Dominating sets in graphs with no long induced paths

Jessica Shi

Advisor: Maria Chudnovsky

a senior thesis submitted in partial fulfillment
of the requirements for the degree of Bachelor of Arts in Mathematics at
Princeton University

2018

This thesis represents my own work in accordance with University regulations.
Jessica Shi.

Abstract

3-coloring is a classically difficult problem, and as such, it is of interest to consider the computational complexity of 3-coloring restricted to certain classes of graphs. P_t-free graphs are of particular interest, and the problem of 3-coloring P_8-free graphs remains open. One way to prove that 3-coloring graph class $\mathcal{G}$ is polynomial is by showing that for all $G \in \mathcal{G}$, there exists a constant bounded dominating set in G; that is to say, G contains a dominating set S such that $|S| \leq K_{\mathcal{G}}$ for constant $K_{\mathcal{G}}$.

In this paper, we prove that there exist constant bounded dominating sets in subclasses of P_t-free graphs. Specifically, we prove that excepting certain reducible configurations which can be disregarded in the context of 3-coloring, there exist constant bounded dominating sets in $\{P_6, \text{triangle}\}$-free and $\{P_7, \text{triangle}\}$-free graphs. We also provide a semi-automatic proof for the latter case, due to the algorithmic nature of the proof.

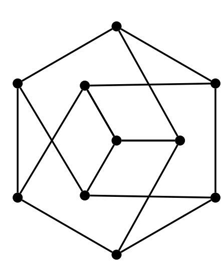
Acknowledgements

I would like to thank my advisor, Professor Maria Chudnovsky, for her invaluable support and guidance in this work. Without her help and direction, this thesis would not have been possible.

I would also like to thank the many friends that I have made throughout my time at Princeton. Thank you to my incredible roommate of four years, Katherine Pizano, for never dragging me to a horror movie and for all of the shared Oreos and chocolates. Thank you to Savannah Du, for joining me in many egg-cellent eggs-cursions and being a constant voice of reason. Finally, thank you to Heesu Hwang for multiplying small numbers and remembering the arithmetic mean-harmonic mean (AM-HM) inequality in a time of crisis.

I would like to thank my family, for supporting and encouraging me my entire life. A very special thank you to Audrey Shi, for supplying copious amounts of sugar and candid photo collages of our parents.

Lastly, thank you to Princeton University, and in particular, Colonial Club and the Princeton University Mathematics Department, for providing a tight-knit community and unforgettable times. Thank you to Worcester College at the University of Oxford, for the best bubble tea and immaculate lawns. A final thank you to the Princeton University Computer Science Department, for their computing resources.



Contents

Chapter 1

Introduction

1.1 Context

For $k > 0$, a *k-coloring* of a graph G is a function $c : V(G) \to [k]$ such that $c(u) \neq c(v)$ for all edges $(u, v) \in E(G)$. The problem of determining the smallest k such that a given graph G admits a k-coloring is a classically difficult problem, and indeed, is one of Karp's NP-complete problems [13]. The problem remains difficult even if k is fixed; specifically, determining whether a given graph G admits a k-coloring for fixed $k \geq 3$ is NP-complete [20]. This problem is known as the *k-coloring problem.*

As such, it is of interest to determine the classes of graphs in which the k-coloring problem is polynomially solvable. We focus on classes of graphs that forbid certain induced subgraphs. For a graph H, we say a graph G is H-free if H is not an induced subgraph of G. More generally, for a set of graphs S, we say a graph G is S-free if no graph in S is an induced subgraph of G. There are several results regarding the k-colorability of S-free graphs.

Kamiński and Lozin [15] proved that for any $k, g \geq 3$, the k-coloring problem on graphs with no cycles of length $\leq g$ is NP-complete. Thus, for any $k \geq 3$, the k-coloring problem on H-free graphs for any graph H containing a cycle is NP-complete. Moreover, for any $k \geq 3$ and any forest H with a vertex of degree ≥ 3, the k-coloring problem on H-free graphs is NP-complete [11, 14, 4]. Thus, the only remaining graphs of interest are graphs in which all components are paths.

Note that it is simple to show that k-coloring P_t-free graphs for $t \leq 4$ is polynomial [19]. For $t = 5$, Hoàng *et al.* [10] proved that k-coloring P_5-free graphs is polynomial for all k. Huang [12] proved that 5-coloring P_6-free graphs and 4-coloring P_7-free graphs are both NP-complete. Chudnovsky *et al.* [6, 7] recently extended this to show that 4-coloring P_6-free graphs is polynomial. These cover all cases of k-coloring P_t-free graphs for $k \geq 4$.

This thesis focuses more specifically on the case where $k = 3$. Randerath and Schiermeyer [18] proved that 3-coloring P_6-free graphs is polynomial, and Bonomo *et al.* [2] proved that 3-coloring P_7-free graphs is polynomial. The 3-colorability of P_8-free graphs remains an open problem.

1.2 Our work

One way to consider the 3-colorability of a graph class is via list coloring. For any graph G and any function $L : V(G) \to \mathcal{P}(\mathbb{Z}^+)$ (where $\mathcal{P}(\mathbb{Z}^+)$ denotes the powerset of the positive integers), a *list coloring* of (G, L) is a function $c : V(G) \to \cup_{v \in V(G)} L(v)$ such that $c(u) \neq c(v)$ for all edges $(u, v) \in E(G)$ and $c(v) \in L(v)$ for all $v \in V(G)$. The problem of determining if a pair (G, L) where $|L(v)| \leq k$ for all $v \in V(G)$ admits a list coloring is known as the *list k-coloring problem.* Edwards [9] proved that the list 2-coloring problem is polynomial, through a reduction to 2-SAT.

Now, for any graph G, a *dominating set* is a set of vertices $S \subseteq V(G)$ such that every vertex in $V(G) \setminus S$ has a neighbor in S. Consider any graph class $\mathcal{G}$ with constant bounded dominating sets; in other words, every $G \in \mathcal{G}$ has a dominating set S such that $|S| \leq K_{\mathcal{G}}$, for some constant $K_{\mathcal{G}}$. In order to 3-color any $G \in \mathcal{G}$, we can find said dominating set S in polynomial time and consider all possibilities of 3-coloring the induced subgraph of G on S. There are a constant number of such colorings, and for each possibility, the problem of 3-coloring the remaining vertices in G reduces to a list 2-coloring problem on $G \setminus S$.[1] We can solve each of these list 2-coloring problems in polynomial time, so the 3-coloring problem on $\mathcal{G}$ is polynomial.

Thus, it is of interest to consider whether P_t-free graphs admit constant bounded dominating sets. Bacsó and Tuza [1] showed that every connected P_5-free graph has a dominating clique or a dominating P_3. Using solely this, it is not true that every P_5-free graph admits a constant bounded dominating set; however, we are considering P_5-free graphs in the context of 3-coloring, and any P_5-free graph with a clique of size ≥ 4 is clearly not 3-colorable. As such, in our algorithm to 3-color P_5-free graphs, we can simply check for cliques of size ≥ 4, and if there are none, there exists a constant bounded dominating set of size ≤ 3.

In this thesis, we will similarly consider constant bounded dominating sets for $\{P_6$,

[1] This is because for each $v \in V(G) \setminus S$, v is adjacent to an already colored vertex in S. Every such v can be colored at most 2 colors, which forms the list 2-coloring problem.

triangle}-free and $\{P_7$, triangle}-free graphs, contingent on certain restrictions related to 3-coloring, which we call *reducible configurations*. We defer a full definition to Section 2.

1.3 Our results

We build upon Chudnovsky *et al.*'s [5] characterization of $\{P_6, \text{triangle}\}$-free graphs to show that excepting reducible configurations, $\{P_6, \text{triangle}\}$-free graphs have constant bounded dominating sets.

We also build upon Bonomo *et al.*'s [3] characterization of $\{P_7, \text{triangle}\}$-free graphs to show that excepting reducible configurations, $\{P_7, \text{triangle}\}$-free graphs have constant bounded dominating sets. The proofs for $\{P_7, \text{triangle}\}$-free graphs are somewhat tedious and algorithmic; we provide an additional semi-automatic proof for this case.

Note that by the process detailed in Section 1.2, these results lead directly to a polynomial time algorithm for 3-coloring $\{P_6, \text{triangle}\}$-free and $\{P_7, \text{triangle}\}$-free graphs.

In Section 2, we introduce preliminary notations and definitions. In Section 3, we prove our result for $\{P_6, \text{triangle}\}$-free graphs, and in Section 4, we prove our result for $\{P_7, \text{triangle}\}$-free graphs.

Chapter 2

Preliminaries

For the purposes of this paper, every graph G is *simple*, that is to say, undirected, unlabeled, without self-loops, and without multiple edges. We denote the vertex set of G by $V(G)$, and the edge set of G by $E(G)$. For a set of vertices $U \subseteq V(G)$, we denote the *vertex-induced subgraph* of G on U by $G[U]$. For a vertex $v \in V(G)$, we define the *neighborhood* of v, denoted by $N(v)$, to be the set of vertices in $V(G) \setminus \{v\}$ adjacent to v. Moreover, a *stable set* is a set of vertices that are all pairwise non-adjacent.

Note that we are finding constant bounded dominating sets with respect to 3-coloring; indeed, there exist graphs within these classes in general that have no constant bounded dominating set. However, there are certain *reducible configurations* that can be simplified out of any graph G without changing its 3-colorability, and it suffices to consider only graphs without such configurations. The configurations we use here closely follow those introduced by Chudnovsky [4], and notably, can be identified in polynomial time. Specifically, our reducible configurations include *dominating vertices*, vertices with degree < 3, and *nontrivial homogeneous pairs of stable sets.*

1. Dominating vertices: A *dominating vertex* is a vertex $v \in V(G)$ such that there exists $u \in V(G)$ where $N(u) \subseteq N(v)$.[1] If G contains such a dominating vertex, then G is 3-colorable if and only if $G \setminus \{u\}$ is 3-colorable (by coloring u the same color as v).

2. Vertices with degree < 3: If G contains a vertex v with degree < 3, then G is 3-colorable if and only if $G \setminus \{v\}$ is 3-colorable (by coloring v a color that is not the color of any of its neighbors).

3. Nontrivial homogeneous pairs of stable sets: For any pair of disjoint, non-empty

[1] Necessarily, u and v are non-adjacent.

sets $U, V \subseteq V(G)$, (U, V) is a *homogeneous pair* if every vertex not in $U \cup V$ is either complete or anticomplete to U, and either complete or anticomplete to V. A homogeneous pair (U, V) is *nontrivial* if there exists an edge between U and V, and $|U| + |V| \geq 3$. If U and V are also stable, then (U, V) is a *homogeneous pair of stable sets*.

Consider a nontrivial homogeneous pair of stable sets (U, V) in G, and let $u \in U$ be adjacent to $v \in V$. Then, G is 3-colorable if and only if $G \setminus ((U \setminus \{u\}) \cup (V \setminus \{v\}))$ is 3-colorable (by coloring all vertices in $U \setminus \{u\}$ the same color as u, and all vertices in $V \setminus \{v\}$ the same color as v).

Note that in the proof of Theorem 4.1, we can relax the nontrivial homogeneous pair of stable sets configuration; it suffices instead to exclude any graph G that is bipartite.[2]

Also, we define a *twin* to be a vertex $v \in V(G)$ such that there exists $u \in V(G)$ where $N(u) = N(v)$. By definition, a twin is a dominating vertex.

[2]Note that any bipartite graph G is trivially 3-colorable, and we can detect if any graph G is bipartite in polynomial time.

Chapter 3

$\{P_6, \text{triangle}\}$-free graphs

In this chapter, we prove that $\{P_6, \text{triangle}\}$-free graphs have a constant bounded dominating set, using Chudnovsky *et al.*'s [5] characterization of these graphs.

Theorem 3.1. *If G is a connected $\{P_6, \text{triangle}\}$-free graph with no reducible configurations, then G has a constant bounded dominating set.*

3.1 Setup

We begin by introducing some relevant terminology that is used in Chudnovsky *et al.*'s [5] characterization.

The *Clebsch graph*, shown in Figure 3.1, is a $\{P_6, \text{triangle}\}$-free graph obtained by taking the five-dimensional cube graph and identifying all pairs of opposite vertices. A graph H is *Clebschian* if it is contained within the Clebsch graph.

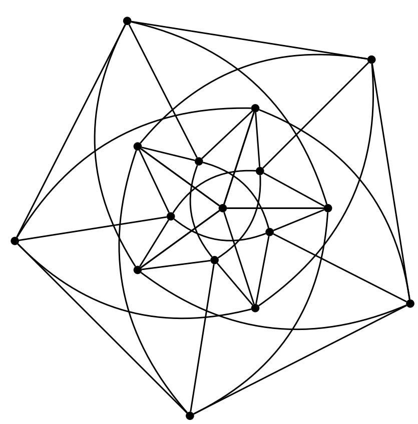


Figure 3.1: The Clebsch graph.

