the edge is registered in the connection matrix, which increments the degrees for the endpoint vertices.

Thus, new graph creation has runtime complexity $\mathcal{O}(n+m)$.

6.2 Runtime Complexity Results

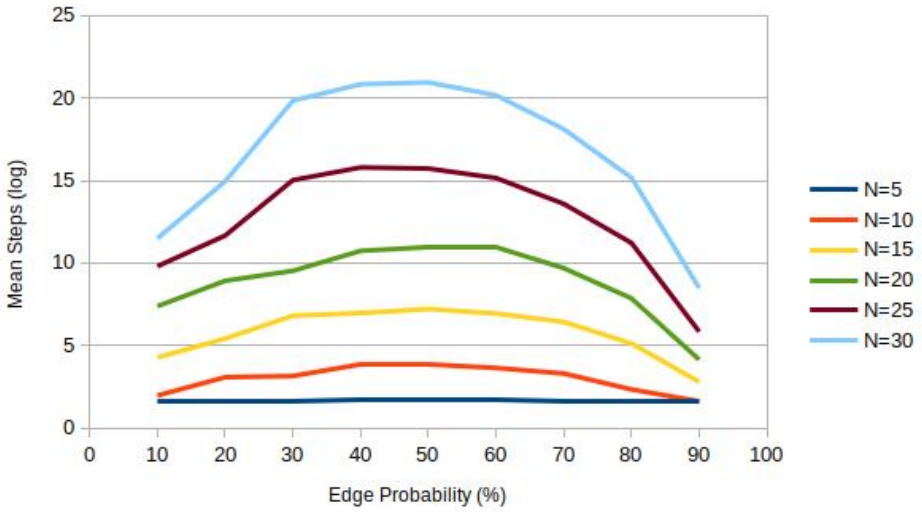


Fig. 88. Proposed algorithm mean number of steps.

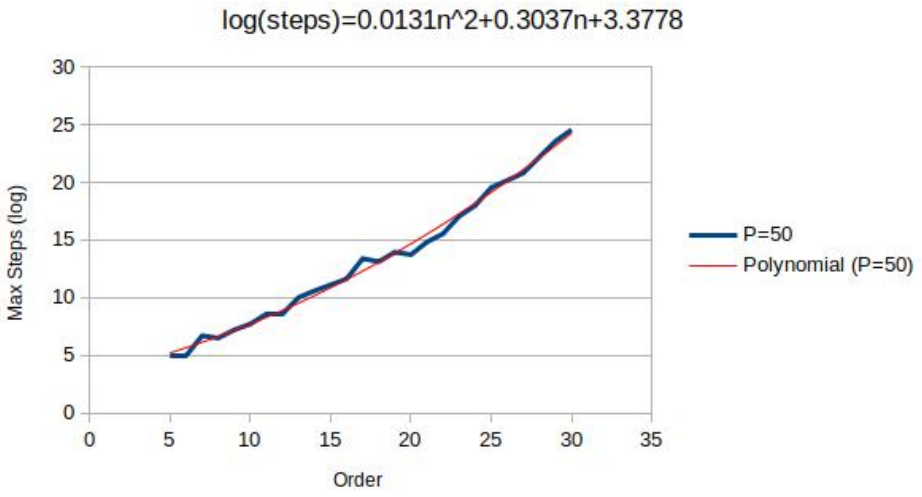


Fig. 89. Proposed algorithm runtime complexity.

