Rainbow matchings of size m in graphs with total color-degree at least $2mn$

Adaela Shearer

Abstract

In this MTH 501 project, we present several results from the article "Rainbow matchings of size m in graphs with total color-degree at least $2mn$" by Jürgen Kritschgau [5]. In one of the main results, this paper describes an algorithm to construct a rainbow matching for such graphs when they are given with a proper edge-coloring. Several other families of graphs are also considered, such as triangle-free graphs and graphs without 4-cycles.

A Math 501 Project

Under the direction of
Dr. J. Caughman

Second Reader
Dr. D. Garton

Submitted in partial fulfillment of the requirements for the degree of

Master of Science in Mathematics

Portland State University
May 25, 2025

Contents

1 Introduction

In this MTH 501 project, we present results from the article "Rainbow matchings of size m in graphs with total color-degree at least $2mn$" by Jürgen Kritschgau [5]. To do so, we will consider several classes of simple graphs in which the edges have been colored (sometimes arbitrarily, sometimes properly). By assuming that the average number of colors appearing at each vertex is sufficiently large, we will see that the existence of a matching with a specified number of distinctly colored edges is guaranteed. (Such a matching is called a rainbow matching.) The families of graphs to be considered will include graphs that are triangle-free, graphs that have no 4-cycles, and arbitrary graphs that are properly colored. The original paper also considers arbitrary colorings of graphs, but we limit our discussion here to the aforementioned families.

Our aim in this project is to present the main mathematical results of the paper, to fill in occasional supplementary details that were left to the reader in the original, and to offer additional commentary that will serve to illustrate results and elucidate proof methods wherever possible. We also coded an implementation of the key algorithm that was central to the proof of the main theorem.

1.1 Context and Motivation

Graph colorings form an important topic in applications of graph theory; many real-world problems modeled by graphs involve partitioning the vertices or edges of a given graph into disjoint sets such that the vertices or edges within these sets are non-adjacent or non-incident. For example, such constraints commonly arise in problems concerning scheduling, either to avoid conflicts or to minimize the total time of composite tasks when various sub-tasks might be able to be performed simultaneously. For more information regarding applications of coloring, we refer the interested reader to the textbook by West [12].

Recall that a **matching** in a graph G is any set of edges in G that share no endpoints. If the edges of G are colored, then any set of edges that is assigned distinct colors is said to be **rainbow** colored. The present paper is focused on the existence of rainbow matchings in various edge-colored graphs.

Rainbow matchings in edge-colored graphs were originally studied due to their connection to transversals of Latin squares (for more about this connection, see [8]). More recently, it was conjectured (by Li and Wang in [10]) that if $m \geq 4$, then any graph with minimum color-degree at least m contains a rainbow matching of size $\lceil m/2 \rceil$. This conjecture was confirmed, and others went on to improve the bounds (see [4], [6]). Studies on other sufficient conditions

that guarantee a rainbow matching of various sizes have also been conducted in [2] and [7].

In any edge-colored graph, the **color-degree** of a vertex v is the number of distinct colors assigned to edges incident upon v. In the paper under consideration, rainbow matchings are sought in edge-colored graphs whose vertices attain a given average color-degree.

Fix any non-negative integer m. The aspirational goal of this line of research would be to show that any edge-colored graph with average color-degree at least $2m$ must contain a rainbow matching of size m. In the present article, this is shown to be true for any edge-colored graph G that falls into any of the following cases:

1. G is triangle-free (indeed an $m+1$ rainbow matching exists), or
2. G has no 4-cycles, or
3. G has at least $8m$ vertices and the coloring is proper, or
4. G has at least $12m^2 + 4m$ vertices and the coloring is arbitrary.

For the most part, our exposition follows that of the original article. However, we focus primarily on the first three of these cases. We also include some illustrative examples and offer code to run an algorithmic implementation of the given construction for the case of properly edge-colored graphs.

1.2 Organization of Paper

For ease of reading, the content of this project is organized into six chapters. Chapter 1 contains a very brief introduction to the topic of rainbow colorings, stating the main results and describing the scope and organization of this paper. Chapter 2 summarizes some definitions concerning graphs, edge-colorings, and matchings that will be needed in our work. Chapter 3 establishes two of the main results of this paper. Specifically, given an edge-colored graph G, we prove that rainbow matchings of various sizes are shown to exist under the assumption that G is free of 3-cycles or free of 4-cycles. In Chapter 4, a key algorithm is given to prove that large rainbow matchings exist for properly edge-colored graphs with sufficiently many vertices and sufficiently high average color-degree. In that chapter, we also prove that the algorithm establishes the desired bound. In Chapter 5, we examine a Python implementation of the above algorithm. We explain how the code functions and present sample output to illustrate the behavior. We have also developed a reverse algorithm to construct the rainbow matching, whose existence has been proven. Finally, in Chapter 6, we conclude with a brief discussion of potential directions for further research.

2 Preliminary Definitions

This chapter offers some basic definitions from graph theory to provide sufficient context for the exposition. For more details on these concepts, we refer the reader to any standard textbook on graph theory (such as West [12]).

2.1 Simple Graphs

Since the main object of concern for this paper will be finite simple graphs, we now review several of the most relevant definitions.

A (finite, simple) **graph** G consists of a (finite) **vertex set** $V(G)$, and a corresponding **edge set** $E(G)$, consisting of a set of 2-element subsets of $V(G)$. Each $e \in E(G)$ is called an **edge**, and its two vertices are called its **endpoints**. We call a graph **empty** when $E(G)$ is empty, i.e., when $|E(G)| = 0$. When u and v are the two endpoints of an edge e, they are said to be **adjacent**, written $u \sim v$. In this case, we often refer to e by its endpoints, as in $e = uv$. We say an edge is **incident** with its endpoints, and we also say two edges are incident when they share an endpoint. For any vertex $u \in V(G)$, we denote by $N_G(u)$ the set of vertices adjacent to u in G. We also introduce $\overline{N_G}(u)$ to be the set of edges incident to u in G. The **degree** of a vertex u is $d_G(u) = |N_G(u)|$, which is also the number of edges that have u as an endpoint. We will drop the subscript on this notation when the context is clear.

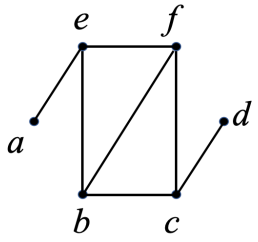

Figure 1: This graph has 6 vertices a, b, c, d, e, and f. It has 7 edges, as indicated, and we say, for example, $a \sim e$ and $a \nsim b$.

Two graphs G and H are said to be **isomorphic** (written $G \cong H$) whenever there exists a bijection $\phi : V(G) \to V(H)$ such that for any $u, v \in V(G)$ we have $u \sim v$ if and only if $\phi(u) \sim \phi(v)$.

Let G be a graph. By a **subgraph** of G, we mean any graph H satisfying $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. If H is a subgraph of G, we denote this by $H \subseteq G$. If $S \subseteq V(G)$, then

we define $G[S]$, the **subgraph induced by** S, to be the union of all subgraphs $H \subseteq G$ that have $V(H) = S$. In other words, $G[S]$ is the subgraph of G that has vertex set S and where two vertices u, v in S are adjacent in $G[S]$ if and only if they are adjacent in G. By a **matching**, we mean any set of edges M with the property that no two edges of M share an endpoint. For any non-negative integer m, a matching consisting of exactly m edges is called an m**-matching**.

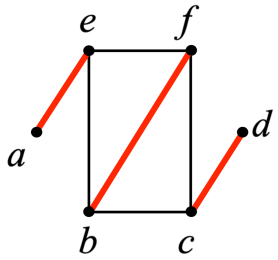

Figure 2: The bold red edges form a 3-matching in the given graph.

2.2 Edge-Colorings

Let G be a graph. We define an **edge-coloring** c on G to be an assignment of colors to edges, determined by a function $c : E(G) \to [r]$, where $[r]$ denotes the set $\{1, ..., r\}$. We refer to the elements of $[r]$ as **colors**, but since they are represented by numbers, they carry an implicit ordering. For each edge e, we refer to $c(e)$ as the color of edge e, and we define the **color-set** of the edge-coloring to be the image $c(G)$, which is the set of colors assigned to at least one edge of G. A **color-class** is the preimage of a color, so it will be a set of edges of the form $c^{-1}(i)$ for some $i \in [r]$. We occasionally use a capital letter, such as R, to denote a generic color or its corresponding color-class. A **proper edge-coloring** of G is an edge-coloring such that $c(e) \neq c(e')$ whenever e and e' share an endpoint.

