2017

Coloring problems in graph theory

Kevin Moss

Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/etd

Part of the Computer Sciences Commons, and the Mathematics Commons

Recommended Citation

Moss, Kevin, "Coloring problems in graph theory" (2017). Graduate Theses and Dissertations. 15383.
https://lib.dr.iastate.edu/etd/15383

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
Coloring problems in graph theory

by

Kevin Moss

A dissertation submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Mathematics

Program of Study Committee:
Bernard Lidický, Co-major Professor
Steve Butler, Co-major Professor
Clifford Bergman
Ryan Martin
Sung-Yell Song

The student author and the program of study committee are solely responsible for the content of this dissertation. The Graduate College will ensure this dissertation is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University
Ames, Iowa
2017

Copyright © Kevin Moss, 2017. All rights reserved.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>v</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>vi</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>viii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>ix</td>
</tr>
<tr>
<td>CHAPTER 1. GENERAL INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Basic Definitions</td>
<td>1</td>
</tr>
<tr>
<td>1.3 Planar Graphs</td>
<td>4</td>
</tr>
<tr>
<td>1.3.1 Graph Coloring</td>
<td>5</td>
</tr>
<tr>
<td>1.3.2 List Coloring and Choosability</td>
<td>5</td>
</tr>
<tr>
<td>1.3.3 Intersection and Union Separation</td>
<td>6</td>
</tr>
<tr>
<td>1.3.4 Packing Coloring</td>
<td>8</td>
</tr>
<tr>
<td>1.4 The Discharging Method</td>
<td>9</td>
</tr>
<tr>
<td>CHAPTER 2. CHOOSABILITY WITH UNION SEPARATION</td>
<td>12</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>12</td>
</tr>
<tr>
<td>2.1.1 Notation</td>
<td>14</td>
</tr>
<tr>
<td>2.2 Non-(k, t)-Choosable Graphs</td>
<td>15</td>
</tr>
<tr>
<td>2.3 Reducible Configurations</td>
<td>16</td>
</tr>
<tr>
<td>2.4 Sparse Graphs</td>
<td>17</td>
</tr>
<tr>
<td>2.5 (4, t)-choosability</td>
<td>18</td>
</tr>
<tr>
<td>2.6 (3, 11)-choosability</td>
<td>20</td>
</tr>
</tbody>
</table>
CHAPTER 3. TOWARDS (3,10)-CHOOSABILITY .............................. 23
  3.1 Introduction ............................................................. 23
  3.2 (3,10)-Choosability .................................................... 23
    3.2.1 Reducible Configurations ....................................... 24
    3.2.2 Proof of Theorem 3.2.2 ....................................... 28

CHAPTER 4. PACKING COLORING ON INFINITE LATTICES .............. 32
  4.1 Introduction ............................................................. 32
    4.1.1 Density on an Infinite Graph .................................. 33
  4.2 Hexagonal Lattice ...................................................... 36
  4.3 Truncated Square Lattice ............................................. 39
  4.4 Two-layer Hexagonal Lattice .......................................... 42
  4.5 Offset Two-Layer Hexagonal Lattice ................................ 43
  4.6 Generating Colorings .................................................. 44
    4.6.1 Backtracking ..................................................... 46
    4.6.2 Random Coloring ................................................ 47
    4.6.3 Priority-Based Random Coloring ................................ 47
    4.6.4 Checking Colorings .............................................. 49
    4.6.5 Choosing Dimensions ............................................ 50
  4.7 SAT Solvers .............................................................. 50
  4.8 Results ................................................................. 51

CHAPTER 5. CONCLUSION ....................................................... 54

APPENDIX A. (3,11)-CHOOSABILITY ........................................ 56

APPENDIX B. SOURCE CODES ................................................. 58
  B.1 Objects ................................................................. 58
    B.1.1 Graph ............................................................. 58
    B.1.2 Vertex ............................................................. 71
    B.1.3 Symmetry ........................................................ 72
    B.1.4 Graph Colorer .................................................... 75
B.1.5 Local Random Graph Colorer ........................................ 83
B.1.6 Naive Random Graph Colorer ...................................... 88
B.1.7 File Reader ............................................................... 90
B.2 Additional Methods .................................................... 94
  B.2.1 Distances ............................................................... 94
  B.2.2 Graph Experiments .................................................. 99

BIBLIOGRAPHY ................................................................. 104
LIST OF TABLES

Table 3.1 Cases with negative charge after applying (R2) .............................. 29
Table 3.2 Cases with negative charge after applying (R3) .............................. 30
Table 3.3 Remaining cases with negative charge after applying (R3) ............... 31
Table 4.1 Density of color $i$ for $1 \leq i \leq 80$ in five attempts at packing $P_2 \square \mathcal{H}$. Attempt 5 resulted in a 205-packing ............................................. 52
Table 4.2 Density of colors 1 through $i$ for $1 \leq i \leq 80$ in five attempts at packing $P_2 \square \mathcal{H}$. Attempt 5 resulted in a 205-packing ............................................. 53
LIST OF FIGURES

Figure 1.1 A graph $G$. ................................................................. 1
Figure 1.3 Graphs $P_5$, a path of length 5; and $C_6$, a cycle of length 6. ...... 3
Figure 1.5 The Cartesian product of two graphs. ..................................... 3
Figure 1.7 The complete graph $K_5$, a tree on 7 vertices, and the complete bipartite graph $K_{3,4}$. ................................................................. 4
Figure 1.8 A plane graph. ................................................................. 4
Figure 1.10 Parts of the triangular, square, and hexagonal lattices. The tilings are infinite and cover the plane. ......................................................... 8
Figure 1.12 Parts of the lattices $P_2 \Box S$ and $P_2 \Box H$. ......................... 9
Figure 2.1 A graph that is not $(k, t)$-choosable. ................................. 15
Figure 2.2 A planar gadget with a $(3, 5)$-list assignment. ....................... 16
Figure 3.1 Configuration C1. ............................................................. 24
Figure 3.2 Configuration C2. ............................................................. 26
Figure 4.1 $\mathcal{H}$ represented on $\mathbb{Z}^2$. ........................................ 37
Figure 4.2 The truncated square lattice. ............................................... 40
Figure 4.3 The truncated square lattice represented on $\mathbb{Z}^2$. .................. 40
Figure 4.4 The tiling pattern for the truncated square lattice. ..................... 41
Figure 4.5 A colored tile in the truncated square lattice. Each tile is colored identically. ................................................................. 41
Figure 4.6 A subgraph $G$ of $S_{tr}$..................................................... 41
Figure 4.7 A second variant of the two-layer hexagonal lattice. Large dots represent two vertices in different layers, joined by an edge.
ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks for those who helped me with various aspects of conducting research and writing this thesis. I am particularly grateful for the guidance, patience, and support provided by Dr. Derrick Stolee in conducting research. His advice has inspired me to pursue a career in Computer Science, giving my future a direction after graduation. I would also like to give thanks to Dr. Bernard Lidický for his outstanding support in both research and writing my thesis, especially under the pressure of my timeframe for graduation. My thanks continue towards Dr. Steve Butler for his support and advice in keeping to a schedule and completing my thesis in time.

I’d like to thank Dr. Sung-Yell Song for his support and advising early in my graduate career, and Dr. Elgin Johnston for his support during my time in high school and as an undergraduate. Many thanks to the remainder of my committee, Dr. Ryan Martin and Dr. Cliff Bergman, as well as to the Discrete Math group, especially including Dr. Leslie Hogben and Dr. Michael Young. Their efforts and contributions have made this work possible.
We consider two branches of coloring problems for graphs: list coloring and packing coloring. We introduce a new variation to list coloring which we call *choosability with union separation*: For a graph $G$, a list assignment $L$ to the vertices of $G$ is a $(k, k + t)$-list assignment if every vertex is assigned a list of size at least $k$ and the union of the lists of each pair of adjacent vertices is at least $k + t$. We explore this new variation and offer comparative results to choosability with intersection separation, a variation that has been studied previously. Regarding packing colorings, we consider infinite lattice graphs and provide bounds to their packing chromatic numbers. We also provide algorithms for coloring these graphs. The lattices we color include two-layer hexagonal lattices as well as the *truncated square lattice*, a 3-regular lattice whose faces have length 4 and 8.
CHAPTER 1. GENERAL INTRODUCTION

1.1 Introduction

Graph theory is the study of graphs, which are discrete structures used to model relationships between pairs of objects. Graphs are key objects studied in discrete mathematics. They are of particular importance in modeling networks, wherein they have applications in computer science, biology, sociology, and many other areas. We focus on coloring problems, which are problems concerning the partitions of a graph. We focus on two main types of colorings: list colorings and packing colorings.

Chapter 1 provides definitions as well as an overview of the current progress in each area. Chapter 2 is a submitted paper that introduces a new variation for list coloring. Chapter 3 provides additional work regarding this variation. In Chapter 4, we consider packing colorings on infinitely large graphs.

1.2 Basic Definitions

A graph $G = (V, E)$ is a set $V = V(G)$ of vertices together with a set $E = E(G)$ of edges, which are two-element subsets of $V$ (i.e. $E \subseteq \binom{V}{2}$). This definition is often referred to as that of a simple graph to distinguish it from other definitions. The typical way to picture a graph is to draw a dot for each vertex and have a line joining two vertices if they share an edge. Figure 1.1 is an example of one such picture.

Figure 1.1: A graph $G$. 
The order of graph $G$, denoted $|G|$, is the number of vertices in $G$; and the size of $G$, denoted $||G||$, is the number of edges in $G$. In Figure 1.1, $|G| = 8$ and $||G|| = 6$. Unless otherwise specified, we assume that a graph has a finite number of vertices and edges.

Two vertices $u$ and $v$ are adjacent if $\{u, v\} \in E$, and an edge $e$ and vertex $v$ are incident if $v \in e$. We typically denote an edge $\{u, v\}$ with the shorter notation $uv$ or equivalently $vu$. For a vertex $v$, the neighborhood of $v$, denoted $N(v)$, is the set of vertices adjacent to $v$; and the degree of $v$, denoted $\text{deg}(v)$, is the order of $N(v)$. If $\text{deg}(v) = k$, we say $v$ is a $k$-vertex; if $\text{deg}(v) \leq k$, we say $v$ is a $k^-$-vertex, and if $\text{deg}(v) \geq k$, we say $v$ is a $k^+$-vertex. We let $\Delta(G)$ and $\delta(G)$ respectively denote the maximum and minimum degrees among all vertices in $G$. In Figure 1.1, $\Delta(G) = 4$ and $\delta(G) = 0$.

Two graphs $G$ and $H$ are isomorphic if there is a bijection $f : V(G) \rightarrow V(H)$ such that $uv \in E(G)$ if and only if $f(u)f(v) \in E(H)$. The function $f$ is called a graph isomorphism. If $f$ is a graph isomorphism from $G$ to itself, than $f$ is called a graph automorphism. We also call isomorphic graphs equivalent. By abuse of notation, we typically treat equivalent graphs as if they are the same graph.

A graph $G$ is connected if for every distinct pair of vertices $u, v \in V(G)$, there exists a set $w_1, w_2, \ldots, w_k \subseteq V(G)$ such that $\{uw_1, w_1w_2, \ldots, w_{k-1}w_k, w_kv\} \subseteq E(V)$. A graph $H$ is a subgraph of $G$ if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G) \cap \binom{V(H)}{2}$ (i.e. $G$ contains $H$). Graph $H$ is an induced subgraph if $E(H) = E(G) \cap \binom{V(H)}{2}$. Graph $H$ is a proper subgraph if $V(H) \neq V(G)$ or $E(H) \neq E(G)$. $H$ is a component of $G$ if $H$ is connected and not a proper subgraph of any other connected subgraph of $G$ (i.e. $H$ is a maximally connected subgraph).

A walk of length $n$ in $G$ is a sequence of vertices $(v_1, v_2, \ldots, v_{n+1})$ such that $v_iv_{i+1} \in E(G)$ for all $1 \leq i \leq n$. A walk is closed if $v_1 = v_{n+1}$. A path $P_n$ is a graph of order $n$ such that there is some labeling of the vertices $V(P_n) = \{v_1, v_2, \ldots, v_n\}$ wherein $E(P_n) = \{v_1v_2, v_2v_3, \ldots, v_{n-1}v_n\}$. A cycle $C_n$ is a path with the additional edge $v_1v_n$. Figure 1.3 is an example of a path and a cycle. Notice that a path corresponds to a walk wherein no vertex is repeated, and a cycle corresponds to a closed walk wherein no vertex is repeated except the first and last vertex. A chorded cycle is a cycle with an additional edge; the edge is called a chord. Cycle $C_3$ is often referred to as a triangle.
Figure 1.3: Graphs $P_5$, a path of length 5; and $C_6$, a cycle of length 6.

For two vertices $u, v \in V(G)$, the distance between $u$ and $v$, denoted $\text{dist}(u, v)$, is the length of the shortest walk from $u$ to $v$ (or $v$ to $u$). If no such walk exists, then $u$ and $v$ are in different components and we say $\text{dist}(u, v) = \infty$. Note that $\text{dist}(u, v) = 1$ if and only if $u$ and $v$ are adjacent, and $\text{dist}(u, v) = 0$ if and only if $u = v$.

For graphs $G$ and $H$, the Cartesian product $G \square H$ is a graph whose vertex set is the Cartesian product $V(G) \times V(H)$ and for any pair of vertices $(u, v)$ and $(u', v')$ in $G \square H$, $(u, v)$ and $(u', v')$ are adjacent if and only if $u = u'$ and $vv' \in E(H)$ or $v = v'$ and $uu' \in E(G)$. Figure 1.5 is an example of a Cartesian product of graphs.

Figure 1.5: The Cartesian product of two graphs.

There are many important families of graphs: A complete graph of order $n$, denoted $K_n$, is a graph that contains every potential edge. That is, $V(K_n)$ is a set of order $n$, and $E(K_n) = \binom{V(K_n)}{2}$. A tree is a connected graph that contains no cycles, and a forest is a graph wherein every component is a tree. An $r$-regular graph is a graph wherein every vertex has degree $r$ (i.e. $\Delta(G) = \delta(G) = r$). A bipartite graph is a graph for which there exists a bipartition of the vertices such that the subgraph induced by each part contains no edges. The complete bipartite graph, denoted $K_{m,n}$, is the bipartite graph with bipartition $U,V$ such that $|U| = m$, $|V| = n$, and for each $u \in U$ and $v \in V$, $uv \in E(K_{m,n})$. Figure 1.7 is an example of a complete graph, a tree, and a complete bipartite graph.
1.3 Planar Graphs

A family of graphs of particular importance to this dissertation is the family of planar graphs.

**Definition 1.3.1.** A planar graph is a graph that can be embedded in the plane. That is, a planar graph can be drawn in $\mathbb{R}^2$ such that every vertex corresponds to a distinct point, every edge corresponds to an arc between its corresponding points, and the interior of each edge contains no vertex and no point of any other edge. A plane graph is a planar graph together with a planar embedding.

A plane graph $G$ has additional attributes called faces. A face in $G$ is a maximal region in $\mathbb{R}^2 \setminus G$; this includes the outer unbounded region. The graph in Figure 1.8 has four faces. Two faces are adjacent if their boundaries share an edge. A face is incident to an edge or vertex if the edge or vertex lies on the boundary of the face. The length of a face $f$, denoted $\text{len}(f)$, is the total length of all disjoint closed walks traversing the boundary of $f$. The number of such edge walks is equal to the number of components with vertices incident to $f$; there is typically only one. If $\text{len}(f) = k$, we say $f$ is a $k$-face; if $\text{len}(f) \leq k$, we say $f$ is a $k^-$-face, and if $\text{len}(f) \geq k$, we say $f$ is a $k^+$-face. The set of all faces in $G$ is denoted $F(G)$.

There are many results involving planar graphs. One particularly well-known result is Euler’s Formula.
Theorem 1.3.2. Euler’s Formula: If $G$ is a connected plane graph, then

$$|V(G)| - |E(G)| + |F(G)| = 2.$$ 

Many proofs to Euler’s Formula are known (see Eppstein [11]).

1.3.1 Graph Coloring

Graph coloring is the labeling of certain elements of a graph; the labels are typically called colors. The traditional elements to color are vertices, and unless otherwise specified, we follow this tradition. For a graph $G = (V, E)$, a $k$-color assignment is a function $c : V \rightarrow \{1, 2, \ldots, k\}$. We say $c$ is proper if for all $uv \in E$, $c(u) \neq c(v)$. We say $G$ is $k$-colorable if there exists a proper $k$-color assignment of $G$. The chromatic number $\chi(G)$ is the smallest $k$ such that $G$ is $k$-colorable. Given a coloring $c$, a color class for color $i$ is a set $X_i = \{v \in V : c(v) = i\}$.

A central result in graph coloring is the Four-Color Theorem.

Theorem 1.3.3. Four-Color Theorem: All planar graphs are 4-colorable.

The Four-Color Theorem was introduced as a conjecture in 1852 [folklore], but a valid proof wasn’t announced until 1976 [1], and it heavily relied on computer aid to exhaust all cases. The proof technique is known as the discharging method, and it continues to be useful in graph theory problems, including those covered in this dissertation.

Though it is easy to find planar graphs that are not 3-colorable, conjectures have been made and results found regarding 3-colorability of planar graphs without certain subgraphs. In particular, Grötzsch showed that planar graphs without 3-cycles are 3-colorable [19]. On a similar avenue, Steinberg conjectured in 1976 that every planar graph without 4- and 5-cycles is 3-colorable (see [20]). This was proven false by Cohen-Addad, Hebdige, Král, Li, and Salgado [8]. However, Borodin, Glebov, and Raspaud [3] showed that planar graphs without 3-cycles sharing edges with cycles of length 4 through 7 are 3-colorable.

1.3.2 List Coloring and Choosability

List coloring is a generalization of graph coloring. Rather than having a global set of colors, each vertex is given an individual list. For a graph $G = (V, E)$, a $k$-list assignment is a function
An \( L \)-coloring is a function \( c \) on \( V \) such that \( c(v) \in L(v) \) for all \( v \in V \). We say \( c \) is proper if \( c(u) \neq c(v) \) for all \( uv \in E \). We say \( G \) is \( k \)-choosable if every \( k \)-list assignment \( L \) of \( G \) admits a proper \( L \)-coloring.

It is clear that list coloring generalizes traditional coloring: a \( k \)-list assignment that assigns each vertex the same set of \( k \) colors emulates the conditions for \( k \)-coloring. One particular question is whether or not all planar graphs are 4-choosable; this turns out to be false [37]. However, it is the case that all planar graphs are 5-choosable [35]. A field of problems involve determining choosability of planar graphs given additional conditions. Theorem 1.3.4 summarizes many of the results.

**Theorem 1.3.4.** A planar graph is 4-choosable if it avoids any of the following subgraphs.

- 3-cycles (folklore)
- 4-cycles (Lam, Xu, Liu [26])
- 5-cycles (Wang, Lih [39])
- 6-cycles (Fijavž, Juvan, Mohar, Škrekovski [15])
- 7-cycles (Farzad [13])
- Chorded 4-cycles and chorded 5-cycles (Borodin, Ivanova [5])

### 1.3.3 Intersection and Union Separation

A variant to choosability involves placing restrictions on the viable list assignments. This version of choosability is called choosability with separation. In particular, choosability with intersection separation involves restricting list assignments to those where the lists of adjacent vertices have an upper bound on intersection size. We say a \( k \)-list assignment \( L \) on a graph \( G \) is a \((k, k - s)\)-list assignment if for all \( uv \in E(G) \), \(|L(u) \cap L(v)| \leq k - s\). We use the notation \( k - s \) to relate intersection separation with the complementary idea of choosability with union separation. We say \( L \) is a \((k, k + s)\)-list assignment if \(|L(u) \cup L(v)| \geq k + s\) for all \( uv \in E(G) \). For \( t = k \pm s \), we say \( G \) is \((k, t)\)-choosable if every \((k, t)\)-list assignment admits a proper coloring.

Observe that for both union and intersection separation, \((k, k)\)-choosability is equivalent to \( k \)-choosability. Furthermore, for \( r \geq s \), \((k, k - s)\)-choosability implies \((k, k - r)\)-choosability and
$(k, k+s)$-choosability implies $(k, k+r)$-choosability. Finally notice that $(k, k+s)$-choosability implies $(k, k-s)$-choosability; the reverse isn’t generally true unless we restrict our list assignments on each vertex to have size exactly $k$.

Intersection separation was introduced by Kratochvıl, Tuza, and Voigt [24], where they proved that all planar graphs are $(4,1)$-choosable and asked the question of whether or not all planar graphs are $(4,2)$-choosable; this problem remains open. There are many examples of planar graphs that are not $(4,3)$-choosable [30]. Regarding 3-choosability, Voigt [37] found examples of triangle-free planar graphs that are not 3-choosable. Škrekovski [32] found non-$(3,2)$-choosable planar graphs, and asked whether or not all planar graphs are $(3,1)$-choosable; this problem also remains open.

As an approach to the problems of $(4,2)$- and $(3,1)$-choosability of planar graphs, many have found partial results by forbidding certain subgraphs. Theorems 1.3.5 and 1.3.6 summarize the results.

**Theorem 1.3.5.** A planar graph is $(3,1)$-choosable if it avoids any of the following subgraphs.

- 3-cycles (Kratochvıl, Tuza, Voigt [24])
- 4-cycles (Choi, Lidický, Stolee [7])
- 5-cycles (Choi, Lidický, Stolee [7])

**Theorem 1.3.6.** (Berikkyzy et al. [2]) A planar graph is $(4,2)$-choosable if it avoids any of the following subgraphs.

- Chorded 5-cycles
- Chorded 6-cycles
- Chorded 7-cycles

Choosability with union separation was recently introduced [25]; the introductory paper is included as Chapter 2.
1.3.4 Packing Coloring

As a departure from list coloring, another variant is packing coloring. A \textit{k-packing coloring} of a graph \(G\) is a color assignment \(c\) such that \(c(u) = c(v) = i\) implies \(\text{dist}(u,v) > i\). The \textit{packing chromatic number} of \(G\), denoted \(\chi_p(G)\), is the smallest \(k\) such that \(G\) admits a valid \(k\)-packing coloring.

Packing coloring was introduced by Goddard et al. [18] under the term \textit{broadcast coloring}. The term was later changed to packing coloring by Brešar, Klavžar, and Rall [6].

Observe that every finite graph has a finite packing chromatic number. One question is whether or not an infinite graph has a finite packing coloring. Infinite graphs of interest are those that can be described easily, so we consider lattice graphs. A \textit{lattice graph} is an infinite graph that has an embedding in Euclidean space of some dimension (typically 2) wherein the embedding forms a regular tiling. The notable infinite graph in one dimension is the \textit{infinite path}. The infinite path is the graph whose vertex set is \(\mathbb{Z}\) and whose edge set is \(\{(i,j) : |i - j| = 1\}\). We denote the infinite path with \(P_\infty\). In two dimensions, the \textit{square lattice} \(S\) satisfies \(S = P_\infty \Box P_\infty\), and it corresponds to a square tiling of the plane. The \textit{triangular lattice} \(T\) is the 6-regular graph corresponding to a triangular tiling of the plane, and the \textit{hexagonal lattice} \(H\) is the 3-regular graph corresponding to a hexagonal tiling of the plane. Figure 1.10 shows a typical drawing of \(T, S, \) and \(H\).

![Diagram of lattice graphs](image_url)

Figure 1.10: Parts of the triangular, square, and hexagonal lattices. The tilings are infinite and cover the plane.

Finbow and Rall [16] proved that the triangular lattice does not have a finite packing chromatic number. The square lattice was shown to have finite packing chromatic number by Goddard et al. [18] with an initial upper bound of 23. Fiala, Klavžar, and Lidický [14]
gave an initial lower bound of 10. The upper bound was improved to 17 by Soukal and Holub [34], while the lower bound was improved to 12 by Ekstein, Fiala, Holub, and Lidicky [10]. B. Martin, Raimondi, Chen, and J. Martin [28] later showed that $13 \leq \chi_p(S) \leq 15$. Regarding the hexagonal lattice $\mathcal{H}$, it was shown in [18] that $6 \leq \chi_p(\mathcal{H}) \leq 8$. In [14], it was shown that $\chi_p(\mathcal{H}) \leq 7$, and Korže and Vesel [22] verified by computer that $\chi_p(\mathcal{H}) \geq 7$. So, we can conclude that $\chi_p(\mathcal{H}) = 7$.

There are further problems to explore regarding the extension of tilings to three dimensions. For instance, Fiala, Klavžar, and Lidický [14] proved that $\chi_p(P_2 \square \mathbb{Z}^2) = \infty$ and thus $\chi_p(\mathbb{Z}^3) = \infty$. They also explored one extension of $\mathcal{H}$ in three dimensions. They showed that $\chi_p(P_m \square \mathcal{H}) = \infty$ for $m \geq 6$, and thus that $\chi_p(\mathbb{Z} \square \mathcal{H}) = \infty$. It is believed that $\chi_p(P_m \square \mathcal{H}) = \infty$ for $m \geq 3$, but this has yet to be shown. Böhm, Lánsky, and Lidický [27] showed that $\chi_p(P_2 \square \mathcal{H}) \leq 526$. Figure 1.12 shows a drawing of $P_2 \square S$ and $P_2 \square \mathcal{H}$.

![Figure 1.12: Parts of the lattices $P_2 \square S$ and $P_2 \square \mathcal{H}$.](image)

### 1.4 The Discharging Method

The discharging method is a proof technique used for many graph coloring problems. It is part of a two-prong approach for inductive proofs. On one hand, there are reducible configurations, i.e. local substructures of a graph that cannot exist in a minimum counterexample, as the configurations are accompanied by instructions to form a smaller counterexample. In addition, there may be forbidden configurations that the authors decide to disallow from their graphs as part of their claim. For a strong claim, it’s desirable to minimize the number of forbidden configurations.
Discharging consists of initially assigning charge to features of a graph (i.e. vertices, edges, or faces) in such a way that the total charge is a known constant. Next, rules are put into place to move charge locally among features in such a way that a claim can be made that the total charge is not equal to the original constant. Such a claim leads to a contradiction. Typically, the constant is negative and charge is moved so that the charge on each individual feature is nonnegative. For plane graphs, Euler’s Formula (see Theorem 1.3.2) is typically used, and charge is assigned to vertices and faces. If charge $\mu(v) = a \deg(v) - 2(a + b)$ is assigned to each vertex $v$ and $\nu(f) = b \text{len}(f) - 2(a + b)$ is assigned to each face $f$, then we can compute total charge as follows.

$$\text{Total Charge} = \sum_{v \in V(G)} \mu(v) + \sum_{f \in F(G)} \nu(f)$$

$$= a \sum_{v \in V(G)} \deg(v) - 2(a + b)|V(G)| + b \sum_{f \in F(G)} \text{len}(f) - 2(a + b)|F(G)|$$

$$= 2a|E(G)| - 2(a + b)|V(G)| + 2b|E(G)| - 6|F(G)|$$

$$= 2(a + b)(|E(G)| - |V(G)| - |F(G)|) = -4(a + b)$$

Typically values of $a$ and $b$ are chosen so that $\mu(v) = \deg(v) - 4$ and $\nu(f) = \text{len}(f) - 4$, though this is just for convenience; a single discharging rule can move charge so one set of functions is analogous to another.