6.3 Bounding Test Results

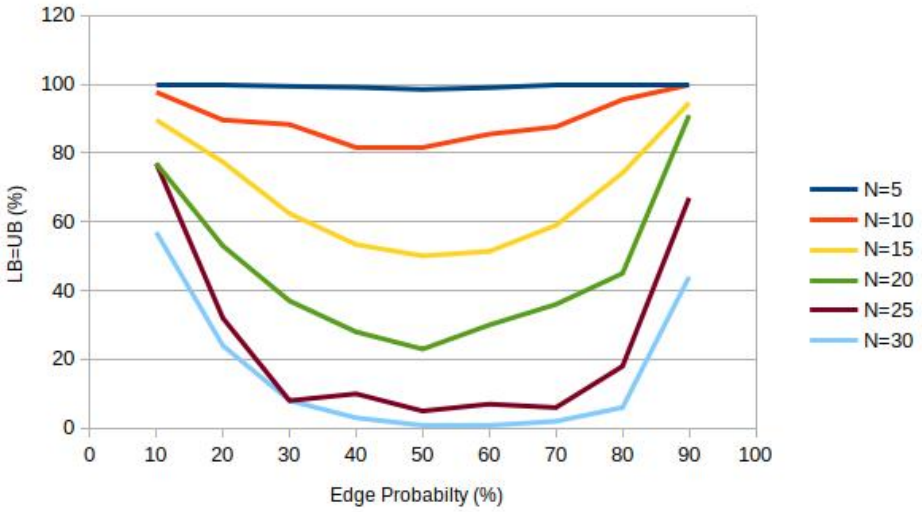


Fig. 90. Proposed algorithm lower/upper bound matching test.

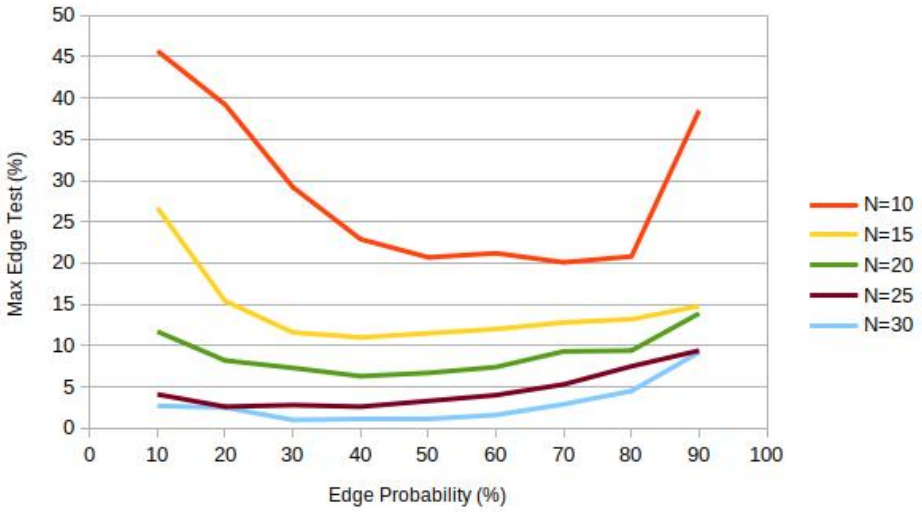


Fig. 91. Proposed algorithm maximum edge threshold test.

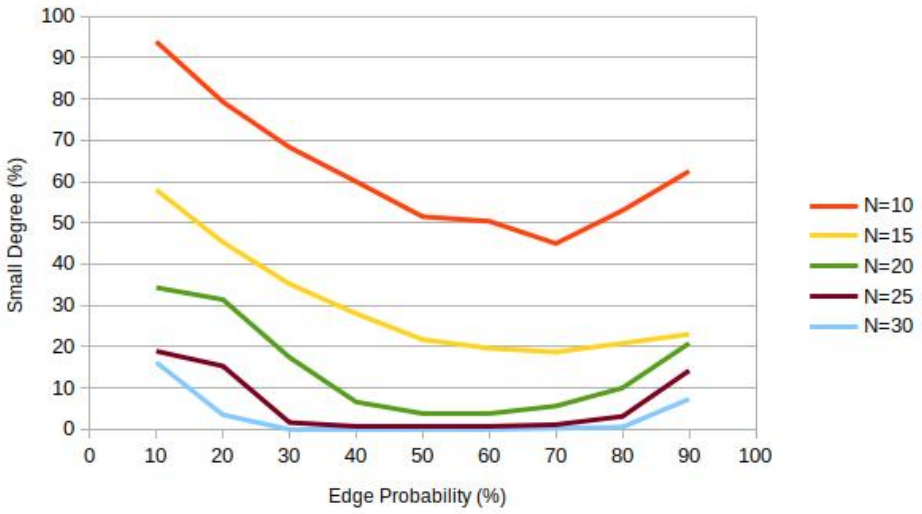


Fig. 92. Proposed algorithm small degree vertex test.

