creative commons
COMMONS DEED

저작자표시-비영리-동일조건변경허락 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.
- 이차적 저작물을 작성할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

동일조건변경허락. 귀하가 이 저작물을 개작, 변형 또는 가공했을 경우에는, 이 저작물과 동일한 이용허락조건하에서만 배포할 수 있습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

이학석사 학위논문

Vertex coloring of plane graphs

(평면 그래프 색칠하기)

2014년 2월

서울대학교 대학원

수리과학부

최 호 석

Vertex coloring of plane graphs

(평면 그래프 색칠하기)

지도교수 김 홍 종

이 논문을 이학석사 학위논문으로 제출함

2013년 10월

서울대학교 대학원

수리과학부

최 호 석

최 호 석의 이학석사 학위논문을 인준함

2013년 12월

위 원 장	국 웅	(인)
부 위 원 장	김 홍 종	(인)
위 원	김 서 령	(인)

Vertex coloring of plane graphs

A dissertation
submitted in partial fulfillment
of the requirements for the degree of
Master of Science
to the faculty of the Graduate School of
Seoul National University

by

Ho-Seok Choe

Dissertation Director : Professor Hong-Jong Kim

Department of Mathematical Sciences
Seoul National University

February 2014

© 2014 Ho-Seok Choe

All rights reserved.

Abstract

Vertex coloring of plane graphs

Ho-Seok Choe

Department of Mathematical Sciences
The Graduate School
Seoul National University

The *four color theorem* states that only four colors are needed to color the regions of any simple planar map so that any two adjacent regions have different colors. This theorem can be interpreted as finding a vertex coloring of plane graphs. This thesis suggests a method to find a vertex coloring of plane graphs with four available colors that includes the following steps:

(i) Convert the given map to a graph and find a maximal plane graph that contains the graph.
(ii) Remove vertices of degree 3 from the maximal plane graph if they exist.
(iii) Find a *hubset* that is a set of independent hubs of wheels.
(iv) Color the vertices that are not the elements of the hubset with three available colors.
(v) Color the vertices contained in the hubset with the fourth color.
(vi) Finally, apply the coloring result to the given map.

Using this process, we obtained a 98% success rate in computer experiments for random graphs of order 40. We will discuss how to improve the coloring process.

Key words: vertex coloring, plane graph, four color, 4-color
Student Number: 2009-20284

Contents

List of Figures

LIST OF FIGURES

Chapter 1

Introduction

1.1 What is the Four Color Theorem?

Suppose that we are going to make a (geographic) map of our nation or a blueprint of our house. We would draw several lines to distinguish each region, and color the regions to increase the visibility of our work. If we are only interested in distinguishing the regions, at least how many colors do we need? In other words, suppose that two regions have different colors if they share a line as their boundary. Then, at least how many colors do we need? The conjecture that only four colors are needed to complete such a coloring task for an arbitrary map was first proposed in 1852 by Francis Guthrie [3]. At long last, Kenneth Appel and Wolfgang Haken proved the four color theorem using a computer in 1976, and their proof was improved in 1996 by Neil Robertson, Daniel Sanders, Paul Seymour, and Robin Thomas [7].

A *region* is a connected open subset of the plane. A *planar map* is a set of disjoint regions of the plane. A point is a *corner* of a map if it is contained in the closures of at least three regions. Two regions of a map are *adjacent* if their closures share a point that is not a corner. The four color theorem asserts that only four colors are needed to color the regions of any simple planar map so that any two adjacent regions have different colors [3].[1]

[1] [3] defines a *planar map* as a set of pairwise disjoint subsets called regions of the plane,

1.2 Generalization of the Four Color Theorem

People may wonder where the number "4" comes from. And the dimensions, the topological structures, or some properties of the spaces containing the given map would probably cross their minds. Here are some examples that help us check our conjectures.

Example 1.2.1.

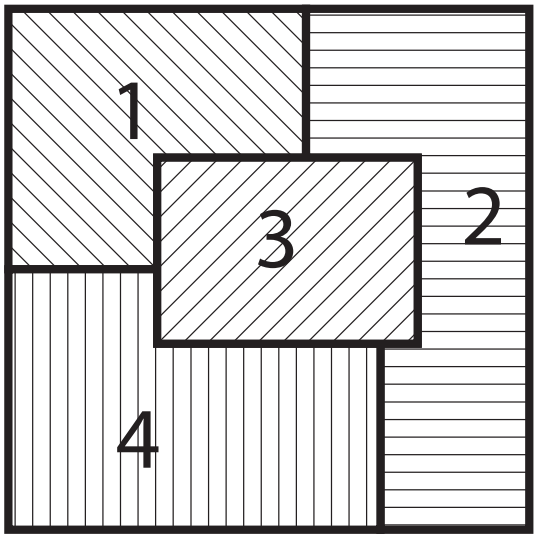

Figure 1.2.1: Map that requires four colors to color the regions

In Figure 1.2.1, we show a map containing four regions (five regions if we count the unbounded region) on a plane. All the regions are pairwise adjacent. Therefore, at least four colors are required to color this map.

and a *simple planar map* as a planar map whose regions are connected open sets.

Example 1.2.2.

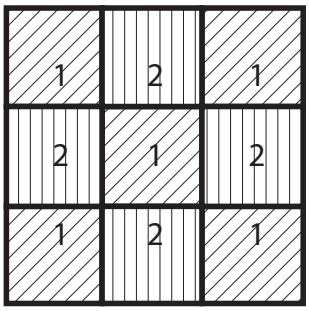

Figure 1.2.2: Chessboard

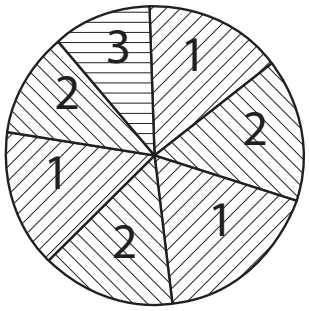

Figure 1.2.3: Pizza with odd pieces

Figure 1.2.2 and Figure 1.2.3 show maps that require less than 4 colors since they have specific structures.

What if the spaces containing the maps are not two dimensional?

Example 1.2.3.

Figure 1.2.4: an interval of the real line

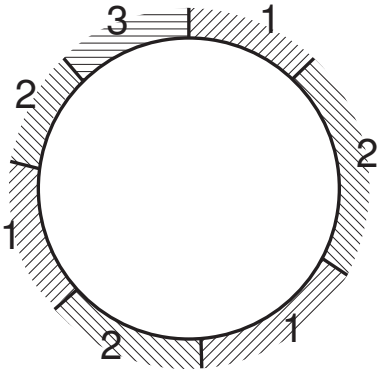

Figure 1.2.5: a circle

Figure 1.2.4 shows an interval of the real line $\mathbb{R}$ and It does not contain a corner since any point p of $\mathbb{R}$ separates $\mathbb{R} \setminus \{p\}$ into two distinct partitions ($\{x \in \mathbb{R} : x < p\}$ and $\{x \in \mathbb{R} : x > p\}$). Therefore, if two regions A and B of $\mathbb{R}$ share a point p as a common frontier and A has been assigned a color, say color 1, then B can be assigned another color, say color 2. By repeating this

work for another frontier of the sum of colored regions, we can color all the regions with only two colors. Figure 1.2.5 shows a circle that can be viewed as joining two ends of a bounded interval containing an odd number of regions. The map in this figure requires three colors since the ends of the interval are joined: The first region and the last region are not adjacent before joining the ends, but they are adjacent after joining the ends. Compare Figure 1.2.5 with Figure 1.2.3.

Example 1.2.4.

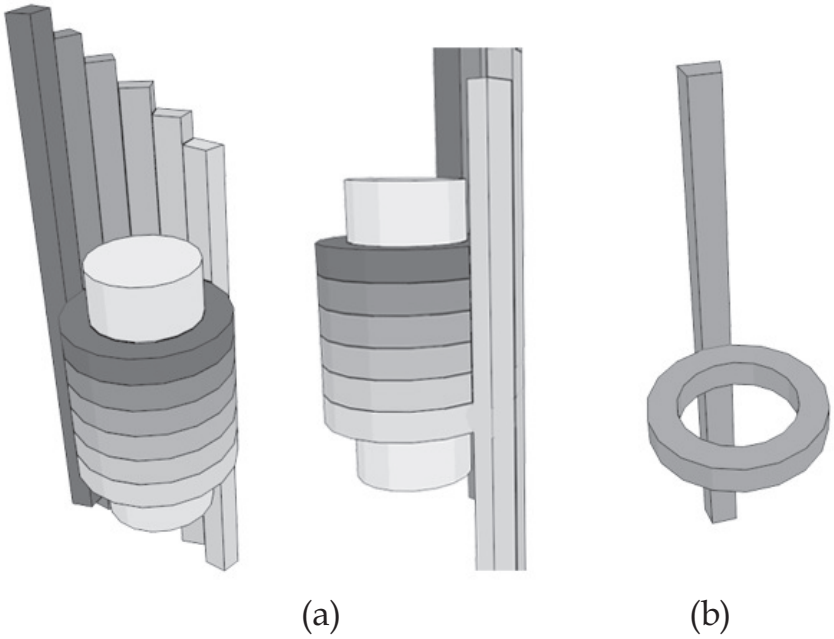

Figure 1.2.6: regions of 3-dimensional space

In Figure 1.2.6, (a) shows a map with seven regions (containing a cylindrical region) of 3-dimensional space. Most regions have shapes such as a sum of a tube and a stick like (b). This map requires seven colors since all the regions are pairwise adjacent. From this figure, we can guess the existence of maps that require infinitely many colors: by adjusting some parameters (width of stick, height of tube, and so on) of each region, and adding more regions of similar shape, we can make a map that requires as many colors as we want.

As the previous example shows, it seems pointless to consider this type of problem for maps of three or higher dimensional spaces. Instead, let us consider maps of 2-dimensional surfaces embedded into a 3-dimensional space. The next example shows that the coloring problems are more complicated and that the topological structures presumably influence the number of required colors.

Example 1.2.5.

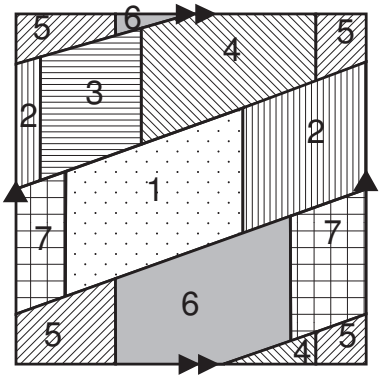

Figure 1.2.7: a map of torus requiring seven colors

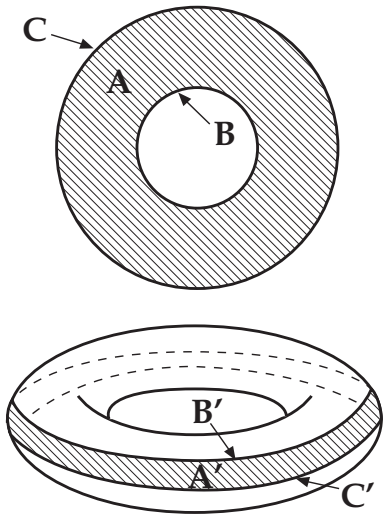

Figure 1.2.8: annulus in the plane and annulus in a torus

Figure 1.2.7 shows a map of a torus requiring seven colors. As we obtain a new adjacency condition by joining the ends of an interval (a region of the real line) in Example 1.2.3, we may get new adjacency conditions by pasting the opposite edges of a square (a region of the plane). In Figure 1.2.8, A is an annular region with its two frontiers B and C in a plane. A' is also an annular region with its two frontiers B' and C' in a torus. A separates its exterior into two pieces but A' does not. If there was a regions whose frontier intersects with B and another region whose frontier intersects with C in the plane, these two regions would never be adjacent. However, if there was a region whose frontier

intersects with B' and another region whose frontier intersects with C' in the torus, these two regions may be adjacent. Considering the fact that a cycle (or a polygon) is a deformation retract of an annulus [6], the plane case agrees with the Jordan curve theorem (Theorem 2.2.1), but the torus case does not.

1.3 Maps with finite regions

In this thesis, we treat maps that contain *finite regions* of the *plane*. However, a map may have two or more unbounded regions and we may want to find another map that conserves the adjacency of all the pairs of regions and has only one unbounded region.

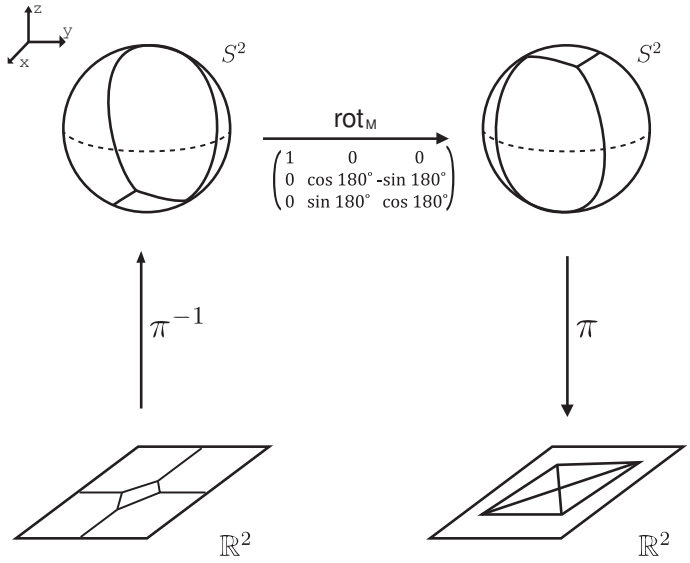

Figure 1.3.1: Example of getting a map that has only one unbounded region from another map that has four unbounded regions.

Recall the well-known stereographic projection $\pi : S^2 \setminus N \to \mathbb{R}^2$ where S^2 is a unit sphere, $N = (0, 0, 1)$ is the north pole of the sphere, and $\mathbb{R}^2$ is the plane isomorphic to the plane $\{(x, y, z)|z = 0\}$ in three-dimensional space [5]. With the help of this projection, we can find the wanted map. Assume that

a map M of the plane has two or more unbounded regions, say $R_1, R_2, \cdots R_n$. Considering that $\pi^{-1}(R_i)$ and $\pi^{-1}(R_j)$ are distinguished in S^2 $(1 \leq i \neq j \leq n)$, it is reasonable to treat the north pole N as a point contained in frontier of every $\pi^{-1}(R_k)$, where $k = 1, 2, \cdots, n$. Let $\mathrm{rot}_M : S^2 \to S^2$ be a rotation of S^2 such that the north pole N is contained in $\mathrm{rot}_M(\pi^{-1}(R))$ for some $R \in M$. Let us remove N and re-send regions of $\mathrm{rot}_M(\pi^{-1}(M))$ to the plane by the projection π so that we obtain a new map of the plane that contains only one unbounded region. This method does not break the adjacency of the regions since the projection π and the rotation rot_M are continuous and bijective. If we want a region with special properties to be unbounded, we can use a similar method.

Chapter 2

Graph representation

This chapter introduces the standard terminologies and some well-known facts from [2].

2.1 Graphs

A *graph* is a pair $G = (V, E)$ of sets such that $E \subseteq [V]^2$. The elements of V are the *vertices* of the graph G, the elements of E are its *edges*. The usual way to picture a graph is by drawing a dot for each vertex and joining two of these dots by a line if the corresponding two vertices form an edge.

A graph with vertex set V is said to be a graph *on* V. The vertex set of a graph G is referred to as $V(G)$, its edge set as $E(G)$. We shall not always distinguish strictly between a graph and its vertex or edge set. For example, we may speak of a vertex $v \in G$ (rather than $v \in V(G)$), and edge $e \in G$, and so on.

The number of vertices of a graph G is its *order* written as $|G|$, and the number of its edges is denoted by $\|G\|$.

A vertex v is *incident* with an edge e if $v \in e$. And e is an edge *at* v. The two vertices incident with an edge are its *endvertices* or *ends*, and an edge *joins* its ends. An edge $\{x, y\}$ is usually written as xy (or yx). If $x \in X$ and $y \in Y$, then xy is an $X - Y$ *edge*. The set of all $X - Y$ edges in a set E

is denoted by $E(X, Y)$; instead of $E(\{x\}, Y)$ and $E(X, \{y\})$ we simply write $E(x, Y)$ and $E(X, y)$. The set of all the edges in E at a vertex v is denoted by $E(v)$.

Two vertices x, y of G are *adjacent* or *neighbors*, if xy is an edge of G. Also, two edges $e \neq f$ are adjacent if they have an end in common. If all the vertices of G are pairwise adjacent, then G is *complete*. A complete graph on n vertices is a K^n. K^3 is called a *triangle*. Pairwise non-adjacent vertices or edges are called *independent*. More formally, a set of vertices or of edges is *independent* (or *stable*) if no two of its elements are adjacent.

Let $G = (V, E)$ and $G' = (V', E')$ be two graphs. We call G and G' *isomorphic*, and write $G \simeq G'$ if there exists a bijection $\varphi : V \to V'$ with $xy \in E \Leftrightarrow \varphi(x)\varphi(y) \in E'$ for all $x, y \in V$. Such a bijection φ is called an *isomorphism*.

A class of graphs that is closed under isomorphism is called a *graph property*. For example, 'containing of triangle' is a graph property.

We set $G \cup G' := (V \cup V', E \cup E')$ and $G \cap G' := (V \cap V', E \cap E')$. If $V' \subseteq V$ and $E' \subseteq E$ then G' is a *subgraph* of G (and G a *supergraph* of G'), written as $G' \subseteq G$. If $G' \subseteq G$ and G' contains all the edges $xy \in E$ with $x, y \in V'$, then G' is an *induced subgraph* of G and we say that V' *induces* or *spans* G' in G, write $G' =: G[V']$.

If U is any set of vertices (usually of G), we write $G - U$ for $G[V \setminus U]$. If $U = \{v\}$ is a singleton, we write $G - v$ rather than $G - \{v\}$. Instead of $G - V(G')$ we simply write $G - G'$. For a subset F of $[V]^2$ we write $G - F := (V, E \setminus F)$ and $G + F := (V, E \cup F)$. As above, $G - \{e\}$ and $G + \{e\}$ are abbreviated to $G - e$ and $G + e$. We call G *edge-maximal* with a given graph property if G itself has the property but no graph $G + xy := (V, E \cup \{xy\})$ does, for non-adjacent vertices $x, y \in G$.

The set of neighbors of a vertex v in G is denoted by $N_G(v)$ or briefly by $N(v)$. More generally for $U \subseteq V$, the neighbors in $V \setminus U$ of vertices in U are called *neighbors of* U, denoted by $N(U)$.

The *degree* $d_G(v) = d(v)$ of a vertex v is the number $|E(v)|$. The number $\delta(G) := \min\{d(v) | v \in V\}$ is the *minimum degree* of G, the number $\Delta(G) :=$

$\max\{d(v)|v \in V\}$ is its *maximum degree*. The number

$$d(G) := \frac{1}{|V|}\sum_{v\in V} d(v)$$

is the *average degree* of G. If we sum up all the vertex degrees in G we count every edge exactly twice: once from each of its ends. Thus

$$|E| = \frac{1}{2}d(G)\cdot|V|. \tag{2.1.0.1}$$

A *path* is a non-empty graph $P = (V, E)$ of the form

$$V = \{x_0, x_1, \cdots, x_k\},\ E = \{x_0x_1, x_1x_2, \cdots, x_{k-1}x_k\}$$

where the x_i are all distinct. The vertices x_0 and x_k are *linked* by P and are called its *ends*. The vertices $x_1, \cdots x_{k-1}$ are the *inner* vertices of P. The number of edges of a path is its *length*. We often refer to a path by the natural sequence of its vertices, writing $P = x_0x_1\cdots x_k$ and calling P a path *from* x_0 *to* x_k (as well as *between* x_0 and x_k).

Given sets A, B of vertices, we call $P = x_0\cdots x_k$ an $A - B$ *path* if $V(P)\cap A = \{x_0\}$ and $V(P)\cap B = \{x_k\}$. As before, we write $a - B$ path rather than $\{a\} - B$ path, etc.

If $P = x_0x_1\cdots x_{k-1}$ is a path and $k \geq 3$, then the graph $C := P + x_{k-1}x_0$ is called a *cycle*. As with paths, we often denote a cycle by its (cyclic) sequence of vertices such as $C = x_0, \cdots, x_{k-1}, x_0$. The *length* of a cycle is its number of edges (or vertices). An edge that joins two vertices of a cycle but is not itself an edge of the cycle is a *chord* of that cycle. An *induced cycle* in G, a cycle in G forming an induced subgraph, is one that has no chords.

A non-empty graph G is called *connected* if any two of its vertices are linked by a path in G. A maximal connected subgraph of G is called a *component* of G.

An acyclic graph, one not containing any cycles, is called a *forest*. A connected forest is called a *tree*. The vertices of degree 1 in a tree are its *leaves*.

Let $r \geq 2$ be an integer. A graph $G = (V, E)$ is called *r-partite* if V admits a partition into r classes such that every edge has its ends in different classes. Vertices in the same partition class must not be adjacent. Instead of '2-partite' one usually says *bipartite*. An r-partite graph in which every two vertices from different partition classes are adjacent is called *complete multipartite(r-partite) graph* and denoted by $K_{n_1,\cdots,n_k}$ where each n_i is the number of vertices of each partition class. If $n_1 = \cdots = n_r =: s$ we abbreviate this to K_s^r.

Let $e = xy$ be an edge of a graph $G = (V, E)$. By G/e we denote the graph (V', E') with vertex set $V' := (V \setminus \{x, y\}) \cup \{v_e\}$ and edge set

$$E' := \{vw \in E | \{v, w\} \cap \{x, y\} = \emptyset\} \cup \{v_e w | xw \in E \setminus \{e\} \text{or } yw \in E \setminus \{e\}\}.$$

More generally, if X is another graph and $\{V_x | x \in V(X)\}$ is a partition of V into connected subsets such that, for any two vertices $x, y \in X$, there is a $V_x - V_y$ edge in G if and only if $xy \in E(X)$, we call G an MX and write $G = MX$. The sets V_x are the *branch sets* of this MX. If $G = MX$ is a subgraph of another graph Y, we call X a *minor* of Y and write $X \preccurlyeq Y$.

If we replace the edges of X with independent paths between their ends (so that none of these paths has an inner vertex on another path or in X), we call the graph G obtained a *subdivision* of X and write $G = TX$. If $G = TX$ is the subgraph of another graph Y, then X is a *topological minor* of Y.

2.2 Planar graphs

A *straight line segment* in the Euclidean plane is a subset of $\mathbb{R}^2$ that has the form $\{p + \lambda(q - p) | 0 \leq \lambda \leq 1\}$ for distinct points $p, q \in \mathbb{R}^2$. A *polygon* is a subset of $\mathbb{R}^2$ that is the union of finitely many straight line segments and is homeomorphic to the unit circle S^1. A *polygonal arc*, or simply an *arc*, is a subset of $\mathbb{R}^2$ which is the union of finitely many straight line segments and is homeomorphic to the closed unit interval $[0, 1]$. The images of 0 and of 1 under such a homeomorphism are the *endpoints* of this polygonal arc, which *links* them and runs *between* them. If P is an arc between x and y, the point set $P \setminus \{x, y\}$ is the *interior* of P and we denote it by $\mathring{P}$.

Let $O \subseteq \mathbb{R}^2$ be an open set. Being linked by an arc in O defines an equivalence relation on O. The corresponding equivalence classes are again open and they are the *regions* of O. The *frontier* of a set $X \subseteq \mathbb{R}^2$ is the set Y of all points $y \in \mathbb{R}^2$ such that every neighborhood of y meets both X and $\mathbb{R}^2 \setminus X$.

Theorem 2.2.1 (Jordan Curve Theorem for Polygons)**.** For every polygon $P \subseteq \mathbb{R}^2$, the set $\mathbb{R}^2 \setminus P$ has exactly two regions. Each of these has the entire polygon P as its frontier.

A good account of the Jordan curve theorem is given in [8] or [6].

Lemma 2.2.2. Let P_1, P_2, P_3 be three arcs, between the same two endpoint but otherwise disjoint.

(i) $\mathbb{R}^2 \setminus (P_1 \cup P_2 \cup P_3)$ has exactly three regions, with frontiers $P_1 \cup P_2$, $P_2 \cup P_3$ and $P_3 \cup P_1$.

(ii) If P is an arc between an inner point of P_1 and an inner point of P_3, whose interior lies in the region of $\mathbb{R}^2 \setminus (P_1 \cup P_3)$ that contains P_2, then $\mathring{P} \cap \mathring{P}_2 \neq \emptyset$.

A *plane graph* is a pair (V, E) of finite sets with the following properties (the elements of V are again called *vertices*, those of E *edges*):

(i) $V \subseteq \mathbb{R}^2$;

(ii) every edge is an arc between two vertices;

(iii) different edges have different sets of endpoints;

(iv) the interior of an edge contains no vertex and no point of any other edge.

For every plane graph G, the set $\mathbb{R}^2 \setminus G$ is open. Its regions are the *faces* of G. We denote the set of faces of G by $F(G)$.

The subgraph of G whose point set is the frontier of a face f is said to *bound* f and is called its *boundary*, and we denote it by $G[f]$. A face is said to be *incident* with the vertices and edges of its boundary.

A plane graph G is called *maximally plane*, or just *maximal*, if we cannot add a new edge to form a plane graph $G' \supsetneq G$ with $V(G') = V(G)$. We call G

a plane *triangulation* if every face of G (including the outer face) is bounded by a triangle.

A *wheel* W_n is a graph with n vertices ($n \geq 4$) formed by connecting a single vertex v to all vertices of an $(n-1)$-cycle. The single vertex v is *hub* and the edges incident with v are called *spokes*. We simply denote it by W when we do not need to consider the length of the cycle. If W is a wheel in a graph G and h is its hub, we say that h *forms a wheel* W in G.

Proposition 2.2.3. A plane graph of order at least 3 is maximally plane if and only if it is a plane triangulation.

Theorem 2.2.4 (Euler's Formula)**.** Let G be a connected plane graph with n vertices, m edges, and l faces. Then

$$n - m + l = 2. \tag{2.2.4.1}$$

Corollary 2.2.5. A plane graph with $n \geq 3$ vertices has at most $3n-6$ edges. Every plane triangulation with n vertices has $3n - 6$ edges.

An *embedding* in the plane, or *planar embedding*, of an (abstract) graph G is an isomorphism between G and a plane graph H. The latter will be called a *drawing* of G. A graph is called *planar* if it can be embedded in the plane. A planar graph is *maximal*, or *maximally planar*, if it is planar but cannot be extended to a larger planar graph by adding an edge (but no vertex).

Proposition 2.2.6.

(i) Every maximal plane graph is maximally planar.
(ii) A planar graph with $n \geq 3$ vertices is maximally planar if and only if it has $3n - 6$ edges.

Theorem 2.2.7 (Kuratowski 1930; Wagner 1937)**.** The following assertions are equivalent for graphs G:

(i) G is planar;
(ii) G contains neither K^5 nor $K_{3,3}$ as a minor;
(iii) G contains neither K^5 nor $K_{3,3}$ as a topological minor.

2.3 Plane graph representation of a map

For a given map $V = \{v_1, v_2, \cdots, v_n\}$ of plane ($n \geq 4$), let us define $E := \{vw \subset V | v, w \text{ are adjacent }\}$, then $G = (V, E)$ forms a graph on V. Two adjacent regions (their closures share a point that is not a corner) in V are also adjacent (they are joined by an edge) in G. The graph G is planar since G has a drawing H in the following sense: let $V' := \{p_1, p_2, \cdots, p_n\}$ be a set of inner points of regions of the plane such that $p_i \in v_i$ for each $i = 1, 2, \cdots, n$. For an adjacent pair of regions, v_i and v_j, there exists a point q that is not a corner on a common frontier of v_i and v_j. Let $P_{i,q}$ be an arc from p_1 to q lying in v_1, and $P_{q,j}$ be an arc from q to p_2 lying in v_2. Let $P_{i,j} := P_{i,q} \cup P_{q,j}$ be the sum of the arcs and E' be the set of such $P_{i,j}$. Let us define a drawing H on V', setting $H := (V', E')$. Then, we can find (graph) isomorphism $\phi : V \to V'$ such that $\phi(v_i) = p_i$ and $\phi(v_i v_j) = P_{i,j}$. Therefore, we can convert the coloring problem of a map of the plane to the coloring problem of vertices of a plane graph.

A *vertex coloring* of a graph $G = (V, E)$ is a function $c : V \to S$ from the vertex set V to a set S such that $c(v) \neq c(w)$ whenever v and w are adjacent. The elements of the set S are called the *available colors*. The smallest integer k is the *(vertex-) chromatic number* of G if G has a *k-coloring* that is a vertex coloring $c : V \to \{1, \cdots, k\}$. We denote the chromatic number by $\chi(G)$. A graph G with $\chi(G) = k$ is called *k-chromatic*; if $\chi(G) \leq k$, we call G *k-colorable* [2]. We also call a vertex coloring with $|S| \leq n$ an *n-coloring solution* of G. Clearly, we are looking for the 4-coloring solutions.

Suppose that G and $G + xy$ are plane graphs such that $x, y \in G$ are non-adjacent vertices. Then all the coloring solutions of $G+xy$ are also the coloring solutions of G. Therefore, we have only to consider a maximal plane graph containing the given graph G. By Proposition 2.2.6, we shall not distinguish between maximal plane graphs and maximal planar graphs. And by Proposition 2.2.3, all the faces, including unbounded faces, of maximal plane graphs are bounded by triangles.

Proposition 2.3.1. Let G be a maximal plane graph with n vertices. The minimum degree $\delta(G)$ of G is at least 3.

Proof. By Corollary 2.2.5, G has $3n-6$ edges. A graph G' obtained by removing vertices or edges from G is also a plane graph. Suppose that there is a vertex v of degree 2. Set $G' := G - v$. Then the order $|G'|$ of G' is $n-1$ and the number $\|G'\|$ of its edges is $3n-6-2 = 3n-8$ since the degree of v is 2. However, $\|G'\| = 3n-8 > 3(n-1)-6$ and by Corollary 2.2.5, this contradicts the fact that G' is a plane graph. □

Proposition 2.3.2. Let G be a maximal plane graph with n vertices and $v \in G$ be a vertex of degree 3. Then the graph $G' := G - v$ is also a maximal plane.

Proof. The number of edges of G' is $(3n-6)-3 = 3(n-1)-6$ and by Corollary 2.2.5, G' is maximally plane. □

Suppose that G has a vertex v of degree 3. Let v_1, v_2, v_3 be the neighbors of v. If we can find a 4-coloring solution $c : V(G-v) \to S$ of $G-v$, then we also can find a 4-coloring solution of G by extending c: define $c(v) := s$ where $s \in S \setminus c(\{v_1, v_2, v_3\})$. By Proposition 2.3.2, $G-v$ is also maximally plane. Therefore, we have only to consider a maximal plane graph with a minimum degree at least 4.

Applying Equation 2.1.0.1 to a maximal plane graph G of order n with $\delta(G) \geq 4$, we could easily obtain that

$$\begin{aligned} d(G) &= \frac{2|E|}{|V|} && (2.3.2.1)\\ &= \frac{2(3n-6)}{n} && (2.3.2.2)\\ &= 6 - \frac{12}{n} && (2.3.2.3)\\ & && (2.3.2.4) \end{aligned}$$

$$\therefore \lim_{n\to\infty} d(G) = 6. \qquad (2.3.2.5)$$

This means that most vertices have degrees less than 6.

Digressively, the fact that a graph G of order at least 4 has a 4-coloring solution is equivalent to the fact that G is 4-partite: assume that G has a

4-coloring solution $c : V(G) \to S$. As an equivalence relation, the 4-coloring solution defines four or less equivalence classes on $V(G)$, and vertices of the same class are independent. Since the n-partite graph of order m is also m-partite where $m > n$, G is 4-partite. Conversely, assume that G is 4-partite. Let us define a coloring solution c' on $V(G)$ such that $c'(v) = c'(w)$ if v, w are in the same class and $c'(v) \neq c'(w)$ if v, w are in different classes. Then the number of the range of c' is at most 4.

If we can prove that if an arbitrary graph G is not 4-partite, then it contains K^5 or $K_{3,3}$ as a minor or a topological minor so that it is not a planar graph by the Theorem of Kuratowski and Wagner (Theorem 2.2.7), then the contraposition asserts that all planar graphs are 4-partite, and therefore have 4-coloring solutions.

Notation 2.3.3. We can assume that a map has only one unbounded region (Section 1.3). Let G be a maximal plane graph containing a subgraph that represents the given map, and let a vertex v of G represent the unbounded region of the map. The edges at v complicate the figure of G. Thus, in this thesis, the edges at v would be omitted and the vertex v would be at the proper location in the figures of the graph representation. And we would represent an edge as a smooth curve even if it is defined as a polygonal arc which is a union of finitely many straight line segments. We can easily distinguish the vertices in the figure of a graph since all vertices have degrees at least 4 by Proposition 2.3.2. See the following example.

Example 2.3.4.

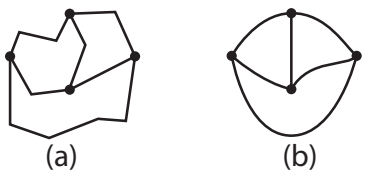

Figure 2.3.1: prefer (b) rather than (a)

Both (a) and (b) of Figure 2.3.1 denote K^4. We use a line segment for representing an edge. But when it is difficult, we use smooth curves such as (b) rather than a union of finite line segments such as (a).