The challenge in discharging lies in developing the rules for moving charge. Some discharging arguments [2] are lengthy due to the high number of cases to consider.

We present Example 1.4.1 to illustrate the discharging method.

**Example 1.4.1.** We will use discharging to show that triangle-free planar graphs are 4-choosable. Suppose there exist one or more triangle-free planar graphs that are not 4-choosable. Among all such graphs, consider one with minimum order; let $G$ be a corresponding plane graph. Suppose $G$ has a vertex $v$ of degree at most 3. Then $G - v$ is 4-choosable, so for any 4-list assignment $L$ of of $G - v$, there is a proper $L$-coloring of $G - v$; this coloring clearly extends to a proper coloring of $G$. It follows that $\delta(G) \geq 4$. We could consider a vertex of degree at most 3 to be a reducible configuration, though minimum degree conditions are often treated as self-evident in discharging proofs.
Consider initial charge functions $\mu : V(G) \to \mathbb{R}; \mu(v) = 2\deg(v) - 6$ and $\nu : F(G) \to \mathbb{R}; \nu(f) = \text{len}(f) - 6$. By Euler’s Formula (Theorem 1.3.2), total charge is $-12$. Observe that all faces in $G$ have length at least 4, so the only features with negative charge are 4-faces and 5-faces. We apply the discharging rule: Each vertex sends charge $\frac{1}{2}$ to each incident face.

Each 4-face had charge $-2$ but was sent charge 2 for a final charge of 0. Each 5-face had initial charge $-1$ but was sent charge $\frac{5}{2}$ for a final charge of $\frac{3}{2}$. All other faces clearly still have non-negative charge. Each vertex $v$ sends at most charge $\frac{1}{2}\deg(v)$ for a final charge of at least $\frac{3}{2}\deg(v) - 6$. Recall that $\delta(G) \geq 4$, so all vertices have non-negative final charge. We have a contradiction, since all features in the graph have non-negative charge and total charge is negative. So, we can conclude that all triangle-free graphs are 4-choosable.

Note that this is not the simplest way to prove the result; if we assigned initial charge $\mu(v) = \deg(v) - 4$ and $\nu(f) = \text{len}(f) - 4$, then sending charge wouldn’t be necessary. However, our proof would then fail to illustrate a discharging rule.
CHAPTER 2. CHOOSABILITY WITH UNION SEPARATION

This chapter is composed of the paper Choosability with Union Separation by Mohit Kumbhat, Kevin Moss, and Derrick Stolee [25].

Abstract. List coloring generalizes graph coloring by requiring the color of a vertex to be selected from a list of colors specific to that vertex. One refinement of list coloring, called choosability with separation, requires that the intersection of adjacent lists is sufficiently small. We introduce a new refinement, called choosability with union separation, where we require that the union of adjacent lists is sufficiently large. For \( t \geq k \), a \((k, t)\)-list assignment is a list assignment \( L \) where \(|L(v)| \geq k\) for all vertices \( v \) and \(|L(u) \cup L(v)| \geq t\) for all edges \( uv \). A graph is \((k, t)\)-choosable if there is a proper coloring for every \((k, t)\)-list assignment. We explore this concept through examples of graphs that are not \((k, t)\)-choosable, demonstrating sparsity conditions that imply a graph is \((k, t)\)-choosable, and proving that all planar graphs are \((3, 11)\)-choosable and \((4, 9)\)-choosable.

2.1 Introduction

For a graph \( G \) and a positive integer \( k \), a \( k \)-list assignment of \( G \) is a function \( L \) on the vertices of \( G \) such that \( L(v) \) is a set of size at least \( k \). An \( L \)-coloring is an assignment \( c \) on the vertices of \( G \) such that \( c(v) \in L(v) \) for all vertices \( v \) and \( c(u) \neq c(v) \) for all adjacent pairs \( uv \). A graph is \( k \)-choosable if there exists an \( L \)-coloring for every \( k \)-list assignment \( L \) of \( G \), and \( G \) is \( k \)-colorable if there exists an \( L \)-coloring for the \( k \)-list assignment \( L(v) = \{1, \ldots, k\} \). The minimum \( k \) for which \( G \) is \( k \)-choosable is called the choosability or the list-chromatic number of \( G \) and is denoted by \( \chi_c(G) \). Erdős, Rubin, and Taylor [12] and independently Vizing [36] introduced the concept of list coloring and demonstrated that there exist graphs that are \( k \)-colorable but
not $k'$-choosable for all $k' \geq k \geq 2$. Since its introduction, choosability has received significant attention and has been refined in many different ways.

One refinement of choosability is called *choosability with separation* and has received recent attention [2, 7, 17, 21, 32] since it was defined by Kratochvíl, Tuza, and Voigt [24]. Let $G$ be a graph and let $s$ be a nonnegative integer called the *separation* parameter. A $(k,k-s)$-list assignment is a $k$-list assignment $L$ such that $|L(u) \cap L(v)| \leq k-s$ for all adjacent pairs $uv$. We say a graph $G$ is $(k,t)$-choosable if, for any $(k,t)$-list assignment $L$, there exists an $L$-coloring of $G$. As the separation parameter $s$ increases, the restriction on the intersection-size of adjacent lists becomes more strict.

We introduce a complementary refinement of choosability called *choosability with union separation*. A $(k,k+s)$-list assignment is a $k$-list assignment $L$ such that $|L(u) \cup L(v)| \geq k+s$ for all adjacent pairs $uv$. We similarly say $G$ is $(k,t)$-choosable to imply choosability with either kind of separation, depending on $t \leq k$ or $k < t$. Observe that if $G$ is $(k,k+s)$-choosable, then $G$ is both $(k,k-r)$-choosable and $(k,k+r)$-choosable for all $r \geq s$. Note that if $L$ is a $(k,k-s)$-list assignment, we may assume that $|L(v)| = k$ as removing colors from lists does not violate the intersection-size requirement for adjacent vertices. However, when considering a $(k,k+s)$-list assignment, we may not remove colors from lists as that may violate the union-size requirement for adjacent vertices. Due to this asymmetry, we do not know if there is a function $f(k,s)$ such that every $(k,k-s)$-choosable graph is also $(k,k+f(s))$-choosable.

Thomassen [35] proved that all planar graphs are 5-choosable. The main question we consider regarding planar graphs and choosability with union separation is identifying minimum integers $t_3$ and $t_4$ such that all planar graphs are $(3,t_3)$-choosable and $(4,t_4)$-choosable. We demonstrate that $6 \leq t_3 \leq 11$ and $6 \leq t_4 \leq 9$.

Kratochvíl, Tuza, and Voigt [23] proved that all planar graphs are $(4,1)$-choosable and conjecture that all planar graphs are $(4,2)$-choosable. Voigt [37] constructed a planar graph that is not $(4,3)$-choosable and hence is not $(4,5)$-choosable. We show that $t_4 \leq 9$. 
Theorem 2.1.1. All planar graphs are \((4, 9)\)-choosable.

A chorded \(\ell\)-cycle is a cycle of length \(\ell\) with one additional edge. For each \(\ell \in \{5, 6, 7\}\), Berikkyzy et al. [2] demonstrated that if \(G\) is a planar graph that does not contain a chorded \(\ell\)-cycle, then \(G\) is \((4, 2)\)-choosable. The case \(\ell = 4\) is notably missing from their results, especially since Borodin and Ivanova [5] proved that if \(G\) is a planar graph that does not contain a chorded 4-cycle or a chorded 5-cycle, then \(G\) is 4-choosable. We prove that if \(G\) is a planar graph containing no chorded 4-cycle, then \(G\) is \((4, 7)\)-choosable (see Theorem 2.5.1).

Kratochvıl, Tuza, and Voigt [23] conjecture that all planar graphs are \((3, 1)\)-choosable. Voigt [38] constructed a planar graph that is not \((3, 2)\)-choosable and hence is not \((3, 4)\)-choosable. In Section 2.2 we construct graphs that are not \((k, t)\)-choosable, including a planar graph that is not \((3, 5)\)-choosable. This hints towards a strong difference between intersection separation and union separation. We show that \(t_3 \leq 11\).

Theorem 2.1.2. All planar graphs are \((3, 11)\)-choosable.

We also consider sparsity conditions that imply \((k, t)\)-choosability. For a graph \(G\), the maximum average degree of \(G\), denoted \(\text{Mad}(G)\), is the maximum fraction \(\frac{2|E(H)|}{|V(H)|}\) among subgraphs \(H \subseteq G\). If \(\text{Mad}(G) < k\), then \(G\) is \((k - 1)\)-degenerate and hence is \(k\)-choosable. Since \(\text{Mad}(K_{k+1}) = k\) and \(\chi_\ell(K_{k+1}) > k\), this bound on \(\text{Mad}(G)\) cannot be relaxed. In Section 2.4, we prove that \(G\) is \((k, t)\)-choosable when \(\text{Mad}(G) < 2k - o(1)\) where \(o(1)\) tends to zero as \(t\) tends to infinity. This is asymptotically sharp as we construct graphs that are not \((k, t)\)-choosable with \(\text{Mad}(G) = 2k - o(1)\).

Many of our proofs use the discharging method. For an overview of this method, see the surveys of Borodin [4], Cranston and West [9], or the overview in Berikkyzy et al. [2]. We use a very simple reducible configuration that is described by Proposition 2.3.1 in Section 2.3.

2.1.1 Notation

A (simple) graph \(G\) has vertex set \(V(G)\) and edge set \(E(G)\). Additionally, if \(G\) is a plane graph, then \(G\) has a face set \(F(G)\). Let \(n(G) = |V(G)|\) and \(e(G) = |E(G)|\). For a vertex \(v \in V(G)\), the set of vertices adjacent to \(v\) is the neighborhood of \(v\), denoted \(N(v)\). The degree
of \( v \), denoted \( d(v) \), is the number of vertices adjacent to \( v \). We say \( v \) is a \( k \)-vertex if \( d(v) = k \), a \( k^- \)-vertex if \( d(v) \leq k \) and a \( k^+ \)-vertex if \( d(v) \geq k \). Let \( G - v \) denote the graph given by deleting the vertex \( v \) from \( G \). For an edge \( uv \in E(G) \), let \( G - uv \) denote the graph given by deleting the edge \( uv \) from \( G \). For a plane graph \( G \) and a face \( f \), let \( \ell(f) \) denote the length of the face boundary walk; say \( f \) is a \( k \)-face if \( \ell(f) = k \) and a \( k^+ \)-face if \( \ell(f) \geq k \).

### 2.2 Non-(\( k, t \))-Choosable Graphs

**Proposition 2.2.1.** For all \( t \geq k \geq 2 \), there exists a bipartite graph that is not \((k, t)\)-choosable.

**Proof.** Let \( u_1, \ldots, u_k \) be nonadjacent vertices and let \( L(u_1), \ldots, L(u_k) \) be disjoint sets of size \( t - k + 1 \). For every element \( (a_1, \ldots, a_k) \in \prod_{i=1}^{k} L(u_i) \), let \( A = \{a_1, \ldots, a_k\} \), create a vertex \( x_A \) adjacent to \( u_i \) for all \( i \in [k] \), and let \( L(x_A) = A \) (see Figure 2.1). Notice that \( |L(u_i) \cup L(x_A)| = t \) for all \( i \in [k] \) and all vertices \( x_A \), so \( L \) is a \((k, t)\)-list assignment. If there is a proper \( L \)-coloring \( c \) of this graph, then let \( A = \{c(u_i) : i \in [k]\} \); the color \( c(x_A) \) is in \( A \) and hence the coloring is not proper. \( \square \)

![Figure 2.1: A graph that is not \((k, t)\)-choosable.](image)

Observe that the graph constructed in Proposition 2.2.1 has average degree \( \frac{2k(t-k+1)^k}{k+(t-k+1)^k} \); as \( t \) increases, this fraction approaches \( 2k \) from below. Observe that when \( k = 2 \) the graph built in Proposition 2.2.1 is planar, giving us the following corollary.

**Corollary 2.2.2.** For all \( t \geq 2 \), there exists a bipartite planar graph that is not \((2, t)\)-choosable.

We now construct a specific planar graph that is not \((3, 5)\)-choosable.

**Proposition 2.2.3.** There exists a planar graph that is not \((3, 5)\)-choosable.
Proof. Let $A$ and $B$ be disjoint sets of size three, and let $c_1, \ldots, c_4$ be distinct colors not in $A \cup B$. Let $v_A$ and $v_B$ be two vertices and let $L(v_A) = A$ and $L(v_B) = B$. For each $a \in A$ and $b \in B$, consider the graph displayed in Figure 2.2; create a copy of this graph where the left vertex is $v_A$ and the right vertex is $v_B$. Assign lists to the interior vertices of this graph using the colors $\{a, b, c_1, \ldots, c_4\}$ as shown in the figure. Observe that $L$ is a $(3, 5)$-list assignment. If there exists a proper $L$-coloring, then let $a \in A$ be the color on $v_A$ and $b \in B$ be the color on $v_B$ and consider the copy of this gadget using these colors. Observe that in the 4-cycle induced by the neighbors of the center vertex, all four colors $c_1, \ldots, c_4$ must be present. Then the coloring is not proper as the center vertex is assigned one of these colors.

2.3 Reducible Configurations

To prove all of our main results, we consider a minimum counterexample and arrive at a contradiction through discharging. In this section, we describe the structures that cannot appear in a minimum counterexample.

Proposition 2.3.1. Let $G$ be a graph, $uv$ an edge in $G$, $t \geq k \geq 3$, and $a = |N(u) \cap N(v)|$ with $a \in \{0, 1, 2\}$. Let $L$ be a $(k, t)$-list assignment and suppose that there exist $L$-colorings of $G - u$, $G - v$, and $G - uv$. If $d(u) + d(v) \leq t + a$, then there exists an $L$-coloring of $G$.

Proof. If $|L(u)| > d(u)$, then the $L$-coloring of $G - u$ extends to an $L$-coloring of $G$ as there is a color in $L(u)$ that does not appear among the neighbors of $u$; thus we assume $|L(u)| \leq d(u)$. By a symmetric argument we may assume $|L(v)| \leq d(v)$. If $L(u) \cap L(v) = \emptyset$, then the $L$-coloring of $G - uv$ is an $L$-coloring of $G$; thus we assume $|L(u) \cap L(v)| \geq 1$ and $|L(u) \cup L(v)| \leq |L(u)| + |L(v)| - 1 \leq d(u) + d(v) - 1$. 
Note that $t + 2 \geq d(u) + d(v) \geq |L(u)| + |L(v)| \geq t + 1$. Thus $a \geq 1$ and either $|L(u)| = d(u)$ or $|L(v)| = d(v)$; assume by symmetry that $|L(u)| = d(u)$. Let $c$ be an $L$-coloring of $G - u$. For $x \in \{u, v\}$, let $L'(x)$ be the colors in $L(x)$ that do not appear among the neighbors $y \in N(x) \setminus \{u, v\}$. Since $c(v) \in L'(v)$, we have $L'(v) \neq \emptyset$. Since $|L(u)| = d(u)$, we have $L'(u) \neq \emptyset$. Observe that $|L'(u) \cup L'(v)| \geq |L(u) \cup L(v)| - |N(u) \cup N(v)| + 2 \geq t - (d(u) + d(v) - a) + 2 \geq 2$. Thus either $|L'(u)| = |L'(v)| = 1$ or $|L'(x)| \geq 2$ for some $x \in \{u, v\}$ and therefore there are choices for $c'(u) \in L'(u)$ and $c'(v) \in L'(v)$ such that $c'(u) \neq c'(v)$. If $c'(y) = c(y)$ for all $y \in V(G) \setminus \{u, v\}$, then $c'$ is an $L$-coloring of $G$. 

\[ \square \]

### 2.4 Sparse Graphs

In this section, we determine a relationship between sparsity and choosability with union separation.

**Theorem 2.4.1.** Let $k \geq 2$ and $t \geq 2k - 1$. If $G$ is a graph with $\text{Mad}(G) < 2k \left(1 - \frac{k}{t+1}\right)$, then $G$ is $(k, t)$-choosable.

**Proof.** Let $c = 2k - \frac{2k^2}{t+1}$. Observe that since $t \geq 2k - 1$ that $c \geq k$. For the sake of contradiction, suppose there exists a graph $G$ with $\text{Mad}(G) < c$ and a $(k, t)$-list assignment $L$ such that $G$ is not $L$-choosable. Select $(G, L)$ among such pairs to minimize $n(G) + e(G)$. Observe that $k \leq |L(v)| \leq d(v)$ for every vertex.

We use discharging to demonstrate $\text{Mad}(G) \geq c$, a contradiction. Assign charge $d(v)$ to every vertex $v$, so the total charge sum is equal to $2e(G)$. We discharge using the following rule:

(R) If $u$ is a vertex with $d(u) < c$, then $u$ pulls charge $\frac{c - d(u)}{d(u)}$ from each neighbor of $u$.

Suppose that $v$ is a vertex that loses charge by (R). Then there exists an edge $uv \in E(G)$ where $d(u) < c$. Note that since $G - uv$ is $L$-choosable, $|L(u) \cap L(v)| \geq 1$ and $d(u) + d(v) \geq t + 1$. It follows that$$d(v) \geq t + 1 - d(u) > t + 1 - c \geq c.$$
Therefore, a vertex either loses charge by (R) or gains charge by (R), not both.

Observe that if \( d(u) < c \), then \( u \) pulls enough charge by (R) to end with charge at least \( c \).

Finally, suppose \( v \) is a vertex with \( d(v) = d \geq c \).

If \( d \geq t + 1 - k \), then neighbors of \( v \) pull charge at most \( \frac{c-k}{k} \) from \( v \). The final charge on \( v \) is given by

\[
d - d\left(\frac{c-k}{k}\right) = d \left(\frac{2k - c}{k}\right) = d \left(\frac{2k - (2k - \frac{2k^2}{t+1})}{k}\right) = d \left(\frac{2k}{t+1}\right) \geq (t + 1 - k) \frac{2k}{t+1} = c.
\]

Now suppose that \( d < t + 1 - k \). If a vertex \( u \) pulls charge from \( v \) by (R), then \( d(u) \geq d' = t + 1 - d \). Thus, \( v \) loses charge at most \( \frac{c-d'}{d'} \) to each neighbor. The final charge on \( v \) is given by

\[
d - d\left(\frac{c-d'}{d'}\right) = d \left(1 - \frac{c-d'}{d'}\right) = d \left(\frac{2d' - c}{d'}\right) = d \left(\frac{2(t + 1 - d) - c}{t + 1 - d}\right).
\]

Observe that \( d \left(\frac{2(t + 1 - d) - c}{t + 1 - d}\right) \geq c \) if and only if if \( 2(t + 1 - d)d - (t + 1)c \geq 0 \). By the quadratic formula, this polynomial (in \( d \)) has roots at \( d \in \left\{ \frac{1}{2} \left( t + 1 \pm \sqrt{(t + 1)(t + 1 - 2c)} \right) \right\} \); the discriminant is nonnegative since \( (t+1)((t+1)-2c) = (t+1-2k)^2 \). Thus, the final charge on \( v \) is below \( c \) if and only if \( d < k \) or \( d > t + 1 - k \), but we are considering \( d \) where \( k \leq c \leq d < t + 1 - k \).

Note that Theorem 2.4.1 implies that a graph \( G \) is \((4,15)\)-choosable when \( \text{Mad}(G) < 8 \left(1 - \frac{4}{16}\right) = 6 \). If \( G \) is planar, then \( \text{Mad}(G) < 6 \) and hence is \((4,15)\)-choosable. There is no \( t \) such that Theorem 2.4.1 implies all planar graphs are \((3,t)\)-choosable. We now directly consider planar graphs and find smaller separations suffice.

## 2.5 (4,\( t \))-choosability

**Proof of Theorem 2.1.1.** Suppose \( G \) is a plane graph minimizing \( n(G) + e(G) \) such that \( G \) is not \( L \)-colorable for some \((4,9)\)-list assignment \( L \). By minimality of \( G \), we can assume that \( d(v) \geq |L(v)| \geq 4 \) for all vertices \( v \) and \( |L(u) \cap L(v)| \geq 1 \) for all adjacent pairs \( uv \). By Proposition 2.3.1, if \( uv \) is an edge in \( G \), then \( d(u) + d(v) > 9 + \min(|N(u) \cap N(v)|, 2) \). Observe that \( \min(|N(u) \cap N(v)|, 2) \) is at least the number of 3-faces incident to the edge \( uv \).
For each \( v \in V(G) \) and \( f \in F(G) \) define \( \mu(v) = d(v) - 4 \) and \( \nu(f) = \ell(f) - 4 \). Note that the total initial charge of \( G \) is \(-8\). For a vertex \( v \), let \( t_3(v) \) be the number of 3-faces incident to \( v \). Apply the following discharging rule.

(R1) If \( v \) is a 5\(^+\)-vertex and \( f \) is an incident 3-face, then \( v \) sends charge \( \frac{\mu(v)}{t_3(v)} \) to \( f \).

All vertices and 4\(^+\)-faces have nonnegative charge after applying (R1).

Let \( f \) be a 3-face with incident vertices \( u, v, w \) where \( d(u) \leq d(v) \leq d(w) \). Since \( \nu(f) = -1 \), it suffices to show that \( f \) receives charge at least 1 in total from \( u, v, \) and \( w \) by (R1).

If \( d(u) \geq 6 \), then \( \frac{\mu(x)}{d(x)} \geq \frac{1}{3} \) for all \( x \in \{u, v, w\} \) and each vertex \( u, v, \) and \( w \) sends charge at least \( \frac{1}{3} \), giving \( f \) nonnegative final charge.

If \( d(u) = 4 \), then \( d(w) \geq d(v) \geq 7 \) since \( d(u) + d(v) \geq 11 \) by Proposition 2.3.1. If \( d(v) \geq 8 \), then each of \( v \) and \( w \) send charge at least \( \frac{1}{2} \), giving \( f \) nonnegative final charge. Thus, suppose \( d(v) = 7 \). Since \( d(u) + d(v) = 11 \), there is not another 3-face incident to the edge \( uv \) by Proposition 2.3.1. Thus, \( t_3(v) \leq 6 \) and hence \( v \) sends charge at least \( \frac{1}{2} \) to \( f \). Similarly, \( w \) sends charge at least \( \frac{1}{2} \) so \( f \) has nonnegative final charge.

If \( d(u) = 5 \), then \( d(v) \geq 6 \) since \( d(u) + d(v) \geq 11 \) by Proposition 2.3.1. If \( d(v) \geq 7 \), then vertex \( u \) sends charge at least \( \frac{1}{5} \) and each of \( v \) and \( w \) send charge at least \( \frac{3}{7} \), giving \( f \) nonnegative final charge. If \( d(v) = 6 \), then there is not another 3-face incident to the edge \( uv \) by Proposition 2.3.1. Thus, \( t_3(v) \leq 5 \) and \( v \) sends charge at least \( \frac{2}{5} \). Similarly, \( w \) sends charge at least \( \frac{2}{5} \) so \( f \) has nonnegative final charge.

We conclude that all vertices and faces have nonnegative charge, so \( G \) has nonnegative total charge, a contradiction.

**Theorem 2.5.1.** If \( G \) is a planar graph and does not contain a chorded 4-cycle, then \( G \) is \((4,7)\)-choosable.

**Proof.** Suppose \( G \) is a plane graph minimizing \( n(G) + e(G) \) such that \( G \) does not contain a chorded 4-cycle and \( G \) is not \( L \)-colorable for some \((4,7)\)-list assignment \( L \). By minimality of \( G \), we can assume that \( d(v) \geq |L(v)| \geq 4 \) for all vertices \( v \) and \( |L(u) \cap L(v)| \geq 1 \) for all adjacent
pairs \(uv\). In particular, no two adjacent vertices of degree 4 share a 3-face by Proposition 2.3.1. Let the initial charge of a vertex be \(d(v) - 4\) and that of a face be \(\ell(f) - 4\). Note that the total initial charge of \(G\) is \(-8\). For a vertex \(v\), let \(t_3(v)\) be the number of 3-faces incident to \(v\). Apply the following discharging rule.

\((\text{R1})\) If \(v\) is a 5\(^+\)-vertex and \(f\) is an incident 3-face, then \(v\) sends charge \(\frac{\mu(v)}{t_3(v)}\) to \(f\).

Since chorded 4-cycles are forbidden, no two 3-faces can share an edge. Hence for each vertex \(v \in V(G)\), there are at most \(\lfloor \frac{d(v)}{2} \rfloor\) 3-faces incident to \(v\). It follows that vertices of degree at least 5 send charge at least \(\frac{1}{2}\) to each incident 3-face. Since a 3-face has at most one incident 4-vertex by Proposition 2.3.1, all 3-faces have nonnegative final charge after \((\text{R1})\). Hence all vertices and faces have nonnegative final charge, so \(G\) has nonnegative total charge, a contradiction.

2.6 \((3, 11)\)-choosability

\textit{Proof of Theorem 2.1.2.} Suppose \(G\) is a plane graph minimizing \(n(G) + e(G)\) such that \(G\) is not \(L\)-colorable for some \((3, 11)\)-list assignment \(L\). By minimality of \(G\), we can assume that \(d(v) \geq |L(v)| \geq 3\) for all vertices \(v\) and \(|L(u) \cap L(v)| \geq 1\) for all adjacent pairs \(uv\). By Proposition 2.3.1, if \(uv\) is an edge in \(G\), then \(d(u) + d(v) > 11 + \min(|N(u) \cap N(v)|, 2)\). Observe that \(\min(|N(u) \cap N(v)|, 2)\) is at least the number of 3-faces incident to the edge \(uv\).

For each \(v \in V(G)\) and \(f \in F(G)\) define initial charge functions \(\mu(v) = d(v) - 6\) and \(\nu(f) = 2\ell(f) - 6\). By Euler’s formula, total charge is \(-12\). Apply the following discharging rules:

\((\text{R1})\) Let \(v\) be a vertex and \(u \in N(v)\).

(a) If \(d(v) = 3\), then \(v\) pulls charge 1 from \(u\).

(b) If \(d(v) = 4\), then \(v\) pulls charge \(\frac{1}{2}\) from \(u\).

(c) If \(d(v) = 5\), then \(v\) pulls charge \(\frac{1}{3}\) from \(u\).

\((\text{R2})\) If \(f\) is a 4\(^+\)-face and \(uv\) is an edge incident to \(f\) with \(d(u) \leq 5\), then \(f\) sends charge \(\frac{1}{2}\) to \(v\).
We claim the final charge on all faces and vertices is nonnegative. Since the total charge sum was preserved during the discharging rules, this contradicts the negative initial charge sum. Observe that no two $5^-$-vertices are adjacent by Proposition 2.3.1, so each face $f$ is incident to at most $\frac{\ell(f)}{2}$ vertices of degree at most five. If $f$ is a 3-face, then $f$ does not lose charge. If $f$ is a $4^+$-face, then $f$ loses charge at most 1 per incident $5^-$-vertex. We have $\frac{\ell(f)}{2} \leq 2\ell(f) - 6$ whenever $\ell(f) \geq 4$, so $f$ has nonnegative final charge.

