Path-Coloring Algorithms for Plane Graphs

Aven Bross
Department of Computer Science
University of Alaska
Fairbanks, AK 99775-6670
dabross@alaska.edu

Glenn G. Chappell
Department of Computer Science
University of Alaska
Fairbanks, AK 99775-6670
chappellg@member.ams.org

Chris Hartman
Department of Computer Science
University of Alaska
Fairbanks, AK 99775-6670
cmhartman@alaska.edu

December 5, 2024

2010 Mathematics Subject Classification. Primary 05C38; Secondary 05C10, 05C15.
Key words and phrases. Path coloring, list coloring, algorithm.

Abstract

A path coloring of a graph G is a vertex coloring of G such that each color class induces a disjoint union of paths. We present two efficient algorithms to construct a path coloring of a plane graph.

The first algorithm, based on a proof of Poh, is given a plane graph; it produces a path coloring of the given graph using three colors.

The second algorithm, based on similar proofs by Hartman and Škrekovski, performs a list-coloring generalization of the above. The algorithm is given a plane graph and an assignment of lists of three colors to each vertex; it produces a path coloring of the given graph in which each vertex receives a color from its list.

Implementations are available for all algorithms that are described.

1 Introduction

All graphs will be finite, simple, and undirected. See West [10] for graph theoretic terms.

A *path coloring* of a graph G is a vertex coloring (not necessarily proper) of G such that each color class induces a disjoint union of paths. A graph G is *path k-colorable* if G admits a path coloring using k colors.

Broere & Mynhardt conjectured [2, Conj. 16] that every planar graph is path 3-colorable. This was proven independently by Poh [8, Thm. 2] and by Goddard [6, Thm. 1].

Theorem 1.1 (Poh 1990, Goddard 1991)**.** *If G is a planar graph, then G is path* 3*-colorable.* □

It is easily shown that the "3" in Theorem 1.1 is best possible. In particular, Chartrand & Kronk [5, Section 3] gave an example of a planar graph whose vertex set cannot be partitioned into two subsets, each inducing a forest.

Hartman [7, Thm. 4.1] proved a list-coloring generalization of Theorem 1.1 (see also Chappell & Hartman [4, Thm. 2.1]). A graph G is *path k-choosable* if, whenever each vertex of G is assigned a list of k colors, there exists a path coloring of G in which each vertex receives a color from its list.

Theorem 1.2 (Hartman 1997)**.** *If G is a planar graph, then G is path* 3*-choosable.* □

Essentially the same technique was used by Škrekovski [9, Thm. 2.2b] to prove a result slightly weaker than Theorem 1.2.

We discuss two efficient path-coloring algorithms based on proofs of the above theorems. We distinguish between a *planar* graph—one that can be drawn in the plane without crossing edges—and a *plane* graph—a graph with a given embedding in the plane.

In Section 2 we outline our graph representations and the basis for our computations of time complexity.

Section 3 covers an algorithm based on Poh's proof of Theorem 1.1. The algorithm is given a plane graph; it produces a path coloring of the given graph using three colors.

Section 4 covers an algorithm based Hartman's proof of Theorem 1.2, along with the proof of Škrekovski mentioned above. The algorithm is given a plane graph and an assignment of a list of three colors to each vertex; it produces a path coloring of the given graph in which each vertex receives a color from its list.

Section 5 provides benchmark results for implementations of each algorithm [3]. The section also discusses how each algorithm may be modified to benefit from parallelism.

2 Graph Representations and Time Complexity

We will represent a graph via *adjacency lists*: a list, for each vertex v, of the neighbors of v. A vertex can be represented by an integer $0 \ldots n-1$, where $n = n(G)$ is the order of the graph.

A plane graph will be specified via a *rotation scheme*: a circular ordering, for each vertex v, of the edges incident with v, in the order they appear around v in the plane embedding; this completely specifies the combinatorial embedding of the graph. Rotation schemes are convenient when we represent a graph using adjacency lists; we simply order the adjacency list for each vertex v in counter-clockwise order around v; no additional data structures are required.

A plane graph is *triangulated* if every face is a 3-cycle, and *weakly triangulated* if every face other than the outer face is a 3-cycle. A graph G is *connected* if given any $u, v \in G$, there exists a u, v-path in G. We say that G is *n-connected* if removing any $n-1$ vertices results in a connected graph. The outer face of a plane graph that is both 2-connected and weakly triangulated is a path if $n = 1$ or $n = 2$, and a cycle if $n \geq 3$.

The input for each algorithm will be a 2-connected, weakly triangulated plane graph with n vertices and m edges, represented via adjacency lists. The input size will be n, the number of vertices. Note that $\mathcal{O}(m) = \mathcal{O}(n)$, so it is equivalent to take the input size to be m, the number of edges. Moreover, arbitrary simple planar graphs may be plane embedded and triangulated in $\mathcal{O}(n)$ time, see Boyer and Myrvold [1].

In Section 4, given an edge uv, we will need a constant time operation to find v's entry in u's adjacency list from u's entry in v's list. We define an *augmented adjacency list* to be an adjacency list such that for every edge uv a reference to v's entry in u's list is stored in u's entry in v's list, and vice versa. Given an adjacency list representation of a graph, an augmented adjacency list representation may be constructed in $\mathcal{O}(m)$ time via the following procedure.

Algorithm 2.1.
Input. An adjacency list representation Adj of a graph G.

Output. An augmented adjacency list representation AugAdj of G with the same rotation scheme as Adj.

Step 1. Construct an augmented adjacency list representation AugAdj with the same rotation scheme as Adj, leaving the reference portion of each entry uninitialized.

Step 2. For each vertex v construct an array Wrk[v] of vertex-reference pairs with length $\deg(v)$. For each v from 0 to $n-1$ iterate through AugAdj[v]. For each neighbor u in AugAdj[v] append the pair $(v, r_v(u))$ to Wrk[u], where $r_v(u)$ is a reference to u's entry in AugAdj[v].

Step 3. For each v from $n-1$ to 0 iterate through Wrk[v]. Upon reaching a pair $(u, r_u(v))$ in Wrk[v] the last element of Wrk[u] will be $(v, r_v(u))$; for details on why this must be the case, see the paragraph below. Use $r_u(v)$ and $r_v(u)$ to look up and assign references for the edge uv in AugAdj[u] and AugAdj[v]. Remove $(v, r_v(u))$ from the back of Wrk[u].

After completing Step 2 in Algorithm 2.1 the array Wrk[v] will contain a pair $(u, r_u(v))$ for each neighbor u of v, sorted in increasing order by the neighbor u. Suppose that v is the current vertex at a given iteration of Step 3 in Algorithm 2.1. For each edge $uw \in E(G)$ such that $u < w$ and $v < w$, prior iterations of Step 3 will have initialized the references for uw in AugAdj[u] and AugAdj[w], and also removed the pair $(w, r_w(u))$ from Wrk[u]. Therefore for each $(u, r_u(v))$ in Wrk[v], the array Wrk[u] will contain only entries for vertices w where $w \leq v$. Since Wrk[u] is sorted in increasing order by the neighboring vertices, the last element of Wrk[u] must be $(v, r_v(u))$.

3 Path Coloring

In this section we describe a linear time algorithm to path 3-color plane graphs. Let's first recount Poh's path 3-coloring proof strategy. Given a cycle C in a plane graph G we define $\text{Int}(C)$ to be the subgraph of G consisting of C and all vertices and edges interior to C. If C is a length 1 or 2 path, we define $\text{Int}(C) = C$. Equivalently, $\text{Int}(C)$ is the maximal 2-connected subgraph of G with outer face C.

Lemma 3.1 (Poh 1990)**.** *Let G be a 2-connected, weakly triangulated plane graph with outer face C. Let $c : V(C) \to S \subsetneq \{1, 2, 3\}$ be a 2-coloring of C such that each color class induces a nonempty path. There exists an extension of c to a path 3-coloring $c : V(G) \to \{1, 2, 3\}$ such that for each $v \in G - C$, if $vu \in E(G)$ with $u \in C$, then $c(v) \neq c(u)$.*

Proof. If $n(G) \leq 3$, then $G = C$ and the path 2-coloring of C is a path 3-coloring of G. We proceed by induction on the order of G.

Let P_1, P_2 be the two paths induced by the 2-coloring of the outer face C. Label the vertices of the outer face $C = v_1, v_2, \ldots, v_k$ in clockwise order such that $P_1 = v_1, v_2, \ldots, v_\ell$ and $P_2 = v_{l+1}, v_{l+2}, \ldots, v_k$.

Suppose C is an induced subgraph of G. Let $u, w \in V(G) - V(C)$ be the vertices such that u, v_1, v_k and $w, v_\ell, v_{\ell+1}$ are faces of G; note that u and w are uniquely determined, but it may be that $u = w$. Since C is an induced cycle in G and $G \neq C$, $G - C$ is connected. Let $P_3 = u_1, u_2, \ldots, u_r$ be a u, w-path in $G - C$ of minimal length, and note that P_3 is an induced subgraph of $G - C$.

Color each vertex of P_3 with the color in $\{1, 2, 3\} - S$ not used in the 2-coloring of C. Let $C_1 = v_1, v_2, \ldots, v_\ell, u_r, u_{r-1}, \ldots, u_1$ and $C_2 = u_1, u_2, \ldots, u_r, v_{\ell+1}, v_{l+2}, \ldots, v_k$. The subgraphs $\text{Int}(C_1)$ and $\text{Int}(C_2)$ together with the coloring c each satisfy the requirements of the lemma. By the inductive hypothesis there exist extensions of c to a path 3-coloring of $\text{Int}(C_1)$ and a path 3-coloring of $\text{Int}(C_2)$. Since $\text{Int}(C_1)$ and $\text{Int}(C_2)$ only share the vertices in P_3 on their respective outer faces, the colorings agree and form a path 3-coloring of G.