Example 2.3.5.

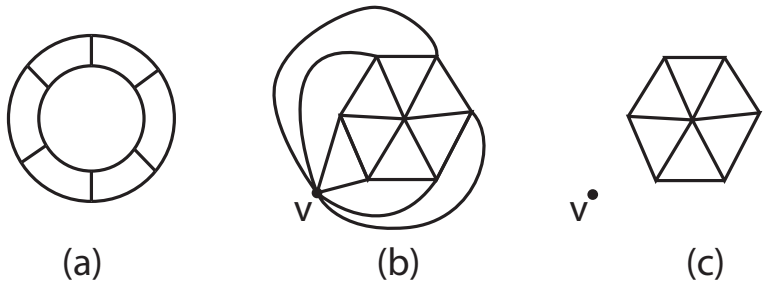

Figure 2.3.2: prefer (c) rather than (b) for a graph notation of map (a)

In Figure 2.3.2, (b) and (c) are the same graph representations of map (a). (c) is obtained from (b) omitting edges at the vertex v that represents the unbounded region of map (a). We prefer figures such as (c) rather than figures such as (b) for a graph representation of a map.

Proposition 2.3.6 (Five Color Theorem)**.** Every planar graph is 5-colorable.

The proof of the Five Color Theorem is given in [2].

Theorem 2.3.7 (Grötzsch 1959)**.** Every planar graph not containing a triangle is 3-colorable [2][9].

Chapter 3

Finding a 4-coloring solution

In this chapter, we will discuss how to color the vertices.

3.1 An introduction of a coloring method

Let G be a plane graph and v be a vertex of G. Assume that we will color the vertices of G one by one and that v have k number of colored neighbours at a moment. Let S denote a set of all colors that have been assigned to the neighbors. Clearly, $|S| =: n \leq k \leq d_G(v)$. Then, we say that v *has* n *colors on its neighbors*. If a vertex v has n colors on its neighbors and another vertex w has m colors on its neighbors and $m < n$, then we say that v has *more* colors on its neighbors than w or that w has *less* colors on its neighbors than v.

In a plane graph G, a *maximal triangle patch without a wheel* or simply a *patch* is a subgraph P of G with the following properties:

(i) P is a triangulation.
(ii) P does not contain a wheel graph.
(iii) If a triangle T_1 of G shares an edge with a triangle T_2 of P, then T_1 is also a triangle of P.

A patch P is *simple* if the closure of the sum of all its faces is a simple region of the plane. Let $\mathbb{P}$ be a set of all patches of G. A vertex v of G is *patch-free from* $\mathbb{P}$ or simply *patch-free* if v is not contained in any member of $\mathbb{P}$.

In a maximal plane graph G, an *independent hubset* or simply a *hubset* is a set H of independent vertices such that each member is a hub of a wheel in G. H is *maximal* if $G - H$ does not contain any wheels. Let us refer to the members of a hubset as *hubs* unless any confusion arises.

Let G be a plane graph, v be a vertex of G, $c : V(G) \to S$ be a vertex coloring of G, and A be a subset of $V(G)$. A usually does not contain v. A *set of adjacent colors* of v with respect to $c|_A$ is the image set $c|_A(A \cap N_G(v))$ in S and it is denoted by $AC_{c|_A}(v)$ or simply $AC_c(v)$ or $AC(v)$. The number $|AC_c(v)|$ is the *AC-number* of v.

Let G be a plane graph of order n. We would like to color its vertices one by one. Since we do not have enough colors, we should decide which vertex to color. If, at a moment, a non-colored vertex $v \in G$ has three colors on its neighbors and we have only four available colors, we have to color this vertex v rather than other vertices. It seems natural to select a vertex that has more colors on its neighbors rather than other vertices that have less colors on their neighbors, among the non-colored vertices. In such a point of view, we can consider a coloring method by constructing a finite sequence of pairs (G_k, c_k) in the following way:

(i) $G_1 = (\{v_1\}, \emptyset)$ where $v_1 \in V(G)$.
(ii) Define a vertex coloring, $c_1 : G_1 \to S$ on G_1.
(iii) Define an induced subgraph $G_k := G[G_{k-1} \cup \{v_k\}]$, where $v_k \in G \setminus G_{k-1}$ is a vertex such that the AC-number $|AC_{c_{k-1}}(v_k)|$ is maximum.
(iv) Define a vertex coloring, $c_k : G_k \to S$ such that $c_k|_{G_{k-1}} = c_{k-1}$ and $c_k(v_k) \in S \setminus AC_{c_{k-1}}(v)$.

The above process does not always work for a 4-element set S, say $\{1, 2, 3, 4\}$. To improve the above process, we can add other conditions for selecting v_k or a process for exchanging some assigned colors. However, this thesis suggests another way:

(i) Convert the given map to a graph and find a maximal plane graph that contains the graph.
(ii) Remove vertices of degree 3 from the maximal plane graph if they exist.

(iii) Find a *hubset* in the graph.
(iv) Color the vertices that are not the elements of the hubset with three available colors.
(v) Color the vertices contained in the hubset with the fourth color.
(vi) Finally, apply the coloring result to the given map.

3.2 Examples of coloring

Example 3.2.1. Find a coloring solution of the given map, Figure 3.2.1.

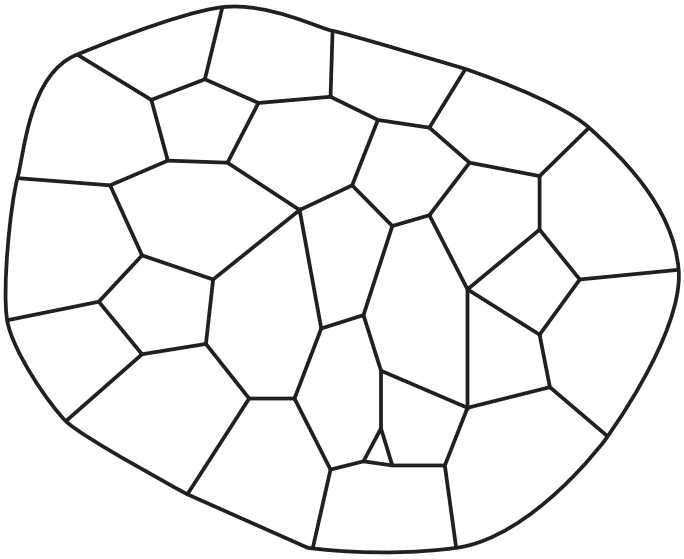

Figure 3.2.1: given map

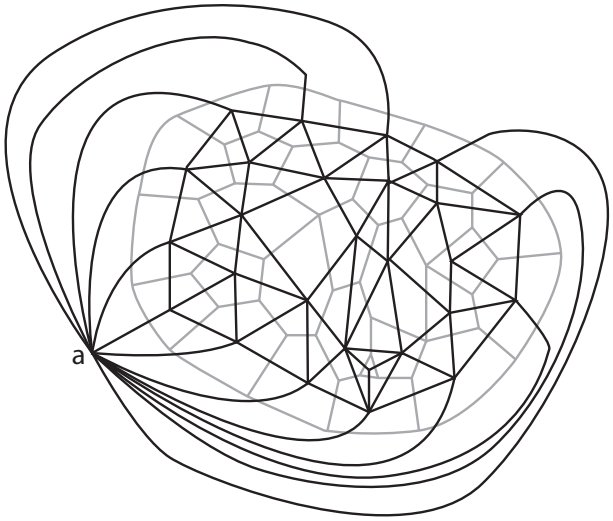

Figure 3.2.2: Step 1

Solution. Step 1: Find a graph representation of the given map in Figure 3.2.1 as described in Section 2.3.

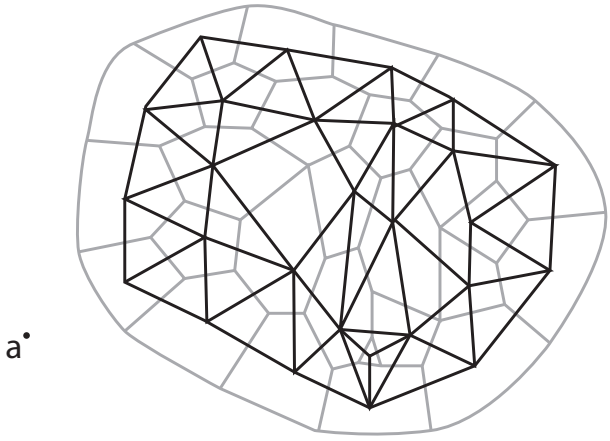

Figure 3.2.3: Step 2

Step 2: Omit the edges at a in the same way as Figure 2.3.3 in Example 2.3.5, where a is the vertex that represents the unbounded region of the given map.

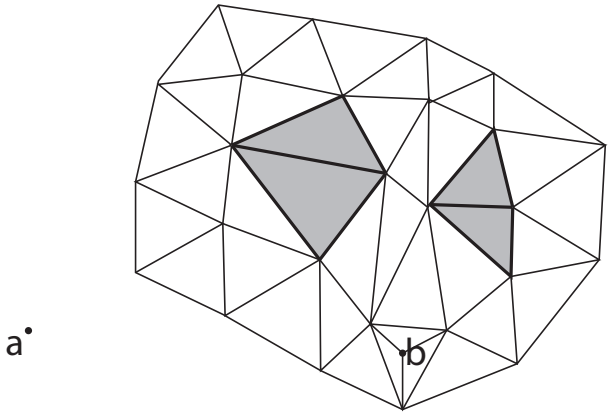

Figure 3.2.4: Step 3

Step 3: By Proposition 2.2.3, the graph in Figure 3.2.3 is not a maximally plane since it contains two faces whose boundaries are not triangles. Find a maximal plane graph containing Figure 3.2.3.

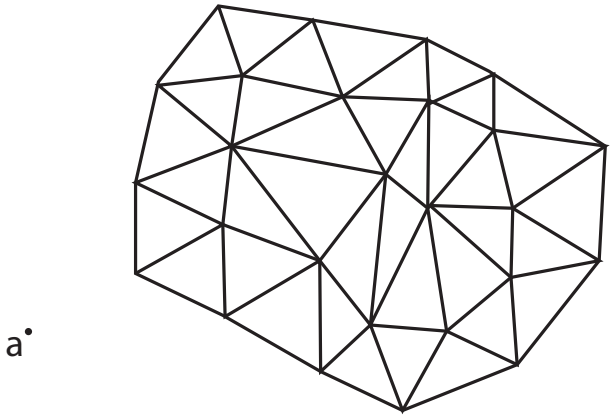

Figure 3.2.5: Step 4, a graph G

Step 4: By Proposition 2.3.2 and the arguments below, the vertex b in Figure 3.2.4 can be removed since its degree is three. Let G denote this graph.

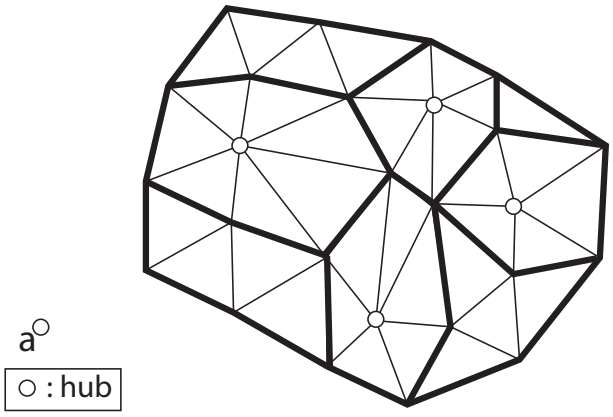

Figure 3.2.6: Step 5

Step 5: Find a hubset H. In Figure 3.2.6, the hubs are represented by white circles. Note that a is also a hub of a wheel.

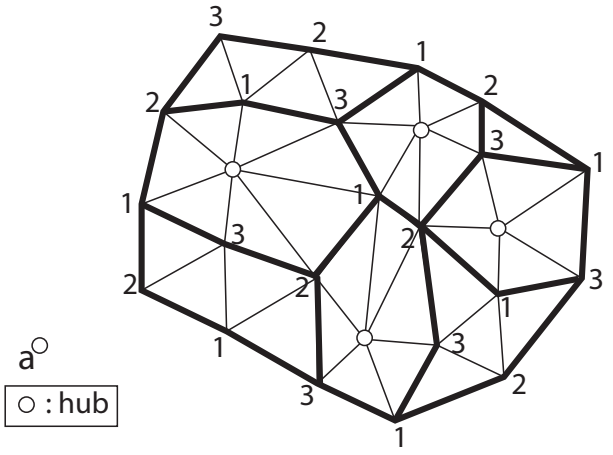

Figure 3.2.7: Step 6

Step 6: Following the vertex-selecting order discussed in Section 3.1, color the patches of $G - H$ with three available colors, say color 1, color 2, and color 3. Note that if $G - H$ had not contained any patches, this step would have been a specific example of the Grötzsch's theorem (Theorem 2.3.7).

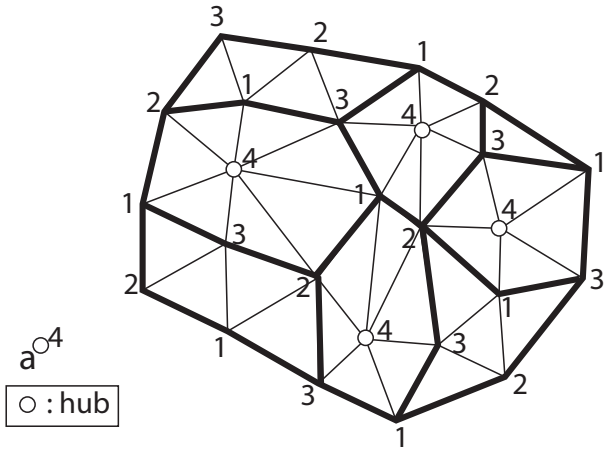

Figure 3.2.8: Step 7

Step 7: Color the hubs with color 4. Note that we never used color 4 at Step 6 and that all the hubs are independent by the definition of the hubset.

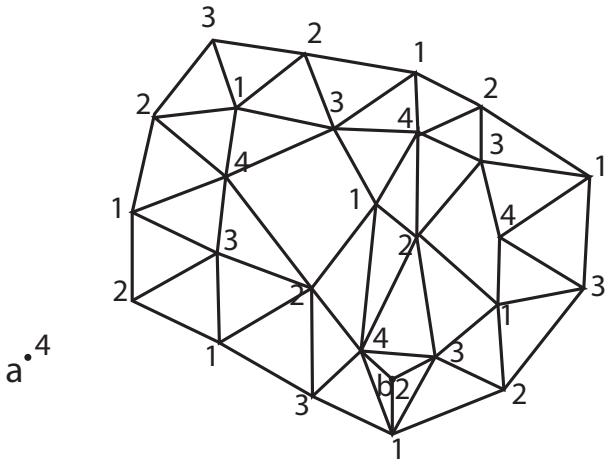

Figure 3.2.9: Step 8

Step 8: Apply the coloring result to the original graph in Figure 3.2.3 obtained at Step 2. Do not forget to color the vertex b that was removed at Step 4. We have only to assign b color 2 since $AC(b) = \{1, 3, 4\}$.

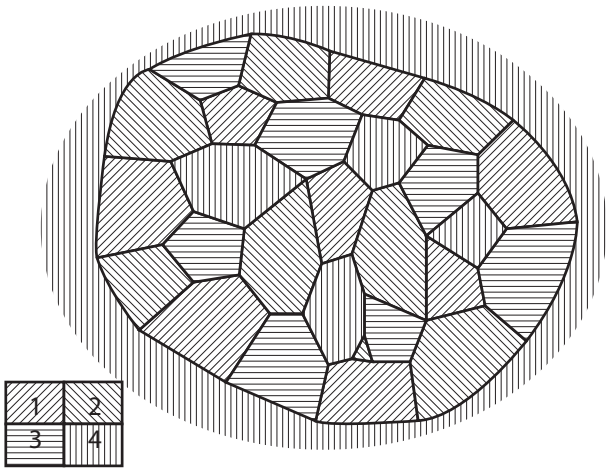

Figure 3.2.10: Step 9

Step 9: Color the region of the given map according to the vertex coloring we have found. □

The coloring process such as the one in Example 3.2.1 does not always success. See the following examples.

Example 3.2.2.

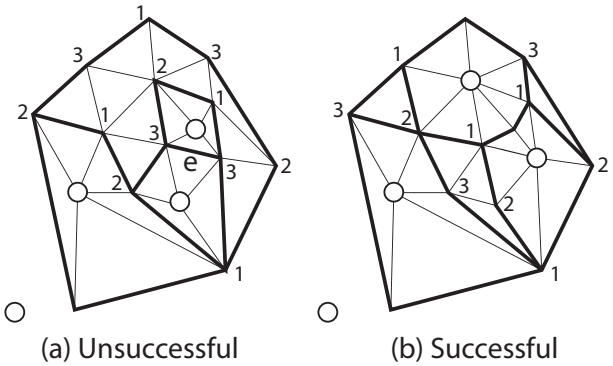

Figure 3.2.11

Figure 3.2.11 shows the different choices of hubsets for the same graph. (a) shows the wrong choice of hubset – the ends of an edge e have the same color.

Example 3.2.3.

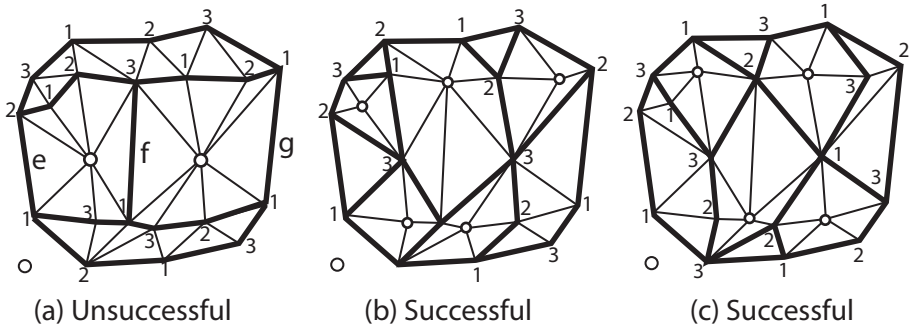

Figure 3.2.12

Figure 3.2.12 also shows the different choices of hubsets for the same graph. The ends of one of the edges $\{e, f, g\}$ in (a) have the same color: lower ends of these edges must have the same color, color 1 as figured, but the upper ends of the edges must have pairwise different colors. Since we have only three available colors, it is unsuccessful by the pigeonhole principle.

Example 3.2.4.

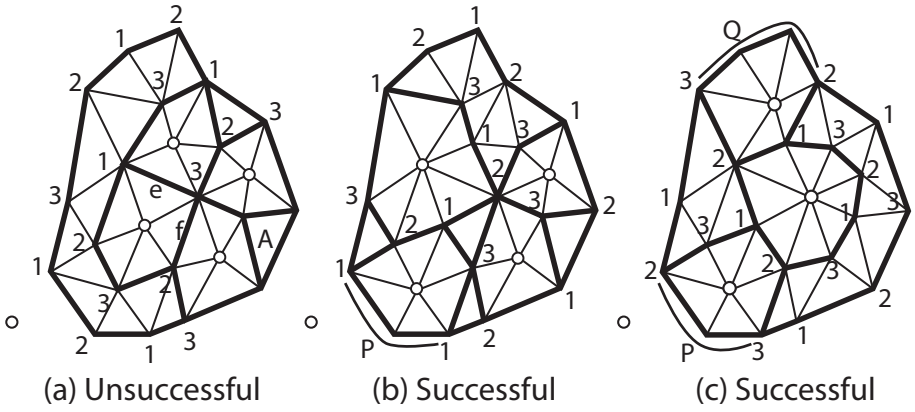

Figure 3.2.13

In Figure 3.2.13 (a), every vertex of a triangle A has color 3 on its neighbors. Thus, it is unsuccessful since we have only three available colors for the vertices of the triangle A. A path $e \cup f$ of length 2 has color 1 on one end and color 2 on the other end. Therefore, the only inner vertex has to be assigned

color 3. Every inner vertex of paths P and Q plays a role as a buffer that prevents the failure of coloring since it always has two neighbors except hubs so that it has at most two colors on its neighbors before coloring hubs.

Example 3.2.5.

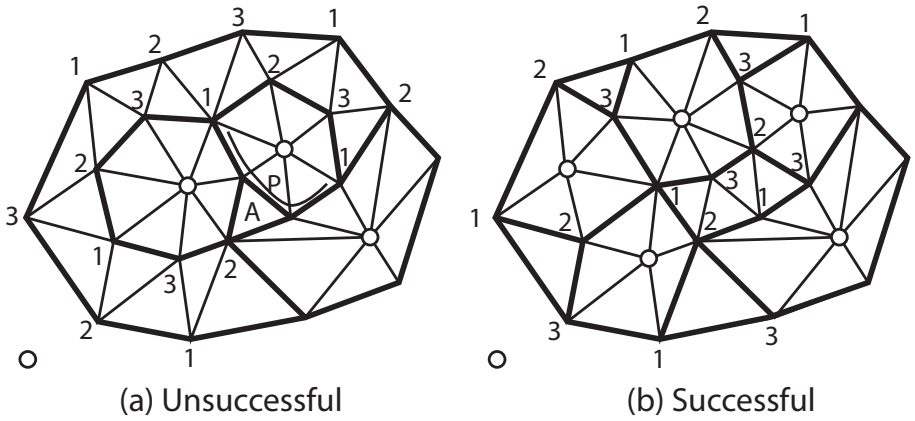

Figure 3.2.14

In Figure 3.2.14 (a), P is a path of length 3 and both of its ends have been assigned color 1. Therefore, one of the two inner points of P has to be colored with color 2. However, the triangle A contains the two inner points of P, and has another vertex that already has been colored with color 2. Therefore, (a) is unsuccessful.

3.3 Observations

Definition 3.3.1. Let G be a plane graph containing at least one triangle. An *overedge sequence* is a sequence $\{v_1, v_2, \cdots\}$ of vertices such that there exists an edge e_i that is the opposite edge of vertex v_i in a triangle T and the opposite edge of vertex v_{i+1} in a triangle T' of G. Note that T and T' can denote the same triangle, and that v_i and v_j can denote the same vertex even if $i \neq j$. Two vertices v_1 and v_2 of G are *overedge* if there exists an overedge sequence containing both v_1 and v_2.

Proposition 3.3.2. For the vertices in a patch P, the overedge relation is an equivalence relation.

Proof. (Reflexivity) Since P is a triangulation, for any vertex v of P, there exists a triangle T that contains v. Let e be an opposite edge of v in T, then a sequence $\{v, v, \cdots\}$ is an overedge sequence since e is an opposite edge of every element of the sequence in T.
(Symmetry) If v_1 and v_2 are members of a sequence, then v_2 and v_1 are also members of the sequence.
(Transitivity) Assume that v_i and v_j are members of an overedge sequence A, and v_j and v_k are members of an overedge sequence B. Then we have a subsequence $\{v_i, \cdots, v_j\}$ of A and another subsequence $\{v_j, \cdots, v_k\}$ of B. By joining the two sequences, we get an overedge sequence $\{v_i, \cdots, v_j, \cdots, v_k\}$. □

Proposition 3.3.3. The overedge relation defines three equivalence classes on a simple patch P.

Proof. Let P be a simple patch and T_1 be a triangle of P. We have three distinct pairs, each of which is a form of (v_1^i, e_1^i) such that v_1^i is an opposite vertex of edge e_1^i where $i = 1, 2, 3$. Let T_2 be another triangle of P that shares an edge e_1^i with T_1. There is only one opposite vertex of the edge e_1^i in T_2 and let v_2^i denote this vertex. Similarly, in a triangle T_j of P that shares an edge e_k^i with $T_1 \cup \cdots \cup T_{j-1}$, let the opposite vertex of e_k^i be denoted by v_{k+1}^i for some index k. Note that T_j shares only one edge with $T_1 \cup \cdots \cup T_{j-1}$ and that there exists only one opposite vertex of e_k^i in T_j since P is simple and it does not contain any wheels. Therefore, a vertex v of $T_1 \cup \cdots \cup T_j$ cannot have two notations; for example, $v_{k_1}^{i_1} = v = v_{k_2}^{i_2}$ where $i_1 \neq i_2$, so that the number of equivalence classes is less than three. □

Corollary 3.3.4. A simple patch P requires exactly three colors to color its vertices. Two vertices v and w of P have the same color if they are overedge.

Proof. Assume that the two vertices v and w are overedge in a simple patch P. Then v and w are non-adjacent in P since P does not contain K^4 that is

a wheel graph. (However, v and w can be adjacent in a supergraph of P. See the Example 3.3.7.) Therefore, we can assign the same color to vertices in the same equivalence class. By Proposition 3.3.3, we need exactly three colors. □

Assume that we find a maximal hubset H of a plane graph G. Consider the graph $G - H$. A patch has six numbers of 3-coloring solutions since a triangle also has six numbers of 3-coloring solutions. And for a colored patch P, if we want a vertex $v \in P$ that was already colored to have a different color, we can just exchange the color of $[v]$ and the color of $[w]$, where $[v]$ and $[w]$ are the different equivalence classes containing v and w, respectively.

Let us consider how a colored patch affects the colorings of another patch in $G - H$. Assume that P and Q are two patches and P has been colored. If they share a vertex v, then we have two choices of 3-coloring solutions for Q. If the patches are linked by an edge vw where $v \in P$, $w \in Q$, then we have four choices of 3-coloring solutions for Q. If the patches are linked by a $P-Q$ path vuw of length 2, where $v \in P$, $w \in Q$, then the vertex coloring of P does not affect the choice of the vertex coloring of Q. Even if the ends of the path have different colors, we can always assign the third color to the inner vertices of the path since the degree of all the inner vertices is 2 in $G - H$. This means that a coloring of P never affects the coloring of Q if all $P - Q$ paths have lengths at least 2. Such a path of length at least 2 can be seen as a graph representation of an interval of the real line such as Figure 1.2.4. Three available colors are enough to color maps of such an interval. When we finish coloring some patches and if there does not exist non-colored vertex of AC-number 2, we should select a vertex that is contained in a non-colored patch linked to the sum of colored patches by a path of shorter length. Note that a vertex that is shared by two patches can be thought as a linking path of length 0. However, which vertices would be the inner vertices of such a path linking two patches in $G - H$? The answer is vertices of degree 4 in G.

Proposition 3.3.5. Let G be a maximal plane graph with $\delta(G) \geq 4$ and H be a maximal hubset of G. Let W^1 and W^2 be two wheels such that $h_1, h_2 \in H$ are hubs of W^1 and W^2, respectively, and W^1 and W^2 share a path P of length at least 2. Then the degree of the inner vertices of P is 4 in G.

Proof. Let v be an inner vertex of P. Then we obtain $d_G(v) \geq 4$ since $\delta(G) \geq 4$. The vertex v has two neighbors v_1 and v_2 in P and two neighbors h_1 and h_2 that are not vertices in P. However, v does not have any other neighbors: each vh_iv_jv $(i, j = 1, 2)$ forms a triangle. If a vertex w is inside of one of the triangles, the degree $d_G(w)$ of w is 3 since G is maximally planar. This contradicts the assumption that $\delta(G) \geq 4$. If w is outside all the triangles, this also contradicts the Jordan curve theorem (Theorem 2.2.1) since $h_1v_1h_2v_2h_1$ forms a cycle containing all the triangles. □

For a maximal hubset H of a maximal plane graph G, we can always color the patch-free vertices of $G - H$ with 3 available colors if they have degree 4 in G. However, if a patch-free vertex v of $G - H$ has a degree at least 6 in G, we are not always able to color the vertex v. See the next example.

Example 3.3.6.

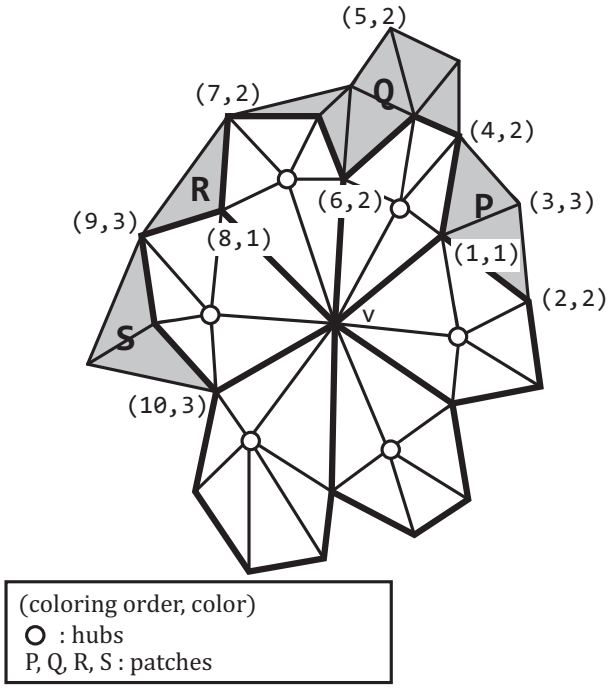

Figure 3.3.1

Due to the wrong choice of a maximal hubset, finding a 3-coloring solution is unsuccessful (Figure 3.3.1). Let v_i denote the i-th colored vertex in the figure. We start coloring v_1 with color 1 and we have no problem until the vertex v_4 is colored with color 2. In patch Q, v_5, v_6 and v_7 should be colored with color 2 since they are overedge with v_4 which has been colored with color 2. When we color the vertex v_8, we have to color it with color 1 because if we color it with color 3 then v which is not a hub, would have three colors on its neighbors. Also, v already has hubs that would be colored with color 4 as its neighbors; thus, v would require the fifth color. This implies that v_9 should be colored with color 3. Since v_{10} is overedge with v_9, it should be colored with color 3. Finally, v requires the fifth color.

A patch-free vertex such as v in Figure 3.3.1 has a degree at least 6. However, Equation 2.3.2.5 says that such a vertex does not appear frequently. Despite this, we can usually find another 3-coloring solution by exchanging colors of two classes of some patches. As Example 3.3.6 states, if there is no 3-coloring solution, we should take another maximal hubset.

Example 3.3.7.

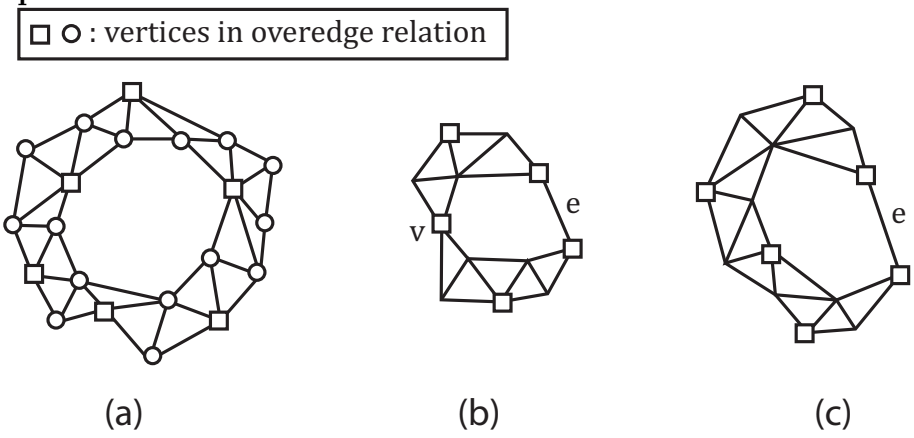

Figure 3.3.2

The patch of (a) in Figure 3.3.2 is not simple and the overedge relation cannot define three equivalence classes. (b) shows that two ends of an edge e that are contained in different patches. They are overedge with a vertex v so that they

are also overedge. (c) shows a situation similar to the case of (b), two vertices are overedge in a patch but they are linked by an edge e.

Situations such as the next example frequently occur.

Example 3.3.8. Consider a path P of length 3 whose ends are colored with the same color, say color 1. Then the two inner vertices v_1 and v_2 have to be colored with color 2 and color 3 or with color 3 and color 2, respectively. See Figure 3.2.14 in Example 3.2.5.

The previous example shows that although the length of a path (that links two patches) is at least 2, if some inner vertices of the path have degrees more than 4, we have to consider these inner vertices when we color the patches containing some of these inner vertices or the patches containing other vertices that are adjacent to some of these inner vertices.

What should we also consider? Let G be a maximal plane graph and H be a maximal hubset in G. Let G' be a graph on $\mathbb{P} \cup \bar{\mathbb{P}}$ where $\mathbb{P}$ is the set of patches in $G - H$ and $\bar{\mathbb{P}}$ is the set of patch-free vertices that have degree at least 2 in $G-H$. Let us define an edge set $E(G')$ of G' such that $vw \in E(G')$ if $v, w \in V(G')$ share a vertex or are linked by an edge. Obviously, G' is a plane graph. The difficult concepts are perhaps the cycles in G' because a coloring of a vertex $v \in G'$ affects the coloring of other vertices along the cycles of G' that contain v and it affects v again. Since the cycles containing v may form complex structures in G' or the colorings of two neighbors of v may affect each other, we should consider the structure when we color v. However, we should only consider the subgraphs of G' that are isomorphic to the subdivisions of two complete graphs, K^3 and K^4, since the Theorem of Kuratowski and Wagner (Theorem 2.2.7) asserts that any subdivision of K^n for $n \geq 5$ does not appear in the planar graph G'. Let K be a subdivision of a complete graph containing v in G'. Set a weight on each edge of K for $vw \in E(K)$: if v and w share n vertices and are linked by m edges, then set the weight $W(vw)$ of $vw \in E(K)$ as $3n+2m$. The quantity $(\sum_{e \in E(K)} W(e))/|K|$ can be a standard for guessing the success rate of coloring locally. The heavier weight, the harder it is for the coloring to be successful.