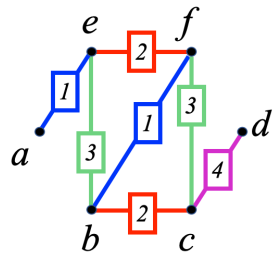

Figure 3: The graph above has a proper edge-coloring with four colors. We sometimes label colored edges with an integer to distinguish the color.

We call a graph G **rainbow** under c when c is injective on $E(G)$. In particular, a matching M is rainbow if, for all distinct $e, e' \in M$, we have $c(e) \neq c(e')$. We define the **color-degree** of a vertex $v \in V(G)$, denoted $\hat{d}_G(v)$, to be the number of colors c assigns to the edges incident to v in G. We will drop the subscript on this notation when the context is clear. Note that in a proper coloring, $\hat{d}_G(v) = d_G(v)$, the degree of $v \in G$. The **total color-degree** of G with respect to c is the sum of all the color-degrees in the graph and is denoted by $\hat{d}(G)$ as follows:

$$\hat{d}(G) = \sum_{v \in V(G)} \hat{d}(v).$$

The **average color-degree** of a graph G is obtained by dividing the total color-degree by $|V(G)|$, and is sometimes more convenient to work with than $\hat{d}(G)$ directly. The **minimum color-degree** and **maximum color-degree** of G are denoted $\hat{\delta}(G)$ and $\hat{\Delta}(G)$, respectively. For any color R, we let $\hat{d}^R(v)$ denote the **R-color-degree** of v, that is, the number of R-colored edges incident to v.

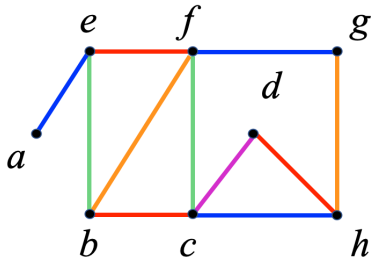

Figure 4: The graph G above uses five colors, thus has five color-classes, and is a proper coloring. If we let B be the color-class of blue edges, then $B = \{ae, fg, ch\}$. We say $\hat{d}(a) = 1$ and $\hat{d}(c) = 4$. We note also that $\hat{\delta}(G) = 1$ and $\hat{\Delta}(G) = 4$.

Finally, we let $G - v$ denote the graph G with the vertex v deleted, and similarly define $G - S$ for any subset $S \subseteq V(G)$. We may also delete sets of edges, so that, for example, $G - R$ may denote the graph G with the edges in color-class R deleted. When convenient, we let $c(e)$ denote the color-class of the edge e, so that $G - c(e)$ denotes the graph G without any of the edges that belong to the color-class containing the edge e.

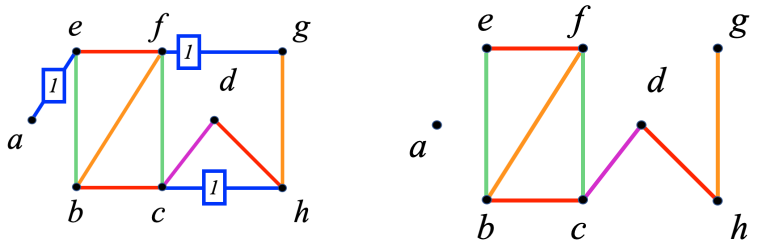

Figure 5: The above shows the removal of the color-class of blue edges. Call the graph on the left G and the set of blue edges B. Then the graph on the right represents $G-B = G-c(ae)$.

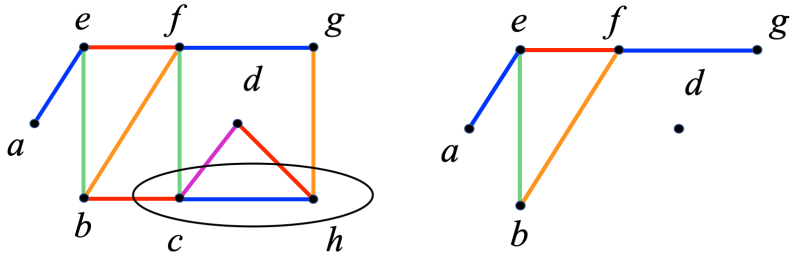

Figure 6: The above shows the removal of vertices c and h. Call the graph on the left G. Then the graph on the right can be written as $G - c - h = G - N(d)$.

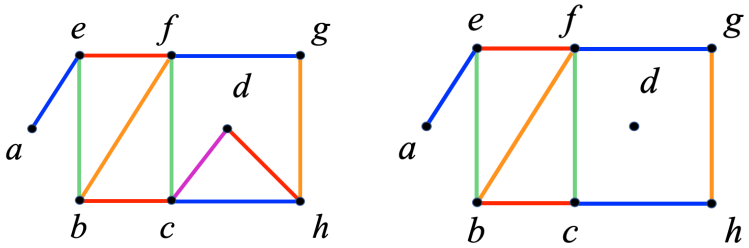

Figure 7: Alternatively, we may remove just the incident edges of a vertex. Say S is the set of edges incident to d. Then the graph on the right shows $G - S = G - \overline{N}(d)$.

3 Triangle-free and C_4-free Graphs

In this section, we consider rainbow matchings for graphs that are triangle-free or C_4-free, so we begin by defining those terms. A graph G is said to be **triangle-free** whenever it contains no subgraph isomorphic to the cyclic graph C_3. A graph G is said to be C_4**-free** whenever it contains no subgraph isomorphic to the cyclic graph C_4.

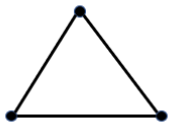

Figure 8: A graph is *triangle-free* if it has no subgraph isomorphic to the 3-cycle C_3 shown here.

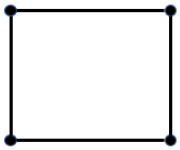

Figure 9: A graph is C_4*-free* if it has no subgraph isomorphic to the 4-cycle graph C_4 shown here.

We have an example of a triangle-free graph in Figure 10, and an example of a C_4-free graph can be seen in Figure 11.

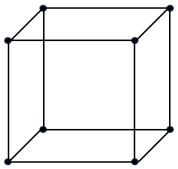

Figure 10: The cube graph Q_3 is triangle-free but not C_4 free.

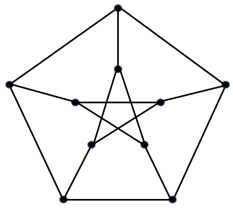

Figure 11: The Peterson graph is both C_3 and C_4 free.

3.1 Triangle-free Graphs

In the case of triangle-free graphs, we can use a counting argument to guarantee the existence of a rainbow matching of the desired size.

Theorem 3.1. *Let G be a triangle-free graph on n vertices. Let c be any edge-coloring of G where the total color-degree satisfies $\hat{d}(G) > 2mn$ for some integer m such that $0 \leq m < n$. Then c admits a rainbow matching of size $m + 1$.*

Proof. Let G be a triangle-free graph on n vertices. Let c be an edge-coloring of G with $\hat{d}(G) > 2mn$. For the sake of contradiction, let M be a maximum rainbow matching of size $k \leq m$ with edges u_iv_i for $1 \leq i \leq k$. Among all such matchings, assume M is chosen so that $\sum_{v \in V(M)} \hat{d}(v)$ is minimized. Without loss of generality, suppose $c\,(u_iv_i) = i$ for $1 \leq i \leq k$.

Claim 1. For each i $(1 \leq i \leq k)$, we have $\hat{d}(u_i) + \hat{d}(v_i) \leq n$.