Suppose C is not an induced subgraph. Then there exists an edge $v_i v_j \in E(G) - E(C)$ such that $i \leq \ell < j$. Let $C_1 = v_1, v_2, \ldots, v_i, v_j, v_{j+1}, \ldots, v_k$ and $C_2 = v_i, v_{i+1}, \ldots, v_j$. By the inductive hypothesis, there exists an extension of c to a path 3-coloring of $\text{Int}(C_1)$ and $\text{Int}(C_2)$. Again the subgraphs only share vertices on their outer faces, and thus the colorings agree and combine form a path 3-coloring of G. □

Let G be a triangulated plane graph. We may trivially path 2-color the outer triangle. Applying Lemma 3.1 extends this coloring to a path 3-coloring of G.

For an arbitrary planar graph G we may compute an embedding and add edges to produce a triangulated plane graph G'. Any path 3-coloring of G' will also be a path 3-coloring of G. Thus Theorem 1.1 follows from Lemma 3.1.

In the construction of the path P_3 in Poh's proof, the induced u, w-path was picked to be a u, w-path of shortest length. Thus a natural way to implement Poh's algorithm is to locate u, and then use a breadth-first search to construct a u, w-path and/or locate a chord edge.

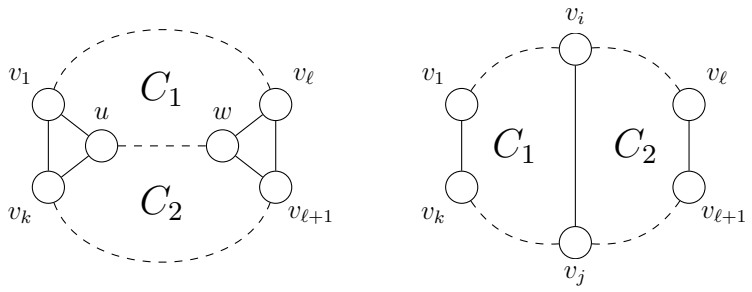

Figure 1: The proof of Lemma 3.1 when C is induced (left) and not induced (right).

Algorithm 3.2.
Input. Let $C = v_1, v_2, \ldots, v_k$ be a cycle in a 2-connected, weakly triangulated plane graph G with adjacency list representation Adj. Let c be an array of colors representing a 2-coloring of C such that each color class induces a path, respectively labelled $P_1 = v_1, v_2, \ldots, v_\ell$ and $P_2 = v_\ell, v_{\ell+1}, \ldots, v_k$. Assume that $c[v] = 0$ for each $v \in \text{Int}(C) - C$.

Output. For each vertex $v \in \text{Int}(C) - C$, a nonzero color will be assigned to $c[v]$ such that c represents a path 3-coloring of $\text{Int}(C)$ extending the original 2-coloring of C. Moreover, if $v \in \text{Int}(C) - C$ has a neighbor $u \in C$, then $c[v] \neq c[u]$.

Base case. If $k = 3$, then $C = \text{Int}(C)$ and G is colored.

Recursive step. Iterate through $\text{Adj}[v_1]$ to locate the vertex u immediately counterclockwise from v_k. Note that since G is triangulated, v_1, u, v_k is a face of G.

Case 1. Suppose that $c[u] \neq 0$. If $c[u] = c[v_k]$, then $u \in P_2$ and thus $u = v_{k-1}$ since G is triangulated and P_2 is an induced path. Recursively apply the algorithm to the cycle $C' = v_1, v_2, \ldots, v_{k-1}$.

Otherwise $c[u] = c[v_1]$, and it must be that $u = v_2 \in P_1$. Recursively apply the algorithm to the cycle $C' = v_2, v_3, \ldots, v_{k-1}$.

Case 2. Suppose that $c[u] = 0$ and therefore $u \in \text{Int}(C) - C$. Perform a breadth-first search of $\text{Int}(C) - C$ starting from the vertex u. Terminate the search upon locating a vertex w with adjacent neighbors $v_i \in P_1$ and $v_j \in P_2$ such that $i \neq 1$ or $j \neq k$.

Backtrack along the breadth-first search tree to construct a u, w-path of minimum length $P_3 = u_1, u_2, \ldots, u_r$. Color the vertices in P_3 with the third color not used in the 2-coloring of C. Define

$$C_1 = v_1, v_2, \ldots, v_i, u_r, u_{r-1}, \ldots, u_1 \text{ and } C_2 = u_1, u_2, \ldots, u_r, v_j, v_{j-1}, \ldots, v_k.$$

Apply the algorithm separately to C_1 and C_2. If $i = \ell$ and $j = \ell + 1$, then C was an induced cycle and we are done. Otherwise, also apply the algorithm to $C_3 = v_i, v_{i+1}, \ldots, v_j$.

Unfortunately Algorithm 3.2 is not linear. Consider the family of graphs $\{G_k\}_{k \in \mathbb{N}}$ depicted in Figure 3. Fix $k \in \mathbb{N}$ and note that $n = n(G_k) = k(k+1)/2 + 3$. Assume that the outer triangle is path 2-colored such that the top vertex is assigned a color distinct from the bottom two. At depth i of Algorithm 3.2 the shortest path through the interior

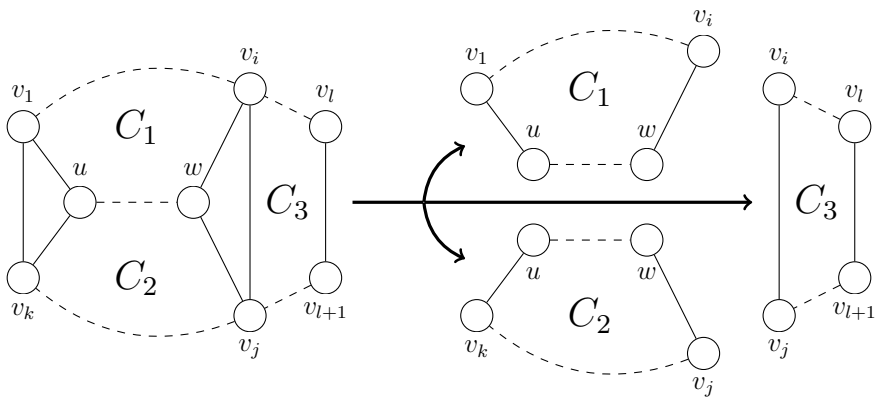

Figure 2: Dividing G along the edge v_iv_j and the uw-path P_3 in Algorithm 3.2.

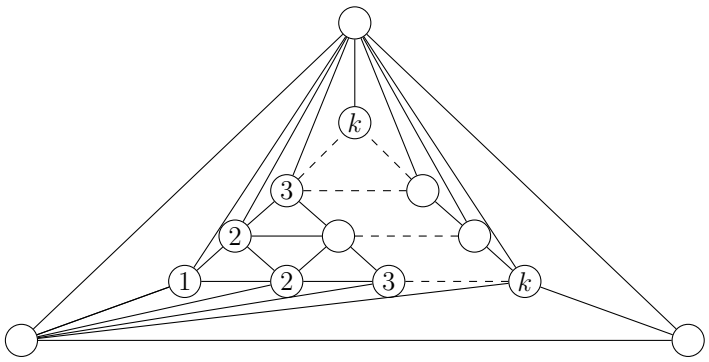

Figure 3: The collection of graphs $\{G_k\}_{k\in\mathbb{N}}$ for which Algorithm 3.2 performs poorly.

will be the path of length $r = k - i + 1$ directly along the base of the inner triangle. A breadth-first search of this inner triangle will hit all $r(r+1)/2$ vertices in order to find this path. Therefore the total number of operations performed will be

$$\Theta\left(\sum_{r=1}^{k} \frac{r(r+1)}{2}\right) = \Theta(n^{3/2}).$$

So Poh's algorithm implemented with breadth-first search is $\Omega(n^{3/2})$.

However, the correctness of Poh's proof only relied on locating some induced u, w-path. We will show that there exists a linear time implementation of Poh's algorithm that does not always find the shortest u, w-path.

For any subgraph H of G, let $N(H)$ be the set of vertices in $V(G)$ with a neighbor in $V(H)$. Our strategy will be to construct an induced path $P_3 = u_1, u_2, \ldots, u_d$ consisting of vertices in $N(P_1) - V(C)$.

Algorithm 3.3.
Input. Assume that $C = v_1, v_2, \ldots, v_k$ is an induced cycle in a 2-connected, weakly triangulated plane graph G with adjacency list representation Adj.

Let c be a length n array of colors representing a 2-coloring of C such that each color class induces a path, respectively $P_1 = v_1, v_2, \ldots, v_i$ and $P_2 = v_{i+1}, v_{i+2}, \ldots, v_k$. Let c_{P_1} and c_{P_2} be the colors for P_1 and P_2, respectively. Assume $c[v] = 0$ for each $v \in \text{Int}(C) - C$.

Let S be a length n array of integer marks and let m_{P_1} be an integer such that for each $v \in \text{Int}(C) - C$ we have $S[v] = m_{P_1}$ if and only if $v \in N(P_1)$.

Let $u \in \text{Int}(C) - C$ be the unique vertex such that v_1, u, v_k is a face. The vertex u and the entry for v_k in Adj$[u]$, together with Adj, c, and S, will serve as the concrete representation of the algorithm input.

Output. For each vertex $v \in \text{Int}(C) - C$, a nonzero color will be assigned to $c[v]$ such that c represents a path 3-coloring of Int(C) extending the original 2-coloring of C. Moreover, if $v \in N(P_1) - C$, then $c[v] \neq c_{P_1}$, and if $v \in N(P_2) - C$, then $c[v] \neq c_{P_2}$.

Procedure. The algorithm will construct a u, w-path P_3 in $\text{Int}(C) - C$ such that for each $v \in P_3$, $S[v] = m_{P_1}$, that is, $V(P_3) \subset N(P_1)$. The vertex $w \in \text{Int}(C) - C$ is the unique vertex such that w, v_i, v_{i+1} is a face; w will not be known prior to constructing P_3.

