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

February 20, 2025

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 2-connected 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 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 subgraph of G which has outer face C with the embedding inherited from G.

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 $P_1 = v_1, v_2, \ldots, v_i$ and $P_2 = v_{i+1}, v_{i+2}, \ldots, v_k$.

Suppose C is an induced subgraph of G. Let $u, w \in V(G) - V(C)$ be the vertices such that u, v_k, v_1 and w, v_i, v_{i+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 $P_3 = u_1, u_2, \ldots, u_j$ 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 $C_1 = v_1, v_2, \ldots, v_\ell, u_j, u_{j-1}, \ldots, u_1$ and $C_2 = u_1, u_2, \ldots, u_j, v_{i+1}, v_{i+2}, \ldots, v_k$. 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 $v_r v_s \in E(G) - E(C)$ such that $r \leq i < s$. Let $C_1 = v_1, v_2, \ldots, v_r, v_s, v_{s+1}, \ldots, v_k$ and $C_2 = v_r, v_{r+1}, \ldots, v_s$. 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. $\square$

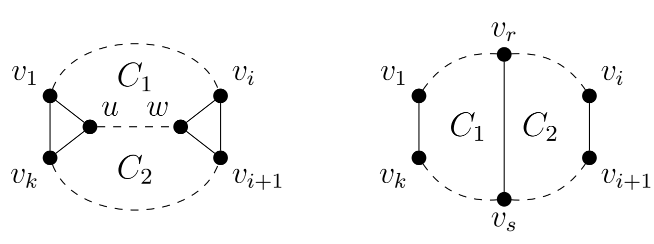

Figure 1: The proof of Lemma 3.1 if C is induced (left) or has a chord (right).

Let G be a triangulated plane graph. We may trivially path 2-color the outer triangle. By Lemma 3.1 this extends to a path 3-coloring of G. If G is an arbitrary planar graph, we may compute an embedding and add edges to produce a triangulated plane graph G'. Theorem 1.1 follows immediately as any path coloring of G' is also a path coloring of G.

A natural way to implement Poh's algorithm is to locate u, and then use a breadth-first search to construct a minimal u, w-path and/or locate a chord edge, see Algorithm 3.2.

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

The concrete input will be Adj, c, and the tuple $(v_1, v_i, v_{i+1}, v_k, v_1v_k)$, where v_1v_k is represented by a reference to v_k's entry in Adj$[v_1]$.

Output. For each vertex $v \in \mathrm{Int}(C) - C$, a color in $\{1, 2, 3\}$ will be assigned to $c[v]$ such that c represents a path 3-coloring of $\mathrm{Int}(C)$ extending the original 2-coloring of C. Moreover, if $v \in \mathrm{Int}(C) - C$ has a neighbor $u \in C$, then $c[v] \neq c[u]$.

Base case. If $v_1 = v_i$ and $v_k = v_{i+1}$, then C is a length 2-path and $\mathrm{Int}(C) = C$ is path 2-colored.

Recursive step. Let u be the vertex immediately counter-clockwise from v_k in Adj$[v_1]$, i.e. the next entry in Adj$[v_1]$ after v_1v_k.

Case 1. Suppose that $c[u] \neq 0$. If $c[u] = c[v_k]$, then $u = v_{k-1} \in P_2$. Make a recursive call with the input $(v_1, v_i, v_{i+1}, u, v_1u)$ to color $\mathrm{Int}(C')$, where $C' = v_1, v_2, \dots, v_{k-1}$. Otherwise, it must be that $c[u] = c[v_1]$ and $u = v_2 \in P_1$. Iterate through Adj$[u]$ to find the entry for v_k. Make a recursive call with the input $(u, v_i, v_{i+1}, v_k, uv_k)$ to color $\mathrm{Int}(C')$, where $C' = v_2, v_3, \dots, v_k$.

Case 2. Suppose that $c[u] = 0$ and therefore $u \in \mathrm{Int}(C) - C$. Perform a breadth-first search in $\mathrm{Int}(C) - C$ starting from u. Such a search may be implemented by only visiting $v \in G$ if $c[v] = 0$. Terminate the search upon finding a vertex w that has a neighbor $v_r \in P_1$ immediately counter-clockwise from a neighbor $v_s \in P_2$, i.e. $c[v_r] = c[v_1]$ and $c[v_s] = c[v_k]$. Backtrack along the search tree from w to find a minimal u, w-path P_3. For each vertex $v \in P_3$, assign $c[v] \leftarrow c_{P_3}$, where $c_{P_3} \in \{1, 2, 3\} - \{c[v_1], c[v_k]\}$.

Make a recursive call with the input (v_1, v_r, w, u, v_1u) to color $\mathrm{Int}(C_1)$, where C_1 is the cycle formed by the paths $v_1, v_2, \dots, v_r$ and P_3, and edges v_1u, v_rw.