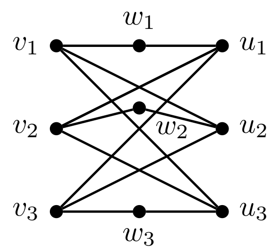


Figure 3.2: A climbable graph.

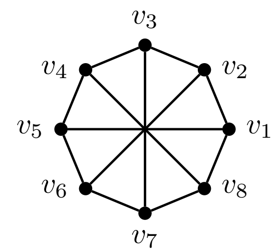


Figure 3.3: The V_8 graph.

We also define *climbable* graphs. Let $K_{n,n}$ be a complete bipartite graph with bipartition $\{v_1, \ldots, v_n\}, \{u_1, \ldots, u_n\}$. We construct a graph H_n by taking $K_{n,n}$ and subdividing each edge (v_i, u_i) for all i; more precisely, we delete each edge (v_i, u_i) and add a vertex w_i adjacent to v_i and u_i. A graph H is *climbable* if it is isomorphic to an induced subgraph of H_n for some n. An example of a climbable graph is shown in Figure 3.2.

A bipartite graph H with bipartition (U, V) is an *antisubmatching* relative to (U, V) if every vertex in U has at most one non-neighbor in V and vice versa. The V_8 *graph*, as shown in Figure 3.3, is a graph obtained by taking a cycle of length 8 and adding an edge between all pairs of opposite vertices. More precisely, a V_8 graph is a graph with vertices $\{v_1, \ldots, v_8\}$ where for any $i \neq j$, (v_i, v_j) is an edge if and only if $i - j = 1$, 4, or $7 \mod 8$. We use this to define a V_8 *expansion*. Let $H_{1,5}$ and $H_{3,7}$ be antisubmatchings relative to (V_1, V_5) and (V_3, V_7) respectively, each with at least one edge. Let H be a graph obtained by taking the V_8 graph and replacing (v_1, v_5) with (V_1, V_5) and (v_3, v_7) with (V_3, V_7). Also, we may delete some vertices in $\{v_2, v_4, v_6, v_8\}$. Then, H is a V_8 *expansion*.

Finally, for any homogeneous pair of stable sets (U, V), (U, V) is *simplicial* if every vertex not in $U \cup V$ with a neighbor in U is adjacent to every vertex not in $U \cup V$ with a neighbor in V.

We can now introduce a primary result from Chudnovsky *et al.* [5] that will serve as the basis of our proof.

Lemma 3.2 (Chudnovsky *et al.* [5])**.** *If G is a connected $\{P_6, triangle\}$-free graph with no twins, then either*

1. *G is Clebschian, climbable, or a V_8 expansion, or*

2. *G admits a nontrivial simplicial homogeneous pair of stable sets.*

3.2 Proof of Theorem 3.1

We consider all of the possibilities of G given by Lemma 3.2. If G is Clebschian, then trivially, the number of vertices in G is bounded by 16, so there exists a constant bounded dominating set of size ≤ 16.

Consider the case where G is climbable. Thus, G is isomorphic to an induced subgraph of H_n for some n, where H_n is constructed by taking $K_{n,n}$ (with bipartition $\{v_1, \ldots, v_n\}, \{u_1, \ldots, u_n\}$) and subdividing each edge (v_i, u_i) for all i. Let w_i denote the vertices added in this subdivision; by definition, $\deg(w_i) = 2$, so if $w_i \in V(G)$, then $\deg(w_i) \leq 2$, which is a reducible configuration. As such, G is necessarily an induced subgraph of $K_{n,n}$. This means that G has a dominating set of size ≤ 2, as desired.

Consider the case where G is a V_8 expansion. We again use notation given in the definition of a V_8 expansion in Section 3.1, where G is constructed from a V_8 graph with vertices $\{v_1, \ldots, v_8\}$, and where (v_1, v_5) and (v_3, v_7) are replaced with antisubmatchings relative to (V_1, V_5) and (V_3, V_7) respectively. Note that by definition, V_1, V_3, V_5, and V_7 are all nonempty.

For all $i = 1, 3, 5, 7$, let $D_i = V_i$ if $|V_i| = 1$, and otherwise, let $D_i = \{x_i, y_i\}$ for any $x_i, y_i \in V_i$. For all $i = 2, 4, 6, 8$, let $D_i = \{v_i\}$ if $v_i \in V(G)$, and otherwise, let $D_i = \emptyset$. We claim that $D = \bigcup_{i\in[8]} D_i$ forms a constant bounded dominating set of G.

Note that it suffices to consider $z_i \in V_i$ such that $z_i \notin D_i$ for $i = 1, 3, 5, 7$. Without loss of generality, consider $z_1 \in V_1$ where $z_1 \notin D_1$. Note that if $v_2 \in V(G)$ or $v_8 \in V(G)$, then z_1 by definition has a neighbor in D. Thus, assume that $v_2, v_8 \notin V(G)$. If $|V_5| = 1$, say $V_5 = D_5 = \{x_5\}$, then in order for G to be connected, z_1 must be adjacent to x_5, as desired. Otherwise, if $|V_5| > 1$, then $D_5 = \{x_5, y_5\}$ and z_1 must be adjacent to at least one of x_5 and y_5, since there exists an antisubmatching relative to (V_1, V_5). In all cases, z_1 has a neighbor in D. The arguments for $i = 3, 5, 7$ follow symmetrically. Thus, D is a constant bounded dominating set of G.

Finally, consider the case where G admits a nontrivial simplicial homogeneous pair (U, V). This is precisely a reducible configuration, which contradicts our hypothesis.

Thus, in all cases, G has a constant bounded dominating set. □

Chapter 4

$\{P_7, \text{triangle}\}$-free graphs

In this chapter, we prove that $\{P_7, \text{triangle}\}$-free graphs with no reducible configurations have a constant bounded dominating set, using Bonomo *et al.*'s [3] characterization of these graphs.

Theorem 4.1. *If G is a connected $\{P_7, \text{triangle}\}$-free graph with no reducible configurations, then G has a constant bounded dominating set.*

4.1 Setup

We begin with Bonomo *et al.*'s [3] characterization of $\{P_7, \text{triangle}\}$-free graphs.[1] Since G is not bipartite, G must contain an odd cycle. The shortest induced odd cycle in G must be either C_5 or C_7, since G is $\{P_7, \text{triangle}\}$-free.

If G is C_5-free, then $V(G) = V_1 \cup \ldots \cup V_7$, where each V_i is nonempty and stable, and V_i is complete to V_{i+1}.[2] Clearly, there exists a constant bounded dominating set of size 7, formed by taking $v_i \in V_i$ for each i.

Thus, assume G contains an induced C_5. We now define certain classes of vertices in G. Let C be an induced C_5, where $C = \{c_1, \ldots, c_5\}$. We call C the *base graph.* Note that all subscripts in the remainder of this section relate to C, and as such are taken modulo 5. The neighborhood of C is given by two *types* of vertices (see Figure 4.1):

- *clone* vertices, which are vertices with neighbors c_{i-1} and c_{i+1} in C for some i, and
- *leaf* vertices, which are vertices with neighbor c_i in C for some i.

[1]Note that in Bonomo *et al.*'s [3] characterization, it is not necessary for G to have no reducible configurations.

[2]Where subscripts are taken modulo 7.

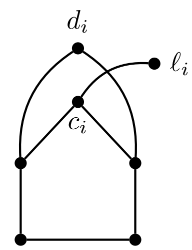


Figure 4.1: A clone d_i and a leaf ℓ_i, both on index i.

For each i, let D_i denote the set of clone vertices with neighbors c_{i-1} and c_{i+1}, and let L_i denote the set of leaf vertices with neighbor c_i. We denote by i the *index* of D_i and L_i. Also, we call $A = \bigcup_i D_i \cup L_i$ *anchors*, and we call $G[A]$ the *induced anchor graph.*

We denote by $E = V(G) \setminus (A \cup C)$ the set of vertices in G which are neither anchors nor in the base graph, and we call these vertices *linkers.* Note that E is anticomplete to C. Moreover, the components of E are singletons or edges, and any edge component must be anticomplete to L_i. We call $G[E]$ the *induced linker graph.*

This concludes Bonomo *et al.*'s [3] characterization. Now, assume for purposes of contradiction that G has no constant bounded dominating set. As such, we must have a non-constant set of linkers E' with pairwise disjoint neighborhoods.[3] Throughout this proof, we will delete vertices from E', although in such a way that E' remains a non-constant size. When we state properties of vertices in or relating to E', we mean more precisely that we can prune E' such that E' remains a non-constant size and the property holds.

Since the components of the induced linker graph are singletons and edges, and each linker must have degree ≥ 3, each linker must be adjacent to at least 2 anchors. Note that there are a constant number of types and indices that any given anchor can have. We can use pigeonhole principle to delete vertices in E' such that we are left with a non-constant set of linkers, where each linker is adjacent to the same two types and indices of anchors (see Figure 4.2).

Moreover, we can prune E' such that a non-constant number of vertices remain and E' is stable. This is clear because the components of E' are singletons or edges, so we must have a non-constant number of components in E'.

We now proceed with casework on the types and indices of these two anchors. We

[3]This follows from a result by Du and Wan [8], where since G is K_4-free (by definition), $\gamma(G) \leq 11\alpha_2(G) - 5$, where $\gamma(G)$ denotes the domination number of G and $\alpha_2(G)$ denotes the 2-independence number of G. Since $\gamma(G)$ is non-constant, $\alpha_2(G)$ must be non-constant, and as such we have a non-constant set of vertices in G with pairwise disjoint neighborhoods. Since anchors must eventually share neighbors, we have a non-constant set of linkers in G with pairwise disjoint neighborhoods.

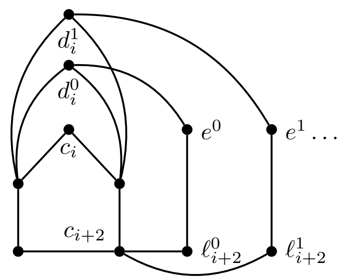


Figure 4.2: For example, here we have reduced E' to $E' = \{e^0, e^1, \ldots\}$, where each linker e^r is adjacent to a clone on index i (d_i^r) and a leaf on index $i+2$ (ℓ_{i+2}^r). Note that each e^r may be adjacent to other anchors and linkers as well.

introduce some notation to clarify this casework. Let $E' = \{e^0, e^1, \ldots\}$ (in general, the superscript refers to analogous copies, while the subscript refers to indices). For each $e^r \in E'$, let a^r and b^r denote the two anchors with the requisite types and indices as given by the pigeonhole principle, so $(e^r, a^r), (e, b^r) \in E(G)$. Note that e^r may be adjacent to other anchors or other linkers.

4.2 a^r, b^r have different types or indices

First, consider the case where a^r and b^r are anchors of different types or are anchors of different indices. There are a few cases that can be immediately eliminated.

Property 1. For all r, if a^r and b^r are both leaves, then they must be on non-adjacent indices.

Proof. If a^r and b^r are leaves on adjacent indices, say c_i and c_{i+1} respectively, then a^r, e^r, b^r, c_{i+1}, c_{i+2}, c_{i+3}, c_{i+4} forms a P_7. Thus, a^r and b^r must be on non-adjacent indices. □

Property 2. For all r, a^r and b^r do not share a neighbor in C.

Proof. If a^r and b^r share a neighbor, say c_i, then note that without loss of generality, a^r is also adjacent to c_j, where $i \neq j$ and where c_j is non-adjacent to b^r. Then, we obtain a P_7 given by $b^r, e^r, a^r, c_j, a^s, e^s, b^s$, where because G is triangle-free, a^r, a^s is non-adjacent to b^r, b^s. As such, a^r and b^r cannot share a neighbor in C. □

By Properties 1 and 2, there exist only four possibilities for the types and indices of a^r and b^r (up to isomorphism; see Figure 4.3):

1. a^r is a clone on index c_i and b^r is a clone on index c_{i+1},

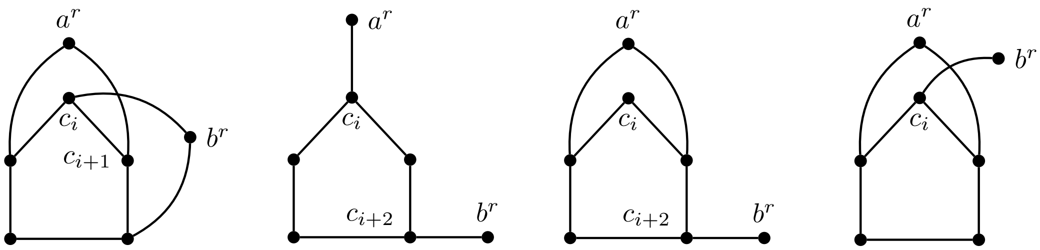


Figure 4.3: Possibilities for the types and indices of a^r and b^r, up to isomorphism.