Each vertex of the path P_3 will be colored with $c_{P_3} \in \{1, 2, 3\} - \{c_{P_1}, c_{P_2}\}$. Each vertex in $N(P_3) - C - P_3$ will be marked with a new unique mark m_{P_3}.

We will store u_j, the last vertex added to $P_3 = u_1, u_2, \ldots, u_j$, along with the entry for u_{j-1} in Adj$[u_j]$. Define $u_0 = v_k$ for the purpose of identifying u_{j-1} when $j = 1$. Track the last edge encountered between a vertex $u_r \in P_3$ and a vertex $v_s \in P_2$, represented by u_r and the entry for v_s in Adj$[u_r]$. Initialize $j \leftarrow 1$, $u_1 \leftarrow u$, and $u_r v_s \leftarrow u v_k$.

Step 1. Iterate through the neighbors of u_j in counter-clockwise order, starting with the neighbor immediately counter-clockwise from u_{j-1}. Track an optional vertex $y \leftarrow$ NULL indicating the neighbor of u_j in $N(P_1) - C$ that will become u_{j+1} once all other the neighbors of u_j have been handled. Also store an optional edge $u_j v_\ell \leftarrow$ NULL, represented by the entry for v_j in Adj$[u_j]$, indicating the last neighbor of u_j in P_1 that was encountered. At each neighbor v of u_j one of the following cases will be satisfied.

Case 1.1. Suppose that $u_j v_\ell =$ NULL, that is, no neighbor of u_j in P_1 has been encountered yet. See Figure 4 for a sketch of each sub-case.

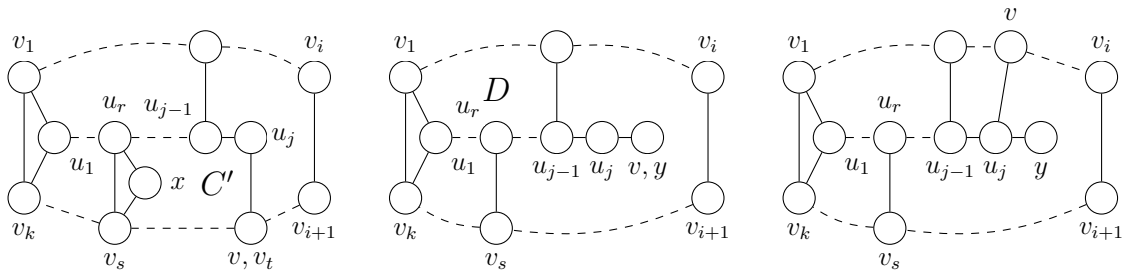

Figure 4: Algorithm 3.3, Case 1.1.2 (left), Case 1.1.3 (middle), and Case 1.1.4 (right).

Case 1.1.1. Suppose $c[v] = 0$ and $S[v] \neq m_{P_1}$, that is, $v \in \text{Int}(C) - C - N(P_1)$. Assign $S[v] \leftarrow m_{P_3}$.

Case 1.1.2. Suppose that $c[v] = c_{P_2}$, that is, $v = v_t \in P_2$. Observe that $P_1' = u_r, u_{r+1}, \ldots, u_j$ and $P_2' = v_s, v_{s+1}, \ldots, v_t$ are colored induced paths that together with the

edges $u_r v_s$ and $u_j v_t$ form an induced cycle C'. Moreover, the algorithm will have already marked each vertex in $N_{P_3} \cap V(\text{Int}(C') - C')$ with m_{P_3}. Let x be the neighbor of u_r immediately counter-clockwise from v_s. If $c[x] = 0$, make a recursive call with $u' \leftarrow x$ and $u'v'_k \leftarrow xv_s$ to path 3-color $\text{Int}(C')$. Then assign $u_r v_s \leftarrow u_j v_t$ to track that $u_j v_t$ is now the last edge between P_3 and P_2 that we've encountered.

Case 1.1.3. Suppose that $c[v] = 0$ and $S[v] = m_{P_1}$, that is, $v \in N(P_1) - C$. If $y \neq$ NULL, assign $S[v] \leftarrow m_{P_3}$. Otherwise, assign $y \leftarrow v$ and color $c[y] \leftarrow c_{P_3}$. We claim that $u_1, u_2, \ldots, u_j, y$ is an induced path. Since $u_j \in N(P_1)$, there exists an edge $u_j v_t$ where $v_t \in P_1$. Observe that $D = v_1, v_2, \ldots, v_t, u_j, u_{j-1}, \ldots, u_1$ is a cycle in $\text{Int}(C)$. Since $y \notin \text{Int}(D)$, if an edge yu_e exists with $u_e \in P_3 - u_j$, then the entry for y in $\text{Adj}[u_e]$ is between u_{e-1} and u_{e+1} counter-clockwise. But then y would have been encountered before u_{e+1} when iterating through $\text{Adj}[u_e]$, a contradiction since $S[y] = m_{P_1}$.

Case 1.1.4. Suppose that $c[v] = c_{P_1}$, that is, $v \in P_1$. Assign $u_j v_\ell \leftarrow u_j v$. If $y =$ NULL, it must be that $v = v_i \in P_1$ is the last vertex of P_1 and therefore $u_j = w$. To see this, note that the neighbor a of u_j immediately clockwise from v is adjacent to $v \in P_1$, but a was not assigned to y. Therefore $c[a] = c_{P_2}$ and so $a = v_{i+1}$ and $v = v_i$. Since u_j, v_i, v_{i+1} is a face, $u_j = w$ by definition.

Case 1.2. Suppose that $u_j v_\ell \neq$ NULL. Let z be the neighbor of u_j immediately counter-clockwise from v_ℓ. See Figure 5 for a sketch of each sub-case.

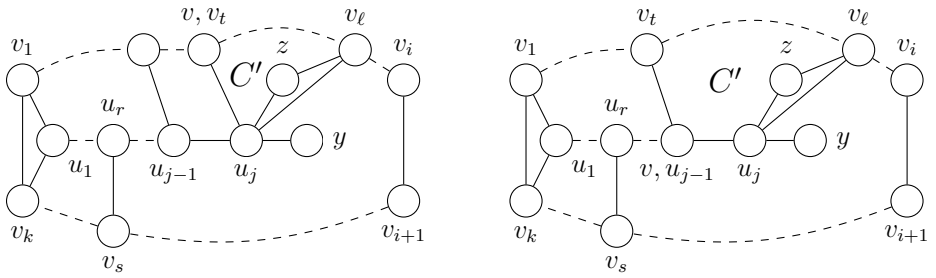

Figure 5: Algorithm 3.3, Case 1.2.2 (left) and Case 1.2.3 (right).

Case 1.2.1. Suppose that $c[v] = 0$, that is, $v \in \text{Int}(C) - C - P_3$. Assign $S[v] \leftarrow m_{P_3}$.

Case 1.2.2. Suppose that $c[v] = c_{P_1}$, that is, $v = v_t \in P_1$. Observe that $C' = u_j, v_t, v_{t+1}, \ldots, v_\ell$ is an induced cycle. Moreover, the algorithm will have already marked each vertex in $N(P_3) \cap V(\text{Int}(C') - C')$ with m_{P_3}. If $c[z] = 0$, make a recursive call with $u' \leftarrow z$ and $u'v'_k \leftarrow zv_\ell$ to path 3-color $\text{Int}(C')$. Otherwise, $z = v$ and $\text{Int}(C') = C'$.

Case 1.2.3. Suppose that $c[v] \neq 0$ and $c[v] \neq c_{P_1}$. Then it must be that $c[v] = c_{P_3}$ and $v = u_{j-1}$. Choose the largest t such that $v_t \in P_1$ and $u_{j-1}v_t$ is an edge. Observe that $C' = u_j, u_{j-1}, v_t, v_{t+1}, \ldots, v_\ell$ is an induced cycle. Moreover, the algorithm will have marked each vertex in $N(P_3) \cap V(\text{Int}(C') - C')$ with m_{P_3}. If $c[z] = 0$, make a recursive call with $u' \leftarrow z$ and $u'v'_k \leftarrow zv_\ell$ to color $\text{Int}(C')$. Otherwise, $z = v$ and $\text{Int}(C') = C'$.

Step 2. If $y \neq$ NULL, then assign $j \leftarrow j+1$, $u_j \leftarrow y$, $y \leftarrow$ NULL, and $u_j v_\ell \leftarrow$ NULL, then return to Step 1. Otherwise, $u_j = w$ and $\text{Int}(C)$ has been path 3-colored.