Proof of Claim 1. Fix any i $(1 \leq i \leq k)$. As indicated in Figure 12, let $A = N(u_i) - \{v_i\} - N(v_i)$, let $B = N(u_i) \cap N(v_i)$, and let $C = N(v_i) - \{u_i\} - N(u_i)$. Since G has n vertices, $|A| + |B| + |C| + 2 \leq n$. Since G is triangle-free, $|B| = 0$. Now $d(u_i) = |A| + 1$ and

$d(v_i) = |C| + 1$, and therefore $d(u_i) + d(v_i) \leq n$. Since $\hat{d}(u_i) \leq d(u_i)$ and $\hat{d}(v_i) \leq d(v_i)$, the claim follows. □

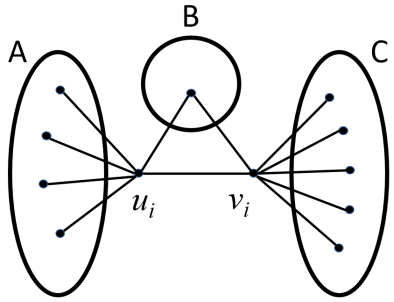

Figure 12: Since G is triangle-free, u_i and v_i cannot share a neighbor. Such a shared vertex would result in a triangle on the vertices u_i, v_i, and their shared neighbor in B.

Let H denote the subgraph induced on the vertices not in M. In other words, let $H = G[S]$ where $S = V(G) \setminus V(M)$. Suppose $e \in E(H)$. Since M has maximum size, it must be the case that $c(e) \in \{1, ..., k\}$. Without loss of generality, suppose that $c(H) = \{1, ..., j\}$ for some $0 \leq j \leq k$.

Claim 2. For all $v \in V(H)$, we have $\hat{d}(v) \leq j + k$.

Proof of Claim 2. Fix any $v \in V(H)$. Consider Figure 13. Note that since $|c(H)| = j$, we have at most j colors appearing on the edges of the set $\{vw \mid w \in N(v) \cap V(H)\}$. Since G is triangle-free, we know that $|N(v) \cap V(M)| \leq k$, as otherwise we would obtain a triangle as indicated by the dashed line. Therefore, we have at most k colors appearing on the edges of the set $\{vw \mid w \in N(v) \cap V(M)\}$. It follows that $\hat{d}_G(v) \leq j + k$, as desired. □

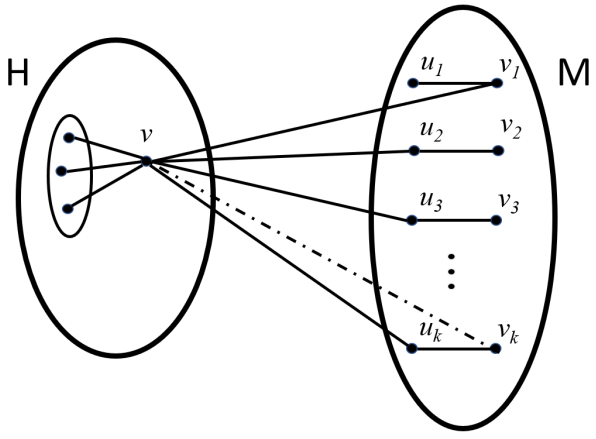

Figure 13: Illustrating the proof of Claim 2 for a vertex $v \in V(H)$.

Claim 3. For any $i \in \{1, \ldots, j\}$, we have $\hat{d}(u_i) + \hat{d}(v_i) \leq 2(j+k)$.

Proof of Claim 3. Since $i \in \{1, \ldots, j\}$, there exists an edge $xy \in H$ with $c(xy) = i$. By Claim 2, we know that $\hat{d}(x) + \hat{d}(y) \leq 2(j+k)$. Since $V(H) \cap V(M) = \varnothing$, swapping xy with u_iv_i would not change the color-set of M. But M was chosen so that $\sum_{v \in V(M)} \hat{d}(v)$ is minimized, so it follows that $\hat{d}(u_i) + \hat{d}(v_i) \leq \hat{d}(x) + \hat{d}(y)$. □

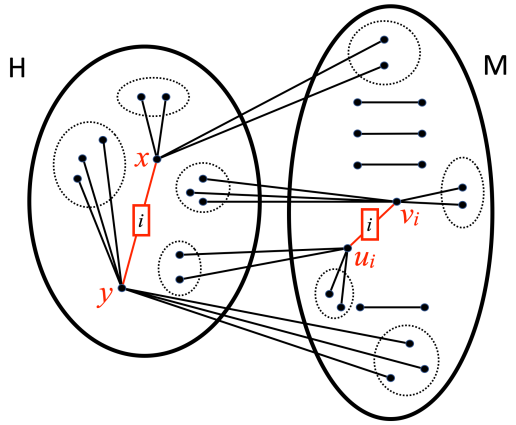

Figure 14: Consider the figure above. Any edge with one endpoint in H and one endpoint in M will be either fully in H or M after swapping edges. The total color-degree of vertices in M was minimized, so there are at least as many colors incident with xy as with u_iv_i.

Now consider the following calculation, where each step is justified below:

$$
\begin{aligned}
2mn &< \sum_{i=1}^{k} (\hat{d}(u_i) + \hat{d}(v_i)) + \sum_{v \in H} \hat{d}_G(v) && (1)\\
&\leq \sum_{i=1}^{j} (\hat{d}(u_i) + \hat{d}(v_i)) + \sum_{i=j+1}^{k} (\hat{d}(u_i) + \hat{d}(v_i)) + \sum_{v \in H} \hat{d}_G(v) && (2)\\
&\leq 2j(k+j) + (k-j)n + (n-2k)(j+k) && (3)\\
&= 2jk + 2j^2 + 2nk - 2jk - 2k^2 && (4)\\
&\leq 2(j^2 - k^2 + nk) && (5)\\
&\leq 2(k^2 + nk - k^2) && (6)\\
&\leq 2mn. && (7)
\end{aligned}
$$

Below, we offer justification for the steps above.

(1) We suppose that G has $\hat{d}(G) > 2mn$. Since the vertices of G are partitioned by M and H, we have that

$$\hat{d}(G) \leq \hat{d}(M) + \hat{d}(H) = \sum_{i=1}^{k} \hat{d}(u_i) + \hat{d}(v_i) + \sum_{v \in H} \hat{d}_G(v).$$

(2) Here we split the first summation to separately count color degrees for edges of M whose color appears in H versus the rest.

(3) Each term of the first sum is bounded by $2(k+j)$ by Claim 3. There are $k-j$ terms in the second sum, each bounded by n by Claim 1. There are $(n-2k)$ terms in the final sum, each bounded by $j+k$, by Claim 2.

(4)-(7) Algebra and simplification utilizing $j \leq k \leq m$.

This is a contradiction to the total color-degree of G; therefore, $k > m$. Thus c admits a rainbow matching of size $k \geq m+1$. ■

A key element to the proof of Theorem 2.1 is the bound $\hat{d}(v) + \hat{d}(u) \leq n$, where uv is an edge in a maximum-size rainbow matching. We can obtain a similar bound in C_4-free graphs that will be explained in the following section.

3.2 C_4-free Graphs

In this section, we apply a similar technique to graphs that have no 4-cycles.

Theorem 3.2. *Let G be a C_4-free graph on n vertices. Let c be any edge-coloring of G where the total color-degree satisfies $\hat{d}(G) \geq 2mn$ for some integer m such that $0 \leq m < n$. Then c admits a rainbow matching of size m.*

Proof. Let G be a C_4-free graph on n vertices. Let c be an edge-coloring of G with $\hat{d}(G) > 2mn$. For the sake of contradiction, let M be a maximum rainbow matching of size $k < m$ with edges u_iv_i for $1 \leq i \leq k$. Among all such matchings, assume M is chosen so that $\sum_{v \in V(M)} \hat{d}(v)$ is minimized. Without loss of generality, suppose $c(u_iv_i) = i$ for $1 \leq i \leq k$.

Claim 1. For each i $(1 \leq i \leq k)$, we have $\hat{d}(u_i) + \hat{d}(v_i) \leq n+1$.

Proof of Claim 1. Fix any i $(1 \leq i \leq k)$. As indicated in Figure 15, we let $A = N(u_i) - \{v_i\} - N(v_i)$, we let $B = N(u_i) \cap N(v_i)$, and we let $C = N(v_i) - \{u_i\} - N(u_i)$.

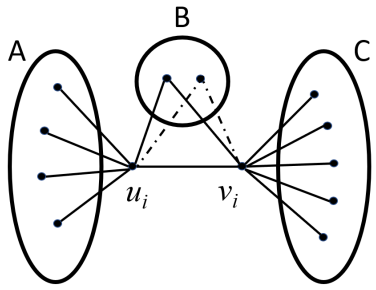

Figure 15: As G is C_4-free, we see that u_i and v_i share at most 1 vertex.