2. a^r is a leaf on index c_i and b^r is a leaf on index c_{i+2},
3. a^r is a clone on index c_i and b^r is a leaf on index c_{i+2}, or
4. a^r is a clone on index c_i and b^r is a leaf on index c_i.

Importantly, in possibilities 1, 2, and 3, there exists j such that $a^r, c_j, c_{j+1}, c_{j+2}, b^r$ forms an induced P_5. We formalize this:

Property 3. For all r, either a^r and b^r share the same index, or there exists j such that $a^r, c_j, c_{j+1}, c_{j+2}, b^r$ forms an induced P_5.

Proof. This follows immediately from Properties 1 and 2. □

Property 4. For all $r \neq s$, if a^r and b^r do not share the same index, then a^r is adjacent to b^s.

Proof. Assume for purposes of contradiction that a^r is non-adjacent to b^s. By Property 3, there exists j such that a^r, c_j, c_{j+1}, c_{j+2}, b^r forms an induced P_5. Then, e^r, a^r, c_j, c_{j+1}, c_{j+2}, b^s, e^s forms a P_7, which is a contradiction. Thus, a^r is adjacent to b^s. □

Property 5. For all r, e^r is not part of an edge component in E.

Proof. If there exists a non-constant number of r such that e^r is a singleton, then we can simply prune all e^r such that e^r is in an edge component from E', and our property holds. Thus, assume for purposes of contradiction that there exists a non-constant number of r such that e^r is in an edge component, and prune all e^r such that e^r is a singleton from E'.

For each r, let e^r be adjacent to linker f^r. Note that necessarily, since e^r cannot be adjacent to any leaf vertices, by Property 2, a^r must be a clone on index c_i and b^r must be a clone on index c_{i+1}.

Now, for any $r \neq s$, note that f^r is adjacent to a^s if and only if f^r is adjacent to b^s. For example, if f^r is adjacent to a^s but not b^s, then $c_{i+2}, b^s, c_i, c_{i-1}, a^s, f^r, e^r$ forms a P_7. The other case follows symmetrically.

Moreover, note that if f^r is non-adjacent a^s and f^s is non-adjacent to a^r, then f^r, e^r, a^r, c_{i+1}, a^s, e^s, f^s forms a P_7. Thus, for any $r \neq s$, at least one of (f^r, a^s) and (f^s, a^r) is an edge.

Now, fixing some r, by the pigeonhole principle, there exists distinct s, t such that either $(f^r, a^s), (f^r, a^t) \in E(G)$ or $(f^r, a^s), (f^r, a^t) \notin E(G)$. In the former case, note that f^r, a^s, b^t forms a triangle (by Property 4), which is a contradiction. In the latter case, note that we must have $(f^s, a^r), (f^t, a^r) \in E(G)$. Now, if $(f^s, a^t) \in E(G)$, we again have that f^s, a^r, b^t forms a triangle. Thus, $(f^s, a^t) \notin E(G)$, which means that $(f^t, a^s) \in E(G)$. We now have that f^t, a^r, b^s forms a triangle, which is a contradiction.

Thus, for any r, e^r must not be part of an edge component in E. □

Since for any r, e^r is not part of an edge component in E, in order for $\deg(e^r) \geq 3$, e^r must be adjacent to a third anchor, say d^r. We can again use the pigeonhole principle to ensure that all e^r in E' are adjacent to a third anchor of the same type and index, and we can repeat all of our previous arguments for a^r and b^r on the pairs a^r and d^r, and b^r and d^r.

Property 6. For all r, e^r is adjacent to two anchors that are of the same type and are on the same index. Importantly, these two anchors are of the same type and index across all e^r.

Proof. By Properties 1 and 2, we see that there are precisely two cases in which neither of the pairs in $\{(a^r, d^r), (b^r, d^r)\}$ consists of two anchors of the same type and index. These are given by (up to isomorphism; see Figure 4.4):

1. a^r is a leaf on index c_i, b^r is a clone on index c_{i+2}, and d^r is a clone on index c_{i+3}, or
2. a^r is a leaf on index c_i, b^r is a clone on index c_i, and d^r is a leaf on index c_{i+3}.

The first case is impossible by Property 4 and since G is triangle-free (note that for distinct r, s, t, we have a^r, b^s, d^t forms a triangle, which is a contradiction). In the second

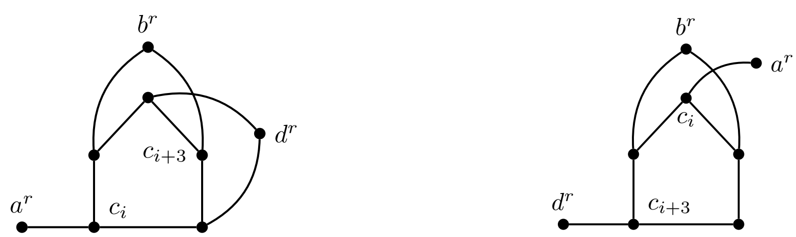


Figure 4.4: Impossible cases of the types and indices of a^r, b^r, and d^r, up to isomorphism.

case, we claim that for all $r \neq s$, a^r is adjacent to b^s. This is clear because otherwise, $b^s, c_{i+1}, c_{i+2}, c_{i+3}, d^s, a^r, e^r$ forms a P_7. Now, since G is triangle-free, we again receive a contradiction (for the same reason as in the first case).

Thus, in all cases, at least one of the pairs in $\{(a^r, d^r), (b^r, d^r)\}$ consists of two anchors of the same type and index. □

By Property 6, it suffices to use the arguments in Section 4.3 to complete the proof.

4.3 a^r, b^r have the same type and index

Consider the case where both anchors are of the same type and are on the same index. Without loss of generality, let a^r and b^r be adjacent to $c_i \in C$.

a^r dominates b^r (and vice versa)

Note that if a^r and b^r are adjacent to no other vertices, then each a^r is dominated by b^r. Thus, we must have some vertex d_a^r that is adjacent to a^r but not to b^r.

We claim that each a^r is adjacent to a distinct d_a^r of this form; that is to say, $d_a^r \neq d_a^s$ for all $r \neq s$. Note that if $d_a^r = d_a^s$, then since d_a^r is non-adjacent to b^s, we have that $b^s, e^s, a^s, d_a^r, a^r, e^r, b^r$ forms a P_7. Thus, $d_a^r \neq d_a^s$.

Similarly, each b^r is dominated by a^r, so we must have an analogous vertex d_b^r that is adjacent to b^r but not to a^r. Note that $d_b^r \neq d_b^s$ for all $r \neq s$.

By the pigeonhole principle, we can delete vertices in E' such that we are left with a non-constant set of linkers, where the corresponding d_a^r are either all er vertices or all anchors of the same type and the same index. We repeat this process with d_b^r.

We now note several properties of d_a^r and d_b^r.

Property 7. For all $r \neq s$, d_a^r is adjacent to a^s if and only if d_a^r is adjacent to b^s. Similarly, d_b^r is adjacent to a^s if and only if d_b^r is adjacent to b^s.

Proof. This property holds because otherwise, we can discover an induced P_7. For example, if d_a^r is adjacent to a^s but not b^s, then $b^s, e^s, a^s, d_a^r, a^r, e^r, b^r$ forms a P_7. The other cases follow analogously. □

Property 8. For all $r \neq s$, d_a^r is non-adjacent to e^s. Similarly, d_b^r is non-adjacent to e^s.

Proof. Let D_a denote the set of all d_a^r for all r. Note that each d_a^r can be adjacent to at most one vertex in E', since the vertices in E' have pairwise disjoint neighborhoods, so there at most $|E'|$ edges between D_a and E' in G.

Now, we can construct a new graph H where $V(H) = \{h_r \;\forall\; r\}$ and $E(H) = \{(h_r, h_s) \mid (d_a^r, e_s) \in E(G)\}$. Note that necessarily, $|E(H)| \le |E'|$, so there exists a stable set of size at least $\sum_{h_r \in H}(1 + \deg(h_r))^{-1} \ge |E'|/3$. [21][4] Denote this stable set by S_H, and for each $e_r \in E'$, if $h_r \notin S_H$, remove e_r from E'.

Note that we will still have a non-constant number of vertices in E' remaining, since $|S_H| \ge |E'|/3$. Moreover, we now have no edges between D_a and E' in G, since d_a^r is by definition non-adjacent to e^r. Thus, for any $r \ne s$, d_a^r is non-adjacent to e^s. The other case follows analogously. □

Property 9. For all $r \ne s$, d_a^r is non-adjacent to d_a^s. Similarly, d_a^r is non-adjacent to d_b^s and d_b^r is non-adjacent to d_b^s.

Proof. This property holds because otherwise, there exists a P_7. For example, if d_a^r is adjacent to d_a^s, then $e^r, a^r, d_a^r, d_a^s, a^s, e^s, b^s$ forms a P_7. Note that d_a^r is non-adjacent to a^s, b^s and d_a^s is non-adjacent to a^r since G is triangle-free and by Property 7. Moreover, d_a^r is non-adjacent to e^s and d_a^s is non-adjacent to e^r by Property 8. The other cases follow similarly. □

Property 10. For all r, d_a^r is non-adjacent to d_b^r.

Proof. If there exists a non-constant number of r such that d_a^r is non-adjacent to d_b^r, then we can simply prune all e^r such that d_a^r is adjacent to d_b^r from E', and our property holds. Thus, assume for purposes of contradiction that there exists a non-constant number of r such that d_a^r is adjacent to d_b^r, and prune all e^r such that d_a^r is non-adjacent to d_b^r from E'.

Note that by Properties 8 and 9, for any $r \ne s$, it suffices to characterize the edges between $\{d_a^r, d_b^r\}$ and $\{a^s, b^s\}$, and between $\{d_a^s, d_b^s\}$ and $\{a^r, b^r\}$. By Property 7, any edge to a^r and a^s from these sets defines the edges to b^r and b^s, so it suffices to discuss only a^r and a^s here. We claim that without loss of generality, the only edges that exist between these sets are either $\{(d_a^r, a^s), (d_a^s, a^r)\} \subseteq E(G)$ or $\{(d_a^r, a^s), (d_b^s, a^r)\} \subseteq E(G)$.

Note that if there are no edges between any of these sets, then $d_b^r, d_a^r, a^r, c_i, a^s, d_a^s, d_b^s$ forms a P_7. Thus, without loss of generality we must have $(d_a^r, a^s) \in E(G)$. Moreover, if d_a^s and d_b^s are non-adjacent to a^r, then $b^r, e^r, a^r, d_a^r, a^s, d_a^s, d_b^s$ forms a P_7. Thus, either $(d_a^s, a^r) \in E(G)$ or $(d_b^s, a^r) \in E(G)$. Note that no other edges between these sets can be added to $E(G)$, since otherwise we will have a triangle. We have shown our claim.

[4]Note that this follows from the arithmetic mean-harmonic mean (AM-HM) inequality, and since $\sum_{h_r \in H} \deg(h_r) = 2 \cdot |E'|$.

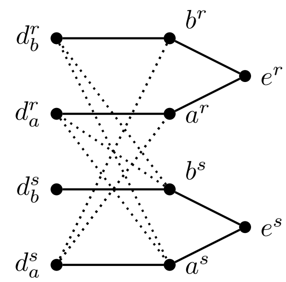


Figure 4.5: An induced subgraph of G depicting the edges relating to d_a^r and d_b^r, based on the properties proven in Section 4.3, where $r \neq s$. Note that proven non-edges are not drawn, and the dotted edges denote edges that may or may not exist for any $r \neq s$.

Now, consider any distinct r, s, t. We first claim that we cannot have the following scenario: d_a^r is either adjacent or non-adjacent to both a^s and a^t, and d_a^s is either adjacent or non-adjacent to both a^r and a^t. If we have the stated scenario, then there exists a P_7; for example, if d_a^r is adjacent to both a^s and a^t, and d_a^s is adjacent to both a^r and a^t, then our P_7 is given by $e^r, b^r, d_a^s, b^t, d_a^r, b^s, e^s$. The other cases follow similarly.[5]

Thus, without loss of generality, d_a^r must be adjacent to precisely one of a^s and a^t, and d_a^s must be adjacent to precisely one of a^r and a^t. There exists a P_7 in any case; for example, if d_a^r is adjacent to a^s and d_a^s is adjacent to a^t, then $b^r, e^r, a^r, d_a^r, a^s, d_a^s, a^t$ forms a P_7. The other cases follow similarly.[6]

This concludes all cases, so d_a^r must be non-adjacent to d_b^r. □

Figure 4.5 depicts the edges and non-edges relating to d_a^r and d_b^r, based on the properties proven in this section.