Each $5^-$-vertex gains exactly enough charge through (R1) so that the final charge is nonnegative.

Suppose $v$ is a $6^+$-vertex. We introduce some notation to describe the structure near $v$. For an edge $uv$, let $a(uv)$ be the number of 3-faces incident to the edge $uv$. Note that if $d(u) < 6$ and $a(uv) = 0$, then $v$ sends charge at most 1 to $u$ by (R1) and gains charge at least 1 via $uv$ by (R2), giving a nonnegative net difference in charge. Thus, if $v$ ends with negative charge, it must be due to some number of $5^-$-vertices $u \in N(v)$ with $a(uv) > 0$.

For $k \in \{3, 4, 5\}$, let $D_k$ be the set of neighbors $u$ of $v$ such that $u$ is a $k$-vertex and $a(uv) = 2$; let $d_k = |D_k|$. Let $D_3^*$ be the set of neighbors $u$ of $v$ such that $u$ is a 3-vertex and $a(uv) = 1$; let $d_3^* = |D_3^*|$. If $u \in D_k$, then $v$ gains no charge via $uv$ in (R2). If $u \in D_3^*$, then $v$ loses charge 1 to $u$ in (R1) but gains charge $\frac{1}{2}$ via $uv$ in (R2). Therefore, the final charge of $v$ at least $\mu(v) - d_3 - \frac{1}{2}d_3^* - \frac{1}{2}d_4 - \frac{1}{5}d_5$. Recall $\mu(v) = d(v) - 6$, so if $v$ has negative final charge, then
\[
d_3 + \frac{1}{2}d_3^* + \frac{1}{2}d_4 + \frac{1}{5}d_5 > d(v) - 6. \tag{2.1}
\]

Let $D = D_3 \cup D_3^* \cup D_4 \cup D_5$. For each 3-face $uvw$ incident to $v$, at most one of $u, w$ is in $D$. If $u \in D$, $w \in N(v) \setminus D$, and $uvw$ is a 3-face, then $u$ gives a strike to $w$. Each vertex in $D_3 \cup D_4 \cup D_5$ contributes two strikes, and each vertex in $D_3^*$ contributes one strike. The total number of strikes is $2d_3 + d_3^* + 2d_4 + 2d_5$ and each vertex $w \in N(v) \setminus D$ receives at most two strikes, so $2d_3 + d_3^* + 2d_4 + 2d_5 \leq 2(d(v) - (d_3 + d_3^* + d_4 + d_5))$. Equivalently,
\[
2d_3 + \frac{3}{2}d_3^* + 2d_4 + 2d_5 \leq d(v). \tag{2.2}
\]
We now have $d(v) \geq 6$ and the two inequalities (2.1) and (2.2). Also recall that since $d(u) + d(v) > 11 + \min(|N(u) \cap N(v)|, 2)$, we have the following implications: if $d(v) \leq 10$ then $d_3 = 0$; if $d(v) \leq 9$, then $d_3^* + d_4 = 0$; if $d(v) \leq 8$, then $d_5 = 0$.

If we subtract (2.1) from (2.2), then we find the following inequality.

$$d_3 + d_3^* + \frac{3}{2} d_4 + \frac{9}{5} d_5 < 6.$$  \hfill (2.3)

There are 77 tuples $(d_3, d_3^*, d_4, d_5)$ of nonnegative integers that satisfy (2.3); see Appendix A for the full list. None of these tuples admit a value $d(v)$ that satisfies (2.1) and the implications. Therefore, there is no $6^+$-vertex $v$ with negative final charge. We conclude that all vertices and faces have nonnegative final charge. But total charge is $-12$, a contradiction. Thus a minimum counterexample does not exist and all planar graphs are $(3, 11)$-choosable.
CHAPTER 3. TOWARDS (3,10)-CHOOSABILITY

3.1 Introduction

We suspect that Theorem 2.1.2 can be made stronger. In particular, we introduce Conjecture 3.2.1. An approach using discharging has yielded promising results, but the complexity of the discharging argument increased significantly when stepping from (3, 11)-choosability to (3, 10)-choosability. We provide Theorem 3.2.2 as a partial result.

3.2 (3,10)-Choosability

Conjecture 3.2.1. All planar graphs are (3,10)-choosable.

We introduce some notation to aid in the description of configurations relevant to the discharging argument. These are similar to those in the proof of Theorem 2.1.2: Let $G$ be a plane graph. For a vertex $v \in V(G)$ and for $k \in \{3, 4, 5\}$, let $D_k(v)$ be the set of neighbors $u$ of $v$ such that $u$ is a $k$-vertex and edge $uv$ is incident to two 3-faces. Let $d_k(v) = |D_k(v)|$. Let $D^*_3(v)$ be the set of neighbors $u$ of $v$ such that such that $v$ is a 3-vertex and exactly one face incident to edge $uv$ is a 3-face. Let $d^*_3(v) = |D^*_3(v)|$.

Theorem 3.2.2. Suppose $G$ is a plane graph of minimum (order + size) such that $G$ is not (3,10)-choosable, and let $L$ be a (3,10)-list assignment of $G$ that does not admit a proper $L$-coloring. Then $G$ contains a vertex $v$ that satisfies one of the following cases:

- $\deg(v) = 11$, $L(v) = 9$, and $(d_3(v), d^*_3(v), d_4(v), d_5(v)) = (3, 3, 0, 0)$
- $\deg(v) = 11$, $L(v) = 9$, and $(d_3(v), d^*_3(v), d_4(v), d_5(v)) = (4, 2, 0, 0)$
- $\deg(v) = 12$, $L(v) \in \{9, 10\}$, and $(d_3(v), d^*_3(v), d_4(v), d_5(v)) = (5, 1, 0, 0)$
• \( \deg(v) = 12 \), \( L(v) \in \{9, 10\} \), and \( (d_3(v), d_5^*(v), d_4(v), d_5(v)) = (6, 0, 0, 0) \)

Before proceeding with the proof of Theorem 3.2.2, we provide Lemmas 3.2.3, 3.2.4, and 3.2.5; each of which describes a reducible configuration.

### 3.2.1 Reducible Configurations

#### Lemma 3.2.3

Let \( C_1 \) be the graph shown in Figure 3.1. Suppose a plane graph \( G \) contains a (not necessarily induced) copy of \( C_1 \) wherein \( \deg(v) = 10 \) and \( \deg(u_2) = \deg(u_4) = \deg(u_6) = 3 \). Suppose \( L \) is a \((3, 10)\)-list assignment of \( G \) such that \( |L(v)| < 10 \) and every proper subgraph of \( G \) can be properly \( L \)-colored. Then \( G \) can be properly \( L \)-colored.

![Figure 3.1: Configuration C1.](image)

**Proof:** We will extend a proper \( L \)-coloring \( c \) of \( G - \{v, u_2, u_4, u_6\} \) to \( G \). Note that \( |L(w)| \leq \deg(w) \) for every \( w \in V(G) \), or else a proper coloring of \( G - \{w\} \) extends to a proper coloring of \( G \). In particular, \( |L(u_2)| = 3 \). We know \( G - \{u_2, v\} \) can be properly colored, so \( |L(u_2) \cap L(v)| \geq 1 \). Hence \( |L(v)| + |L(u_2)| = |L(v) \cup L(u_2)| + |L(v) \cap L(u_2)| \geq 11 \) and \( |L(v)| \geq 8 \). We will consider the cases \( |L(v)| = 8 \) and \( |L(v)| = 9 \) separately.

First suppose \( |L(v)| = 8 \). Let the colors chosen for \( u_1, u_3, u_5, \) and \( u_7 \) in \( c \) be \( c_1, c_3, c_5, \) and \( c_7 \) respectively. For \( i \in \{2, 4, 6\} \), if \( \{c_{i-1}, c_{i+1}\} \not\subseteq L(u_i) \), then \( u_i \) is guaranteed an available color if colored after \( v \). We know \( |L(u_i) \setminus L(v)| \geq 2 \), so if \( \{c_{i-1}, c_{i+1}\} \subseteq L(u_i) \) and \( c_{i-1} \in L(v) \) or \( c_{i+1} \in L(v) \), then \( u_i \) is guaranteed an available color if colored after \( v \). If at least three of \( c_1, c_3, c_5, \) and \( c_7 \) lie in \( L(v) \), then we can obtain a proper coloring by greedily coloring in
order $v, u_2, u_4, u_6$. If exactly two of $c_1, c_3, c_5$, and $c_7$ lie in $L(v)$, then at most one of $u_2, u_4$, and $u_6$, is not guaranteed an available color if colored after $v$. Greedily color in an ordering with this vertex before $v$ and the others after, and we have a proper coloring. If one or fewer of $c_1, c_3, c_5$, and $c_7$ lie in $L(v)$, then we can obtain a proper coloring by greedily coloring in order $u_2, u_4, u_6, v$.

Now suppose $|L(v)| = 9$. Again let the colors chosen for $u_1, u_3, u_5$, and $u_7$ be $c_1, c_3, c_5$, and $c_7$ respectively. For $i \in \{2, 4, 6\}$, if $\{c_{i-1}, c_{i+1}\} \not\subseteq L(u_i)$, then $u_i$ is guaranteed an available color if colored after $v$. We know $|L(u_i)\setminus L(v)| \geq 1$, so if $\{c_{i-1}, c_{i+1}\} \subseteq L(u_i) \cap L(v)$, then $u_i$ is guaranteed an available color if colored after $v$. If $\{c_1, c_3, c_5, c_7\} \subseteq L(v)$, then we can obtain a proper coloring by greedily coloring in order $v, u_2, u_4, u_6$. If exactly three of $c_1, c_3, c_5$, and $c_7$ lie in $L(v)$, then at least one of $u_2, u_4, u_6$ can be colored after $v$. Greedily color in an ordering with this vertex after $v$ and the others before, and we have a proper coloring. If two or fewer of $c_1, c_3, c_5$, and $c_7$ lie in $L(v)$, then we can obtain a proper coloring by greedily coloring in order $u_2, u_4, u_6, v$. \hfill \square

Lemma 3.2.4. Let $C2$ be the graph shown in Figure 3.2. Suppose a graph $G$ contains a (not necessarily induced) copy of $C2$ wherein $\text{deg}(v) = 10$ and $\text{deg}(u_2) = \text{deg}(u_4) = 3$. Suppose $L$ is a $(3,10)$-list assignment of $G$ such that $|L(v)| = 10$ and every proper subgraph of $G$ can be properly colored. If one of the following holds, then $G$ can be properly colored.

(a) $|L(u_3)| \geq \text{deg}(u_3) - 1$

(b) $|L(u_3)| = \text{deg}(u_3) - 2$, exactly one of $u_1u_3$ and $u_3u_5$ lies in $E(G)$, and $\text{deg}(u_3) \leq 10$

(c) $|L(u_3)| = \text{deg}(u_3) - 3$, $u_1u_3, u_3u_5 \in E(G)$, and $\text{deg}(u_3) \leq 11$

Proof: First note that $|L(w)| \leq \text{deg}(w)$ for every $w \in V(G)$, or else a proper coloring of $G - \{w\}$ extends to a proper coloring of $G$. Now consider a proper coloring of $G - \{v\}$. If there is no way to extend the coloring to $G$, then it must be the case that each neighbor of $v$ has a distinct color. Let the color choices of $u_1, u_2, u_3, u_4$, and $u_5$ be $a, b, c, d, e$ respectively. If $L(u_2) \neq \{a, b, c\}$, then we can choose $b$ for $v$ and recolor $u_2$. Similarly, we can find a proper coloring if $L(u_4) \neq \{c, d, e\}$. So suppose $L(u_2) = \{a, b, c\}$ and $L(u_4) = \{c, d, e\}$. 

Let \( k = |\{u_1u_3, u_3u_5\} \cap E(G)|. \) In case (a), there at most \(|L(u_3)| - 2 - k\) neighbors of \( u_3 \) outside \( \{u_1, u_2, v, u_4, u_5\} \). So, there are at least \( 2 + k \) color choices for \( u_3 \) that do not interfere with vertices outside \( \{u_1, u_2, v, u_4, u_5\} \), one of which is color \( c \). We will recolor \( u_3 \) with another available color, denoted \( c' \). If \( u_1u_3 \in E(G) \), then we have enough available colors to choose \( c' \neq a \). Similarly, if \( u_3u_5 \in E(G) \), then we have enough available colors to choose \( c' \neq e \). To complete the coloring, color \( u_2 \) and \( u_4 \) with \( c \). Since \( |L(v)| = \deg(v) \) and two neighbors of \( v \) have the same color, there is an available color for \( v \).

In cases (b) and (c), there at most \(|L(u_3)| - 2\) neighbors of \( u_3 \) outside \( \{u_1, u_2, v, u_4, u_5\} \). So, there are at least two color choices for \( u_3 \) that do not interfere with vertices outside \( \{u_1, u_2, v, u_4, u_5\} \), one of which is color \( c \). We denote the other color \( c' \). We know \( |L(u_3)| \leq 8 \) and \( |L(u_2) \cup L(u_3)| \geq 10 \), so \( |L(u_2) \setminus L(u_3)| \geq 2 \) and \( a, b \notin L(u_3) \). Similarly, \( d, e \notin L(u_3) \). It follows that \( c' \notin \{a, e\} \). So, if we color \( u_3 \) with \( c' \) and \( u_2 \) and \( u_4 \) with \( c \), there is an available color for \( v \) that completes a proper coloring.

\[ \square \]

**Lemma 3.2.5.** Suppose \( G \) is a plane graph with \((3,10)\)-list assignment \( L \). If a vertex \( u \in V(G) \) satisfies \(|L(u)| = 8 \) and \( d_3(u) + \frac{d_3^+(u)}{2} \geq \deg(u) - 7 \), and \( G - \{u\} \) has a proper \( L \)-coloring, then \( G \) has a proper \( L \)-coloring.

**Proof:** Let \( D = D_3(u) \cup D_3^+(u) \), \( P = N(u) \setminus D \), and \( p = |P| \). Choose a proper \( L \)-coloring of \( G - D - \{u\} \). We call each vertex in \( G - D - \{u\} \) precolored. We claim the coloring extends
to a proper $L$-coloring of $G$. In particular, we claim there is an ordering of $D \cup \{u\}$ along which we may greedily color to obtain a proper coloring. Any vertex in $D$ colored before $u$ is guaranteed an available color, since its list size is three and at most two neighbors have already been colored.

In the coloring, $p$ neighbors of $u$ are precolored. Let $F \subseteq P$ be the set of precolored vertices that have had a color chosen which is contained in $L(u)$, and let $f = |F|$. For vertex $v \in F$, let the color chosen for $v$ be $\alpha$. Suppose there exists some $w \in D$ incident to $v$. If $\alpha \notin L(w)$, then no matter where $w$ is in our ordering, there will be a color available for $w$. Suppose $\alpha \in L(w)$. We know $|L(w)| = 3$ and $|L(w) \cup L(u)| \geq 10$, so $|L(v) \cap L(u)| \leq 1$ and $L(v) \cap L(u) = \{\alpha\}$. Again, no matter where $w$ is in the ordering of a greedy coloring, there will be a color available for $w$. In both cases, we choose to color $w$ after $u$. Vertices in $D$ not incident to a vertex in $F$ will be colored before $u$.

For each $v \in F$, we say $v$ gives a strike to any adjacent vertex in $D$. A strike denotes a counter. Each vertex in $D_3(u)$ is adjacent to two vertices in $P$, and each vertex in $D^*_3(u)$ is adjacent to one vertex in $P$. So, there are $2d_3(u) + d^*_3(u)$ edges between $P$ and $D$. Each vertex in $P$ is adjacent to at most two vertices in $D$, so there are at most $2(p - f)$ edges between vertices in $P \setminus F$ and vertices in $D$. It follows that there are at least $2d_3(u) + d^*_3(u) - 2(p - f)$ edges between vertices in $F$ and vertices in $D$. A strike is given along each such edge, and each vertex in $D$ receives at most two strikes, so there are at least $d_3(u) + \frac{d^*_3(u)}{2} - (p - f)$ vertices with at least one strike in $D$. Each such vertex is colored after $u$ in our ordering, so at most

$$(\deg(u) - p) - (d_3(u) + \frac{d^*_3(u)}{2} - (p - f)) = \deg(u) - d_3(u) - \frac{d^*_3(u)}{2} - f$$

vertices in $D$ are colored before $u$ in our ordering. Each such vertex is potentially forced to choose a color in $L(u)$, so including the $f$ precolored vertices with a color choice already in $L(u)$, there are at least $8 - (\deg(u) - d_3(u) - \frac{d^*_3(u)}{2})$ available colors available for $u$ in our greedy coloring. If there is at least one available color, we will obtain a proper coloring of $G$. So if $d_3(u) + \frac{d^*_3(u)}{2} \geq \deg(u) - 7$, we can properly color $G$. \qed
3.2.2 Proof of Theorem 3.2.2

Proof of Theorem 3.2.2: Suppose that $G$ is a plane graph of minimum order + size such that $G$ is not $(3,10)$-choosable, and let $L$ be a $(3,10)$-list assignment of $G$ that does not admit a proper $L$-coloring. Since $G$ is a minimum counterexample, $G$ does not contain any reducible configurations. From Proposition 2.3.1, $G$ does not contain any vertices of degree $\leq 2$, any pair of adjacent vertices with degrees totaling $\leq 10$, any pair of adjacent vertices sharing a 3-face with degrees totaling 11, or any pair of adjacent vertices sharing two 3-faces with degrees totaling 12.

For each $v \in V(G)$ and $f \in F(G)$ define initial charge functions $\mu_0(v) = \deg(v) - 6$ and $\nu_0(f) = 2\ell(f) - 6$. By Euler’s formula, total charge is $-12$. Apply the following discharging rules:

(R1) (a) Each 3-vertex pulls charge 1 from adjacent vertices.

(b) Each 4-vertex pulls charge $\frac{1}{2}$ from adjacent vertices.

(c) Each 5-vertex pulls charge $\frac{1}{5}$ from adjacent vertices.

(R2) If $f$ is a $4^+$-face, then for each edge $uv$ incident to $f$ with $\deg(u) \leq 5$, $f$ sends charge $\frac{1}{2}$ to $v$.

(R3) Let $v$ be a vertex with negative charge after (R2). For each vertex $w$ adjacent to $v$ such that $w$ and $v$ share two neighbors in $D_3(v)$, $v$ pulls charge $\frac{1}{5}$ from $w$.

For $i \in \{1,2,3\}$, let $\mu_i(v)$ and $\nu_i(f)$ be the charge on each vertex $v$ and face $f$ immediately after applying (R$i$). We claim the final charge on all faces and vertices is nonnegative. Since the total charge sum was preserved during the discharging rules, this contradicts the negative charge sum from the initial charge values. Observe that no two $5^-$-vertices are incident, so each face $f$ is incident to $\leq \frac{\ell(f)}{2}$ such vertices. If $f$ is a 3-face, then $f$ loses no charge. If $f$ is a $4^+$-face, then $f$ loses charge at most 1 per incident $5^-$-vertex. We have $\frac{\ell(f)}{2} \leq 2\ell(f) - 6$, so $\nu_3(f) \geq 0$.

Each vertex that loses charge in (R3) is adjacent to a 3-vertex, so no $7^-$-vertices lose charge by (R3). Each $5^-$-vertex gains exactly enough charge through (R1) so that the final charge is
nonnegative. Suppose \( v \) is a \( 6^+ \)-vertex. If \( u \) is an adjacent 3-vertex such that \( uv \) is incident to two \( 4^+ \)-faces, then \( v \) loses charge 1 to \( u \) in (R1) but gains charge 1 back from each face via edge \( uv \) in (R2). If \( u \) is an adjacent 4-vertex or 5-vertex such that \( uv \) is incident at least one \( 4^+ \)-face, then \( v \) loses charge \( \leq \frac{1}{2} \) to \( u \) in (R1) but gains charge \( \geq \frac{1}{2} \) back via edge \( uv \) in (R2). So, it suffices to count the remaining cases.

If \( u \in D_k(v) \) for \( k = 3, 4, \) or 5, then \( v \) loses charge 1, \( \frac{1}{2} \), and \( \frac{1}{5} \) to \( u \) respectively in (R1); no charge is gained via \( uv \) in (R2). If \( u \in D_3^*(v) \), then \( v \) loses charge 1 to \( u \) in (R1) but gains \( \frac{1}{2} \) charge back via \( uv \) in (R2). So, \( \mu_2(v) = \mu_0(v) - d_3(v) - \frac{1}{2}d_3^*(v) - \frac{1}{5}d_5(v) - \frac{1}{5}d_5(v) \). Recall \( \mu_0(v) = \deg(v) - 6 \), so if \( \mu_2(v) < 0 \), then we must have \( \deg(v) - 6 < d_3(v) + \frac{1}{2}d_3^*(v) + \frac{1}{5}d_4(v) + \frac{1}{5}d_5(v) \).

Let \( D(v) = D_3(v) \cup D_4(v) \cup D_5(v) \cup D_3^*(v) \). For each 3-face \( uvw \) incident to \( v \), at most one of \( u, w \) is in \( D(v) \). If \( u \in D(v) \), we say \( u \) gives a strike to \( w \). Similarly, if \( w \in D(v) \), then \( w \) gives a strike to \( u \). Each vertex in \( D_3(v) \cup D_4(v) \cup D_5(v) \) contributes two strikes, and each vertex in \( D_3^*(v) \) contributes 1 strike. So, the total number of strikes is \( 2(d_3(v) + d_4(v) + d_5(v)) + d_3^*(v) \). Each vertex adjacent to \( v \) that is not in \( D \) can have at most two strikes, so
\[
d_3(v) + d_4(v) + d_5(v) + d_3^*(v) + \frac{1}{2}(2(d_3(v) + d_4(v) + d_5(v)) + d_3^*(v)) \leq \deg(v).
\]
Equivalently,
\[
2(d_3(v) + d_4(v) + d_5(v)) + \frac{2}{5}d_3^*(v) \leq \deg(v).
\]

We have two inequalities: \( d_3(v) + \frac{1}{2}d_3^*(v) + \frac{1}{2}d_4(v) + \frac{1}{5}d_5(v) > \deg(v) - 6 \) and \( 2(d_3(v) + d_4(v) + d_5(v)) + \frac{2}{5}d_3^*(v) \leq \deg(v) \). We also note that our reducible configurations justify the following: If \( v \) is a \( 9^- \)-vertex, then \( d_3(v) = 0 \). If \( v \) is an \( 8^- \)-vertex, then \( d_3^*(v) = d_4 = 0 \). If \( v \) is a \( 7^- \)-vertex, then \( d_5(v) = 0 \). It is also clear that \( d_3(v), d_3^*(v), d_4(v), d_5(v) \geq 0 \). With an integer program, we may verify that the only valid integer 5-tuples \((\deg(v), d_3(v), d_3^*(v), d_4(v), d_5(v))\) that satisfy the above inequalities are those in Table 3.1.

Table 3.1: Cases with negative charge after applying (R2).

<table>
<thead>
<tr>
<th>( \deg(v) )</th>
<th>( d_3 )</th>
<th>( d_3^* )</th>
<th>( d_4 )</th>
<th>( d_5 )</th>
<th>( \mu_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-1/2</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-1/2</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>-1/5</td>
</tr>
</tbody>
</table>
Now consider configurations C1 (Figure 3.1) and C2 (Figure 3.2) described by Lemmas 3.2.3 and 3.2.4. If \(|L(v)| \leq 9\), then in each case, the graph contains C1. So, suppose \(|L(v)| = 10\). Let \(u\) be a neighbor of \(v\) such that \(u\) and \(v\) share two neighbors in \(D_3(v)\). Configuration C2 ensures that \(|L(u)| \leq \deg(u) - 2\) and, in particular, \(\mu_0(u) \geq 0\). In (R3), \(u\) sends charge \(\frac{1}{5}\) to \(v\). Enough charge is sent to ensure \(\mu_3(v) \geq 0\) in all of the above cases. We are left to consider \(\mu_3(u)\).

Suppose \(\mu_3(u) < 0\). Let \(v_1, v_2, \ldots, v_{\deg(u)}\) be the neighbors of \(u\) in clockwise order, such that \(v = v_3\). Note that \(D_3(v) \cap (D_3(u) \cup D_3^*(u)) = \{v_2, v_4\}\), and \(uv_2v_4\) form a 4-cycle with chord \(uv\). If \(v_2 \in D_3(u)\), then \(v_1\) is adjacent to \(v\) and hence does not pull charge from \(u\) during (R3). If \(v_2 \in D_3^*(u)\), then \(v_2\) and \(v_1\) are not adjacent, so \(v_1\) does not pull charge from \(u\) during (R3). Similarly, \(v_5\) does not pull charge from \(u\) during (R3). It follows that at most \(\frac{1}{2}(d_3(u) + d_3^*(u))\) vertices pull charge from \(u\) during (R3). So, \(\mu_3(u) \geq \mu_2(u) - \frac{1}{10}(d_3(u) + d_3^*(u))\).

If we return to the integer program and weaken the charge inequality to allow 5-tuples that satisfy \(\mu_2(u) < \frac{1}{10}(d_3(u) + d_3^*(u))\), we have the solutions given in Table 3.2.

<table>
<thead>
<tr>
<th>case</th>
<th>(\deg(u))</th>
<th>(d_3)</th>
<th>(d_3^*)</th>
<th>(d_4)</th>
<th>(d_5)</th>
<th>(\mu_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>1/2</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-1/2</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-1/2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>case</th>
<th>(\deg(u))</th>
<th>(d_3)</th>
<th>(d_3^*)</th>
<th>(d_4)</th>
<th>(d_5)</th>
<th>(\mu_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>10</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>10</td>
<td>10</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>-1/10</td>
</tr>
<tr>
<td>13</td>
<td>11</td>
<td>11</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>11</td>
<td>11</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>11</td>
<td>11</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>12</td>
<td>12</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1/2</td>
</tr>
<tr>
<td>17</td>
<td>12</td>
<td>12</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

We can rule out most of these cases by considering reducible configurations. We know \(u\) is incident to a degree 3 vertex \(x\), wherein \(|L(u) \cap L(x)| \geq 1\) or else a proper \(L\)-coloring of \(G - ux\) extends to a proper \(L\)-coloring of \(G\), so \(|L(u)| + 3 = |L(u)| + |L(x)| = |L(u) \cup L(x)| + |L(u) \cap L(x)| \geq 11\) and \(|L(u)| \geq 8\). Recall \(|L(u)| \leq \deg(u) - 2\). This rules out case 1. We have already established \(\mu_2(u) \geq 0\), so cases 6, 7, 9, and 12 are ruled out. If \(\deg(u) \leq 10\) and
$|D_3(v) \cap D_3(u)| \geq 1$, then $|L(u)| \leq \deg(u) - 3$ or else the graph contains $C_2$. In cases 2, 5, 8, 10, and 11, there must be at least one such $v$ with $|D_3(v) \cap D_3(u)| \geq 1$, so these cases can be ruled out. If $\deg(u) \leq 11$ and $|D_3(v) \cap D_3(u)| = 2$, then $|L(u)| \leq \deg(u) - 4$ or else the graph contains $C_2$. This rules out case 15. So, we are left with the cases shown in Table 3.3.

