Two Graph Vertex Partitioning Algorithms for Part Consolidation in Axiomatic Design

Jeffery A. Cavallaro
Graduate Student
Mathematics Department
San Jose State University
jeffery.cavallaro@sjsu.edu

September 21, 2019

1 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 textbooks used [1, 3] and class notes compiled by the author during his undergraduate and graduate graph theory classes at SJSU.

1.1 Simple Graphs

The problem of part consolidation is best served by a class of graphs called *simple graphs*:

Definition: Simple Graph

A *simple graph* is a mathematical object represented by a tuple $G = (V, E, \ldots)$ consisting of a non-empty and finite set of *vertices* (also called *nodes*) $V(G)$, a finite and possibly empty set of edges $E(G)$, and zero of more relations. Each edge is represented by a two-element subset of $V(G)$ called the *endpoints* of the edge:

$$E(G) \subseteq \mathcal{P}_2\left(V(G)\right)$$

Each relation has $V(G)$ or $E(G)$ as its domain and is used to associated vertices or edges with problem-specific attributes.

For the remainder of this work, the use of the term "graph" implies a "simple graph."

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:

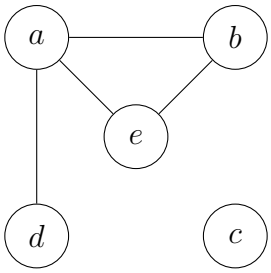
1. Every two vertices of a graph are the endpoints of at most one edge; there are no so-called *multiple* edges between two vertices.
2. The two endpoint vertices of an edge are always distinct; there are no so-called *loop* edges on a single vertex.
3. The two endpoint vertices are unordered, suggesting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

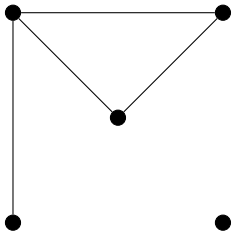
A part consolidation problem can be represented by a graph whose vertices are the functional requirements (FRs) of the design and whose edges indicate which endpoint FRs should never be combined into a single part.

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 Figure 1.1.

When referring to the edges in a graph, the following common notation will be used:

Notation: Edge





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

$E = E(G) = \big\{\{a, b\}, \{a, d\}, \{a, e\}, \{b, e\}\big\}$

Figure 1: An Example Graph (labeled and unlabeled)

The edge $\{u, v\}$ is represented by the simple juxtaposition uv or vu.

Note that there is no requirement that every vertex in a graph be an endpoint to some edge:

Definition: Isolated Vertex

Let G be a graph and let $u \in V(G)$. To say that u is an *isolated* vertex means that it is not an endpoint for any edge in $E(G)$:

$$\forall\, e \in E(G), u \notin e$$

In the example graph of Figure 1.1, notice that vertex c is an isolated vertex.

When two vertices are the endpoints of the same edge the vertices are said to be *adjacent* or are called *neighbors*:

Definition: Adjacent Vertices

Let G be a graph and let $u, v \in V(G)$. To say that u and v are *adjacent* vertices, also called *neighbors*, means that they are the endpoints of some edge $e \in E(G)$:

$$\exists\, e \in E(G), e = uv$$

The edge e is said to *join* its two endpoint vertices u and v. Furthermore, the edge e is said to be *incident* to its endpoint vertices u and v.

In the example graph of Figure 1.1, notice that vertex a is adjacent to vertices b, e, and d; and vertex b is adjacent to vertex e.

We can also speak of adjacent edges, which are edges that share an endpoint:

Definition: Adjacent Edges

Let G be a graph and left $e, f \in E(G)$. To say that e and f are *adjacent* edges means that they share some endpoint $v \in E(G)$:

$$\exists\, v \in V(G), e \cap f = \{v\}$$

or similarly:

$$|e \cap f| = 1$$

Note that two edges in a simple graph can only share one endpoint; otherwise, the two edges would be multiple edges, which are not allowed in simple graphs.

In the example graph of Figure 1.1, notice that ab is adjacent to ad, ae, and be; and ae is adjacent to be.

1.2 Order and Size

Two of the most important characteristics of a graph are the number of vertices in the graph, called the *order* of the graph, and the number of edges in the graph, called the *size* of the graph:

Definition: Order

Let G be a graph. The *order* of G, denoted by n or $n(G)$, is the number of vertices in G:

$$n = n(G) = |V(G)|$$

Definition: Size

Let G be a graph. The *size* of G, denoted by m or $m(G)$, is the number of edges in G:

$$m = m(G) = |E(G)|$$

In the example graph of Figure 1.1, notice that $n = 5$ and $m = 4$.