We have to choose a "good" maximal hubset. Most of the unsuccessful cases of the previous examples contain a patch comparatively bigger than the patches of the successful cases if we define the size of a patch as the number of its faces. It seems better to break the "big" patches into smaller pieces. Assume that we construct a maximal hubset H by adding new members (hubs) one by one. To avoid generating "big" patches or non-simple patches, we can select a vertex as a new hub forming a wheel that shares as many edges as possible with some other wheels that are formed by the hubs that we already found.

It would also be better to take a hubset H in plane graph G such that the patches in $G - H$ are linked by paths of longest possible length. By Proposition 3.3.5 and the arguments before it, it seems better not to select vertices of degree 4 in G as members of the hubset. However, for a maximal plane graph G containing vertices of degree 4, assume that we have found a "good" hubset H so that we have a 4-coloring solution c that assigns the members of H color 4 and assigns some other vertices of degree 4, without loss of generality, color 1. Note that the other vertex coloring c' obtained from c by exchanging color 1 and color 4 is also a 4-coloring solution of G. Inversely, we can find another hubset H' by taking vertices that are assigned color 4 by c' as its members. H' contains vertices of degree 4 in G, and it may not be a maximal.

We suggest the following conditions for finding a maximal hubset.

(i) Select a vertex as a new hub such that the wheel formed by this hub shares as many edges as possible with other wheels formed by hubs that are already selected.

(ii) If a vertex v is overedge with a hub h that we already found, and if v and h are simultaneously adjacent to other 4-degree vertices, select the vertex v as new hub rather than other vertices. If there exist multiple vertices such as v, select the one that has more neighbors of degree 4.

(iii) Choose a maximal hubset H such that $G - H$ would contain less number of faces that are bounded by triangles.

Chapter 4

Computer experiments

Since the method for finding the vertex coloring is not always successful, we would like to check the success rate. This chapter provides some source codes of computer programs and the explanations of them. You can also refer to the comments that are contained in the codes. The codes are written in MATLAB-like language. These codes may work for MATLAB, FreeMat, Octave, and so on. In fact, the codes are written and checked in FreeMat. This thesis, therefore, recommends running the codes in FreeMat.

The *adjacency matrix* $A = (a_{ij})_{n\times n}$ of a graph $G := (V, E)$ is defined by

$$a_{ij} := \begin{cases} 1 & \text{if } v_i v_j \in E \\ 0 & \text{otherwise.} \end{cases}$$

For convenience, we do not distinguish a graph and its adjacency matrix as long as no confusion arises. For example, the adjacency matrix of a graph G is denoted by G, and the graph whose adjacency matrix is A is denoted by A.

4.1 Process flow and the result

The process flow of computer codes is described as follows:

(i) Generate a random maximal plane graph. In fact, the codes generate a random adjacency matrix A of a maximal plane graph.

(ii) Get B by removing the vertices of degree 3 from A and find a hubset H of B. Even if the thesis suggest three conditions for finding the maximal hubset at the end of Section 3.3, the codes takes only condition (i) to find new hubs while constructing a hubset because implementing all the conditions complicates the codes.

(iii) Find patches in $B - H$.

(iv) Construct another adjacency matrix C of a virtual graph whose vertices are equivalence classes of the overedge relation or patch-free vertices of degree more than 4 in B.

(v) Generate the list of probable colorings of C with three available colors.

(vi) If a vertex coloring of C exists in the list then return `success=1`; otherwise return `success=0`.

For a random maximal plane graph of order 40, we obtained the success rate: (number of successes)/(number of tries)= $923/940 \approx 98\%$.

4.2 Source codes

hoRun.m :

```
numSuccess = 0;
wrongEqClassVertexTable = 0;

for loopCount=1:1000
% Iterate 1000 times.
 disp('free memory')
 clear A faces maxIndHubset tempMaxIndHubset B indexB facesB
    patch patchFree patchFree4Degree sizeEqClass
    facesRemovedHub C eqClassVertexTable colorVector success

 disp('Generate random adjacency matrix of a maximal plane
    graph');
 [A,faces] = hoGenRandAdjMat(40);

 disp('Remove vertices of degree 3 and find hubset');
```

```
[maxIndHubset,tempMaxIndHubset,B,indexB,facesB]=
   hoSelectArbMaxIndHubset(A,faces);

disp('Find patches and patch-free vertices');
[patch,patchFree,patchFree4Degree,sizeEqClass,facesRemovedHub
   ]=hoFindPatches(B,tempMaxIndHubset,facesB);

disp('Create a virtual adjacency matrix');
[C,eqClassVertexTable] = hoVirtualAdjMat(patch,patchFree,B,
   tempMaxIndHubset);

disp('Check the eqClassVertexTable');
for i=1:size(eqClassVertexTable,1)
 if(eqClassVertexTable(i,1)==eqClassVertexTable(i,2)|
    eqClassVertexTable(i,1)==eqClassVertexTable(i,3)|
    eqClassVertexTable(i,2)==eqClassVertexTable(i,3))
  disp('wrong eqClassVertexTable');
  wrongEqClassVertexTable=1;
  break;
 end
end

if(wrongEqClassVertexTable==1)
 wrongEqClassVertexTable=0;
 continue;
end

disp('Generate a list of vertex coloring');
colorVector=hoGenColorList(C,eqClassVertexTable);

disp('Check the list of vertex coloring');
success=hoCheck3Colorable(C,colorVector);

if(success==1)
 numSuccess=numSuccess+1;
end

disp('loopCount :')
loopCount
```

```
 disp('numSuccess : ')
 numSuccess
end

disp('success rate :')
numSuccess/loopCount
```

hoGenRandAdjMat.m :

```
function [A,faces] = hoGenRandAdjMat(numVertex)

if(numVertex < 5)
% Treat graphs of order > 4
 disp('number of vertices is less than 4');
 exit();
end

% Maximal plane graph has 3*(number of vertices)-6 edges.
numEdge = 3*numVertex-6;

% A will be an adjacency matrix
A = zeros(numVertex,numVertex);

% To draw maximal graph, we first draw a triangle.
A(2,1) = 1;
A(3,1) = 1;
A(3,2) = 1;

% Faces will be the index set of faces. We got a triangle.
faces(1,1:3) = [1,2,3];

% numTriangle is the number of the faces.
numFace = 1;

% Call the vertices incident to the unbounded face by external
% vertices. Boundary(cycle) of the unbounded face for the
% triangle which formed by first, second and third drawn
% vertices.
```

```
boundaryUnboundedFace = [1,2,3];

% Length of the boundary of the unbounded face. When a new
% vertex and new edges incident to the new vertex are added,
% this value will increse by (3 - (number of the new edges)).
lengthBoundary = 3;

% Now we add a new vertex.
% We draw a new vertex in the unbounded face and draw edges to
% vertices already drawn. the availableNumEdge is maximum
% number of edges incident to new vertex. If a new vertex
% added there need at least 2 edges incident to the new
% vertex. And we already used 3 edges for the first
% triangle. Since the new vertex would drawn in unbounded
% face, the number of edges also bounded by the number of
% vertices of the unbounded face.
%
% original code :
% availableNumEdge =
% min(numEdge-3-2*(numVertex-3),lengthBoundary);
% optimized code :
%
availableNumEdge = min(numEdge-2*numVertex+3,lengthBoundary);

for i=4:numVertex-1
% i-th vertex would have currentNumEdge number of edge with
% vertices already drawn. And we know 1 < currentNumEdge <
% availableNumbEdge+1, so we choose randomly between 2 and
% availableNumbEdge
 if(availableNumEdge==2)
  currentNumEdge = 2;
 else
  currentNumEdge = rem(floor(10000*rand(1,1)),availableNumEdge
     -2)+2;
 end

% The vertices adjacent to the new i-th vertex forms a path
% P in the boundary(cycle) of the unbounded face. Thus we
% can choose the path by choosing its end. Choose it randomly
```

```
% among external vertices.
 currentStartingVertex = rem(floor(10000*rand(1,1)),
    lengthBoundary)+1;

% The boundary of the unbounded face is a cycle but we
% denoted it as vector whose name is boundaryUnboundedFace.
% To use this vector as like a cycle we write this vector
% twice and take some needed part of it.
 tempBoundary = [boundaryUnboundedFace,boundaryUnboundedFace(1:
    size(boundaryUnboundedFace,2)-1)];

% Initiate currentAddedRow which would be i-th row of
% Adjacency matrix A.
 currentAddedRow = zeros(1,i-1);

% From the informations of currentNumEdge and
% currentStartingVertex, construct currentAddedRow.
 for j=1:currentNumEdge
  index=j-1+currentStartingVertex;
  currentAddedRow(tempBoundary(index)) = 1;
 end

% Add new i-th row to the adjacency matrix A.
 A(i,1:i-1) = currentAddedRow;
 clear currentAddedRow;

% And we also have new faces. Add them
 for j=1:currentNumEdge-1
  numFace = numFace+1;
  index=j-1+currentStartingVertex;
  faces(numFace,1:3)=[tempBoundary(index),tempBoundary(index+1)
     ,i];
 end

% lengthBoundary was changed. Reset it.
 lengthBoundary = lengthBoundary+3-currentNumEdge;

% The boundary of the unbounded face was also changed. Its
% new boundary is (an end of P) - (the new vertex) -
```

```
% (another end of P) - (vertices on boundary \ vertices on P))
 boundaryUnboundedFace = [tempBoundary(currentStartingVertex),i
    ,tempBoundary(currentStartingVertex+currentNumEdge-1:size(
    tempBoundary,2))];
 boundaryUnboundedFace = boundaryUnboundedFace(1:lengthBoundary
    );

 clear tempBoundary;

% availableNumEdge could also be changed. Reset it.
 availableNumEdge = min(numEdge-3-currentNumEdge-2*(numVertex-i
    ),lengthBoundary);
end

% We construct the last row of the adjacency matrix A.
currentAddedRow = zeros(1,numVertex);

% The last vertex is adjacent with all vertices of the
% boundary of the unbounded face.
for j=1:lengthBoundary
 currentAddedRow(boundaryUnboundedFace(j))=1;
end

% Add the last row to the adjaency matrix A.
A(numVertex,:) = currentAddedRow;

% Add the faces.
for j=1:lengthBoundary-1
 numFace = numFace+1;
 faces(numFace,1:3)=[boundaryUnboundedFace(j),
    boundaryUnboundedFace(j+1),numVertex];
end

numFace=numFace+1;
faces(numFace,1:3)=[boundaryUnboundedFace(1),
   boundaryUnboundedFace(lengthBoundary),numVertex];

% A is symmetric. For convenience we do not consider this,
% and we just only construct the lower part of A. By adding
```

```
% the transpose matrix of A, we complete it.
A=A+A';

% Sum of all '1' of A have to be equal to the twice of
% number of edges. Check it.
sum(sum(A)) == numEdge*2;
```

hoRemove3degree.m :

```
function [B,indexB,faceB] = hoRemove3degree(A,indexB,faces)
% Remove vertices of 3 degree from given adjacency matrix A.
% give indexB = 0 at first.
% When we remove a 3-degree vertex, another vertex could be
% 3-degree vertex (The vertex of degree 4 which is adjacent
% to the removed vertex). Thus we remove a 3-degree vertex,
% we should restart the removing operation on result matrix.
% In the given adjacency matrix A, i-th row(or column)
% represent i-th drawn vertex. However, in the result matrix
% B, the i-th row does not represent i-th vertex, anymore.
% Thus indexB would tell us, which vertex i-th row of B
% represent for. For example if indexB(3)=5 then the third
% row of B contains the adjacency information of fifth
% vertex of which adjacency information in A is obviously
% fifth row of A.
% When we remove 3-degree vertex, the three faces that are
% incident to the vertex were removed and there appear a new
% face bounded by the triangle containing the three neighbours
% of the removed vertex.

% Initiate result matrix.
B = A;

% Get size of B for FOR sentence.
numVertex = size(B,1);

% Initiate faces.
faceB = faces;

% Get number of faces that we have found.
```

```
numFace = size(faceB,1);

% Initiate indexB.
if(indexB == 0)
 indexB = 1:numVertex;
end

for i = 1:numVertex
% If we find 3-degree vertex, remove it and reset the indexB.

 if(sum(B(i,:))==3)
% If the sum of i-th row of B is equal to 3, the i-th vertex
% of B has the degree 3. We remove the i-th row and the i-th
% column of B.

  if(i==1)
   B = B(2:numVertex,2:numVertex);
   indexB = indexB(2:numVertex);
  elseif (i==numVertex)
   B = B(1:numVertex-1,1:numVertex-1);
   indexB = indexB(1:numVertex-1);
  else
   B = [B(1:i-1,1:i-1),B(1:i-1,i+1:numVertex);B(i+1:numVertex
       ,1:i-1),B(i+1:numVertex,i+1:numVertex)];
   indexB = [indexB(1:i-1),indexB(i+1:numVertex)];
  end

  j=1;
  tempCount = 0;

% We delete the faces containing the vertices of degree 3,
% that were removed at previous IF sentence.
  while(j<numFace+1)
   tempFaceBit =hoConvertIndexToBit(faceB(j,:),numVertex);

% Since we have deleted the i-th 3-degree vertex, we remove
% the faces that is incident to the i-th vertex.
   if(tempFaceBit(i)==1)
    if(j==1)
```

```
     faceB = faceB(2:numFace,:);
    elseif(j==numFace)
     faceB = faceB(1:numFace-1,:);
    else
     faceB = [faceB(1:j-1,:);faceB(j+1:numFace,:)];
    end
    numFace = numFace-1;
    tempCount = tempCount+1;
   else
    j = j+1;
    tempCount = 0;
   end
  end

% We add a new face that is bounded by the triangle whose
% vertices are the neighbors of i-th vertex.
  faceB(numFace+1,:) = hoConvertBitToIndex(A(i,:));
  numFace = numFace+1;

% Convert the index of A to the index of B.
% Since we removed a vertex, the vertices that has indices
% greater than the index of the removed vertex have indices
% decreased.
  for j=1:numFace
   for k=1:3
    if(faceB(j,k)>i)
     faceB(j,k)=faceB(j,k)-1;
    end
   end
  end

% Stop the process and restart same removing process with B
% again.
  [B,indexB,faceB] = hoRemove3degree(B,indexB,faceB);
  break;
 end
end
```

hoSelectArbMaxIndHubset.m :

```
function [maxIndHubset,tempMaxIndHubset,B,indexB,facesB]=
    hoSelectArbMaxIndHubset(A,faces)
% A is an adjacency matrix of a maximal plane graph G.
% maxIndHubset(i) = 1 if i-th vertex of G is contained
% in maximal independent hubset which we would construct.

% Remove vertices of degree 3.
[B,indexB,facesB] = hoRemove3degree(A,0,faces);

% number of vertices of G.
numVertex = size(B,1);

% Initiate the tempMaxIndHubset of which i-th component
% is 1 if i-th vertex, of which information of adjacency is
% denoted by i-th row of B, would be contained in
% maximal independent hubset.
tempMaxIndHubset = zeros(1,numVertex);

% Initiate canBeHub which is vertices that can be an hub.
canBeHub = ones(1,numVertex);

% If we consider about the relation between the sphere
% and the plane all vertices have same conditions for
% being selected as the first hub. However, for the
% convenience to figure the graph in our thesis, we select
% the last vertex.
tempMaxIndHubset(numVertex) = 1;

% And the vertices adjacent to the new hub can be hub
% any more. remove these from canBeHub.
for i=1:numVertex
 if(B(numVertex,i)==1)
  inversedAdjRow(i)=0;
 elseif (B(numVertex,i)==0)
  inversedAdjRow(i)=1;
 end
end
```

```
canBeHub(numVertex) = 0;
canBeHub = canBeHub.*inversedAdjRow;

% We would select new hubs till canBeHub is empty set.
preventInfiniteLoop = 1;

while((sum(canBeHub) ~= 0) & (preventInfiniteLoop < (numVertex
   *3)))
% We would select a new hub which has neighbors that are also
% the neighbors of hubs we already selected, as many as
% possible. countNb(i) is the number of neighbors that are
% also the neighbors of hubs we already selected, of i-th
% vertex. countNb(i) = 0 if the i-th vertex does NOT
% contained in canBeHub.

 countNb = zeros(1,numVertex);

 for i=1:numVertex
  if(canBeHub(i)==1)
   for j=1:numVertex
    if(tempMaxIndHubset(j)==1)
         countNb(i) = countNb(i)+sum(B(i,:).*B(j,:));
        end
   end
  end
 end
% Find  vertices that have maximum number of neighbors
% that are also neighbors of hubs we already selected,
% among the elements of canBeHub.
 tempMaxValue = 0;
 for i=1:numVertex
  if(tempMaxValue < countNb(i))
   tempMaxValue = countNb(i);
  end
 end

% Make set of indices of vertices of which countNb(i) is
% equal to the tempMaxValue.
```

```
 j = 1;
 for i=1:numVertex
  if(countNb(i) == tempMaxValue)
   indexSetMaxValue(j) = i;
   j = j+1;
  end
 end

% Select the vertex as a new hub RANDOMLY
 selectedIndex = indexSetMaxValue(rem(floor(10000*rand(1,1)),j
    -1)+1);
 clear indexSetMaxValue;

% and insert it into maxIndHubset.
 tempMaxIndHubset(selectedIndex) = 1;

% Remove the vertices that are adjacent to the new hub from
% canBeHub.
 for i=1:numVertex
  if (B(selectedIndex,i)==1)
   inversedAdjRow(i)=0;
  elseif (B(selectedIndex,i)==0)
   inversedAdjRow(i)=1;
  end
 end

 canBeHub(selectedIndex)=0;
 canBeHub = canBeHub.*inversedAdjRow;

% prevent infinitely many loop of WHILE statement.
 preventInfiniteLoop = preventInfiniteLoop+1;
end

% Initiate maxIndHubset.
maxIndHubset = zeros(1,size(A,1));

% Convert hubs on B to hubs on A.
for i=1:numVertex
 if(tempMaxIndHubset(i)==1)
```

```
  maxIndHubset(indexB(i))=1;
 end
end
```

hoFindPatches.m :

```
function [patch,patchFree,patchFree4Degree,sizeEqClass,
   facesRemovedHub]=hoFindPatches(B,tempMaxIndHubset,facesB)
% B is adjacency matrix which does not contain 3 degree
% vertices; B could be obtained by 'hoRemove3degree(A)' where
% A is an adjacency matrix A which could be generated by
%'hoGenRanAdjMat' tempMaxIndHubset is a maximal independent
% hubset of B.

% Get size of B
numVertex = size(B,1);

% Initiate nonHubVertices that would not be hubs.
nonHubVertices = ones(1,numVertex);

% Initiate faces. Put face information of B into 'faces'.
facesRemovedHub = facesB;

% Get number of faces
numFace = size(facesRemovedHub,1);

% Remove hubs specified in tempMaxIndHubset from
% nonHubVertices.
for i=1:numVertex
% Generate inversed vector of tempMaxIndHubset.
% In this vector, if i-th vertex contained in the hubset,
% the value of i-th component is 0, otherwise 1.
 if(tempMaxIndHubset(i)==1)
  inversedTempMaxIndHubset(i)=0;
 elseif(tempMaxIndHubset(i)==0)
  inversedTempMaxIndHubset(i)=1;
 end
end
% The componentwise multiplication of a nonHubVertices and
```

```
% the inversed vector of tempMaxIndHubset, removes the hubs
% described by tempMaxIndHubset from nonHubverices.
 nonHubVertices = nonHubVertices.*inversedTempMaxIndHubset;

% Remove faces containing hubs.
% In fact, this removes rows of facesReovedHub that contains
% hubs.
for i=1:numVertex
 if(tempMaxIndHubset(i)==1)
% i-th vertex is a hub.
  j=1;
  while(j<numFace+1)
   tempFaceBit = hoConvertIndexToBit(facesRemovedHub(j,:),
      numVertex);
   if(tempFaceBit(i)==1)
% If a j-th row of facesRemovedHub contains i-th vertex
% remove it from facesRemovedHub that is a list of faces.
    if(j==1)
     facesRemovedHub = facesRemovedHub(2:numFace,:);
    elseif(j==numFace)
     facesRemovedHub = facesRemovedHub(1:numFace-1,:);
    else
     facesRemovedHub = [facesRemovedHub(1:j-1,:);
        facesRemovedHub(j+1:numFace,:)];
    end
    numFace = numFace-1;
   else
    j = j+1;
   end
  end
 end
end

% Initiate some variables.
isFace = 0;
containFace = 0;
numPatch = 0;

% And rename the list of faces.
```

```
facesDB = facesRemovedHub;
sizeFacesDB = size(facesDB,1);

% Since all the faces in facesDB are triangles, each of them
% are contained in some patches.
while(sizeFacesDB > 0)
 numPatch=numPatch+1;
% patch(i,j,k)-th vertex is contained in i-th patch.
% And this vertex is classed by overedge relation so that
% patch(i,j,k)-th vertex is contained in j-th class.

% Process first face
 patch(numPatch,1,1) = facesDB(1,1);
 patch(numPatch,2,1) = facesDB(1,2);
 patch(numPatch,3,1) = facesDB(1,3);

 sizeEqClass(numPatch,1:3)=[1,1,1];

% Remove the face processed from facesDB.
 facesDB = facesDB(2:sizeFacesDB,:);
 sizeFacesDB = sizeFacesDB-1;

 if(sizeFacesDB>0)
  i=1;
  while(i<1+sizeFacesDB)
   eqClass = 0;
   for j=1:sizeEqClass(numPatch,1)
    for k=1:sizeEqClass(numPatch,2)
     if(B(patch(numPatch,1,j),patch(numPatch,2,k))==1)
% Check whether both patch(numPatch,1,j)-th vertex and
% patch(numPatch,2,k)-th vertex are the ends of the same edge.
% One end is from class 1 the other end is from class 2.
% Therefore if the face (triangle) shares the edge with this
% patch, the opposite vertex of the edge in the face is
% contained in class 3.
      tempEdgeContainedBit = hoConvertIndexToBit([patch(
         numPatch,1,j),patch(numPatch,2,k)],numVertex);
      for l=1:sizeFacesDB
```

```
       tempFaceBit = hoConvertIndexToBit(facesDB(l,:),numVertex
          );
       checkShareEdgeBit = tempFaceBit-tempEdgeContainedBit;
       countOne = 0;
       countZero = 0;
       for m=1:numVertex
        if(checkShareEdgeBit(m)==1)
         countOne = countOne+1;
        elseif(checkShareEdgeBit(m)==0)
         countZero = countZero+1;
        end
       end
       if(countOne==1&countZero==numVertex-1)
% We can use only countOne==1 for the IF sentence, but for
% precision, additional condition contZero==numVertex-1 is
% used.
        isFace=1;
% The opposite vertex of the edge in the face is contained in
% class 3.
        eqClass = 3;
        break;
       end
      end
     end
     if(isFace==1)
      break;
     end
    end
    if(isFace==1)
     break;
    end
   end

   if(isFace==0)
        for j=1:sizeEqClass(numPatch,2)
     for k=1:sizeEqClass(numPatch,3)
      if(B(patch(numPatch,2,j),patch(numPatch,3,k))==1)
% Check whether both patch(numPatch,2,j)-th vertex and
% patch(numPatch,3,k)-th vertex are the ends of the same edge.
```

```
% One end is from class 2 the other end is from class 3.
% Therefore if the face (triangle) shares the edge with this
% patch,the opposite vertex of the edge in the face is
% contained in class 1.
       tempEdgeContainedBit = hoConvertIndexToBit([patch(
          numPatch,2,j),patch(numPatch,3,k)],numVertex);
       for l=1:sizeFacesDB
        tempFaceBit = hoConvertIndexToBit(facesDB(l,:),
           numVertex);
        checkShareEdgeBit = tempFaceBit-tempEdgeContainedBit;
        countOne = 0;
        countZero = 0;
        for m=1:numVertex
         if(checkShareEdgeBit(m)==1)
          countOne = countOne+1;
         elseif(checkShareEdgeBit(m)==0)
          countZero = countZero+1;
         end
        end
        if(countOne==1&countZero==numVertex-1)
% We can use only countOne==1 for the IF sentence, but for
% precision, additional condition contZero==numVertex-1 is
% used.
         isFace=1;
% The opposite vertex of the edge in the face is contained in
% class 1.
         eqClass = 1;
         break;
        end
       end
      end
      if(isFace==1)
       break;
      end
     end
     if(isFace==1)
      break;
     end
    end
```

```
   end

   if(isFace==0)
    for j=1:sizeEqClass(numPatch,3)
     for k=1:sizeEqClass(numPatch,1)
      if(B(patch(numPatch,3,j),patch(numPatch,1,k))==1)
% Check whether both patch(numPatch,3,j)-th vertex and
% patch(numPatch,1,k)-th vertex are the ends of the same edge.
% One end is from class 3 the other end is from class 1.
% Therefore if the face (triangle) shares the edge with this
% patch,the opposite vertex of the edge in the face is
% contained in class 2.
       tempEdgeContainedBit = hoConvertIndexToBit([patch(
          numPatch,3,j),patch(numPatch,1,k)],numVertex);
       for l=1:sizeFacesDB
        tempFaceBit = hoConvertIndexToBit(facesDB(l,:),
           numVertex);
        checkShareEdgeBit = tempFaceBit-tempEdgeContainedBit;
        countOne = 0;
        countZero = 0;
        for m=1:numVertex
         if(checkShareEdgeBit(m)==1)
          countOne = countOne+1;
         elseif(checkShareEdgeBit(m)==0)
          countZero = countZero+1;
         end
        end
        if(countOne==1&countZero==numVertex-1)
% We can use only countOne==1 for the IF sentence, but for
% precision, additional condition contZero==numVertex-1 is
% used.
         isFace=1;
% The opposite vertex of the edge in the face is contained in
% class 2.
         eqClass = 2;
         break;
        end
       end
      end
```

```
      if(isFace==1)
       break;
      end
     end
     if(isFace==1)
      break;
     end
    end
   end

   if(isFace==1)
% Remove the processed face form facesDB
    if(l==1)
     facesDB = facesDB(2:sizeFacesDB,:);
    elseif(l==sizeFacesDB)
     facesDB = facesDB(1:sizeFacesDB-1,:);
    else
     facesDB = [facesDB(1:l-1,:);facesDB(l+1:sizeFacesDB,:)];
    end
    sizeFacesDB = sizeFacesDB-1;

% Update the information.
    sizeEqClass(numPatch,eqClass)=sizeEqClass(numPatch,eqClass)
       +1;
    patch(numPatch,eqClass,sizeEqClass(numPatch,eqClass))=
       hoConvertBitToIndex(checkShareEdgeBit);
    isFace=0;
    i=1;
   else
    i=i+1;
   end
  end
 end
end

% Find patch-free vertices among described in nonHubVertices.
index4 = 0;
indexGeneral = 0;
patchFree4Degree = 0;
```

```
patchFree = 0;
for i=1:numVertex
 if(nonHubVertices(i)==1)
  incidentToFace = 0;
  for j=1:numFace
   for k=1:3
    if(facesRemovedHub(j,k)==i)
% If i-th vertex is incident to some faces described in
% facesRemovedHub then the vertex is not patch-free.
     incidentToFace = 1;
     break;
    end
   end
   if(incidentToFace==1)
    break;
   end
  end
  if(incidentToFace==0)
% i-th vertex is patch-free.
% Check whether i-th vertex has the degree 4.
   if(sum(B(i,:))==4)
    index4 = index4+1;
    patchFree4Degree(index4)=i;
   else
    indexGeneral = indexGeneral+1;
    patchFree(indexGeneral)=i;
   end
  end
 end
end
```

hoVirtualAdjMat.m :

```
function [C,eqClassVertexTable] = hoVirtualAdjMat(patch,
   patchFree,B,tempMaxIndHubset)
% This function generate the virtual graph C whose vertices
% symbolize the equivalence classes of patches of B, or
% patch-free vertices of B.
```

```
numVertex = size(B,1);
numPatch = size(patch,1);
numPatchFree = size(patchFree,2);

maxNumVertexInEqClass = size(patch,3);

% eqClassVertexTable(i,j)-th vertex of C symbolize an
% equivalence class of i-th patch of B - "hubset", where the
% members of "hubset" is described in tempMaxIndHubset.
%
% In the first patch, there are three equivalence classes.
% We denote each equivalence class 1,2,3.
eqClassVertexTable(1,1:3)=[1,2,3];
newVertexIndex = 3;
shareVertex = 0;

% Check whether two patches, tempPatchBit_1 and
% tempPatchBit_2, share a vertex.
% If they share a vertex v, [v] of tempPatchBit_1 and [v] of
% tempPatchBit_2 has the same name, where [v] is the
% equivalence class that contains [v].
for indexPatch=2:numPatch
 for i=1:indexPatch-1
  for j=1:3
   for k=1:3
    tempPatchBit_1 = zeros(1,numVertex);
    tempPatchBit_2 = zeros(1,numVertex);
    for n=1:maxNumVertexInEqClass
     if(patch(i,j,n)~=0)
      tempPatchBit_1(patch(i,j,n))=1;
     end
     if(patch(indexPatch,k,n)~=0)
      tempPatchBit_2(patch(indexPatch,k,n))=1;
     end
    end
% If two patches shares a vertex....
    if(sum(tempPatchBit_1.*tempPatchBit_2)~=0)
     shareVertex = 1;
% the k-th equivalence class in (indexPatch)-th patch and
```

```
% j-th equivalence class in i-th patch are symbolized by same
% vertex of C.
     eqClassVertexTable(indexPatch,k) = eqClassVertexTable(i,j)
        ;
     break;
    end
   end
   if(shareVertex == 1)
    break;
   end
  end
 end
 if(shareVertex == 1)
  for l=1:3
% Give names to the equivalence classes that does not contain
% the shared vertex of (indexPatch)-th patch.
   if(eqClassVertexTable(indexPatch,l)==0)
    newVertexIndex=newVertexIndex+1;
    eqClassVertexTable(indexPatch,l)=newVertexIndex;
   end
  end
% Reset shareVertex.
  shareVertex = 0;
 elseif(shareVertex == 0)
% If (indexPatch)-th patch does not share any vertex with
% other m-th patch, where m<indexPatch, give names to these
% three equivalence classes of (indexPatch)-th patch.
  for l=1:3
   newVertexIndex=newVertexIndex+1;
   eqClassVertexTable(indexPatch,l)=newVertexIndex;
  end
 end
end

% Construct Virtual Adjacency Matrix C
% If i-th row (or column) of C symbolize an equivalence class
% in a patch and j-th row (or column) symbolize a path-free
% vertex, then i < j.
%
```

```
% First, we construct lower triangular matrix.
% The three vertices that symbolize the three equivalence
% classes in the same patch are pairwise adjacent in C.
for i=1:numPatch
 if(eqClassVertexTable(i,1)>eqClassVertexTable(i,2))
  C(eqClassVertexTable(i,1),eqClassVertexTable(i,2))=1;
 elseif(eqClassVertexTable(i,1)<eqClassVertexTable(i,2))
  C(eqClassVertexTable(i,2),eqClassVertexTable(i,1))=1;
 end
 if(eqClassVertexTable(i,1)>eqClassVertexTable(i,3))
  C(eqClassVertexTable(i,1),eqClassVertexTable(i,3))=1;
 elseif(eqClassVertexTable(i,1)<eqClassVertexTable(i,3))
  C(eqClassVertexTable(i,3),eqClassVertexTable(i,1))=1;
 end
 if(eqClassVertexTable(i,2)>eqClassVertexTable(i,3))
  C(eqClassVertexTable(i,2),eqClassVertexTable(i,3))=1;
 elseif(eqClassVertexTable(i,2)<eqClassVertexTable(i,3))
  C(eqClassVertexTable(i,3),eqClassVertexTable(i,2))=1;
 end
end

% If a vertex in the equivalence class of a patch and another
% vertex in the equivalence class of another patch are adjacent
% in B, the vertices that symbolize the two equivalence
% classes are adjacent in C.
for i=1:numPatch-1
 for j=1+i:numPatch
  for k=1:3
   for l=1:3
    for m=1:maxNumVertexInEqClass
     for n=1:maxNumVertexInEqClass
      if((patch(i,k,m)~=0)&(patch(j,l,n)~=0))
       if(B(patch(i,k,m),patch(j,l,n))==1)
        if(eqClassVertexTable(i,k)>eqClassVertexTable(j,l))
         C(eqClassVertexTable(i,k),eqClassVertexTable(j,l))=1;
        else
         C(eqClassVertexTable(j,l),eqClassVertexTable(i,k))=1;
        end
       end
```

```
      end
     end
    end
   end
  end
 end
end

% Specify the adjacency of patch-free vertices.
if(patchFree~=0)
 numNonPatchFreeVertex = size(C,1);
 for i=1:numPatchFree
  newVertexIndex=newVertexIndex+1;
% Find the neighbors of the i-th patch-free vertex in B.
  tempNeighborsIndex=hoConvertBitToIndex(B(patchFree(i),:));
  twoPatchFreesAreAdj = 0;
  for j=1:size(tempNeighborsIndex,2)
   includingPatchNumJ=0;
   for s=1:numPatch
    for t=1:3
     for u=1:maxNumVertexInEqClass
      if(patch(s,t,u)==tempNeighborsIndex(j))
% If a patch include some neighbors of i-th patch-free
% vertex....
       includingPatchNumJ=1;
       break;
      end
     end
     if(includingPatchNumJ==1)
      break;
     end
    end
    if(includingPatchNumJ==1)
     break;
    end
   end
   if(includingPatchNumJ==1)
% ...then specify the adjacency information in C.
    C(newVertexIndex,eqClassVertexTable(s,t))=1;
```

```
   elseif(includingPatchNumJ==0&i==1)
% If the first patch-free vertex is never adjacent to any
% other vertices contained in some patch then add zero vector
% in C for the vertex symbolizing the first patch-free vertex
% in C.
    C(newVertexIndex,:)=zeros(1,size(C,2));
   elseif(includingPatchNumJ==0&i~=1)
% If the i-th patch-free vertex is never adjacent to any
% other vertices contained in some patch (i~=1) then Check
% whether this i-th patch-free vertex is adjacent to the
% other u-th patch-free vertices where u<i.
    for u=1:i-1
     if(tempNeighborsIndex(j)==patchFree(u))
% If i-th patch-free vertex and m-th patch-free vertex are
% adjacent, specify the adjacency information in C.
      twoPatchFreesAreAdj=1;
      C(newVertexIndex,numNonPatchFreeVertex+u)=1;
     end
    end
    if(twoPatchFreesAreAdj == 0)
% If not, add zero vector in C for the vertex symbolizing
% this i-th patch-free vertex in C.
     C(newVertexIndex,:)=zeros(1,size(C,2));
    end
   end
  end
 end
end

% The adjacency matrix C should be symmetric.
% We constructed only lower triangular part of C.
% Let us make C symmetric matrix.
tempC = zeros(newVertexIndex,newVertexIndex);
tempC(:,1:size(C,2))=C;
clear C
C = tempC+tempC';
```