Table 3.3: Remaining cases with negative charge after applying (R3).

| case | $\deg(u)$ | $d_3$ | $d_3^*$ | $d_4$ | $d_5$ | $\mu_2$ | $|L(u)|$ |
|------|-----------|-------|--------|-------|-------|--------|--------|
| 3    | 10        | 2     | 4      | 0     | 0     | 0      | 8      |
| 4    | 10        | 3     | 2      | 0     | 0     | 0      | 8      |
| 13   | 11        | 3     | 3      | 0     | 0     | $1/2$  | 8 or 9 |
| 14   | 11        | 4     | 2      | 0     | 0     | 0      | 8 or 9 |
| 16   | 12        | 5     | 1      | 0     | 0     | $1/2$  | 8, 9, or 10 |
| 17   | 12        | 6     | 0      | 0     | 0     | 0      | 8, 9, or 10 |

In Table 3.3, we include possible values of $|L(u)|$. The lower bound, $|L(u)| = 8$, is fixed. Lemma 3.2.4 establishes the upper bound. Lemma 3.2.5 ensures $|L(u)| \neq 8$ in our remaining cases. The only cases left are those we have admitted in the theorem. \qed
CHAPTER 4. PACKING COLORING ON INFINITE LATTICES

4.1 Introduction

The United States Federal Communications Commission placed rules and regulations on radio towers, among which is the requirement that towers that broadcast at the same frequencies must be far enough apart so that they do not interfere. The distance a broadcast signal reaches is related to its frequency, so the distance a pair of towers broadcasting the same frequency must be apart is related to the frequency of the broadcasts. These frequency assignment regulations have inspired the idea of broadcast coloring [18], which is now known as packing coloring. Radio towers, particularly cell phone towers, are typically placed in patterns that match the vertices of a lattice, commonly a hexagonal lattice, so lattices are of particular interest for packing coloring problems.

Another problem of interest is Question 4.1.1. Of the graphs studied, all planar subcubic graphs (i.e. planar graphs of maximum degree 3) have had low packing chromatic number. By contrast, planar graphs of maximum degree 4 with infinite packing chromatic number have been found; e.g. the infinite 4-regular tree [33].

**Question 4.1.1** (R. Škrekovski [14]). *Is there an upper bound for the packing chromatic number of all planar graphs with maximum degree at most 3?*

In this chapter, we introduce a variety of lattices and explore their packing colorings. Our goal is to find or improve the upper or lower bounds for their packing chromatic numbers.

For convenience, we let $\mathbb{Z}^2$ be the vertex set of each two-dimensional lattice. When we consider three dimensional lattices wherein one dimension has finite order $m$, such as those in Figure 1.12, the vertex set is denoted $\mathbb{Z}^2 \times \{1, 2, \ldots, m\}$. For each $1 \leq i \leq m$, the subgraph induced by $\mathbb{Z}^2 \times \{i\}$ is called a layer of the lattice.
We provide packing colorings for a lattice in the form of regular patterns. The vertex set of the lattice is assigned a partition which we call a *tiling*. Each part is called a *tile*, and we provide a color assignment for each tile.

### 4.1.1 Density on an Infinite Graph

In a partial packing of a graph, it is useful to have a notion of density, i.e. proportion of the vertices that have been assigned a color. For a finite graph, the definition is trivial. However, many challenges arise when attempting to define density for an infinite graph. For a graph $G$, we provide Definition 4.1.2 to describe the proportion $\text{dens}(S)$ of vertices that are covered by a set $S \subseteq V(G)$.

First we introduce the following notation: For a graph $G$, a vertex $v \in V(G)$, and an integer $k \geq 0$, we define $N_k(G, v) = \{w \in V(G) : \text{dist}(v, w) = k\}$ and $N_{\leq k}(G, v) = N_0(G, v) \cup N_1(G, v) \cup \cdots \cup N_k(G, v)$. When $G$ is clear from context, we instead use $N_k(v)$ and $N_{\leq k}(v)$.

**Definition 4.1.2.** Let $G$ be a connected graph such that all vertices in $G$ have finite degree, and let $v$ be a vertex in $G$. If

$$\lim_{k \to \infty} \frac{|N_{k+1}(v)|}{|N_{\leq k}(v)|} = 0,$$

then we define the *density* of a set $S \subseteq V(G)$ to be

$$\text{dens}(S) = \limsup_{k \to \infty} \frac{|S \cap N_{\leq k}(v)|}{|N_{\leq k}(v)|}.$$  

We must show that Definition 4.1.2 is well-defined. First notice that if all vertices in $G$ have finite degree, then $|N_{k+1}(v)|$ and $|N_{\leq k}(v)|$ are finite. We also have $0 \leq \frac{|S \cap N_{\leq k}(v)|}{|N_{\leq k}(v)|} \leq 1$, so $\text{dens}(S)$ is guaranteed to exist for our choice of $v$. We include Proposition 4.1.4 to show that if any vertex satisfies the limit condition, then all vertices satisfy that condition. We then provide Proposition 4.1.5 to show that $\text{dens}(S)$ does not depend on our choice of $v$. Lemma 4.1.3 is used in the proofs of the propositions.

Definition 4.1.2 describes the proportion of $V(G)$ that is covered by $S$. For a finite graph, this conforms to the expected relation that $\text{dens}(S) = \frac{|S|}{|V(G)|}$. For an infinite graph, we see that density can be approximated by choosing a vertex and finding density on larger and larger
subgraphs centered at that vertex. We use the limit superior for density rather than a limit, since otherwise we could construct $S$ in such a way that density fluctuates and is not well-defined. We could define separate notions of upper- and lower-density, but upper-density is more valuable to us since it provides us the inequality $\text{dens}(S_1 \cup S_2) \leq \text{dens}(S_1) + \text{dens}(S_2)$ for any $S_1, S_2 \subseteq V(G)$.

Suppose we have a partial packing of a graph $G$. That is, we have color classes $X_1, X_2, \ldots, X_k \subseteq V(G)$ such that for each $u, v \in X_i$, $\text{dist}(u, v) > i$. (It is sufficient but not necessary for our color classes to be disjoint.) We say the density of color $i$ is equal to $\text{dens}(X_i)$. For our partial packing to be a complete packing, we must necessarily have $\text{dens}(X_1 \cup X_2 \cup \cdots \cup X_k) = 1$. So, we must have $\text{dens}(X_1) + \text{dens}(X_2) + \cdots + \text{dens}(X_k) \geq 1$. So, it is valuable to find an upper bound for the density of each color class. We provide Theorem 4.1.6 to give a general upper bound for a particular color class. Corollary 4.1.7 provides additional specifications for this upper bound.

**Lemma 4.1.3.** Let $G$ be a connected graph such that all vertices in $G$ have finite degree, and let $u$ be a vertex in $G$. If $\lim_{k \to \infty} \frac{|N_{k+1}(u)|}{|N_{k}(u)|} = 0$, then for any positive integer $\ell$,

$$\lim_{k \to \infty} \frac{|N_{k+\ell}(v) \setminus N_{k}(v)|}{|N_{k}(v)|} = 0.$$ 

**Proof:** Suppose $\lim_{k \to \infty} \frac{|N_{k+1}(u)|}{|N_{k}(u)|} = 0$, and let $\ell$ be a positive integer. For $\epsilon > 0$, choose $\delta > 0$ such that $\delta \ell(\delta + 1)^\ell < \epsilon$, and let $n$ be a positive integer such that $k \geq n$ implies $\frac{|N_{k+1}(u)|}{|N_{k}(u)|} < \delta$. We have

$$\frac{|N_{k+\ell}(u) \setminus N_{k}(u)|}{|N_{k}(u)|} = \frac{|N_{k+\ell}(u)|}{|N_{k}(u)|} + \frac{|N_{k+\ell-1}(u)|}{|N_{k}(u)|} + \cdots + \frac{|N_{k+1}(u)|}{|N_{k}(u)|}$$

$$= \left( \frac{|N_{k+\ell}(u)|}{|N_{k+\ell-1}(u)|} \cdot \frac{|N_{k+\ell-1}(u)|}{|N_{k+\ell-2}(u)|} \cdots \frac{|N_{k+1}(u)|}{|N_{k}(u)|} \right) + \cdots + \frac{|N_{k+1}(u)|}{|N_{k}(u)|}$$

$$\leq \delta \left( \frac{|N_{k+\ell-1}(u)|}{|N_{k+\ell-2}(u)|} + 1 \right) \cdots \left( \frac{|N_{k+1}(u)|}{|N_{k}(u)|} + 1 \right) + \cdots + \delta$$

$$\leq \delta (\delta + 1)^{\ell-1} + \delta (\delta + 1)^{\ell-2} + \cdots + \delta$$

$$\leq \delta \ell(\delta + 1)^\ell < \epsilon.$$ 

So, $\lim_{k \to \infty} \frac{|N_{k+\ell}(u) \setminus N_{k}(u)|}{|N_{k}(u)|} = 0$. 

\qed
Proposition 4.1.4. Let $G$ be a connected graph such that all vertices in $G$ have finite degree, and let $u$ and $v$ be distinct vertices in $G$. If $\lim_{k \to \infty} \frac{|N_{k+1}(u)|}{|N_{\leq k}(v)|} = 0$, then $\lim_{k \to \infty} \frac{|N_{k+1}(v)|}{|N_{\leq k}(v)|} = 0$.

**Proof:** Let $d = \text{dist}(u, v)$. We have by the triangle inequality that for any $k > d$, $N_{k+1}(v) \subseteq N_{\leq k+d+1}(u) \setminus N_{\leq k-d}(u)$ and $N_{\leq k-d}(u) \subseteq N_{\leq k}(v)$. So, $\frac{|N_{k+1}(v)|}{|N_{\leq k}(v)|} \leq \frac{|N_{\leq k+d+1}(u) \setminus N_{\leq k-d}(u)|}{|N_{\leq k-d}(u)|}$. It follows by Lemma 4.1.3 that $\lim_{k \to \infty} \frac{|N_{k+1}(v)|}{|N_{\leq k}(v)|} = 0. \square$

Proposition 4.1.5. Let $G$ be a connected graph such that all vertices in $G$ have finite degree, and for any vertex $v \in V(G)$, $\lim_{k \to \infty} \frac{|N_{k+1}(v)|}{|N_{\leq k}(v)|} = 0$. Let $u$ and $v$ be distinct vertices in $G$ and $S \subseteq V(G)$. Then

$$\limsup_{k \to \infty} \frac{|S \cap N_{\leq k}(u)|}{|N_{\leq k}(u)|} = \limsup_{k \to \infty} \frac{|S \cap N_{\leq k}(v)|}{|N_{\leq k}(v)|}.$$  

**Proof:** Let $d = \text{dist}(u, v)$, $U_k = N_{\leq k}(u)$, and $V_k = N_{\leq k}(v)$. By Lemma 4.1.3, $\lim_{k \to \infty} \frac{|U_{k+d}|}{|U_k|} = \lim_{k \to \infty} \frac{|U_{k+d}|}{|U_k|} + 1 = 1$, and $\lim_{k \to \infty} \frac{|V_{k+d}|}{|V_k|} = \lim_{k \to \infty} 1 - \frac{|U_{k+d}|}{|U_k|} = 1$. We know $U_{k-d} \subseteq V_k \subseteq U_{k+d}$, so $1 \leq \liminf_{k \to \infty} \frac{|V_k|}{|U_k|} \leq \limsup_{k \to \infty} \frac{|V_k|}{|U_k|} \leq 1$ and $\lim_{k \to \infty} \frac{|V_k|}{|U_k|} = 1$. Consider the following:

$$\left| \frac{|S \cap U_k|}{|U_k|} - \frac{|S \cap V_k|}{|V_k|} \right| = \left| \frac{|S \cap U_k|}{|U_k|} - \frac{|S \cap V_k|}{|V_k|} \right| + \left| \frac{|S \cap U_k|}{|U_k|} - \frac{|S \cap V_k|}{|V_k|} \right| = \left| \frac{|S \cap V_k|}{|V_k|} \right| + \left| \frac{|S \cap U_k|}{|U_k|} \right| = \left| \frac{|S \cap V_k|}{|V_k|} \right| + \left| \frac{|S \cap U_k|}{|U_k|} \right| = \left| \frac{|S \cap V_k|}{|V_k|} \right| + \left| \frac{|S \cap U_k|}{|U_k|} \right|$$

We have $\lim_{k \to \infty} \left| \frac{|S \cap U_k|}{|U_k|} - \frac{|S \cap V_k|}{|V_k|} \right| = \lim_{k \to \infty} \left| \frac{|U_{k+d}|}{|U_k|} - \frac{|U_{k+d}|}{|V_k|} \right| + \left| \frac{|V_k|}{|U_k|} - 1 \right| = 1 - 1 + |1 - 1| = 0$, so $\limsup_{k \to \infty} \frac{|S \cap U_k|}{|U_k|} = \limsup_{k \to \infty} \frac{|S \cap V_k|}{|V_k|}. \square$

Theorem 4.1.6. Let $G$ be a graph that satisfies the conditions of Definition 4.1.2. Let $i$ be a positive integer, and let $X_i \subseteq V(G)$ such that $\text{dist}(u, v) > i$ for all distinct vertices $u, v \in X_i$. Then $\text{dens}(X_i) \leq \frac{1}{M}$, where $M = \min\{|N_{\leq i}(v)} : v \in V(G)\}$ if $i$ is even, and $M =
\[ \min\{ |N_{\leq \frac{i+1}{2}}(v)| + \frac{1}{s_v} |N_{\geq \frac{i+1}{2}}(v)| : v \in V(G) \} \text{ if } i \text{ is odd} \]
\[ s_v := \max \left\{ |S| : u \in N_{\leq \frac{i+1}{2}}(v), S \subseteq N_{\geq \frac{i+1}{2}}(u), \text{ and } w_1 \neq w_2 \in S \Rightarrow \text{dist}(w_1, w_2) > i \right\}. \]

**Proof:** If \( X_i = \emptyset \), then our result is trivial, so suppose \( X_i \) is nonempty. Let \( v \) be a vertex in \( G \) and \( k \) be sufficiently large so that \( X_i \cap N_k(v) \neq \emptyset \). We consider the subgraph of \( G \) induced by \( N_{\leq k}(v) \). Assign initial charge 1 to all vertices in \( N_k(v) \). Next, let each vertex send its charge to the nearest vertex in \( X_i \), splitting charge equally in the case of a tie. For any vertex \( x \in X_i \) such that \( N_{\frac{i+1}{2}}(x) \subseteq N_k(v) \), we claim that \( x \) has charge at least \( M \). Indeed, all vertices in \( N_{\frac{i+1}{2}}(x) \) have sent charge 1 to \( x \), and if \( i \) is odd, then all vertices in \( N_{\frac{i+1}{2}}(x) \) have sent charge at least \( \frac{1}{s_v} \) to \( x \).

Total charge remains constant, so \( |X_i \cap N_{\leq k-\frac{1}{4}}| \cdot M \leq |N_{\leq k}(v)| \). We have
\[ \frac{|X_i \cap N_{\leq k-\frac{1}{4}}|}{|N_{\leq k}(v)|} = \frac{|X_i \cap N_k(v)|}{|N_{\leq k}(v)|} - \frac{|X_i \cap (N_{\leq k}(v) \setminus N_{\leq k-\frac{1}{4}})|}{|N_{\leq k}(v)|}, \]
and by Lemma 4.1.3, \( \lim_{k \to \infty} \frac{|X_i \cap (N_{\leq k}(v) \setminus N_{\leq k-\frac{1}{4}})|}{|N_{\leq k}(v)|} = 0 \), so \( \limsup_{k \to \infty} \frac{|X_i \cap N_{\leq k}(v)|}{|N_{\leq k}(v)|} \leq \frac{1}{M} \).

**Corollary 4.1.7.** Let \( G \) be a graph that satisfies the conditions of Definition 4.1.2, \( i \) be a positive odd integer, \( v \) a vertex in \( G \), and \( s_v \) the value defined in Theorem 4.1.6. Then \( s_v \leq \max\{ \deg(u) : u \in N_{\frac{i+1}{2}}(v) \} \).

**Proof:** Suppose \( u \in N_{\frac{i+1}{2}}(v) \) and \( S \subseteq N_{\frac{i+1}{2}}(u) \) such that any pair of distinct vertices in \( S \) are at distance greater than \( i \) apart. Then each neighbor of \( u \) is within distance \( \frac{i-1}{2} \) of at most one vertex in \( S \), so \( u \) is within distance \( \frac{i+1}{2} \) of at most \( \deg(u) \) vertices in \( S \). It follows that \( |S| \leq \deg(u) \).

### 4.2 Hexagonal Lattice

Consider first the hexagonal lattice \( \mathcal{H} \) introduced in Section 1.3.4 and drawn in Figure 1.10. Definition 4.2.1 describes \( \mathcal{H} \) with vertex set \( \mathbb{Z}^2 \), and Figure 4.1 gives a drawing of \( \mathcal{H} \) with this vertex set.

**Definition 4.2.1.** The hexagonal lattice, denoted \( \mathcal{H} \), has vertex set \( \mathbb{Z}^2 \). Vertices \((x_1, y_1)\) and \((x_2, y_2)\) with \( x_1 + y_1 \leq x_2 + y_2 \) are adjacent if and only if one of the following conditions is satisfied.
Recall that $\chi_p(H) = 7$, as established in [14] and [22]. We provide Proposition 4.2.2 and Corollary 4.2.3 to determine distances between vertices on $H$, as well as Proposition 4.2.4 to determine the number of vertices at a given distance from a particular vertex. These results will be valuable when considering multi-layer versions of $H$.

**Proposition 4.2.2.** The distance between vertices $v = (0,0)$ and $w = (x,y)$ in $H$ is given by the following formula.

$$
\text{dist}(v, w) = \begin{cases} 
|x| + |y| & \text{if } |x| \geq |y|; \\
2|y| & \text{if } |x| < |y| \text{ and } x \equiv y \mod 2; \\
2|y| - 1 & \text{if } |x| < |y|, \ x \not\equiv y \mod 2, \ \text{and } y > 0; \\
2|y| + 1 & \text{if } |x| < |y|, \ x \not\equiv y \mod 2, \ \text{and } y < 0.
\end{cases}
$$

**Proof:** We introduce notation to designate a walk from $v$ in $H$. Consider Figure 4.1. At each vertex, we may move left, right, and up or down; we use the letters $\ell, r, u,$ and $d$ to designate these respective steps. A sequence of these letters designates a walk (e.g. $ururu$). A sequence is valid if and only if $u$ only occurs in odd positions and $d$ only occurs in even positions. A walk ends at vertex $(x,y)$ if $x$ is the number of $r$’s minus the number of $\ell$’s and $y$ is the number of $u$’s minus the number of $d$’s.

Any walk that ends at $w$ clearly needs $|x|$ instances of $r$ or $\ell$ and $y$ instances of $u$ or $d$, so $\text{dist}(v, w) \geq |x| + |y|$. Consider $|x| \geq |y|$. If $x, y \geq 0$, then a walk with $y$ instances of $ur$ followed
by \(x - y\) instances of \(r\) ends at \(w\). If \(x \geq 0\) and \(y < 0\), then a walk with \(|y|\) instances of \(rd\) followed by \(x - |y|\) instances of \(r\) ends at \(w\). Similar walks exist for \(x < 0\) by replacing \(r\) with \(\ell\). So in each case, \(\text{dist}(v, w) \leq |x| + |y|\) and we can conclude \(\text{dist}(v, w) = |x| + |y|\).

Now consider \(|x| < |y|\). Any walk to \(w\) must contain \(|y|\) instances of \(u\) if \(y > 0\) or \(|y|\) instances of \(d\) if \(|y| < 0\); each instance can only occur in odd or even positions respectively. Each walk must also contain a number of \(\ell\)'s and \(r\)'s of the same parity as \(x\). Suppose \(y > 0\). If \(x \equiv y \mod 2\), then a walk that consists of \(|y|\) instances of \(u\*,\) where each \(*\) is \(r\) or \(\ell\) with frequency \(\frac{y + x}{2}\) and \(\frac{y - x}{2}\) respectively will end at \(w\). If \(x \not\equiv y \mod 2\), then we may remove the last \(*\) from the previous walk and change the frequency of each \(r\) and \(\ell\) to \(\frac{y + x - 1}{2}\) and \(\frac{y - x - 1}{2}\), and the walk will end at \(w\). These are the shortest walks that can both contain the necessary number of \(u\)'s and the necessary parity of \(*\)'s, so \(\text{dist}(u, w) = 2|y|\) and \(2|y| - 1\) respectively.

If \(y < 0\) and \(x \equiv y \mod 2\), then a walk that consists of \(|y|\) instances of \(*d\), where each \(*\) is \(r\) or \(\ell\) with frequency \(\frac{y + x}{2}\) and \(\frac{y - x}{2}\) respectively will end at \(w\). If \(x \not\equiv y \mod 2\), then we append another \(*\) from the previous walk and change the frequency of each \(r\) and \(\ell\) to \(\frac{y + x + 1}{2}\) and \(\frac{y - x + 1}{2}\), and the walk will end at \(w\). Similar to above, these are the shortest walks that can both contain the necessary number of \(u\)'s and the necessary parity of \(*\)'s, so \(\text{dist}(u, w) = 2|y|\) and \(2|y| + 1\) respectively.

**Corollary 4.2.3.** Suppose \(v = (x_1, y_1)\) and \(w = (x_2, y_2)\) are vertices in \(H\). Let \(x = |x_2 - x_1|\) and \(y = |y_2 - y_1|\). Then

\[
\text{dist}(v, w) = \begin{cases} 
  x + y & \text{if } x \geq y; \\
  2y & \text{if } x < y \text{ and } x \equiv y \mod 2; \\
  2y - 1 & \text{if } x < y, \ x \not\equiv y \mod 2, \text{ and either } y_2 > y_1 \text{ and } x_1 \equiv y_1 \mod 2; \\
  & \text{or } y_2 < y_1 \text{ and } x_1 \not\equiv y_1 \mod 2; \\
  2y + 1 & \text{if } x < y, \ x \not\equiv y \mod 2, \text{ and either } y_2 < y_1 \text{ and } x_1 \equiv y_1 \mod 2; \\
  & \text{or } y_2 > y_1 \text{ and } x_1 \not\equiv y_1 \mod 2.
\end{cases}
\]

**Proof:** If \(x_1 \equiv y_1 \mod 2\), then the function \(f((x, y)) = (x - x_1, y - y_1)\) is an automorphism of \(H\), so \(\text{dist}(v, w) = \text{dist}((0, 0), (x_2 - x_1, y_2 - y_1))\). Otherwise, \(f((x, y)) = (x - x_1, y_1 - y)\) is
an automorphism of \( H \) and \( \operatorname{dist}(v, w) = \operatorname{dist}((0,0), (x_2 - x_1, y_1 - y_2)) \). The result follows from Proposition 4.2.2.

**Proposition 4.2.4.** Let \( v \in V(H) \) and \( k \) be a positive integer. The number of vertices of distance \( k \) from \( v \) is exactly \( 3k \).

**Proof:** Without loss of generality, suppose \( v = (0,0) \). Consider Proposition 4.2.2. We will count the number of vertices \( (x,y) \) that satisfy \( \operatorname{dist}(v, (x,y)) = k \). First suppose \( k \equiv 0 \mod 2 \). Then for each \( 0 \leq i \leq \frac{k}{2} \), each pair \( \pm(\frac{k}{2} + i), \pm(\frac{k}{2} - i) \) satisfies \( |x| \geq |y| \) and is at distance \( k \) from \( v \). For each \( i < \frac{k}{2} \), there are 4 such vertices, and for \( i = \frac{k}{2} \), there are two such vertices. So, there are \( 4(\frac{k}{2}) + 2 = 2k + 2 \) vertices at distance \( k \) from \( v \) satisfying \( |x| \geq |y| \). If \( |x| < |y| \), then we must have \( y = \pm \frac{k}{2}, -\frac{k}{2} < x < \frac{k}{2} \), and \( x \equiv \frac{k}{2} \mod 2 \). There are \( 2(\frac{k}{2} - 1) = k - 2 \) such vertices. So, there are \( 3k \) vertices at distance \( k \) from \( v \).

Now suppose \( k \equiv 1 \mod 2 \). We first count the number of vertices \( (x,y) \) satisfying \( |x| \geq |y| \) and \( |x| + |y| = k \). We must have \( x = \pm(\frac{k+1}{2} + i) \) and \( y = \pm(\frac{k-1}{2} - i) \) for some \( 0 \leq i \leq \frac{k-1}{2} \). For \( i < \frac{k-1}{2} \), there are 4 such vertices, and for \( i = \frac{k-1}{2} \), there are two such vertices. So, there are \( 4(\frac{k-1}{2}) + 2 = 2k \) vertices at distance \( k \) from \( v \) satisfying \( |x| \geq |y| \). Now suppose \( |x| < |y| \). We consider \( y > 0 \) and \( y < 0 \) separately. If \( y > 0 \), then \( y = \frac{k+1}{2} \) and we must have \( -\frac{k+1}{2} < x < \frac{k+1}{2} \) and \( x \neq \frac{k+1}{2} \mod 2 \). There are \( \frac{k+1}{2} \) such vertices. If \( y < 0 \), then \( y = \frac{k-1}{2} \) and we must have \( -\frac{k-1}{2} < x < \frac{k-1}{2} \) and \( x \neq \frac{k-1}{2} \mod 2 \). There are \( \frac{k-1}{2} \) such vertices. So, there are \( \frac{k+1}{2} + \frac{k-1}{2} = k \) vertices at distance \( k \) from \( v \) satisfying \( |x| < |y| \) and \( 3k \) total vertices at distance \( k \) from \( v \). \(\square\)

### 4.3 Truncated Square Lattice

The truncated square lattice, like the hexagonal lattice, is a planar 3-regular infinite graph. We consider it due to its relevance to Question 4.1.1. The origin of its name comes from geometry, where a truncation is applied to each vertex in the square lattice; this could be considered in graph theoretic terms as replacing each vertex in \( S \) with a 4-cycle. Figure 4.2 gives a typical representation of the lattice. We present a formal definition of the truncated square lattice with vertex set \( \mathbb{Z}^2 \), as shown in Figure 4.3.
Definition 4.3.1. The truncated square lattice, denoted $S_{tr}$, has vertex set $\mathbb{Z}^2$. Vertices $(x_1, y_1)$ and $(x_2, y_2)$ with $x_1 + y_1 \leq x_2 + y_2$ are adjacent if and only if one of the following conditions is satisfied.