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

Theorem

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:

Definition: Degenerate Cases

- 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, the graph is *non-trivial*.
- An *empty* graph is a graph with possibly some isolated vertices but with no edges ($m = 0$).

Hence, both the null and trivial graphs are empty.

1.3 Graph Tuple Relations

Various problems in graph theory require that vertices and edges be assigned values of particular attributes. This is accomplished by adding relations to the graph tuple that map the vertices and/or edges to their attribute values. Note that there are no particular limitations on the nature of such a relation — everything from a basic relation to a bijective function is possible, depending on the problem.

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.

1.3.1 Labels

One of the possible relations in a graph tuple is a bijective function that assigns each vertex an identifying label. When such a function is present, the graph is said to be a *labeled* graph:

Definition: Labeled Graph

To say that a graph G is *labeled* means that its vertices are considered to be distinct and are assigned identifying names (labels) by adding a bijective labeling function to the graph tuple:

$$\ell : V(G) \to L$$

where L is a set of labels (names). Otherwise, the vertices are considered to be identical (only the structure of the graph matters) and the graph is *unlabeled*.

The vertices in a labeled graph are typically draw 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 Figure 1.1: 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:

$$FR_1, FR_2, FR_3, \ldots, FR_n$$

1.3.2 Vertex Color

Other graph theory problems require that the graph's vertex set 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 is by adding a *coloring* function to the graph tuple:

$$c : V(G) \to C$$

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 corresponds to empty sets in the distribution.

The most popular coloring scheme for a graph requires that adjacent vertices be assigned different colors:

Definition: Proper Coloring

A coloring c on a graph G is called *proper* when no two adjacent vertices are assigned the same color:

$$\forall\, u, v \in V(G), uv \in E(G) \implies c(u) \neq c(v)$$

A proper coloring c 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.

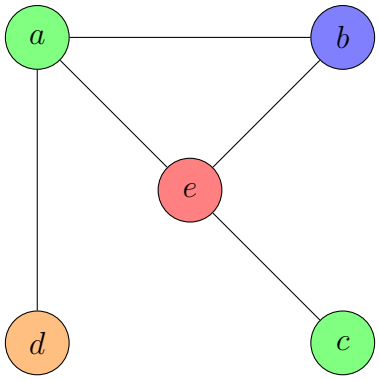
An example of a 4-coloring is shown in Figure 1.3.2.

Since there is no requirement that a coloring c be surjective, the codomain C may contain unused colors. For example, the codomain of the coloring shown in Figure 1.3.2 might be:

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

and hence c is surjective and G is 4-colorable. But we can always add an unused color to C:

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



$c(a) = green$
$c(b) = blue$
$c(c) = green$
$c(d) = orange$
$c(e) = red$

Figure 2: A Graph with a 4-coloring

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.

Thus, we can make the following statement:

Proposition 1

Let G be a graph:

$$G \text{ is } k\text{-colorable} \implies G \text{ is } (k+1)\text{-colorable}$$

By inductive application of Proposition 1, one can arrive at the following conclusion:

Proposition 2

Let G be a graph:

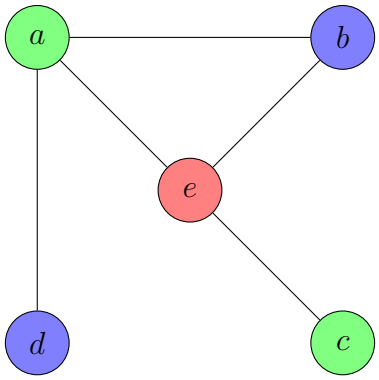
$$G \text{ is } k\text{-colorable} \implies G \text{ is } (k+r)\text{-colorable for some } r \in \mathbb{N}.$$

Since $k \in \mathbb{N}$, by the well-ordering principle, there exists some minimum k such that a graph G is k-colorable:

Definition: Chromatic Coloring

The minimum k such that a graph G is k-colorable 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.

Returning to the example 4-coloring of Figure 1.3.2, 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 Figure 1.3.2. 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.