4.3.1 c_i dominates e^r

If e^r is adjacent to no other vertices, then e^r is dominated by c_i. Thus, there exists some vertex d_e^r that is adjacent to e^r but not c_i. Note that necessarily, this vertex is distinct from all of the vertices introduced thus far; for all $r \neq s$, since d_a^s and d_b^s are non-adjacent to e^r, we have $d_e^r \neq d_a^s, d_b^s$. Also, for all $r \neq s$, there must exist a distinct $d_e^r \neq d_e^s$, since e^r and e^s have disjoint neighborhoods.

We once again use the pigeonhole principle to delete vertices in E' such that we are left

[5]Note that by our earlier claim, if d_a^r is non-adjacent to both a^s and a^t, then necessarily d_b^r is adjacent to both a^s and a^t, which we use to form the desired P_7. This similarly applies to d_a^s.

[6]Again, if we fix that d_a^r is adjacent to a^s without loss of generality, in the case where d_a^s is adjacent to a^r, we can use d_b^s instead of d_a^s and b^s instead of a^s to obtain the necessary adjacencies, since d_b^s must be adjacent to a^t.

with a non-constant set of linkers where the corresponding d^r_e are either all er vertices or all anchors of the same type and index.

Now, let $G_r = \{d^r_a, d^r_b, d^r_e, a^r, b^r, e^r\}$. We would like to make two claims:

Property 11. For all r, the only edges within G_r are those given by definition, namely $(a^r, e^r), (b^r, e^r), (d^r_e, e^r), (d^r_a, a^r), (d^r_b, b^r) \in E(G)$.

Property 12. For all $r \neq s$, excepting potential edges between the sets $\{a^r, b^r, d^r_a, d^r_b\}$ and $\{a^s, b^s, d^s_a, d^s_b\}$, the only edges between G_r and G_s are $(d^r_e, a^s), (d^r_e, b^s), (d^s_e, a^r), (d^s_e, b^r) \in E(G)$.

Note that since G is triangle-free and by Property 10, in order to show Property 11, it suffices to show that $(d^r_a, d^r_e), (d^r_b, d^r_e) \notin E(G)$. Moreover, we can show the following simple property of edges between G_r and G_s.

Property 13. For all $r \neq s$, d^r_e is non-adjacent to d^s_e.

Proof. If d^r_e is an anchor, then this follows trivially because G is triangle-free. Otherwise, if d^r_e is a linker, then this follows because the components of er vertices are either singletons or edges. Thus, d^r_e is non-adjacent to d^s_e. □

Because of this property and because G is triangle-free, in order to prove Property 12, it suffices to show that $(d^r_e, a^s), (d^r_e, b^s), (d^s_e, a^r), (d^s_e, b^r) \in E(G)$.

We now proceed with casework on d^r_e, and in each case we will show that (d^r_a, d^r_e), $(d^r_b, d^r_e) \notin E(G)$ and (d^r_e, a^s), (d^r_e, b^s), (d^s_e, a^r), $(d^s_e, b^r) \in E(G)$.

d^r_e is an anchor

First, assume that d^r_e is an anchor. Note that since d^r_e is an anchor, and by construction is of a different type or index than a^r and b^r, all of the arguments in Section 4.2 regarding a^r and b^r apply to d^r_e and a^r, and to d^r_e and b^r. Thus, in order to prove Property 12, by Property 4, it suffices to consider the case where a^r, b^r, and d^r_e are anchors on the same index. We will use liberal casework on d^r_a.

Property 14. For all $r \neq s$, if a^r, b^r, and d^r_e are anchors on the same index and if d^r_a is a linker, then $(d^r_e, a^s), (d^r_e, b^s), (d^s_e, a^r), (d^s_e, b^r) \in E(G)$.

Proof. Assume for purposes of contradiction that the property does not hold. Consider any $x^r \in \{a^r, b^r\}$ and $x^s \in \{a^s, b^s\}$. If d^r_e and d^s_e are non-adjacent to x^s and x^r respectively, then $d_e^r, e^r, x^r, c_i, x^s, e^s, d_e^s$ forms a P_7 by Property 8. Thus, without loss of generality, d_e^r is adjacent to a^s and b^s. We now proceed with casework on the type of a^r.

First, consider the case where a^r is a leaf. Then, a^r and b^r are adjacent to only c_i in C, and d_e^r is adjacent to c_{i-1} and c_{i+1}. Note that for any r, we must have d_a^r is adjacent to d_e^r; otherwise, $c_{i+3}, c_{i+2}, c_{i+1}, d_e^r, e^r, a^r, d_a^r$ forms a P_7. Also, if d_e^s is adjacent to a^r but not b^r, then $c_{i+3}, c_{i+2}, c_{i+1}, d_e^s, a^r, e^r, b^r$ forms a P_7; a similar argument applies if d_e^s is adjacent to b^r but not a^r, so necessarily, d_e^s is non-adjacent to both a^r and b^r. Finally, note that we must have d_a^s is adjacent to a^r, since otherwise, $e^r, a^r, c_i, b^s, e^s, d_e^s, d_a^s$ forms a P_7. Now, we have that $c_{i+3}, c_{i+2}, c_{i+1}, d_e^s, d_a^s, a^r, e^r$ forms a P_7, which is a contradiction.

Consider the case where a^r is a clone. Then, a^r and b^r are adjacent to c_i and c_{i+2}, and d_e^r is adjacent to c_{i+1}. For any r, we must have d_a^r is non-adjacent to d_e^r; otherwise, $c_{i+4}, c_{i+3}, c_{i+2}, b^r, e^r, d_e^r, d_a^r$ forms a P_7. Similarly, d_a^r must be non-adjacent to d_e^s, since otherwise, $c_{i+4}, c_{i+3}, c_{i+2}, a^r, d_a^r, d_e^s, e^s$ forms a P_7.

Now, if d_e^s is non-adjacent a^r, then we have $d_a^r, a^r, e^r, d_e^r, c_{i+1}, d_e^s, e^s$ forms a P_7. Necessarily, d_e^s is adjacent to a^r and non-adjacent to b^r.

We claim that d_a^s is non-adjacent to a^r, b^r, since otherwise, $b^s, d_e^r, c_{i+1}, d_e^s, a^r, d_a^s, b^r$ forms a P_7. Now, $b^r, e^r, a^r, d_e^s, e^s, a^s, d_a^s$ forms a P_7, which is a contradiction.

In all cases, we reach a contradiction, so we must have $(d_e^r, a^s), (d_e^r, b^s), (d_e^s, a^r), (d_e^s, b^r) \in E(G)$. □

Property 15. For all $r \neq s$, if a^r, b^r, and d_e^r are anchors on the same index and if d_a^r is an anchor, then $(d_e^r, a^s), (d_e^r, b^s), (d_e^s, a^r), (d_e^s, b^r) \in E(G)$.

Proof. Assume for purposes of contradiction that the property does not hold. As in the proof of Property 14, consider any $x^r \in \{a^r, b^r\}$ and $x^s \in \{a^s, b^s\}$. If d_e^r and d_e^s are non-adjacent to x^s and x^r respectively, then $d_e^r, e^r, x^r, c_i, x^s, e^s, d_e^s$ forms a P_7 by Property 8. Thus, without loss of generality, d_e^r is adjacent to a^s and b^s.

Let d_a^r be adjacent to c_k. Precisely one of (d_a^r, a^s) and (d_a^s, a^r) must be an edge. If neither are edges, then b^r, e^r, a^r, d_a^r, c_k, d_a^s, a^s forms a P_7. If both are edges, then e^s, b^s, d_a^r, c_k, d_a^s, b^r, e^r forms a P_7.

We proceed with casework on d_e^r.

First, consider the case where d_e^r is also adjacent to c_k. d_e^s cannot be non-adjacent to both a^r and b^r, since otherwise, $b^r, e^r, a^r, d_a^r, c_k, d_e^s, e^s$ forms a P_7. Moreover, d_a^s must be adjacent to a^r, since otherwise, $b^r, e^r, a^r, d_e^s, c_k, d_a^s, a^s$ (if d_e^s is adjacent to a^r; the case where d_e^s is adjacent to b^r follows similarly).

If d_e^s is adjacent to a^r, then $b^s, e^s, d_e^s, c_k, d_a^s, b^r, e^r$ forms a P_7. If d_e^s is adjacent to b^r, then $a^s, e^s, d_e^s, c_k, d_a^r, a^r, e^r$ forms a P_7. Thus, we receive a contradiction in either scenario.

Consider the case where d_e^r is non-adjacent to c_k. If a^r is a leaf, then note that necessarily, d_a^r is a leaf on index c_{i+2}. Then, if without loss of generality (d_a^s, a^r) is a non-edge, we have $d_a^s, c_{i+2}, c_{i+3}, c_{i+4}, c_i, a^r, e^r$ forms a P_7, which is a contradiction. Thus, a^r must be a clone. We have a^r is adjacent to c_i and c_{i+2}, and d_e^r is adjacent to c_{i+1}. Moreover, d_a^r must be a leaf on index c_{i+3}.

We now claim that $\{d_a^r, d_a^s\}$ is complete to $\{d_e^r, d_e^s\}$. Otherwise, if for example d_a^r is non-adjacent to d_e^r, then $d_a^r, c_{i+3}, c_{i+4}, c_i, c_{i+1}, d_e^r, e^r$ forms a P_7. The other cases follow similarly. Moreover, d_e^s must be adjacent to b^r, since otherwise, $e^r, b^r, c_i, c_{i+1}, d_e^s, d_a^r, c_{i+3}$ forms a P_7.

Now, d_a^s is adjacent to a^r, since otherwise, $e^r, a^r, c_i, c_{i+1}, d_e^s, d_a^s, c_{i+3}$ forms a P_7. Finally, we have $d_a^r, c_{i+3}, d_a^s, b^r, c_i, b^s, e^s$ forms a P_7, which is a contradiction.

In all cases, we reach a contradiction, so we must have $(d_e^r, a^s), (d_e^r, b^s), (d_e^s, a^r), (d_e^s, b^r) \in E(G)$. □

Property 16. For all $r \neq s$, if d_e^r is an anchor, then $(d_e^r, a^s), (d_e^r, b^s), (d_e^s, a^r), (d_e^s, b^r) \in E(G)$.

Proof. By Property 4, it suffices to consider the case where a^r, b^r, and d_e^r are anchors on the same index. By Properties 14 and 15, this property holds. □

Property 17. For all r, if d_e^r is an anchor, then d_a^r is non-adjacent to d_e^r. Symmetrically, if d_e^r is an anchor, then d_b^r is non-adjacent to d_e^r.

Proof. If there exists a non-constant number of r such that d_e^r is non-adjacent to d_a^r, then we can simply prune all e^r such that d_e^r is adjacent to d_a^r from E', and our property holds. Thus, assume for purposes of contradiction that there exists a non-constant number of r such that d_e^r is adjacent to d_a^r, and prune all e^r such that d_e^r is non-adjacent to d_a^r from E'.

Now, $d_a^r, d_e^r, b^s, c_i, b^r, d_a^s, d_e^s$ forms a P_7. This is a contradiction, so we must have d_a^r is adjacent to d_e^r. The other case follows symmetrically. □

d_e^r is a linker

Let d_e^r be a linker. Since (e^r, d_e^r) forms an edge component in E, it must be anticomplete to any leaf vertex. Thus, a^r and b^r are clones; without loss of generality, let a^r and b^r be adjacent to c_i and c_{i+2}.

Property 18. For all r, if d_e^r is a linker, then d_a^r is a linker. Similarly, if d_e^r is a linker, then d_b^r is a linker.

Proof. Assume for purposes of contradiction that d_a^r is an anchor. d_a^r must be non-adjacent to c_i and c_{i+2}, since G is triangle-free. Let d_a^r be adjacent to c_j, where $j \neq i, i+2$.

If d_a^r is non-adjacent to a^s and d_a^s is non-adjacent to a^r, then $b^r, e^r, a^r, d_a^r, c_j, d_a^s, a^s$ forms a P_7 (by Properties 7 and 8). Without loss of generality, let d_a^r be adjacent to a^s.

If d_a^s is adjacent to a^r as well, then $e^r, b^r, d_a^s, c_j, d_a^r, b^s, e^s$ forms a P_7 (by Properties 7 and 8). Thus, d_a^s is non-adjacent to a^r.

We now proceed with casework on the type of d_a^r.