Since G has n vertices, $|A| + |B| + |C| + 2 \leq n$. Since G is C_4-free, $|B| \leq 1$. Now $d(u_i) = |A| + |B| + 1$ and $d(v_i) = |B| + |C| + 1$, and therefore

$$\begin{aligned} d(u_i) + d(v_i) &= |A| + 2|B| + |C| + 2 \\ &= (|A| + |B| + |C| + 2) + |B| \\ &\leq n + |B| \\ &\leq n + 1. \end{aligned}$$

Since $\hat{d}(u_i) \leq d(u_i)$ and $\hat{d}(v_i) \leq d(v_i)$, the claim follows. □

Let H denote the subgraph induced on the vertices not in M. In other words, let $H = G[S]$ where $S = V(G) \setminus V(M)$. Suppose $e \in E(H)$. Since M has maximum size, it must be the case that $c(e) \in \{1, ..., k\}$. Without loss of generality, suppose that $c(H) = \{1, ..., j\}$ for some $0 \leq j \leq k$.

Claim 2. If $xy \in E(H)$, then $\hat{d}(x) + \hat{d}(y) \leq 2j + 2k$.

Proof of Claim 2. As $x, y \in V(H)$, each are incident to at most j colors assigned to edges in H. As indicated in Figure 16, the edge xy can share at most two incident edges with any edge in M without creating a C_4 subgraph.

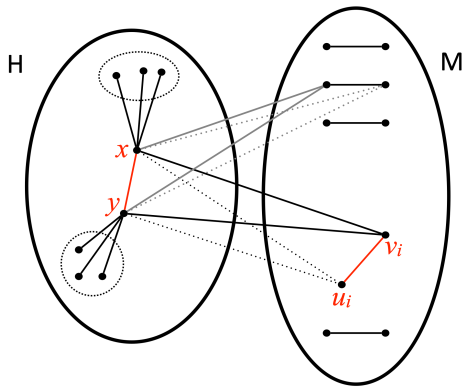

Figure 16: Between edges xy and u_iv_i, there are four possible edges. Notice that G can have at most 2 of these without creating a C_4 subgraph. Thus the edge xy is incident with at most $2k$ edges that have an endpoint in M.

So we have at most $2k$ edges that are incident on both $V(M)$ and $\{x, y\}$. So x, y are incident to at most $2k$ colors on these edges. Thus, $\hat{d}(x) + \hat{d}(y) \leq 2j + 2k$. □

Claim 3. For any $i \in \{1, \ldots, j\}$, we have $\hat{d}(u_i) + \hat{d}(v_i) \leq 2(j + k)$.

Proof of Claim 3. Since $i \in \{1, \ldots, j\}$, there exists an edge $xy \in H$ with $c(xy) = i$. By Claim 2, we know that $\hat{d}(x) + \hat{d}(y) \leq 2(j + k)$. Since $V(H) \cap V(M) = \varnothing$, swapping xy with u_iv_i would not change the color-set of M. But M was chosen so that $\sum_{v \in V(M)} \hat{d}(v)$ is minimized, so it follows that $\hat{d}(u_i) + \hat{d}(v_i) \leq \hat{d}(x) + \hat{d}(y)$. □

Claim 4. $\sum_{v \in H} \hat{d}_G(v) \leq (n - 2k)(j + k) + k$.

Proof of Claim 4. The $(n - 2k)j$ term comes from the fact that H has $n - 2k$ vertices, each of which can see at most every color in $\{1, ..., j\}$ on its edges in H. This accounts for all but the colors on edges from H to M. We will show that there are at most $(n - 2k)k + k$ such edges by contradiction. Suppose that there are $(n - 2k)k + k + 1$ edges from H to M. By the pigeonhole principle, there exists an edge $u_iv_i \in M$ that receives at least $n - 2k + 2$ edges from H. Each vertex in H can send at most two edges to u_iv_i. Therefore, there must exist two vertices in H that each send two edges to u_iv_i, witnessing a C_4 subgraph; this is a contradiction. □

Now consider

$$\begin{aligned}
2mn &\leq \sum_{i=1}^{k} (\hat{d}(u_i) + \hat{d}(v_i)) + \sum_{v \in H} \hat{d}_G(v) \\
&\leq \sum_{i=1}^{j} (\hat{d}(u_i) + \hat{d}(v_i)) + \sum_{i=j+1}^{k} (\hat{d}(u_i) + \hat{d}(v_i)) + \sum_{v \in H} \hat{d}_G(v) \\
&\leq j(2k + 2j) + (k - j)(n + 1) + (n - 2k)(j + k) + k \\
&= 2kj + 2j^2 + nk + k - nj - j + nj + nk - 2kj - 2k^2 + k \\
&= 2j^2 + 2nk - j + 2k - 2k^2 \\
&\leq 2j^2 - 2k^2 + 2k - j - 2n + 2mn \\
&< 2mn.
\end{aligned}$$

This is a contradiction on the total color-degree of G; therefore, $k \geq m$. Thus c admits a rainbow matching of size $k \geq m$. ■

4 Properly Edge-Colored Graphs

In this section, we consider properly edge-colored graphs. We will analyze a greedy algorithm that constructs a matching as it appears in [5] with adjustments inspired by [3]. This type of graph is a main focus for [1], [9], and [11].

Theorem 4.1. *Let G be a graph on n vertices. Let c be any proper edge-coloring of G where the total color-degree satisfies $\hat{d}(G) \geq 2mn$ for some integer m such that $0 \leq m < n/8$. Then c admits a rainbow matching of size m.*

4.1 Algorithm

The algorithm given below will be used in the proof of Theorem 4.1 to show there is no counterexample to the theorem. Kritschgau originally modified an algorithm from [1] and [3], adjusting bounds.

Note: In our first attempted implementation, the algorithm could run beyond m steps, creating the potential to have negative bounds for the vertex degree and color classes. In practice, this meant that some steps did not remove any edges, which created an issue in the situation leading to the contradiction used for the proof in [5]. To avoid this issue, we have reintroduced a condition from [3] so that the algorithm terminates under two conditions: when the graph is empty or $i = m - 1$. In addition, instead of removing vertices, we have adjusted the algorithm to simply remove the set of incident edges for a given vertex.

Algorithm. Consider the following algorithm, initializing $G_0 := G$;

1. `If` G_{i-1} `is empty or` $i = m - 1$, `pass to 5,`
2. `if there exists` $v \in V(G_{i-1})$ `with` $\hat{d}(v) \geq 3(m-i)+1$, `then` $G_i = G_{i-1} - \overline{N}(v)$ `and return to 1,`
3. `else, if there exists color-class` R `with` $|R| \geq 2(m-i)+1$ `in` G_{i-1}, `then` $G_i = G_{i-1} - R$ `and return to 1,`
4. `else, if there exists` $uv \in E(G_{i-1})$, `then` $G_i = G_{i-1} - \overline{N}(u) - \overline{N}(v) - c(uv)$ `and return to 1,`
5. `return` $i - 1$.

4.2 Proof of Theorem 4.1

Assume that G is an edge-minimal counterexample to Theorem 4.1. Consider the algorithm above. To prove the theorem, we must first prove the lemmas below.

Lemma 1. Suppose the algorithm returns $k \leq m$. Then G_i contains a matching of size $k - i$ for $0 \leq i \leq k$.

Proof. We will prove the claim by reverse induction on i. If $i = k$, then G_i is empty, and the claim is true. Assume that the claim is true for i. We will prove the claim for $i - 1$. By the induction hypothesis, there exists a matching $M \subseteq G_i$ of size $k - i$. There are three cases:

Case 1: Assume $G_i = G_{i-1} - \overline{N}(v)$ where $\hat{d}(v) \geq 3(m - i) + 1$. By construction, $v \notin V(M)$. Since $\hat{d}(v) \geq 3(m - i) + 1$, there exists $u \in N(v)$, such that $u \notin V(M)$ and $c(uv) \notin c(M)$. Then $M' = M \cup \{uv\}$ is a rainbow matching of size $k - i + 1$. A visual representation of this can be seen in Figure 17.