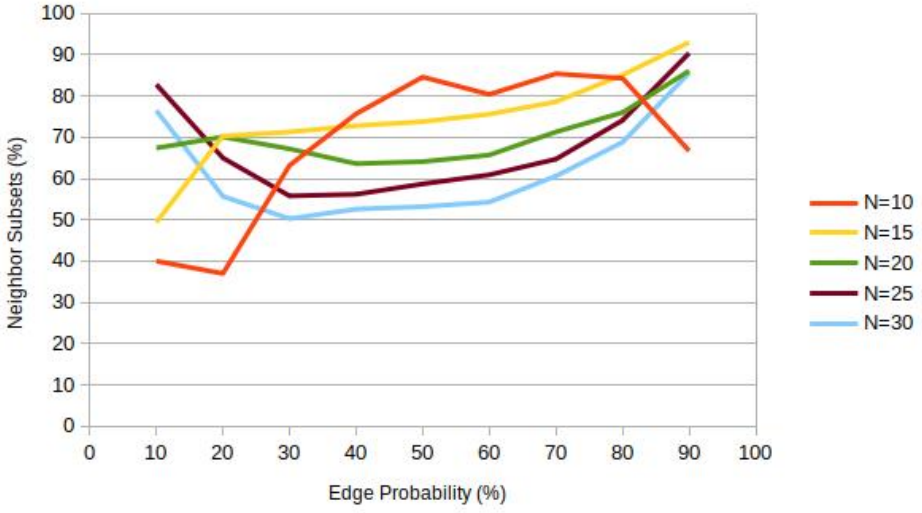


Fig. 93. Proposed algorithm neighborhood subset test.

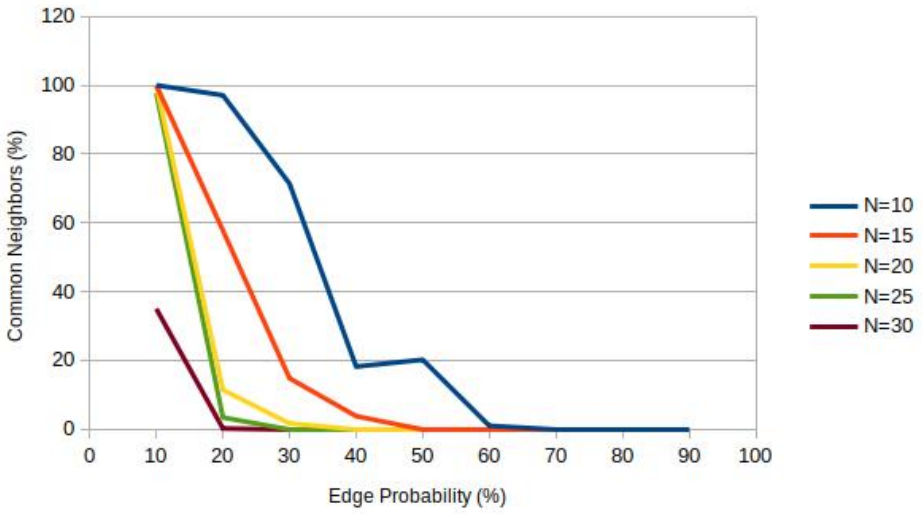


Fig. 94. Proposed algorithm minimum common neighbors upper bound test.