- $y_1 = y_2$ and $x_2 - x_1 = 1$
- $x_1 = x_2, y_2 - y_1 = 1, y_1 \equiv 0 \mod 2$, and $x \equiv 0$ or $3 \mod 4$
- $x_1 = x_2, y_2 - y_1 = 1, y_1 \equiv 1 \mod 2$, and $x \equiv 1$ or $2 \mod 4$

Figure 4.2: The truncated square lattice.

Figure 4.3: The truncated square lattice represented on $\mathbb{Z}^2$.

In Theorem 4.3.2, we provide initial bounds for $\chi_p(S_{tr})$. Stronger bounds can be found by using a SAT-solver (see Section 4.7). In fact, a colleague [29] has found that $\chi_p(S_{tr}) = 7$. Our initial bounds were found by hand.

Theorem 4.3.2. $5 \leq \chi_p(S_{tr}) \leq 11$
Proof: For the upper bound, we provide an 11-packing of $S_{tr}$. Figure 4.4 provides a tiling of the square lattice into $4 \times 16$ tiles. Figure 4.5 provides a color assignment to each tile.

![Figure 4.4: The tiling pattern for the truncated square lattice.](image)

For the lower bound, it suffices to show that the subgraph $G$ shown in Figure 4.6 does not admit a 4-packing. We use the term inner and outer vertices to denote vertices respectively on and not on the inner 8-cycle.

![Figure 4.5: A colored tile in the truncated square lattice. Each tile is colored identically.](image)

For the lower bound, it suffices to show that the subgraph $G$ shown in Figure 4.6 does not admit a 4-packing. We use the term inner and outer vertices to denote vertices respectively on and not on the inner 8-cycle.

![Figure 4.6: A subgraph $G$ of $S_{tr}$.](image)

Suppose we have a 4-packing of $G$. A maximum packing of color 1 fills 8 vertices in one of two arrangements that are equivalent by symmetry. In the remaining graph, a maximal filling of color 4 can cover at most two outer vertices or one inner vertex ($2o/1i$). Colors 2 and 3 can each cover at most four outer vertices, two inner vertices, or two outer vertices and one inner
vertex \((4o/2i/2o1i)\). There is no combination of these options that fills both four outer vertices and four inner vertices.

We next consider when color 1 is assigned a maximal packing that does not fill 8 vertices. Notice that color 1 fills four outer vertices and at most three inner vertices. There are at least five inner vertices left to fill. In \(C_8\), colors 2 and 3 can fill at most two vertices each, and color 4 can fill at most one. So, the inner vertices must be filled with these maximum packings. However, color 2 can then fill at most two outer vertices, color 4 at most 1, and color 3 cannot fill any. So, not all outer vertices are filled and we may conclude that \(\chi_p(S_{tr}) \geq \chi_p(G) \geq 5\). 

**4.4 Two-layer Hexagonal Lattice**

The Two-Layer Hexagonal Lattice is the graph \(P_2 \square H\). Figure 1.12 gives its typical representation in two layers. We say \(V(P_2 \square H) = \mathbb{Z}^2 \times \{1, 2\}\), and vertices \((x_1, y_1, z_1)\) and \((x_2, y_2, z_2)\) are adjacent if and only if \(z_1 = z_2\) and \((x_1, y_1)\) and \((x_2, y_2)\) are adjacent in \(H\), or if \((x_1, y_1) = (x_2, y_2)\) and \(z_1 \neq z_2\).

We provide Proposition 4.4.1 to establish an upper bound for the density of a color class \(X_i\).

**Proposition 4.4.1.** Let \(i\) be a positive integer and \(X_i \subseteq V(P_2 \square H)\) such that every pair of distinct vertices \(u, v \in X_i\) satisfies \(\text{dist}(u, v) > i\). Let \(k = \lfloor \frac{i}{2} \rfloor\). Then

\[
\text{dens}(X_i) \leq \begin{cases} 
\frac{1}{2} & \text{if } i = 1; \\
\frac{1}{3k^2 + 2} & \text{if } i \equiv 0 \mod 2; \\
\frac{1}{3k^2 + 2k + 3} & \text{if } i > 1 \text{ and } i \equiv 1 \mod 2.
\end{cases}
\]

**Proof:** Let \(v = (x, y, z)\) be a vertex in \(P_2 \square H\). First notice that if \(k \geq 1\), then \(N_k(v) = N_k(H, (x, y)) + N_{k-1}(H, (x, y))\). So, \(|N_{\leq k}(v)| = |N_k(H, (x, y))| + 2|N_{k-1}(H, (x, y))| + \cdots + 2|N_0(H, (x, y))| = 3k + 2 + 2 \sum_{j=1}^{k-1} 3j = 3k + 2 + 3(k - 1)k = 3k^2 + 2\). We also have \(|N_{k+1}(v)| = 3(k + 1) + 3k = 6k + 3\). Consider Theorem 4.1.6. If \(i\) is even or \(i = 1\), our result follows from the theorem. If \(i > 1\) is odd, it suffices to show that the value \(s_v\) defined in the theorem is equal to 3.
For any vertex \( u \in V(P_2 \square \mathcal{H}) \), consider the set \( S \subseteq N_{i+1}(u) \) wherein any pair of distinct vertices \( w_1, w_2 \in S \) satisfy \( \text{dist}(w_1, w_2) > i \). Consider a set of shortest paths from \( u \) to the vertices in \( S \). These paths must be pairwise disjoint, so they must contain distinct neighbors of \( u \). If one contains the layer-crossing edge incident to \( u \), we may augment it to avoid that edge. So, there are at most \( \deg(u) - 1 = 3 \) distinct paths, and it follows that \( s_v = 3 \).

Our main goal is to find a packing of \( P_2 \square \mathcal{H} \). With an upper bound on density of a color class, we may determine an absolute lower bound, as well as an estimate, for the number of additional colors needed to complete a partial packing. To provide a more accurate estimate, we constructed individual color classes which we suspect to have the maximum possible density; these color classes are provided in Remark 4.4.2.

In practice, color classes interfere with each other, so Proposition 4.4.1 does not provide the expected density for a color class. In particular, it is likely that an optimal packing assigns a maximum packing to color 1; i.e. a packing where color 1 covers every other vertex. All vertices not assigned color 1 are then an even distance from each other, so each color \( 2i \) behaves like color \( 2i + 1 \). Remark 4.4.2 provides color classes for odd colors.

**Remark 4.4.2.** Let \( i \) be a positive odd integer, and let \( \ell = \lfloor \frac{i}{4} \rfloor \). Let \( X_i \subseteq V(P_2 \square \mathcal{H}) \) be defined as follows: If \( i \equiv 1 \mod 4 \), then let

\[
X_i = \{( (3\ell + 1)a - b, (\ell + 1)a + (4\ell + 1)b + (2\ell + 1)c, c + 1) : a, b \in \mathbb{Z}, c \in \{0, 1\} \}.
\]

If \( i \equiv 1 \mod 4 \), then let

\[
X_i = \{( (6\ell + 4)a + (3\ell + 2)c, (2\ell + 2)b + (\ell + 1)c, c + 1) : a, b \in \mathbb{Z}, c \in \{0, 1\} \}.
\]

Observe that for any distinct pair of vertices \( u, v \in X_i \), \( \text{dist}(u, v) > i \). So, \( X_i \) is a valid color class for color \( i \). Furthermore, \( \text{dens}(X_i) = \frac{1}{3k^2 + 3k + 2} \), where \( k = \frac{i - 1}{2} \).

### 4.5 Offset Two-Layer Hexagonal Lattice

Another variation for the two-layer hexagonal lattice, denoted \( \mathcal{H}_{\text{off}} \), has the layers offset from one-another, as shown in Figure 4.7. In this variation, only half of the vertices are
connected between layers. The vertex set is $\mathbb{Z}^2 \times \{1, 2\}$, and vertices $(x_1, y_1, z_1)$ and $(x_2, y_2, z_2)$ are adjacent if and only if one of the below conditions is satisfied.

- $z_1 = z_2 = 1$ and $(x_1, y_1)$ and $(x_2, y_2)$ are adjacent in $\mathcal{H}$.
- $z_1 = z_2 = 2$ and $(x_1 + 1, y_1)$ and $(x_2 + 1, y_2)$ are adjacent in $\mathcal{H}$.
- $(x_1, y_1) = (x_2, y_2)$, $z_1 \neq z_2$, and $x_1 + y_1 \equiv 1 \mod 2$.

![Figure 4.7](image)

Figure 4.7: A second variant of the two-layer hexagonal lattice. Large dots represent two vertices in different layers, joined by an edge.

One major difference between $\mathcal{H}_{\text{off}}$ and $P_2 \Box \mathcal{H}$ is that $\mathcal{H}_{\text{off}}$ permits two distinct color classes for color 1 with density $\frac{1}{2}$. These color classes are $X_1 = \{(x, y, z) \in V(\mathcal{H}_{\text{off}}) : x + y + z \equiv 0 \mod 2\}$ and $X'_1 = \{(x, y, z) \in V(\mathcal{H}_{\text{off}}) : x + y \equiv 0 \mod 2\}$. Color class $X_1$ is also viable in $P_2 \Box \mathcal{H}$. In both graphs, it causes any pair of remaining vertices to have even distance apart, causing conflict with even colors. We found that generating colorings starting with $X_1$ produced less dense color classes in $\mathcal{H}_{\text{off}}$ as opposed to $P_2 \Box \mathcal{H}$; it is possible that there is no finite packing of $\mathcal{H}_{\text{off}}$ that uses $X_1$. On the other hand, $X'_1$ permits higher density colorings for even colors. We suspect that with $X'_1$, $\mathcal{H}_{\text{off}}$ has a reasonably small packing coloring.

### 4.6 Generating Colorings

Rather than coloring an infinite graph directly, we tiled the graph and generated colorings of a tile. Each tile is a subgraph with wrap-around to simulate distances to vertices in other tiles. That is, in a tile of width $w$, adjacencies between vertices in columns $w$ and $w + 1$ of the original graph corresponded to adjacencies between vertices in columns $w$ and 1 in the tile.
Wraparound was similarly defined for rows. We also had the requirement that if a vertex was within distance $i$ of its copy in another tile, then we could not use color $i$ in the tile. So, tiles had to be sufficiently large to include large colors. We placed copies of a tile next to each other to construct larger tiles.

With wraparound, we could ensure that a packing of the tile could extend to a packing of the graph. We could similarly attempt to color a tile without wrap-around. A tile without wrap-around is a subgraph of the original graph, so if it does not admit a $k$-packing, then the original graph does not admit a $k$-packing. This helps us make conclusions about the lower bound of the packing chromatic number of the graph.

Whenever a coloring of a tile was generated (primarily with wrap-around), we cross-listed it with other generated colorings. The graphs we colored were highly symmetrical, so we could reduce the total number of colorings we needed to consider by only taking one representative per symmetry group. We also didn’t care what color a vertex was assigned when comparing colorings, only whether or not it was assigned a color, since only uncolored vertices are of concern when generating future color classes. Lastly, we only concern ourselves with maximal colorings, so if the set of colored vertices of one coloring was a subset of the colored vertices of another, the former coloring need not be considered.

We represented each tile as a multi-dimensional array of integers. Three dimensions sufficed; these corresponded to rows, columns, and layers. Each coordinate in the array corresponded to a vertex, and the integer assigned to that vertex was the assigned color. A vertex assigned value 0 was considered uncolored.

We used Java to generate colorings for various graph families. We defined various objects in Java: We had an abstract graph object and child objects for each graph family. It was useful to have a vertex object that stored the coordinates of a vertex; with its own object, we could more easily iterate on sets of vertices and store extra data in the vertices themselves. We also had a symmetry object to help us iterate on the symmetries of a graph. We had various objects for coloring tiles, notably including graphColorer, graphColorerRandom, and graphColorerLocalRandom. We created many files to store colorings, so we had a fileReader
object for reading these files. We also had various classes to run sophisticated coloring experiments.

Three algorithms were used: An exhaustive backtracking algorithm, a random coloring algorithm, and a priority-based random coloring algorithm.

### 4.6.1 Backtracking

Backtracking is a general algorithm for considering all solutions for a problem. The algorithm generates partial solutions, one piece at a time, and backtracks when it discovers the partial solution is not valid. We used backtracking to exhaustively construct all colorings for a particular range of colors and set of uncolored vertices in a tile. Algorithm 1 is pseudo-code for this algorithm. The algorithm itself is recursive; starting from the first vertex, it picks a valid color and moves to the next. This continues until it considers the last vertex, wherein it checks the coloring. That is, it checks whether or not the coloring is new and maximal versus all previously generated colorings. If so, it will store the new coloring. The algorithm then backtracks and considers a different coloring.

**Data:** A list of vertices \( \{v_i\}_{1 \leq i \leq n} \), and a list of colors \( \{c_j\}_{1 \leq j \leq k} \).

**Function** \( \text{ColorNext}(v_i) \)

\[
\begin{align*}
\text{for } & 1 \leq j \leq k \text{ do} \\
& \quad \text{if } \text{Vertex } v_i \text{ can be assigned color } c_j \text{ then} \\
& \quad \quad \text{Assign color } c_j \text{ to vertex } v_i. \\
& \quad \quad \text{if } i < n \text{ then} \\
& \quad \quad \quad \text{colorNext}(v_{i+1}) \\
& \quad \quad \quad \text{else} \\
& \quad \quad \quad \text{Check coloring} \\
& \quad \text{Assign no color to } v_i \\
& \text{if } i < n \text{ then} \\
& \quad \text{ColorNext}(v_{i+1}) \\
& \text{else} \\
& \quad \text{Check coloring}
\end{align*}
\]

**Algorithm 1:** A recursive function for generating all colorings of a graph. The function is initialized with \( \text{ColorNext}(v_1) \).

Backtracking allows us to consider all colorings of a tile. It’s useful for generating color classes for the smaller colors, but for larger tiles and colors, it is infeasible. If we have a range of
$k$ colors ($k + 1$ including the no-color option), and we have $n$ vertices to consider, backtracking takes at most $O((k + 1)^n)$ time. In practice, it usually takes much less time.

### 4.6.2 Random Coloring

A second approach to assigning a color class is to randomly assign vertices until no further assignment colors may be assigned. In this way, the color class is maximal. We provide Algorithm 2 to describe this approach.

**Data:** A list of vertices $\{v_i\}_{1 \leq i \leq n}$, and a color $c$.

**while** The list of vertices is nonempty **do**

| Choose a random vertex $v$ from the list **Assign color** $c$ to vertex $v$ **for** Vertices $u$ in the list **do**
| if Vertex $u$ is within distance $c$ of $v$ **then**
| Remove $u$ from the list

**Check coloring**

**Algorithm 2:** A random coloring algorithm for a graph.

The main weakness to this approach is that the colorings it produces are rarely optimal. To fit as many vertices as possible in a color class, it is typically necessary to pick them as close to each other as possible. This algorithm does not place any priority on picking vertices that are close to each other, so it has to be run many times to increase the chances of producing a good coloring. We found that running this algorithm for a long time was sufficient to find quality colorings, but only when we kept the dimensions of the tiles small.

### 4.6.3 Priority-Based Random Coloring

To increase the quality of our random coloring algorithm, we used a system that placed priority on vertices close to already-chosen vertices. We added tokens to each vertex to designate their priority, and when a new vertex was to be chosen, we would only consider those with maximum priority. Algorithm 3 demonstrates this method.

Notice that after Algorithm 3 assigns color $c$ to vertex $v$, it assigns tokens to vertices at distance between $c$ and $2c$ of $v$, wherein closer vertices receive more tokens. The idea is that we want our next vertex $u$ to be chosen such that $u$ causes relatively few vertices to be removed from $L$. Vertices in $N_{\leq c}(v)$ have already been removed, so we attempt to maximize
**Data:** A list $L = \{v_i\}_{1 \leq i \leq n}$ of vertices, and a color $c$.

Initially set 0 tokens on each vertex in $L$

Create a list $P = \emptyset$ of vertices with priority

```plaintext
while $L \neq \emptyset$ do
  if $P \neq \emptyset$ then
    Randomly choose a vertex $v$ from $P$
  else
    Randomly choose a vertex $v$ from $L$

Assign color $c$ to vertex $v$

Set $P = \emptyset$

for Vertices $u$ in $L$ do
  if Vertex $u$ is within distance $c$ of $v$ then
    Remove $u$ from the $L$
  else if Vertex $u$ is within distance $2c$ of $v$ then
    Add $2c - \text{dist}(u,v)$ tokens to $u$
    if $u$ has more tokens than those in $P$ (or $P$ is empty) then
      Set $P = \{u\}$
    else if $u$ has an equal amount of tokens as those in $P$ then
      Add $u$ to $P$
  Check coloring
```

**Algorithm 3:** A random coloring algorithm that places priority on vertices close to already-chosen vertices.

$N_{\leq c}(v) \cap N_{\leq c}(u)$ by minimizing $\text{dist}(u,v)$. Vertices are ranked by their distance from $v$, and tokens are assigned by their rank. If $u$ is at distance $\geq 2c$ from $v$, then $N_{\leq c}(v) \cap N_{\leq c}(u) = \emptyset$, so we have a cutoff wherein no tokens are assigned to vertices at distance $\geq 2c$ from $v$. Once tokens have been assigned, only vertices with the maximum number of tokens (i.e. maximum rank) are considered for the next vertex chosen.

A vertex may receive tokens from multiple previously assigned vertices. The goal remains the same that a newly chosen vertex $u$ should cause relatively few vertices to be removed from $L$, so $u$ should still be close to previously assigned vertices. Our strategy for assigning tokens prioritizes vertices that are close to many previously chosen vertices, but it isn’t necessarily clear that ranking by simply taking the sum of tokens from chosen vertices is optimal. Depending on the graph, there may be better ways to rank vertices. However, in practice Algorithm 3 is fast and produces good results.
4.6.4 Checking Colorings

The main difficulty in finding a packing of a graph is determining what constitutes a good color class. The simplest approach is to generate color classes one-at-a-time and only carry over the colorings with the highest density. In practice however, color classes with lower density may allow higher density for later color classes.

For the graphs we considered, optimal density of color classes dropped off dramatically after the first few colors. So, it was vital to have high combined density for lower colors. Algorithm 3 wouldn’t necessarily consider the highest combined density colorings, so we only used it to generate color classes for higher-value colors. We primarily used Algorithm 1 for the lower-value colors. We also ran various experiments of generating color classes individually and merging them.

In each of Algorithms 2, and 3; we had to check a coloring and decide whether or not to store and carry it over to the next set of colors. The first property to check was density. We generated multiple colorings for a color class and kept track of the maximum discovered density. We had a threshold that the density for each particular color class had to be at least 95% of maximum. If a newly generated color class didn’t meet this threshold, we would discard it. If the maximum density increased, we would update our list of graphs and discard those that didn’t meet the threshold.

After considering density, we placed a limit on the number of colorings we would carry over to the next color. If we already had more than a particular number of colorings, we would only store more if they had higher density than previously generated colorings. This limit scaled with the dimensions of the tiles, providing a rough limit on file size for the files that stored colorings. More precisely, we allowed at most \( \frac{1,600,000}{\text{width} \times \text{height}} \) colorings, or at most two if dimensions were sufficiently large. We also allowed at most half of our limit to contain colorings at density strictly below the maximum. This resulted in a rough file size of 8Mb for our stored colorings. With a limit on number of colorings, we could spend less time checking colorings that met our maximum density and more on generating colorings that may improve it. In addition, we had a manageable number of colorings to carry over and use as base cases for higher colors.
The last property we would consider for a coloring was whether or not it was a repeat. If the set of uncolored vertices was equivalent to that of another coloring by some symmetry, we would discard the coloring.

4.6.5 Choosing Dimensions

Before generating colorings, we must decide on the dimensions of the tile. A smaller tile can be colored more quickly, so its optimal coloring can be found quickly with much higher probability. However, larger tiles may allow for better colorings. For the initial colors, we generated colorings for graphs with various dimensions and chose those with the highest density to continue coloring. For later colors, we generated new color classes on the original tiles as well as tiles formed by placing copies of the original tiles next to each other to form larger tiles. We generated colorings on these tiles with various dimensions for a short amount of time each, chose the dimensions that had the highest density colorings, and ran our coloring algorithms for longer on those dimensions.

4.7 SAT Solvers

A boolean expression is a composition of boolean variables, each of which can be TRUE or FALSE, with operators and parentheses on those variables. Many operators exist, but it suffices to use the operators AND (\(\land\)), OR (\(\lor\)), and NOT (\(\neg\)). A boolean satisfiability problem (SAT) is the problem of determining whether or not a boolean expression has an assignment that renders the expression TRUE. Satisfiability problems are fundamental to computer science, so SAT-solvers have received a large amount of attention. Many good SAT-solvers exist, so if a mathematical problem can be represented as a reasonably-sized satisfiability problem, the problem can be run on a SAT-solver to produce results.

B. Martin, Raimondi, Chen, and J. Martin introduced the idea of using a SAT-solver to approach packing coloring problems [28]. For input in a SAT-solver, they represent a satisfiability problem as a list of clauses which are combined with the AND operator. In each clause, there is a list of variables with the NOT operator allowed only on the variables themselves, and the variables are combined with the OR operator.
The satisfiability problem asked is whether or not there exists a $k$-packing of a tile with particular dimensions. For each vertex $v$ and color $i$, there is a boolean variable $x_{v,i}$ wherein $x_{v,i}$ is TRUE if and only if $v$ is assigned color $i$. The clauses in the satisfiability problem are of two types. The first type is $x_{v,1} \lor x_{v,2} \lor \cdots \lor x_{v,k}$, wherein we must have that $v$ is assigned at least one color. It is not necessary for $v$ to be assigned at most one color, since with multiple colors, we could choose any available color for a particular packing. There is a clause of this type for every vertex. The second type of clause is $x_{u,i} \lor x_{v,i}$ for particular vertices $u$ and $v$. There is a clause of this type for every $u, v$ and $i$ that satisfy $\text{dist}(u, v) \leq i$.

4.8 Results

The main graph we colored was $P_2 \square H$. Böhm, Lánský, and Lidický had initially provided the bound that $\chi_p(P_2 \square H) \leq 526$, and our main goal was improving that bound. Our result is given in Theorem 4.8.1. The result was found by beginning with a $48 \times 48$ tile that had colors 1 through 15 already filled. We then applied our priority-based random coloring algorithm for the remaining colors, one color at a time. At every other color, we generated colorings on tiles with various multiples of the original tile’s dimensions, and carried over the colorings with the highest density. We stopped allowing the dimensions of the tile to increase when it reached a width by height of $1152 \times 1536$ at color 90. The tiling is provided at [31].

Theorem 4.8.1. $\chi_p(P_2 \square H) \leq 205$

A 205-packing was found on the fifth attempt at coloring $P_2 \square H$. Tables 4.1 and 4.2 summarize the densities of colors in these attempts. Notice that the main discrepancy was in colors 4 and 5. A higher density at these colors allowed for our result. We suspect that with further improvements, we may be able to lower the upper bound even further.

In addition to coloring $P_2 \square H$, we ran our algorithms on $H_{\text{off}}$. While we didn’t find a finite packing, we found our algorithms to be effective at quickly assigning adequate color classes for higher colors. We are confident that with more time and using an alternate color class for color 1, we will be able to find a finite packing of $H_{\text{off}}$. 
<table>
<thead>
<tr>
<th>Color</th>
<th>Attempt 1</th>
<th>Attempt 2</th>
<th>Attempt 3</th>
<th>Attempt 4</th>
<th>Attempt 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.00162753</td>
<td>0.00144676</td>
<td>0.00303819</td>
<td>0.00520833</td>
<td>0.00126140</td>
</tr>
<tr>
<td>0.0125</td>
<td>0.00162753</td>
<td>0.00144676</td>
<td>0.00303819</td>
<td>0.00520833</td>
<td>0.00126140</td>
</tr>
<tr>
<td>0.025</td>
<td>0.00162753</td>
<td>0.00144676</td>
<td>0.00303819</td>
<td>0.00520833</td>
<td>0.00126140</td>
</tr>
<tr>
<td>0.03125</td>
<td>0.00162753</td>
<td>0.00144676</td>
<td>0.00303819</td>
<td>0.00520833</td>
<td>0.00126140</td>
</tr>
<tr>
<td>0.0625</td>
<td>0.00162753</td>
<td>0.00144676</td>
<td>0.00303819</td>
<td>0.00520833</td>
<td>0.00126140</td>
</tr>
<tr>
<td>0.125</td>
<td>0.00162753</td>
<td>0.00144676</td>
<td>0.00303819</td>
<td>0.00520833</td>
<td>0.00126140</td>
</tr>
<tr>
<td>0.25</td>
<td>0.00162753</td>
<td>0.00144676</td>
<td>0.00303819</td>
<td>0.00520833</td>
<td>0.00126140</td>
</tr>
<tr>
<td>0.5</td>
<td>0.00162753</td>
<td>0.00144676</td>
<td>0.00303819</td>
<td>0.00520833</td>
<td>0.00126140</td>
</tr>
</tbody>
</table>

Table 4.1: Density of color $i$ for $1 \leq i \leq 80$ in five attempts at packing $P_2 \sqcap H$. Attempt 5 resulted in a 208-packing.
Table 4.2: Density of colors 1 through 80 in five attempts at packing $P_2 \cap H$.

<table>
<thead>
<tr>
<th>Attempt</th>
<th>Density</th>
<th>Attempt</th>
<th>Density</th>
<th>Attempt</th>
<th>Density</th>
<th>Attempt</th>
<th>Density</th>
<th>Attempt</th>
<th>Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.969</td>
<td>0.75</td>
<td>0.87</td>
<td>0.888</td>
<td>0.906</td>
<td>0.888</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.871</td>
<td>0.902</td>
<td>0.871</td>
<td>0.888</td>
<td>0.906</td>
<td>0.888</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.925</td>
<td>0.910</td>
<td>0.925</td>
<td>0.932</td>
<td>0.910</td>
<td>0.932</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.858</td>
<td>0.858</td>
<td>0.858</td>
<td>0.858</td>
<td>0.858</td>
<td>0.858</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0.876</td>
<td>0.876</td>
<td>0.876</td>
<td>0.876</td>
<td>0.876</td>
<td>0.876</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: The table continues with more data points for each attempt, but the pattern remains consistent.

53
CHAPTER 5. CONCLUSION

In this dissertation, we introduced choosability with union separation and explored a variety of problems that arise naturally from its extension from choosability and its relation to intersection choosability. In particular, we showed that all planar graphs are \((3, 11)\)- and \((4, 9)\)-choosable. We also explored the challenges that arise when using a discharging argument to consider \((3, 10)\)-choosability of planar graphs. We believe discharging will be sufficient to prove \((3, 10)\)-choosability, though larger reducible configurations will likely need to be considered. Further problems regarding union choosability include finding the smallest \(k\) such that all planar graphs are \((3, k)\)-choosable or the smallest \(k\) such that all planar graphs are \((4, k)\)-choosable. We also asked whether or not there is a function \(f(k, t)\) such that \((k, k + t)\)-choosability of a graph implies \((k, k - t)\)-choosability.