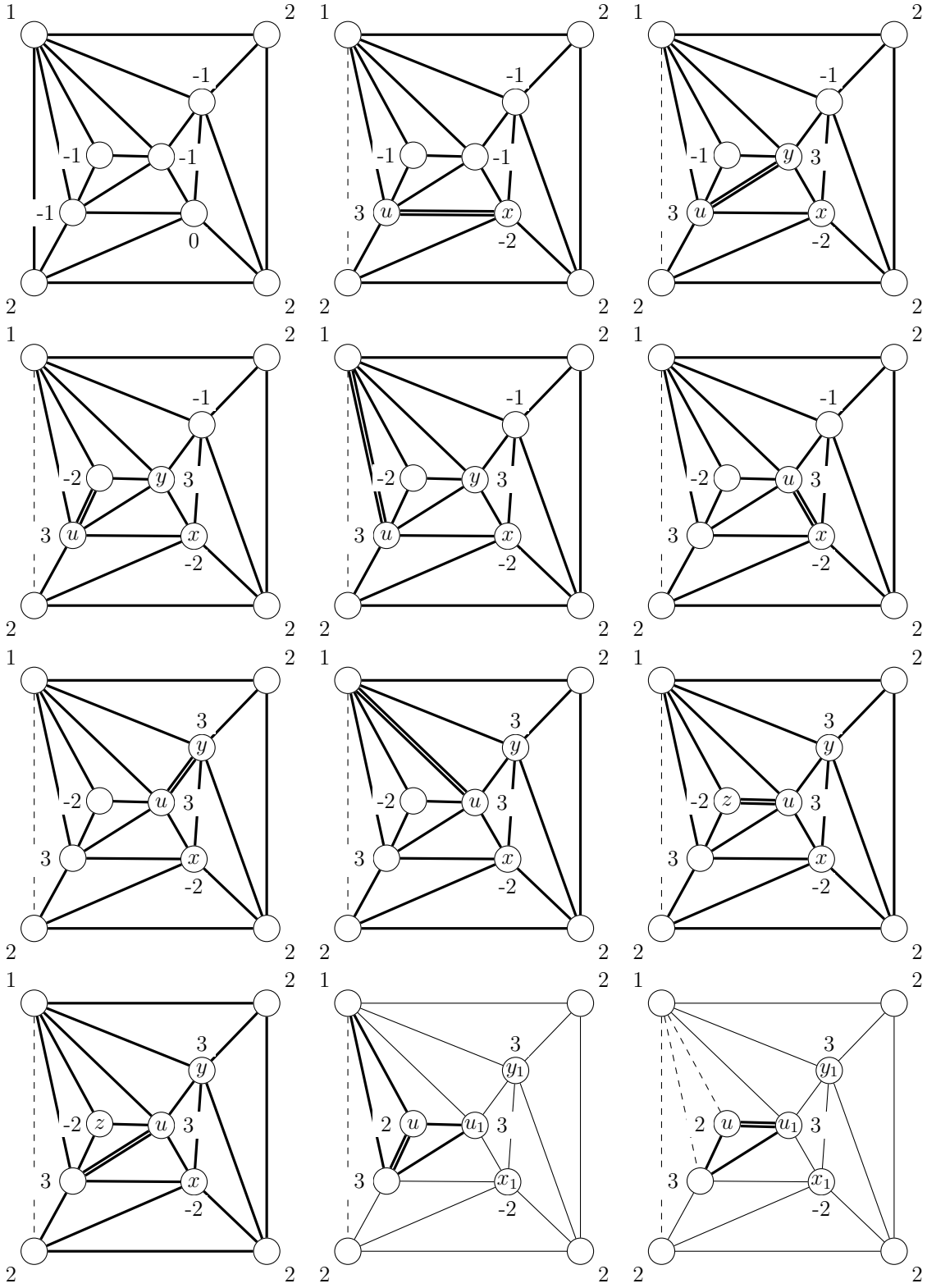

Figure 6: An example of Algorithm 3.3. Each vertex $v \in G$ is labelled with $c[v]$ if $c[v] > 0$, and labelled with $-S[v]$ otherwise.

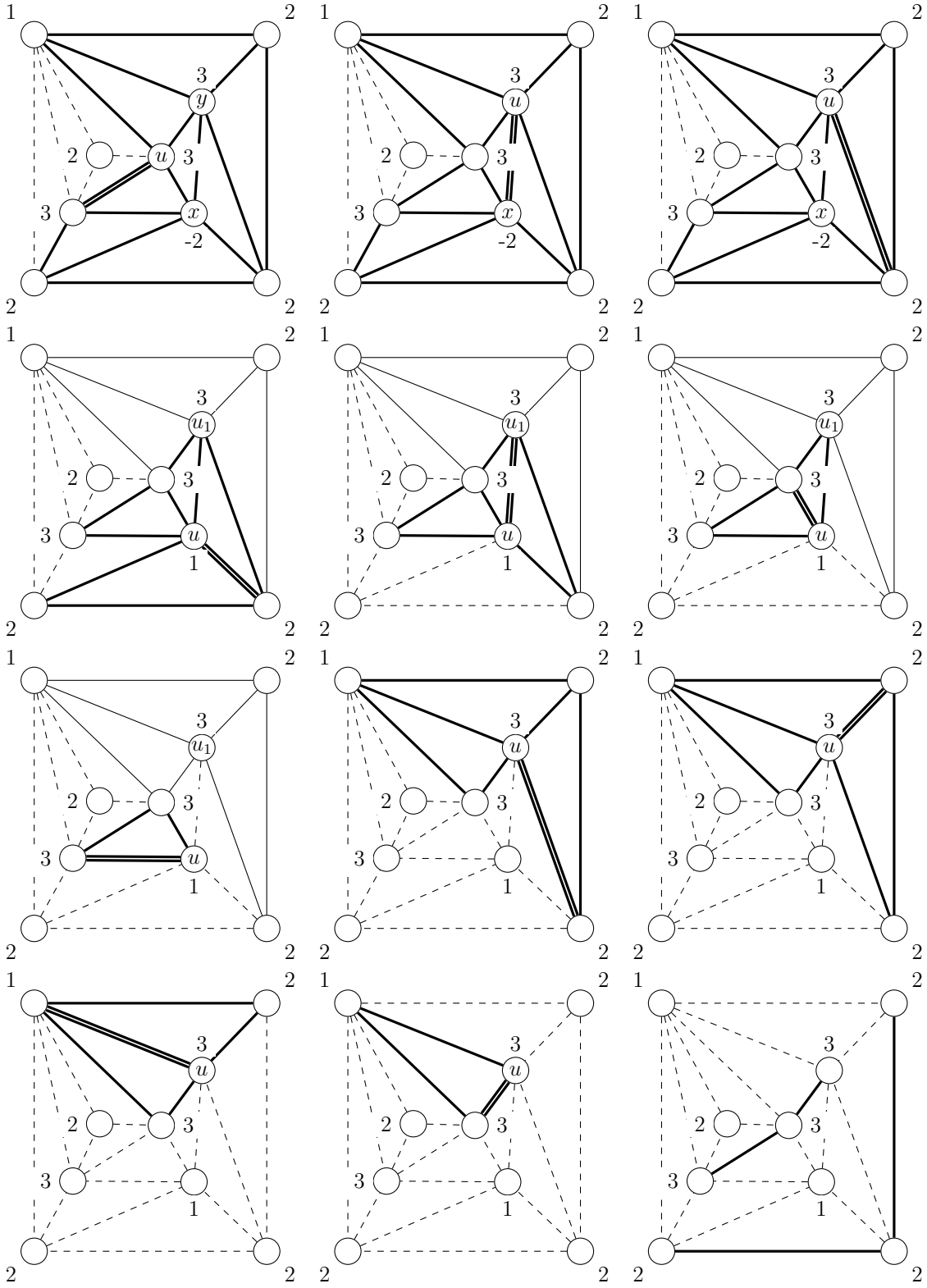

Figure 7: An Algorithm 3.3 example, continued from Figure 6.

While executing Algorithm 3.3 we iterate through the adjacency list of each vertex $v \in G$ exactly twice: once to orient Adj[v] around a particular edge when v is colored, and once to examine each neighbor in Adj[v] during Step 1. Therefore the time complexity of the algorithm is $\mathcal{O}(m) = \mathcal{O}(n)$. See Figure 6 and Figure 7 for a concrete example of Algorithm 3.3.

Given a triangulated plane graph G with adjacency list representation Adj, we can set up the initial conditions for Algorithm 3.3 as follows. First, create length n integer arrays for c and S, initializing each entry to zero. Let $C = v_1, v_2, v_3$ be the outer triangle of G, labeled in clockwise order. Color $c[v_1] \leftarrow 1$, then color $c[v_2] \leftarrow 2$ and $c[v_3] \leftarrow 2$. Iterate through Adj[v_1] and mark $S[v] \leftarrow 1$ for each neighbor v of v_1. Let u be the vertex immediately counter-clockwise from v_3 in Adj[v_1]. The vertex u and the entry for v_3 in Adj[u], together with Adj, c, and S form a valid input for Algorithm 3.3.

4 Path List-Coloring

In this section we describe a linear time algorithm to path color plane graphs such that each vertex receives a color from a specified list. Hartman showed that this is always possible when each vertex is given a list of 3 colors [7, Thm. 4.1]. Around the same time Škrekovski proved a slightly weaker result using the same coloring strategy [9, Thm. 2.2b].

The path list-coloring procedure discussed in this section is based on the constructive proofs found in Hartman and Škrekovski's papers, but it "localizes" the logic to proceed through the graph one edge at a time. The resulting alogorithm will produce different colorings in some situations.

A *list assignment* of a graph G is a function $L : V(G) \to P_{<\aleph_0}(\mathbb{N})$ assigning a finite list of colors to each vertex. If L is a list assignment of G, an *L-coloring* of G is a coloring function $c : V(G) \to \mathbb{N}$ such that $c(v) \in L(v)$ for each $v \in V(G)$.

Given a graph G and a coloring c of G, for each $v \in G$ we define $\deg_c(v)$ to be the number of neighbors of v that share a color with v. Equivalently, $\deg_c(v)$ is the degree of v in the subgraph of G induced by the color class of $c(v)$.

Let C be the outer face of a 2-connected, weakly triangulated plane graph G and let $u, v \in C$ be vertices. If $n(G) \geq 3$, then C is a cycle and we define $C(u, v)$ to be the clockwise u, v-path around C. If $n(G) < 3$ then we define $C(u, v) = C$ if $u \neq v$, and $C(u, u) = u$.

Lemma 4.1. *Let G be a 2-connected, weakly triangulated plane graph with outer face C. Let $x, y, z \in C$ be vertices (not necessarily distinct) such that $z \in C(x, y)$. Let L be a list assigment of G such that*

$$\begin{aligned} &|L(v)| = 1 \text{ for } v \in \{x, y, z\}, \\ &|L(v)| \geq 2 \text{ for } v \in C - \{x, y, z\}, \\ &|L(v)| \geq 3 \text{ for } v \in G - C, \end{aligned}$$

and if $v \in C(x, z) - z$, then $L(v) \cap L(z) = \emptyset$. There exists a path L-coloring of G such

that $deg_c(x) \leq 1$, $deg_c(y) \leq 1$, *and* $deg_c(z) \leq 1$. *Moreover, if* $z = y$ *or* $C(z, y) = z, y$ *and* $L(z) \cap L(y) = \emptyset$, *then* $deg_c(z) = 0$.