Case 2: Assume $G_i = G_{i-1} - R$ for some color R with $|R| \geq 2(m - i) + 1$. This implies that $c(e) \neq R$ for all $e \in E(M)$. Since c is a proper coloring and $|R| \geq 2(m-i)+1$, there exists $e \in G_{i-1}$ such that $c(e) = R$ and $M' = M \cup \{e\}$ is a rainbow matching. A visual representation of this can be seen in Figure 18.

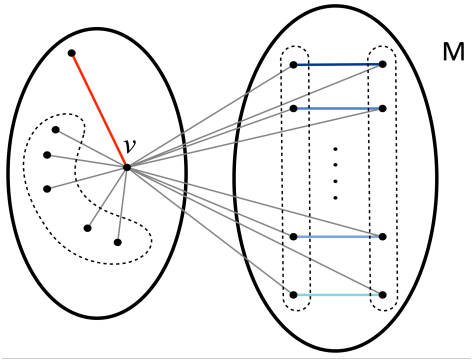

Figure 17: Suppose we form G_i by removing a high degree vertex v from G_{i-1}. Notice v can be adjacent to $2(k-i)$ vertices in M. Furthermore, as $m-i+1 \geq k-i$, we are sure to have a uniquely colored edge with v as an endpoint. This edge is in G_{i-1} and can safely be added to M to create a $k-i+1$ rainbow matching.

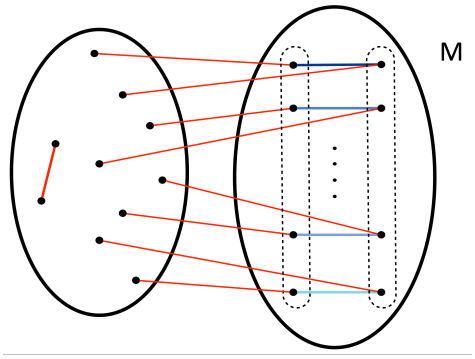

Figure 18: If we have a color-class with at least $2(m-i)+1$ edges, then, if M is a $k-i$ matching, there is at least one edge not incident with M. Thus, if we remove the color-class when forming G_i, there is at least one edge in G_{i-1} that we could add to the matching of G_i to create a $k-i+1$ rainbow matching.

Case 3: Assume that $G_i = G_{i-1} - \overline{N}(v) - \overline{N}(u) - c(uv)$ for some $uv \in E(G_{i-1})$. By construction $\{u, v\}$ is disjoint from $V(M)$ and $c(e) \neq c(uv)$ for all $e \in M$. Therefore, $M' = M \cup \{uv\}$ is a rainbow matching.

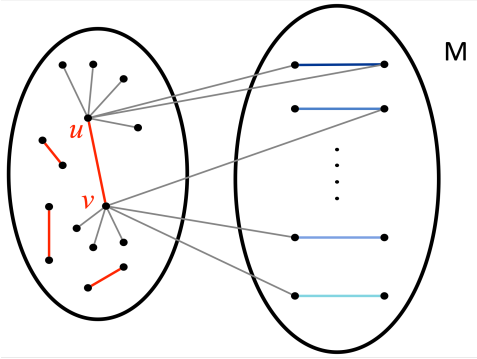

Figure 19: Suppose we remove an edge, the incident edges of its endpoints, and its color-class from G_{i-1} to form G_i. Notice that our edge removed has a color distinct from the edges of M. So there is at least one edge in G_{i-1} that we could add to the matching of G_i to create a $k-i+1$ rainbow matching.

This concludes the proof of the lemma. □

Since G is an edge-minimal counterexample, the algorithm applied to G will return $k < m$. We will now derive a contradiction.

Lemma 2. Let $W(G_i)$ denote the difference of total color-degree between G_i and G_{i-1} under c. Then for all $1 \leq i \leq k$, we have $W(G_i) \leq 2n$.

Proof. Recall we let $W(G_i)$ denote the difference of total color-degree between G_i and G_{i-1} under c. We consider $W(G_i)$ under the conditions 2,3, and 4 of the algorithm.

Case 1: Assume $G_i = G_{i-1} - v$ where $\hat{d}(v) \geq 3(m-i)+1$. Notice that v is incident to at most $n-1$ edges. Therefore, deleting v will remove at most $2(n-1)$ color-degrees, so $W(G_i) \leq 2(n-1) = 2n-2 < 2n$.

Case 2: Assume $G_i = G_{i-1} - R$ for some color R with $|R| \geq 2(m-i)+1$. Because c is proper, $|R| \leq \lfloor n/2 \rfloor$. Deleting all edges of color R reduces the color-degree by at most n, so $W(G_i) \leq n < 2n$.

Case 3: Assume that $G_i = G_{i-1} - v - u - c(uv)$ for some $uv \in E(G_{i-1})$. Since G_i is not constructed by step 2 , we know that $\hat{d}(u), \hat{d}(v) \leq 3(m-i)$. Furthermore, since G_i is not constructed by step 3, we know that $|c(uv)| \leq 2(m-i)$. This implies that

$$\begin{aligned} W(G_i) &= 2(\hat{d}(v) + \hat{d}(u)) + 2|c(uv)| \\ &\leq 16(m-i) \\ &\leq 2n \end{aligned}$$

Since $W(G_i) \leq 2n$ under all cases, we can be sure $W(G_i) \leq 2n$ for all $1 \leq i \leq k$. □

Using the above lemma, we can now say,

$$\begin{aligned} 2nm &\leq \hat{d}(G) \\ &= \sum_{i=1}^{k} W(G_i) \\ &\leq k(2n) = 2nk \end{aligned}$$

As $k < m$, $2nk < 2mn$, which means we have $2mn < 2mn$. This is a contradiction; therefore, the theorem has been proven. ■

5 Enacting the Algorithm in Python

To get a better idea of how the algorithm works and its connection to how one may build a rainbow matching from backtracking the algorithm, we can code the algorithm in Python. We will be exploring the standard algorithm as described by the algorithm. We also created variations on this algorithm, which are acknowledged later. We present an abridged version of the code for one function below. Full code for all variations can be found on GitHub.

5.1 Forwards Algorithm

To run the algorithm, we needed multiple tools to implement it under the different conditions. We created a function that runs on any given graph to give a proper coloring. This function cycles through the edge set randomly and assigns a color based on an integer, taking the smallest integer possible. We also have a function that picks a random set from a set of sets. This allows us to take a random color-class under condition 3 of the algorithm when multiple color-classes achieve the desired cardinality. We also register a new color sequence for use later and a function to create a dictionary of edges and their corresponding color.

```
!pip install igraph
import igraph as ig
import array as arr
import matplotlib.pyplot as plt
import random
from igraph import Graph
import matplotlib as mpl

def get_random_list(list_of_lists):
  if not list_of_lists:
    return None
  return random.choice(list_of_lists)

def greedy_edge_coloring(graph):
    num_edges = graph.ecount()
    colors = [0] * num_edges
    available_colors = set()
    used_edges = []
       ⋮
    return colors

mpl.color_sequences.register('color_classes', ['red', 'blue', 'yellow',
                                                'green', 'pink', 'purple',
```

```
24                                                  'cyan', 'darkgreen',
25                                                  'orange', 'indigo',
26                                                  'magenta', 'olive',
27                                                  'black'])
28
29 colors = mpl.color_sequences['color_classes']
```

Listing 1: Global Algorithms

To make it easier to compare runs, we establish a graph elsewhere and then call a copy of the graph and its edge-color dictionary to run each algorithm on. We pull the edge colors for each edge in line 4 and establish a set of edge sets partitioned into their respective color classes in lines 5 through 9. We also establish empty dictionaries `rain_choices` and `cond_sequence`. We will be adding to these dictionaries in each step of the function. These dictionaries will later be used by a reverse algorithm for creating a rainbow matching on the graph `g`.

The function takes a given graph, its edge-color dictionary, and a chosen value of m. This choice decides how long the function will run. Note that while we can choose any value for m, a smaller value decreases the bounds in conditions 2 and 4, making it more likely for the function to terminate for `i-1 = m` instead of terminating by reaching an empty graph. Furthermore, though we can pick infinitely large values of m, it is unnecessary to pick $m \geq n/2$, as we naturally can't find a matching with more than $n/2$ edges. The overarching for loop runs i from 1 to $m+1$ to keep the bounds in conditions 2 and 3 nonnegative.