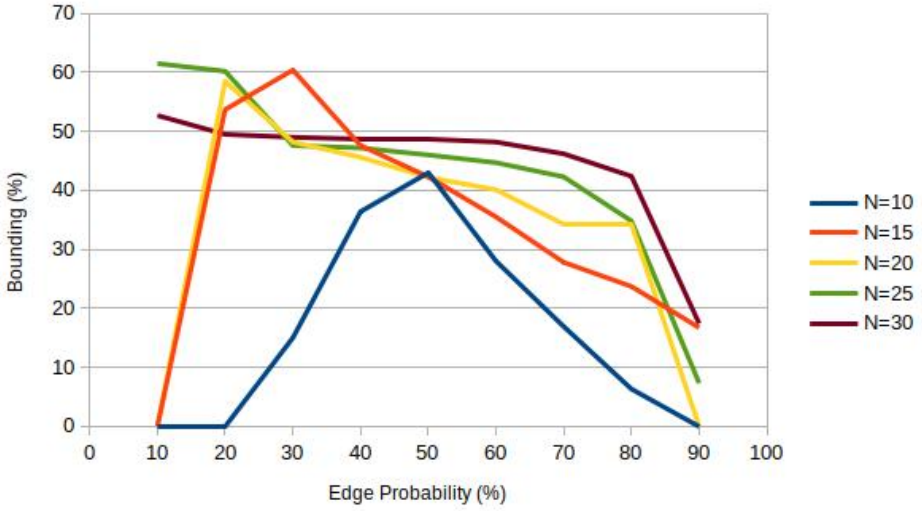


Fig. 95. Proposed algorithm bounding test.

6.4 Runtime Duration Results

7 TOASTER DESIGN CASE STUDY

This section contains a simplistic case study of how the proposed algorithm could be used to compare the FR graphs resulting from two slightly different designs of the basic kitchen toaster shown in Fig. 96.

Fig. 96. An example toaster.

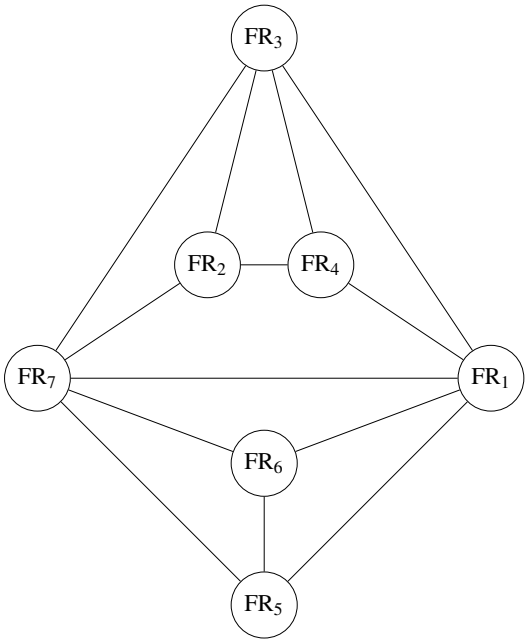
The FRs are shown in Table 6. It is assumed that the design is either uncoupled or decoupled and hence the independence of the FRs is as strong as possible.

Table 6

Toaster Functional Requirements

FR_1	Body contains all parts
FR_2	Can be safely moved while hot
FR_3	Can hold two slices of bread
FR_4	Heats each slice of bread on both sides
FR_5	Toasting is manually started
FR_6	Toasting is automatically or can be manually stopped
FR_7	Heat level can be controlled

The graph G_1 for the first candidate design with $n = 7$ and $m = 13$ is shown in Fig. 97.



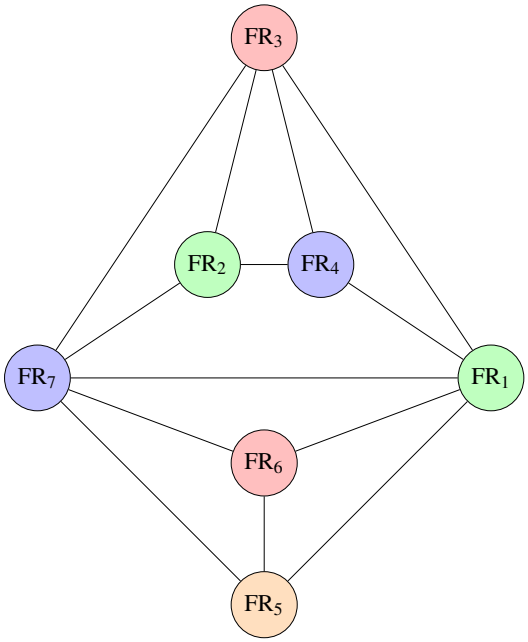
G_1

Fig. 97. First candidate design.