hoGenColorList.m :

```
function colorVector=hoGenColorList(C,eqClassVertexTable)
% This function generate a list of 3-colorings for the virtual
% graph C.

% Free the memory.
clear colorVector;

numVertex=size(C,1);
numTriangle = size(eqClassVertexTable,1);

% Since we have at least one patches, each of the first three
% rows (or column) of C represents each equivalence class of
% the first patch. i-th vertex of C will have the color that
% the value of colorVector(i). Since different equivalence
% class have to be assigned different color, the first three
% can be [1,2,3], [2,1,3], [3,2,1], [2,3,1], [3,1,2] or
% [1,3,2]. Without loss of generality, we select [1,2,3].
colorVector = [1,2,3];
for i=2:numTriangle
 numColoredVertex = size(colorVector,2);
 numVertexAppeared = 0;

 for j=1:3
% eqClassVertexTable contain the information of vertices of C
% that are equivalence classes of patches of B (each row of
% eqClassVertexTable is symbolize a patch). Each row of
% eqClassVertexTable is also a face bounded by a triangle in C.

  if(eqClassVertexTable(i,j)<numColoredVertex+1)

% We are deciding the color of eqClassvertextable(i,j)-th
% VERTICES, for j=1,2,3. We are deciding the colors of the
% three vertices at once. However, some of these vertices can
% be specified in eqClassVertexTable MULTIPLE times. If some
% of these vertices were already colored, we should skip
% coloring them. This IF sentence check how many vertices of
% the three vertices (for j=1,2,3) have been colored.
```

```
% numVertexAppeared has the value of the numbers of vertices
% that have already been colored.
   numVertexAppeared = numVertexAppeared+1;
  end
 end

 if(numVertexAppeared==0)
% All the vertices described in the i-th row of
% eqClassVertexTable are never appeared. There is not a
% vertex that was colored at previous step(<i). Locally, we
% have six possible colorings.
  for j=1:size(colorVector,1)
   tempColorVector(6*(j-1)+1,:)=[colorVector(j,:),1,2,3];
   tempColorVector(6*(j-1)+2,:)=[colorVector(j,:),1,3,2];
   tempColorVector(6*(j-1)+3,:)=[colorVector(j,:),2,1,3];
   tempColorVector(6*(j-1)+4,:)=[colorVector(j,:),2,3,1];
   tempColorVector(6*(j-1)+5,:)=[colorVector(j,:),3,1,2];
   tempColorVector(6*(j-1)+6,:)=[colorVector(j,:),3,2,1];
  end
  colorVector=tempColorVector;
  clear tempColorVector;

 elseif(numVertexAppeared==1)
% One of vertices described in the i-th row of
% eqClassVertexTable was appeared. The one of
% eqClassVertexTable(i,j) (j=1,2,3) have been colored at
% previous step(<i). Locally, we have two possible colorings.

  for j=1:3
   if(eqClassVertexTable(i,j)<numColoredVertex+1)
    break;
   end
  end
  for k=1:size(colorVector,1)
   if(colorVector(k,eqClassVertexTable(i,j))==1)
% If the color of the vertex that was already colored is
% color 1, the rest two vertices can have color 2, color 3 or
% color 3, color 2.
    tempColorVector(2*(k-1)+1,:)=[colorVector(k,:),2,3];
```

```
    tempColorVector(2*(k-1)+2,:)=[colorVector(k,:),3,2];
   elseif(colorVector(k,eqClassVertexTable(i,j))==2)
% If the color of the vertex that was already colored is
% color 2, the rest two vertices can have color 1, color 3 or
% color 3, color 1.
    tempColorVector(2*(k-1)+1,:)=[colorVector(k,:),1,3];
    tempColorVector(2*(k-1)+2,:)=[colorVector(k,:),3,1];
   elseif(colorVector(k,eqClassVertexTable(i,j))==3)
% If the color of the vertex that was already colored is
% color 3, the rest two vertices can have color 1, color 3 or
% color 2, color 1.
    tempColorVector(2*(k-1)+1,:)=[colorVector(k,:),1,2];
    tempColorVector(2*(k-1)+2,:)=[colorVector(k,:),2,1];
   end
  end
  colorVector=tempColorVector;
  clear tempColorVector;

 elseif(numVertexAppeared==2)
% Two of vertices described in the i-th row of
% eqClassVertexTable was appeared. The Two of
% eqClassVertexTable(i,j) (j=1,2,3) have been colored at
% previous step(<i). Locally, we have one possible coloring.
  for j=1:3
   if(eqClassVertexTable(i,j)>numColoredVertex)
    break;
   end
  end
  for k=1:size(colorVector,1)
   if(j==3)
% The non-colored vertex is eqClassVertexTable(i,3)-th vertex.
    if((colorVector(k,eqClassVertexTable(i,1))==1&colorVector(k
       ,eqClassVertexTable(i,2))==2)|(colorVector(k,
       eqClassVertexTable(i,1))==2&colorVector(k,
       eqClassVertexTable(i,2))==1))
% The colors of the two colored vertices are color 1, color 2
% or color 2, color 1. Locally, we can assign only color 3 to
% the rest vertex.
     tempColorVector(k,:)=[colorVector(k,:),3];
```

```
    elseif((colorVector(k,eqClassVertexTable(i,1))==1&
       colorVector(k,eqClassVertexTable(i,2))==3)|(colorVector(
       k,eqClassVertexTable(i,1))==3&colorVector(k,
       eqClassVertexTable(i,2))==1))
% The colors of the two colored vertices are color 1, color 3
% or color 3, color 1. Locally, we can assign only color 2 to
% the rest vertex.
     tempColorVector(k,:)=[colorVector(k,:),2];
    elseif((colorVector(k,eqClassVertexTable(i,1))==2&
       colorVector(k,eqClassVertexTable(i,2))==3)|(colorVector(
       k,eqClassVertexTable(i,1))==3&colorVector(k,
       eqClassVertexTable(i,2))==2))
% The colors of the two colored vertices are color 2, color 3
% or color 3, color 2. Locally, we can assign only color 1 to
% the rest vertex.
     tempColorVector(k,:)=[colorVector(k,:),1];
    elseif(colorVector(k,eqClassVertexTable(i,1))==colorVector(
       k,eqClassVertexTable(i,2)))
% The two colored vertice could have the same color and this
% is wrong. Temporarily, we can assign only color 4 to the
% rest vertex.
     tempColorVector(k,:)=[colorVector(k,:),4];
    end
   elseif(j==2)
% The non-colored vertex is eqClassVertexTable(i,2)-th vertex.
    if((colorVector(k,eqClassVertexTable(i,1))==1&colorVector(k
       ,eqClassVertexTable(i,3))==2)|(colorVector(k,
       eqClassVertexTable(i,1))==2&colorVector(k,
       eqClassVertexTable(i,3))==1))
% The colors of the two colored vertices are color 1, color 2
% or color 2, color 1. Locally, we can assign only color 3 to
% the rest vertex.
     tempColorVector(k,:)=[colorVector(k,:),3];
    elseif((colorVector(k,eqClassVertexTable(i,1))==1&
       colorVector(k,eqClassVertexTable(i,3))==3)|(colorVector(
       k,eqClassVertexTable(i,1))==3&colorVector(k,
       eqClassVertexTable(i,3))==1))
% The colors of the two colored vertices are color 1, color 3
% or color 3, color 1. Locally, we can assign only color 2 to
```

```
% the rest vertex.
     tempColorVector(k,:)=[colorVector(k,:),2];
    elseif((colorVector(k,eqClassVertexTable(i,1))==2&
       colorVector(k,eqClassVertexTable(i,3))==3)|(colorVector(
       k,eqClassVertexTable(i,1))==3&colorVector(k,
       eqClassVertexTable(i,3))==2))
% The colors of the two colored vertices are color 2, color 3
% or color 3, color 2. Locally, we can assign only color 1 to
% the rest vertex.
     tempColorVector(k,:)=[colorVector(k,:),1];
    elseif(colorVector(k,eqClassVertexTable(i,1))==colorVector(
       k,eqClassVertexTable(i,3)))
% The two colored vertice could have the same color and this
% is wrong. Temporarily, we can assign only color 4 to the
% rest vertex.
     tempColorVector(k,:)=[colorVector(k,:),4];
    end
   elseif(j==1)
% The non-colored vertex is eqClassVertexTable(i,1)-th vertex.
    if((colorVector(k,eqClassVertexTable(i,2))==1&colorVector(k
       ,eqClassVertexTable(i,3))==2)|(colorVector(k,
       eqClassVertexTable(i,2))==2&colorVector(k,
       eqClassVertexTable(i,3))==1))
% The colors of the two colored vertices are color 1, color 2
% or color 2, color 1. Locally, we can assign only color 3 to
% the rest vertex.
     tempColorVector(k,:)=[colorVector(k,:),3];
    elseif((colorVector(k,eqClassVertexTable(i,2))==1&
       colorVector(k,eqClassVertexTable(i,3))==3)|(colorVector(
       k,eqClassVertexTable(i,2))==3&colorVector(k,
       eqClassVertexTable(i,3))==1))
% The colors of the two colored vertices are color 1, color 3
% or color 3, color 1. Locally, we can assign only color 2 to
% the rest vertex.
     tempColorVector(k,:)=[colorVector(k,:),2];
    elseif((colorVector(k,eqClassVertexTable(i,2))==2&
       colorVector(k,eqClassVertexTable(i,3))==3)|(colorVector(
       k,eqClassVertexTable(i,2))==3&colorVector(k,
       eqClassVertexTable(i,3))==2))
```

```
% The colors of the two colored vertices are color 2, color 3
% or color 3, color 2. Locally, we can assign only color 1 to
% the rest vertex.
     tempColorVector(k,:)=[colorVector(k,:),1];
    elseif(colorVector(k,eqClassVertexTable(i,2))==colorVector(
       k,eqClassVertexTable(i,3)))
% The two colored vertices could have the same color and this
% is wrong. Temporarily, we can assign only color 4 to the
% rest vertex.
     tempColorVector(k,:)=[colorVector(k,:),4];
    end
   end
  end
  colorVector=tempColorVector;
  clear tempColorVector;

% Remove the rows of colorVector that contain color 4.
  l=size(colorVector,2);
  tempIndex = 1;
  for k=1:size(colorVector,1)
   if(colorVector(k,l)~=4)
    tempColorVector(tempIndex,:) = colorVector(k,:);
    tempIndex=tempIndex+1;
   end
  end
  colorVector=tempColorVector;
  clear tempColorVector;

 end
end

% The coloring of vertices of C that symbolize the
% equivalence class of patches was Done. Now, we decide the
% coloring of patch-free vertices, one by one.
%
% numVerticesInPatch is the number of vertices that have been
% already colored.
numVerticesInPatch = size(colorVector,2);
```

```
% The number of patch-free vertices is
% (the numVertex - numVerticesInPatch)
for i = numVerticesInPatch+1:numVertex
 newLengthColorVector = size(colorVector,2)+1;
 tempIndex = 1;
 for j=1:size(colorVector,1)
% If the (i-th) patch-free vertex has 'color m' we set
% adjacentColorSet(m) = 1, for m=1,2,3.
  adjacentColorSet = [0,0,0];
  for k=1:i-1
   if(C(i,k)==1)
    adjacentColorSet(colorVector(j,k))=1;
   end
  end
  if(sum(adjacentColorSet)==3)
% The (i-th) patch-free vertex has three different colors on
% its neighbors. This is wrong. We temporarily set the color
% of this vertex as color 4.
   tempColorVector(tempIndex,:) = [colorVector(j,:),4];
   tempIndex = tempIndex+1;
  elseif(sum(adjacentColorSet)==2)
% The (i-th) patch-free vertex has two different colors on
% its neighbors.
   for l=1:3
% The (i-th) patch-free vertex does NOT have 'color l' on its
% neighbor.
    if(adjacentColorSet(l)==0)
     break;
    end
   end
% We assign this (i-th) patch-free vertex the last color.
   tempColorVector(tempIndex,:)=[colorVector(j,:),l];
   tempIndex=tempIndex+1;
  elseif(sum(adjacentColorSet)==1)
% The (i-th) patch-free vertex has one color on its neighbors.
   for l=1:3
% The (i-th) patch-free vertex have 'color l' on its neighbor.
    if(adjacentColorSet(l)==1)
     break;
```

```
    end
   end
% We assign this (i-th) patch-free vertex the two possible
% colors.
   if(l==1)
% color l is color 1.
    tempColorVector(tempIndex,:)=[colorVector(j,:),2];
    tempColorVector(tempIndex+1,:)=[colorVector(j,:),3];
    tempIndex=tempIndex+2;
   elseif(l==2)
% color l is color 2.
    tempColorVector(tempIndex,:)=[colorVector(j,:),1];
    tempColorVector(tempIndex+1,:)=[colorVector(j,:),3];
    tempIndex=tempIndex+2;
   elseif(l==3)
% color l is color 3.
    tempColorVector(tempIndex,:)=[colorVector(j,:),1];
    tempColorVector(tempIndex+1,:)=[colorVector(j,:),2];
    tempIndex=tempIndex+2;
   end
  elseif(sum(adjacentColorSet)==0)
% The (i-th) patch-free vertex not colored neighbor.
% We can assign any colors (among the three colors) on it.
   tempColorVector(tempIndex,:)=[colorVector(j,:),1];
   tempColorVector(tempIndex+1,:)=[colorVector(j,:),2];
   tempColorVector(tempIndex+2,:)=[colorVector(j,:),3];
   tempIndex=tempIndex+3;
  end
 end
 colorVector = tempColorVector;
 clear tempColorVector;

% Remove the rows of colorVector that contain color 4.
 l=size(colorVector,2);
 tempIndex = 1;
 for k=1:size(colorVector,1)
  if(colorVector(k,l)~=4)
   tempColorVector(tempIndex,:)=colorVector(k,:);
   tempIndex=tempIndex+1;
```

```
  end
 end
 colorVector=tempColorVector;
 clear tempColorVector;
end
```

hoCheck3Colorable.m :

```
function success=hoCheck3Colorable(C,colorVector)
% This checks whether the 3-coloring solution of C is contained
% in colorVector. If it is return 1, otherwise return 0.
numColorList = size(colorVector,1);
numVertex = size(C,1);

for i=1:numColorList
% Assume the row is a 3-coloring solution.
 success=1;
 for j=1:numVertex
  for k=1:j-1
% Check every pair of adjacent vertices.
   if(C(j,k)==1&colorVector(i,j)==colorVector(i,k))
% If the two adjacent vertices have the same color, return 0.
    success = 0;
    break;
   end
  end
  if(success == 0)
% escape the FOR sentence.
   break;
  end
 end
 if(success==1)
% If there exists a 3-coloring solution, print it.
  colorVector(i,:)
  break;
 end
end
```

hoConvertBitToIndex.m :

```
function result=hoConvertBitToIndex(v)
% This convert Bit-representation of set of vertices to
% Index-representation. For example, let V = {1,3,5} be a set
% of vertices. V can be represent as [1,3,5] or [1,0,1,0,1].
% This function convert [1,0,1,0,1] to [1,3,5].

index = 1;
for i=1:size(v,2)
 if(v(i)==1)
  result(index)=i;
  index=index+1;
 end
end
```

hoConvertIndexToBit.m :

```
function result=hoConvertIndexToBit(v,numVertex)
% This convert Index-representation of set of vertices to
% Bit-representation. For example, let V = {1,3,5} be a set of
% vertices. V can be represent as [1,3,5] or [1,0,1,0,1]. This
% function convert [1,3,5] to [1,0,1,0,1].

result = zeros(1,numVertex);
for i=1:size(v,2)
 result(v(i))=1;
end
```

hoTranslateAdjMat.m :

```
function trans=hoTranslateAdjMat(A)
% This is a utility to help drawing the graph of A. When we
% draw i-th vertex of graph of A, the i-th row of 'trans' tell
% us which vertices should be adjacent to the i-th vertex. For
% example, if the sixth row of 'trans' is [1,4,0,0,0], we
% should draw line between first vertex and sixth vertex, and
% between fourth vertex and sixth vertex when we draw the
% sixth vertex.
```

```
for i=2:size(A,1)
 tempIndex=1;
 for j=1:i-1
  if(A(i,j)==1)
   trans(i-1,tempIndex)=j
   tempIndex=tempIndex+1;
  end
 end
end

trans
```

If you have all the previously specified codes, then you can get the result just by executing the following command.

```
--> hoRun
```

4.3 Manual of the programs

This section contains an example describing how to use the codes.

The `[A,faces] = hoGenRandAdjMat(20)` command generates a random adjacency matrix `A` of a maximal plane graph of order `20` and `faces` which is a list of its faces.

```
--> [A,faces] = hoGenRandAdjMat(20)
A =
 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 1 0 1 1 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
 1 1 0 0 1 0 0 1 0 1 0 0 0 0 1 0 0 1 0 0
   .
   .
   .
(omitted)

faces =
  1  2  3
  1  2  4
  3  1  5
   .
   .
   .
(omitted)
```

The `[maxIndHubset,tempMaxIndHubset,B,indexB,facesB] = hoSelectArbMaxIndHubset(A,faces)` command generates an adjacency matrix `B` obtained from `A` by removing the information of the vertices of degree 3. It also generates a vector `tempMaxIndHubset` that contains the information of a maximal hubset of `B`.

```
--> [maxIndHubset,tempMaxIndHubset,B,indexB,facesB]=
   hoSelectArbMaxIndHubset(A,faces)
maxIndHubset =
 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0
tempMaxIndHubset =
 0 0 0 0 1 0 1 0 0 0 0 1
```

```
B =
 0  1  1  1  1  1  0  0  0  0  0  0
 1  0  1  1  0  0  1  0  0  0  0  0
 1  1  0  0  1  0  1  0  1  0  1  1
 1  1  0  0  0  1  1  0  1  0  0  0
 1  0  1  0  0  1  0  1  0  1  1  0
 1  0  0  1  1  0  0  1  1  0  0  0
 0  1  1  1  0  0  0  0  1  0  0  0
 0  0  0  0  1  1  0  0  1  1  0  0
 0  0  1  1  0  1  1  1  0  1  0  1
 0  0  0  0  1  0  0  1  1  0  1  1
 0  0  1  0  1  0  0  0  0  1  0  1
 0  0  1  0  0  0  0  0  1  1  1  0
indexB =
  1   2   3   4   5   6   8   9  10  12  15  18
facesB =
  1   2   3
  1   2   4
  3   1   5
    .
    .
 (omitted)
```

The `[patch,patchFree,patchFree4Degree,sizeEqClass, facesRemovedHub]=hoFindPatches(B,tempMaxIndHubset,facesB)` command finds patches (`patch`), patch-free vertices (`patchFree`), and some other information. The `patch` is a 3-dimensional array of numbers. The member (m, n, l) of `patch` denotes the l-th vertex of the n-th equivalence class of the overedge relation in the m-th patch. In the following example, there is one patch that is refined into three equivalence classes. The vertices 1 and 9 are contained in the first equivalence class of the first patch. '0' means empty.

```
--> [patch,patchFree,patchFree4Degree,sizeEqClass,
   facesRemovedHub]=hoFindPatches(B,tempMaxIndHubset,facesB)
patch =
(:,:,1) =
  1   2   3
(:,:,2) =
```

```
  9   6   4
(:,:,3) =
  0  10   8
patchFree =
 0
patchFree4Degree =
 11
sizeEqClass =
 2 3 3
facesRemovedHub =
  1   2   3
  1   2   4
  1   4   6
  8   6   9
  6   4   9
  8   9  10
```

The `[C,eqClassVertexTable] = hoVirtualAdjMat(patch,patchFree,B ,tempMaxIndHubset)` command creates a virtual adjacency matrix `C` of an abstract graph on the sum of equivalence classes of patches and patch-free vertices of degree more than 4 in `B`. In the example, `C` describes a triangle since we had one patch without a patch-free vertex of degree more than 4.

```
--> [C,eqClassVertexTable] = hoVirtualAdjMat(patch,patchFree,B,
   tempMaxIndHubset)
C =
 0 1 1
 1 0 1
 1 1 0
eqClassVertexTable =
 1 2 3
```

The `colorVector=hoGencolorList(C,eqClassVertexTable)` command generates `colorVector` which is a list of probable 3-colorings of `C`.

```
--> colorVector=hoGencolorList(C,eqClassVertexTable)
colorVector =
 1 2 3
```

The `success=hoCheck3Colorable(C,colorVector)` command checks whether the list contains a 3-coloring solution of `C`. If so, it returns `success=1` and if not, it returns `success=0`. This command also shows the 3-coloring solution. The value of the n-th component of the vector `ans` is the color assigned to the n-th vertex of `C`. We can find the vertex coloring of `B` with the information contained in `patch`. In this example, vertices 1 and 9 would be colored with color 1; vertices 2, 6, and 10 would be colored with color 2; and vertices 3, 4, and 8 would be colored with color 3.

```
--> success=hoCheck3Colorable(C,colorVector)
ans =
 1 2 3
success =
 1
```

The checking process is finished. However, the `hoTranslateAdjMat(B)` helps us to figure the graph of `B`. See the following.

```
--> hoTranslateAdjMat(B)
   .
   .
   .
 (omitted)
   .
   .
   .
ans =
  1   0   0   0   0
  1   2   0   0   0
  1   2   0   0   0
  1   3   0   0   0
  1   4   5   0   0
  2   3   4   0   0
  5   6   0   0   0
  3   4   6   7   8
  5   8   9   0   0
  3   5  10   0   0
  3   9  10  11   0
```

From the matrix named `ans`, we can figure the graph on the paper. The n-th row of `ans` shows what vertices would be the neighbors of the $(n+1)$-th vertex among n vertices (that are already drawn) when we draw the $(n+1)$-th vertex. For example, when we draw the sixth vertex on the paper, we draw lines between the sixth vertex and the first vertex, between the sixth vertex and the fourth vertex, and between the sixth vertex and the fifth vertex, since the fifth row of `ans` is `[1 4 5 0 0]`. Note that whenever we finish adding a vertex on the paper, There should be at most one face whose boundary is not a triangle. Obviously, if there exists the unique face whose boundary is not a triangle, then it is the unique unbounded face.

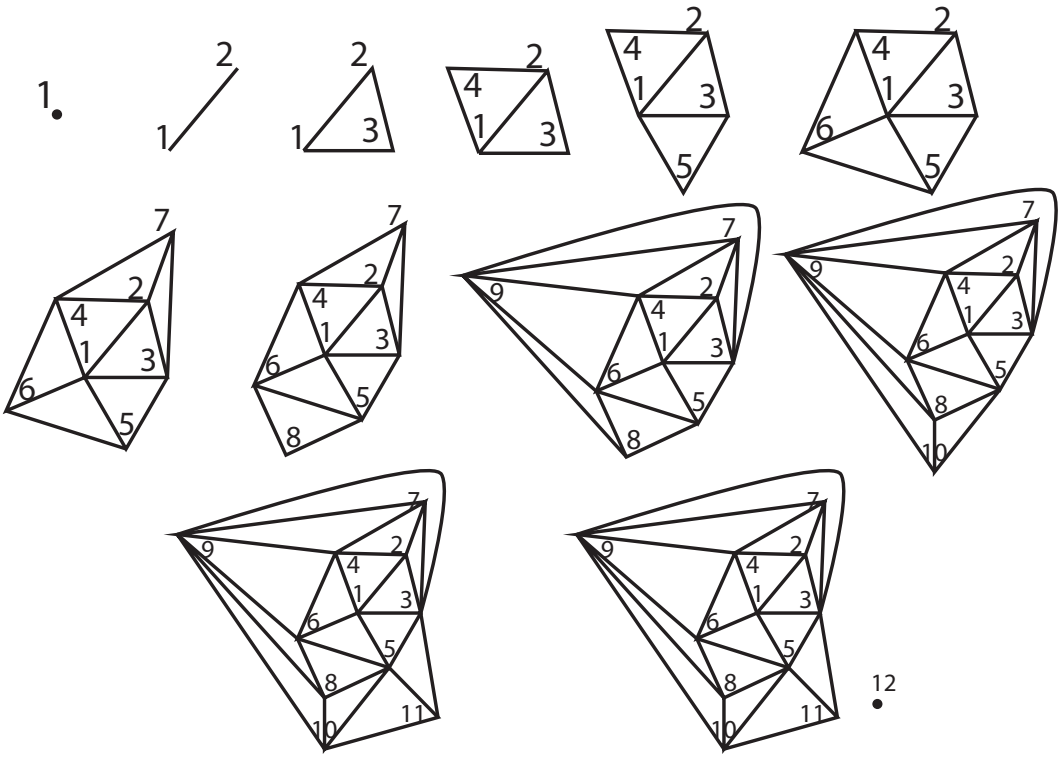

Figure 4.3.1: Drawing graph B.

Also, we can figure the hubs and their wheels with the help of the command `hoConvertBitToIndex(tempMaxIndHubset)`. In the example, hubs are the fifth, seventh, and twelfth drawn vertices.

```
--> hoConvertBitToIndex(tempMaxIndHubset)
ans =
  5  7  12
```

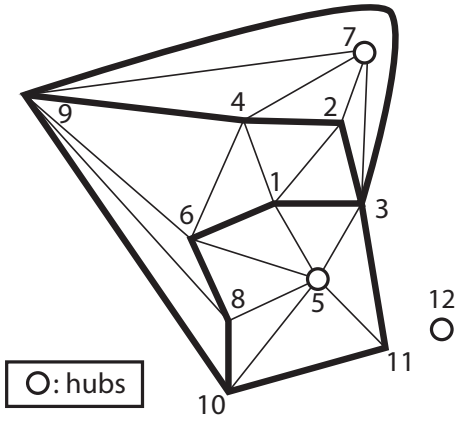

Figure 4.3.2: Drawing hubs and wheels on graph B.

It shows that there is one patch, and the members of each equivalence class of the patch are as the 3-dimensional array `patch` describes. We can easily find a 4-coloring solution of this graph.

Bibliography

[1] Béla Bollobás, *Mordern Graph Theory*, Springer, 1998.

[2] Reinhard Diestel, *Graph Theory*, Springer, 2006.

[3] Georges Gonthier, Formal Proof–The Four-Color Theorem, *Notices of the American Mathematical Society* **55**(11) (2008), 1382–1393.

[4] Frank Harary, *Graph Theory*, Perseus Books, 1994.

[5] George A. Jennings, *Modern Geometry with Applications*, Springer, 1994.

[6] James R. Munkres, *Topology*, Prentice Hall, 2000.

[7] Neil Robertson, Daniel Sanders, Paul Seymour, and Robin Thomas, The Four-color Theorem, *Journal of Combinatorial Theory, Series B* **70** (1997), 2–44.

[8] John Stillwell, *Classical Topology and Combinatorial Group Theory*, Springer, 1980.

[9] Carsten Thomassen, A short list color proof of Götzsch's theorem, *Journal of Combinatorial Theory, Series B* **88** (2003), 189–192.

국문초록

사색 문제로 알려진 *사색 이론*은 지도 상의 서로 인접한 두 영역들이 같은 색상을 가지지 않도록 채색할 때, 네 가지 색상만으로도 조건을 충분히 만족시키며 모든 영역들을 채색할 수 있다는 이론이다. 사색 이론은 평면 그래프의 꼭짓점을 채색하는 문제로 해석될 수 있다. 평면 그래프의 꼭짓점을 채색하기 위해 우리는 다음과 같은 방법에 대해 알아보도록 한다.

1. 주어진 지도를 그래프로 표현하고, 이 그래프를 포함하며 꼭짓점의 수는 같고 가능한 많은 모서리를 가지는 평면 그래프를 하나 찾는다.
2. 이 평면 그래프에서 차수가 3 이하인 꼭짓점이 있다면 제거한다.
3. 이 그래프에서 서로 인접하지 않으며 각각 어떤 차륜형 부분 그래프들의 중심이 되는 꼭짓점들로 이루어진 *중심점 집합*을 찾는다.
4. 중심점 집합에 원소가 아닌 꼭짓점들을 세 가지 색상로만 채색한다.
5. 중심점 집합의 원소가 되는 꼭짓점들을 앞의 세가지 색상과는 다른 네 번째 색상으로 채색한다.
6. 이 결과를 처음에 주어진 지도에 적용한다.

위와 같은 방법을 이용하여, 무작위로 생성된 꼭짓점의 개수가 40인 그래프를 채색하는 컴퓨터 실험에서 약 98%의 채색 성공률을 얻을 수 있었다. 이 채색 방법을 개선하는 것에 대해 논하도록 한다.

주요어휘: 꼭짓점 색칠하기, 평면 그래프, 4색, 사색, 사색문제, 사색이론
학번: 2009-20284

감사의 글

부족하고 미욱한 학생을 지도하고 이끌어주신 김홍종 선생님께 깊은 감사를 드립니다. 항상 격려해주시는 선생님의 그늘이 있었기에 이렇게 학위 논문을 무사히 마칠 수 있었습니다.

바쁜 일정에도 시간을 내어 논문 심사를 맡아 주신 국웅 선생님과 김서령 선생님께 감사를 드립니다.

또한 고성은 선생님, 신동관 선생님, 권오인 선생님, 김태희 선생님, 박춘재 선생님, 이상진 선생님, 이승훈 선생님, 정은옥 선생님, 팽성훈 선생님, 최인송 선생님, 남현수 선생님, 이선미 선생님께 뒤늦은 감사의 뜻을 전하고자 합니다. 특히 이상진 선생님께서 가르쳐 주신 위상 수학의 기초와 정은옥 선생님께 배운 컴퓨터의 활용은 이 논문을 쓰는데 많은 도움이 되었습니다.

논문의 내용을 함께 공부하여 주셨으며 글을 쓰는데 있어서 구체적인 조언을 많이 주신 박경동 선배님께 감사의 말씀을 드립니다. 더불어 어렵게 여겼던 내용들을 친절히 설명하여 주셨던 변태창 선생님께도 감사의 말씀을 전하고자 합니다.

함께 공부하였고 우매한 물음에도 기꺼이 응해주었던 김동훈, 신필수, 장윤수, 채우리, 최우철 외 대학원 동기들에게 감사를 드립니다.

수리과학부를 뒷받침해 주시는 이국현 선생님과 오해자 선생님을 비롯한 행정실 직원분들께 감사의 뜻을 전하고자 합니다.

또한 함께 공부하며 많은 이야기를 들려준 이완호, 김신영, 신동원, 박종민, 최익준, 방규호, 김이랑, 김상우, 김성원, 김승현, 김지나, 노영한, 박연경, 유춘식, 허윤호, 최선화, 김태연, 윤용섭, 김효진, 김성은, 임예옥, 장효정, 김용운, 서우덕, 이주희, 정미선, 정진, 홍석영, 권유선, 김영은, 신지수, 강대용, 박성호, 김민혜, 이채영, 권시은, 남혜현, 황미애, 강진구, 박정배, 김혜진, 문경식, 손재원, 오필선, 강정석, 곽성근, 김현수, 류대호, 정현민, 김녹형 외 선후배님들과 동문들에게 감사의 말씀을 전합니다.

영어 교정에 많은 도움을 주신 김수잔 선생님께 감사를 드립니다.

지금도 여러 고민들을 함께 나누어 주는 김재현, 김범진, 이상봉, 이우진, 이주안, 최영철, 최윤석, 이성현, 서정우, 윤용한 외 친구들에게 감사의 뜻을 전합니다.