Consider the case where d_a^r is a clone. Without loss of generality, d_a^r must be adjacent to c_{i+1} and c_{i+3}. We now discover a P_7, namely $d_a^s, c_{i+3}, d_a^r, b^s, c_i$, b^r, e^r, which is a contradiction.

Consider the case where d_a^r is a leaf. Necessarily, d_a^r is non-adjacent to d_e^s (where we may have $r = s$), since (e^s, d_e^s) forms an edge in E and as such cannot be adjacent to any leaf. Consider any $x^r \in \{a^r, b^r\}$ and $x^s \in \{a^s, b^s\}$. If d_e^r and d_e^s are non-adjacent to x^s and x^r respectively, then $d_e^r, e^r, x^r, c_i, x^s, e^s, d_e^s$ forms a P_7. As such, we must have either d_e^r is adjacent to both a^s and b^s, or d_e^s is adjacent to both a^r and b^r.

If d_e^r is adjacent to both a^s and b^s, then we obtain a P_7 given by $d_a^s, c_j, d_a^r, b^s, d_e^r, e^r, b^r$, which is a contradiction. If d_e^s is adjacent to both a^r and b^r, then we obtain a P_7 given by $c_j, d_a^s, a^s, e^s, d_e^s, a^r, e^r$, which is a contradiction.

In all cases, we obtain a contradiction. Thus, d_a^r is not an anchor. The proof for d_b^r follows similarly. □

Property 19. For all r, s, if d_e^r is a linker, then d_a^r is non-adjacent to d_e^s. Similarly, if d_e^r is a linker, then d_b^r is non-adjacent to d_e^s.

Proof. Assume for purposes of contradiction that d_a^r is adjacent to d_e^s. Since the only components in E' are singletons or edges, and (d_e^s, e^s) is an edge, d_e^s must be non-adjacent to any other linker. As such, d_a^r is an anchor. This contradicts Property 18. Thus, for all r, s, d_a^r is non-adjacent to d_e^s. The other case follows symmetrically. □

Property 20. For all $r \neq s$, if d_e^r is a linker, then $(d_e^r, a^s), (d_e^r, b^s), (d_e^s, a^r), (d_e^s, b^r) \in E(G)$.

Proof. First, as in the proof of 18, we consider any $x^r \in \{a^r, b^r\}$ and $x^s \in \{a^s, b^s\}$. If d_e^r and d_e^s are non-adjacent to x^s and x^r respectively, then $d_e^r, e^r, x^r, c_i, x^s, e^s, d_e^s$ forms a P_7. As such, without loss of generality, we must have d_e^s is adjacent to both a^r and b^r.

Now, assume for purposes of contradiction that d_e^r is non-adjacent to a^s. If d_a^s is non-adjacent to a^r, then $d_e^r, e^r, a^r, d_e^s, e^s, a^s, d_a^s$ forms a P_7. Thus, d_a^s is adjacent to a^r. Moreover, if d_e^r is non-adjacent to b^s, then $d_e^r, e^r, a^r, d_a^s, a^s, e^s, b^s$ forms a P_7. Thus, d_e^r is adjacent to

b^s. Finally, if d_a^r is non-adjacent to a^s, then $d_a^r, a^r, e^r, d_e^r, b^s, e^s, a^s$ forms a P_7 (by Property 7). Thus, d_a^r is adjacent to a^s (and b^s, by Property 7).

Given these adjacencies, we note that $d_e^s, b^r, e^r, d_e^r, b^s, d_a^r, a^s$ forms a P_7. This is a contradiction, so we must have d_e^r is adjacent to a^s. By a symmetric argument, d_e^r is adjacent to b^s, as desired. □

We have now shown our desired claims. Namely,

Property 11. For all r, the only edges within G_r are those given by definition, namely $(a^r, e^r), (b^r, e^r), (d_e^r, e^r), (d_a^r, a^r), (d_b^r, b^r) \in E(G)$.

Proof. This holds by Properties 10, 17, and 19. □

Property 12. For all $r \neq s$, excepting potential edges between the sets $\{a^r, b^r, d_a^r, d_b^r\}$ and $\{a^s, b^s, d_a^s, d_b^s\}$, the only edges between G_r and G_s are $(d_e^r, a^s), (d_e^r, b^s), (d_e^s, a^r), (d_e^s, b^r) \in E(G)$.

Proof. This holds by Properties 8, 9, 13, 16, and 20. □

We make one further claim regarding the edges between G_r and G_s.

Property 21. For all $r \neq s$, d_a^r is adjacent to a^s, b^s if and only if d_b^r is adjacent to a^s, b^s.

Proof. Assume for purposes of contradiction that d_a^r is adjacent to a^s, b^s, but d_b^r is non-adjacent to a^s, b^s. Then, $d_e^r, a^s, d_a^r, a^r, d_e^s, b^r, d_b^r$ forms a P_7, by Properties 11 and 12. Thus, if d_a^r is adjacent to a^s, b^s, then d_b^r is adjacent to a^s, b^s. The other direction follows similarly. □

Figure 4.6 depicts the edges and non-edges relating to d_a^r, d_b^r, and d_e^r, based on the properties proven in this section.

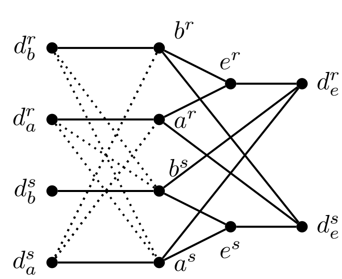


Figure 4.6: An induced subgraph of G depicting the edges relating to d_a^r, d_b^r, and d_e^r, based on the properties proven in Section 4.3.1, where $r \neq s$. Note that proven non-edges are not drawn, and the dotted edges denote edges that may or may not exist for any $r \neq s$.

4.3.2 Contradiction

We now use Properties 11 and 12 to obtain a contradiction.

Property 22. For all r, e^r is not adjacent to two anchors that are of the same type and are on the same index, where these two anchors are of the same type and index across all e^r.

Proof. Assume for purposes of contradiction that a^r and b^r are anchors of the same type and index.

For any $r \neq s$, consider G_r and G_s. By Properties 11 and 12, the only edges that we have not defined within $G[G_r \cup G_s]$ are those between d_a^r, d_b^r and a^s, b^s, and between d_a^s, d_b^s and a^r, b^r. By Property 7, it suffices to discuss only edges and non-edges to a^r and a^s. By Property 21, it suffices to discuss only edges and non-edges to d_a^r and d_a^s.

Consider any distinct r, s, t. If d_a^r is adjacent to a^s but not a^t, then we obtain a P_7 given by $e^t, a^t, d_e^s, a^r, d_a^r, a^s, d_b^r$. Thus, d_a^r is either adjacent to both a^s and a^t, or non-adjacent to both a^s and a^t. This applies symmetrically to d_a^s and d_a^t as well.

As such, without loss of generality, we must have either d_a^r is non-adjacent to a^s, a^t and d_a^s is non-adjacent to a^r, a^t, or d_a^r is adjacent to a^s, a^t and d_a^s is adjacent to a^r, a^t. In the former case, $d_a^r, a^r, d_e^s, a^t, d_e^r, a^s, d_a^s$ gives a P_7, which is a contradiction. In the latter case, $e^r, b^r, d_a^s, a^t, d_a^r, b^s, e^s$ gives a P_7, which is a contradiction.

In all cases, we obtain a contradiction. Thus, a^r and b^r cannot be anchors of the same type and index. □

Property 6 contradicts Property 22. Thus, G has a constant bounded dominating set. □

4.4 Semi-automatic proof of Theorem 4.1

We also provide a semi-automatic proof of Theorem 4.1. Specifically, the proofs in Section 4.2 and 4.3 can be automated, minus the logic to derive the existence of d_a^r, d_b^r, and d_e^r for all r, and minus Property 8.

The semi-automatic proof works by taking the structures created in Section 4.1 and considering all possible edges (in a somewhat optimized manner) to prove Properties 6 and 22. The contradiction then follows immediately.

The proofs are in file `proof.py`, with Property 6 in function `prop_6` and Property 22 in function `prop_22`. The assumption of Property 8 appears in function `set_up_nonedges`. All code can be found in Appendix A.

Chapter 5

Conclusion

In this work, we have studied the methodology of using constant bounded dominating sets to show that the 3-coloring problem on certain graph classes is polynomial. We first showed that excepting reducible configurations, $\{P_6, \text{triangle}\}$-free graphs have constant bounded dominating sets, based on Chudnovsky *et al.*'s [5] prior characterization. We also showed that excepting reducible configurations, $\{P_7, \text{triangle}\}$-free graphs have constant bounded dominating sets, building upon Bonomo *et al.*'s [3] characterization, and we provided a semi-automatic proof for this result.

In the future, we would like to extend our work to consider P_6-free and P_7-free graphs, without the triangle-free restriction. Ultimately, we hope that this work provides insight on potentially finding constant bounded dominating sets in P_8-free graphs, to address the open 3-coloring problem on P_8-free graphs.

Appendix A

Code for the semi-automatic proof of Theorem 4.1

In this appendix, we present the code for the semi-automatic proof of Theorem 4.1, as detailed in Section 4.4.