Running the Bron Kerbosch and greedy last-first algorithms indicate that $\chi(G_1) = 4$. The 4-chromatic coloring returned by the greedy algorithm with:

$$C = \{\text{green}, \text{blue}, \text{red}, \text{orange}\}$$

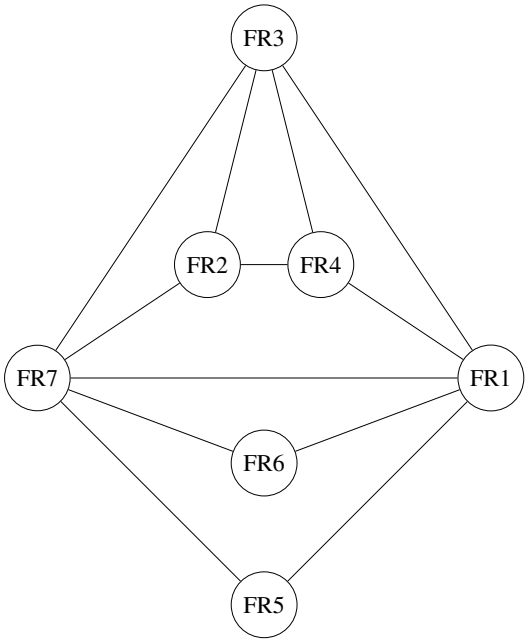
is show in Fig. 98.



G_1

Fig. 98. First design chromatic coloring.

Notice in the design that FR_5 (start toasting) and FR_6 (stop toasting) have been forced into separate parts, conceivably to accommodate the separate "cancel" button shown in Fig. 96. But what if the designer decides to eliminate the cancel button and allow manual cancellation via the lever? Thus, FR_5 and FR_6 no longer need to be separated, so the edge between their vertices can be eliminated. The result is shown in Fig. 99.



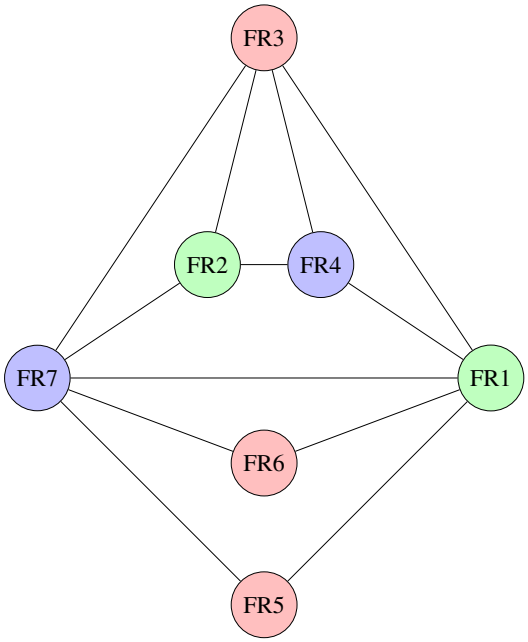
G_2

Fig. 99. Second candidate design.

Now, running the Bron Kerbosch and greedy last-first algorithms indicate that $\chi(G_1) = 3$. The 3-chromatic coloring returned by the greedy algorithm with:

$$C = \{\text{green}, \text{blue}, \text{red}\}$$

is show in Fig. 100.



G_2

Fig. 100. Second design chromatic coloring.

This process gives the designer the feedback that the second design requires only three parts instead of four, and thus has less information content and hence a higher chance of success than the first design. It will be up to the designer to weigh this result against other aspects of the design.

Note that in both cases, the chromatic number is equal to the lower and upper bounds. In fact, the author found it rather difficult to synthesis a case study with meaningful relationships between the FRs such that this was not the case. The random graph analysis supports this, with such a high percentage of the random graphs exhibiting this quality.

8 CONCLUSIONS

Literature Cited

[1] N. P. Suh, *The Principles of Design*. New York: Oxford University Press, 1990.

[2] R. A. Shirwaiker and G. E. Okudan, "Triz and axiomatic design: A review of case-studies and a proposed synergistic use," *Journal of Intelligent Manufacturing*, vol. 19, pp. 33–47, February 2008.

[3] N. P. Suh, "Axiomatic design theory for systems," *Research in Engineering Design*, vol. 10, pp. 189–209, 1998.

[4] S. Behdad, J. Cavallaro, P. K. Gopalakrishnan, and S. Jahanbekam, "A graph coloring technique for identifying the minimum number of parts for physical integration in product design," in *Proceedings of the ASME 2019 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference*, no. DETC2019-98251 in IDETC/CIE 2019, (Anaheim, CA), Aug 18–21 2019.