Iterate through Adj$[u]$ to find the entry for v_k. Make a recursive call with input (u, w, v_s, v_k, uv_k) to color $\mathrm{Int}(C_2)$, where C_2 is the cycle formed by the paths P_3 and $v_k, v_{k-1}, \dots, v_s$, and edges v_ku, v_sw.

Finally, iterate through Adj$[v_r]$ to find the entry for v_s. Make a recursive call with input $(v_r, v_i, v_{i+1}, v_s, v_rv_s)$ to color $\mathrm{Int}(C_3)$ where $C_3 = v_r, v_{r+1}, \dots, v_s$.

Let G be a triangulated plane graph with n vertices. Let Adj be an adjacency list representation of G, and let $C = v_1, v_2, v_3$ be the outer triangle of G labeled in clockwise order. We may compute a path 3-coloring c of G as follows. Create a length n integer

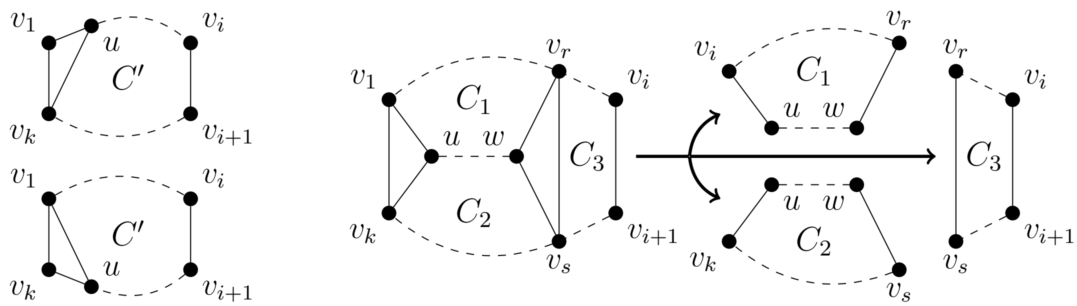

Figure 2: Algorithm 3.2, Case 1 (left) and Case 2 (right).

array c and initialize each entry to zero. Assign $c[v_1] \leftarrow 1$, $c[v_2] \leftarrow 2$, and $c[v_3] \leftarrow 2$. Finally, apply Algorithm 3.2 with input Adj, c, and $(v_1, v_1, v_2, v_3, v_1v_3)$.

Unfortunately, Algorithm 3.2 is not linear. Consider the sequence of "pyramid graphs" $\{A_k\}_{k\in\mathbb{N}}$ depicted in Figure 3. Fix $k \in \mathbb{N}$ and note that $n = n(A_k) = k(k+1)/2 + 3$. Assume that the outer triangle is 2-colored such that the top vertex is assigned a color distinct from the bottom two. At recursive depth i, Algorithm 3.2 will perform a breadth first search of the pyramid-shaped interior subgraph with sides of length $r = k-i$, starting from the vertex in the bottom left corner. The search will hit all $r(r+1)/2$ vertices of the interior pyramid. Therefore the total number of operations performed will be

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

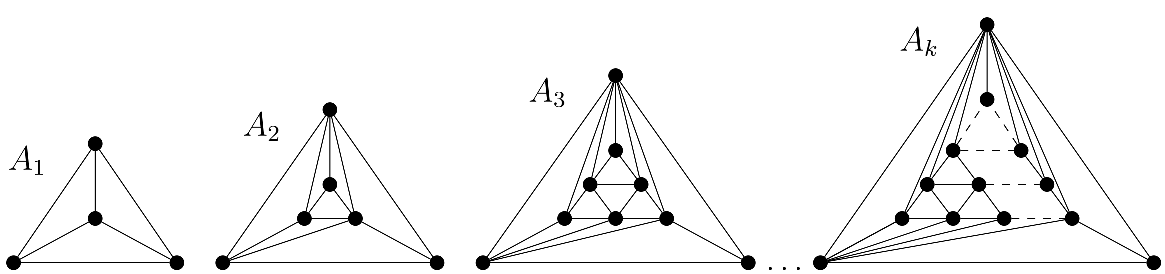

Figure 3: The sequence of "pyramid graphs" on which Algorithm 3.2 performs poorly.

It is possible to improve Algorithm 3.2 by instead performing a bidirectional breadth-first search from u and w simultaneously. However, such an implementation will have the same lower bound time complexity that was computed above. A bidirectional breadth-first search will still visit half of the vertices in each interior pyramid of A_k, and thus only improve the "bad case" performance by a constant factor.

The correctness of Poh's proof, however, relied only on locating locating an induced u, w-path. Algorithm 3.3 describes a linear time implementation of Poh's algorithm that locates and colors an induced u, w-path that is not always of minimal length.

Algorithm 3.3.

Input. Let $C = v_1, v_2, \ldots, v_k$ be an induced cycle in a 2-connected, weakly triangulated plane graph G with n vertices. Let Adj be an adjacency list representation of G. Assume that $\mathrm{Int}(C) \neq C$ and let $u \in \mathrm{Int}(C) - C$ be the unique vertex such that u, v_k, v_1 is a face.