언제나 아낌없이 주시고 보살펴 주시는 부모님께, 그리고 항상 응원을 해준 두 동생들에게 감사의 마음을 전합니다.

creative commons
COMMONS DEED

저작자표시-비영리-동일조건변경허락 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.
- 이차적 저작물을 작성할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

동일조건변경허락. 귀하가 이 저작물을 개작, 변형 또는 가공했을 경우에는, 이 저작물과 동일한 이용허락조건하에서만 배포할 수 있습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

이학석사 학위논문

Vertex coloring of plane graphs

(평면 그래프 색칠하기)

2014년 2월

서울대학교 대학원

수리과학부

최 호 석

Vertex coloring of plane graphs

(평면 그래프 색칠하기)

지도교수 김 홍 종

이 논문을 이학석사 학위논문으로 제출함

2013년 10월

서울대학교 대학원

수리과학부

최 호 석

최 호 석의 이학석사 학위논문을 인준함

2013년 12월

위 원 장	국 웅	(인)
부 위 원 장	김 홍 종	(인)
위 원	김 서 령	(인)

Vertex coloring of plane graphs

A dissertation
submitted in partial fulfillment
of the requirements for the degree of
Master of Science
to the faculty of the Graduate School of
Seoul National University

by

Ho-Seok Choe

Dissertation Director : Professor Hong-Jong Kim

Department of Mathematical Sciences
Seoul National University

February 2014

© 2014 Ho-Seok Choe

All rights reserved.

Abstract

Vertex coloring of plane graphs

Ho-Seok Choe

Department of Mathematical Sciences
The Graduate School
Seoul National University

The *four color theorem* states that only four colors are needed to color the regions of any simple planar map so that any two adjacent regions have different colors. This theorem can be interpreted as finding a vertex coloring of plane graphs. This thesis suggests a method to find a vertex coloring of plane graphs with four available colors that includes the following steps:

(i) Convert the given map to a graph and find a maximal plane graph that contains the graph.
(ii) Remove vertices of degree 3 from the maximal plane graph if they exist.
(iii) Find a *hubset* that is a set of independent hubs of wheels.
(iv) Color the vertices that are not the elements of the hubset with three available colors.
(v) Color the vertices contained in the hubset with the fourth color.
(vi) Finally, apply the coloring result to the given map.

Using this process, we obtained a 98% success rate in computer experiments for random graphs of order 40. We will discuss how to improve the coloring process.

Key words: vertex coloring, plane graph, four color, 4-color
Student Number: 2009-20284

Contents

List of Figures

LIST OF FIGURES

Chapter 1

Introduction

1.1 What is the Four Color Theorem?

Suppose that we are going to make a (geographic) map of our nation or a blueprint of our house. We would draw several lines to distinguish each region, and color the regions to increase the visibility of our work. If we are only interested in distinguishing the regions, at least how many colors do we need? In other words, suppose that two regions have different colors if they share a line as their boundary. Then, at least how many colors do we need? The conjecture that only four colors are needed to complete such a coloring task for an arbitrary map was first proposed in 1852 by Francis Guthrie [3]. At long last, Kenneth Appel and Wolfgang Haken proved the four color theorem using a computer in 1976, and their proof was improved in 1996 by Neil Robertson, Daniel Sanders, Paul Seymour, and Robin Thomas [7].

A *region* is a connected open subset of the plane. A *planar map* is a set of disjoint regions of the plane. A point is a *corner* of a map if it is contained in the closures of at least three regions. Two regions of a map are *adjacent* if their closures share a point that is not a corner. The four color theorem asserts that only four colors are needed to color the regions of any simple planar map so that any two adjacent regions have different colors [3].[1]

[1] [3] defines a *planar map* as a set of pairwise disjoint subsets called regions of the plane,

1.2 Generalization of the Four Color Theorem

People may wonder where the number "4" comes from. And the dimensions, the topological structures, or some properties of the spaces containing the given map would probably cross their minds. Here are some examples that help us check our conjectures.

Example 1.2.1.

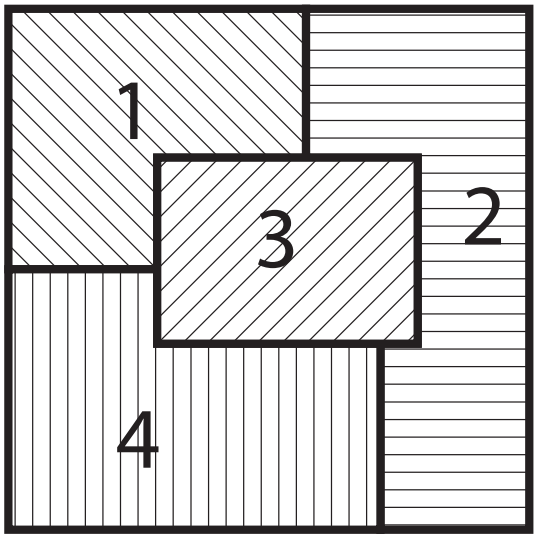

Figure 1.2.1: Map that requires four colors to color the regions

In Figure 1.2.1, we show a map containing four regions (five regions if we count the unbounded region) on a plane. All the regions are pairwise adjacent. Therefore, at least four colors are required to color this map.

and a *simple planar map* as a planar map whose regions are connected open sets.

Example 1.2.2.

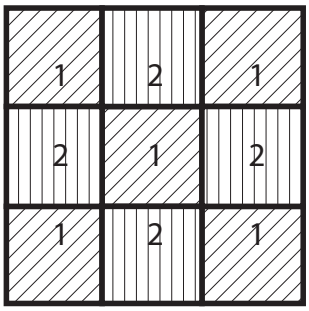

Figure 1.2.2: Chessboard

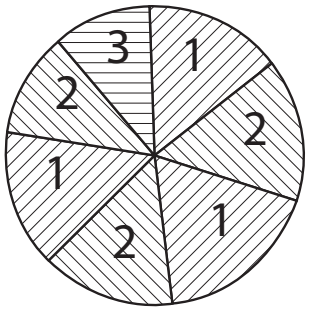

Figure 1.2.3: Pizza with odd pieces

Figure 1.2.2 and Figure 1.2.3 show maps that require less than 4 colors since they have specific structures.

What if the spaces containing the maps are not two dimensional?

Example 1.2.3.

Figure 1.2.4: an interval of the real line

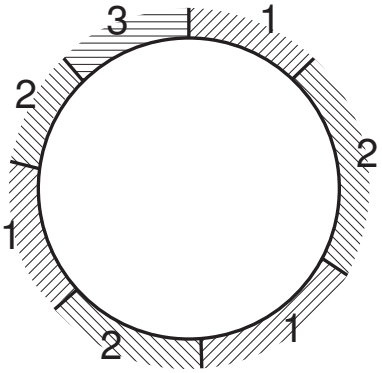

Figure 1.2.5: a circle

Figure 1.2.4 shows an interval of the real line $\mathbb{R}$ and It does not contain a corner since any point p of $\mathbb{R}$ separates $\mathbb{R} \setminus \{p\}$ into two distinct partitions ($\{x \in \mathbb{R} : x < p\}$ and $\{x \in \mathbb{R} : x > p\}$). Therefore, if two regions A and B of $\mathbb{R}$ share a point p as a common frontier and A has been assigned a color, say color 1, then B can be assigned another color, say color 2. By repeating this

work for another frontier of the sum of colored regions, we can color all the regions with only two colors. Figure 1.2.5 shows a circle that can be viewed as joining two ends of a bounded interval containing an odd number of regions. The map in this figure requires three colors since the ends of the interval are joined: The first region and the last region are not adjacent before joining the ends, but they are adjacent after joining the ends. Compare Figure 1.2.5 with Figure 1.2.3.

Example 1.2.4.

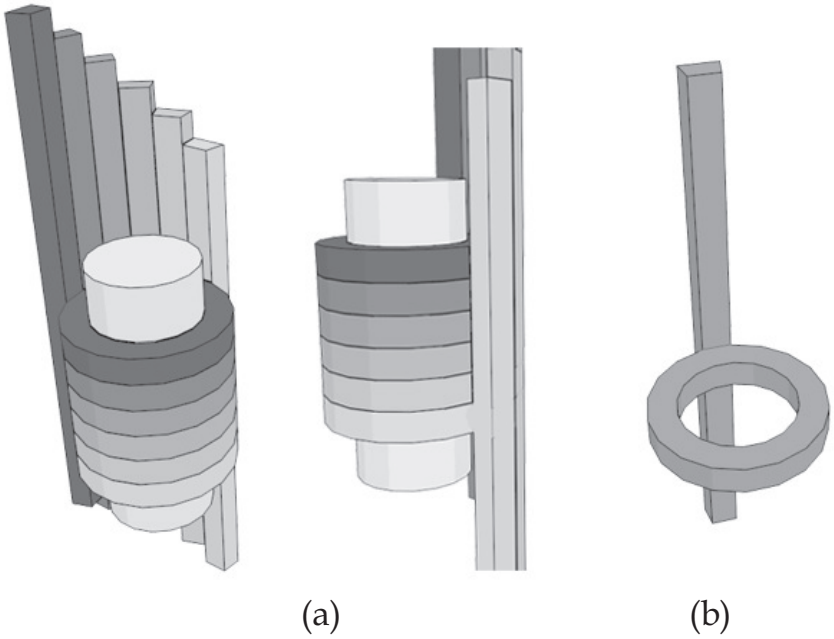

Figure 1.2.6: regions of 3-dimensional space

In Figure 1.2.6, (a) shows a map with seven regions (containing a cylindrical region) of 3-dimensional space. Most regions have shapes such as a sum of a tube and a stick like (b). This map requires seven colors since all the regions are pairwise adjacent. From this figure, we can guess the existence of maps that require infinitely many colors: by adjusting some parameters (width of stick, height of tube, and so on) of each region, and adding more regions of similar shape, we can make a map that requires as many colors as we want.

As the previous example shows, it seems pointless to consider this type of problem for maps of three or higher dimensional spaces. Instead, let us consider maps of 2-dimensional surfaces embedded into a 3-dimensional space. The next example shows that the coloring problems are more complicated and that the topological structures presumably influence the number of required colors.

Example 1.2.5.

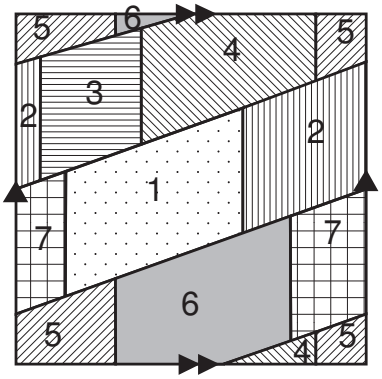

Figure 1.2.7: a map of torus requiring seven colors

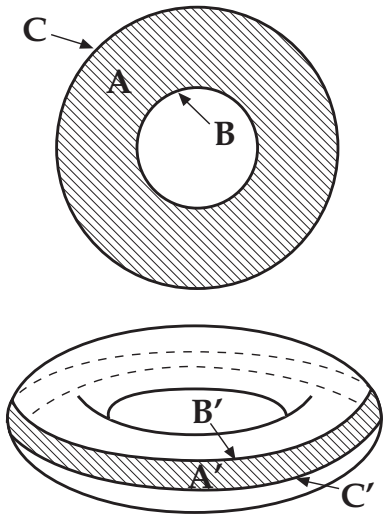

Figure 1.2.8: annulus in the plane and annulus in a torus

Figure 1.2.7 shows a map of a torus requiring seven colors. As we obtain a new adjacency condition by joining the ends of an interval (a region of the real line) in Example 1.2.3, we may get new adjacency conditions by pasting the opposite edges of a square (a region of the plane). In Figure 1.2.8, A is an annular region with its two frontiers B and C in a plane. A' is also an annular region with its two frontiers B' and C' in a torus. A separates its exterior into two pieces but A' does not. If there was a regions whose frontier intersects with B and another region whose frontier intersects with C in the plane, these two regions would never be adjacent. However, if there was a region whose frontier

intersects with B' and another region whose frontier intersects with C' in the torus, these two regions may be adjacent. Considering the fact that a cycle (or a polygon) is a deformation retract of an annulus [6], the plane case agrees with the Jordan curve theorem (Theorem 2.2.1), but the torus case does not.

1.3 Maps with finite regions

In this thesis, we treat maps that contain *finite regions* of the *plane*. However, a map may have two or more unbounded regions and we may want to find another map that conserves the adjacency of all the pairs of regions and has only one unbounded region.

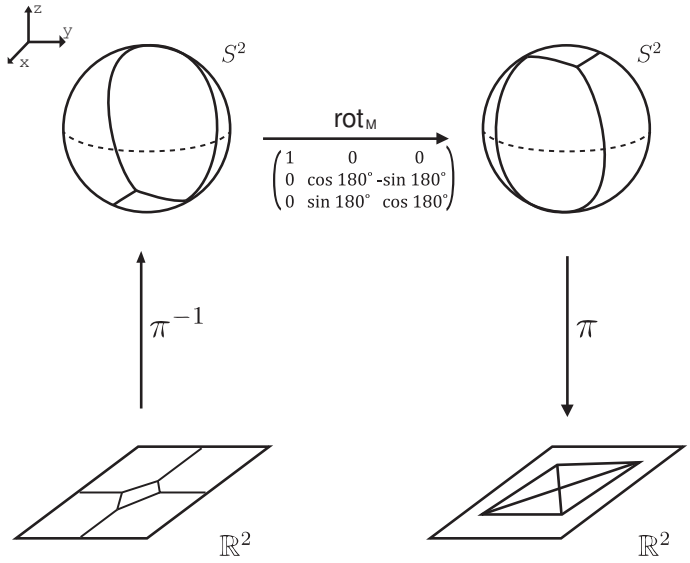

Figure 1.3.1: Example of getting a map that has only one unbounded region from another map that has four unbounded regions.

Recall the well-known stereographic projection $\pi : S^2 \setminus N \to \mathbb{R}^2$ where S^2 is a unit sphere, $N = (0, 0, 1)$ is the north pole of the sphere, and $\mathbb{R}^2$ is the plane isomorphic to the plane $\{(x, y, z)|z = 0\}$ in three-dimensional space [5]. With the help of this projection, we can find the wanted map. Assume that

a map M of the plane has two or more unbounded regions, say $R_1, R_2, \cdots R_n$. Considering that $\pi^{-1}(R_i)$ and $\pi^{-1}(R_j)$ are distinguished in S^2 $(1 \leq i \neq j \leq n)$, it is reasonable to treat the north pole N as a point contained in frontier of every $\pi^{-1}(R_k)$, where $k = 1, 2, \cdots, n$. Let $\mathrm{rot}_M : S^2 \to S^2$ be a rotation of S^2 such that the north pole N is contained in $\mathrm{rot}_M(\pi^{-1}(R))$ for some $R \in M$. Let us remove N and re-send regions of $\mathrm{rot}_M(\pi^{-1}(M))$ to the plane by the projection π so that we obtain a new map of the plane that contains only one unbounded region. This method does not break the adjacency of the regions since the projection π and the rotation rot_M are continuous and bijective. If we want a region with special properties to be unbounded, we can use a similar method.

Chapter 2

Graph representation

This chapter introduces the standard terminologies and some well-known facts from [2].

2.1 Graphs

A *graph* is a pair $G = (V, E)$ of sets such that $E \subseteq [V]^2$. The elements of V are the *vertices* of the graph G, the elements of E are its *edges*. The usual way to picture a graph is by drawing a dot for each vertex and joining two of these dots by a line if the corresponding two vertices form an edge.

A graph with vertex set V is said to be a graph *on* V. The vertex set of a graph G is referred to as $V(G)$, its edge set as $E(G)$. We shall not always distinguish strictly between a graph and its vertex or edge set. For example, we may speak of a vertex $v \in G$ (rather than $v \in V(G)$), and edge $e \in G$, and so on.

The number of vertices of a graph G is its *order* written as $|G|$, and the number of its edges is denoted by $\|G\|$.

A vertex v is *incident* with an edge e if $v \in e$. And e is an edge *at* v. The two vertices incident with an edge are its *endvertices* or *ends*, and an edge *joins* its ends. An edge $\{x, y\}$ is usually written as xy (or yx). If $x \in X$ and $y \in Y$, then xy is an $X - Y$ *edge*. The set of all $X - Y$ edges in a set E

is denoted by $E(X, Y)$; instead of $E(\{x\}, Y)$ and $E(X, \{y\})$ we simply write $E(x, Y)$ and $E(X, y)$. The set of all the edges in E at a vertex v is denoted by $E(v)$.

Two vertices x, y of G are *adjacent* or *neighbors*, if xy is an edge of G. Also, two edges $e \neq f$ are adjacent if they have an end in common. If all the vertices of G are pairwise adjacent, then G is *complete*. A complete graph on n vertices is a K^n. K^3 is called a *triangle*. Pairwise non-adjacent vertices or edges are called *independent*. More formally, a set of vertices or of edges is *independent* (or *stable*) if no two of its elements are adjacent.

Let $G = (V, E)$ and $G' = (V', E')$ be two graphs. We call G and G' *isomorphic*, and write $G \simeq G'$ if there exists a bijection $\varphi : V \to V'$ with $xy \in E \Leftrightarrow \varphi(x)\varphi(y) \in E'$ for all $x, y \in V$. Such a bijection φ is called an *isomorphism*.

A class of graphs that is closed under isomorphism is called a *graph property*. For example, 'containing of triangle' is a graph property.

We set $G \cup G' := (V \cup V', E \cup E')$ and $G \cap G' := (V \cap V', E \cap E')$. If $V' \subseteq V$ and $E' \subseteq E$ then G' is a *subgraph* of G (and G a *supergraph* of G'), written as $G' \subseteq G$. If $G' \subseteq G$ and G' contains all the edges $xy \in E$ with $x, y \in V'$, then G' is an *induced subgraph* of G and we say that V' *induces* or *spans* G' in G, write $G' =: G[V']$.

If U is any set of vertices (usually of G), we write $G - U$ for $G[V \setminus U]$. If $U = \{v\}$ is a singleton, we write $G - v$ rather than $G - \{v\}$. Instead of $G - V(G')$ we simply write $G - G'$. For a subset F of $[V]^2$ we write $G - F := (V, E \setminus F)$ and $G + F := (V, E \cup F)$. As above, $G - \{e\}$ and $G + \{e\}$ are abbreviated to $G - e$ and $G + e$. We call G *edge-maximal* with a given graph property if G itself has the property but no graph $G + xy := (V, E \cup \{xy\})$ does, for non-adjacent vertices $x, y \in G$.

The set of neighbors of a vertex v in G is denoted by $N_G(v)$ or briefly by $N(v)$. More generally for $U \subseteq V$, the neighbors in $V \setminus U$ of vertices in U are called *neighbors of* U, denoted by $N(U)$.

The *degree* $d_G(v) = d(v)$ of a vertex v is the number $|E(v)|$. The number $\delta(G) := \min\{d(v) | v \in V\}$ is the *minimum degree* of G, the number $\Delta(G) :=$

$\max\{d(v)|v \in V\}$ is its *maximum degree*. The number

$$d(G) := \frac{1}{|V|}\sum_{v\in V} d(v)$$

is the *average degree* of G. If we sum up all the vertex degrees in G we count every edge exactly twice: once from each of its ends. Thus

$$|E| = \frac{1}{2}d(G)\cdot|V|. \tag{2.1.0.1}$$

A *path* is a non-empty graph $P = (V, E)$ of the form

$$V = \{x_0, x_1, \cdots, x_k\},\ E = \{x_0x_1, x_1x_2, \cdots, x_{k-1}x_k\}$$

where the x_i are all distinct. The vertices x_0 and x_k are *linked* by P and are called its *ends*. The vertices $x_1, \cdots x_{k-1}$ are the *inner* vertices of P. The number of edges of a path is its *length*. We often refer to a path by the natural sequence of its vertices, writing $P = x_0x_1\cdots x_k$ and calling P a path *from* x_0 *to* x_k (as well as *between* x_0 and x_k).

Given sets A, B of vertices, we call $P = x_0\cdots x_k$ an $A - B$ *path* if $V(P)\cap A = \{x_0\}$ and $V(P)\cap B = \{x_k\}$. As before, we write $a - B$ path rather than $\{a\} - B$ path, etc.

If $P = x_0x_1\cdots x_{k-1}$ is a path and $k \geq 3$, then the graph $C := P + x_{k-1}x_0$ is called a *cycle*. As with paths, we often denote a cycle by its (cyclic) sequence of vertices such as $C = x_0, \cdots, x_{k-1}, x_0$. The *length* of a cycle is its number of edges (or vertices). An edge that joins two vertices of a cycle but is not itself an edge of the cycle is a *chord* of that cycle. An *induced cycle* in G, a cycle in G forming an induced subgraph, is one that has no chords.

A non-empty graph G is called *connected* if any two of its vertices are linked by a path in G. A maximal connected subgraph of G is called a *component* of G.

An acyclic graph, one not containing any cycles, is called a *forest*. A connected forest is called a *tree*. The vertices of degree 1 in a tree are its *leaves*.

Let $r \geq 2$ be an integer. A graph $G = (V, E)$ is called *r-partite* if V admits a partition into r classes such that every edge has its ends in different classes. Vertices in the same partition class must not be adjacent. Instead of '2-partite' one usually says *bipartite*. An r-partite graph in which every two vertices from different partition classes are adjacent is called *complete multipartite(r-partite) graph* and denoted by $K_{n_1,\cdots,n_k}$ where each n_i is the number of vertices of each partition class. If $n_1 = \cdots = n_r =: s$ we abbreviate this to K_s^r.

Let $e = xy$ be an edge of a graph $G = (V, E)$. By G/e we denote the graph (V', E') with vertex set $V' := (V \setminus \{x, y\}) \cup \{v_e\}$ and edge set

$$E' := \{vw \in E | \{v, w\} \cap \{x, y\} = \emptyset\} \cup \{v_e w | xw \in E \setminus \{e\} \text{or } yw \in E \setminus \{e\}\}.$$

More generally, if X is another graph and $\{V_x | x \in V(X)\}$ is a partition of V into connected subsets such that, for any two vertices $x, y \in X$, there is a $V_x - V_y$ edge in G if and only if $xy \in E(X)$, we call G an MX and write $G = MX$. The sets V_x are the *branch sets* of this MX. If $G = MX$ is a subgraph of another graph Y, we call X a *minor* of Y and write $X \preccurlyeq Y$.

If we replace the edges of X with independent paths between their ends (so that none of these paths has an inner vertex on another path or in X), we call the graph G obtained a *subdivision* of X and write $G = TX$. If $G = TX$ is the subgraph of another graph Y, then X is a *topological minor* of Y.

2.2 Planar graphs

A *straight line segment* in the Euclidean plane is a subset of $\mathbb{R}^2$ that has the form $\{p + \lambda(q - p) | 0 \leq \lambda \leq 1\}$ for distinct points $p, q \in \mathbb{R}^2$. A *polygon* is a subset of $\mathbb{R}^2$ that is the union of finitely many straight line segments and is homeomorphic to the unit circle S^1. A *polygonal arc*, or simply an *arc*, is a subset of $\mathbb{R}^2$ which is the union of finitely many straight line segments and is homeomorphic to the closed unit interval $[0, 1]$. The images of 0 and of 1 under such a homeomorphism are the *endpoints* of this polygonal arc, which *links* them and runs *between* them. If P is an arc between x and y, the point set $P \setminus \{x, y\}$ is the *interior* of P and we denote it by $\mathring{P}$.

Let $O \subseteq \mathbb{R}^2$ be an open set. Being linked by an arc in O defines an equivalence relation on O. The corresponding equivalence classes are again open and they are the *regions* of O. The *frontier* of a set $X \subseteq \mathbb{R}^2$ is the set Y of all points $y \in \mathbb{R}^2$ such that every neighborhood of y meets both X and $\mathbb{R}^2 \setminus X$.

Theorem 2.2.1 (Jordan Curve Theorem for Polygons)**.** For every polygon $P \subseteq \mathbb{R}^2$, the set $\mathbb{R}^2 \setminus P$ has exactly two regions. Each of these has the entire polygon P as its frontier.

A good account of the Jordan curve theorem is given in [8] or [6].

Lemma 2.2.2. Let P_1, P_2, P_3 be three arcs, between the same two endpoint but otherwise disjoint.

(i) $\mathbb{R}^2 \setminus (P_1 \cup P_2 \cup P_3)$ has exactly three regions, with frontiers $P_1 \cup P_2$, $P_2 \cup P_3$ and $P_3 \cup P_1$.

(ii) If P is an arc between an inner point of P_1 and an inner point of P_3, whose interior lies in the region of $\mathbb{R}^2 \setminus (P_1 \cup P_3)$ that contains P_2, then $\mathring{P} \cap \mathring{P}_2 \neq \emptyset$.

A *plane graph* is a pair (V, E) of finite sets with the following properties (the elements of V are again called *vertices*, those of E *edges*):

(i) $V \subseteq \mathbb{R}^2$;

(ii) every edge is an arc between two vertices;

(iii) different edges have different sets of endpoints;

(iv) the interior of an edge contains no vertex and no point of any other edge.

For every plane graph G, the set $\mathbb{R}^2 \setminus G$ is open. Its regions are the *faces* of G. We denote the set of faces of G by $F(G)$.

The subgraph of G whose point set is the frontier of a face f is said to *bound* f and is called its *boundary*, and we denote it by $G[f]$. A face is said to be *incident* with the vertices and edges of its boundary.

A plane graph G is called *maximally plane*, or just *maximal*, if we cannot add a new edge to form a plane graph $G' \supsetneq G$ with $V(G') = V(G)$. We call G

a plane *triangulation* if every face of G (including the outer face) is bounded by a triangle.

A *wheel* W_n is a graph with n vertices ($n \geq 4$) formed by connecting a single vertex v to all vertices of an $(n-1)$-cycle. The single vertex v is *hub* and the edges incident with v are called *spokes*. We simply denote it by W when we do not need to consider the length of the cycle. If W is a wheel in a graph G and h is its hub, we say that h *forms a wheel* W in G.

Proposition 2.2.3. A plane graph of order at least 3 is maximally plane if and only if it is a plane triangulation.

Theorem 2.2.4 (Euler's Formula)**.** Let G be a connected plane graph with n vertices, m edges, and l faces. Then

$$n - m + l = 2. \tag{2.2.4.1}$$

Corollary 2.2.5. A plane graph with $n \geq 3$ vertices has at most $3n-6$ edges. Every plane triangulation with n vertices has $3n - 6$ edges.

An *embedding* in the plane, or *planar embedding*, of an (abstract) graph G is an isomorphism between G and a plane graph H. The latter will be called a *drawing* of G. A graph is called *planar* if it can be embedded in the plane. A planar graph is *maximal*, or *maximally planar*, if it is planar but cannot be extended to a larger planar graph by adding an edge (but no vertex).

Proposition 2.2.6.

(i) Every maximal plane graph is maximally planar.
(ii) A planar graph with $n \geq 3$ vertices is maximally planar if and only if it has $3n - 6$ edges.

Theorem 2.2.7 (Kuratowski 1930; Wagner 1937)**.** The following assertions are equivalent for graphs G:

(i) G is planar;
(ii) G contains neither K^5 nor $K_{3,3}$ as a minor;
(iii) G contains neither K^5 nor $K_{3,3}$ as a topological minor.

2.3 Plane graph representation of a map

For a given map $V = \{v_1, v_2, \cdots, v_n\}$ of plane ($n \geq 4$), let us define $E := \{vw \subset V | v, w \text{ are adjacent }\}$, then $G = (V, E)$ forms a graph on V. Two adjacent regions (their closures share a point that is not a corner) in V are also adjacent (they are joined by an edge) in G. The graph G is planar since G has a drawing H in the following sense: let $V' := \{p_1, p_2, \cdots, p_n\}$ be a set of inner points of regions of the plane such that $p_i \in v_i$ for each $i = 1, 2, \cdots, n$. For an adjacent pair of regions, v_i and v_j, there exists a point q that is not a corner on a common frontier of v_i and v_j. Let $P_{i,q}$ be an arc from p_1 to q lying in v_1, and $P_{q,j}$ be an arc from q to p_2 lying in v_2. Let $P_{i,j} := P_{i,q} \cup P_{q,j}$ be the sum of the arcs and E' be the set of such $P_{i,j}$. Let us define a drawing H on V', setting $H := (V', E')$. Then, we can find (graph) isomorphism $\phi : V \to V'$ such that $\phi(v_i) = p_i$ and $\phi(v_i v_j) = P_{i,j}$. Therefore, we can convert the coloring problem of a map of the plane to the coloring problem of vertices of a plane graph.

A *vertex coloring* of a graph $G = (V, E)$ is a function $c : V \to S$ from the vertex set V to a set S such that $c(v) \neq c(w)$ whenever v and w are adjacent. The elements of the set S are called the *available colors*. The smallest integer k is the *(vertex-) chromatic number* of G if G has a *k-coloring* that is a vertex coloring $c : V \to \{1, \cdots, k\}$. We denote the chromatic number by $\chi(G)$. A graph G with $\chi(G) = k$ is called *k-chromatic*; if $\chi(G) \leq k$, we call G *k-colorable* [2]. We also call a vertex coloring with $|S| \leq n$ an *n-coloring solution* of G. Clearly, we are looking for the 4-coloring solutions.

Suppose that G and $G + xy$ are plane graphs such that $x, y \in G$ are non-adjacent vertices. Then all the coloring solutions of $G+xy$ are also the coloring solutions of G. Therefore, we have only to consider a maximal plane graph containing the given graph G. By Proposition 2.2.6, we shall not distinguish between maximal plane graphs and maximal planar graphs. And by Proposition 2.2.3, all the faces, including unbounded faces, of maximal plane graphs are bounded by triangles.

Proposition 2.3.1. Let G be a maximal plane graph with n vertices. The minimum degree $\delta(G)$ of G is at least 3.

Proof. By Corollary 2.2.5, G has $3n-6$ edges. A graph G' obtained by removing vertices or edges from G is also a plane graph. Suppose that there is a vertex v of degree 2. Set $G' := G - v$. Then the order $|G'|$ of G' is $n-1$ and the number $\|G'\|$ of its edges is $3n-6-2 = 3n-8$ since the degree of v is 2. However, $\|G'\| = 3n-8 > 3(n-1)-6$ and by Corollary 2.2.5, this contradicts the fact that G' is a plane graph. □

Proposition 2.3.2. Let G be a maximal plane graph with n vertices and $v \in G$ be a vertex of degree 3. Then the graph $G' := G - v$ is also a maximal plane.

Proof. The number of edges of G' is $(3n-6)-3 = 3(n-1)-6$ and by Corollary 2.2.5, G' is maximally plane. □

Suppose that G has a vertex v of degree 3. Let v_1, v_2, v_3 be the neighbors of v. If we can find a 4-coloring solution $c : V(G-v) \to S$ of $G-v$, then we also can find a 4-coloring solution of G by extending c: define $c(v) := s$ where $s \in S \setminus c(\{v_1, v_2, v_3\})$. By Proposition 2.3.2, $G-v$ is also maximally plane. Therefore, we have only to consider a maximal plane graph with a minimum degree at least 4.

Applying Equation 2.1.0.1 to a maximal plane graph G of order n with $\delta(G) \geq 4$, we could easily obtain that

$$\begin{aligned} d(G) &= \frac{2|E|}{|V|} && (2.3.2.1)\\ &= \frac{2(3n-6)}{n} && (2.3.2.2)\\ &= 6 - \frac{12}{n} && (2.3.2.3)\\ & && (2.3.2.4) \end{aligned}$$

$$\therefore \lim_{n\to\infty} d(G) = 6. \qquad (2.3.2.5)$$

This means that most vertices have degrees less than 6.

Digressively, the fact that a graph G of order at least 4 has a 4-coloring solution is equivalent to the fact that G is 4-partite: assume that G has a

4-coloring solution $c : V(G) \to S$. As an equivalence relation, the 4-coloring solution defines four or less equivalence classes on $V(G)$, and vertices of the same class are independent. Since the n-partite graph of order m is also m-partite where $m > n$, G is 4-partite. Conversely, assume that G is 4-partite. Let us define a coloring solution c' on $V(G)$ such that $c'(v) = c'(w)$ if v, w are in the same class and $c'(v) \neq c'(w)$ if v, w are in different classes. Then the number of the range of c' is at most 4.

If we can prove that if an arbitrary graph G is not 4-partite, then it contains K^5 or $K_{3,3}$ as a minor or a topological minor so that it is not a planar graph by the Theorem of Kuratowski and Wagner (Theorem 2.2.7), then the contraposition asserts that all planar graphs are 4-partite, and therefore have 4-coloring solutions.

Notation 2.3.3. We can assume that a map has only one unbounded region (Section 1.3). Let G be a maximal plane graph containing a subgraph that represents the given map, and let a vertex v of G represent the unbounded region of the map. The edges at v complicate the figure of G. Thus, in this thesis, the edges at v would be omitted and the vertex v would be at the proper location in the figures of the graph representation. And we would represent an edge as a smooth curve even if it is defined as a polygonal arc which is a union of finitely many straight line segments. We can easily distinguish the vertices in the figure of a graph since all vertices have degrees at least 4 by Proposition 2.3.2. See the following example.

Example 2.3.4.

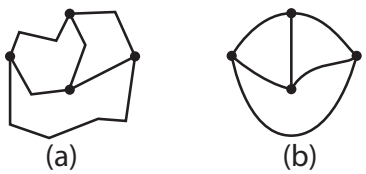

Figure 2.3.1: prefer (b) rather than (a)

Both (a) and (b) of Figure 2.3.1 denote K^4. We use a line segment for representing an edge. But when it is difficult, we use smooth curves such as (b) rather than a union of finite line segments such as (a).