$c(a) = green$
$c(b) = blue$
$c(c) = green$
$c(d) = blue$
$c(e) = red$

Figure 3: A Graph with a Chromatic 3-coloring

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:

Definition: Independent Set

Let G be a graph and let $S \subseteq V(G)$. To say that S is an *independent* set means that all of the vertices in S are non-adjacent in G:

$$\forall\, u, v \in S, uv \notin E(G)$$

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 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.

1.4 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.

1.4.1 Graph Equality

Graph equality follows from equality of the vertex and edge sets:

Definition: Graph Equality

Let G and H be graphs. To say that G is equal to H, denoted $G = H$, means that $V(G) = V(H)$ and $E(G) = E(H)$.

Note that this definition of equality ignores any additional relations that may be added to the graph tuples since those relations tend to be added by specific problems and do not reflect the actual parts of the graphs.

1.4.2 Subgraphs

Since graph equality follows from vertex and edge set equality, there should also be a concept of a *subgraph* resulting from the subsets of those sets:

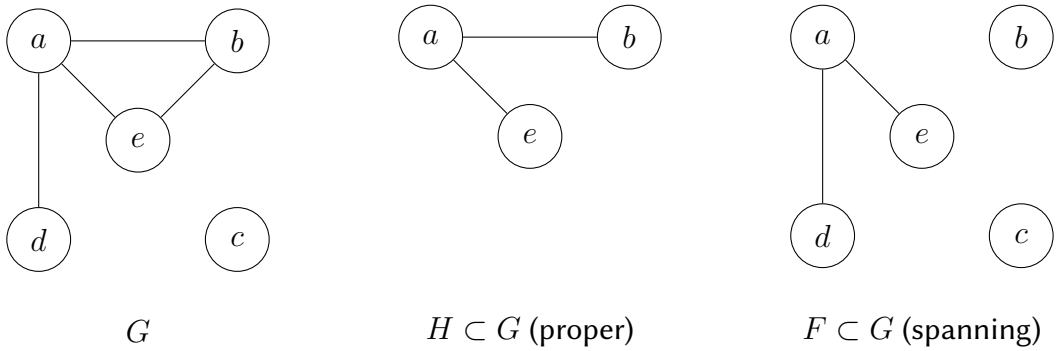
Definition: Subgraph

Let G and H be two graphs:

- To say that H is a *subgraph* of G, denoted $H \subseteq G$, means that $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$.
- To say that H is a *proper subgraph* of G, denoted $H \subset G$, means that $H \subseteq G$ but $H \neq G$: $V(H) \subset V(G)$ or $E(H) \subset E(G)$.
- To say that H is a *spanning subgraph* of G means that H is a subgraph of G such that $V(H) = V(G)$ and $E(H) \subseteq E(G)$.

Thus, given a graph G and a subgraph H, there should be a sequence of zero or more vertex and/or edge removals to obtain H from G. Likewise, there should be a sequence of zero or more vertex and/or edge additions to obtain G from H. If H is a proper subgraph of G then H and G differ by at least one removed vertex or one removed edge. If H is a spanning subgraph of G then H contains all of the vertices in G but may differ by removed edges only. Per the definition, a graph is always a subgraph of itself ($G \subseteq G$) and the null graph is a subgraph of every graph.

The concept of subgraphs is demonstrated by graphs G, H, and F in Figure 1.4.2. 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)

Figure 4: Subgraph Examples

1.4.3 Induced Subgraphs

An *induced* subgraph is a special type of subgraph:

Definition: Induced Subgraph

Let G be a graph and let S be a non-empty subset of $V(G)$. The subgraph of G *induced* by S, denoted $G[S]$, is a subgraph H such that:

- $V(H) = S$
- $u, v \in V(H)$ and $uv \in E(G) \implies uv \in E(H)$

Such a subgraph H is called an *induced subgraph* of G.

Note that when a vertex is removed to make an induced subgraph then all of that vertex's incident edges are also removed. However, for every pair of vertices included in an induced subgraph, if the vertices are the endpoints of a particular edge then that edge must also be included in the subgraph. In the examples of Figure 1.4.2, 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. Figure 1.4.3 adds edge be so that H is now an induced subgraph of G.

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

Figure 5: Induced Subgraph Example

1.5 Mutators

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

1.5.1 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.

Figure 1.5.1 shows an example of vertex removal: vertices c and e are removed, along with their incident edges ae and be.