Note that the function `powerset` in the file `proof_utils.py` is taken from the Python 3.6.5 Standard Library `itertools` documentation [17]. Also, the semi-automatic proof of Property 6 requires a set of files, where each file contains a list of all labeled triangle-free graphs in graph6 format of size n, for $1 \leq n \leq 4$. The default format of these files is `k3freel/g_[n].txt`, although this can be specified in the code. We omit these files from this thesis, although we note that `nauty` and `Traces` offer relatively simple graph generation of this format [16].

```
"""
File: proof.py
Author: Jessica Shi
Date: 4/29/2018

This file contains the constructions to prove Properties 6 and 22, assuming
Property 8. The main proofs are found in prop_6() and prop_22() respectively.
"""

import proof_utils as utils
import functools
import itertools
```

```
import networkx as nx
from sympy.utilities.iterables import multiset_permutations

def add_clone(i, node, g):
  """
  Adds a clone node to graph g, with edges to ((i-1)%5) and ((i+1)%5).

  Args:
    i (int): index of clone to be added
    node (node): clone to be added
    g (Graph): graph

  Returns:
    Graph: graph with added clone
  """
  g.add_node(node, type=("clone", i))
  g.add_edges_from([(node, (i-1) % 5), (node, (i+1) % 5)])
  return g

def add_leaf(i, node, g):
  """
  Adds a leaf node to graph g, with an edge to (i % 5).

  Args:
    i (int): index of leaf to be added
    node (node): leaf to be added
    g (Graph): graph

  Returns:
    Graph: graph with added leaf
  """
  g.add_node(node, type=("leaf", i))
  g.add_edges_from([(node, i % 5)])
  return g

def add_linker(node, g):
  """
  Adds a linker node to graph g, with no edges.
```

```
  Args:
    node (node): linker to be added
    g (Graph): graph

  Returns:
    Graph: graph with added linker
  """
  g.add_node(node, type="linker")
  return g

def set_up_nonedges(num_rep, list_clones, list_linkers, list_d_clones,
                  list_d_linkers):
  """
  Sets up the non-edges for the proof of Property 22, in prop_22. Uses
  Property 8 to define certain non-edges.

  Args:
    num_rep (int): number of repetitions
    list_clones (list(nodes)): list of a^r, b^r clones
    list_linkers (list(nodes)): list of e^r linkers in E'
    list_d_clones (list(nodes)): list of d_a^r, d_b^r
    list_d_linkers (list(nodes)): list of d_e^r

  Returns:
    set(tuple(nodes)): set of specified non-edges
  """
  # Set up non-edges
  # This follows from the definitions of d_a^r, d_b^r
  nonedges_set = set([(("da",i),("b",i)) for i in range(num_rep)] +
                  [(("db",i),("a",i)) for i in range(num_rep)])

  # This is because edges to C are well-defined
  nonedges_set.update(itertools.product(list_d_clones, range(5)))
  nonedges_set.update(itertools.product(list_d_linkers, range(5)))
  nonedges_set.update(itertools.product(list_linkers, range(5)))
  nonedges_set.update(itertools.product(list_clones, range(5)))

  # This is because all vertices in E' have pairwise disjoint neighborhoods
  nonedges_set.update(itertools.product(list_d_linkers, list_linkers))
```

```
    nonedges_set.update(itertools.product(list_clones, list_linkers))

    # This is by construction of E'
    nonedges_set.update(itertools.combinations(list_linkers, 2))

    # This is by definition of C
    nonedges_set.update(itertools.combinations(range(5), 2))

    # This is because G is triangle-free
    nonedges_set.update(itertools.combinations(list_clones, 2))

    # This follows from Property 8
    nonedges_set.update(itertools.product(list_d_clones, list_linkers))

    # Add the opposite ordering of tuples to the set, for ease of lookup
    nonedges_set_opp = [nonedge[::-1] for nonedge in nonedges_set]
    nonedges_set.update(nonedges_set_opp)

    return nonedges_set

def prop_22():
    """
    Proves Property 22, given Property 8. Tests (in an optimized, recursive
    manner) all possible edges given that the linkers in E' are adjacent
    to two anchors of the same type and index and that there exist
    d_a^r, d_b^r, and d_e^r for each repetition r to fix dominating vertices.
    Shows that in all scenarios, there exists either an induced triangle
    or an induced P_7, so as such, it is not possible for the linkers in E'
    to be adjacent to two anchors of the same type and index.
    """
    isg_lst = [nx.complete_graph(3), nx.path_graph(7)]
    g_base = nx.cycle_graph(5)
    num_rep = 3

    # Set up clones, linkers, and vertices relating to dominating vertices
    list_clones = ([("a",i) for i in range(num_rep)] +
                   [("b",i) for i in range(num_rep)])
    list_linkers = [("e",i) for i in range(num_rep)]
    list_d_clones = ([("da",i) for i in range(num_rep)] +
```

```
            [("db",i) for i in range(num_rep)])
list_d_linkers = [("de",i) for i in range(num_rep)]

g_base.add_nodes_from(list_linkers)

nonedges_set = set_up_nonedges(num_rep, list_clones, list_linkers,
                               list_d_clones, list_d_linkers)

# Consider all possibilities for anchors, d_a^r, d_b^r, and d_e^r
anchor_types = [functools.partial(add_clone, 0),
                functools.partial(add_leaf, 0)]

d_func_lst_c = [add_linker]
d_func_lst_l = [add_linker]
for i in range(5):
  if i != 0:
    d_func_lst_c.append(functools.partial(add_clone, i))
    d_func_lst_l.append(functools.partial(add_leaf, i))
  d_func_lst_c.append(functools.partial(add_leaf, i))
  d_func_lst_l.append(functools.partial(add_clone, i))

for anchor_func, d_func_lst in zip(anchor_types,
                                   [d_func_lst_c, d_func_lst_l]):
  for d_func_tup in itertools.combinations_with_replacement(d_func_lst, 3):
    for (d_a_func, d_b_func, d_e_func) in multiset_permutations(d_func_tup):
      g = g_base.copy()
      # Add anchors, d_a^r, d_b^r, d_e^r
      for i in range(num_rep):
        g = anchor_func(("a",i), g)
        g = anchor_func(("b",i), g)
        g = d_a_func(("da",i), g)
        g = d_b_func(("db",i), g)
        g = d_e_func(("de",i), g)
        g.add_edges_from([(("a",i),("e",i)),(("b",i),("e",i)),
                          (("a",i),("da",i)), (("b",i),("db",i))])
      g.add_edges_from(zip(list_d_linkers, list_linkers))
      # Check all possibilities of unspecified edges, and print
      # any graphs that produce a graph without a triangle or a P7
      is_all_contra = utils.is_all_contra(g, nonedges_set, isg_lst)
```

```
      if not is_all_contra:
        print g.nodes(data="type")

def prop_6():
  """
  Proves Property 6. Tests all combinations of the types and indices of
  anchors a^r and b^r (and d^r or a second linker) adjacent to the linkers
  in E', and shows that the only allowable combinations are those in which
  at least two of {a^r, b^r, d^r} have the same type and index.

  Note that this proof is incomplete in that in the case where e^r is
  adjacent to three anchors, there are two situations in which all of
  {a^r, b^r, d^r} have different types and indices. These situations
  disappear if another repetition is added (only 2 repetitions are tested
  for this case); however, this does significantly increase the runtime.
  """
  anchor_func_lst = []
  for i in range(5):
    anchor_func_lst.append(functools.partial(add_clone, i))
    anchor_func_lst.append(functools.partial(add_leaf, i))
  base_g = nx.cycle_graph(5)
  isg_lst = [nx.complete_graph(3), nx.path_graph(7)]

  # Generate and check initial graph, with 2 repetitions and 2 anchors
  # adjacent to each linker
  fail_lst, anchor_edges_dict = utils.check_base_anchors(
    base_g, anchor_func_lst, 2, isg_lst, 2
  )

  # Consider the case where 3 anchors are adjacent to each linker
  s_fail_lst = utils.check_add_anchor(
    fail_lst, anchor_func_lst, 2, isg_lst, 2, anchor_edges_dict,
    update_dict=False
  )[0]

  # Output the cases in which it is possible for each linker in E'
  # to be attached to 3 anchors
  print "Case: 3 anchors: "
  for g in utils.only_isomorphic(s_fail_lst):
```

```
  print (g[0].nodes(data="type"))
  print (g[0].edges())

# Consider the case where each linker is adjacent to another linker
fail_lst, anchor_edges_dict = utils.check_add_linkers(
  fail_lst, anchor_func_lst, isg_lst, range(2)
)

# Add another repetition, and the linkers for that repetition
fail_lst = utils.check_add_rep(
  utils.only_isomorphic(fail_lst), anchor_func_lst, 2, isg_lst,
  2, anchor_edges_dict, update_dict=False
)[0]
fail_lst = utils.check_add_linkers(
  utils.only_isomorphic(fail_lst), anchor_func_lst, isg_lst,
  [2], update_dict=False)[0]

# Output the cases in which it is possible for each linker in E'
# to be attached to 2 anchors and an additional linker
print "Case: 2 anchors, 1 linker: "
for g in utils.only_isomorphic(fail_lst):
  print (g[0].nodes(data="type"))
  print (g[0].edges())
```

```python
"""
File: proof_utils.py
Author: Jessica Shi
Date: 4/29/2018

This file contains the functions to construct and check possible edges
in graphs with anchor and linker nodes, attached to a base graph.
While they are heavily tailored to somewhat optimally proving Properties
6 and 22, they can be used for any structure that involves a base graph
and functions to construct anchor nodes adjacent to that base graph.
"""

import copy
import itertools
import networkx as nx
import os
from operator import itemgetter
from networkx.algorithms.isomorphism import is_isomorphic
from sympy.utilities.iterables import multiset_permutations

def find_induced_subgraph(g, isg):
  """
  Checks if graph isg is an induced subgraph of graph g, and
  if so, returns one such subgraph in g.

  Args:
    g (Graph): graph to be checked
    isg (Graph): induced subgraph

  Returns:
    Graph: induced subgraph of g if isg is an induced subgraph of g,
      None otherwise
  """
  nodes = list(g)
  for set_isg in itertools.combinations(nodes, len(isg)):
    if is_isomorphic(nx.subgraph(g, set_isg), isg):
      return nx.subgraph(g, set_isg)
  return None
```

```
def find_induced_subgraphs(g, isg_lst):
  """
  Checks if any graph in isg_lst is an induced subgraph of graph g, and
  if so, returns one such subgraph in g.

  Args:
    g (Graph): graph to be checked
    isg_lst (list(Graph)): list of induced subgraphs

  Returns:
    Graph: induced subgraph of g if a graph in isg_lst is an induced subgraph
      of g, None otherwise
  """
  for isg in isg_lst:
    find_isg = find_induced_subgraph(g, isg)
    if find_isg is not None:
      return find_isg
  return None

def is_all_contra(g, nonedges_set, isg_lst):
  """
  Recusively checks all unspecified edges in graph g (where specified
  edges are given by g, and specified non-edges are given by nonedges_set),
  and determines if all of these variations of g have an induced subgraph
  in isg_lst or an induced K_4. If so, returns true, and otherwise,
  returns false.

  Args:
    g (Graph): graph to be checked
    nonedges_set (set(tuple(nodes))): set of specified non-edges in g
    isg_lst (list(Graph)): list of induced subgraphs

  Returns:
    bool: True if all possibilities of g have an induced subgraph in
      isg_lst or an induced K_4, False otherwise
  """
  isg = find_induced_subgraphs(g, isg_lst + [nx.complete_graph(4)])
  if isg is None:
```

```
    return False
  is_contra = True
  for nonedge in nx.non_edges(isg):
    if nonedge not in nonedges_set:
      nonedges_set.update([nonedge, nonedge[::-1]])
      g_new = g.copy()
      g_new.add_edge(*nonedge)
      is_contra = is_contra and is_all_contra(g_new, nonedges_set, isg_lst)
      if not is_contra:
        return is_contra
  return is_contra

def is_induced_subgraph(g, isg):
  """
  Checks if graph isg is an induced subgraph of graph g.

  Args:
    g (Graph): graph to be checked
    isg (Graph): induced subgraph

  Returns:
    bool: True if isg is an induced subgraph of g, False
          otherwise
  """
  nodes = list(g)
  for set_isg in itertools.combinations(nodes, len(isg)):
    if is_isomorphic(nx.subgraph(g, set_isg), isg):
      return True
  return False

def is_induced_subgraphs(g, isg_lst):
  """
  Checks if any graphs in isg_lst are induced subgraphs of graph g.

  Args:
    g (Graph): graph to be checked
    isg_lst (list(Graph)): list of induced subgraphs

  Returns:
```

```
    bool: True if any graph in isg_lst is an induced subgraph of g,
          False otherwise
  """
  for isg in isg_lst:
    if is_induced_subgraph(g, isg):
      return True
  return False

def is_dominated_vert(g):
  """
  Checks if there is a dominating vertex in graph g, and if so, returns
  true along with a dominating vertex. Otherwise, returns false.

  Args:
    g (Graph): graph to be checked

  Returns:
    (bool, tuple): True along with a pair consisting of a dominating vertex
      and a vertex it dominates, if there is a dominating vertex in g;
      (False, None) otherwise
  """
  nodes = list(g)
  for pair in itertools.combinations(nodes, 2):
    first = set(g.neighbors(pair[0]))
    second = set(g.neighbors(pair[1]))
    if first.issubset(second):
      return (True, pair)
    elif second.issubset(first):
      return (True, pair[::-1])
  return (False, None)

def add_anchors(g, linker, anchor_funcs, rep_idx, unique_idxs):
  """
  Adds to graph g the anchors given in anchor_funcs, each with an edge
  to linker.

  Each added anchor has the form (rep_idx, anchor_idx, unique_idx),
  where anchor_idx is given by anchor_funcs and unique_idx is given by
  unique_idxs (in order).
```

```
  Args:
    g (Graph): graph to be modified
    linker (node): node in g (that represents a linker)
    anchor_funcs (list((int, function))):
      anchors to be added; the int represents the index associated with
      the type of anchor, and the function takes as input a node and a graph,
      and adds the node to the graph as an anchor
    rep_idx (int): repetition index
    unique_idxs (list(int)):
      list of unique indices to distinguish between anchors added to the
      same repetition of the same type; must be the same length as
      anchor_funcs

  Returns:
    Graph: graph with the added anchors
    list(node): list of the added anchors
  """
  assert len(anchor_funcs) == len(unique_idxs)
  anchors = []
  for ((anchor_idx, anchor_func), unique_idx) in zip(anchor_funcs,
                                                     unique_idxs):
    anchors.append((rep_idx, anchor_idx, unique_idx))
    g = anchor_func(anchors[-1], g)
    g.add_edge(linker, anchors[-1])
  return (g, anchors)

def handle_failures(g, isg_lst, fail_lst,
                    anchor_set=None,
                    subg=None,
                    anchor_edges_dict=None):
  """
  Checks if graph g has an induced subgraph in isg_lst or an induced K_4. If
  g does have such an induced subgraph, adds g to fail_lst and updates
  anchor_edges_dict with an entry with key anchor_set and value subg if
  anchor_edges_dict is given.

  Args:
    g (Graph): graph to be checked
```

```
    isg_lst (list(Graph)): list of induced subgraphs
    fail_lst (list(Graph)): list to add g to if g has one of the specified
      induced subgraphs
    anchor_set (tuple(int)):
      tuple of sorted indices that represent the types of anchors used to
      construct g
    subg (Graph): a subgraph of g that encapsulates the edges
      between the anchors
    anchor_edges_dict (dict(tuple(int),Graph)):
      a dictionary that maps tuples of sorted anchor type indices to
      subgraphs of g that encapsulate the edges between anchors

  Returns:
    None
  """
  if (not is_induced_subgraphs(g,
                           isg_lst + [nx.complete_graph(4)])):
    fail_lst.append(g)
    if anchor_edges_dict is not None:
      if anchor_set not in anchor_edges_dict:
        anchor_edges_dict[anchor_set] = []
      anchor_edges_dict[anchor_set].append(subg)

def check_all_edges(g, subg, isg_lst, subg_fp, anchor_set=None,
                    anchor_edges_dict=None):
  """
  Runs through all possibilities of edges between the vertices in subg of
  graph g, where the possibilities are stored as graphs in graph6 format in
  folder subg_fp.

  Checks if g has an induced subgraph in isg_lst or an induced K_4, and
  updates anchor_edges_dict (if given) if g has such an induced subgraph.

  Args:
    g (Graph): graph to be checked
    subg (list(node)): list of nodes in g to try all
      possible edges of
    isg_lst (list(Graph)): list of induced subgraphs
    subg_fp (str): name of folder containing graphs in graph6 format
```

```
      that represent the possible edges in subg (importantly, not up to
      isomorphism; graphs are expected to be labeled); graphs are expected
      to be stored in files labeled "g_[num].txt", where num is the number of
      edges in the graphs in that file
    anchor_set (tuple(int)):
      tuple of sorted indices that represent the types of anchors used to
      construct g
    anchor_edges_dict (dict(tuple(int),Graph)):
      a dictionary that maps tuples of sorted anchor type indices to subgraphs
      of g that encapsulate the edges between anchors

  Return:
    list(Graph): list of graphs that have one of the specified induced
      subgraphs, considering all possible edges among subg
  """
  fail_lst = []
  if len(subg)==0:
    return [g]
  with open(subg_fp + "/g_" + str(len(subg)) + ".txt", "r") as subg_f:
    for subg_g6 in subg_f:
      subg_edges = nx.parse_graph6(subg_g6.rstrip("\n"))
      # Relabel subgraph given by subg_fp to subgraph in g
      g_mod = nx.compose(
        nx.relabel_nodes(subg_edges,
                         dict(zip(sorted(subg_edges.nodes()), subg))), g
      )
      handle_failures(g_mod, isg_lst, fail_lst,
                      anchor_set=anchor_set,
                      subg=g_mod.subgraph(subg),
                      anchor_edges_dict=anchor_edges_dict)
  return fail_lst

def check_base_anchors(base_graph, anchor_func_lst, num_anchors,
                       isg_lst, num_rep, anchor_edges_fp="k3freel",
                       update_dict=True):
  """
  Runs through all possibilities of adding num_anchors anchors from
  anchor_func_lst to the graph base_graph, with num_rep repetitions.
```

```
Considers all possible edges between anchors as given in graph6 format in
the folder anchor_edges_fp. Checks if any of these graphs has an induced
subgraph in isg_lst or an induced K_4, and returns the graphs that have no
such induced subgraphs.

Also, if update_dict is true, returns a dictionary mapping all combinations
of anchors to allowable edges between those anchors.

Args:
  base_graph (Graph): initial graph
  anchor_func_lst (list(function)):
    list of functions that take as input an anchor node and a graph, and
    add that node to the graph as an anchor
  num_anchors (int): number of anchors to be added on each repetition
  isg_lst (list(Graph)): list of induced subgraphs
  num_rep (int): number of repetitions
  anchor_edges_fp (str): folder containing all possible edges between
    anchors in graph6 format
  update_dict (bool): indicates whether to keep a dictionary mapping
    tuples of anchor type indices to allowable edges between those anchors

Returns:
  (list(Graph,list(node),list(node),tuple(int))):
    list of graphs that have none of the indicated induced subgraphs,
    with corresponding anchors, linkers, and anchor type indices