Proof. Define $c(x) \in L(x)$, $c(y) \in L(y)$, and $c(z) \in L(z)$ to be the color in each respective list.

Suppose that $m = |E(G)| \leq 2$ and therefore $G = C$. If $m = 0$, then $n(G) = 1$ and $x = y$ and G is colored. If $m = 1$, then $n(G) = 2$. If $x \neq y$, then $C = x, y$ and G is colored. If $x = y$, then the remaining vertex $v \in G - C(x, y)$ satisfies $|L(v)| \geq 2$. Choose $c(v) \in L(v) - \{c(x)\}$.

We proceed by induction on the number of edges $m = |E(G)|$. Let $u_1, u_2, \ldots, u_k$ label the neighbors of z in counter-clockwise order such that u_1 is the vertex immediately counter-clockwise from z around C and u_k is the vertex immediately clockwise from z around C.

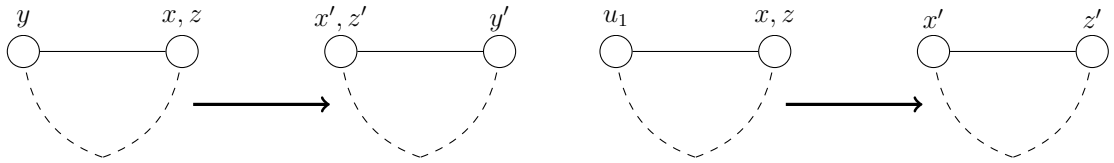

Figure 8: The proof of Lemma 4.1, Case 1 (left) and Case 2 (right).

Case 1. Suppose that $u_1 = y$. Note that $z = x$ since $z \in C(x, y)$. Therefore $V(C(z, y)) = V(C)$ and $|V(C(z, y))| \geq 3$. Apply the lemma with vertices re-labeled as $x' = y$, $y' = x$, and $z' = y$ to find a path L-coloring c of G.

Case 2. Suppose that $u_1 \neq y$ and $z = x$. Then $u_1 \notin \{x, y, z\}$ and $|L(u_1)| \geq 2$. Pick $c(u_1) \in L(u_1) - c(z)$. Define $L'(u_1) = \{c(u_1)\}$ and $L'(v) = L(v)$ for each $v \in G - u_1$. Apply the lemma with vertices re-labeled as $x' = u_1$, $y' = y$, and $z' = z$.

Case 3. Suppose that $u_1 \neq y$ and $z \neq x$. Our strategy will be to apply the inductive hypothesis to $G - zu_1$ with the embedding inherited from G. Because $u_1 \in C(x, z)$, it is guaranteed that $L(u_1) \cap L(z) = \emptyset$. Thus any path L-coloring of $G - zu_1$ will also be a path L-coloring of G.

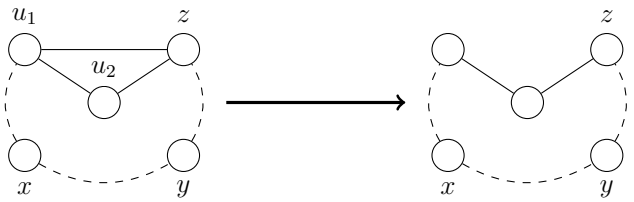

Figure 9: The proof of Lemma 4.1, Case 3.1.

Case 3.1. Suppose that $u_2 \in G - C$. Define $L'(u_2) = L(u_2) - c(z)$ and $L'(v) = L(v)$ for each $v \in G - u_2$. Note that $|L'(u_2)| \geq |L(v)| - 1 \geq 2$. Let C' be the outer face of $G - zu_1$ and observe that $C'(x, z)$ is equal to $C(x, z)$ with the edge u_1z removed and

the path u_1, u_2, z added. Since $|L'(u_2)| \geq 2$ and $L'(u_2) \cap L'(z) = \emptyset$, we may apply the inductive hypothesis to find a path L'-coloring of $G - zu_1$.

Case 3.2. Suppose that $u_2 \in C$. Observe that u_2 is a cut-vertex of $G - zu_1$. Define $C_1 = C(u_2, u_1) + u_1u_2$ and $C_2 = C(z, u_2) + zu_2$. Define $G_1 = \text{Int}(C_1)$ and $G_2 = \text{Int}(C_2)$. The subgraphs G_1 and G_2 are the two 2-connected components (blocks) of $G - zu_1$. Note that if $k = 2$, then $G_1 = G - z$ and $G_2 = z, u_2$.

In each subsequent case we will apply the inductive hypothesis to produce a path L-coloring c_1 of G_1 and a path L-coloring c_2 of G_2 such that $c_1(u_2) = c_2(u_2)$. Let c be the L-coloring of G defined by $v \mapsto c_1(v)$ for $v \in G_1$ and $v \mapsto c_2(v)$ for $v \in G_2$. Observe that $\deg_c(v) = \deg_{c_1}(v)$ for each $v \in G_1 - u_2$, $\deg_c(v) = \deg_{c_2}(v)$ for each $v \in G_2 - u_2$, and $\deg_c(u_2) = \deg_{c_1}(u_2) + \deg_{c_2}(u_2)$. To show that c is a path L-coloring of G it suffices to show that $\deg_c(u_2) \leq 2$.

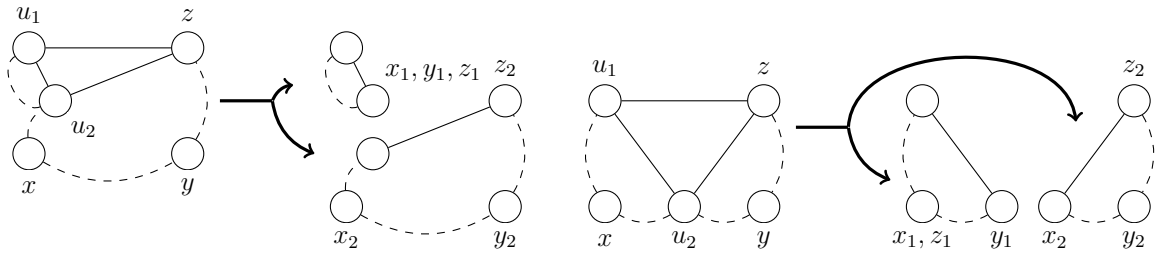

Figure 10: The proof of Lemma 4.1, Case 3.2.1 (left) and Case 3.2.2 (right).

Case 3.2.1. Suppose that $u_2 \in C(x, z)$. Observe that $x, y, z \in C_2$. Apply the inductive hypothesis to produce a path L-coloring c_2 of G_2 with designated vertices $x_2 = x$, $y_2 = y$, and $z_2 = z$. Define $L'(u_2) = \{c_2(u_2)\}$ and $L'(v) = L(v)$ for each $v \in G_1 - u_2$. Apply the inductive hypothesis to produce a path L'-coloring c_1 of G_1 with the single designated vertex $x_1 = y_1 = z_1 = u_2$. Since $\deg_{c_1}(u_2) = 0$, it follows that c is a path L-coloring of G satisfying the lemma.

Case 3.2.2. Suppose that $u_2 \in C(y, x) - x - y$. Pick $c(u_2) \in L(u_2) - c(z)$. Define $L'(u_2) = \{c(u_2)\}$ and $L'(v) = L(v)$ for each $v \in G - u_2$. Observe that $x \in C_1$ and $z, y \in C_2$. Apply the inductive hypothesis to produce path L'-coloring c_1 of G_1 with designated vertices $x_1 = z_1 = x$ and $y_1 = u_2$. Then find a path L'-coloring c_2 of G_2 with designated vertices $x_2 = u_2$, $y_2 = y$, and $z_2 = z$. Note that $\deg_{c_1}(u_2) \leq 1$ and $\deg_{c_2}(u_2) \leq 1$, and thus c is a path L-coloring of G satisfying the lemma.

Case 3.2.3. Suppose that $u_2 \in C(z, y)$. Note that $z \neq y$. There are two distinct cases to consider.

Case 3.2.3.1. Suppose that $c(z) \in L(u_2)$. Then either $u_2 = y$ and $L(z) \cap L(y) = \{c(z)\}$, or $C(z, y) \neq z, y$. Define $L'(u_2) = \{c(z)\}$ and $L'(v) = L(v)$ for each $v \in G - u_2$. Construct a path L'-coloring c_1 of G_1 with designated vertices $x_1 = x$, $y_1 = y$, and $z_1 = u_2$. Similarly, construct a path L'-coloring c_2 of G_2 with designated vertices $x_2 = u_2$, $y_2 = z$, and $z_2 = u_2$. Since $\deg_{c_1}(u_2) \leq 1$ and $\deg_{c_2}(u_2) = 1$, it follows that c is a path L-coloring of G.

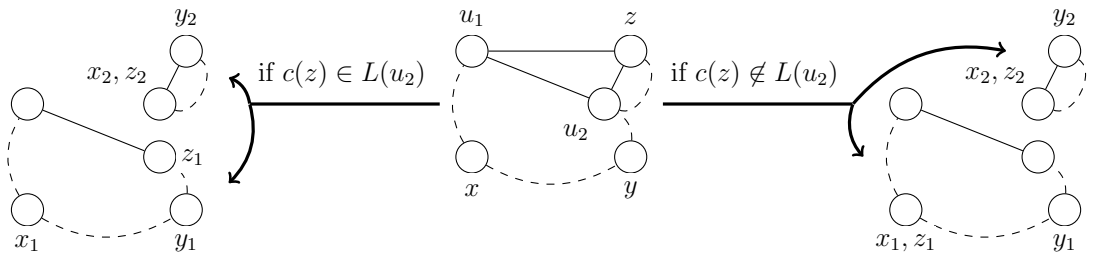

Figure 11: The proof of Lemma 4.1, Case 3.2.3.