We also explored packing coloring and packing chromatic numbers. We surveyed the hexagonal lattice and multi-layered versions of it. We applied a variety of algorithms to attempt to find an upper bound for the packing chromatic number of two different two-layer hexagonal lattices. With one interpretation, we improved the upper bound on the packing chromatic number from 526 to 205. We suspect that the packing chromatic number is lower with the other interpretation, but further work is required. For both versions, we believe that exploring the lower bound to the packing chromatic number will lead to more promising results.

We also directed attention towards the problem of finding 3-regular planar graphs with large packing chromatic numbers. In doing so, we explored the truncated square lattice and found its packing chromatic number to be the same as that of the hexagonal lattice. For future work, we wish to consider other 3-regular subgraphs of the square lattice, or more pertinently, 3-regular graphs obtained by performing augmentations to other graphs. The overarching question here is whether there exists an upper bound to the packing chromatic number of 3-regular planar
graphs, or there exist such graphs with arbitrarily high and possibly infinite packing chromatic number.
APPENDIX A. (3,11)-CHOOSABILITY

List of Tuples

The following list of tuples \((d_3, d_3^*, d_4, d_5)\) satisfy inequality \((2.3)\). Recall that \(d(v) \geq 6\), \(d_5 > 0\) implies \(d(v) \geq 9\), \(d_3^* + d_4 > 0\) implies \(d(v) \geq 10\), and \(d_3 > 0\) implies \(d(v) \geq 11\). After these implications are applied, we find that the tuple \((d_3, d_3^*, d_4, d_5, d(v))\) violates inequality \((2.1)\).

<table>
<thead>
<tr>
<th>Tuple</th>
<th>Inequality Violated</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0, 0, 0, 0)</td>
<td>for (d(v) \geq 6)</td>
</tr>
<tr>
<td>(0, 0, 0, 1)</td>
<td>for (d(v) \geq 9)</td>
</tr>
<tr>
<td>(0, 0, 0, 2)</td>
<td>for (d(v) \geq 9)</td>
</tr>
<tr>
<td>(0, 0, 0, 3)</td>
<td>for (d(v) \geq 9)</td>
</tr>
<tr>
<td>(0, 0, 1, 0)</td>
<td>for (d(v) \geq 10)</td>
</tr>
<tr>
<td>(0, 0, 1, 1)</td>
<td>for (d(v) \geq 10)</td>
</tr>
<tr>
<td>(0, 0, 1, 2)</td>
<td>for (d(v) \geq 10)</td>
</tr>
<tr>
<td>(0, 0, 2, 0)</td>
<td>for (d(v) \geq 10)</td>
</tr>
<tr>
<td>(0, 0, 2, 1)</td>
<td>for (d(v) \geq 10)</td>
</tr>
<tr>
<td>(0, 0, 3, 0)</td>
<td>for (d(v) \geq 10)</td>
</tr>
<tr>
<td>(0, 0, 3, 1)</td>
<td>for (d(v) \geq 10)</td>
</tr>
<tr>
<td>(0, 1, 0, 0)</td>
<td>for (d(v) \geq 10)</td>
</tr>
<tr>
<td>(0, 1, 0, 1)</td>
<td>for (d(v) \geq 10)</td>
</tr>
<tr>
<td>(0, 1, 0, 2)</td>
<td>for (d(v) \geq 10)</td>
</tr>
<tr>
<td>(0, 1, 1, 0)</td>
<td>for (d(v) \geq 10)</td>
</tr>
<tr>
<td>(0, 1, 1, 1)</td>
<td>for (d(v) \geq 10)</td>
</tr>
<tr>
<td>(0, 1, 2, 0)</td>
<td>for (d(v) \geq 10)</td>
</tr>
<tr>
<td>(0, 1, 2, 1)</td>
<td>for (d(v) \geq 10)</td>
</tr>
<tr>
<td>(0, 1, 3, 0)</td>
<td>for (d(v) \geq 10)</td>
</tr>
<tr>
<td>(0, 2, 0, 0)</td>
<td>for (d(v) \geq 10)</td>
</tr>
<tr>
<td>(0, 2, 0, 1)</td>
<td>for (d(v) \geq 10)</td>
</tr>
<tr>
<td>(0, 2, 1, 0)</td>
<td>for (d(v) \geq 10)</td>
</tr>
<tr>
<td>(0, 2, 1, 1)</td>
<td>for (d(v) \geq 10)</td>
</tr>
<tr>
<td>(0, 2, 2, 0)</td>
<td>for (d(v) \geq 10)</td>
</tr>
<tr>
<td>(0, 2, 2, 1)</td>
<td>for (d(v) \geq 10)</td>
</tr>
<tr>
<td>(1, 0, 0, 0)</td>
<td>for (d(v) \geq 11)</td>
</tr>
<tr>
<td>(1, 0, 0, 1)</td>
<td>for (d(v) \geq 11)</td>
</tr>
<tr>
<td>(1, 0, 0, 2)</td>
<td>for (d(v) \geq 11)</td>
</tr>
<tr>
<td>(1, 0, 1, 0)</td>
<td>for (d(v) \geq 11)</td>
</tr>
<tr>
<td>(1, 0, 1, 1)</td>
<td>for (d(v) \geq 11)</td>
</tr>
<tr>
<td>(1, 0, 2, 0)</td>
<td>for (d(v) \geq 11)</td>
</tr>
<tr>
<td>(1, 0, 2, 1)</td>
<td>for (d(v) \geq 11)</td>
</tr>
<tr>
<td>(1, 0, 3, 0)</td>
<td>for (d(v) \geq 11)</td>
</tr>
<tr>
<td>(1, 1, 0, 0)</td>
<td>for (d(v) \geq 11)</td>
</tr>
<tr>
<td>(1, 1, 0, 1)</td>
<td>for (d(v) \geq 11)</td>
</tr>
<tr>
<td>(1, 1, 0, 2)</td>
<td>for (d(v) \geq 11)</td>
</tr>
<tr>
<td>(1, 1, 1, 0)</td>
<td>for (d(v) \geq 11)</td>
</tr>
<tr>
<td>(1, 1, 1, 1)</td>
<td>for (d(v) \geq 11)</td>
</tr>
<tr>
<td>(1, 1, 2, 0)</td>
<td>for (d(v) \geq 11)</td>
</tr>
<tr>
<td>(1, 1, 2, 1)</td>
<td>for (d(v) \geq 11)</td>
</tr>
</tbody>
</table>
(2, 1, 1, 0) fails (2.1) for \(d(v) \geq 11\).  
(2, 2, 0, 0) fails (2.1) for \(d(v) \geq 11\).  
(2, 2, 0, 1) fails (2.1) for \(d(v) \geq 11\).  
(2, 2, 1, 0) fails (2.1) for \(d(v) \geq 11\).  
(2, 3, 0, 0) fails (2.1) for \(d(v) \geq 11\).  
(3, 0, 0, 0) fails (2.1) for \(d(v) \geq 11\).  
(3, 0, 0, 1) fails (2.1) for \(d(v) \geq 11\).  
(3, 0, 1, 0) fails (2.1) for \(d(v) \geq 11\).  
(3, 1, 0, 0) fails (2.1) for \(d(v) \geq 11\).  
(3, 1, 0, 1) fails (2.1) for \(d(v) \geq 11\).  
(3, 1, 1, 0) fails (2.1) for \(d(v) \geq 11\).  
(3, 1, 1, 1) fails (2.1) for \(d(v) \geq 11\).  
(3, 2, 0, 0) fails (2.1) for \(d(v) \geq 11\).  
(4, 0, 0, 0) fails (2.1) for \(d(v) \geq 11\).  
(4, 0, 0, 1) fails (2.1) for \(d(v) \geq 11\).  
(4, 0, 1, 0) fails (2.1) for \(d(v) \geq 11\).  
(4, 1, 0, 0) fails (2.1) for \(d(v) \geq 11\).  
(4, 1, 0, 1) fails (2.1) for \(d(v) \geq 11\).  
(5, 0, 0, 0) fails (2.1) for \(d(v) \geq 11\).
APPENDIX B. SOURCE CODES

The following source codes were developed in Java to generate packing colorings.

B.1 Objects

B.1.1 Graph

The graph object is abstract and sets vertices to be coordinates in a 3-dimensional integer array. Child classes primarily need to define distances between vertices, symmetries of the graph, and the largest -value color that can fit in the graph (in case it is part of a tiling).

```java
package GraphPacking;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;

/**
 * An abstract graph
 * @author Kevin Moss
 */
public abstract class graph {
    protected int[][][] array;
    protected int width, height, layers;
    graph[][][] distanceGraphs;
    /**
     * Performs modular arithmetic. Ensures output is not negative.
     * @param number integer a in a%b
     * @param base integer b in a%b
     * @return number%base within range [0,base)
     */
    protected int mod(int number, int base){
        int temp = number % base;
        temp = (temp < 0) ? temp+base : temp;
        return temp;
    }
    /**
     * Creates a 3D integer array corresponding to the vertices of a graph
     * @param width Width of the array
     * @param height Height of the array
     * @param layers number of layers in the array
     */
```
```java
public graph(int width, int height, int layers){
    array = new int[height][width][layers];
    this.width = width;
    this.height = height;
    this.layers = layers;
}
/**
 * Creates a 3D integer array corresponding to the vertices of a graph.
 * @param array A 3D integer array.
 */
public graph(int[]][[] array){
    this.array = array;
    width = array[0].length;
    height = array.length;
    layers = array[0][0].length;
}
/**
 * Creates a 3D integer array corresponding to the vertices of a graph.
 * This constructor is useful for changing the graph type.
 * @param g
 */
public graph(graph g){
    array = g.getArray();
    width = g.getWidth();
    height = g.getHeight();
    layers = g.getLayers();
}
/**
 * Distance between any pair of vertices
 * @param v1 A vertex
 * @param v2 A vertex
 * @return Distance between vertex v1 and vertex v2
 */
public abstract int distance(vertex v1, vertex v2){
    return (distance(v1,v2,width,height,layers));
}
/**
 * Distance between any pair of vertices on a graph of the given size.
 * Useful for graphType. Otherwise use the shorter distance method.
 * @param v1 A vertex
 * @param v2 A vertex
 * @return Distance between vertex v1 and vertex v2
 */
public abstract int distance(vertex v1, vertex v2, int width, int height, int layers);
/**
 * Returns width of the array
 * @return width of the array
 */
public int getWidth(){
    return width;
}
/**
 * Returns height of the array
 * @return height of the array
 */
public int getHeight(){
```
public int getLayers() {
    return layers;
}

/**
  * Returns number of layers in the array
  * @return number of layers in the array
  */
public int[][][] getArray() {
    return array;
}

/**
  * Returns a duplicate of the array. Changing the duplicate will not change the
  * original.
  * @return A duplicate of the array.
  */
public int[][][] getArrayClone() {
    int[][][] arrayCopy = new int[height][width][layers];
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            for (int k = 0; k < layers; k++) {
                arrayCopy[i][j][k] = array[i][j][k];
            }
        }
    }
    return arrayCopy;
}

public int getNumVertices() {
    return width * height * layers;
}

/**
  * Sets a value in the array
  * @param xPos horizontal position from the left
  * @param yPos vertical position from the top
  * @param zPos layer number
  * @param color The value to be set
  */
public void setArrayVal(int xPos, int yPos, int zPos, int color) {
    array[yPos][xPos][zPos] = color;
}

/**
  * Sets a value in the array
  * @param v vertex to color
  * @param color The value to be set
  */
public void setArrayVal(vertex v, int color) {
    array[v.getY()][v.getX()][v.getZ()] = color;
}

/**
  * Gets a value in the array
  * @param xPos horizontal position from the left
  * @param yPos vertical position from the top
  * @param color The value to be set
  */
public void setArrayVal(int xPos, int yPos, int color) {
    array[yPos][xPos][zPos] = color;
}
public int getArrayVal(int xPos, int yPos, int zPos)
{
    return array[yPos][xPos][zPos];
}

/**
 * Gets a value in the array
 * @param v Vertex for which the value should be returned
 * @return The value at the given position
 */
public int getArrayVal(vertex v)
{
    return array[v.getY()][v.getX()][v.getZ()];
}

/**
 * Prints an array using a FileWriter
 * @param out A FileWriter
 * @throws IOException
 */
public void printArray(FileWriter out) throws IOException
{
    String line = "";
    for(int layer = 0; layer < layers-1; layer++)
    {
        for(int j = 0; j < width; j++)
        {
            line += array[i][j][layer] + " ";
        }
        line += " : ";
    }
    for(int j = 0; j < width-1; j++)
    {
        line += array[i][j][layers-1] + " ";
    }
    line += array[i][width-1][layers-1];
    out.write(line);
    out.write(System.lineSeparator());
}

/**
 * Prints an array using a FileWriter
 * @param out A FileWriter
 * @throws IOException
 */
public void printArray(File f) throws IOException
{
    FileWriter out = new FileWriter(f, true);
    printArray(out);
    out.close();
}

/**
 * Appends the array to the end of a File with the given fileName
 * @param fileName The name of a File
 * @throws IOException
 */
public void printArray(String fileName) throws IOException
{
    FileWriter out = new FileWriter(fileName, true);
    printArray(out);
    out.close();
}
** Print an array to the console
*/
public void printArray()
{
    for (int i = 0; i < height; i++)
    {
        String line = "";
        for (int layer = 0; layer < layers - 1; layer++)
        {
            for (int j = 0; j < width; j++)
            {
                line += array[i][j][layer] + " ";
            }
            line += " : ";
        }
        for (int j = 0; j < width - 1; j++)
        {
            line += array[i][j][layers - 1] + " ";
        }
        line += array[i][width - 1][layers - 1];
        System.out.println(line);
    }
    System.out.println();
}
/**
 * Prints the graph array to the console after applying the symmetry
 * @param s A symmetry
 */
public void printArray(s)
{
    printArray(getArray(s));
}
/**
 * Returns the proportion of the array values that are not less than minColor
 * or greater than maxColor
 * @param minColor The lower bound for colors not outside the range
 * @param maxColor The upper bound for colors not outside the range.
 * @return The proportion of the array values that are not less than minColor
 * or greater than maxColor
 */
public double density(int minColor, int maxColor)
{
    int numOutOfRange = 0;
    for (int i = 0; i < height; i++)
    {
        for (int j = 0; j < width; j++)
        {
            for (int k = 0; k < layers; k++)
            {
                if (array[i][j][k] < minColor || array[i][j][k] > maxColor)
                {
                    numOutOfRange++;
                }
            }
        }
    }
    return 1 - (double)(numOutOfRange)/(width*height*layers);
}
/**
 * Returns the proportion of array values that are nonzero
 * @return the proportion of array values that are nonzero
 */
public double density()
{
    int numZeros = 0;
    for (int i = 0; i < height; i++)
    {
```java
for (int j = 0; j < width; j++) {
    for (int k = 0; k < layers; k++) {
        if (array[i][j][k] == 0) {
            numZeros ++;
        }
    }
}
return 1 - (double)(numZeros)/(width * height * layers);
}/**
 * Returns the density of a given array
 * @param graphArray A graph array
 * @return */
public static double density(int[][][] graphArray) {
    int numZeros = 0;
    int height = graphArray.length;
    int width = graphArray[0].length;
    int layers = graphArray[0][0].length;
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            for (int k = 0; k < layers; k++) {
                if (graphArray[i][j][k] == 0) {
                    numZeros ++;
                }
            }
        }
    }
    return 1 - (double)(numZeros)/(width * height * layers);
}/**
 * Gets the minimum distance from a vertex to a copy of itself in another tile.
 * Used to ensure a color is not used in too small a tile.
 * @return An integer distance */
public abstract int getTileDiameter();
/**
 * Returns an array of symmetries for the current graph.
 * This method should be overwritten to include more than the trivial symmetry.
 * @return A list of symmetries for the current graph.
 */
public symmetry[] symmetryList() {
    return symmetryList(width, height, layers);
}/**
 * Returns an array of symmetries for the current graph type, given new dimensions.
 * This method should be overwritten to include more than the trivial symmetry.
 * @return A list of symmetries for the current graph type.
 */
public abstract symmetry[] symmetryList(int width, int height, int layers);
/**
 * Gets the graph array after applying the symmetry
 * @param s a symmetry
 * @return the array after applying the symmetry */
```
public int[][][] getArray(symmetry s){
    int[][][] a = new int[height][width][layers];
    for(int y = 0; y < height; y++){
        for(int x = 0; x < width; x++){
            for(int z = 0; z < layers; z++){
                a[y][x][z] = s.getArrayVal(this, x, y, z);
            }
        }
    }
    return a;
}

/**
 * Checks if the current graph has precisely the same uncolored vertices as
 * those in another graph.
 * @param h The other graph.
 * @return True if the graph arrays are zero at the same set of entries.
 */
public boolean arraysMatch(graph h){
    for(int i = 0; i < height; i++){
        for(int j = 0; j < width; j++){
            for(int k = 0; k < layers; k++){
                if(array[i][j][k] == 0 & h.getArrayVal(j, i, k) == 0){
                    return false;
                }
            }
        }
    }
    return true;
}

/**
 * Checks if the current graph has precisely the same uncolored vertices as
 * those in another graph array.
 * @param graphArray The other graph array.
 * @return True if the graph arrays are zero at the same set of entries.
 */
public boolean arraysMatch(int[][][] graphArray){
    for(int i = 0; i < height; i++){
        for(int j = 0; j < width; j++){
            for(int k = 0; k < layers; k++){
                if(array[i][j][k] == 0 & graphArray[i][j][k] == 0){
                    return false;
                }
            }
        }
    }
    return true;
}

/**
 * Checks if the current graph has precisely the same uncolored vertices as
 * those in another graph array after applying the symmetry.
 * @param graphArray The other graph array.
 * @param s A symmetry.
 * @return True if the graph arrays are zero at the same set of entries.
 */
public boolean arraysMatch(int[][][] graphArray, symmetry s){
    return s.arraysMatch(graphArray, this);
}

/**
* Appends a graph array to the end of a file.
* @param graphArray
* @param f
* @throws IOException
* /
}

public static void printArray(int[][][] graphArray, File f) throws IOException{
    FileWriter out = new FileWriter(f, true);
    int height = graphArray.length;
    int width = graphArray[0].length;
    int layers = graphArray[0][0].length;
    for(int i = 0; i < height; i++){
        String line = "";
        for(int layer = 0; layer < layers-1; layer++){
            for(int j = 0; j < width; j++){
                line += graphArray[i][j][layer] + " ";
            }
            line += " : ";
        }
        for(int j = 0; j < width-1; j++){
            line += graphArray[i][j][layers-1] + " ";
        }
        line += graphArray[i][width-1][layers-1] + " ";
        out.write(line);
        out.write(System.lineSeparator());
    }
    out.write(System.lineSeparator());
    out.close();
}

/**
 * Prints the graph array to the console.
 * @param array A graph array.
 */

public static void printArray(int[][][] array){
    int height = array.length;
    int width = array[0].length;
    int layers = array[0][0].length;
    for(int i = 0; i < height; i++){
        String line = "";
        for(int layer = 0; layer < layers-1; layer++){
            for(int j = 0; j < width; j++){
                line += array[i][j][layer] + " ";
            }
            line += " : ";
        }
        for(int j = 0; j < width-1; j++){
            line += array[i][j][layers-1] + " ";
        }
        line += array[i][width-1][layers-1] + " ";
        System.out.println(line);
    }
    System.out.println();
}
/**
@param width Width of the array
@param height Height of the array
@param layers Number of layers in the array
@param firstEntryIsOne True if the array should start with 1.
@returns an integer array: int[height][width][layers]

public static int[][][] alternatingOnesArray(int width, int height, int layers, boolean firstEntryIsOne)
{
  int[][][] a = new int[height][width][layers];
  for (int y = 0; y < height; y++){
    for (int x = 0; x < width; x++){
      for (int z = 0; z < layers; z++){
        if ((x+y+z)%2 == 0 ^ !firstEntryIsOne){
          a[y][x][z] = 1;
        }
      }
    }
  }
  return a;
}

/**
 * Returns an array where every other entry is 1. The first entry will be 1.
 * @param width Width of the array
 * @param height Height of the array
 * @param layers Number of layers in the array
 * @return an integer array: int[height][width][layers]
 */
public static int[][][] alternatingOnesArray(int width, int height, int layers) {
  return alternatingOnesArray(width, height, layers, true);
}

/**
 * Returns a new copy of an array with all instances of one value replaced by another value.
 * @param graphArray A graph array.
 * @param oldColor The value to be replaced.
 * @param newColor The new value.
 * @return A new graph array.
 */
public static int[][][] replaceColor(int[][][] graphArray, int oldColor, int newColor) {
  int[][][] newArray = new int[graphArray.length][graphArray[0].length][graphArray[0][0].length];
  for (int y = 0; y < graphArray.length; y++){
    for (int x = 0; x < graphArray[0].length; x++){
      for (int z = 0; z < graphArray[0][0].length; z++){
        if (graphArray[y][x][z] == oldColor){
          newArray[y][x][z] = newColor;
        } else {
          newArray[y][x][z] = graphArray[y][x][z];
        }
      }
    }
  }
  return newArray;
}

/**
 * Returns an array where every other entry is 1.
 * @param width Width of the array
 * @param height Height of the array
 * @param layers Number of layers in the array
 * @param firstEntryIsOne True if the array should start with 1.
 * @returns an integer array: int[height][width][layers]
 */
public void replaceColor(int oldColor, int newColor) {
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            for (int z = 0; z < layers; z++) {
                if (array[y][x][z] == oldColor) {
                    array[y][x][z] = newColor;
                }
            }
        }
    }
}

public vertex[] getVertices() {
    vertex[] vertices = new vertex[width * height * layers];
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            for (int z = 0; z < layers; z++) {
                vertices[z + x * layers + y * width * layers] = new vertex(x, y, z);
            }
        }
    }
    return vertices;
}

public vertex[] getZeros() {
    ArrayList<vertex> zeros = new ArrayList<vertex>();
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            for (int k = 0; k < layers; k++) {
                if (array[i][j][k] == 0) {
                    zeros.add(new vertex(j, i, k));
                }
            }
        }
    }
    return zeros.toArray(new vertex[zeros.size()]);
}

public static int[][][] expandArray(int[][][] oldArray, int verticalCopies, int horizontalCopies, int layerCopies){
int newHeight = oldArray.length * verticalCopies;
int newWidth = oldArray[0].length * horizontalCopies;
int newLayers = oldArray[0][0].length * layerCopies;

int[ ][ ][] newArray = new int[newHeight][newWidth][newLayers];
for (int y = 0; y < newHeight; y++)
    for (int x = 0; x < newWidth; x++)
        for (int z = 0; z < newLayers; z++)
            newArray[y][x][z] = oldArray[y%oldArray.length][x%oldArray[0].length][z%oldArray[0][0].length];
return newArray;
}

public static int[ ][ ][] expandArray(int[ ][ ][] oldArray, int verticalCopies, int horizontalCopies)
{
    return expandArray(oldArray, verticalCopies, horizontalCopies, 1);
}

/**
 * Generates an expanded graph of the same type as the given one. Expands the array by creating duplicates of it.
 * @param g A graph
 * @param verticalCopies Number of vertical copies to produce
 * @param horizontalCopies Number of horizontal copies to produce
 * @param layerCopies Number of layer copies to produce
 * @return An expanded graph
 */
public static graph expandGraph(graph g, int verticalCopies, int horizontalCopies, int layerCopies)
{
    int[ ][ ][] newGraphArray = expandArray(g.toArray(), verticalCopies, horizontalCopies, layerCopies);
    return g.makeNewGraph(newGraphArray);
}

/**
 * Generates expanded versions of all graphs in the array. Preserves graph types.
 * @param graphs An array of graphs
 * @param verticalCopies Number of vertical copies to produce
 * @param horizontalCopies Number of horizontal copies to produce
 * @param layerCopies Number of layer copies to produce
 * @return An array of expanded graphs.
 */
public static graph[] expandGraphs(graph[] graphs, int verticalCopies, int horizontalCopies, int layerCopies)
{
    graph[] newGraphs = new graph[graphs.length];
    for (int i = 0; i < graphs.length; i++)
        newGraphs[i] = expandGraph(graphs[i], verticalCopies, horizontalCopies, layerCopies);
    return newGraphs;
}

/**
 * Returns a new graph of the same type with the given graph array. Useful for generating the proper type of graph.
 * @param graphArray The graph array.
 * @return A graph of the same type.
 */
public abstract graph makeNewGraph(int [][] graphArray);
/**
 * Returns a new graph of the same type with the given dimensions. Useful for
generating the proper type of graph.
 * @param width The graph’s width.
 * @param height The graph’s height.
 * @param layers The graph’s number of layers.
 * @return A graph of the same type.
 */
public abstract graph makeNewGraph(int width, int height, int layers);
/**
 * Checks that the packing is valid; i.e. that two vertices in color class i
are at distance > i apart.
 * @return True if the packing is valid.
 */
public boolean validateDistanceColoring()
{
    ArrayList<ArrayList<vertex>> vPos = new ArrayList<ArrayList<vertex>>(); // list
    of color classes
    for(int y = 0; y < height; y++)
    {
        for(int x = 0; x < width; x++)
        {
            for(int z = 0; z < layers; z++)
            {
                vertex v = new vertex(x,y,z);
                int vVal = getArrayVal(v);
                if(vVal != 0)
                {
                    while(vVal > vPos.size())
                    {
                        vPos.add(new ArrayList<vertex>());
                    }
                    vPos.get(vVal - 1).add(v);
                }
            }
        }
    }
    for(int i = 0; i < vPos.size(); i++)
    {
        ArrayList<vertex> vertices = vPos.get(i);
        for(int j = 0; j < vertices.size(); j++)
        {
            for(int k = j + 1; k < vertices.size(); k++)
            {
                if(distance(vertices.get(j), vertices.get(k)) <= i + 1)
                {
                    return false;
                }
            }
        }
    }
    return true;
}
/**
 * Checks that a particular color class is valid; i.e. that any pair of
vertices in the color class
are at distance greater than the color apart
 * @param color
 * @return
 */
public boolean validateDistanceColoring(int color)
{
    ArrayList<vertex> vertices = new ArrayList<vertex>();
    for(int y = 0; y < height; y++)
    {
        for(int x = 0; x < width; x++)
        {
            for(int z = 0; z < layers; z++)
            {
                vertex v = new vertex(x,y,z);
            }
        }
    }
int vVal = getArrayVal(v);
if (vVal == color) {
    vertices.add(v);
}
}

for (int i = 0; i < vertices.size(); i++) {
    for (int j = i+1; j < vertices.size(); j++) {
        if (distance(vertices.get(i), vertices.get(j)) <= color) {
            return false;
        }
    }
}
return true;

/**
 * This optional method is intended to print some sort of message regarding the graph’s value.
 * @param color The largest color currently used in the graph.
 * /
public void printProgress(int color);{}

/**
 * Makes a separate copy of the graph.
 * @return A copy of the graph.
 * /
public graph makeCopy(){
    graph g = makeNewGraph(width, height, layers);
    vertex[] vertices = getVertices();
    for (vertex v: vertices) {
        g.setArrayVal(v, getArrayVal(v));
    }
    return g;
}

/**
 * Gets the max density from an array of graphs
 * @param graphs An array of graphs
 * @return Max density from the graphs in the array
 * /
public static double findMaxDensity(graph[] graphs){
    double maxDensity = 0;
    for (graph g: graphs) {
        double density = g.density();
        if (density > maxDensity) {
            maxDensity = density;
        }
    }
    return maxDensity;
}
B.1.2 Vertex

The vertex object is simple but fundamental. It is used to simplify iterating on coordinates in graph arrays, as well as to add flexibility by storing extra information on the vertices themselves.

```java
package GraphPacking;

/**
 * Stores 2D or 3D points with integer coordinates
 * @author Kevin Moss
 */
public class vertex {
    private int x = 0, y = 0, z = 0;
    private int tokens = 0; // used for skewed choosing when choosing randomly
    private int tempTokens; // used for graphColorerMaximal
    private int vertexNum; // used to store a vertex's index

    /**
     * Initializes 2D point
     * @param x the x coordinate
     * @param y the y coordinate
     */
    public vertex(int x, int y){
        this.x = x;
        this.y = y;
    }

    /**
     * initializes 3D point
     * @param x the x coordinate
     * @param y the y coordinate
     * @param z the z coordinate
     */
    public vertex(int x, int y, int z){
        this.x = x;
        this.y = y;
        this.z = z;
    }

    public int getX(){
        return x;
    }

    public int getY(){
        return y;
    }

    public int getZ(){
        return z;
    }

    public int getTokens(){
        return tokens;
    }

    public void setTokens(int tokens){
    }
}
B.1.3 Symmetry

The `symmetry` object, similar to `vertex`, is simple but useful. It is used to simplify iterating on the symmetries in a graph.
private boolean xFlip = false;
private boolean yFlip = false;
private boolean zFlip = false;
private boolean swapXY = false; // for the square grid; rotating 90 degrees

public symmetry() {
}
public symmetry(int xShift, int yShift) {
    this.xShift = xShift; this.yShift = yShift;
}
public symmetry(int xShift, int yShift, int zShift) {
    this.xShift = xShift; this.yShift = yShift; this.zShift = zShift;
}
public symmetry(int xShift, int yShift, boolean xFlip) {
    this.xShift = xShift; this.yShift = yShift;
    this.xFlip = xFlip;
}
public symmetry(int xShift, int yShift, int zShift, boolean xFlip) {
    this.xShift = xShift; this.yShift = yShift; this.zShift = zShift;
    this.xFlip = xFlip;
}
public symmetry(int xShift, int yShift, int zShift, boolean xFlip, boolean yFlip) {
    this.xShift = xShift; this.yShift = yShift; this.zShift = zShift;
    this.xFlip = xFlip; this.yFlip = yFlip;
}
public symmetry(int xShift, int yShift, int zShift, boolean xFlip, boolean yFlip, boolean zFlip) {
    this.xShift = xShift; this.yShift = yShift; this.zShift = zShift;
    this.xFlip = xFlip; this.yFlip = yFlip; this.zFlip = zFlip;
}
public symmetry(int xShift, int yShift, int zShift, boolean xFlip, boolean yFlip, boolean zFlip, boolean swapXY) {
    this.xShift = xShift; this.yShift = yShift; this.zShift = zShift;
    this.xFlip = xFlip; this.yFlip = yFlip; this.zFlip = zFlip;
    this.swapXY = swapXY;
}

public int getXShift() {
    return xShift;
}
public int getYShift() {
    return yShift;
}
public int getZShift() {
    return zShift;
}
public boolean getXFlip() {
    return xFlip;
}

/**
 * Returns an array value in a graph after applying the symmetry.
 * The array value returned is the that of the new position of the coordinate after
 * applying the symmetry to the original graph.
 * @param g A graph.
 * @param xPos The x position of the array value.
@param yPos The y position of the array value.
@param zPos The z position of the array value.
@return The array value after applying the symmetry.