[5] S. Behdad, P. K. Gopalakrishnan, S. Jahanbekam, and H. Klein, "Graph partitioning technique to identify physically integrated design concepts," in *Proceedings of the ASME 2018 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference*, no. DETC2018-85646 in IDETC/CIE 2018, (Quebec, Canada), Aug 26–29 2018.

[6] Y. Tang, S. Yang, and Y. F. Zhao, "Sustainable design for additive manufacturing through functionality integration and part consolidation," in *Handbook of Sustainability in Additive Manufacturing* (S. S. Muthu and M. M. Savalani, eds.), vol. 1, pp. 101–144, Singapore: Springer, 2016.

[7] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco: W.H. Freeman and Company, 1979.

[8] N. Christofides, "An algorithm for the chromatic number of a graph," *The Computer Journal*, vol. 14, pp. 38–39, January 1971.

[9] C. C. Wang, "An algorithm for the chromatic number of a graph," *Journal of the Association for Computer Machinery*, vol. 21, no. 3, pp. 385–391, 1974.

[10] D. G. Corneil and B. Graham, "An algorithm for determining the chromatic number of a graph," *SIAM Journal on Computing*, vol. 2, pp. 311–318, December 1973.

[11] G. Chartrand and P. Zhang, *A First Course in Graph Theory*. Mineola, New York: Dover Publications, 2012.

[12] D. B. West, *Introduction to Graph Theory*. Upper Saddle River, NJ: Prentice Hall, 2^{nd} ed., 2001.

[13] A. A. Zykov, *On Some Properties of Linear Complexes*, vol. 7 of *American Mathematical Society Translations Series One*. American Mathematical Society, 1949 (translated 1952).

[14] J. Hopcroft and R. Tarjan, "Algorithm 447: Efficient algorithms for graph manipulation," *Communications of the ACM*, vol. 16, pp. 372–378, June 1973.

[15] C. McDiarmid, "Determining the chromatic number of a graph," *SIAM Journal on Computing*, vol. 8, pp. 1–14, 02 1979.

[16] M. Sipser, *Introduction to the Theory of Computation*. Boston, MA: Cengage Learning, 3^{rd} ed., 2013.

[17] H. Johnston, "Cliques of a graph—variations on the bron-kerbosch algorithm," *International Journal of Computer and Information Sciences*, vol. 5, no. 3, pp. 209–238, 1976.

[18] T. Zhang, "Triangle free graphs and their chromatic number." http://www.math.uchicago.edu/~may/VIGRE/VIGRE2008/REUPapers/Zhang.pdf. Accessed: 2019-12-01.

[19] C. Edwards and C. Elphick, "Lower bounds for the clique and the chromatic numbers of a graph," *Discrete Applied Mathematics*, vol. 5, no. 1, pp. 51–64, 1983.

[20] M. Xiao and H. Nagamouchi, "Exact algorithms for maximum independent set," *Information and Computation*, vol. 255, pp. 126–146, 2017.

[21] C. Bron and J. Kerbosch, "Algorithm 457: Finding all cliques of an undirected graph," *Communications of the ACM*, vol. 16, pp. 575–577, September 1973.

[22] J. Moon and L. Moser, "On cliques in graphs," *Israel Journal of Mathematics*, vol. 3, pp. 23–28, 1965.

[23] D. Welsh and M. Powell, "An upper bound for the chromatic number of a graph and its application to timetabling problems," *The Computer Journal*, vol. 10, no. 1, pp. 85–86, 1967.

[24] D. W. Matula, G. Marble, and J. D. Isaacson, "Graph coloring algorithms," in *Graph Theory and Computing* (R. C. Read, ed.), pp. 109–122, New York: Academic Press, 1972.

[25] R. C. Read, "An introduction to chromatic polynomials," *Journal of Combinatorial Theory*, vol. 4, pp. 52–71, 1968.

[26] B. Graham, "An algorithm to determine the chromatic number of a graph," Master's thesis, University of Toronto, Ontario, Canada, October 1972.

[27] "The online encyclopedia of integer sequences: A000110." https://oeis.org/A000110. Accessed: 2019-12-29.