Case 3.2.3.2. Suppose that $c(z) \notin L(u_2)$. Find a path L-coloring c_1 of G_1 with designated vertices $x_1 = x$, $y_1 = y$, and $z_1 = x$. Define $L'(u_2) = \{c_1(u_2)\}$ and $L'(v) = L(v)$ for each $v \in G_2 - u_2$. Find a path L'-coloring c_2 of G_2 with designated vertices $x_2 = u_2$, $y_2 = z$, and $z_2 = u_2$. Because $C(z_2, y_2) = z_2, y_2$ and $L'(z_2) \cap L'(y_2) = \emptyset$, it is guaranteed that $\deg_{c_2}(u_2) = \deg_{c_2}(z_2) = 0$. Thus c is a path L-coloring of G.

Suppose that $C(z, y) = z, y$. Then it must be that $u_2 = y$ and $k = 2$. Therefore $G_2 = C_2 = z, y$ and $\deg_c(z) = 0$ since $c(z) \notin L(u_2)$. $\square$

Let G be a planar graph and L a list assignment such that $|L(v)| \geq 3$ for each $v \in G$. Compute a planar embedding and add edges to produce a triangulated plane graph G'. Pick $u \in G'$ to be a vertex on the outer face and $c(u) \in L(u)$. Define L' to be the list assignment such that $L'(u) = \{c(u)\}$ and $L'(v) = L(v)$ for $v \in G' - u$. By Lemma 4.1 there exists a path L'-coloring c of G'. Clearly c is also a path L-coloring of G. Therefore Theorem 1.2 follows immediately from Lemma 4.1.

The proof of Lemma 4.1 is a constructive algorithm that may be implemented for plane graphs represented by rotation scheme ordered augmented adjacency lists. We will assume that both forward and backward iteration in augmented adjacency lists is a constant time operation. The available C implementation represents each augmented adjacency list as an array of entries [3].

Algorithm 4.2.
Input. Let C be a cycle or length 2 path in a 2-connected, weakly triangulated plane graph G represented by augmented adjacency lists. Let $x, y, z \in C$ be vertices such that $z \in C(x, y)$.

Let L be an array of lists of colors such that for $v \in \{x, y, z\}$ the list $L[v]$ has length one, for $v \in C - \{x, y, z\}$ the list $L[v]$ has length two or three, and for $v \in \text{Int}(C) - C$ the list $L[v]$ has length three. Furthermore, for each $v \in C(x, z) - z$ assume that $L[v]$ does not contain the color in $L[z]$.

Let N be an array of pairs of references such that for each $v \in C$ the pair $N[v] = (r_1, r_2)$ contains a reference r_1 to the neighbor the immediately counter-clockwise from v around C, and a reference r_2 to the neighbor immediately clockwise from v around C.

Let S be an array of integers such that for each $v \in C(x, z) - z$ we have $S[v] = S[x]$ and for all vertices $v \in C - C(x, z)$ we have $S[v] \neq S[x]$. Furthermore, for each vertices $v \in C$ assume that $S[v] \neq 0$, and for each $v \in \text{Int}(C) - C$ assume that $S[v] = 0$.

Finally, let M be an array of integers such that for all vertices $v \in C(z, y)$ we have $M[S[v]] = S[y]$, and for all vertices $v \in C - C(z, y)$ we have $M[S[v]] \neq S[y]$.

Output. Elements will be removed from the lists in L such that for each $v \in \text{Int}(C)$ the list $L[v]$ will contain a single color. The coloring of $\text{Int}(C)$ corresponding to the remaining colors will be a path coloring c such that $\deg_c(x) \leq 1$, $\deg_c(y) \leq 1$, and $\deg_c(z) \leq 1$. If $z = y$, or if $C(z, y) = z, y$ and $L[z]$ doesn't contain the color in $L[y]$, then $\deg_c(z) = 0$.

Procedure. Let $N[z] = (r_1, r_2)$. Define the vertex u_1 to be the neighbor of z corresponding to r_1.

Base Case. If $r_1 = r_2$, then $C = z, u_1$ is a length 2 path. If $u_1 \neq x$ and $u_1 \neq y$, remove the color in $L[z]$ from $L[u_1]$. If more than one color still remains in $L[u_1]$, remove arbitrary colors until a single color remains.

Recursive Step. Suppose that $r_1 \neq r_2$. Note that $n(C) > 2$ and therefore C is a cycle. Define u_2 to be the neighbor of z immediately counter-clockwise from u_1.

Case 1. Suppose that $u_1 = y$. In this case it must be that $z = x$. Assign $S[x]$ and $S[y]$ new unique marks. Assign $M[S[x]] \leftarrow S[x]$ and $M[S[y]] \leftarrow S[y]$. Make a recursive call re-assigning $x' \leftarrow y$, $y' \leftarrow x$, and $z' \leftarrow y$.

Case 2. Suppose that $z = x$ and $u_1 \neq y$. Remove the color in $L[z]$ from $L[u_1]$, and then remove arbitrary colors from $L[u_1]$ until a single color remains. Assign $x \leftarrow u_1$, set $S[u_1]$ to be a new unique mark, assign $M[S[u_1]] \leftarrow S[u_1]$, and make a recursive call.

Case 3. Suppose that $z \neq x$ and $u_1 \neq y$. It is this case that makes use of the back references in the augmented adjacency list representation.

Case 3.1. Suppose that $S[u_2] = 0$, that is, $u_2 \in \text{Int}(C) - C$. Assign $S[u_2] \leftarrow S[x]$ and initialize $N[u_2] \leftarrow (s_1, s_2)$ where s_1 is a reference to the entry for u_1 in $\text{Adj}[u_2]$, and s_2 is a reference to the entry for z. Remove the color in $L[z]$ from $L[u_2]$ and remove the edge zu_1 by adjusting $N[u_1]$ and $N[z]$. Make a recursive call with x, y, and z the same as before.

Case 3.2. Suppose that $S[u_2] \neq 0$, that is, $u_2 \in C$. Just as in Case 3.2 of the proof of Lemma 4.1, we will separately consider the two blocks G_1 and G_2 of $\text{Int}(C) - zu_1$.

It is simple to remove the edge zu_1 by adjusting $N[z]$ and $N[u_1]$ to exclude the corresponding entries. It remains to "split" the neighborhood of u_2 at the edge u_2z such that the recursive call on G_1 considers only the neighbors of u_2 counter-clockwise from z, and the call on G_2 considers z and all neighbors clockwise from z.

Let $N[u_2] = (s_1, s_2)$. Let s_3 be a reference to z's entry in $\text{Adj}[u_2]$ and $s_3 + 1$ a reference to the next entry counter-clockwise from z in $\text{Adj}[u_2]$. To represent the subgraph G_1 from the proof's Case 3.2, we assign $N[u_2] \leftarrow (s_3 + 1, s_2)$. To represent the graph G_2, we assign $N[u_2] \leftarrow (s_1, s_3)$.

Case 3.2.1. Suppose that $S[u_2] = S[x]$, i.e. $u_2 \in C(x, z)$. Make a recursive call on G_2 with $x_2 \leftarrow x$, $y_2 \leftarrow y$, $z_2 \leftarrow z$. Assign $S[u_2]$ a new unique mark, set $M[S[u_2]] \leftarrow S[u_2]$, and make a recursive call on G_1 with x_1, y_1, z_1 all equal to u_2.

Case 3.2.2. Suppose that $z \neq x$ and $S[u_2] \neq 0$, but $S[u_2] \neq S[x]$ and $M[S[u_2]] \neq S[y]$. In other words, suppose that $u_2 \in C(y, x) - x - y$. First remove the color in $L[z]$ from $L[u_2]$, if it exists, and then remove arbitrary colors until $L[u_2]$ is of length one. Assign

$S[u_2] = S[x]$. Make a recursive call on G_1 with $x_1 \leftarrow x$, $y_1 \leftarrow u_2$, and $z_1 \leftarrow x$. Assign $S[u_2]$ a new unique mark and $M[S[u_2]] \leftarrow S[u_2]$. Make a recursive call on G_2 with $x_2 \leftarrow u_2$, $y_2 \leftarrow y$, and $z_2 \leftarrow z$, splitting $N[u_2]$ at the edge u_2z to contain z and all neighbors clockwise from z.

Case 3.2.3. Suppose that $z \neq x$ and $M[S[u_2]] = S[y]$, i.e. $u_2 \in C(z, y)$. There are two cases to consider.

Case 3.2.3.1. Suppose that the color in $L[z]$ is in $L[u_2]$. Remove all colors other than the one in $L[z]$ from $L[u_2]$. Make a recursive call on G_1 with $x_1 \leftarrow x$, $y_1 \leftarrow y$, and $z_1 \leftarrow u_2$. Then assign $S[z]$ and $S[u_2]$ the same new unique mark, and assign $M[S[z]] \leftarrow S[z]$. Make a recursive call on G_2 with $x_2 \leftarrow u_2$, $y_2 \leftarrow z$, and $z_2 \leftarrow u_2$.

Case 3.2.3.2. Suppose that the color in $L[z]$ is not in $L[u_2]$. Assign $M[S[x]] \leftarrow S[y]$. Make a recursive call on G_1 with $x_1 \leftarrow x$, $y_1 \leftarrow y$, and $z_1 \leftarrow x$. Assign $S[z]$ and $S[u_2]$ the same new unique mark, and set $M[S[z]] \leftarrow S[z]$. Make a recursive call on G_2 with $x_2 \leftarrow u_2$, $y_2 \leftarrow z$, and $z_2 \leftarrow u_2$.