If $|S| = 1$ then an alternate syntax can be used. Assume $v \in V(G)$:

$$G - v = G - \{v\}$$

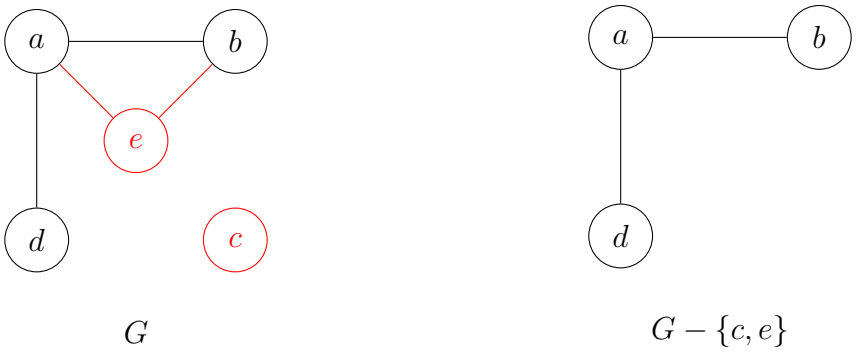


Figure 6: Vertex Removal Example

The proposed algorithm uses vertex removal to simplify a k-colorable graph into a simpler subgraph that is still k-colorable.

1.5.2 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 $G + uv$.

Figure 1.5.2 shows an example of edge addition: edge cd is added.

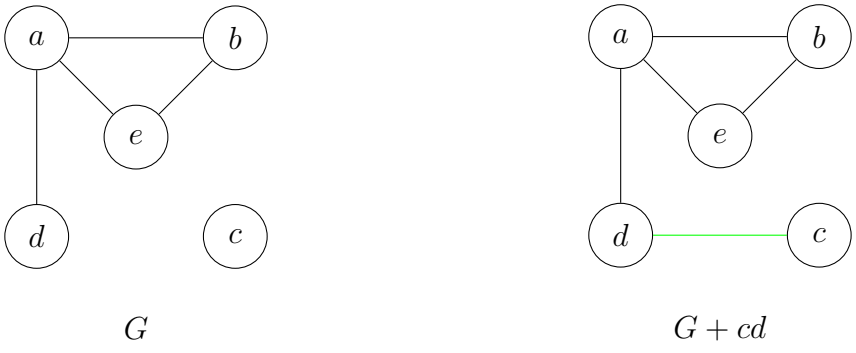


Figure 7: Edge Addition Example

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

1.5.3 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:

$$G - X = H\left(V(G), E(G) - X\right)$$

Thus, only edges are remove — 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 (no edges).

Figure 1.5.3 shows an example of edge removal: edges ae and be are removed.

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

Figure 8: Edge Removal Example

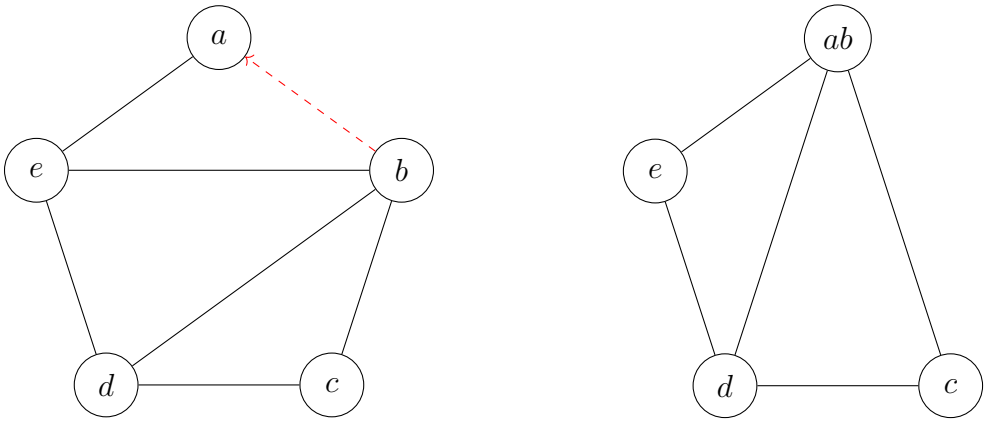
If $|X| = 1$ then an alternate syntax can be used. Assume $e \in E(G)$:

$$G - e = G - \{e\}$$

1.5.4 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.

Figure 1.5.4 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.



G $\qquad\qquad$ $G \cdot ab$

Figure 9: 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.

1.6 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.