Example 2.3.5.

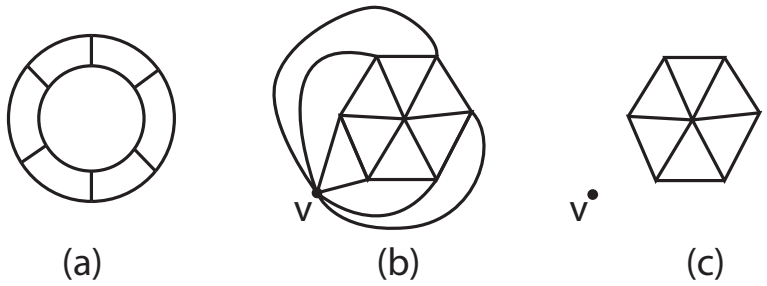

Figure 2.3.2: prefer (c) rather than (b) for a graph notation of map (a)

In Figure 2.3.2, (b) and (c) are the same graph representations of map (a). (c) is obtained from (b) omitting edges at the vertex v that represents the unbounded region of map (a). We prefer figures such as (c) rather than figures such as (b) for a graph representation of a map.

Proposition 2.3.6 (Five Color Theorem)**.** Every planar graph is 5-colorable.

The proof of the Five Color Theorem is given in [2].

Theorem 2.3.7 (Grötzsch 1959)**.** Every planar graph not containing a triangle is 3-colorable [2][9].

Chapter 3

Finding a 4-coloring solution

In this chapter, we will discuss how to color the vertices.

3.1 An introduction of a coloring method

Let G be a plane graph and v be a vertex of G. Assume that we will color the vertices of G one by one and that v have k number of colored neighbours at a moment. Let S denote a set of all colors that have been assigned to the neighbors. Clearly, $|S| =: n \leq k \leq d_G(v)$. Then, we say that v *has* n *colors on its neighbors*. If a vertex v has n colors on its neighbors and another vertex w has m colors on its neighbors and $m < n$, then we say that v has *more* colors on its neighbors than w or that w has *less* colors on its neighbors than v.

In a plane graph G, a *maximal triangle patch without a wheel* or simply a *patch* is a subgraph P of G with the following properties:

(i) P is a triangulation.
(ii) P does not contain a wheel graph.
(iii) If a triangle T_1 of G shares an edge with a triangle T_2 of P, then T_1 is also a triangle of P.

A patch P is *simple* if the closure of the sum of all its faces is a simple region of the plane. Let $\mathbb{P}$ be a set of all patches of G. A vertex v of G is *patch-free from* $\mathbb{P}$ or simply *patch-free* if v is not contained in any member of $\mathbb{P}$.

In a maximal plane graph G, an *independent hubset* or simply a *hubset* is a set H of independent vertices such that each member is a hub of a wheel in G. H is *maximal* if $G - H$ does not contain any wheels. Let us refer to the members of a hubset as *hubs* unless any confusion arises.

Let G be a plane graph, v be a vertex of G, $c : V(G) \to S$ be a vertex coloring of G, and A be a subset of $V(G)$. A usually does not contain v. A *set of adjacent colors* of v with respect to $c|_A$ is the image set $c|_A(A \cap N_G(v))$ in S and it is denoted by $AC_{c|_A}(v)$ or simply $AC_c(v)$ or $AC(v)$. The number $|AC_c(v)|$ is the *AC-number* of v.

Let G be a plane graph of order n. We would like to color its vertices one by one. Since we do not have enough colors, we should decide which vertex to color. If, at a moment, a non-colored vertex $v \in G$ has three colors on its neighbors and we have only four available colors, we have to color this vertex v rather than other vertices. It seems natural to select a vertex that has more colors on its neighbors rather than other vertices that have less colors on their neighbors, among the non-colored vertices. In such a point of view, we can consider a coloring method by constructing a finite sequence of pairs (G_k, c_k) in the following way:

(i) $G_1 = (\{v_1\}, \emptyset)$ where $v_1 \in V(G)$.
(ii) Define a vertex coloring, $c_1 : G_1 \to S$ on G_1.
(iii) Define an induced subgraph $G_k := G[G_{k-1} \cup \{v_k\}]$, where $v_k \in G \setminus G_{k-1}$ is a vertex such that the AC-number $|AC_{c_{k-1}}(v_k)|$ is maximum.
(iv) Define a vertex coloring, $c_k : G_k \to S$ such that $c_k|_{G_{k-1}} = c_{k-1}$ and $c_k(v_k) \in S \setminus AC_{c_{k-1}}(v)$.

The above process does not always work for a 4-element set S, say $\{1, 2, 3, 4\}$. To improve the above process, we can add other conditions for selecting v_k or a process for exchanging some assigned colors. However, this thesis suggests another way:

(i) Convert the given map to a graph and find a maximal plane graph that contains the graph.
(ii) Remove vertices of degree 3 from the maximal plane graph if they exist.

(iii) Find a *hubset* in the graph.
(iv) Color the vertices that are not the elements of the hubset with three available colors.
(v) Color the vertices contained in the hubset with the fourth color.
(vi) Finally, apply the coloring result to the given map.

3.2 Examples of coloring

Example 3.2.1. Find a coloring solution of the given map, Figure 3.2.1.

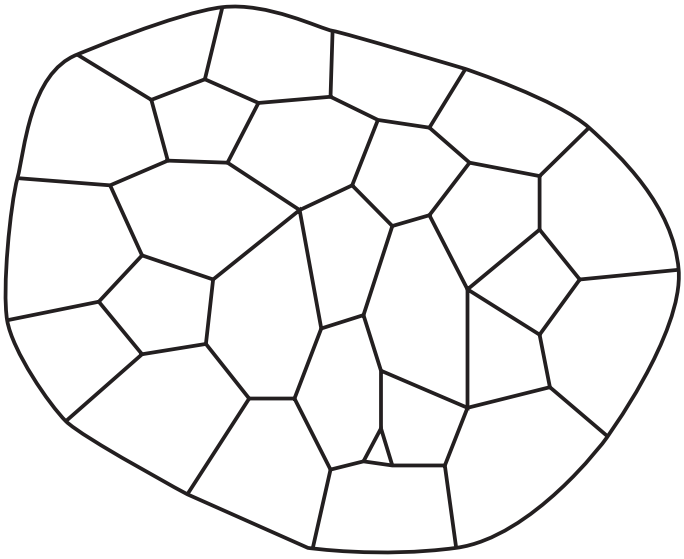

Figure 3.2.1: given map

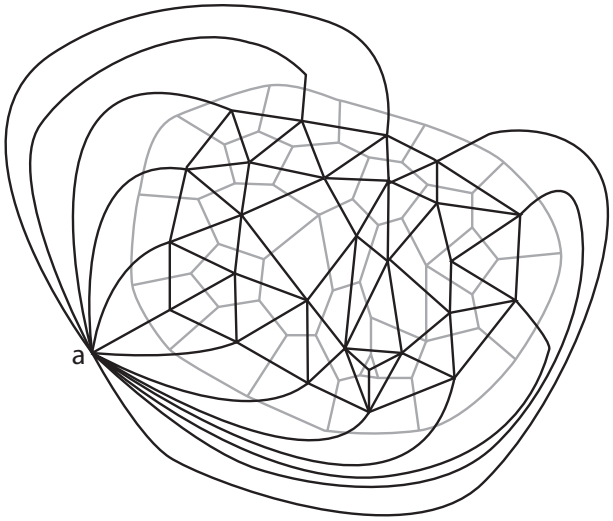

Figure 3.2.2: Step 1

Solution. Step 1: Find a graph representation of the given map in Figure 3.2.1 as described in Section 2.3.

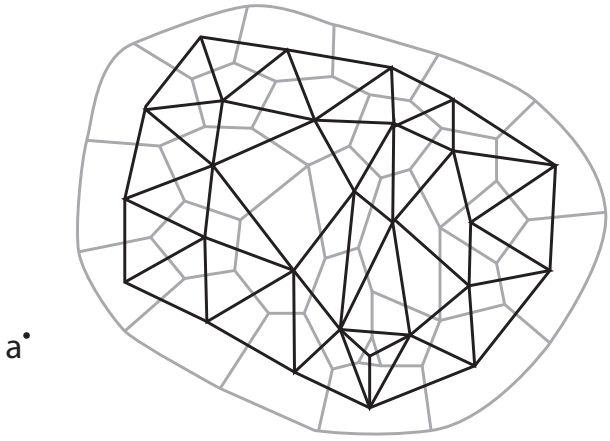

Figure 3.2.3: Step 2

Step 2: Omit the edges at a in the same way as Figure 2.3.3 in Example 2.3.5, where a is the vertex that represents the unbounded region of the given map.

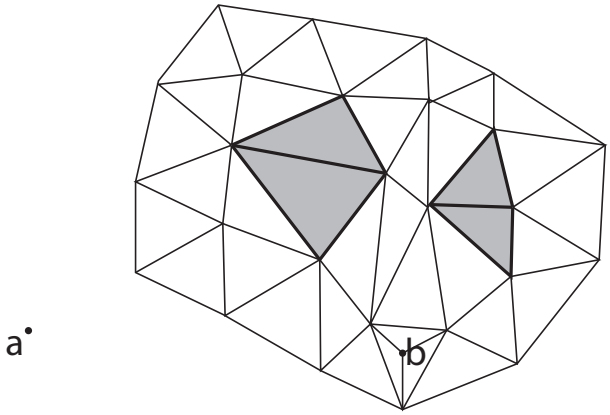

Figure 3.2.4: Step 3

Step 3: By Proposition 2.2.3, the graph in Figure 3.2.3 is not a maximally plane since it contains two faces whose boundaries are not triangles. Find a maximal plane graph containing Figure 3.2.3.

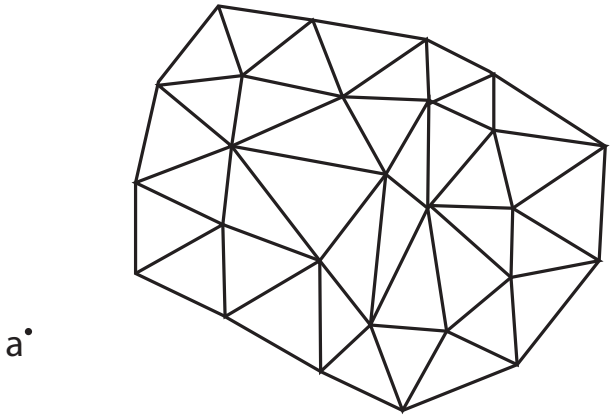

Figure 3.2.5: Step 4, a graph G

Step 4: By Proposition 2.3.2 and the arguments below, the vertex b in Figure 3.2.4 can be removed since its degree is three. Let G denote this graph.

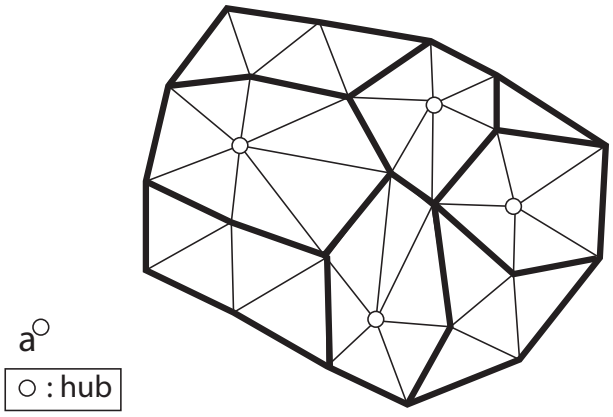

Figure 3.2.6: Step 5

Step 5: Find a hubset H. In Figure 3.2.6, the hubs are represented by white circles. Note that a is also a hub of a wheel.

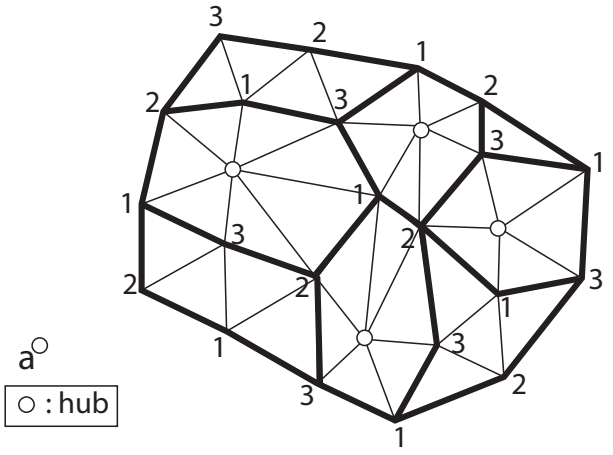

Figure 3.2.7: Step 6

Step 6: Following the vertex-selecting order discussed in Section 3.1, color the patches of $G - H$ with three available colors, say color 1, color 2, and color 3. Note that if $G - H$ had not contained any patches, this step would have been a specific example of the Grötzsch's theorem (Theorem 2.3.7).

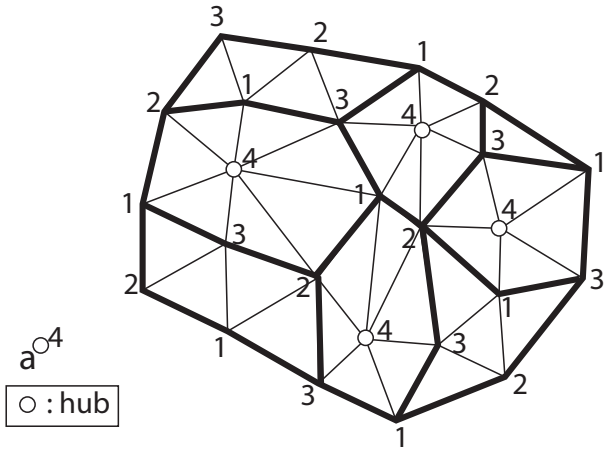

Figure 3.2.8: Step 7

Step 7: Color the hubs with color 4. Note that we never used color 4 at Step 6 and that all the hubs are independent by the definition of the hubset.

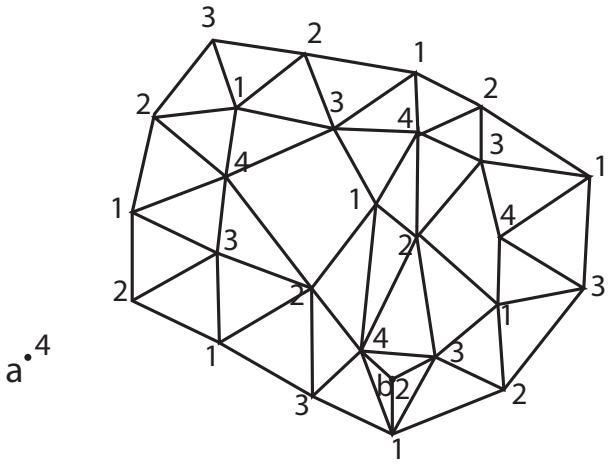

Figure 3.2.9: Step 8

Step 8: Apply the coloring result to the original graph in Figure 3.2.3 obtained at Step 2. Do not forget to color the vertex b that was removed at Step 4. We have only to assign b color 2 since $AC(b) = \{1, 3, 4\}$.

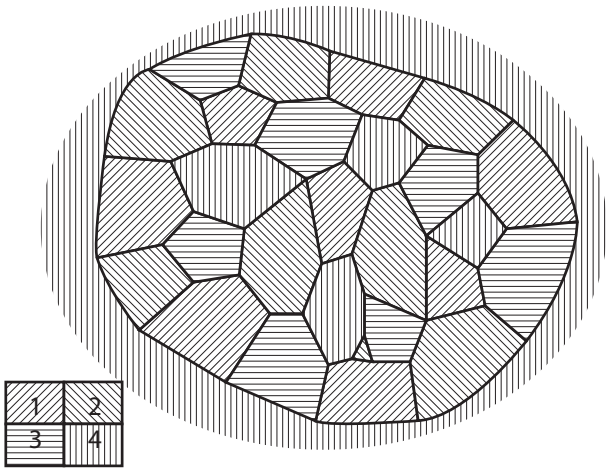

Figure 3.2.10: Step 9

Step 9: Color the region of the given map according to the vertex coloring we have found. □

The coloring process such as the one in Example 3.2.1 does not always success. See the following examples.

Example 3.2.2.

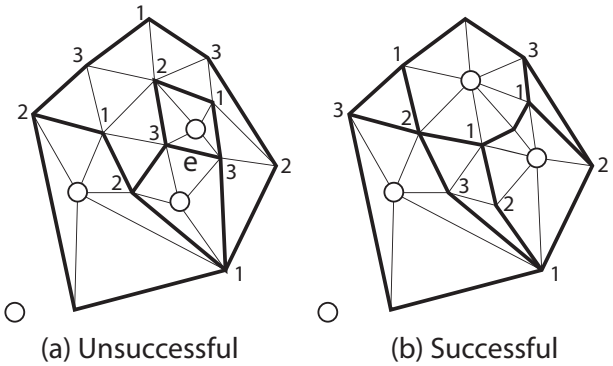

Figure 3.2.11

Figure 3.2.11 shows the different choices of hubsets for the same graph. (a) shows the wrong choice of hubset – the ends of an edge e have the same color.

Example 3.2.3.

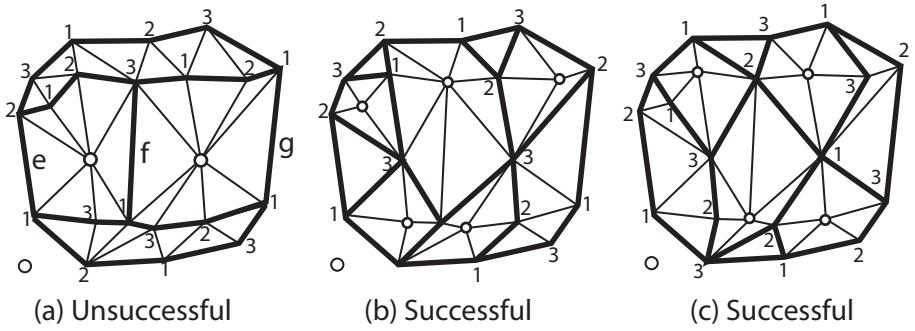

Figure 3.2.12

Figure 3.2.12 also shows the different choices of hubsets for the same graph. The ends of one of the edges $\{e, f, g\}$ in (a) have the same color: lower ends of these edges must have the same color, color 1 as figured, but the upper ends of the edges must have pairwise different colors. Since we have only three available colors, it is unsuccessful by the pigeonhole principle.

Example 3.2.4.

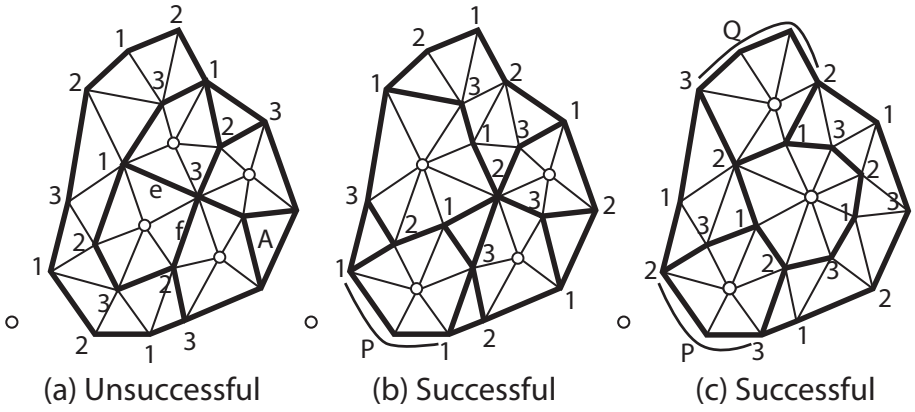

Figure 3.2.13

In Figure 3.2.13 (a), every vertex of a triangle A has color 3 on its neighbors. Thus, it is unsuccessful since we have only three available colors for the vertices of the triangle A. A path $e \cup f$ of length 2 has color 1 on one end and color 2 on the other end. Therefore, the only inner vertex has to be assigned

color 3. Every inner vertex of paths P and Q plays a role as a buffer that prevents the failure of coloring since it always has two neighbors except hubs so that it has at most two colors on its neighbors before coloring hubs.

Example 3.2.5.

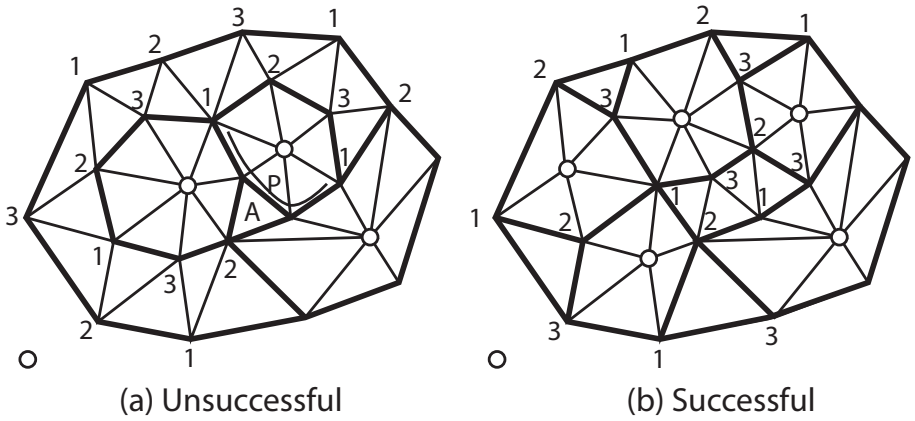

Figure 3.2.14

In Figure 3.2.14 (a), P is a path of length 3 and both of its ends have been assigned color 1. Therefore, one of the two inner points of P has to be colored with color 2. However, the triangle A contains the two inner points of P, and has another vertex that already has been colored with color 2. Therefore, (a) is unsuccessful.

3.3 Observations

Definition 3.3.1. Let G be a plane graph containing at least one triangle. An *overedge sequence* is a sequence $\{v_1, v_2, \cdots\}$ of vertices such that there exists an edge e_i that is the opposite edge of vertex v_i in a triangle T and the opposite edge of vertex v_{i+1} in a triangle T' of G. Note that T and T' can denote the same triangle, and that v_i and v_j can denote the same vertex even if $i \neq j$. Two vertices v_1 and v_2 of G are *overedge* if there exists an overedge sequence containing both v_1 and v_2.

Proposition 3.3.2. For the vertices in a patch P, the overedge relation is an equivalence relation.

Proof. (Reflexivity) Since P is a triangulation, for any vertex v of P, there exists a triangle T that contains v. Let e be an opposite edge of v in T, then a sequence $\{v, v, \cdots\}$ is an overedge sequence since e is an opposite edge of every element of the sequence in T.
(Symmetry) If v_1 and v_2 are members of a sequence, then v_2 and v_1 are also members of the sequence.
(Transitivity) Assume that v_i and v_j are members of an overedge sequence A, and v_j and v_k are members of an overedge sequence B. Then we have a subsequence $\{v_i, \cdots, v_j\}$ of A and another subsequence $\{v_j, \cdots, v_k\}$ of B. By joining the two sequences, we get an overedge sequence $\{v_i, \cdots, v_j, \cdots, v_k\}$. □

Proposition 3.3.3. The overedge relation defines three equivalence classes on a simple patch P.

Proof. Let P be a simple patch and T_1 be a triangle of P. We have three distinct pairs, each of which is a form of (v_1^i, e_1^i) such that v_1^i is an opposite vertex of edge e_1^i where $i = 1, 2, 3$. Let T_2 be another triangle of P that shares an edge e_1^i with T_1. There is only one opposite vertex of the edge e_1^i in T_2 and let v_2^i denote this vertex. Similarly, in a triangle T_j of P that shares an edge e_k^i with $T_1 \cup \cdots \cup T_{j-1}$, let the opposite vertex of e_k^i be denoted by v_{k+1}^i for some index k. Note that T_j shares only one edge with $T_1 \cup \cdots \cup T_{j-1}$ and that there exists only one opposite vertex of e_k^i in T_j since P is simple and it does not contain any wheels. Therefore, a vertex v of $T_1 \cup \cdots \cup T_j$ cannot have two notations; for example, $v_{k_1}^{i_1} = v = v_{k_2}^{i_2}$ where $i_1 \neq i_2$, so that the number of equivalence classes is less than three. □

Corollary 3.3.4. A simple patch P requires exactly three colors to color its vertices. Two vertices v and w of P have the same color if they are overedge.

Proof. Assume that the two vertices v and w are overedge in a simple patch P. Then v and w are non-adjacent in P since P does not contain K^4 that is

a wheel graph. (However, v and w can be adjacent in a supergraph of P. See the Example 3.3.7.) Therefore, we can assign the same color to vertices in the same equivalence class. By Proposition 3.3.3, we need exactly three colors. □

Assume that we find a maximal hubset H of a plane graph G. Consider the graph $G - H$. A patch has six numbers of 3-coloring solutions since a triangle also has six numbers of 3-coloring solutions. And for a colored patch P, if we want a vertex $v \in P$ that was already colored to have a different color, we can just exchange the color of $[v]$ and the color of $[w]$, where $[v]$ and $[w]$ are the different equivalence classes containing v and w, respectively.

Let us consider how a colored patch affects the colorings of another patch in $G - H$. Assume that P and Q are two patches and P has been colored. If they share a vertex v, then we have two choices of 3-coloring solutions for Q. If the patches are linked by an edge vw where $v \in P$, $w \in Q$, then we have four choices of 3-coloring solutions for Q. If the patches are linked by a $P-Q$ path vuw of length 2, where $v \in P$, $w \in Q$, then the vertex coloring of P does not affect the choice of the vertex coloring of Q. Even if the ends of the path have different colors, we can always assign the third color to the inner vertices of the path since the degree of all the inner vertices is 2 in $G - H$. This means that a coloring of P never affects the coloring of Q if all $P - Q$ paths have lengths at least 2. Such a path of length at least 2 can be seen as a graph representation of an interval of the real line such as Figure 1.2.4. Three available colors are enough to color maps of such an interval. When we finish coloring some patches and if there does not exist non-colored vertex of AC-number 2, we should select a vertex that is contained in a non-colored patch linked to the sum of colored patches by a path of shorter length. Note that a vertex that is shared by two patches can be thought as a linking path of length 0. However, which vertices would be the inner vertices of such a path linking two patches in $G - H$? The answer is vertices of degree 4 in G.

Proposition 3.3.5. Let G be a maximal plane graph with $\delta(G) \geq 4$ and H be a maximal hubset of G. Let W^1 and W^2 be two wheels such that $h_1, h_2 \in H$ are hubs of W^1 and W^2, respectively, and W^1 and W^2 share a path P of length at least 2. Then the degree of the inner vertices of P is 4 in G.

Proof. Let v be an inner vertex of P. Then we obtain $d_G(v) \geq 4$ since $\delta(G) \geq 4$. The vertex v has two neighbors v_1 and v_2 in P and two neighbors h_1 and h_2 that are not vertices in P. However, v does not have any other neighbors: each vh_iv_jv $(i, j = 1, 2)$ forms a triangle. If a vertex w is inside of one of the triangles, the degree $d_G(w)$ of w is 3 since G is maximally planar. This contradicts the assumption that $\delta(G) \geq 4$. If w is outside all the triangles, this also contradicts the Jordan curve theorem (Theorem 2.2.1) since $h_1v_1h_2v_2h_1$ forms a cycle containing all the triangles. □

For a maximal hubset H of a maximal plane graph G, we can always color the patch-free vertices of $G - H$ with 3 available colors if they have degree 4 in G. However, if a patch-free vertex v of $G - H$ has a degree at least 6 in G, we are not always able to color the vertex v. See the next example.

Example 3.3.6.

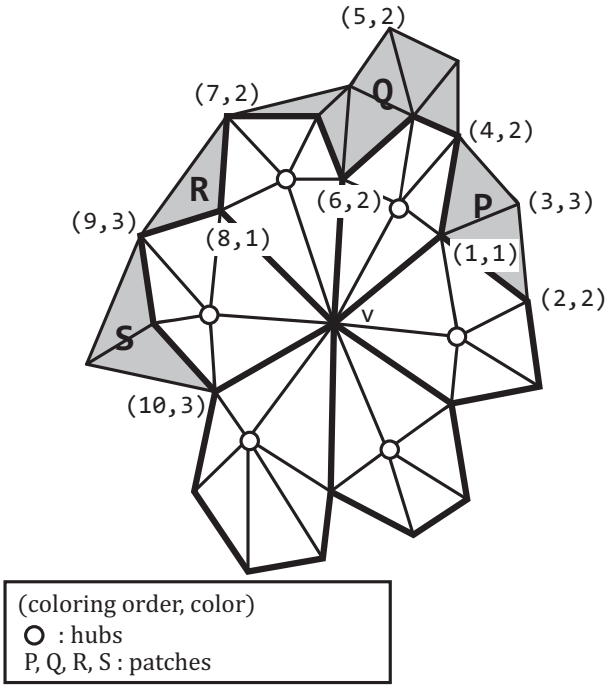

Figure 3.3.1

Due to the wrong choice of a maximal hubset, finding a 3-coloring solution is unsuccessful (Figure 3.3.1). Let v_i denote the i-th colored vertex in the figure. We start coloring v_1 with color 1 and we have no problem until the vertex v_4 is colored with color 2. In patch Q, v_5, v_6 and v_7 should be colored with color 2 since they are overedge with v_4 which has been colored with color 2. When we color the vertex v_8, we have to color it with color 1 because if we color it with color 3 then v which is not a hub, would have three colors on its neighbors. Also, v already has hubs that would be colored with color 4 as its neighbors; thus, v would require the fifth color. This implies that v_9 should be colored with color 3. Since v_{10} is overedge with v_9, it should be colored with color 3. Finally, v requires the fifth color.

A patch-free vertex such as v in Figure 3.3.1 has a degree at least 6. However, Equation 2.3.2.5 says that such a vertex does not appear frequently. Despite this, we can usually find another 3-coloring solution by exchanging colors of two classes of some patches. As Example 3.3.6 states, if there is no 3-coloring solution, we should take another maximal hubset.

Example 3.3.7.

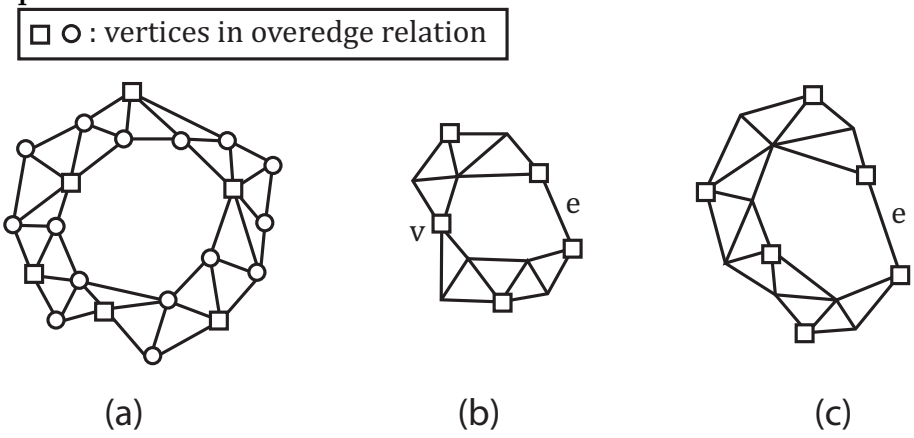

Figure 3.3.2

The patch of (a) in Figure 3.3.2 is not simple and the overedge relation cannot define three equivalence classes. (b) shows that two ends of an edge e that are contained in different patches. They are overedge with a vertex v so that they

are also overedge. (c) shows a situation similar to the case of (b), two vertices are overedge in a patch but they are linked by an edge e.

Situations such as the next example frequently occur.

Example 3.3.8. Consider a path P of length 3 whose ends are colored with the same color, say color 1. Then the two inner vertices v_1 and v_2 have to be colored with color 2 and color 3 or with color 3 and color 2, respectively. See Figure 3.2.14 in Example 3.2.5.

The previous example shows that although the length of a path (that links two patches) is at least 2, if some inner vertices of the path have degrees more than 4, we have to consider these inner vertices when we color the patches containing some of these inner vertices or the patches containing other vertices that are adjacent to some of these inner vertices.

What should we also consider? Let G be a maximal plane graph and H be a maximal hubset in G. Let G' be a graph on $\mathbb{P} \cup \bar{\mathbb{P}}$ where $\mathbb{P}$ is the set of patches in $G - H$ and $\bar{\mathbb{P}}$ is the set of patch-free vertices that have degree at least 2 in $G-H$. Let us define an edge set $E(G')$ of G' such that $vw \in E(G')$ if $v, w \in V(G')$ share a vertex or are linked by an edge. Obviously, G' is a plane graph. The difficult concepts are perhaps the cycles in G' because a coloring of a vertex $v \in G'$ affects the coloring of other vertices along the cycles of G' that contain v and it affects v again. Since the cycles containing v may form complex structures in G' or the colorings of two neighbors of v may affect each other, we should consider the structure when we color v. However, we should only consider the subgraphs of G' that are isomorphic to the subdivisions of two complete graphs, K^3 and K^4, since the Theorem of Kuratowski and Wagner (Theorem 2.2.7) asserts that any subdivision of K^n for $n \geq 5$ does not appear in the planar graph G'. Let K be a subdivision of a complete graph containing v in G'. Set a weight on each edge of K for $vw \in E(K)$: if v and w share n vertices and are linked by m edges, then set the weight $W(vw)$ of $vw \in E(K)$ as $3n+2m$. The quantity $(\sum_{e \in E(K)} W(e))/|K|$ can be a standard for guessing the success rate of coloring locally. The heavier weight, the harder it is for the coloring to be successful.