Let G be a triangulated plane graph with n vertices and m edges, represented by augmented adjacency lists. Let L be a size n array of color lists of length 3, representing a list assignment of G. We may path L-color G using Algorithm 4.2 as follows.

Create an array of reference pairs N of size n, and initialize $N[v]$ for each $v \in C$ on the outer face of G to satisfy the requirements of Algorithm 4.2. Create a size n array of integers S, and initialzie $S[v] \leftarrow 1$ for each $v \in C$ and $S[v] \leftarrow 0$ for each $v \in G - C$. Pick an arbitrary $u \in C$, assign $S[u] \leftarrow 2$. Remove all but one color from $L[u]$. Create a size $2m + 1$ array of integers M and initialize $M[i] \leftarrow i$ for each $i \in \{0, 1, 2\}$. Finally, apply Algorithm 4.2 with $x = y = z = u$ to reduce each list in L to represent a path L-coloring of G.

Every recursive case in Algorithm 4.2 performs a fixed number of constant time operations. Each case removes one edge from the graph representation, except for Case 1 and Case 2. In Case 1 a single recursive call is made with an input that will not itself satisfy Case 1. In Case 2 a recursive call is made with an input that will not itself satisfy Case 1 or Case 2. Therefore the number of operations performed is $\mathcal{O}(m) = \mathcal{O}(n)$. See Figure 12 and Figure 13 for a concrete example of Algorithm 4.2.

5 Experimental performance and parallelism

A C implementation of each algorithm was run on randomly generated triangulated plane graphs up to $10,000,000$ vertices [3], with benchmark timings shown in Figure 15. The sample size for each benchmark was $10^9/n(G)$, i.e. each algorithm was run on 10^4 graphs with $n = 10^5$ vertices. Benchmark results for a simple breadth-first search algorithm are also shown as a baseline linear algorithm that hits every half-edge of the graph. All benchmarks were run on an x86_64 Intel N100 processor. The benchmark source code is included with the implementation.

The performance of Algorithm 4.2 was surprisingly close to Algorithm 3.3, despite the fact that Algorithm 4.2 is more complex and has a larger memory footprint. However,

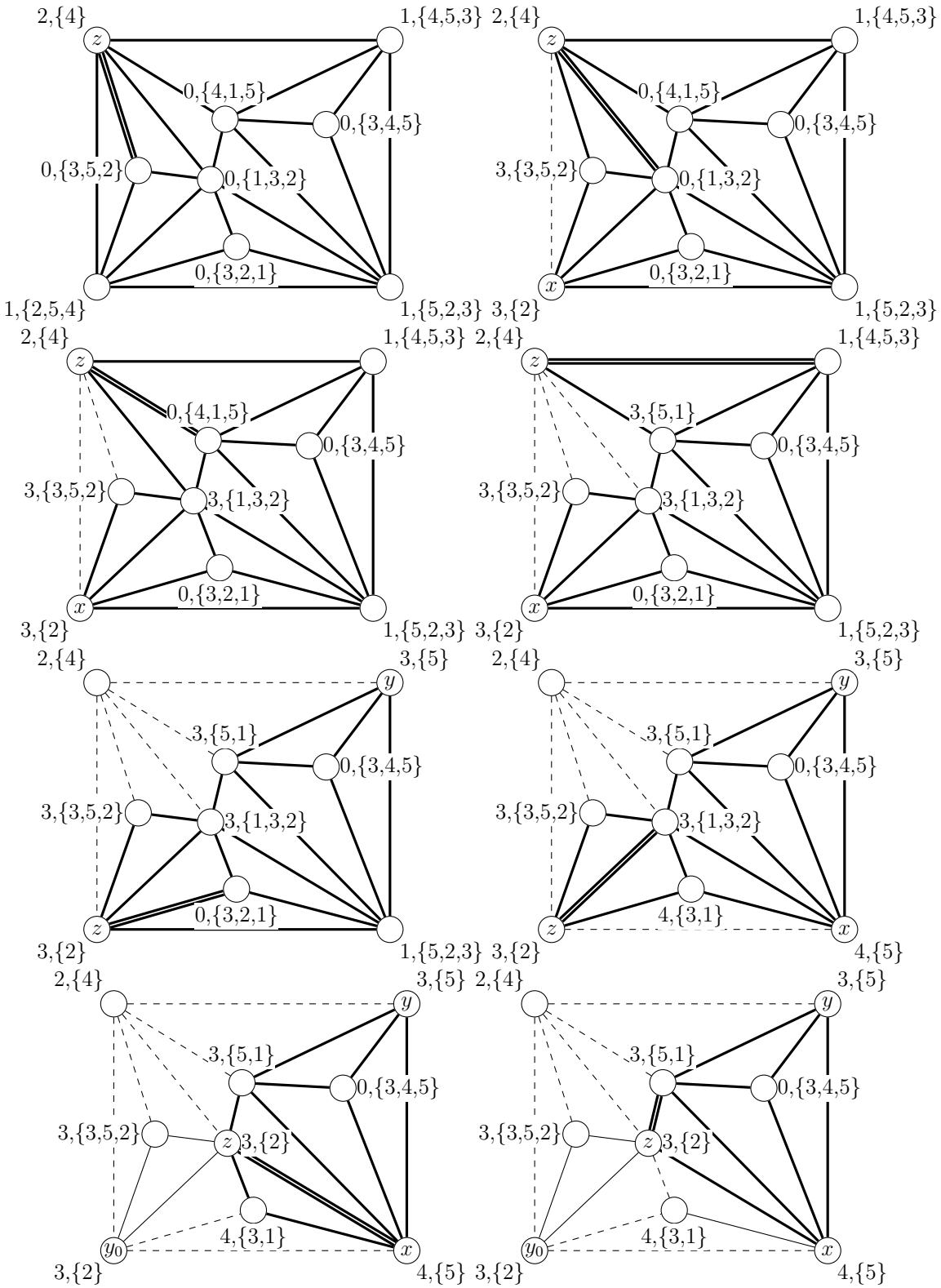

Figure 12: An example of Algorithm 4.2. Each vertex $v \in G$ is labelled with $S[v], L[v]$.

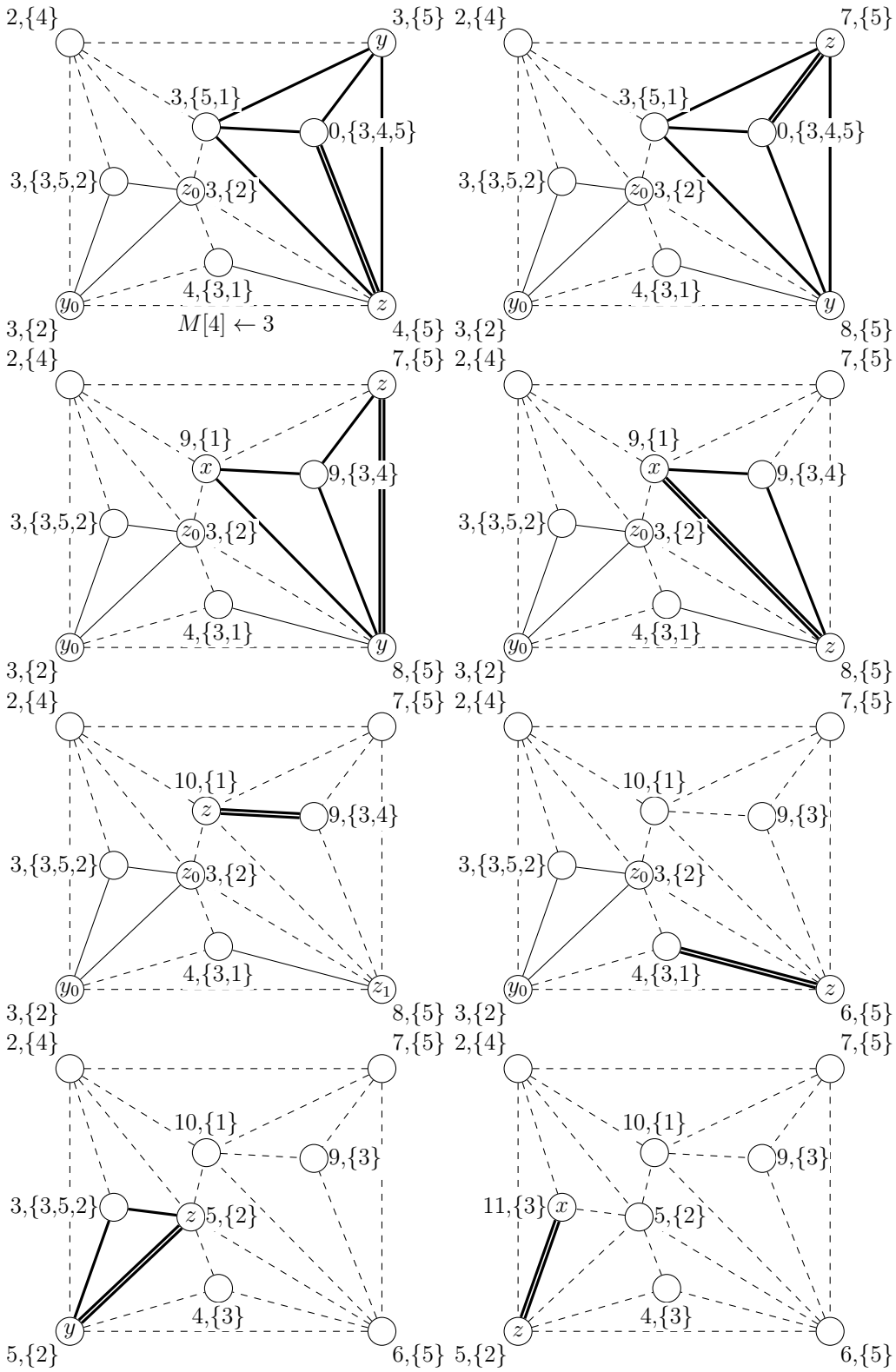

Figure 13: An Algorithm 4.2 example, continued from Figure 12.