```java
public int getArrayVal(graph g, int xPos, int yPos, int zPos)
{
    int width = g.getWidth();
    int height = g.getHeight();
    int layers = g.getLayers();
    int xNew = xFlip ? (xShift - xPos + width) % width : (xShift + xPos + width) % width;
    int yNew = yFlip ? (yShift - yPos + height) % height : (yShift + yPos + height) % height;
    int zNew = zFlip ? (zShift - zPos + layers) % layers : (zShift + zPos + layers) % layers;
    if (swapXY)
    {
        int temp = xNew;
        xNew = yNew;
        yNew = temp;
    }
    return g.getArrayVal(xNew, yNew, zNew);
}
```

Returns an array value in a graph after applying the symmetry.
The array value returned is the that of the new position of the vertex after applying the symmetry to the original graph.

```java
public int getArrayVal(graph g, vertex v)
{
    return getArrayVal(g, v.getX(), v.getY(), v.getZ());
}
```

Checks if the graph array is zero at precisely the same locations as the shifted graph.

```java
public boolean arraysMatch(int[][][] fixedGraphArray, graph shiftGraph)
{
    for (int y = 0; y < shiftGraph.getHeight(); y++){
        for (int x = 0; x < shiftGraph.getWidth(); x++){
            for (int z = 0; z < shiftGraph.getLayers(); z++){
                if (fixedGraphArray[y][x][z] == 0 && getArrayVal(shiftGraph, x, y, z) == 0)
                {
                    return false;
                }
            }
        }
    }
    return true;
}
```

Checks if the symmetry is the default symmetry.

```java
public boolean isDefault()
{
```
B.1.4 Graph Colorer

The graphColorer object primarily implements backtracking, but also stores some information for other coloring objects.

```java
package GraphPacking;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.ArrayList;
/
∗∗
∗
∗
A backtracking graph coloring algorithm. Recursively generates colorings of a
graph.
∗
@author Kevin Moss
*/
public class graphColorer {
private graph g;
private int minColor, maxColor;
public static String directory = "./offsetGraphOutput"; // directory can be
changed before running algorithm
private File outFile;
private int width, height, layers;
private ArrayList<vertex>[][][][] neighborhoods; // <= color[y][x][z]
private double densityThreshold = 0;
static final double EPSILON = 0.00000000000001; // A sufficiently small value.
    // Used for comparing doubles.
private boolean maxDensityOnly = false; // True if only the graphs with maximum
density should be stored.
symmetry[] symmetries;

boolean keepCandidatesInMem = false;
ArrayList<graph> candidates = new ArrayList<graph>();

public graphColorer(graph g, String graphName, int minColor, int maxColor)
throws IOException{
    this(g, graphName, minColor, maxColor, false, false);
}
/**
 * Instantiates a graphColorer object and runs the backtracking algorithm.
 * The coloring algorithm runs upon instantiation of this object, so the object
 * may be discarded after running.
 * @param g The graph on which to run the algorithm.
 * @param graphName Part of the name of the file to be generated and written on
 * @param minColor The minimum value among the range of colors to use.
 * @param maxColor The maximum value among the range of colors to use.
 * @param maxDensityOnly True if only graphs with the maximum density should be
 * recorded.
 */
```
* @param keepCandidatesInMem True to keep all recorded graphs in memory. Speeds up cross-checking at the cost of space.
* @throws IOException */
public graphColorer ( graph g , String graphName , int minColor , int maxColor , boolean maxDensityOnly , boolean keepCandidatesInMem ) throws IOException{  
if ( minColor <= 0){
    throw new IllegalArgumentException ("minColor must be at least 1");
} else if (minColor > maxColor){
    throw new IllegalArgumentException("minColor cannot be greater than maxColor");
} else if (g.getTileDiameter() <= maxColor){
    throw new IllegalArgumentException("Graph dimensions are too small to fit maxColor");
}
  this . g = g;
  this . minColor = minColor;
  this . maxColor = maxColor;
  width = g.getWidth();
  height = g.getHeight();
  layers = g.getLayers();
  File fileDir = new File(directory);
  fileDir.mkdir();
  String outFileName = graphName + "C" + maxColor + "D" + height + "x" + width + "x" + layers + ".txt";
  outFile = new File(directory + "/" + outFileName);
  outFile.createNewFile();
  symmetries = g.symmetryList();

  this . keepCandidatesInMem = keepCandidatesInMem;
  if (keepCandidatesInMem){
    fileReader fr = new fileReader(outFile);
    while (fr.hasNextGraph()){
      graph tempGraph = g.makeNewGraph(fr.getNextGraphArray());
      candidates . add(tempGraph);
    }
    fr . close();
  }
  
  if (maxDensityOnly){
    this . maxDensityOnly = true;
    densityThreshold = findMaxDensity(outFile);
  }
  neighborhoods = generateNeighborhoods();
  colorNext(new vertex(0,0,0));

  if (maxDensityOnly){
    deleteSparseColorings(outFile,1);
  }
}

/**
 * Finds the maximum density of a graph within the file.
 * @param file A file.
 * @return A density.
```java
public static double findMaxDensity(File file) {
    double densityThreshold = 0;
    FileReader f;
    try {
        f = new FileReader(file);
        while (f.hasNextGraph()) {
            int[][][] graphArray = f.getNextGraphArray();
            double tempDensity = graph.density(graphArray);
            densityThreshold = densityThreshold < tempDensity ? tempDensity : densityThreshold;
            f.close();
        }
        catch (Exception e) {
        }
        return densityThreshold;
    }
    /*
     * A recursive algorithm. For each possible color of the current vertex, chooses that color
     * and moves the algorithm to the next color.
     * If at the last vertex in the graph, checks and possibly stores the coloring.
     * @param v1 A vertex.
     * @throws IOException
     */
    private void colorNext(vertex v1) throws IOException {
        if (v1.getY() >= g.getHeight()) {
            if (!keepCandidatesInMem) {
                if (checkColoring()) {
                    g.printArray(outFile);
                }
            } else {
                if (checkColoring(candidates)) {
                    candidates.add(g.makeCopy());
                    g.printArray(outFile);
                }
            }
        } else {
            vertex v2 = nextVertex(v1);
            if (g.getArrayVal(v1) == 0) {
                for (int c = minColor; c <= maxColor; c++) {
                    boolean canUseColor = true;
                    search:
                    for (int cTemp = minColor; cTemp <= c; cTemp++) {
                        for (vertex v : neighborhoods[cTemp - minColor][v1.getY()][v1.getX()][v1.getZ()]) {
                            if (g.getArrayVal(v) == c) {
                                canUseColor = false;
                                break search;
                            }
                        }
                    }
                    if (canUseColor) {
                        g.setArrayVal(v1, c);
                        colorNext(v2);
                    }
                }
                g.setArrayVal(v1, 0);
            }
        }
    }
```
colorNext(v2);
}

/**
 * Determines the next vertex in the list of uncolored vertices.
 * @param v A vertex
 * @return A vertex
 */
private vertex nextVertex(vertex v){
    if (v.getZ() < layers - 1){
        return new vertex(v.getX(), v.getY(), v.getZ() + 1);
    }
    if (v.getX() < width - 1){
        return new vertex(v.getX() + 1, v.getY(), 0);
    }
    return new vertex(0, v.getY() + 1, 0);
}//To make the algorithm simpler and more efficient, we could iterate on the
list of uncolored vertices rather than all vertices.

/**
 * Determines whether or not a coloring should be stored.
 * First checks that the density meets the required threshold.
 * Then checks if the coloring is maximal (no more colors can be added).
 * Finally cross-checks the coloring with other stored colorings.
 * @return True if the coloring should be stored.
 * @throws FileNotFoundException
 */
private boolean checkColoring() throws FileNotFoundException{
    if (maxDensityOnly){
        if (g.density() < densityThreshold - EPSILON){
            return false;
        }
    }
    if (!maximal()){}
    return false;
}

fileReader f = new fileReader(outFile);
while (f.hasNextGraph()){
    int [][][] graphArray = f.getNextGraphArray();
    for (symmetry s : symmetries){
        if (g.arraysMatch(graphArray, s)){
            f.close();
            return false;
        }
    }
}
f.close();
return true;

/**
 * Determines whether or not a coloring should be stored.
 * First checks that the density meets the required threshold.
 * Then checks if the coloring is maximal (no more colors can be added).
 * Finally cross-checks the coloring with other stored colorings.
 * @param graphs The graphs to be cross-checked with.
 * @return True if the coloring should be stored.
 * @throws FileNotFoundException
 */
private boolean checkColoring(ArrayList<graph> graphs) {
    if (maxDensityOnly) {
        if (g.density() < densityThreshold - EPSILON) {
            return false;
        }
    }
    if (!maximal()) {
        return false;
    }
    for (graph h : graphs) {
        for (symmetry s : symmetries) {
            if (g.arraysMatch(h.getArray(), s)) {
                return false;
            }
        }
    }
    return true;
}

/**
 * To save time at the cost of space, a list of the vertices within a certain
distance of each vertex is generated.
 * The lists are iterated on rather than checking pairwise distance for each
vertex.
 * @return An ArrayList of vertices within a certain distance of each vertex in
the graph.
*/
private ArrayList<vertex>[][] generateNeighborhoods() {
    @SuppressWarnings("unchecked")
    ArrayList<vertex>[][] neighborhoods = (ArrayList<vertex>[][]) new
    ArrayList[maxColor - minColor + 1][height][width][layers];
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            for (int k = 0; k < layers; k++) {
                vertex v1 = new vertex(j, i, k);
                for (int c = minColor; c <= maxColor; c++) {
                    neighborhoods[c - minColor][i][j][k] = new ArrayList<vertex>();
                }
            }
            for (int i2 = 0; i2 < height; i2++) {
                for (int j2 = 0; j2 < width; j2++) {
                    vertex v2 = new vertex(j2, i2, k);
                    int distTemp = g.distance(v1, v2);
                    if (distTemp <= maxColor) {
                        distTemp = distTemp < minColor ? minColor : distTemp;
                        neighborhoods[distTemp - minColor][i][j][k].add(v2);
                    }
                }
            }
        }
    }
    return neighborhoods;
}
private boolean maximal() {
    for (int y = 0; y < height; y++){
        for (int x = 0; x < width; x++){
            for (int z = 0; z < layers; z++){
                if (g.getArrayVal(x, y, z) == 0){
                    boolean[] unusableColors = new boolean[maxColor - minColor + 1];
                    boolean vertexIsColorable = true;
                    int largestUnfoundColor = maxColor; // usableColors[k] == true for all k > largestUnfoundColor
                    searchVertex: for (int c = minColor; c <= maxColor; c++){
                        if (c > largestUnfoundColor){
                            return false;
                        }
                    for (vertex v : neighborhoods[c-minColor][y][x][z]){ if (g.getArrayVal(v) >= c && g.getArrayVal(v) <= largestUnfoundColor){
                            if (unusableColors[g.getArrayVal(v)-minColor] == false){
                                usableColors[g.getArrayVal(v)-minColor] = true;
                                int temp = 0;
                                for (int i = minColor; i <= largestUnfoundColor; i++){
                                    if (!usableColors[i - minColor]){ temp = i ; }
                                }
                                if (temp == 0){
                                    vertexIsColorable = false;
                                    break searchVertex; // continue searching next vertex
                                }
                                largestUnfoundColor = temp;
                            }
                        }
                    }
                    }
                }
            }
        }
    }
    return true;
}"**
* Deletes all colorings in a file that do not meet or exceed the density threshold.
* @param graphFile A file
* @param densityThreshold A density threshold
* @throws IOException
*/
public static void deleteSparseColorings(File graphFile, double densityThreshold) throws IOException{
    if (!graphFile.getName().endsWith(".txt")){
        throw new IllegalArgumentException("Graph file must be of type .txt");
    }
String filePath = graphFile.getAbsolutePath();
String outputName = filePath.substring(0, filePath.length() - 4);
File temp = new File(outputName + "temp.txt");
temp.createNewFile();
fileReader r = new fileReader(graphFile);
while (r.hasNextGraph()){
    int[][][] graphArray = r.getNextGraphArray();
    if (graph.density(graphArray) >= densityThreshold - EPSILON){
        graph.printArray(graphArray, temp);
    }
}
r.close();
graphFile.delete();
temp.renameTo(graphFile);
}
/**
 * Generates colorings with a static method (for convenience) by creating a
 * graphColorer object.
 * @param g The graph on which to run the algorithm.
 * @param graphName Part of the name of the file to be generated and written on
 * .
 * @param minColor The minimum value among the range of colors to use.
 * @param maxColor The maximum value among the range of colors to use.
 * @param maxDensityOnly True if only graphs with the maximum density should be
 * recorded.
 * @param keepCandidatesInMem True to keep all recorded graphs in memory.
 * Speeds up cross-checking at the cost of space.
 * @throws IOException
 */
public static void generateColorings(graph g, String graphName, int minColor, int maxColor, boolean maxDensityOnly, boolean keepCandidatesInMem) throws IOException{
    @SuppressWarnings("unused")
    graphColorer c = new graphColorer(g, graphName, minColor, maxColor, maxDensityOnly, keepCandidatesInMem);
}
public static void generateColorings(graph g, String graphName, int minColor, int maxColor) throws IOException{
    @SuppressWarnings("unused")
    graphColorer c = new graphColorer(g, graphName, minColor, maxColor, false, false);
}
/**
 * Counts the number of graphs in a file that meet or exceed the density
 * threshold.
 * @param f A file.
 * @param densityThreshold A density.
 * @return An integer.
 * @throws FileNotFoundException
 */
public static int countGraphs(File f, double densityThreshold) throws FileNotFoundException{
    fileReader r = new fileReader(f);
    int numGraphs = 0;
    if (densityThreshold == 0){
        while (r.hasNextGraph()){
            r.getNextGraphArray();
        }
    } else{
        while (r.hasNextGraph()){
            r.getNextGraphArray();
        }
    }
    return numGraphs;
}
numGraphs++;
}
}
else {
    while (r.hasNextGraph()) {
        double tempDensity = graph.density(r.getNextGraphArray());
        if (tempDensity >= densityThreshold - EPSILON)
            numGraphs++;
    }
}
return numGraphs;
}

/**
 * Checks if a graph array is maximal. That is, there is no other vertex for
 * which the graph exists.
 * @param graphArray
 * @param graphType
 * @return
 */
public static boolean isMaximal(int[][][] graphArray, graph graphType) {
    ArrayList<Integer> colorsInGraph = new ArrayList<Integer>();
    ArrayList<vertex> zeros = new ArrayList<vertex>();
    int width = graphArray[0].length;
    int height = graphArray.length;
    int layers = graphArray[0][0].length;
    boolean isMaximal = true;
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            for (int z = 0; z < layers; z++) {
                if (graphArray[y][x][z] == 0) {
                    zeros.add(new vertex(x, y, z));
                } else {
                    boolean newColor = true;
                    for (int i : colorsInGraph) {
                        if (i == graphArray[y][x][z]) {
                            newColor = false;
                            break;
                        }
                    }
                    if (newColor) {
                        colorsInGraph.add(graphArray[y][x][z]);
                    }
                }
            }
        }
    }
    for (int color : colorsInGraph) {
        @SuppressWarnings("unchecked")
        ArrayList<vertex> tempZeros = (ArrayList<vertex>) zeros.clone();
        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                for (int z = 0; z < layers; z++) {
                    if (graphArray[y][x][z] == color) {
                        vertex v = new vertex(x, y, z);
                        int i = tempZeros.size();
                        while (i > 0) {
                            ...
B.1.5 Local Random Graph Colorer

The `graphColorerLocalRandom` object implements a priority-based random coloring algorithm. Once a vertex is colored, priority is given to nearby vertices.