1.6.1 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*:

Definition: 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:

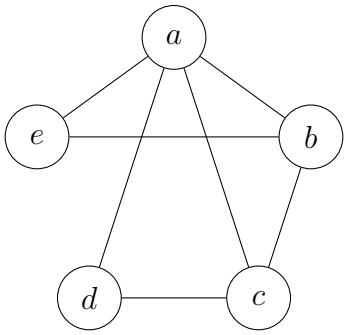
Proposition 3

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 Figure 1.6.1. Note that 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.

Note that in the general case, vertices and edges are allowed to be repeated during a walk. Certain special walks can be defined by restricting such repeats:

Definition: Special Walks



$W_1 = (a, b, e, a, c)$ is open
$W_2 = (a, e, b, c, a)$ is closed

$|W_1| = |W_2| = 4$

Figure 10: Open and Closed Walks in a Graph

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 Figure 1.6.1.

When discussing the connectedness of a graph, the main concern is the existence of paths between vertices:

Definition: Connected 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

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 lease 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 minimal 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 Figure 1.6.1.

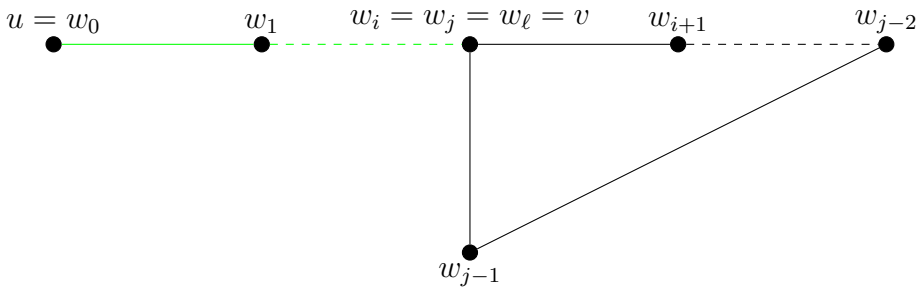


Figure 11: Repeated Vertex at End Case

Let $P' = (u = w_0, w_1, \ldots, w_i = v)$ be the walk shown in green in the figure. 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 Figure 1.6.1.

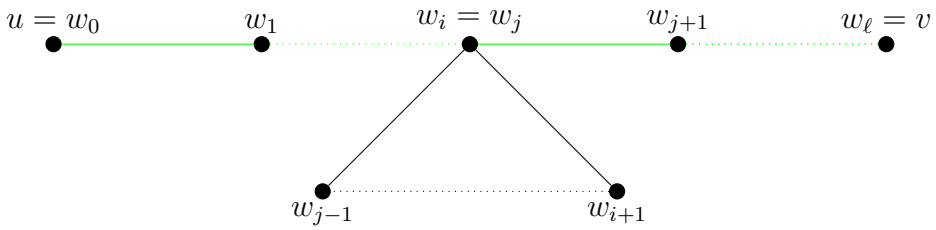


Figure 12: Repeated Vertex Inside Case

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 figure. 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. ∎

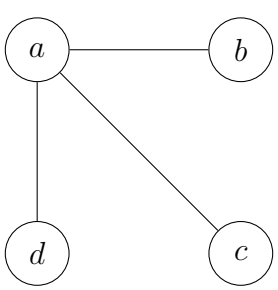
1.6.2 Connected

A *connected* graph is a graph whose vertices are all connected:

Definition: Connected Graph

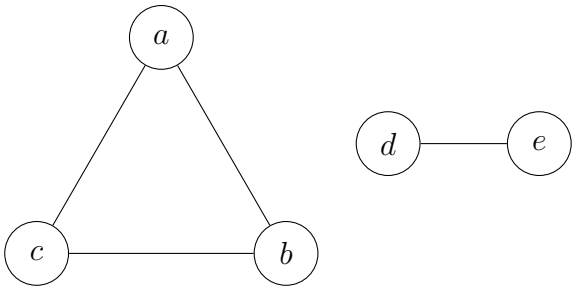
To say that a graph G is *connected* means that for all $u, v \in V(G)$ there exists a $u - v$ path. Otherwise, G is said to be *disconnected*.

Examples of connected and disconnected graphs are shown in figure 1.6.2.



(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

Figure 13: 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).