Line 16 allows us to later call on color classes by a color name instead of just an index. The for loop introduced at line 19 will help us keep track of edge colors and the size of the color classes.

```
1 g = reg.copy()
2 edge_colors = reg_edge_colors.copy()
3
4 g.es["color"] = [edge_colors.get(i, "black") for i in range(len(g.es))]
5 color_groups = {color: [] for color in  mpl.color_sequences['color_classes
     ']}
6 for edge in g.es:
7     color = edge["color"]
8     if color in color_groups:
9         color_groups[color].append(edge.index)
10
11 rain_choices = {}
12 cond_sequence = {}
13
```

```
14 def algo_random(g,edge_colors,m):
15   for i in range(1,m+1):
16     color_groups = {color: [] for color in
17                     mpl.color_sequences['color_classes']}
18
19     for edge in g.es:
20         color = edge["color"]
21             if color in color_groups:
22             color_groups[color].append(edge.index)
23             edge_color_size = [len(a) for a in color_groups.values()]
```

Listing 2: Algorithm Setup

Approaching condition 1 is rather simple. We want to make sure the function terminates if the graph has become empty. In this case, we append the empty graph at step `i` to the `rain_choices` dictionary for use later.

```
1     if len(g.es) == 0:
2         cond_sequence.setdefault(i, []).append(1)
3         g_i = g.subgraph_edges(edges=[])
4         rain_choices[i] = g_i
5         return (g,i)
```

Listing 3: Condition 1

Condition 2 is triggered if there is a vertex of suitably high degree dependent on the value of `i`. Thus, we check the maximum degree of the graph. If there is a vertex that satisfies the degree, we record condition 2 in the `cond_sequence` dictionary at its corresponding `i` index. To account for multiple vertices of suitable degree, we create a list. The for loop added any appropriate vertices to the `lrg_enough` list. We then choose a random vertex from the list and get the incident edges for the vertex. The set of edges is used to create a subgraph of `g` to be saved as `g_i` in the `rain_choices` dictionary. The for loop starting on line 12 ensures that for the edges in `g_i` we use the appropriate edge colors per the coloring of `g`. Then we remove the set of edges from `g` per the instruction of the algorithm. The function utilizes many tools from the `igraph` library. The `delete_edges` command not only removes a given set of edges but also updates `g`, so we don't need to update it manually. Line 17 resets the `lrg_enough` list for future `i` iterations.

```
1     elif (max(g.degree()) >= 3*(m-i)+1):
2         cond_sequence.setdefault(i, []).append(2)
3         lrg_enough = []
4         for v in g.vs:
5             if g.degree(v) >= 3*(m-i)+1:
```

```
6             lrg_enough.append(v.index)
7       vert = random.choice(lrg_enough)
8
9       edges_to_add = g.vs[vert].incident()
10      g_i = g.subgraph_edges(edges_to_add, delete_vertices=False)
11
12    for edge in g_i.es:
13        original_edge = g.es.find(_source=edge.source, _target=edge.
    target)
14        edge["color"] = original_edge["color"]
15    rain_choices[i] = g_i
16    g.delete_edges(g.vs[vert].incident())
17    lrg_enough = []
```

Listing 4: Condition 2

Similarly to condition 2, condition 3 is triggered when the maximum size of any of the color classes exceeds the bound per `i`. The if statement in line 5 has the additional condition that the color-class must be nonempty. The for loop in line 9 collects the edge indices of a given color to create the set `edges_to_delete`. We save the appropriate condition and `g_i` to our dictionaries for use later, then remove the set of edges from `g`.

```
1    elif (max(edge_color_size) >= 2*(m-i) + 1):
2     cond_sequence.setdefault(i, []).append(3)
3     lrg_enough_cc = []
4     for color in color_groups:
5         if len(color_groups[color]) >= 2*(m-i) + 1 and len(color_groups[
    color]) > 0:
6             lrg_enough_cc.append(color_groups[color])
7     color_class = random.choice(lrg_enough_cc)
8
9     for color, edge_indices in color_groups.items():
10      if color_class[0] in edge_indices:
11          color_cc = color
12          break
13
14    edges_to_delete = color_groups[color_cc]
15    g_i = g.subgraph_edges(edges_to_delete, delete_vertices=False)
16    for edge in g_i.es:
17        original_edge = g.es.find(_source=edge.source, _target=edge.
    target)
18        edge["color"] = original_edge["color"]
19    rain_choices[i] = g_i
20
```

```
21      g.delete_edges(edges_to_delete)
22      lrg_enough_cc = []
```

Listing 5: Condition 3

Condition 4 of the algorithm stipulates that we remove any edge, the incident edges of its endpoints, and its color class. So for a nonempty edge set, we choose `rm_edge` a random edge from the edge set `g.es`. The for loop in line 6 serves to collect the edges of the same color-class of `rm_edge`. We then obtain the `target` and `source` of `rm_edge`. We create `g_i` at line 13 before obtaining the incident edges of the `source` and `target`. This can be done at this step because when reversing the algorithm later, we will only be adding `rm_edge` to the growing matching. We add the endpoint incident edges to our set of edges in the color-class of `rm_edge`, then remove the entire set from `g`. If we hit no conditions, we return `g`, `i`, and a print statement to make it clear under which condition we terminated. We call the algorithm using `g`, its `edge_colors`, and a chosen value of `m`. Here, we are running for `m=5`.

```
1     else:
2       cond_sequence.setdefault(i, []).append(4)
3       if len(g.es) > 0:
4           rm_edge = random.choice(g.es)
5           edge_cc_adj=[]
6           for e in g.es:
7             if e["color"] == rm_edge["color"]:
8               edge_cc_adj.append(e.index)
9
10          source_vertex = rm_edge.source
11          target_vertex = rm_edge.target
12
13          g_i = g.subgraph_edges(rm_edge, delete_vertices=False)
14          for edge in g_i.es:
15              original_edge = g.es.find(_source=edge.source, _target=edge.
    target)
16              edge["color"] = original_edge["color"]
17          rain_choices[i] = g_i
18
19          incident_edges_source = g.incident(source_vertex)
20          incident_edges_target = g.incident(target_vertex)
21          incident_edges = list(set(incident_edges_source +
22                                    incident_edges_target))
23          edge_cc_adj.extend(incident_edges)
24          g.delete_edges(edge_cc_adj)
25          edge_cc_adj = []
26
```

```
return (g,"Terminating for i-1=m",i)
print(algo_random(g, edge_colors, 5)[1])
```

Listing 6: Condition 4

5.2 Example Run

For demonstration, we run the algorithm with the bounds presented by [5], taking random choices on conditions 2,3, and 4. We will use a 4-regular graph on 15 vertices. This graph makes for a good example as it has decently sized color classes, so when we run the function with `m = 5`, we will hit conditions 2,3, and 4. Here we create the graph and coloring in a global cell, and then create a copy for every run. This allows us to consistently experiment on the same graph since the `Graph.K_Regular` creates a random graph on each run. The code below shows the construction in the global cell. The for loop creates a dictionary of edge indices and their assigned color by the coloring algorithm for use by the `algo_random` function.

```
reg = Graph.K_Regular(15,4)
reg.vs["label"] = list(range(reg.vcount()))
col_reg_edge_colors = greedy_edge_coloring(reg)
reg.es['color'] = col_reg_edge_colors
reg_edge_colors = {}

for i, color in enumerate(col_reg_edge_colors):
  ⋮
print(reg_edge_colors)
```

We can make our copy of the example graph and call our function.

```

g = reg.copy()
edge_colors = reg_edge_colors.copy()
g.es["color"] = [edge_colors.get(i, "black") for i in range(len(g.es))]
color_groups = {color: [] for color in  mpl.color_sequences['color_classes
    ']}
for edge in g.es:
    color = edge["color"]
    if color in color_groups:
        color_groups[color].append(edge.index)

print(algo_random(g, edge_colors, 5)[1])
```

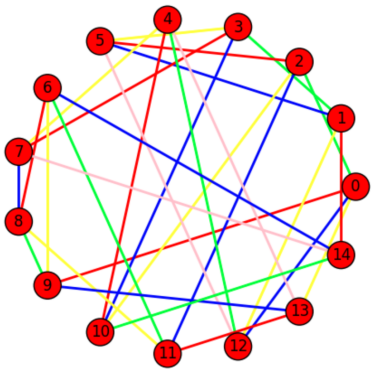

Figure 20: The function starts by printing the graph with its colored edges, using the provided `edge_colors` dictionary. We call this graph `g`.