  (dict(tuple(int),Graph)): dictionary mapping tuples of anchor
    type indices to allowable edges between those anchors; None if
    update_dict is false
"""
fail_lst = []
anchor_edges_dict = {} if update_dict else None
# Consider all permutations of anchors with replacement
anchor_func_combs = itertools.combinations_with_replacement(
  enumerate(anchor_func_lst), num_anchors
)
for anchor_func_comb in anchor_func_combs:
  for anchor_funcs in multiset_permutations(anchor_func_comb):
    anchor_funcs = sorted(list(anchor_funcs), key=itemgetter(0))
    anchor_set = tuple(sorted(zip(*anchor_funcs)[0]))
```

```
    g = base_graph.copy()
    anchors = []
    linkers = []
    # In each repetition, add a linker and the corresponding anchors
    for rep_idx in range(num_rep):
      linkers.append((rep_idx, 0))
      g.add_node(linkers[-1])
      (g, new_anchors) = add_anchors(
        g, linkers[-1], anchor_funcs, rep_idx, range(num_anchors)
      )
      anchors.append(new_anchors)
    # Consider all edges between anchors and check for failure
    fail_lst += zip(
      check_all_edges(g, list(itertools.chain(*anchors)),
                      isg_lst, anchor_edges_fp,
                      anchor_set=anchor_set,
                      anchor_edges_dict=anchor_edges_dict),
      itertools.repeat(anchors),
      itertools.repeat(linkers),
      itertools.repeat(anchor_set)
    )
  fail_lst = fail_lst if update_dict else only_isomorphic(fail_lst)
  return fail_lst, anchor_edges_dict

def check_add_reps(g_spec_lst, anchor_func_lst, num_anchors, isg_lst,
                   rep_idxs, anchor_edges_dict, update_dict=True):
  """
  See check_add_rep.

  Adds multiple repetitions to each graph in g_spec_lst, as in check_add_rep.

  Args:
    g_spec_lst (list(Graph,list(node),list(node),tuple(int))):
      list of graphs with corresponding anchors, linkers, and anchor type
      indices (in order)
    anchor_func_lst (list(function)):
      list of functions that take as input an anchor node and a graph, and
      add that node to the graph as an anchor
    num_anchors (int): number of anchors to be added on each repetition
```

```
    isg_lst (list(Graph)): list of induced subgraphs
    rep_idxs (iterable(int)): indices of repetitions to be added
    anchor_edges_dict (dict(tuple(int),Graph)):
      dictionary mapping tuples of anchor type indices to allowable edges
      between those anchors; the number and repetitions of these anchors
      must match those in g_spec_lst
    update_dict (bool): indicates whether to keep an updated
      dictionary mapping tuples of anchor type indices to allowable edges
      between those anchors, considering the new repetitions

  Returns:
    (list(Graph,list(node),list(node),tuple(int))):
      list of graphs that have none of the indicated induced subgraphs,
      with corresponding anchors, linkers, and anchor type indices
    (dict(tuple(int),Graph)): dictionary mapping tuples of anchor
      type indices to allowable edges between those anchors, considering
      the new repetitions; None if update_dict is false
  """
  for rep_idx in rep_idxs:
    g_spec_lst, anchor_edges_dict = check_add_rep(
      g_spec_lst, anchor_func_lst, num_anchors, isg_lst, rep_idx,
      anchor_edges_dict,
      update_dict=True if rep_idx != list(rep_idxs)[-1] else update_dict
    )
  return g_spec_lst, anchor_edges_dict

def check_add_rep(g_spec_lst, anchor_func_lst, num_anchors, isg_lst,
                  rep_idx, anchor_edges_dict, update_dict=True):
  """
  Given a list of graphs with their corresponding anchors and linkers
  (in g_spec_lst), adds another repetition to each graph; that is to
  say, adds another linker to each graph adjacent to anchors of the same
  type as those used to construct previous anchors.

  Considers all possible edges between the newly added anchors and the
  previous anchors, using a dictionary of allowable edges to reduce
  possibilities.

  Returns all graphs that have neither an induced subgraph in isg_lst or an
```

```
induced K_4.

If update_dict is true, also returns a dictionary mapping all combinations
of anchors to allowable edges between those anchors, considering the new
repetition.

Args:
  g_spec_lst (list(Graph,list(node),list(node),tuple(int))):
    list of graphs with corresponding anchors, linkers, and
    anchor type indices (in order)
  anchor_func_lst (list(function)):
    list of functions that take as input an anchor node and a graph,
    and add that node to the graph as an anchor
  num_anchors (int): number of anchors to be added on each repetition
  isg_lst (list(Graph)): list of induced subgraphs
  rep_idx (int): index of repetition to be added
  anchor_edges_dict (dict(tuple(int),Graph)):
    dictionary mapping tuples of anchor type indices to allowable
    edges between those anchors; the number and repetitions of
    these anchors must match those in g_spec_lst
  update_dict (bool): indicates whether to keep an updated dictionary
    mapping tuples of anchor type indices to allowable edges between
    those anchors, considering the new repetitions

Returns:
  (list(Graph,list(node),list(node),tuple(int))):
    list of graphs that have none of the indicated induced subgraphs,
    with corresponding anchors, linkers, and anchor type indices
  (dict(tuple(int),Graph)): dictionary mapping tuples of anchor
    type indices to allowable edges between those anchors, considering
    the new repetitions; None if update_dict is false
"""
fail_lst = []
anchor_edges_dict_new = {} if update_dict else None
for (g, anchors, linkers, anchor_set) in g_spec_lst:
  if anchor_set not in anchor_edges_dict:
    continue
  fail_g_lst = []
  anchor_funcs = [(anchor_set_idx, anchor_func_lst[anchor_set_idx])
```

```
                for anchor_set_idx in anchor_set]
g_anch = g.copy()
# Add a linker and corresponding anchors for the new repetition
linkers.append((rep_idx, 0))
g_anch.add_node(linkers[-1])
(g_anch, new_anchors) = add_anchors(
  g_anch, linkers[-1], anchor_funcs, rep_idx, range(num_anchors))
# Consider all permutations of edges with repetition allowed
# between all combinations of the newly added anchors and the
# previous anchors
edges_combs = itertools.combinations_with_replacement(
  anchor_edges_dict[anchor_set], len(anchors)
)
for edges_comb in edges_combs:
  for edges_lst in multiset_permutations(edges_comb):
    g_mod = g_anch.copy()
    # Add specified edges between the newly added anchors and
    # the previous anchors
    for (edges_idx, edges) in enumerate(edges_lst):
      map_anchors = (anchors[0:edges_idx] +
                     anchors[edges_idx+1:] +
                     [new_anchors])
      dict_anchors = [(rep_anch_idx,) + anchor[1:]
                      for (rep_anch_idx,
                           map_anchor) in enumerate(map_anchors)
                      for anchor in map_anchor]
      map_anchors = list(itertools.chain(*map_anchors))
      relabel = nx.relabel_nodes(edges,
                                 dict(zip(dict_anchors, map_anchors)))
      g_mod = nx.compose(relabel,g_mod)
    # Check for failures in the new graph with specified edges
    subg = g_mod.subgraph(list(itertools.chain(*anchors)) + new_anchors)
    handle_failures(g_mod, isg_lst, fail_g_lst, anchor_set=anchor_set,
                    subg=subg, anchor_edges_dict=anchor_edges_dict_new)
# Consolidate failed graphs with their anchors, linkers, and anchor set
anchors.append(new_anchors)
fail_lst += zip(fail_g_lst,
                itertools.repeat(anchors),
                itertools.repeat(linkers),
```

```
                  itertools.repeat(anchor_set))
  fail_lst = fail_lst if update_dict else only_isomorphic(fail_lst)
  return fail_lst, anchor_edges_dict_new

def powerset(iterable):
  """
  From the Python Standard Library itertools documentation.
  Generates the powerset of iterable.
  """
  s = list(iterable)
  return itertools.chain.from_iterable(itertools.combinations(s, r)
                                    for r in range(len(s)+1))

def check_add_anchors(g_spec_lst, anchor_func_lst, anchor_idxs, isg_lst,
                    num_rep, anchor_edges_dict, update_dict=True):
  """
  See check_add_anchor.

  Adds multiple anchors in each repetition to each graph in g_spec_lst, as
  in check_add_anchor.

  Args:
    g_spec_lst (list(Graph,list(node),list(node),tuple(int))):
      list of graphs with corresponding anchors, linkers, and
      anchor type indices (in order)
    anchor_func_lst (list(function)):
      list of functions that take as input an anchor node and a graph,
      and add that node to the graph as an anchor
    anchor_idxs (iterable(int)): indices of anchors to be added
    isg_lst (list(Graph)): list of induced subgraphs
    num_rep (int): number of repetitions of the graphs in g_spec_lst
    anchor_edges_dict (dict(tuple(int),Graph)):
      dictionary mapping tuples of anchor type indices to allowable
      edges between those anchors; the number and repetitions of
      these anchors must match those in g_spec_lst
    update_dict (bool): indicates whether to keep an updated dictionary
      mapping tuples of anchor type indices to allowable edges
      between those anchors, considering the new anchors
```

```
    Returns:
      (list(Graph,list(node),list(node),tuple(int))):
        list of graphs that have none of the indicated induced subgraphs,
        with corresponding anchors, linkers, and anchor type indices
      (dict(tuple(int),Graph)): dictionary mapping tuples of anchor
        type indices to allowable edges between those anchors, considering
        the new anchors; None if update_dict is false
    """
    for anchor_idx in anchor_idxs:
      g_spec_lst, anchor_edges_dict = check_add_anchor(
        g_spec_lst, anchor_func_lst, anchor_idx, isg_lst, num_rep,
        anchor_edges_dict,
        update_dict=(True if anchor_idx != list(anchor_idxs)[-1]
                     else update_dict)
      )
    return g_spec_lst, anchor_edges_dict

  def check_add_anchor(g_spec_lst, anchor_func_lst, anchor_idx,
                       isg_lst, num_rep, anchor_edges_dict,
                       update_dict=True):
    """
    Given a list of graphs with their corresponding anchors and linkers
    (in g_spec_lst), adds another anchor to each linker in each graph
    (where anchors are of the same type).

    Considers all possible edges between the newly added anchors and the
    previous anchors, using a dictionary of allowable edges to reduce
    possibilities.

    Returns all graphs that have neither an induced subgraph in isg_lst or an
    induced K_4.

    If update_dict is true, also returns a dictionary mapping all combinations
    of anchors to allowable edges between those anchors, considering the
    newly added anchors.

    Args:
      g_spec_lst (list(Graph,list(node),list(node),tuple(int))):
        list of graphs with corresponding anchors, linkers, and
```

```
      anchor type indices (in order)
    anchor_func_lst (list(function)):
      list of functions that take as input an anchor node and a graph,
      and add that node to the graph as an anchor
    anchor_idx (int): index of anchors to be added
    isg_lst (list(Graph)): list of induced subgraphs
    num_rep (int): number of repetitions of the graphs in g_spec_lst
    anchor_edges_dict (dict(tuple(int),Graph)):
      dictionary mapping tuples of anchor type indices to allowable
      edges between those anchors; the number and repetitions of
      these anchors must match those in g_spec_lst
    update_dict (bool): indicates whether to keep an updated dictionary
      mapping tuples of anchor type indices to allowable edges between
      those anchors, considering the new anchors

  Returns:
    (list(Graph,list(node),list(node),tuple(int))):
      list of graphs that have none of the indicated induced subgraphs,
      with corresponding anchors, linkers, and anchor type indices
    (dict(tuple(int),Graph)): dictionary mapping tuples of anchor
      type indices to allowable edges between those anchors, considering
      the new anchors; None if update_dict is false
  """
  anchor_edges_dict_new = {} if update_dict else None
  fail_lst = []
  for (g, anchors, linkers, anchor_set) in g_spec_lst:
    # Consider all possible anchor types to add
    for (anchor_func_idx, anchor_func) in enumerate(anchor_func_lst):
      g_anch = g.copy()
      new_anchors = []
      # Add a new anchor to each repetition
      for rep_idx in range(num_rep):
        (g_anch, new_anchors_temp) = add_anchors(
          g_anch, (rep_idx,0), [(anchor_func_idx, anchor_func)], rep_idx,
          [anchor_idx]
        )
        new_anchors += new_anchors_temp
      # Consider all possible edges between the new anchors
      # and the previous anchors
```

```
edges_comb = [[] for _ in range(len(anchor_set))]
to_cont = False
for anchor_set_idx in range(len(anchor_set)):
  anchor_subset = tuple(sorted(anchor_set[:anchor_set_idx] +
                           anchor_set[anchor_set_idx+1:] +
                           (anchor_func_idx,)))
  if anchor_subset in anchor_edges_dict:
    edges_comb[anchor_set_idx] = anchor_edges_dict[anchor_subset]
  else:
    to_cont = True
    break
if to_cont:
  continue
# Iterate through all possible edges between the new anchors
# and the previous anchors
fail_g_lst = []
for edges_lst in itertools.product(*edges_comb):
  g_mod = g_anch.copy()
  # Add the specified edges
  for (edges_idx, edges) in enumerate(edges_lst):
    anchors_subset = [[anchor for anchor in anchor_lst
                       if anchor[2] != edges_idx]
                      for anchor_lst in anchors]
    map_anchors = (list(itertools.chain(*anchors_subset)) +
                  [(rep_idx, anchor_func_idx, anchor_idx)
                   for rep_idx in range(num_rep)])
    sort_anchors = zip(*sorted(
      anchors_subset[0] + [(0, anchor_func_idx, anchor_idx)],
      key=itemgetter(1)
    ))[2]
    dict_anchor_idx = dict(zip(sort_anchors, range(len(anchor_set))))
    dict_anchors = [anchor[0:2] + (dict_anchor_idx[anchor[2]],)
                   for anchor in map_anchors]
    relabel = nx.relabel_nodes(edges,
                           dict(zip(dict_anchors, map_anchors)))
    g_mod = nx.compose(relabel, g_mod)
  # Check for failures in the new graph with the specified edges
  subg = g_mod.subgraph(list(itertools.chain(*anchors))+
                      new_anchors)
```

```
      handle_failures(g_mod, isg_lst, fail_g_lst,
                      anchor_set=tuple(sorted(anchor_set +
                                              (anchor_func_idx,))),
                      subg=subg, anchor_edges_dict=anchor_edges_dict_new)
    # Consolidate failed graphs with their anchors, linkers, and anchor set
    new_anchors = copy.deepcopy(anchors)
    for (anchor_lst_idx, anchor_lst) in enumerate(new_anchors):
      anchor_lst.append((anchor_lst_idx, anchor_func_idx, anchor_idx))
    fail_lst += zip(fail_g_lst,
                    itertools.repeat(new_anchors),
                    itertools.repeat(linkers),
                    itertools.repeat(tuple(sorted(anchor_set +
                                                  (anchor_func_idx,)))))
  fail_lst = fail_lst if update_dict else only_isomorphic(fail_lst)
  return fail_lst, anchor_edges_dict_new

def check_add_linkers(g_spec_lst, anchor_func_lst, isg_lst, rep_idxs,
                      update_dict=True):
  """
  Given a list of graphs with their corresponding anchors and linkers
  (in g_spec_lst), adds a new linker adjacent to each linker corresponding
  to the repetition indices in rep_idxs.

  Considers all possible edges between the newly added linkers and the anchors.

  Returns all graphs that have neither an induced subgraph in isg_lst or K_4
  as an induced subgraph.

  If update_dict is true, also returns a dictionary mapping all combinations
  of anchors to allowable edges between those anchors, considering the new
  linkers.

  Args:
    g_spec_lst (list(Graph,list(node),list(node),tuple(int))):
      list of graphs with corresponding anchors, linkers, and anchor type
      indices (in order)
    anchor_func_lst (list(function)):
      list of functions that take as input an anchor node and a graph,
      and add that node to the graph as an anchor
```

```
  isg_lst (list(Graph)): list of induced subgraphs
  rep_idxs (int): indices of repetitions to which the new linkers are added
  update_dict (bool): indicates whether to keep an updated dictionary
    mapping tuples of anchor type indices to allowable edges between those
    anchors, considering the new linkers

Returns:
  (list(Graph,list(node),list(node),tuple(int))):
    list of graphs that have none of the indicated induced subgraphs,
    with corresponding anchors, linkers, and anchor type indices
  (dict(tuple(int),Graph)): dictionary mapping tuples of anchor
    type indices to allowable edges between those anchors, considering
    the new linkers; None if update_dict is false
"""
for rep_idx in rep_idxs:
  anchor_edges_dict_new = {} if update_dict else None
  fail_lst = []
  for (g, anchors, linkers, anchor_set) in g_spec_lst:
    # Add a new linker adjacent to every linker corresponding to
    # repetition rep_idx
    g_lnk = g.copy()
    new_linker = (rep_idx, 1)
    linkers.append(new_linker)
    g_lnk.add_node(new_linker)
    g_lnk.add_edge(new_linker, (rep_idx, 0))
    # Consider all combinations of edges between the new linker and
    # the anchors
    edges_comb = itertools.product([new_linker],
                                   list(itertools.chain(*anchors)))
    fail_g_lst = []
    for edges_lst in powerset(edges_comb):
      # Add the specified edges
      g_mod = g_lnk.copy()
      g_mod.add_edges_from(list(edges_lst))
      subg = g_mod.subgraph(list(itertools.chain(*anchors)))
      # Check for failures in the new graph with the specified edges
      handle_failures(
        g_mod, isg_lst, fail_g_lst, anchor_set=anchor_set, subg=subg,
        anchor_edges_dict=(
```

```
          anchor_edges_dict_new
          if rep_idx == list(rep_idxs)[-1]
          else None
        )
      )
    # Consolidate failed graphs with their anchors, linkers, and anchor set
    fail_lst += zip(fail_g_lst,
                    itertools.repeat(anchors),
                    itertools.repeat(linkers),
                    itertools.repeat(anchor_set))
  g_spec_lst = fail_lst if update_dict else only_isomorphic(fail_lst)
  return g_spec_lst, anchor_edges_dict_new

def only_isomorphic(graphs):
  """
  Prunes graphs such that all graphs in graphs are pairwise non-isomorphic.

  Args:
    graphs (list(Graph) or list(tuple(Graph,...)):
      list of graphs to be pruned; may also be in the format of
      tuples in which the first entry is a graph and the remaining entries
      contain other info

  Returns:
    list(Graph) or list(tuple(Graph,...)):
      list of pairwise non-isomorphic graphs, where the format matches that
      of the input
  """
  iso_lst = []
  for i in range(len(graphs)):
    keep = True
    for g in graphs[i+1:]:
      if ((type(g) is tuple and nx.is_isomorphic(graphs[i][0], g[0])) or
          (type(g) is not tuple and nx.is_isomorphic(graphs[i], g))):
        keep = False
        break
    if keep:
      iso_lst.append(graphs[i])
  return iso_lst
```