1.6.3 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:

Definition: Component

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 $\mathscr{G}$:

$$\forall\, F \in \mathscr{G} - \{H\}, H \not\subset F$$

The number of distinct components in G is denoted by:

$$k = k(G)$$

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

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. ∎

1.6.4 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 use Proposition 2 to 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 Figure 1.6.2. 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 [2]. 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. This algorithm has a runtime complexity of $\mathcal{O}(\max(n, m))$.

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.

1.7 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:

Definition: Neighbor

Let G be a graph and let $u, v \in V(G)$. To say that u is a *neighbor* of v (and vice-versa) means that $uv \in E(G)$.

Thus, neighbor vertices are adjacent. Note that for simple graphs, a vertex is never a neighbor of itself.

The set of all neighbors for a vertex is referred to as the vertex's neighborhood:

Definition: Neighborhood

Let G be a graph and let $u \in V(G)$. The *neighborhood* of u, denoted by $N(u)$, is the set of all neighbors of u in G:

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

The degree of a vertex is then defined to be the cardinality of its neighborhood:

Definition: Degree

Let G be a graph and let $u \in G$. The *degree* of u, denoted by $\deg_G(u)$ or $\deg(u)$, is the cardinality of the neighborhood of u:

$$\deg(u) = |N(u)|$$

Note that 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:

Notation

Let G be a graph:

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

And so, for a graph G or order n, it must be the case that 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:

Definition: Vertex Types

Let G be a graph of order n and let $u \in V(G)$:

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

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:

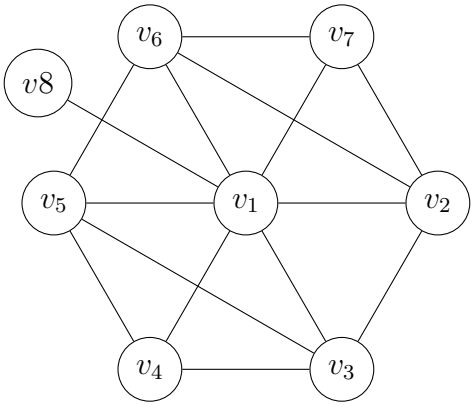
Theorem: First Theorem of Graph Theory

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 Figure 1.7; note that the sum of the vertex degrees is 30, which is twice the number of edges in the graph.



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	

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

Figure 14: Vertex Degrees

1.8 Special Graphs

We conclude this introductory section on graph theory with a discussion of some special classes of graphs that are important to the execution of the proposed algorithm.

1.8.1 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 iff $n = 1$. Examples of empty graphs are shown in Figure 1.8.1.

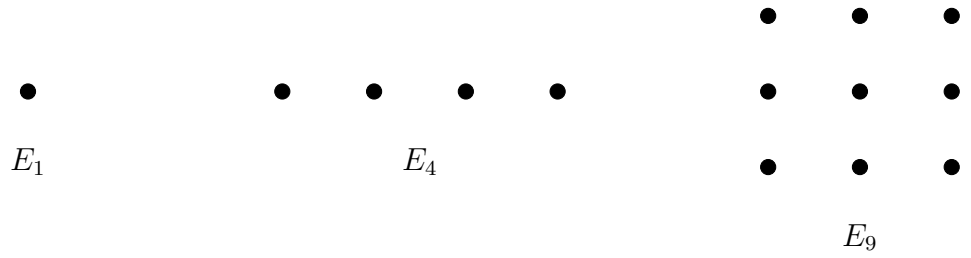


Figure 15: Empty Graphs

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.

1.8.2 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 Figure 1.8.2.

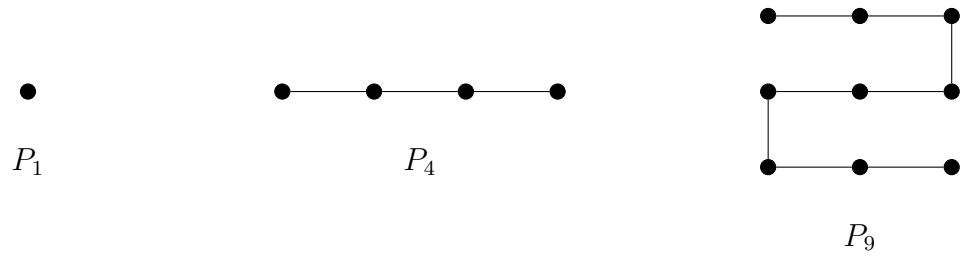


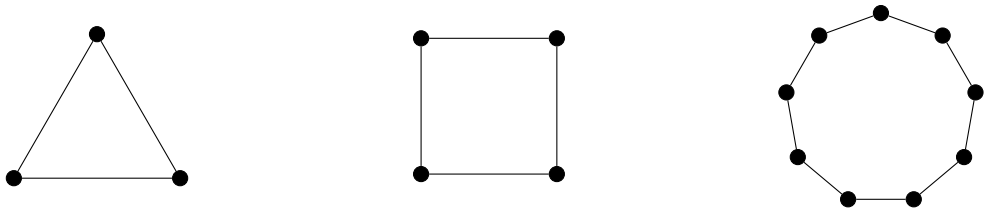
Figure 16: Path Graphs

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

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

1.8.3 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.