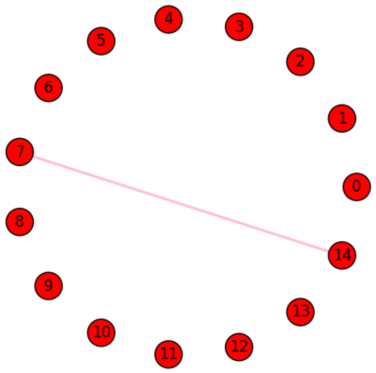

Figure 21: At `i = 1`, we hit condition 4. The condition is added to the condition sequence for the run; `cond_sequence = {1: [4]}`. A random edge is chosen, in this case, the edge with ID 24 and color pink. We remove the edge, the incident edges of its endpoints, and the pink color class. The subgraph above containing edge 24 and the remaining isolated vertices of `g` are saved in the `rain_choices` dictionary as `g_1`.

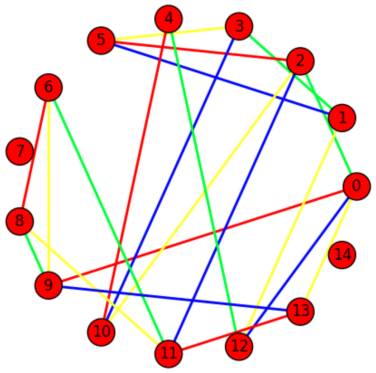

Figure 22: Graph `g` is updated after removing the edge of `g_1`, the incident edges of its endpoints, and the pink color-class.

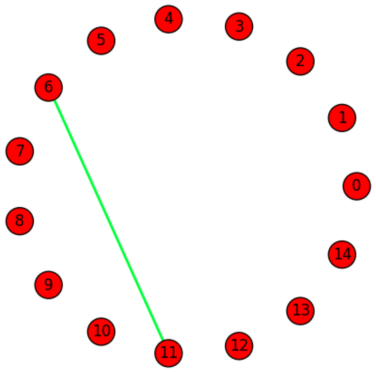

Figure 23: At `i = 2`, we hit condition 4 again. The condition is added to the condition sequence for the run; `cond_sequence = {1: [4], 2: [4]}`. A random edge is chosen, in this case, the edge with ID 16 and color green. We remove the edge, the incident edges of its endpoints, and the green color class. The function saves the above subgraph `g_2` to the `rain_choices` dictionary.

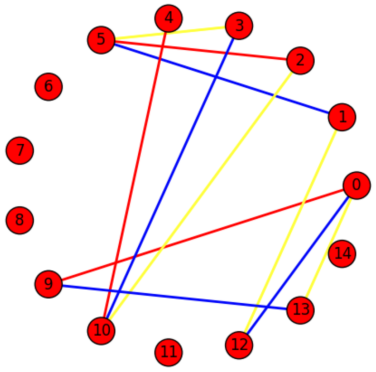

Figure 24: Graph `g` is updated after removing the edge of `g_2`, the incident edges of its endpoints, and the green color-class.

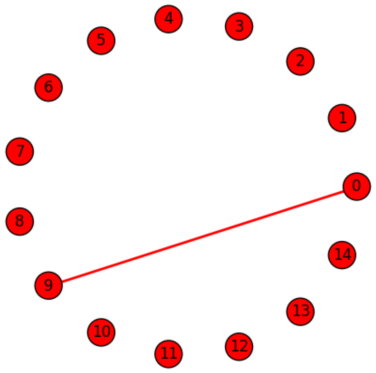

Figure 25: At `i = 3`, we hit condition 4 again. The condition is added to the condition sequence for the run; `cond_sequence = {1: [4], 2: [4], 3: [4]}`. Random edge with ID 0 and color red. The function saves the subgraph `g_3` to the `rain_choices` dictionary.

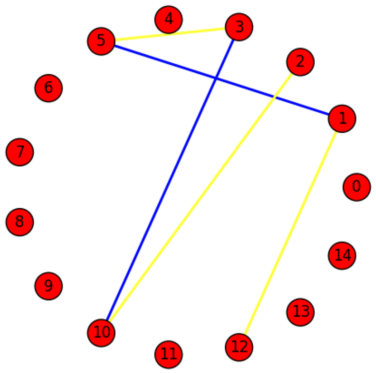

Figure 26: Graph `g` is updated after removing the edge of `g_3`, the incident edges of its endpoints, and the red color-class.

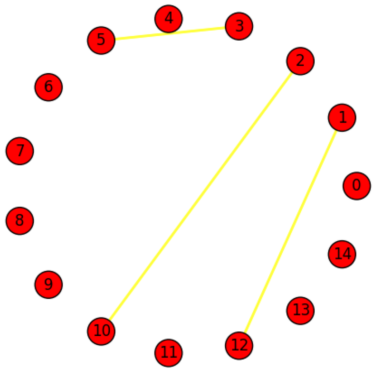

Figure 27: At `i = 4`, we hit condition 3. The condition is added to the condition sequence for the run; `cond_sequence = {1: [4], 2: [4], 3: [4], 4: [3]}`. The function chooses the yellow color class. The subgraph containing all edges of color yellow and the remaining isolated vertices of `g` are saved in the `rain_choices` dictionary as `g_4`.

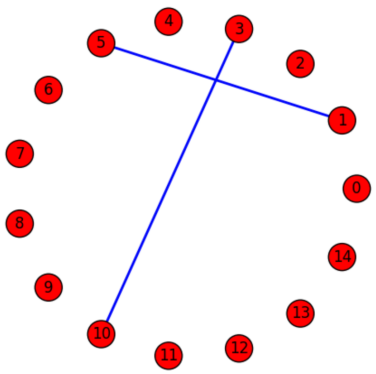

Figure 28: Graph `g` is updated after removing the yellow color-class.

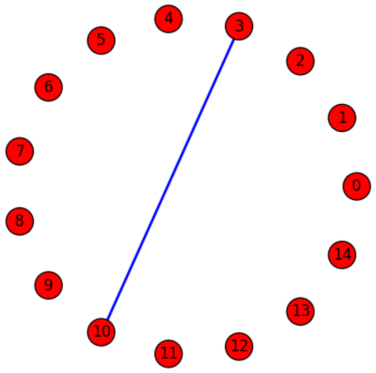

Figure 29: At `i = 5`, we hit condition 2. The condition is added to the condition sequence for the run; `cond_sequence = {1: [4], 2: [4], 3: [4], 4: [3], 5: [2]}`. At this step, the degree required to qualify for condition 2 is `3*(m-i)+1 = 1`, so any vertex with degree 1 or more may be chosen. The function chooses vertex 10, so all incident edges of vertex 10 are removed. The `rain_choices` dictionary saves `g_5` as the graph of vertex 10, its incident edges, and the remaining isolated vertices of `g`.

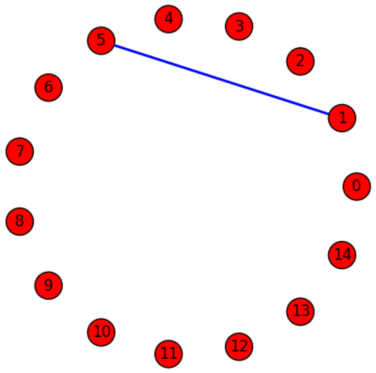

Figure 30: Graph `g` is updated after removing the edges of `g_5`. At `i = 6`, the function terminates for `i-1 = m`.

5.3 Reverse Matching Algorithm

The reverse matching algorithm takes the input graph, the condition dictionary built in the forward algorithm, the subgraphs for each step of the forward algorithm arranged in a dictionary, and the original dictionary of edge colors for our graph.

```
1 def reverse_engineer(g, cond_sequence, rain_choices, edge_colors):
```

Listing 7: Reverse Engineer Algorithm

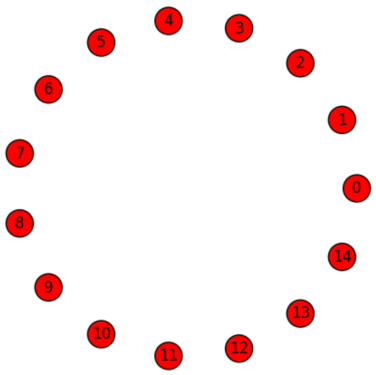

Figure 31: The `reverse_engineer` function starts with an empty graph on the same vertices utilized in the forward running algorithm.