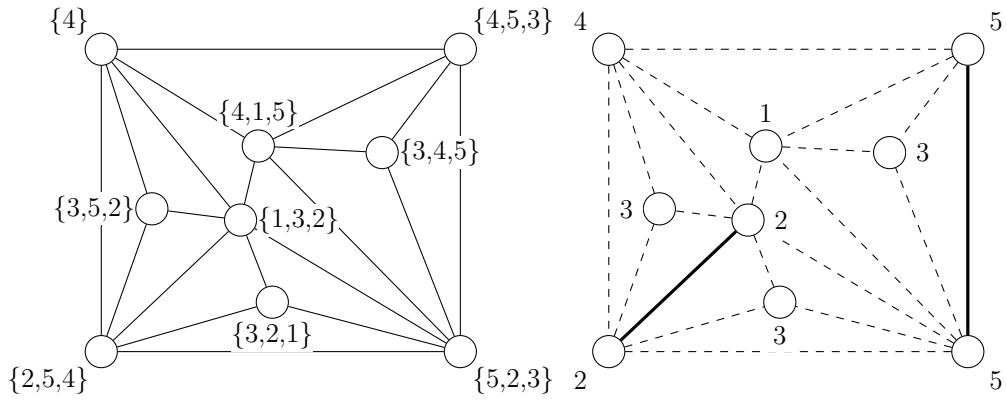

Figure 14: The initial list assignment L (left) and the resulting path L-coloring (right) from the Algorithm 4.2 example shown in Figure 12 and Figure 13.

	$n = 10^3$	$n = 10^4$	$n = 10^5$	$n = 10^6$	$n = 10^7$
Breadth-first search	$2.42 \cdot 10^4$	$2.83 \cdot 10^5$	$3.67 \cdot 10^6$	$1.01 \cdot 10^8$	$1.26 \cdot 10^9$
Algorithm 2.1	$6.58 \cdot 10^4$	$6.61 \cdot 10^5$	$1.78 \cdot 10^7$	$4.17 \cdot 10^8$	$4.76 \cdot 10^9$
Algorithm 3.2	$2.39 \cdot 10^5$	$3.47 \cdot 10^6$	$5.09 \cdot 10^7$	$1.19 \cdot 10^9$	$1.88 \cdot 10^{10}$
Algorithm 3.3	$5.74 \cdot 10^4$	$6.33 \cdot 10^5$	$7.20 \cdot 10^6$	$1.32 \cdot 10^8$	$1.51 \cdot 10^9$
Algorithm 4.2	$6.43 \cdot 10^4$	$7.28 \cdot 10^5$	$9.84 \cdot 10^6$	$1.83 \cdot 10^8$	$2.11 \cdot 10^9$

Figure 15: Average time (nanoseconds) per triangulated plane graph.

Algorithm 4.2 requires an augmented adjacency list representation, which the benchmarks for Algorithm 2.1 show is relatively expensive to construct.

Both Algorithm 3.3 and Algorithm 4.2 perform recursive calls that operate on subgraphs that are disjoint except for select vertices on their respective outer faces. Therefore it is possible to maintain a stack of independent recursive frames and have a pool of threads operate on these frames concurrently.

In the case of Algorithm 3.3 this can be easily implemented as each frame writes only to vertices interior to the outer face. Therefore the algorithm can maintain a shared lock-guarded stack from which all threads push and pop recursive frames. If no frames are available, a thread must idle until either a new frame is pushed to the stack by another thread or until all threads are idle.

To reduce lock contention, each thread should also maintain a small local stack of frames, and only access the shared stack when this local stack is full or empty. For the implementation and hardware used for benchmarks, the most optimal setup proved to be a local stack of 16 frames for each thread. If a local stack filled up, 8 frames were pushed to the shared stack. If a local stack emptied, 8 frames were popped from the shared stack.

Benchmarks showed performance improvements when applying the parallel Algorithm 3.3 to plane graphs on the order of 10^4 vertices or more; see Figure 16.

Threads	$n = 10^3$	$n = 10^4$	$n = 10^5$	$n = 10^6$	$n = 10^7$
1	$5.74 \cdot 10^4$	$6.33 \cdot 10^5$	$7.20 \cdot 10^6$	$1.32 \cdot 10^8$	$1.51 \cdot 10^9$
2	$5.69 \cdot 10^4$	$4.50 \cdot 10^5$	$4.17 \cdot 10^6$	$7.48 \cdot 10^7$	$8.56 \cdot 10^8$
3	$5.38 \cdot 10^4$	$3.41 \cdot 10^5$	$3.25 \cdot 10^6$	$5.61 \cdot 10^7$	$6.46 \cdot 10^8$
4	$5.51 \cdot 10^4$	$2.90 \cdot 10^5$	$2.74 \cdot 10^6$	$4.69 \cdot 10^7$	$5.41 \cdot 10^8$

Figure 16: Parallel Algorithm 3.3 average time (nanoseconds) per graph.

Algorithm 4.2 is trickier to adapt for parallel execution. The first difficulty to consider is that recursive frames are dependent on each other: the recursive calls made in Algorithm 4.2 cannot be re-ordered. By storing the mark data and neighbor range ($S[v]$ and $N[v]$) for each of x, y, and z alongside each frame, recursive call order becomes independent for Case 3.2.2 and Case 3.2.3.1, as well as for Case 3.2.1 when $v = x$. Unfortunately, Case 3.2.3.2 and Case 3.2.1 when $v \neq x$ will have one recursive call depend on the reduced list $L[u_2]$ produced by the the other call. A smaller, but no less important detail is that two threads must never use the same face mark or else we risk corruption and data races against the array M.

We can ensure unique marks across threads by using a shared atomic mark counter. Contention can be minimized by having threads reserve marks in large chunks, rather than performing an atomic fetch-add operation for each new mark.

The hurdle of dependent frames can be overcome by maintaining a shared lock-guarded object pool of frames and a stack of frame references. Recursive frames are added to the pool when they appear in the usual course of the algorithm, but are only pushed to the shared stack when their subgraph is ready to be colored. For each vertex $v \in G$, maintain an optional reference to a frame that will become ready when v has been colored. To allow multiple frames to await the coloring of a single vertex, each frame in the object pool will store an "intrusive linked list," that is, an optional reference to the next frame in line. When a frame is added to the pool, and it needs to await a vertex that has another frame waiting on it, the new frame is added to the linked list of the existing frame. When a vertex v is colored, i.e. has its color list reduced to length 1, walk the linked list of frames waiting on v and push a reference to each frame onto the shared stack.

Lock contention can be reduced in a similar manner to Algorithm 3.3 by having each thread maintain a local stack of frames. Unfortuntately, if a frame is created that needs to await the coloring of a vertex, then it must always be added to the shared pool since we do not know which thread will end up coloring that vertex. Moreover, if a frame colors a vertex that has a frame waiting on it, then the lock must be acquired to access the shared pool and stack of frame references.

Benchmarks showed small performance benefits when applying the parallel version of Algorithm 4.2 to plane graphs on the order of 10^4 vertices or more, and significant benefits for graphs with 10^6 vertices or more; see Figure 17. The speed improvements observed were less than those achieved for the parallel Algorithm 3.3, but Algorithm 4.2 requires far more coordination between threads.

Threads	$n = 10^3$	$n = 10^4$	$n = 10^5$	$n = 10^6$	$n = 10^7$
1	$6.43 \cdot 10^4$	$7.28 \cdot 10^5$	$9.84 \cdot 10^6$	$1.83 \cdot 10^8$	$2.11 \cdot 10^9$
2	$8.03 \cdot 10^4$	$6.77 \cdot 10^5$	$8.21 \cdot 10^6$	$1.31 \cdot 10^8$	$1.46 \cdot 10^9$
3	$8.10 \cdot 10^4$	$6.35 \cdot 10^5$	$7.56 \cdot 10^6$	$1.06 \cdot 10^8$	$1.16 \cdot 10^9$
4	$8.92 \cdot 10^4$	$6.31 \cdot 10^5$	$7.50 \cdot 10^6$	$9.56 \cdot 10^7$	$1.03 \cdot 10^9$

Figure 17: Parallel Algorithm 4.2 average time (nanoseconds) per graph.

References

[1] J. Boyer and W. Myrvold, On the cutting edge: simplified $O(n)$ planarity by edge addition, *J. Graph Algorithms Appl.* **8** (2004), 241–273.

[2] I. Broere and C. M. Mynhardt, Generalized colorings of outerplanar and planar graphs, *Graph theory with applications to algorithms and computer science* (Kalamazoo, Mich., 1984), pp. 151–161, Wiley-Intersci. Publ., Wiley, New York, 1985.

[3] A. Bross, *Implementing path coloring algorithms on planar graphs*, Masters Project, University of Alaska, 2017, available from `http://github.com/permutationlock/path_coloring_bgl` and `http://github.com/permutationlock/libavengraph`.

[4] G. G. Chappell and C. Hartman, Path choosability of planar graphs, in preparation.

[5] G. Chartrand and H. V. Kronk, The point-arboricity of planar graphs, *J. London. Math. Soc.* **44** (1969), 612–616.

[6] W. Goddard, Acyclic colorings of planar graphs, *Discrete Math.* **91** (1991), no. 1, 91–94.

[7] C. M. Hartman, *Extremal Problems in Graph Theory*, Ph.D. Thesis, University of Illinois, 1997.

[8] K. S. Poh, On the linear vertex-arboricity of a planar graph, *J. Graph Theory* **14** (1990), no. 1, 73–75.

[9] R. Škrekovski, List improper colourings of planar graphs, *Combin. Probab. Comput.* **8** (1999), no. 3, 293–299.

[10] D. B. West, *Introduction to Graph Theory, 2nd ed.*, Prentice Hall, Upper Saddle River, NJ, 2000.