```java
package GraphPacking;
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Random;
/**
 * Class for coloring large grids randomly.
 * @author Kevin Moss
 */
public class graphColorerLocalRandom {
    //private String graphName;
    protected graph g;
    protected double startDensity;
    protected int color;
    protected File outFile;
    protected int width, height, layers;
    public static double densityThreshold = 0;
    protected static double densityMax = 0;
    public final static double proportionOfMaxAllowed = 0.95;
    protected static int numDenseGraphs = 0;
    protected vertex[] zerosBase;
    protected Random seed;
    /**
     * Instantiates a graphColorerLocalRandom object.
     * A separate method runs an iteration of the colorer; it may be used multiple times.
     * @param g A graph that the object will color.
     * @param graphName Name of the graph.
     * @param color The color class that will be generated.
     * @throws IOException
     */
```
public graphColorerLocalRandom(graph g, String graphName, int color) throws IOException{
    //Static variables should be initialized first with initializeDensityVars if
    if (color <= 0){
        throw new IllegalArgumentException("minColor must be at least 1");
    } else if (g.getTileDiameter() <= color){
        throw new IllegalArgumentException("Graph dimensions are too small to fit maxColor");
    }
    this.g = g;
    this.color = color;
    width = g.getWidth();
    height = g.getHeight();
    layers = g.getLayers();
    seed = new Random();

    File fileDir = new File(graphColorer.directory);
    fileDir.mkdir();
    String outFileName = graphName + "C" + color + "D" + height + "x" + width + "x" + layers + ".txt";
    outFile = new File(graphColorer.directory + "/" + outFileName);
    outFile.createNewFile();
    startDensity = g.density();
    zerosBase = g.getZeros();
}

/**
 * Instantiates a graphColorerLocalRandom object with a preset density threshold.
 * A separate method runs an iteration of the colorer; it may be used multiple times.
 * @param g A graph that the object will color.
 * @param graphName Name of the graph.
 * @param color The color class that will be generated.
 * @param density A density threshold.
 * @throws IOException
 */
public graphColorerLocalRandom(graph g, String graphName, int color, double density) throws IOException{
    this(g, graphName, color);
    densityMax = density > densityMax ? density : densityMax;
    if (startDensity < densityMax){
        densityThreshold = (densityMax - startDensity)*proportionOfMaxAllowed + startDensity;
    } else {
        densityThreshold = startDensity;
    }
}

/**
 * Initializes static variables, particularly numDenseGraphs.
 * If we have a set (>= 1) of graphColorerLocalRandom objects
 * (different graphs, but same dimensions and working color class),
 * then this method should be run before generating colorings with the objects.
 * @param graphName The prefix associated with the working file.
 * @param color The current working color.
 * @param startDensity The initial density threshold (in case the working file
 * is empty or a higher density is desired).
public static void initializeDensityVars(String graphName, int color, double startDensity, int height, int width, int layers) throws IOException{
    String outFileName = graphName + "C" + color + "D" + height + "x" + width + "x" + layers + ".txt";
    File outFile = new File(graphColorer.directory + "/" + outFileName);
    if (outFile.createNewFile()){
        densityMax = 0;
        numDenseGraphs = 0;
        densityThreshold = startDensity;
    } else {
        densityMax = graphColorer.findMaxDensity(outFile);
        if (startDensity < densityMax){
            densityThreshold = (densityMax - startDensity)*proportionOfMaxAllowed + startDensity;
        } else {
            densityThreshold = startDensity;
        }
        numDenseGraphs = graphColorer.countGraphs(outFile, densityThreshold);
    }
}

/**
 * Generates a coloring of the graph, checks and possibly records the coloring, then resets the graph.
 * Static variables densityMax, densityThreshold, and numDenseGraphs are updated as needed.
 * @return True if the graph generated has higher density than the previous max.
 * @throws IOException
 */
public boolean colorGraph() throws IOException{
    ArrayList<vertex> zeros = new ArrayList<vertex>();
    for (vertex v: zerosBase){
        zeros.add(v);
    }

    ArrayList<vertex> priority = new ArrayList<vertex>();
    int priorityNum = 0;

    int numZeros = zeros.size();

    while (!zeros.isEmpty()){
        vertex choice;
        if (priority.isEmpty()){  
            choice = zeros.get((int)(zeros.size()*Math.random()));
        } else {
            choice = priority.get((int)(priority.size()*Math.random()));
        }
        priority.clear();
        priorityNum = 0;
        g.setArrayVal(choice, color);
numZeros --;
for (int i = 0; i < zeros.size(); i++){
    vertex v = zeros.get(i);
    int distance = g.distance(choice, v); //TODO: use a pre-generated distance graph to save time
    if (distance <= color){
        zeros.remove(i);
        i--;
    } else {
        if (distance < 2*color){
            v.addTokens(2*color - distance);
            if (v.getTokens() > priorityNum){
                priorityNum = v.getTokens();
                priority.clear();
            }
            if (v.getTokens() == priorityNum){
                priority.add(v);
            }
        }
    }
}

boolean densityImprovement = false;

double density = 1 - (double)(numZeros)/(width*height*layers);
boolean checkCol = true;

if (density < densityThreshold - graphColorer.EPSILON){
    checkCol = false;
} else if (density > densityMax + graphColorer.EPSILON){
    //densityThreshold should be updated after every graph
    densityMax = density;
    densityThreshold = (densityMax - startDensity)*proportionOfMaxAllowed + startDensity;
    densityImprovement = true;
} else if (density < densityMax - graphColorer.EPSILON && numDenseGraphs > 800000/(g.getHeight()*g.getWidth()) && numDenseGraphs > 0){
    checkCol = false; //too many graphs at or above this density
} else if (numDenseGraphs > 1600000/(g.getHeight()*g.getWidth()) && numDenseGraphs > 1){
    checkCol = false; //too many graphs at this density
}
if (checkCol){
    numDenseGraphs ++;
    g.printArray(outFile);
}

for (vertex v : zerosBase){
    g.setArrayVal(v, 0);
    v.resetTokens();
}

return densityImprovement;

/**
 * Runs the color generating algorithm a set number of times.
 * @param numTrials Number of times to run the algorithm.
 */
public void colorGraph(int numTrials) throws IOException{
    for(int i = 0; i < numTrials; i++){
        if(colorGraph()){
            numDenseGraphs = 1;
            graphColorer.deleteSparseColorings(outFile, densityThreshold);
        }
    }
}

public static int generateColorings(graph g, String graphName, int color, int numTrials) throws IOException{
    graphColorerLocalRandom gclr = new graphColorerLocalRandom(g, graphName, color);
    int numGraphs = numDenseGraphs;
    int progress = 0;
    gclr.colorGraph(numTrials);
    if(numDenseGraphs > 160000/g.getHeight()*g.getWidth() && numDenseGraphs > 0){
        progress = 2;
    } else if(numDenseGraphs > numGraphs){
        progress = 1;
    } else if(numDenseGraphs < numGraphs){
        progress = 3;
    }
    return progress;
}

public graph getGraph(){
    return g;
}
B.1.6 Naive Random Graph Colorer

The graphColorerNaiveRandom object implements a simple random coloring algorithm. Vertices are colored entirely at random until no further vertices can be assigned the given color.
public boolean colorGraph() throws IOException {
    ArrayList<vertex> zeros = new ArrayList<vertex>();
    for (vertex v: zerosBase){
        zeros.add(v);
    }
    int numZeros = zeros.size();
    while (!zeros.isEmpty()){
        vertex choice = zeros.get((int)(zeros.size()*Math.random()));
        g.setArrayVal(choice, color);
        numZeros--;
        for (int i = 0; i < zeros.size(); i++){
            vertex v = zeros.get(i);
            int distance = g.distance(choice, v); //TODO: use a pre-generated
distance graph to save time
            if (distance <= color){
                zeros.remove(i);
                i--;
            }
        }
    }
    boolean densityImprovement = false;
    double density = 1 - (double)(numZeros)/(width*height*layers);
    boolean checkCol = true;
    if (density < densityThreshold - graphColorer.EPSILON){
        checkCol = false;
    } else if (density > densityMax + graphColorer.EPSILON){
        //densityThreshold should be updated after every graph
        densityMax = density;
        densityThreshold = (densityMax - startDensity)*proportionOfMaxAllowed +
        startDensity;
        densityImprovement = true;
    } else if (density < densityMax - graphColorer.EPSILON && numDenseGraphs >
    800000/(g.getHeight()*g.getWidth()) && numDenseGraphs > 0){
        checkCol = false; //too many graphs at or above this density
    } else if (numDenseGraphs > 1600000/(g.getHeight()*g.getWidth()) &&
    numDenseGraphs > 1){
        checkCol = false; //too many graphs at this density
    } if (checkCol){
        numDenseGraphs++;
        g.printArray(outFile);
    }
    for (vertex v: zerosBase){
        g.setArrayVal(v, 0);
    }
    return densityImprovement;
}

public static int generateColorings(graph g, String graphName, int color, int
numTrials) throws IOException{
package GraphPacking;
import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.Scanner;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class FileReader {
  private Scanner sc;
  boolean scAtEnd = false;

  public FileReader(File f) throws FileNotFoundException {
    sc = new Scanner(f);
  }
  public boolean hasNextGraph(){
public hexGrid getNextGrid()
{
    hexGrid h = new hexGrid(4, 4);
    if (sc.hasNextLine()){
        ArrayList<int[]> tempList = new ArrayList<int[]>();
        String line = sc.nextLine();
        while (!line.isEmpty()){
            String[] lineArray = line.split(" ");
            int[] intArray = new int[lineArray.length];
            for (int i = 0; i < lineArray.length; i++){
                intArray[i] = Integer.parseInt(lineArray[i]);
            }
            tempList.add(intArray);
            line = sc.nextLine();
        }
        int listSize = tempList.size();
        int[][] tempArray = new int[2][listSize][tempList.get(0).length];
        for (int i = 0; i < listSize; i++){
            tempArray[0][i] = tempList.get(i);
        }
        h.setArray(tempArray);
        return h;
    } else {
        throw new IllegalArgumentException("There are no more grids to be read.");
    }
}

public hexGridP2 getNextGridP2()
{
    hexGridP2 h = new hexGridP2(4, 4);
    if (sc.hasNextLine()){
        ArrayList<int[]> tempList = new ArrayList<int[]>();
        String line = sc.nextLine();
        while (!line.isEmpty()){
            String[] line1 = line.split(":");
            String[] lineArray1 = line1[0].split(" ");
            String[] lineArray2 = line1[1].split(" ");
            int[] intArray1 = new int[lineArray1.length];
            int[] intArray2 = new int[lineArray2.length];
            for (int i = 0; i < lineArray1.length; i++){
                intArray1[i] = Integer.parseInt(lineArray1[i]);
            }
            for (int i = 0; i < lineArray2.length; i++){
                intArray2[i] = Integer.parseInt(lineArray2[i]);
            }
            tempList.add(intArray1);
            tempList.add(intArray2);
            line = sc.nextLine();
        }
        int listHeight = tempList.size() / 2; // should always be even
        int listWidth = tempList.get(0).length;
        int[][][] tempArray = new int[2][listHeight][listWidth][2];
        for (int i = 0; i < listHeight; i++){
            for (int j = 0; j < listWidth; j++){
                for (int k = 0; k < 2; k++){
                    tempArray[0][i][j][k] = tempList.get(2*i + k)[j];
                }
            }
        }
    }
h.setArray(tempArray);
return h;
else {
    throw new IllegalArgumentException("There are no more grids to be read.");
}
}

public int[][] getNextGraphArray()
{
    if(sc.hasNextLine()){
        ArrayList<int>[][] tempList = new ArrayList<int>[][]();
        String line = sc.nextLine();
        while(!line.isEmpty()){
            String[] rows = line.split(" ");
            String[] row0 = rows[0].split(" ");
            int[][] lineToInt = new int[row0.length][rows.length];
            for(int j = 0; j < row0.length; j++){
                lineToInt[j][0] = Integer.parseInt(row0[j]);
            }
            for(int i = 1; i < rows.length; i++){
                String[] thisRow = rows[i].split(" ");
                for(int j = 0; j < thisRow.length; j++){
                    lineToInt[j][i] = Integer.parseInt(thisRow[j]);
                }
            }
            tempList.add(lineToInt);
            line = sc.nextLine();
        }
        int[][] graphArray = tempList.toArray(new int[tempList.size()][][][]);
        return graphArray;
    } else {
        throw new IllegalArgumentException("There are no more graphs to be read.");
    }
}

dummyGraph getNextGraph()
{
    return new dummyGraph(getNextGraphArray());
}

hexGridP2v2 getNextGridP2v2()
{
    hexGridP2v2 h = new hexGridP2v2(4,4);
    if(sc.hasNextLine()){
        ArrayList<int>[] tempList = new ArrayList<int>[](4); //make sure to init size when you first use temp array
        String line = sc.nextLine();
        while(!line.isEmpty()){
            String[] line2 = line.split(" ");
            String[] lineArray = line2[0].split(" ");
            String[] lineArray2 = line2[1].split(" ");
            int[] intArray = new int[lineArray.length];
            int[] intArray2 = new int[lineArray2.length];
            for(int i = 0; i < lineArray.length; i++){
                intArray[i] = Integer.parseInt(lineArray[i]); //maybe use try-catch?
            }
            for(int i = 0; i < lineArray2.length; i++){
                intArray2[i] = Integer.parseInt(lineArray2[i]); //maybe use try-catch?
            }
            tempList.add(intArray);
            line = sc.nextLine();
        }
        return new hexGridP2v2(intArray, intArray2);
    } else {
        throw new IllegalArgumentException("There are no more grids to be read.");
    }
}
tempList.add(intArray2);
line = sc.nextLine();
}
int listHeight = tempList.size() / 2; // should always be even
int listWidth = tempList.get(0).length;
int[][][] tempArray = new int[listHeight][listWidth][2];
for (int i = 0; i < listHeight; i++) {
    for (int j = 0; j < listWidth; j++) {
        for (int k = 0; k < 2; k++) {
            tempArray[i][j][k] = tempList.get(2 * i + k)[j];
        }
    }
}
h.setArray(tempArray);
return h;
} else {
    throw new IllegalArgumentException("There are no more grids to be read.");
}
}
public void close(){
    sc.close();
}

public static graph[] readDoc(File f, graph graphType) throws FileNotFoundException{
    fileReader fr = new fileReader(f);
    ArrayList<graph> graphs = new ArrayList<graph>();
    while (fr.hasNextGraph()) {
        int[][][] graphArray = fr.getNextGraphArray();
        graphs.add(graphType.makeNewGraph(graphArray));
    }
    fr.close();
    return graphs.toArray(new graph[graphs.size()]);
}
/**
 * Parses a fileName and gets the color and dimensions in an integer array.
 * The file should be of the type [prefix]C[color]D[height]x[width]x[layers].txt
 * @param f A file
 */
public static int[] parseName(File f){
    String s = f.getName();
    String pattern = "([+][\d]+)C(\\d+)D(\\d+)x(\\d+)x(\\d+)\..txt";
    Pattern r = Pattern.compile(pattern);
    Matcher m = r.matcher(s);
    if (m.find()) {
        int[] out = new int[4];
        out[0] = Integer.parseInt(m.group(2));
        out[1] = Integer.parseInt(m.group(3));
        out[2] = Integer.parseInt(m.group(4));
        out[3] = Integer.parseInt(m.group(5));
        return out;
    } else {
        return null;
    }
B.2 Additional Methods

Beyond the above objects, a variety of other objects and methods were written. A selection of them are included here.

B.2.1 Distances

Each graph type has a distance method. Alternatively, we could write a \textit{neighbors} method to determine if two vertices are adjacent, and do a breadth-first search to find the distance between two vertices. We could pre-compute the distance between any pair of vertices and store them in an array to prevent loss of time-efficiency. We recommend this procedure if many distance methods are to be made.

In this section, we include both distance methods and other methods that must be over-written.

B.2.1.1 Hexagonal Lattice

```java
public int distance(vertex v1, vertex v2, int width, int height, int layers) {
  int horizontal = mod(v2.getX() - v1.getX(), width); // horizontal distance, right from x1
  int vertical = mod(v2.getY() - v1.getY(), height); // vertical distance, down from y1
  boolean goUp = false; // initially assume path goes down
  boolean efficient = false;
  /*
   * The grid is bipartite; the distance between two points can be determined entirely by horizontal and vertical
   * differences if the points are in the same part or if the horizontal difference is at least as great as the vertical
   * difference. If the distance is nontrivial, there are two cases, which I denote efficient and non efficient.
   */
  if (horizontal > width/2) {
    horizontal = width - horizontal;
    // distance behaves the same going left or right, since the graph is horizontally symmetric from any point
  }
  if (vertical > height/2) {
    vertical = height - vertical;
    goUp = true; // shorter path goes up
  } else if (vertical == height/2) {
```
efficient = true; // up or down, one of the two paths is efficient
}
if (horizontal >= vertical){
    return horizontal + vertical;
} else if ((horizontal + vertical)%2 == 0){ //vertices are in the same part
    return 2*vertical;
} else {
    if (!(goUp ^ ((v1.getX()+v1.getY()%2 != 0))){
        efficient = true;
    }
    return efficient ? 2*vertical - 1 : 2*vertical + 1;
}

public int getTileDiameter() {
    return width > 2*height ? width : 2*height;
}

public symmetry[] symmetryList(int width, int height, int layers) { //assumes the
graph has alternating 1's
symmetry[] symList = new symmetry[width*height];
for(int x = 0; x < width; x++){
    for(int y = 0; y < height; y++){
        if ((x+y)%2 == 0) {
            symList[x + y*width] = new symmetry(x,y,false);
        } else {
            symList[x + y*width] = new symmetry((x+width-1)%width,y,true);
        }
    }
}
return symList;
}

B.2.1.2 Truncated Square Lattice

public int distance(vertex v1, vertex v2, int width, int height, int layers) { //
    TODO: more
    int x1 = v1.getX(), x2 = v2.getX(), y1 = v1.getY(), y2 = v2.getY();
    switch((2*(y1%2)+x1)%4) //four types of vertices for v1; using symmetry to
                        //reduce it to one type
        case 0://original
            break;
        case 1:
            x1 = width - 1 - x1; x2 = width - 1 - x2; //flip both vertical and horizontal
            y1 = height - 1 - y1; y2 = height - 1 - y2;
            break;
        case 2:
            y1 = height - 1 - y1; y2 = height - 1 - y2; //flip vertical
            break;
        case 3:
            x1 = width - 1 - x1; x2 = width - 1 - x2; //flip horizontal
            break;
    }
    int x = (x2 - x1 + width) % width;
    int y = (y2 - y1 + height) % height;
    //The graph has been translated so that one vertex is at (0,0) and the other at
    (x,y)
    int dist = 0;
```java
int h, w;
if (height / 2 < y) {
    h = height - y + 1;
    dist += height - y;
} else {
    h = y;
    dist += y;
}
if (width / 2 <= x) {
    w = width - x - 1;
    dist += width - x;
} else {
    w = x;
    dist += x;
}
int plus = 2*h - w > 3 ? ((int)((2*h - w)/4))*2 : 0;
return dist + plus;
}
public int getTileDiameter() {
    return width < 2*height ? width : 2*height;
}

B.2.1.3 Two-Layer Hexagonal Lattice

public int distance(vertex c1, vertex c2, int width, int height, int layers) {
    int horizontal = mod(c2.getX() - c1.getX(), width); //horizontal distance, right from x1
    int vertical = mod(c2.getY() - c1.getY(), height); //vertical distance, down from y1
    int zDir = (c2.getZ() > c1.getZ()) ? c2.getZ() - c1.getZ() : c1.getZ() - c2.getZ();
    boolean goUp = false; // initially assume path goes down
    boolean efficient = false;
    /*
    * The grid is bipartite; the distance between two points can be determined
    * entirely by horizontal and vertical
    * differences if the points are in the same part or if the horizontal
    * difference is at least as great as the vertical
    * difference. If the distance is nontrivial, there are two cases, which I
    * denote efficient and non efficient.
    */
    if (horizontal > width/2) {
        horizontal = width - horizontal;
        //distance behaves the same going left or right, since the graph is
        //horizontally symmetric from any point
    }
    if (vertical > height/2) {
        vertical = height - vertical;
        goUp = true; // shorter path goes up
    } else if (vertical == height/2) {
        efficient = true; // up or down, one of the two paths is efficient
    }
    if (horizontal >= vertical) {
        return horizontal + vertical + zDir;
    } else if (((horizontal + vertical) % 2 == 0)) { //vertices are in the same part
        return 2*vertical + zDir;
    }
```
```java
} else {
    if (!(goUp ^ ((c1.getX() + c1.getY()) % 2 != 0))) {
        efficient = true;
    }
    return efficient ? 2 * vertical - 1 + zDir : 2 * vertical + 1 + zDir;
}

public int getTileDiameter() {
    return width > 2 * height ? width : 2 * height;
}

public symmetry[] symmetryList(int width, int height, int layers) {
    // Assumes the graph has alternating 1's
    symmetry[] s = new symmetry[width * height];
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            boolean xFlip = ((i + j) % 2 == 1);
            int xShift = xFlip ? (j + 1) % width : j;
            int yShift = i;
            s[j + width * i] = new symmetry(xShift, yShift, xFlip);
        }
    }
    return s;
}

B.2.1.4 Offset Two-Layer Hexagonal Lattice

public int distance(vertex v1, vertex v2, int width, int height, int layers) {
    /* Each vertex is adjacent to the vertices horizontal to it.
    Vertex (0,0,0) is adjacent to the vertex directly below it.
    Vertex (1,0,0) is adjacent to the vertex directly above it, etc.
    Only half of the vertices are adjacent to those in the other layer. Vertex
    (0,0,0) is not.
    The second layer is offset for vertical adjacencies. */
    int x1 = v1.getX(), y1 = v1.getY(), z1 = v1.getZ();
    int x2 = v2.getX(), y2 = v2.getY(), z2 = v2.getZ();
    int horizontal = mod(x2 - x1, width); // horizontal distance, right from x1
    int vertical = mod(y2 - y1, height); // vertical distance, down from y1
    boolean goUp = false; // initially assume path goes down; use only when
    vertical != 0, height/2
    boolean efficient = false; // efficient if path goes down and has edge down from
    v1 (or up and up respectively)
    if (horizontal > width/2) {
        horizontal = width - horizontal;
        // distance behaves the same going left or right, since the graph is
        horizontally symmetric from any point
    }
    if (vertical > height/2) {
        vertical = height - vertical;
        goUp = true; // shorter path goes up
    } else if (vertical == height/2) {
        efficient = true; // up or down, one of the two paths is efficient
    }
    int sameLayerDistance;
    if (horizontal >= vertical){
```
sameLayerDistance = horizontal + vertical;

} else if(((horizontal + vertical)%2 == 0){ // vertex projections onto single layer are in the same part
sameLayerDistance = 2*vertical;
}
else {
if(!((goUp ) ^ ((x1+y1+z1)%2 != 0)){ // z1==1 flips vertical behavior of grid
efficient = true;
}
sameLayerDistance = efficient? 2*vertical − 1: 2*vertical + 1;
}

if(z1 == z2){
return sameLayerDistance;
}
else {
boolean even1 = (x1+y1)%2 == 0 ? true : false;
boolean even2 = (x2+y2)%2 == 0 ? true : false;
if(!even1 & & !even2){
return sameLayerDistance + 1;
}

if( vertical == height/2){
if(even1 & & even2 & & vertical > horizontal){
return sameLayerDistance − 1;
} else {
return sameLayerDistance + 1;
}
}

if( vertical > horizontal){
if((even2 && z1is0) & & (z1is0 ^ goUp)){
return sameLayerDistance − 1;
}

if( vertical >= horizontal){
if(((even2 && (!z1is0 ^ goUp) || vertical == 0){
return sameLayerDistance + 3;
}
return sameLayerDistance + 1;
}

public int getTileDiameter() {
return width < 2*height ? width : 2*height;
}

public symmetry[] symmetryList(){// assumes the graph has alternating 1's
symmetry[] s = new symmetry[width*height ];
for(int i = 0; i < height; i++) {
for(int j = 0; j < width; j++) {
boolean xFlip = ((i+j)%2 == 1);
int xShift = xFlip ? (j+1)%width : j;
int yShift = i;
s[j + width*i] = new symmetry(xShift , yShift , xFlip);
}
}
return s;
}
B.2.2 Graph Experiments

In this section, we include necessary methods to run both `singleColorExperiment` and `expansionColorExperiment`; methods that were used directly from the `main` method to generate colorings.

```java
/**
 * Runs graphColorerLocal for the graphs in a file for a given duration.
 * @param graphType A graph of the type used for the experiment.
 * @param color The color class assigned in the experiment.
 * @param inGraphPrefix The prefix for the files to scan before choosing a file to input.
 * @param outGraphPrefix The prefix for the file used to record new graphs.
 * @param widthExp Copies of the input graph to be placed next to each other horizontally.
 * @param heightExp Copies of the input graph to be placed next to each other vertically.
 * @param numMinutes Number of minutes to run the experiment.
 * @param crossCheck True to cross-check a graph with other graphs in the series. Use True for small graphs, and False for large graphs to save time.
 * @param largeGraphs Should be True. The other method is deprecated.
 * @throws IOException
 */
public static void singleColorExperiment(Graph graphType, int color, String inGraphPrefix, String outGraphPrefix, int widthExp, int heightExp, int numMinutes, boolean crossCheck, boolean largeGraphs) throws IOException{
    graphTest.printDate(true);
    File candidate = graphTest.colorExpGetBestFile(color - 1, inGraphPrefix);
    Graph[] graphs = fileReader.readDoc(candidate, graphType);
    graphs[0].printProgress(color - 1);
    if (widthExp != 1 || heightExp != 1) {
        for (int i = 0; i < graphs.length; i++) {
            graphs[i] = graph.expandGraph(graphs[i], heightExp, widthExp, 1);
        }
    }
    System.out.println("Generating graphs for color \"+");
    } }
```

```java
/**
 * Runs an expansion coloring experiment. Colors the graph for varying expansion (tile enlarging) settings, and prints the results.
 * @param graphType A graph of the type used for the experiment.
 * @param color The color class assigned in the experiment.
 * @param graphName The prefix for the input files.
 * @param testName The prefix for the output files. There will be one for each pair of dimensions.
 * @param widthMaxExp The range for width expansion. Should be small (<10).
 * @param heightMaxExp The range for height expansion. Should be small (<10).
 * @param numMinutesExpTest Number of minutes to run each pair of dimensions in the test.
 */
```
* @param crossCheck True to cross-check new graphs with previous graphs. Should be true for small graphs, and false for large graphs.
* @param largeGraphs Should be True. The other method is deprecated.
* @return An integer array with the best expansion results. [1] for width, [0] for height.
* @throws IOException
*/
public static int[] expansionColorExperiment(graph graphType, int color, String graphName, String testName, int widthMaxExp, int heightMaxExp, int numMinutesExpandTest, boolean crossCheck, boolean largeGraphs) throws IOException{
    File candidate = graphTest.colorExpGetBestFile(color-1, graphName);
    graph[] originalGraphs = fileReader.readDoc(candidate, graphType);
    graph[] graphs;
    double originalDensity = 0;
    for(graph g : originalGraphs){
        double temp = g.density();
        originalDensity = temp > originalDensity? temp : originalDensity;
    }

double startDensity = 0;
try{
    File f = graphTest.colorExpGetBestFile(color, testName);
    System.out.println(f.getName());
    startDensity = graphColorer.findMaxDensity(f);
} catch(Exception e){
    startDensity = 0;
}
}

graphTest.printDate(false);
System.out.println(" Start density: "+startDensity);
System.out.println("Running expansion test for color "+color+". Generating graphs.");
double[][] densities = new double[heightMaxExp][widthMaxExp];
for(int x = 1; x <= widthMaxExp; x++){
    for(int y = 1; y <= heightMaxExp; y++){
        graphs = null;
        System.gc();
        graphTest.printDate(false);
        System.out.println(" Running test "+y4"x"+x4;");
        graphs = graph.expandGraphs(originalGraphs, y, x, 1);
        try{
            densities[y-1][x-1] = graphTest.generateRanColorings(graphs, color, testName, startDensity, numMinutesExpandTest, crossCheck, largeGraphs);
        } catch(OutOfMemoryError e){
            System.out.println("Out of memory. Skipping test.");
        }
    }
}

graphTest.printDate(true);
int[] best = {0,0};
double bestSoFar = 0;
System.out.println("Results:");
for(int x = 1; x <= widthMaxExp; x++){
    for(int y = 1; y <= heightMaxExp; y++){
        double temp = densities[y-1][x-1] - originalDensity;
        System.out.println(y+"x"+x+": "+densities[y-1][x-1]+"(\"+temp\")");
    }
}
if \( \text{densities}[y-1][x-1] > \text{bestSoFar} + \text{graphTest.EPSILON} \) || \( \text{densities}[y-1][x-1] >= \text{bestSoFar} - \text{graphTest.EPSILON} \&\& y*x < \text{best}[0]*\text{best}[1]) \{
    \text{bestSoFar} = \text{densities}[y-1][x-1];
    \text{best}[0] = y;
    \text{best}[1] = x;
\}

System.out.println("Best: "+\text{best}[0]+"x"+\text{best}[1]);
return \text{best};

// Part of the graphTest object
/**
 * Generates random colorings from an array of graphs for a given duration.
 * @param graphs An array of graphs
 * @param color The new color to be added
 * @param graphName The prefix for the current graph family
 * @param density Initial max density. Can be 0.
 * @param numMinutes number of minutes the method should run for
 * @param crossCheck True if cross-checking symmetries (increases generation time)
 * @param largeGraphs Should be set to true. The other method is deprecated.
 * @throws IOException
 */
public static double generateRanColorings(graph[] graphs, int color, String graphName, double density, int numMinutes, boolean crossCheck, boolean largeGraphs) throws IOException{
    double returnDensity = 0;
    long startTime = System.currentTimeMillis();
    boolean atDuration = false;
    int numTrialsEach = 1;
    int numIterations = 0;
    graphColorerLocalRandom[] gclrsl = new graphColorerLocalRandom[graphs.length];
    graphColorerRandom[] gcrsl = new graphColorerRandom[graphs.length];
    if(largeGraphs){
        double startDensity = graph.findMaxDensity(graphs);
        graphColorerLocalRandom.initializeDensityVars(graphName, color, startDensity, graphs[0].getHeight(), graphs[0].getWidth(), graphs[0].getLayers());
        for(int i = 0; i < graphs.length; i++){
            gclrsl[i] = new graphColorerLocalRandom(graphs[i], graphName, color, density);
        }
    } else {
        for(int i = 0; i < graphs.length; i++){
            gcrsl[i] = new graphColorerRandom(graphs[i], graphName, color);
        }
    }
    startTime = System.currentTimeMillis(); // resetting start time since generating the gclrsl/gcrsl may take a minute
    while(!atDuration){
        numIterations ++;
        long currentDuration = System.currentTimeMillis() - startTime;
        if(currentDuration/60000 >= numMinutes){
            atDuration = true;
        }
    }
    return returnDensity;
}
```java
int progress = 0;
for (int i = 0; i < graphs.length; i++) {
    int temp;
    if (!largeGraphs) {
        //TODO: Only instantiate a single graphColorer object...
        temp = graphColorerRandom.generateColorings(gcfs[i], numTrialsEach, crossCheck);
    } else {
        temp = graphColorerLocalRandom.generateColorings(gcfs[i], numTrialsEach);
    }
    progress = (temp > progress) ? temp : progress;
}
String progressString = ".";
switch (progress) {
    case 0:
        progressString = ".";
        break;
    case 1:
        progressString = "+";
        break;
    case 2:
        progressString = "+!";
        break;
    default:
        progressString = "+^";
        break;
}
System.out.println(progressString);

if (numIterations % 100 == 0) { // Increase number of trials per iteration
    if (numTrialsEach == 1) {
        numTrialsEach = 10;
    } else if (numTrialsEach < 1000) {
        numTrialsEach += 10;
    }
}

if (numIterations % 20 == 0) {
    System.out.println(" ");
    if (numIterations % 100 == 0 || atDuration) { // Print progress density
        graph tg = graphs[0];
        File outFile = new File(graphColorer.directory + "\"" + graphName + "\"" + color + "\"" + tG.getHeight() + "\"" + tG.getWidth() + "\"" + tG.getLayers() + "\"" + outFile);
        double startDensity = tG.density();
        double endDensity = graphColorer.findMaxDensity(outFile);
        double diff = endDensity - startDensity;
        System.out.println(endDensity + " (" + diff + ") " + graphColorer.countGraphs(outFile, 0) + " graphs")
        returnDensity = endDensity;
    }
    return returnDensity;
}

// Part of the graphTest object
```
/**  
* From the working directory, scans all files with the given prefix and color  
* specification and finds the file with the graph that has the maximum density.  
* @param color An integer designating a color.  
* @param graphName A String designating a file prefix.  
* @return A File.  
*/  
public static File colorExpGetBestFile(int color, String graphName){  
    ArrayList<File> inFiles = new ArrayList<File>();  
    File directory = new File(graphColorer.directory);  
    for(File f : directory.listFiles()){  
        if(f.getName().startsWith(graphName + "C"+color+"D")){  
            inFiles.add(f);  
        }  
    }  
    double bestDensity = 0;  
    if(inFiles.size() == 0){  
        return null;  
    }  
    File candidate = inFiles.get(0);  
    int[] canDim = fileReader.parseName(candidate);  
    for(File f : inFiles){  
        double temp = graphColorer.findMaxDensity(f);  
        if(temp > bestDensity+EPSILON){  
            candidate = f;  
            bestDensity = temp;  
            canDim = fileReader.parseName(f);  
        } else if(temp >= bestDensity - EPSILON){  
            int[] tempDim = fileReader.parseName(f);  
            if(tempDim[1]*tempDim[2]*tempDim[3] < canDim[1]*canDim[2]*canDim[3]){  
                candidate = f;  
                bestDensity = temp;  
                canDim = tempDim;  
            }  
        }  
    }  
    return candidate;  
}  

// Part of the graphTest object  
/**  
* Prints the current date and time  
* @param newLine Set TRUE to start a new line after printing the date  
*/  
public static void printDate(boolean newLine){  
    DateFormat dateFormat = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss");  
    Date date = new Date();  
    if(newLine){  
        System.out.println(dateFormat.format(date));  
    } else {  
        System.out.print(dateFormat.format(date));  
    }  
}
BIBLIOGRAPHY