References

[1] G. Bacsó and Z. Tuza. Dominating cliques in P_5-free graphs. *Periodica Mathematica Hungarica*, 21(4):303–308, Dec 1990.

[2] F. Bonomo, M. Chudnovsky, P. Maceli, O. Schaudt, M. Stein, and M. Zhong. Three-coloring and list three-coloring of graphs without induced paths on seven vertices. *Combinatorica*, May 2017.

[3] F. Bonomo, O. Schaudt, and M. Stein. 3-colouring graphs without triangles or induced paths on seven vertices. 09 2014.

[4] M. Chudnovsky. Coloring graphs with forbidden induced subgraphs. *Proceedings of the ICM*, IV:291–302, 2014.

[5] M. Chudnovsky, P. Seymour, S. Spirkl, and M. Zhong. Triangle-free graphs with no six-vertex induced path. Submitted, 2018.

[6] M. Chudnovsky, S. Spirkl, and M. Zhong. Four-coloring P_6-free graphs I. Extending an excellent precoloring. Submitted, 2018.

[7] M. Chudnovsky, S. Spirkl, and M. Zhong. Four-coloring P_6-free graphs II. Finding an excellent precoloring. Submitted, 2018.

[8] D.-Z. Du and P.-J. Wan. *CDS in Planar Graphs*, pages 183–191. Springer New York, New York, NY, 2013.

[9] K. Edwards. The complexity of colouring problems on dense graphs. *Theoretical Computer Science*, 43:337–343, 1986.

[10] C. T. Hoàng, M. Kamiński, V. Lozin, J. Sawada, and X. Shu. Deciding k-colorability of P_5-free graphs in polynomial time. *Algorithmica*, 57(1):74–81, May 2010.

[11] I. Holyer. The NP-completeness of edge-coloring. *SIAM Journal on Computing*, 10(4):718–720, 1981.

[12] S. Huang. Improved complexity results on k-coloring P_t-free graphs. *European Journal of Combinatorics*, 51:336–346, 2016.

[13] R. M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972.

[14] D. Leven and Z. Galil. NP-completeness of finding the chromatic index of regular graphs. *Journal of Algorithms*, 4(1):35–44, 1983.

[15] V. V. Lozin and M. Kamiński. Coloring edges and vertices of graphs without short or long cycles. *Contributions to Discrete Mathematics*, 2:61–66, 2007.

[16] B. D. McKay and A. Piperno. Practical graph isomorphism, {II}. *Journal of Symbolic Computation*, 60(0):94–112, 2014.

[17] Python Software Foundation. Python language reference, version 3.6.5. `https://docs.python.org/3/index.html`.

[18] B. Randerath and I. Schiermeyer. 3-colorability $\in$ P for P_6-free graphs. *Discrete Applied Mathematics*, 136(2):299–313, 2004. The 1st Cologne-Twente Workshop on Graphs and Combinatorial Optimization.

[19] D. Seinsche. On a property of the class of n-colorable graphs. *Journal of Combinatorial Theory, Series B*, 16(2):191–193, 1974.

[20] L. Stockmeyer. Planar 3-colorability is polynomial complete. *SIGACT News*, 5(3):19–25, July 1973.

[21] V. K. Wei. A lower bound on the stability number of a simple graph. Technical Report 81-11217-9, Bell Laboratories Technical Memorandum, 1981.