We have to choose a "good" maximal hubset. Most of the unsuccessful cases of the previous examples contain a patch comparatively bigger than the patches of the successful cases if we define the size of a patch as the number of its faces. It seems better to break the "big" patches into smaller pieces. Assume that we construct a maximal hubset H by adding new members (hubs) one by one. To avoid generating "big" patches or non-simple patches, we can select a vertex as a new hub forming a wheel that shares as many edges as possible with some other wheels that are formed by the hubs that we already found.

It would also be better to take a hubset H in plane graph G such that the patches in $G - H$ are linked by paths of longest possible length. By Proposition 3.3.5 and the arguments before it, it seems better not to select vertices of degree 4 in G as members of the hubset. However, for a maximal plane graph G containing vertices of degree 4, assume that we have found a "good" hubset H so that we have a 4-coloring solution c that assigns the members of H color 4 and assigns some other vertices of degree 4, without loss of generality, color 1. Note that the other vertex coloring c' obtained from c by exchanging color 1 and color 4 is also a 4-coloring solution of G. Inversely, we can find another hubset H' by taking vertices that are assigned color 4 by c' as its members. H' contains vertices of degree 4 in G, and it may not be a maximal.

We suggest the following conditions for finding a maximal hubset.

(i) Select a vertex as a new hub such that the wheel formed by this hub shares as many edges as possible with other wheels formed by hubs that are already selected.

(ii) If a vertex v is overedge with a hub h that we already found, and if v and h are simultaneously adjacent to other 4-degree vertices, select the vertex v as new hub rather than other vertices. If there exist multiple vertices such as v, select the one that has more neighbors of degree 4.

(iii) Choose a maximal hubset H such that $G - H$ would contain less number of faces that are bounded by triangles.

Chapter 4

Computer experiments

Since the method for finding the vertex coloring is not always successful, we would like to check the success rate. This chapter provides some source codes of computer programs and the explanations of them. You can also refer to the comments that are contained in the codes. The codes are written in MATLAB-like language. These codes may work for MATLAB, FreeMat, Octave, and so on. In fact, the codes are written and checked in FreeMat. This thesis, therefore, recommends running the codes in FreeMat.

The *adjacency matrix* $A = (a_{ij})_{n\times n}$ of a graph $G := (V, E)$ is defined by

$$a_{ij} := \begin{cases} 1 & \text{if } v_i v_j \in E \\ 0 & \text{otherwise.} \end{cases}$$

For convenience, we do not distinguish a graph and its adjacency matrix as long as no confusion arises. For example, the adjacency matrix of a graph G is denoted by G, and the graph whose adjacency matrix is A is denoted by A.

4.1 Process flow and the result

The process flow of computer codes is described as follows:

(i) Generate a random maximal plane graph. In fact, the codes generate a random adjacency matrix A of a maximal plane graph.

(ii) Get B by removing the vertices of degree 3 from A and find a hubset H of B. Even if the thesis suggest three conditions for finding the maximal hubset at the end of Section 3.3, the codes takes only condition (i) to find new hubs while constructing a hubset because implementing all the conditions complicates the codes.

(iii) Find patches in $B - H$.

(iv) Construct another adjacency matrix C of a virtual graph whose vertices are equivalence classes of the overedge relation or patch-free vertices of degree more than 4 in B.

(v) Generate the list of probable colorings of C with three available colors.

(vi) If a vertex coloring of C exists in the list then return `success=1`; otherwise return `success=0`.

For a random maximal plane graph of order 40, we obtained the success rate: (number of successes)/(number of tries)= $923/940 \approx 98\%$.

4.2 Source codes

hoRun.m :

```
numSuccess = 0;
wrongEqClassVertexTable = 0;

for loopCount=1:1000
% Iterate 1000 times.
 disp('free memory')
 clear A faces maxIndHubset tempMaxIndHubset B indexB facesB
    patch patchFree patchFree4Degree sizeEqClass
    facesRemovedHub C eqClassVertexTable colorVector success

 disp('Generate random adjacency matrix of a maximal plane
    graph');
 [A,faces] = hoGenRandAdjMat(40);

 disp('Remove vertices of degree 3 and find hubset');
```

```
[maxIndHubset,tempMaxIndHubset,B,indexB,facesB]=
   hoSelectArbMaxIndHubset(A,faces);

disp('Find patches and patch-free vertices');
[patch,patchFree,patchFree4Degree,sizeEqClass,facesRemovedHub
   ]=hoFindPatches(B,tempMaxIndHubset,facesB);

disp('Create a virtual adjacency matrix');
[C,eqClassVertexTable] = hoVirtualAdjMat(patch,patchFree,B,
   tempMaxIndHubset);

disp('Check the eqClassVertexTable');
for i=1:size(eqClassVertexTable,1)
 if(eqClassVertexTable(i,1)==eqClassVertexTable(i,2)|
    eqClassVertexTable(i,1)==eqClassVertexTable(i,3)|
    eqClassVertexTable(i,2)==eqClassVertexTable(i,3))
  disp('wrong eqClassVertexTable');
  wrongEqClassVertexTable=1;
  break;
 end
end

if(wrongEqClassVertexTable==1)
 wrongEqClassVertexTable=0;
 continue;
end

disp('Generate a list of vertex coloring');
colorVector=hoGenColorList(C,eqClassVertexTable);

disp('Check the list of vertex coloring');
success=hoCheck3Colorable(C,colorVector);

if(success==1)
 numSuccess=numSuccess+1;
end

disp('loopCount :')
loopCount
```

```
 disp('numSuccess : ')
 numSuccess
end

disp('success rate :')
numSuccess/loopCount
```

hoGenRandAdjMat.m :

```
function [A,faces] = hoGenRandAdjMat(numVertex)

if(numVertex < 5)
% Treat graphs of order > 4
 disp('number of vertices is less than 4');
 exit();
end

% Maximal plane graph has 3*(number of vertices)-6 edges.
numEdge = 3*numVertex-6;

% A will be an adjacency matrix
A = zeros(numVertex,numVertex);

% To draw maximal graph, we first draw a triangle.
A(2,1) = 1;
A(3,1) = 1;
A(3,2) = 1;

% Faces will be the index set of faces. We got a triangle.
faces(1,1:3) = [1,2,3];

% numTriangle is the number of the faces.
numFace = 1;

% Call the vertices incident to the unbounded face by external
% vertices. Boundary(cycle) of the unbounded face for the
% triangle which formed by first, second and third drawn
% vertices.
```

```
boundaryUnboundedFace = [1,2,3];

% Length of the boundary of the unbounded face. When a new
% vertex and new edges incident to the new vertex are added,
% this value will increse by (3 - (number of the new edges)).
lengthBoundary = 3;

% Now we add a new vertex.
% We draw a new vertex in the unbounded face and draw edges to
% vertices already drawn. the availableNumEdge is maximum
% number of edges incident to new vertex. If a new vertex
% added there need at least 2 edges incident to the new
% vertex. And we already used 3 edges for the first
% triangle. Since the new vertex would drawn in unbounded
% face, the number of edges also bounded by the number of
% vertices of the unbounded face.
%
% original code :
% availableNumEdge =
% min(numEdge-3-2*(numVertex-3),lengthBoundary);
% optimized code :
%
availableNumEdge = min(numEdge-2*numVertex+3,lengthBoundary);

for i=4:numVertex-1
% i-th vertex would have currentNumEdge number of edge with
% vertices already drawn. And we know 1 < currentNumEdge <
% availableNumbEdge+1, so we choose randomly between 2 and
% availableNumbEdge
 if(availableNumEdge==2)
  currentNumEdge = 2;
 else
  currentNumEdge = rem(floor(10000*rand(1,1)),availableNumEdge
     -2)+2;
 end

% The vertices adjacent to the new i-th vertex forms a path
% P in the boundary(cycle) of the unbounded face. Thus we
% can choose the path by choosing its end. Choose it randomly
```

```
% among external vertices.
 currentStartingVertex = rem(floor(10000*rand(1,1)),
    lengthBoundary)+1;

% The boundary of the unbounded face is a cycle but we
% denoted it as vector whose name is boundaryUnboundedFace.
% To use this vector as like a cycle we write this vector
% twice and take some needed part of it.
 tempBoundary = [boundaryUnboundedFace,boundaryUnboundedFace(1:
    size(boundaryUnboundedFace,2)-1)];

% Initiate currentAddedRow which would be i-th row of
% Adjacency matrix A.
 currentAddedRow = zeros(1,i-1);

% From the informations of currentNumEdge and
% currentStartingVertex, construct currentAddedRow.
 for j=1:currentNumEdge
  index=j-1+currentStartingVertex;
  currentAddedRow(tempBoundary(index)) = 1;
 end

% Add new i-th row to the adjacency matrix A.
 A(i,1:i-1) = currentAddedRow;
 clear currentAddedRow;

% And we also have new faces. Add them
 for j=1:currentNumEdge-1
  numFace = numFace+1;
  index=j-1+currentStartingVertex;
  faces(numFace,1:3)=[tempBoundary(index),tempBoundary(index+1)
     ,i];
 end

% lengthBoundary was changed. Reset it.
 lengthBoundary = lengthBoundary+3-currentNumEdge;

% The boundary of the unbounded face was also changed. Its
% new boundary is (an end of P) - (the new vertex) -
```

```
% (another end of P) - (vertices on boundary \ vertices on P))
 boundaryUnboundedFace = [tempBoundary(currentStartingVertex),i
    ,tempBoundary(currentStartingVertex+currentNumEdge-1:size(
    tempBoundary,2))];
 boundaryUnboundedFace = boundaryUnboundedFace(1:lengthBoundary
    );

 clear tempBoundary;

% availableNumEdge could also be changed. Reset it.
 availableNumEdge = min(numEdge-3-currentNumEdge-2*(numVertex-i
    ),lengthBoundary);
end

% We construct the last row of the adjacency matrix A.
currentAddedRow = zeros(1,numVertex);

% The last vertex is adjacent with all vertices of the
% boundary of the unbounded face.
for j=1:lengthBoundary
 currentAddedRow(boundaryUnboundedFace(j))=1;
end

% Add the last row to the adjaency matrix A.
A(numVertex,:) = currentAddedRow;

% Add the faces.
for j=1:lengthBoundary-1
 numFace = numFace+1;
 faces(numFace,1:3)=[boundaryUnboundedFace(j),
    boundaryUnboundedFace(j+1),numVertex];
end

numFace=numFace+1;
faces(numFace,1:3)=[boundaryUnboundedFace(1),
   boundaryUnboundedFace(lengthBoundary),numVertex];

% A is symmetric. For convenience we do not consider this,
% and we just only construct the lower part of A. By adding
```

```
% the transpose matrix of A, we complete it.
A=A+A';

% Sum of all '1' of A have to be equal to the twice of
% number of edges. Check it.
sum(sum(A)) == numEdge*2;
```

hoRemove3degree.m :

```
function [B,indexB,faceB] = hoRemove3degree(A,indexB,faces)
% Remove vertices of 3 degree from given adjacency matrix A.
% give indexB = 0 at first.
% When we remove a 3-degree vertex, another vertex could be
% 3-degree vertex (The vertex of degree 4 which is adjacent
% to the removed vertex). Thus we remove a 3-degree vertex,
% we should restart the removing operation on result matrix.
% In the given adjacency matrix A, i-th row(or column)
% represent i-th drawn vertex. However, in the result matrix
% B, the i-th row does not represent i-th vertex, anymore.
% Thus indexB would tell us, which vertex i-th row of B
% represent for. For example if indexB(3)=5 then the third
% row of B contains the adjacency information of fifth
% vertex of which adjacency information in A is obviously
% fifth row of A.
% When we remove 3-degree vertex, the three faces that are
% incident to the vertex were removed and there appear a new
% face bounded by the triangle containing the three neighbours
% of the removed vertex.

% Initiate result matrix.
B = A;

% Get size of B for FOR sentence.
numVertex = size(B,1);

% Initiate faces.
faceB = faces;

% Get number of faces that we have found.
```

```
numFace = size(faceB,1);

% Initiate indexB.
if(indexB == 0)
 indexB = 1:numVertex;
end

for i = 1:numVertex
% If we find 3-degree vertex, remove it and reset the indexB.

 if(sum(B(i,:))==3)
% If the sum of i-th row of B is equal to 3, the i-th vertex
% of B has the degree 3. We remove the i-th row and the i-th
% column of B.

  if(i==1)
   B = B(2:numVertex,2:numVertex);
   indexB = indexB(2:numVertex);
  elseif (i==numVertex)
   B = B(1:numVertex-1,1:numVertex-1);
   indexB = indexB(1:numVertex-1);
  else
   B = [B(1:i-1,1:i-1),B(1:i-1,i+1:numVertex);B(i+1:numVertex
       ,1:i-1),B(i+1:numVertex,i+1:numVertex)];
   indexB = [indexB(1:i-1),indexB(i+1:numVertex)];
  end

  j=1;
  tempCount = 0;

% We delete the faces containing the vertices of degree 3,
% that were removed at previous IF sentence.
  while(j<numFace+1)
   tempFaceBit =hoConvertIndexToBit(faceB(j,:),numVertex);

% Since we have deleted the i-th 3-degree vertex, we remove
% the faces that is incident to the i-th vertex.
   if(tempFaceBit(i)==1)
    if(j==1)
```

```
     faceB = faceB(2:numFace,:);
    elseif(j==numFace)
     faceB = faceB(1:numFace-1,:);
    else
     faceB = [faceB(1:j-1,:);faceB(j+1:numFace,:)];
    end
    numFace = numFace-1;
    tempCount = tempCount+1;
   else
    j = j+1;
    tempCount = 0;
   end
  end

% We add a new face that is bounded by the triangle whose
% vertices are the neighbors of i-th vertex.
  faceB(numFace+1,:) = hoConvertBitToIndex(A(i,:));
  numFace = numFace+1;

% Convert the index of A to the index of B.
% Since we removed a vertex, the vertices that has indices
% greater than the index of the removed vertex have indices
% decreased.
  for j=1:numFace
   for k=1:3
    if(faceB(j,k)>i)
     faceB(j,k)=faceB(j,k)-1;
    end
   end
  end

% Stop the process and restart same removing process with B
% again.
  [B,indexB,faceB] = hoRemove3degree(B,indexB,faceB);
  break;
 end
end
```

hoSelectArbMaxIndHubset.m :

```
function [maxIndHubset,tempMaxIndHubset,B,indexB,facesB]=
    hoSelectArbMaxIndHubset(A,faces)
% A is an adjacency matrix of a maximal plane graph G.
% maxIndHubset(i) = 1 if i-th vertex of G is contained
% in maximal independent hubset which we would construct.

% Remove vertices of degree 3.
[B,indexB,facesB] = hoRemove3degree(A,0,faces);

% number of vertices of G.
numVertex = size(B,1);

% Initiate the tempMaxIndHubset of which i-th component
% is 1 if i-th vertex, of which information of adjacency is
% denoted by i-th row of B, would be contained in
% maximal independent hubset.
tempMaxIndHubset = zeros(1,numVertex);

% Initiate canBeHub which is vertices that can be an hub.
canBeHub = ones(1,numVertex);

% If we consider about the relation between the sphere
% and the plane all vertices have same conditions for
% being selected as the first hub. However, for the
% convenience to figure the graph in our thesis, we select
% the last vertex.
tempMaxIndHubset(numVertex) = 1;

% And the vertices adjacent to the new hub can be hub
% any more. remove these from canBeHub.
for i=1:numVertex
 if(B(numVertex,i)==1)
  inversedAdjRow(i)=0;
 elseif (B(numVertex,i)==0)
  inversedAdjRow(i)=1;
 end
end
```

```
canBeHub(numVertex) = 0;
canBeHub = canBeHub.*inversedAdjRow;

% We would select new hubs till canBeHub is empty set.
preventInfiniteLoop = 1;

while((sum(canBeHub) ~= 0) & (preventInfiniteLoop < (numVertex
   *3)))
% We would select a new hub which has neighbors that are also
% the neighbors of hubs we already selected, as many as
% possible. countNb(i) is the number of neighbors that are
% also the neighbors of hubs we already selected, of i-th
% vertex. countNb(i) = 0 if the i-th vertex does NOT
% contained in canBeHub.

 countNb = zeros(1,numVertex);

 for i=1:numVertex
  if(canBeHub(i)==1)
   for j=1:numVertex
    if(tempMaxIndHubset(j)==1)
         countNb(i) = countNb(i)+sum(B(i,:).*B(j,:));
        end
   end
  end
 end
% Find  vertices that have maximum number of neighbors
% that are also neighbors of hubs we already selected,
% among the elements of canBeHub.
 tempMaxValue = 0;
 for i=1:numVertex
  if(tempMaxValue < countNb(i))
   tempMaxValue = countNb(i);
  end
 end

% Make set of indices of vertices of which countNb(i) is
% equal to the tempMaxValue.
```

```
 j = 1;
 for i=1:numVertex
  if(countNb(i) == tempMaxValue)
   indexSetMaxValue(j) = i;
   j = j+1;
  end
 end

% Select the vertex as a new hub RANDOMLY
 selectedIndex = indexSetMaxValue(rem(floor(10000*rand(1,1)),j
    -1)+1);
 clear indexSetMaxValue;

% and insert it into maxIndHubset.
 tempMaxIndHubset(selectedIndex) = 1;

% Remove the vertices that are adjacent to the new hub from
% canBeHub.
 for i=1:numVertex
  if (B(selectedIndex,i)==1)
   inversedAdjRow(i)=0;
  elseif (B(selectedIndex,i)==0)
   inversedAdjRow(i)=1;
  end
 end

 canBeHub(selectedIndex)=0;
 canBeHub = canBeHub.*inversedAdjRow;

% prevent infinitely many loop of WHILE statement.
 preventInfiniteLoop = preventInfiniteLoop+1;
end

% Initiate maxIndHubset.
maxIndHubset = zeros(1,size(A,1));

% Convert hubs on B to hubs on A.
for i=1:numVertex
 if(tempMaxIndHubset(i)==1)
```

```
  maxIndHubset(indexB(i))=1;
 end
end
```

hoFindPatches.m :

```
function [patch,patchFree,patchFree4Degree,sizeEqClass,
   facesRemovedHub]=hoFindPatches(B,tempMaxIndHubset,facesB)
% B is adjacency matrix which does not contain 3 degree
% vertices; B could be obtained by 'hoRemove3degree(A)' where
% A is an adjacency matrix A which could be generated by
%'hoGenRanAdjMat' tempMaxIndHubset is a maximal independent
% hubset of B.

% Get size of B
numVertex = size(B,1);

% Initiate nonHubVertices that would not be hubs.
nonHubVertices = ones(1,numVertex);

% Initiate faces. Put face information of B into 'faces'.
facesRemovedHub = facesB;

% Get number of faces
numFace = size(facesRemovedHub,1);

% Remove hubs specified in tempMaxIndHubset from
% nonHubVertices.
for i=1:numVertex
% Generate inversed vector of tempMaxIndHubset.
% In this vector, if i-th vertex contained in the hubset,
% the value of i-th component is 0, otherwise 1.
 if(tempMaxIndHubset(i)==1)
  inversedTempMaxIndHubset(i)=0;
 elseif(tempMaxIndHubset(i)==0)
  inversedTempMaxIndHubset(i)=1;
 end
end
% The componentwise multiplication of a nonHubVertices and
```

```
% the inversed vector of tempMaxIndHubset, removes the hubs
% described by tempMaxIndHubset from nonHubverices.
 nonHubVertices = nonHubVertices.*inversedTempMaxIndHubset;

% Remove faces containing hubs.
% In fact, this removes rows of facesReovedHub that contains
% hubs.
for i=1:numVertex
 if(tempMaxIndHubset(i)==1)
% i-th vertex is a hub.
  j=1;
  while(j<numFace+1)
   tempFaceBit = hoConvertIndexToBit(facesRemovedHub(j,:),
      numVertex);
   if(tempFaceBit(i)==1)
% If a j-th row of facesRemovedHub contains i-th vertex
% remove it from facesRemovedHub that is a list of faces.
    if(j==1)
     facesRemovedHub = facesRemovedHub(2:numFace,:);
    elseif(j==numFace)
     facesRemovedHub = facesRemovedHub(1:numFace-1,:);
    else
     facesRemovedHub = [facesRemovedHub(1:j-1,:);
        facesRemovedHub(j+1:numFace,:)];
    end
    numFace = numFace-1;
   else
    j = j+1;
   end
  end
 end
end

% Initiate some variables.
isFace = 0;
containFace = 0;
numPatch = 0;

% And rename the list of faces.
```

```
facesDB = facesRemovedHub;
sizeFacesDB = size(facesDB,1);

% Since all the faces in facesDB are triangles, each of them
% are contained in some patches.
while(sizeFacesDB > 0)
 numPatch=numPatch+1;
% patch(i,j,k)-th vertex is contained in i-th patch.
% And this vertex is classed by overedge relation so that
% patch(i,j,k)-th vertex is contained in j-th class.

% Process first face
 patch(numPatch,1,1) = facesDB(1,1);
 patch(numPatch,2,1) = facesDB(1,2);
 patch(numPatch,3,1) = facesDB(1,3);

 sizeEqClass(numPatch,1:3)=[1,1,1];

% Remove the face processed from facesDB.
 facesDB = facesDB(2:sizeFacesDB,:);
 sizeFacesDB = sizeFacesDB-1;

 if(sizeFacesDB>0)
  i=1;
  while(i<1+sizeFacesDB)
   eqClass = 0;
   for j=1:sizeEqClass(numPatch,1)
    for k=1:sizeEqClass(numPatch,2)
     if(B(patch(numPatch,1,j),patch(numPatch,2,k))==1)
% Check whether both patch(numPatch,1,j)-th vertex and
% patch(numPatch,2,k)-th vertex are the ends of the same edge.
% One end is from class 1 the other end is from class 2.
% Therefore if the face (triangle) shares the edge with this
% patch, the opposite vertex of the edge in the face is
% contained in class 3.
      tempEdgeContainedBit = hoConvertIndexToBit([patch(
         numPatch,1,j),patch(numPatch,2,k)],numVertex);
      for l=1:sizeFacesDB
```

```
       tempFaceBit = hoConvertIndexToBit(facesDB(l,:),numVertex
          );
       checkShareEdgeBit = tempFaceBit-tempEdgeContainedBit;
       countOne = 0;
       countZero = 0;
       for m=1:numVertex
        if(checkShareEdgeBit(m)==1)
         countOne = countOne+1;
        elseif(checkShareEdgeBit(m)==0)
         countZero = countZero+1;
        end
       end
       if(countOne==1&countZero==numVertex-1)
% We can use only countOne==1 for the IF sentence, but for
% precision, additional condition contZero==numVertex-1 is
% used.
        isFace=1;
% The opposite vertex of the edge in the face is contained in
% class 3.
        eqClass = 3;
        break;
       end
      end
     end
     if(isFace==1)
      break;
     end
    end
    if(isFace==1)
     break;
    end
   end

   if(isFace==0)
        for j=1:sizeEqClass(numPatch,2)
     for k=1:sizeEqClass(numPatch,3)
      if(B(patch(numPatch,2,j),patch(numPatch,3,k))==1)
% Check whether both patch(numPatch,2,j)-th vertex and
% patch(numPatch,3,k)-th vertex are the ends of the same edge.
```

```
% One end is from class 2 the other end is from class 3.
% Therefore if the face (triangle) shares the edge with this
% patch,the opposite vertex of the edge in the face is
% contained in class 1.
       tempEdgeContainedBit = hoConvertIndexToBit([patch(
          numPatch,2,j),patch(numPatch,3,k)],numVertex);
       for l=1:sizeFacesDB
        tempFaceBit = hoConvertIndexToBit(facesDB(l,:),
           numVertex);
        checkShareEdgeBit = tempFaceBit-tempEdgeContainedBit;
        countOne = 0;
        countZero = 0;
        for m=1:numVertex
         if(checkShareEdgeBit(m)==1)
          countOne = countOne+1;
         elseif(checkShareEdgeBit(m)==0)
          countZero = countZero+1;
         end
        end
        if(countOne==1&countZero==numVertex-1)
% We can use only countOne==1 for the IF sentence, but for
% precision, additional condition contZero==numVertex-1 is
% used.
         isFace=1;
% The opposite vertex of the edge in the face is contained in
% class 1.
         eqClass = 1;
         break;
        end
       end
      end
      if(isFace==1)
       break;
      end
     end
     if(isFace==1)
      break;
     end
    end
```

```
   end

   if(isFace==0)
    for j=1:sizeEqClass(numPatch,3)
     for k=1:sizeEqClass(numPatch,1)
      if(B(patch(numPatch,3,j),patch(numPatch,1,k))==1)
% Check whether both patch(numPatch,3,j)-th vertex and
% patch(numPatch,1,k)-th vertex are the ends of the same edge.
% One end is from class 3 the other end is from class 1.
% Therefore if the face (triangle) shares the edge with this
% patch,the opposite vertex of the edge in the face is
% contained in class 2.
       tempEdgeContainedBit = hoConvertIndexToBit([patch(
          numPatch,3,j),patch(numPatch,1,k)],numVertex);
       for l=1:sizeFacesDB
        tempFaceBit = hoConvertIndexToBit(facesDB(l,:),
           numVertex);
        checkShareEdgeBit = tempFaceBit-tempEdgeContainedBit;
        countOne = 0;
        countZero = 0;
        for m=1:numVertex
         if(checkShareEdgeBit(m)==1)
          countOne = countOne+1;
         elseif(checkShareEdgeBit(m)==0)
          countZero = countZero+1;
         end
        end
        if(countOne==1&countZero==numVertex-1)
% We can use only countOne==1 for the IF sentence, but for
% precision, additional condition contZero==numVertex-1 is
% used.
         isFace=1;
% The opposite vertex of the edge in the face is contained in
% class 2.
         eqClass = 2;
         break;
        end
       end
      end
```

```
      if(isFace==1)
       break;
      end
     end
     if(isFace==1)
      break;
     end
    end
   end

   if(isFace==1)
% Remove the processed face form facesDB
    if(l==1)
     facesDB = facesDB(2:sizeFacesDB,:);
    elseif(l==sizeFacesDB)
     facesDB = facesDB(1:sizeFacesDB-1,:);
    else
     facesDB = [facesDB(1:l-1,:);facesDB(l+1:sizeFacesDB,:)];
    end
    sizeFacesDB = sizeFacesDB-1;

% Update the information.
    sizeEqClass(numPatch,eqClass)=sizeEqClass(numPatch,eqClass)
       +1;
    patch(numPatch,eqClass,sizeEqClass(numPatch,eqClass))=
       hoConvertBitToIndex(checkShareEdgeBit);
    isFace=0;
    i=1;
   else
    i=i+1;
   end
  end
 end
end

% Find patch-free vertices among described in nonHubVertices.
index4 = 0;
indexGeneral = 0;
patchFree4Degree = 0;
```

```
patchFree = 0;
for i=1:numVertex
 if(nonHubVertices(i)==1)
  incidentToFace = 0;
  for j=1:numFace
   for k=1:3
    if(facesRemovedHub(j,k)==i)
% If i-th vertex is incident to some faces described in
% facesRemovedHub then the vertex is not patch-free.
     incidentToFace = 1;
     break;
    end
   end
   if(incidentToFace==1)
    break;
   end
  end
  if(incidentToFace==0)
% i-th vertex is patch-free.
% Check whether i-th vertex has the degree 4.
   if(sum(B(i,:))==4)
    index4 = index4+1;
    patchFree4Degree(index4)=i;
   else
    indexGeneral = indexGeneral+1;
    patchFree(indexGeneral)=i;
   end
  end
 end
end
```

hoVirtualAdjMat.m :

```
function [C,eqClassVertexTable] = hoVirtualAdjMat(patch,
   patchFree,B,tempMaxIndHubset)
% This function generate the virtual graph C whose vertices
% symbolize the equivalence classes of patches of B, or
% patch-free vertices of B.
```

```
numVertex = size(B,1);
numPatch = size(patch,1);
numPatchFree = size(patchFree,2);

maxNumVertexInEqClass = size(patch,3);

% eqClassVertexTable(i,j)-th vertex of C symbolize an
% equivalence class of i-th patch of B - "hubset", where the
% members of "hubset" is described in tempMaxIndHubset.
%
% In the first patch, there are three equivalence classes.
% We denote each equivalence class 1,2,3.
eqClassVertexTable(1,1:3)=[1,2,3];
newVertexIndex = 3;
shareVertex = 0;

% Check whether two patches, tempPatchBit_1 and
% tempPatchBit_2, share a vertex.
% If they share a vertex v, [v] of tempPatchBit_1 and [v] of
% tempPatchBit_2 has the same name, where [v] is the
% equivalence class that contains [v].
for indexPatch=2:numPatch
 for i=1:indexPatch-1
  for j=1:3
   for k=1:3
    tempPatchBit_1 = zeros(1,numVertex);
    tempPatchBit_2 = zeros(1,numVertex);
    for n=1:maxNumVertexInEqClass
     if(patch(i,j,n)~=0)
      tempPatchBit_1(patch(i,j,n))=1;
     end
     if(patch(indexPatch,k,n)~=0)
      tempPatchBit_2(patch(indexPatch,k,n))=1;
     end
    end
% If two patches shares a vertex....
    if(sum(tempPatchBit_1.*tempPatchBit_2)~=0)
     shareVertex = 1;
% the k-th equivalence class in (indexPatch)-th patch and
```

```
% j-th equivalence class in i-th patch are symbolized by same
% vertex of C.
     eqClassVertexTable(indexPatch,k) = eqClassVertexTable(i,j)
        ;
     break;
    end
   end
   if(shareVertex == 1)
    break;
   end
  end
 end
 if(shareVertex == 1)
  for l=1:3
% Give names to the equivalence classes that does not contain
% the shared vertex of (indexPatch)-th patch.
   if(eqClassVertexTable(indexPatch,l)==0)
    newVertexIndex=newVertexIndex+1;
    eqClassVertexTable(indexPatch,l)=newVertexIndex;
   end
  end
% Reset shareVertex.
  shareVertex = 0;
 elseif(shareVertex == 0)
% If (indexPatch)-th patch does not share any vertex with
% other m-th patch, where m<indexPatch, give names to these
% three equivalence classes of (indexPatch)-th patch.
  for l=1:3
   newVertexIndex=newVertexIndex+1;
   eqClassVertexTable(indexPatch,l)=newVertexIndex;
  end
 end
end

% Construct Virtual Adjacency Matrix C
% If i-th row (or column) of C symbolize an equivalence class
% in a patch and j-th row (or column) symbolize a path-free
% vertex, then i < j.
%
```

```
% First, we construct lower triangular matrix.
% The three vertices that symbolize the three equivalence
% classes in the same patch are pairwise adjacent in C.
for i=1:numPatch
 if(eqClassVertexTable(i,1)>eqClassVertexTable(i,2))
  C(eqClassVertexTable(i,1),eqClassVertexTable(i,2))=1;
 elseif(eqClassVertexTable(i,1)<eqClassVertexTable(i,2))
  C(eqClassVertexTable(i,2),eqClassVertexTable(i,1))=1;
 end
 if(eqClassVertexTable(i,1)>eqClassVertexTable(i,3))
  C(eqClassVertexTable(i,1),eqClassVertexTable(i,3))=1;
 elseif(eqClassVertexTable(i,1)<eqClassVertexTable(i,3))
  C(eqClassVertexTable(i,3),eqClassVertexTable(i,1))=1;
 end
 if(eqClassVertexTable(i,2)>eqClassVertexTable(i,3))
  C(eqClassVertexTable(i,2),eqClassVertexTable(i,3))=1;
 elseif(eqClassVertexTable(i,2)<eqClassVertexTable(i,3))
  C(eqClassVertexTable(i,3),eqClassVertexTable(i,2))=1;
 end
end

% If a vertex in the equivalence class of a patch and another
% vertex in the equivalence class of another patch are adjacent
% in B, the vertices that symbolize the two equivalence
% classes are adjacent in C.
for i=1:numPatch-1
 for j=1+i:numPatch
  for k=1:3
   for l=1:3
    for m=1:maxNumVertexInEqClass
     for n=1:maxNumVertexInEqClass
      if((patch(i,k,m)~=0)&(patch(j,l,n)~=0))
       if(B(patch(i,k,m),patch(j,l,n))==1)
        if(eqClassVertexTable(i,k)>eqClassVertexTable(j,l))
         C(eqClassVertexTable(i,k),eqClassVertexTable(j,l))=1;
        else
         C(eqClassVertexTable(j,l),eqClassVertexTable(i,k))=1;
        end
       end
```

```
      end
     end
    end
   end
  end
 end
end

% Specify the adjacency of patch-free vertices.
if(patchFree~=0)
 numNonPatchFreeVertex = size(C,1);
 for i=1:numPatchFree
  newVertexIndex=newVertexIndex+1;
% Find the neighbors of the i-th patch-free vertex in B.
  tempNeighborsIndex=hoConvertBitToIndex(B(patchFree(i),:));
  twoPatchFreesAreAdj = 0;
  for j=1:size(tempNeighborsIndex,2)
   includingPatchNumJ=0;
   for s=1:numPatch
    for t=1:3
     for u=1:maxNumVertexInEqClass
      if(patch(s,t,u)==tempNeighborsIndex(j))
% If a patch include some neighbors of i-th patch-free
% vertex....
       includingPatchNumJ=1;
       break;
      end
     end
     if(includingPatchNumJ==1)
      break;
     end
    end
    if(includingPatchNumJ==1)
     break;
    end
   end
   if(includingPatchNumJ==1)
% ...then specify the adjacency information in C.
    C(newVertexIndex,eqClassVertexTable(s,t))=1;
```

```
   elseif(includingPatchNumJ==0&i==1)
% If the first patch-free vertex is never adjacent to any
% other vertices contained in some patch then add zero vector
% in C for the vertex symbolizing the first patch-free vertex
% in C.
    C(newVertexIndex,:)=zeros(1,size(C,2));
   elseif(includingPatchNumJ==0&i~=1)
% If the i-th patch-free vertex is never adjacent to any
% other vertices contained in some patch (i~=1) then Check
% whether this i-th patch-free vertex is adjacent to the
% other u-th patch-free vertices where u<i.
    for u=1:i-1
     if(tempNeighborsIndex(j)==patchFree(u))
% If i-th patch-free vertex and m-th patch-free vertex are
% adjacent, specify the adjacency information in C.
      twoPatchFreesAreAdj=1;
      C(newVertexIndex,numNonPatchFreeVertex+u)=1;
     end
    end
    if(twoPatchFreesAreAdj == 0)
% If not, add zero vector in C for the vertex symbolizing
% this i-th patch-free vertex in C.
     C(newVertexIndex,:)=zeros(1,size(C,2));
    end
   end
  end
 end
end

% The adjacency matrix C should be symmetric.
% We constructed only lower triangular part of C.
% Let us make C symmetric matrix.
tempC = zeros(newVertexIndex,newVertexIndex);
tempC(:,1:size(C,2))=C;
clear C
C = tempC+tempC';
```