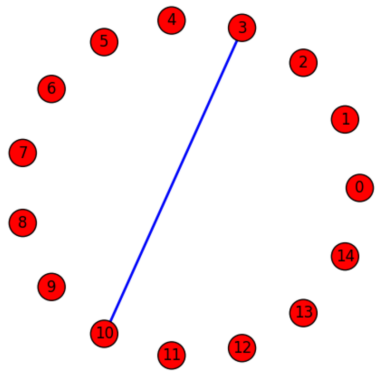

Figure 32: Processing step `i = 5`. The `rain_choices` dictionary records hitting condition `2`, so a specific vertex and its incident edges were removed by the algorithm. The function cycles through, randomly choosing an edge stored in `g_5`. If the edge uses an endpoint or color already utilized, the function will choose a new random edge until it finds a suitable one. Since the previous graph is empty, any edge may be added to the matching safely. The function chooses (3, 10) with the new color blue.

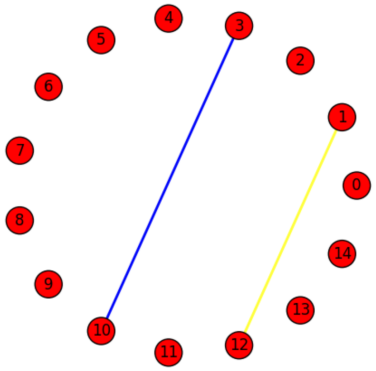

Figure 33: Processing step `i = 4`. The `rain_choices` dictionary records hitting condition `3`, so a specific color-class was removed by the algorithm. We thus add a random suitable edge from that color-class to our matching. The edges stored in `g_4` are (2,10), (3,5), and (1,12). The function notes that (2,10) and (3,5) already share an endpoint with an edge in our matching, so they cannot be added. We add (1, 12) with color yellow.

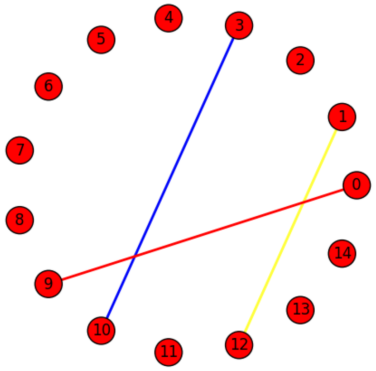

Figure 34: Processing step `i = 3`. The `rain_choices` dictionary records hitting condition `4`, so a specific edge, its neighbors, and its color-class were removed by the algorithm. We thus add the chosen edge to our matching. The edge stored in `g_3` is edge (0, 9) with color red.

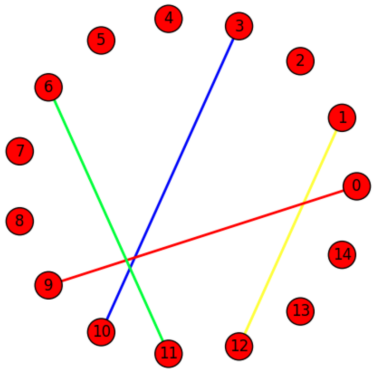

Figure 35: Processing step `i = 2`. The `rain_choices` dictionary records hitting condition `4`, so a specific edge, its neighbors, and its color-class were removed by the algorithm. We thus add the chosen edge to our matching. The edge stored in `g_2` is edge (6, 11) with color green.

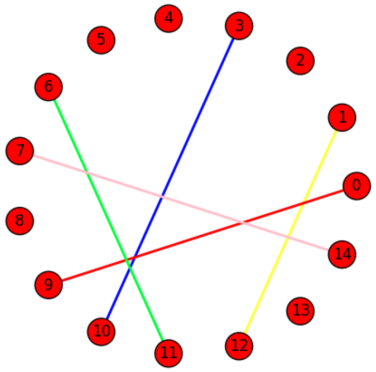

Figure 36: Processing step `i = 1`. The `rain_choices` dictionary records hitting condition `4`, so a specific edge, its neighbors, and its color-class were removed by the algorithm. We thus add the chosen edge to our matching. The edge stored in `g_1` is edge (7, 14) with color pink.

6 Other Research Directions

There are many possible directions for extending this research. First and foremost, there are natural variations to the algorithm that could be explored to compare with the implementation explained here. For example, we list some variations below. Under all algorithm variations, we consider a random edge from the edge set when enacting condition 4 of Theorem 4.1. For the other conditions, we could make different choices. Case 1 below describes the algorithm as we presented it here. The other two could be interesting to further investigate.

1) We consider a random vertex of suitable degree under condition 2 and choose a random suitably large color-class under condition 3.

2) We consider the vertex of maximum degree in the set of sufficient degree under condition 2 and the maximum size color-class from the set of suitably large color-classes under condition 3.

3) We consider the vertex of minimum degree in the set of sufficient degree under condition 2 and the minimum size color-class from the set of suitably large color-classes under condition 3.

Moreover, we could consider variations utilizing the original degree bound for condition 2 presented by [3]. In this case, we would modify condition 2 of the algorithm as presented by Kritschgau to be the following: if there exists $v \in V(G_{i-1})$ with $d(v) > 2(m-i)$, then $G_i = G_{i-1} - \overline{N}(v)$. Under this change, we could also consider the three variations above for a total of six variations on the algorithm.

Additionally, there is an unresolved conjecture given by Kritschgau that any graph G with $\hat{d}(G) \geq 2mn$ contains a rainbow matching of size $m+1$. Though this is still only a conjecture, some improvement to the bounds on graphs not necessarily properly colored has been made recently by Zhou [13]. Given the possibility that algorithmic variations (as described above) could improve the performance, it seems plausible to us that such a conjecture might indeed hold.

7 Bibliography

References

[1] Jennifer Diemunsch, Michael Ferrara, Casey Moffatt, Florian Pfender, and Paul S. Wenger. Rainbow matchings of size $\delta(g)$ in properly edge-colored graphs. Electr. J.

Comb., 12, 2011.

[2] Andras Gyarfas and Gabor N. Sarkozy. Rainbow matchings and partial transversals of latin squares. Discrete Mathematics, 327:96–102, 2012.

[3] Alexandr Kostochka, Florian Pfender, and Matthew Yancey. Large rainbow matchings in large graphs. Combinatorics, Probability and Computing, 21, 04 2012.

[4] Alexandr Kostochka and Matthew Yancey. Large rainbow matchings in edge-coloured graphs. Combinatorics, Probability and Computing, 21:255–263, 03 2012.

[5] Jürgen Kritschgau. Rainbow matchings of size m in graphs with total color degree at least $2mn$. Electr. J. Comb., 27, 2019.

[6] Timothy LeSaulnier, Christopher Stocker, Paul Wenger, and Douglas West. Rainbow matching in edge-colored graphs. Electr. J. Comb., 17, 12 2010.

[7] Allan Lo and Ta Sheng Tan. A note on large rainbow matchings in edge-coloured graphs. Graphs and Combinatorics, 2012.

[8] S. K. Stein. Transversals of latin squares and their generalizations. Pacific Journal of Mathematics, 59(2):27–575, 1975.

[9] G. Wang. Rainbow matchings in properly edge colored graphs. Electron. J. Comb., 18, 2011.

[10] Guanghui Wang and Hao li. Heterochromatic matchings in edge-colored graphs. Electr. J. Comb., 15, 11 2008.

[11] Guanghui Wang, Jianghua Zhang, and Guizhen Liu. Existence of rainbow matchings in properly edge-colored graphs. Frontiers of Mathematics in China, 7, 06 2012.

[12] Douglas Brent West et al. Introduction to graph theory, volume 2. Prentice hall Upper Saddle River, 2001.

[13] Wenling Zhou. Large rainbow matchings in edge-colored graphs with given average color degree. Graphs and Combinatorics, 38, 2021.