C_3 (odd) C_4 (even) C_9 (odd)

Figure 17: Cycle Graphs

Examples of cycle graphs are shown in Figure 1.8.3.

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

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

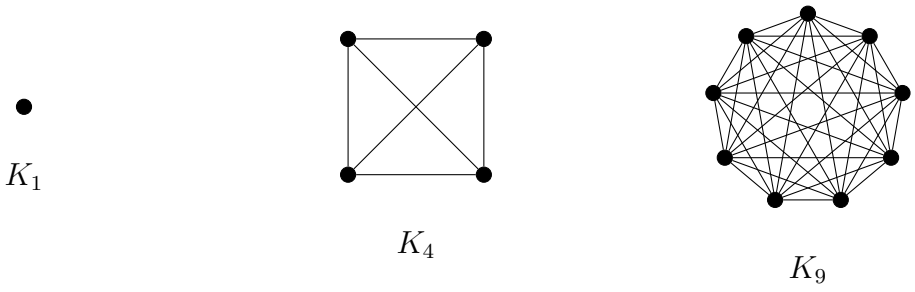
1.8.4 Complete Graphs

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:

$$E(G) = \mathcal{P}_2(V(G))$$

and thus all n vertices are adjacent to each other.

Examples of complete graphs are shown in Figure 1.8.4.



K_1 K_4 K_9

Figure 18: 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.

1.8.5 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

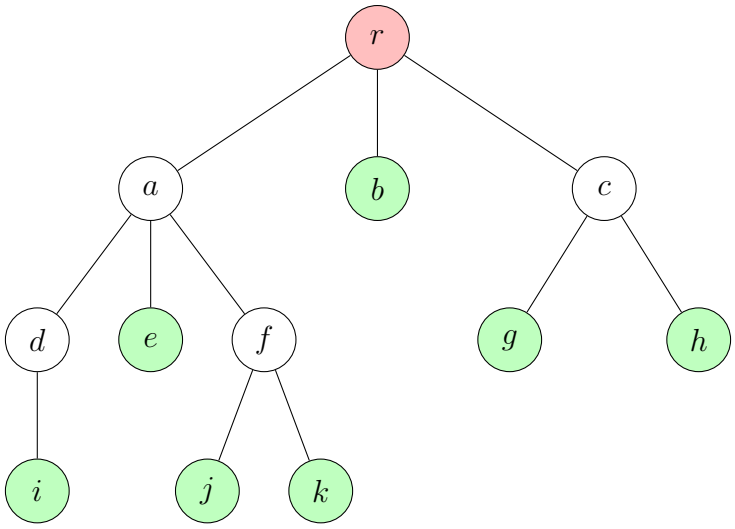


Figure 19: A Tree Organized from Root to Leaves

equidistant from the root vertex. Note that 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 Figure 1.8.5. The root vertex r is shown in red and the leaf vertices b, e, g, h, i, j, k are shown in green.

Trees are important because they can be used to represent the "paper tape" for recursive Turing machine-type algorithms. Each vertex represents a state of the problem and the edges represent "movement" of the tape to select the current state. All states can be visited using a so-called *depth-first* walk. In the example in Figure 1.8.5, 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, the tree is called a *Zykov* tree. These concepts are described in detail in the next section.

2 The Proposed Algorithm

References

[1] Gary Chartrand and Ping Zhang. *A First Course in Graph Theory*. Dover Publications, Mineola, New York, 2012.

[2] John Hopcroft and Robert Tarjan. Algorithm 447: Efficient algorithms for graph manipulation. *Commun. ACM*, 16(6):372–378, June 1973.

[3] Douglas B. West. *Introduction to Graph Theory*. Prentice Hall, Upper Saddle River, NJ, 2^{nd} edition, 2001.