hoGenColorList.m :

```
function colorVector=hoGenColorList(C,eqClassVertexTable)
% This function generate a list of 3-colorings for the virtual
% graph C.

% Free the memory.
clear colorVector;

numVertex=size(C,1);
numTriangle = size(eqClassVertexTable,1);

% Since we have at least one patches, each of the first three
% rows (or column) of C represents each equivalence class of
% the first patch. i-th vertex of C will have the color that
% the value of colorVector(i). Since different equivalence
% class have to be assigned different color, the first three
% can be [1,2,3], [2,1,3], [3,2,1], [2,3,1], [3,1,2] or
% [1,3,2]. Without loss of generality, we select [1,2,3].
colorVector = [1,2,3];
for i=2:numTriangle
 numColoredVertex = size(colorVector,2);
 numVertexAppeared = 0;

 for j=1:3
% eqClassVertexTable contain the information of vertices of C
% that are equivalence classes of patches of B (each row of
% eqClassVertexTable is symbolize a patch). Each row of
% eqClassVertexTable is also a face bounded by a triangle in C.

  if(eqClassVertexTable(i,j)<numColoredVertex+1)

% We are deciding the color of eqClassvertextable(i,j)-th
% VERTICES, for j=1,2,3. We are deciding the colors of the
% three vertices at once. However, some of these vertices can
% be specified in eqClassVertexTable MULTIPLE times. If some
% of these vertices were already colored, we should skip
% coloring them. This IF sentence check how many vertices of
% the three vertices (for j=1,2,3) have been colored.
```

```
% numVertexAppeared has the value of the numbers of vertices
% that have already been colored.
   numVertexAppeared = numVertexAppeared+1;
  end
 end

 if(numVertexAppeared==0)
% All the vertices described in the i-th row of
% eqClassVertexTable are never appeared. There is not a
% vertex that was colored at previous step(<i). Locally, we
% have six possible colorings.
  for j=1:size(colorVector,1)
   tempColorVector(6*(j-1)+1,:)=[colorVector(j,:),1,2,3];
   tempColorVector(6*(j-1)+2,:)=[colorVector(j,:),1,3,2];
   tempColorVector(6*(j-1)+3,:)=[colorVector(j,:),2,1,3];
   tempColorVector(6*(j-1)+4,:)=[colorVector(j,:),2,3,1];
   tempColorVector(6*(j-1)+5,:)=[colorVector(j,:),3,1,2];
   tempColorVector(6*(j-1)+6,:)=[colorVector(j,:),3,2,1];
  end
  colorVector=tempColorVector;
  clear tempColorVector;

 elseif(numVertexAppeared==1)
% One of vertices described in the i-th row of
% eqClassVertexTable was appeared. The one of
% eqClassVertexTable(i,j) (j=1,2,3) have been colored at
% previous step(<i). Locally, we have two possible colorings.

  for j=1:3
   if(eqClassVertexTable(i,j)<numColoredVertex+1)
    break;
   end
  end
  for k=1:size(colorVector,1)
   if(colorVector(k,eqClassVertexTable(i,j))==1)
% If the color of the vertex that was already colored is
% color 1, the rest two vertices can have color 2, color 3 or
% color 3, color 2.
    tempColorVector(2*(k-1)+1,:)=[colorVector(k,:),2,3];
```

```
    tempColorVector(2*(k-1)+2,:)=[colorVector(k,:),3,2];
   elseif(colorVector(k,eqClassVertexTable(i,j))==2)
% If the color of the vertex that was already colored is
% color 2, the rest two vertices can have color 1, color 3 or
% color 3, color 1.
    tempColorVector(2*(k-1)+1,:)=[colorVector(k,:),1,3];
    tempColorVector(2*(k-1)+2,:)=[colorVector(k,:),3,1];
   elseif(colorVector(k,eqClassVertexTable(i,j))==3)
% If the color of the vertex that was already colored is
% color 3, the rest two vertices can have color 1, color 3 or
% color 2, color 1.
    tempColorVector(2*(k-1)+1,:)=[colorVector(k,:),1,2];
    tempColorVector(2*(k-1)+2,:)=[colorVector(k,:),2,1];
   end
  end
  colorVector=tempColorVector;
  clear tempColorVector;

 elseif(numVertexAppeared==2)
% Two of vertices described in the i-th row of
% eqClassVertexTable was appeared. The Two of
% eqClassVertexTable(i,j) (j=1,2,3) have been colored at
% previous step(<i). Locally, we have one possible coloring.
  for j=1:3
   if(eqClassVertexTable(i,j)>numColoredVertex)
    break;
   end
  end
  for k=1:size(colorVector,1)
   if(j==3)
% The non-colored vertex is eqClassVertexTable(i,3)-th vertex.
    if((colorVector(k,eqClassVertexTable(i,1))==1&colorVector(k
       ,eqClassVertexTable(i,2))==2)|(colorVector(k,
       eqClassVertexTable(i,1))==2&colorVector(k,
       eqClassVertexTable(i,2))==1))
% The colors of the two colored vertices are color 1, color 2
% or color 2, color 1. Locally, we can assign only color 3 to
% the rest vertex.
     tempColorVector(k,:)=[colorVector(k,:),3];
```

```
    elseif((colorVector(k,eqClassVertexTable(i,1))==1&
       colorVector(k,eqClassVertexTable(i,2))==3)|(colorVector(
       k,eqClassVertexTable(i,1))==3&colorVector(k,
       eqClassVertexTable(i,2))==1))
% The colors of the two colored vertices are color 1, color 3
% or color 3, color 1. Locally, we can assign only color 2 to
% the rest vertex.
     tempColorVector(k,:)=[colorVector(k,:),2];
    elseif((colorVector(k,eqClassVertexTable(i,1))==2&
       colorVector(k,eqClassVertexTable(i,2))==3)|(colorVector(
       k,eqClassVertexTable(i,1))==3&colorVector(k,
       eqClassVertexTable(i,2))==2))
% The colors of the two colored vertices are color 2, color 3
% or color 3, color 2. Locally, we can assign only color 1 to
% the rest vertex.
     tempColorVector(k,:)=[colorVector(k,:),1];
    elseif(colorVector(k,eqClassVertexTable(i,1))==colorVector(
       k,eqClassVertexTable(i,2)))
% The two colored vertice could have the same color and this
% is wrong. Temporarily, we can assign only color 4 to the
% rest vertex.
     tempColorVector(k,:)=[colorVector(k,:),4];
    end
   elseif(j==2)
% The non-colored vertex is eqClassVertexTable(i,2)-th vertex.
    if((colorVector(k,eqClassVertexTable(i,1))==1&colorVector(k
       ,eqClassVertexTable(i,3))==2)|(colorVector(k,
       eqClassVertexTable(i,1))==2&colorVector(k,
       eqClassVertexTable(i,3))==1))
% The colors of the two colored vertices are color 1, color 2
% or color 2, color 1. Locally, we can assign only color 3 to
% the rest vertex.
     tempColorVector(k,:)=[colorVector(k,:),3];
    elseif((colorVector(k,eqClassVertexTable(i,1))==1&
       colorVector(k,eqClassVertexTable(i,3))==3)|(colorVector(
       k,eqClassVertexTable(i,1))==3&colorVector(k,
       eqClassVertexTable(i,3))==1))
% The colors of the two colored vertices are color 1, color 3
% or color 3, color 1. Locally, we can assign only color 2 to
```

```
% the rest vertex.
     tempColorVector(k,:)=[colorVector(k,:),2];
    elseif((colorVector(k,eqClassVertexTable(i,1))==2&
       colorVector(k,eqClassVertexTable(i,3))==3)|(colorVector(
       k,eqClassVertexTable(i,1))==3&colorVector(k,
       eqClassVertexTable(i,3))==2))
% The colors of the two colored vertices are color 2, color 3
% or color 3, color 2. Locally, we can assign only color 1 to
% the rest vertex.
     tempColorVector(k,:)=[colorVector(k,:),1];
    elseif(colorVector(k,eqClassVertexTable(i,1))==colorVector(
       k,eqClassVertexTable(i,3)))
% The two colored vertice could have the same color and this
% is wrong. Temporarily, we can assign only color 4 to the
% rest vertex.
     tempColorVector(k,:)=[colorVector(k,:),4];
    end
   elseif(j==1)
% The non-colored vertex is eqClassVertexTable(i,1)-th vertex.
    if((colorVector(k,eqClassVertexTable(i,2))==1&colorVector(k
       ,eqClassVertexTable(i,3))==2)|(colorVector(k,
       eqClassVertexTable(i,2))==2&colorVector(k,
       eqClassVertexTable(i,3))==1))
% The colors of the two colored vertices are color 1, color 2
% or color 2, color 1. Locally, we can assign only color 3 to
% the rest vertex.
     tempColorVector(k,:)=[colorVector(k,:),3];
    elseif((colorVector(k,eqClassVertexTable(i,2))==1&
       colorVector(k,eqClassVertexTable(i,3))==3)|(colorVector(
       k,eqClassVertexTable(i,2))==3&colorVector(k,
       eqClassVertexTable(i,3))==1))
% The colors of the two colored vertices are color 1, color 3
% or color 3, color 1. Locally, we can assign only color 2 to
% the rest vertex.
     tempColorVector(k,:)=[colorVector(k,:),2];
    elseif((colorVector(k,eqClassVertexTable(i,2))==2&
       colorVector(k,eqClassVertexTable(i,3))==3)|(colorVector(
       k,eqClassVertexTable(i,2))==3&colorVector(k,
       eqClassVertexTable(i,3))==2))
```

```
% The colors of the two colored vertices are color 2, color 3
% or color 3, color 2. Locally, we can assign only color 1 to
% the rest vertex.
     tempColorVector(k,:)=[colorVector(k,:),1];
    elseif(colorVector(k,eqClassVertexTable(i,2))==colorVector(
       k,eqClassVertexTable(i,3)))
% The two colored vertices could have the same color and this
% is wrong. Temporarily, we can assign only color 4 to the
% rest vertex.
     tempColorVector(k,:)=[colorVector(k,:),4];
    end
   end
  end
  colorVector=tempColorVector;
  clear tempColorVector;

% Remove the rows of colorVector that contain color 4.
  l=size(colorVector,2);
  tempIndex = 1;
  for k=1:size(colorVector,1)
   if(colorVector(k,l)~=4)
    tempColorVector(tempIndex,:) = colorVector(k,:);
    tempIndex=tempIndex+1;
   end
  end
  colorVector=tempColorVector;
  clear tempColorVector;

 end
end

% The coloring of vertices of C that symbolize the
% equivalence class of patches was Done. Now, we decide the
% coloring of patch-free vertices, one by one.
%
% numVerticesInPatch is the number of vertices that have been
% already colored.
numVerticesInPatch = size(colorVector,2);
```

```
% The number of patch-free vertices is
% (the numVertex - numVerticesInPatch)
for i = numVerticesInPatch+1:numVertex
 newLengthColorVector = size(colorVector,2)+1;
 tempIndex = 1;
 for j=1:size(colorVector,1)
% If the (i-th) patch-free vertex has 'color m' we set
% adjacentColorSet(m) = 1, for m=1,2,3.
  adjacentColorSet = [0,0,0];
  for k=1:i-1
   if(C(i,k)==1)
    adjacentColorSet(colorVector(j,k))=1;
   end
  end
  if(sum(adjacentColorSet)==3)
% The (i-th) patch-free vertex has three different colors on
% its neighbors. This is wrong. We temporarily set the color
% of this vertex as color 4.
   tempColorVector(tempIndex,:) = [colorVector(j,:),4];
   tempIndex = tempIndex+1;
  elseif(sum(adjacentColorSet)==2)
% The (i-th) patch-free vertex has two different colors on
% its neighbors.
   for l=1:3
% The (i-th) patch-free vertex does NOT have 'color l' on its
% neighbor.
    if(adjacentColorSet(l)==0)
     break;
    end
   end
% We assign this (i-th) patch-free vertex the last color.
   tempColorVector(tempIndex,:)=[colorVector(j,:),l];
   tempIndex=tempIndex+1;
  elseif(sum(adjacentColorSet)==1)
% The (i-th) patch-free vertex has one color on its neighbors.
   for l=1:3
% The (i-th) patch-free vertex have 'color l' on its neighbor.
    if(adjacentColorSet(l)==1)
     break;
```

```
    end
   end
% We assign this (i-th) patch-free vertex the two possible
% colors.
   if(l==1)
% color l is color 1.
    tempColorVector(tempIndex,:)=[colorVector(j,:),2];
    tempColorVector(tempIndex+1,:)=[colorVector(j,:),3];
    tempIndex=tempIndex+2;
   elseif(l==2)
% color l is color 2.
    tempColorVector(tempIndex,:)=[colorVector(j,:),1];
    tempColorVector(tempIndex+1,:)=[colorVector(j,:),3];
    tempIndex=tempIndex+2;
   elseif(l==3)
% color l is color 3.
    tempColorVector(tempIndex,:)=[colorVector(j,:),1];
    tempColorVector(tempIndex+1,:)=[colorVector(j,:),2];
    tempIndex=tempIndex+2;
   end
  elseif(sum(adjacentColorSet)==0)
% The (i-th) patch-free vertex not colored neighbor.
% We can assign any colors (among the three colors) on it.
   tempColorVector(tempIndex,:)=[colorVector(j,:),1];
   tempColorVector(tempIndex+1,:)=[colorVector(j,:),2];
   tempColorVector(tempIndex+2,:)=[colorVector(j,:),3];
   tempIndex=tempIndex+3;
  end
 end
 colorVector = tempColorVector;
 clear tempColorVector;

% Remove the rows of colorVector that contain color 4.
 l=size(colorVector,2);
 tempIndex = 1;
 for k=1:size(colorVector,1)
  if(colorVector(k,l)~=4)
   tempColorVector(tempIndex,:)=colorVector(k,:);
   tempIndex=tempIndex+1;
```

```
  end
 end
 colorVector=tempColorVector;
 clear tempColorVector;
end
```

hoCheck3Colorable.m :

```
function success=hoCheck3Colorable(C,colorVector)
% This checks whether the 3-coloring solution of C is contained
% in colorVector. If it is return 1, otherwise return 0.
numColorList = size(colorVector,1);
numVertex = size(C,1);

for i=1:numColorList
% Assume the row is a 3-coloring solution.
 success=1;
 for j=1:numVertex
  for k=1:j-1
% Check every pair of adjacent vertices.
   if(C(j,k)==1&colorVector(i,j)==colorVector(i,k))
% If the two adjacent vertices have the same color, return 0.
    success = 0;
    break;
   end
  end
  if(success == 0)
% escape the FOR sentence.
   break;
  end
 end
 if(success==1)
% If there exists a 3-coloring solution, print it.
  colorVector(i,:)
  break;
 end
end
```

hoConvertBitToIndex.m :

```
function result=hoConvertBitToIndex(v)
% This convert Bit-representation of set of vertices to
% Index-representation. For example, let V = {1,3,5} be a set
% of vertices. V can be represent as [1,3,5] or [1,0,1,0,1].
% This function convert [1,0,1,0,1] to [1,3,5].

index = 1;
for i=1:size(v,2)
 if(v(i)==1)
  result(index)=i;
  index=index+1;
 end
end
```

hoConvertIndexToBit.m :

```
function result=hoConvertIndexToBit(v,numVertex)
% This convert Index-representation of set of vertices to
% Bit-representation. For example, let V = {1,3,5} be a set of
% vertices. V can be represent as [1,3,5] or [1,0,1,0,1]. This
% function convert [1,3,5] to [1,0,1,0,1].

result = zeros(1,numVertex);
for i=1:size(v,2)
 result(v(i))=1;
end
```

hoTranslateAdjMat.m :

```
function trans=hoTranslateAdjMat(A)
% This is a utility to help drawing the graph of A. When we
% draw i-th vertex of graph of A, the i-th row of 'trans' tell
% us which vertices should be adjacent to the i-th vertex. For
% example, if the sixth row of 'trans' is [1,4,0,0,0], we
% should draw line between first vertex and sixth vertex, and
% between fourth vertex and sixth vertex when we draw the
% sixth vertex.
```

```
for i=2:size(A,1)
 tempIndex=1;
 for j=1:i-1
  if(A(i,j)==1)
   trans(i-1,tempIndex)=j
   tempIndex=tempIndex+1;
  end
 end
end

trans
```

If you have all the previously specified codes, then you can get the result just by executing the following command.

```
--> hoRun
```

4.3 Manual of the programs

This section contains an example describing how to use the codes.

The `[A,faces] = hoGenRandAdjMat(20)` command generates a random adjacency matrix `A` of a maximal plane graph of order `20` and `faces` which is a list of its faces.

```
--> [A,faces] = hoGenRandAdjMat(20)
A =
 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 1 0 1 1 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
 1 1 0 0 1 0 0 1 0 1 0 0 0 0 1 0 0 1 0 0
   .
   .
   .
(omitted)

faces =
  1  2  3
  1  2  4
  3  1  5
   .
   .
   .
(omitted)
```

The `[maxIndHubset,tempMaxIndHubset,B,indexB,facesB] = hoSelectArbMaxIndHubset(A,faces)` command generates an adjacency matrix `B` obtained from `A` by removing the information of the vertices of degree 3. It also generates a vector `tempMaxIndHubset` that contains the information of a maximal hubset of `B`.

```
--> [maxIndHubset,tempMaxIndHubset,B,indexB,facesB]=
   hoSelectArbMaxIndHubset(A,faces)
maxIndHubset =
 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0
tempMaxIndHubset =
 0 0 0 0 1 0 1 0 0 0 0 1
```

```
B =
 0  1  1  1  1  1  0  0  0  0  0  0
 1  0  1  1  0  0  1  0  0  0  0  0
 1  1  0  0  1  0  1  0  1  0  1  1
 1  1  0  0  0  1  1  0  1  0  0  0
 1  0  1  0  0  1  0  1  0  1  1  0
 1  0  0  1  1  0  0  1  1  0  0  0
 0  1  1  1  0  0  0  0  1  0  0  0
 0  0  0  0  1  1  0  0  1  1  0  0
 0  0  1  1  0  1  1  1  0  1  0  1
 0  0  0  0  1  0  0  1  1  0  1  1
 0  0  1  0  1  0  0  0  0  1  0  1
 0  0  1  0  0  0  0  0  1  1  1  0
indexB =
  1   2   3   4   5   6   8   9  10  12  15  18
facesB =
  1   2   3
  1   2   4
  3   1   5
    .
    .
 (omitted)
```

The `[patch,patchFree,patchFree4Degree,sizeEqClass, facesRemovedHub]=hoFindPatches(B,tempMaxIndHubset,facesB)` command finds patches (`patch`), patch-free vertices (`patchFree`), and some other information. The `patch` is a 3-dimensional array of numbers. The member (m, n, l) of `patch` denotes the l-th vertex of the n-th equivalence class of the overedge relation in the m-th patch. In the following example, there is one patch that is refined into three equivalence classes. The vertices 1 and 9 are contained in the first equivalence class of the first patch. '0' means empty.

```
--> [patch,patchFree,patchFree4Degree,sizeEqClass,
   facesRemovedHub]=hoFindPatches(B,tempMaxIndHubset,facesB)
patch =
(:,:,1) =
  1   2   3
(:,:,2) =
```

```
  9   6   4
(:,:,3) =
  0  10   8
patchFree =
 0
patchFree4Degree =
 11
sizeEqClass =
 2 3 3
facesRemovedHub =
  1   2   3
  1   2   4
  1   4   6
  8   6   9
  6   4   9
  8   9  10
```

The `[C,eqClassVertexTable] = hoVirtualAdjMat(patch,patchFree,B ,tempMaxIndHubset)` command creates a virtual adjacency matrix `C` of an abstract graph on the sum of equivalence classes of patches and patch-free vertices of degree more than 4 in `B`. In the example, `C` describes a triangle since we had one patch without a patch-free vertex of degree more than 4.

```
--> [C,eqClassVertexTable] = hoVirtualAdjMat(patch,patchFree,B,
   tempMaxIndHubset)
C =
 0 1 1
 1 0 1
 1 1 0
eqClassVertexTable =
 1 2 3
```

The `colorVector=hoGencolorList(C,eqClassVertexTable)` command generates `colorVector` which is a list of probable 3-colorings of `C`.

```
--> colorVector=hoGencolorList(C,eqClassVertexTable)
colorVector =
 1 2 3
```

The `success=hoCheck3Colorable(C,colorVector)` command checks whether the list contains a 3-coloring solution of `C`. If so, it returns `success=1` and if not, it returns `success=0`. This command also shows the 3-coloring solution. The value of the n-th component of the vector `ans` is the color assigned to the n-th vertex of `C`. We can find the vertex coloring of `B` with the information contained in `patch`. In this example, vertices 1 and 9 would be colored with color 1; vertices 2, 6, and 10 would be colored with color 2; and vertices 3, 4, and 8 would be colored with color 3.

```
--> success=hoCheck3Colorable(C,colorVector)
ans =
 1 2 3
success =
 1
```

The checking process is finished. However, the `hoTranslateAdjMat(B)` helps us to figure the graph of `B`. See the following.

```
--> hoTranslateAdjMat(B)
   .
   .
   .
 (omitted)
   .
   .
   .
ans =
  1   0   0   0   0
  1   2   0   0   0
  1   2   0   0   0
  1   3   0   0   0
  1   4   5   0   0
  2   3   4   0   0
  5   6   0   0   0
  3   4   6   7   8
  5   8   9   0   0
  3   5  10   0   0
  3   9  10  11   0
```

From the matrix named `ans`, we can figure the graph on the paper. The n-th row of `ans` shows what vertices would be the neighbors of the $(n+1)$-th vertex among n vertices (that are already drawn) when we draw the $(n+1)$-th vertex. For example, when we draw the sixth vertex on the paper, we draw lines between the sixth vertex and the first vertex, between the sixth vertex and the fourth vertex, and between the sixth vertex and the fifth vertex, since the fifth row of `ans` is `[1 4 5 0 0]`. Note that whenever we finish adding a vertex on the paper, There should be at most one face whose boundary is not a triangle. Obviously, if there exists the unique face whose boundary is not a triangle, then it is the unique unbounded face.

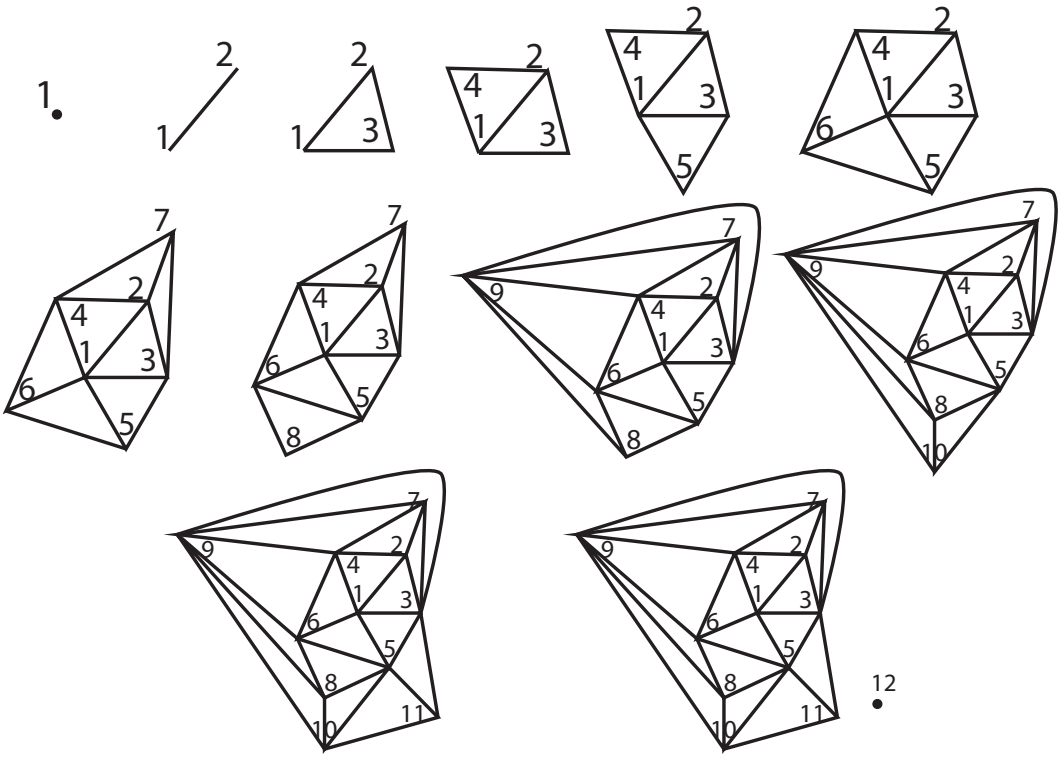

Figure 4.3.1: Drawing graph B.

Also, we can figure the hubs and their wheels with the help of the command `hoConvertBitToIndex(tempMaxIndHubset)`. In the example, hubs are the fifth, seventh, and twelfth drawn vertices.

```
--> hoConvertBitToIndex(tempMaxIndHubset)
ans =
  5  7  12
```

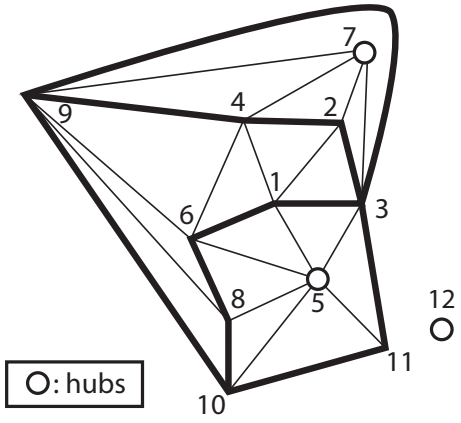

Figure 4.3.2: Drawing hubs and wheels on graph B.

It shows that there is one patch, and the members of each equivalence class of the patch are as the 3-dimensional array `patch` describes. We can easily find a 4-coloring solution of this graph.

Bibliography

[1] Béla Bollobás, *Mordern Graph Theory*, Springer, 1998.

[2] Reinhard Diestel, *Graph Theory*, Springer, 2006.

[3] Georges Gonthier, Formal Proof–The Four-Color Theorem, *Notices of the American Mathematical Society* **55**(11) (2008), 1382–1393.

[4] Frank Harary, *Graph Theory*, Perseus Books, 1994.

[5] George A. Jennings, *Modern Geometry with Applications*, Springer, 1994.

[6] James R. Munkres, *Topology*, Prentice Hall, 2000.

[7] Neil Robertson, Daniel Sanders, Paul Seymour, and Robin Thomas, The Four-color Theorem, *Journal of Combinatorial Theory, Series B* **70** (1997), 2–44.

[8] John Stillwell, *Classical Topology and Combinatorial Group Theory*, Springer, 1980.

[9] Carsten Thomassen, A short list color proof of Götzsch's theorem, *Journal of Combinatorial Theory, Series B* **88** (2003), 189–192.

국문초록

사색 문제로 알려진 *사색 이론*은 지도 상의 서로 인접한 두 영역들이 같은 색상을 가지지 않도록 채색할 때, 네 가지 색상만으로도 조건을 충분히 만족시키며 모든 영역들을 채색할 수 있다는 이론이다. 사색 이론은 평면 그래프의 꼭짓점을 채색하는 문제로 해석될 수 있다. 평면 그래프의 꼭짓점을 채색하기 위해 우리는 다음과 같은 방법에 대해 알아보도록 한다.

1. 주어진 지도를 그래프로 표현하고, 이 그래프를 포함하며 꼭짓점의 수는 같고 가능한 많은 모서리를 가지는 평면 그래프를 하나 찾는다.
2. 이 평면 그래프에서 차수가 3 이하인 꼭짓점이 있다면 제거한다.
3. 이 그래프에서 서로 인접하지 않으며 각각 어떤 차륜형 부분 그래프들의 중심이 되는 꼭짓점들로 이루어진 *중심점 집합*을 찾는다.
4. 중심점 집합에 원소가 아닌 꼭짓점들을 세 가지 색상로만 채색한다.
5. 중심점 집합의 원소가 되는 꼭짓점들을 앞의 세가지 색상과는 다른 네 번째 색상으로 채색한다.
6. 이 결과를 처음에 주어진 지도에 적용한다.

위와 같은 방법을 이용하여, 무작위로 생성된 꼭짓점의 개수가 40인 그래프를 채색하는 컴퓨터 실험에서 약 98%의 채색 성공률을 얻을 수 있었다. 이 채색 방법을 개선하는 것에 대해 논하도록 한다.

주요어휘: 꼭짓점 색칠하기, 평면 그래프, 4색, 사색, 사색문제, 사색이론
학번: 2009-20284

감사의 글

부족하고 미욱한 학생을 지도하고 이끌어주신 김홍종 선생님께 깊은 감사를 드립니다. 항상 격려해주시는 선생님의 그늘이 있었기에 이렇게 학위 논문을 무사히 마칠 수 있었습니다.

바쁜 일정에도 시간을 내어 논문 심사를 맡아 주신 국웅 선생님과 김서령 선생님께 감사를 드립니다.

또한 고성은 선생님, 신동관 선생님, 권오인 선생님, 김태희 선생님, 박춘재 선생님, 이상진 선생님, 이승훈 선생님, 정은옥 선생님, 팽성훈 선생님, 최인송 선생님, 남현수 선생님, 이선미 선생님께 뒤늦은 감사의 뜻을 전하고자 합니다. 특히 이상진 선생님께서 가르쳐 주신 위상 수학의 기초와 정은옥 선생님께 배운 컴퓨터의 활용은 이 논문을 쓰는데 많은 도움이 되었습니다.

논문의 내용을 함께 공부하여 주셨으며 글을 쓰는데 있어서 구체적인 조언을 많이 주신 박경동 선배님께 감사의 말씀을 드립니다. 더불어 어렵게 여겼던 내용들을 친절히 설명하여 주셨던 변태창 선생님께도 감사의 말씀을 전하고자 합니다.

함께 공부하였고 우매한 물음에도 기꺼이 응해주었던 김동훈, 신필수, 장윤수, 채우리, 최우철 외 대학원 동기들에게 감사를 드립니다.

수리과학부를 뒷받침해 주시는 이국현 선생님과 오해자 선생님을 비롯한 행정실 직원분들께 감사의 뜻을 전하고자 합니다.

또한 함께 공부하며 많은 이야기를 들려준 이완호, 김신영, 신동원, 박종민, 최익준, 방규호, 김이랑, 김상우, 김성원, 김승현, 김지나, 노영한, 박연경, 유춘식, 허윤호, 최선화, 김태연, 윤용섭, 김효진, 김성은, 임예옥, 장효정, 김용운, 서우덕, 이주희, 정미선, 정진, 홍석영, 권유선, 김영은, 신지수, 강대용, 박성호, 김민혜, 이채영, 권시은, 남혜현, 황미애, 강진구, 박정배, 김혜진, 문경식, 손재원, 오필선, 강정석, 곽성근, 김현수, 류대호, 정현민, 김녹형 외 선후배님들과 동문들에게 감사의 말씀을 전합니다.

영어 교정에 많은 도움을 주신 김수잔 선생님께 감사를 드립니다.

지금도 여러 고민들을 함께 나누어 주는 김재현, 김범진, 이상봉, 이우진, 이주안, 최영철, 최윤석, 이성현, 서정우, 윤용한 외 친구들에게 감사의 뜻을 전합니다.

언제나 아낌없이 주시고 보살펴 주시는 부모님께, 그리고 항상 응원을 해준 두 동생들에게 감사의 마음을 전합니다.