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 $v \in \mathrm{Int}(C) - C$.

If H is a subgraph of G, let $N(H)$ be the set of vertices in G with a neighbor in H. Let S be a length n array of integer marks and let m_{P_1} be an integer such that for each $v \in \mathrm{Int}(C) - C$ we have $S[v] = m_{P_1}$ if and only if $v \in N(P_1)$, i.e. v has a neighbor in P_1.

The concrete input will be the arrays Adj, c and S, along with the ordered pair (u, uv_k) where uv_k is represented by the entry for v_k in $\mathrm{Adj}[u]$.

Output. For each vertex $v \in \mathrm{Int}(C) - C$, a color in $\{1, 2, 3\}$ will be assigned to $c[v]$ such that c represents a path 3-coloring of $\mathrm{Int}(C)$ 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 $c[v] \neq c_{P_2}$.

Outline. We will construct a u, w-path P_3 in $\mathrm{Int}(C) - C$, as in the proof of Lemma 3.1, such that $V(P_3) \subseteq N(P_1)$. Each $v \in P_3$ will be colored $c[v] \leftarrow c_{P_3} \in \{1, 2, 3\} - \{c_{P_1}, c_{P_2}\}$. Each $v \in N(P_3)$ will be marked $S[v] \leftarrow m_{P_3}$, where m_{P_3} is a new unique mark. A recursive call will be made for each proper induced cycle in $\mathrm{Int}(C)$ consisting of vertices in $C + P_3$.

Procedure. Store u_j, the last vertex added to $P_3 = u_1, u_2, \ldots, u_j$, along with u_{j-1}'s entry in $\mathrm{Adj}[u_j]$. Define $u_0 = v_k$ for the purpose of identifying u_{j-1}. Track the last edge $u_r v_s$ between P_3 and P_2 that was encountered, represented by $u_r \in P_3$ and the entry for $v_s \in P_2$ in $\mathrm{Adj}[u_r]$. Initialize $j \leftarrow 1$, $u_1 \leftarrow u$, $u_r v_s \leftarrow u_1 v_k$, and color $c[u] \leftarrow c_{P_3}$.

Step 1. Iterate through $\mathrm{Adj}[u_j]$ in counter-clockwise order, starting with the entry counter-clockwise from u_{j-1}. At each neighbor one of the following cases will be satisfied.

While iterating through $\mathrm{Adj}[u_j]$, keep track of an optional vertex $y \leftarrow$ NULL that will be added to P_3 and become u_{j+1} once all other neighbors of u_j have been handled. Store an optional edge $u_j v_\ell \leftarrow$ NULL, represented by the entry for v_ℓ in $\mathrm{Adj}[u_j]$, indicating the last neighbor of u_j in P_1 that was encountered.

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.

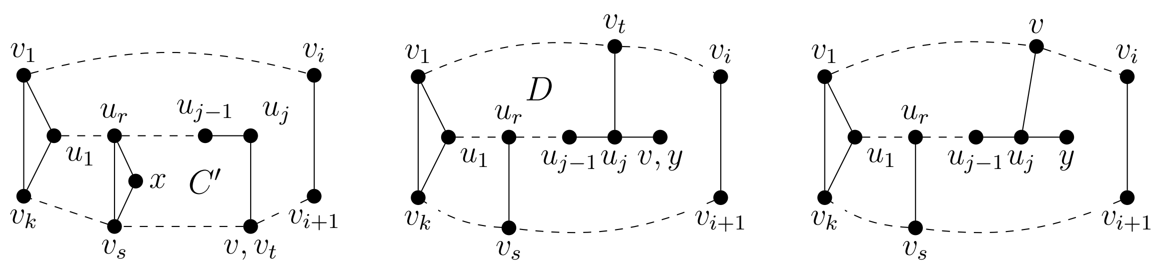

Figure 4: Algorithm 3.3, Case 1.1.2 (left), Case 1.1.3 (middle), and Case 1.1.4 (right).

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

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 $P_1' = u_r, u_{r+1}, \dots, u_j$ and $P_2' = v_s, v_{s+1}, \dots, v_t$ are colored paths that, together with the edges $u_r v_s$ and $u_j v_t$, form an induced cycle C'. Each uncolored vertex in $N(P_3) \cap V(\mathrm{Int}(C'))$ has already been marked 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 input (x, xv_s) to path 3-color $\mathrm{Int}(C')$. Assign $u_r v_s \leftarrow u_j v_t$ to track $u_j v_t$ as the most recent edge between P_3 and P_2.

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 $u_1, u_2, \dots, u_j, y$ 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 $D = v_k, v_1, v_2, \dots, v_t, u_j, u_{j-1}, \dots, u_1$ is a cycle in $\mathrm{Int}(C)$ and $y \notin \mathrm{Int}(D)$. Thus if an edge yu_e exists with $u_e \in P_3 - u_j$, then y's entry in $\mathrm{Adj}[u_e]$ 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 $\mathrm{Adj}[u_e]$, 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 $v = v_i$ and $u_j = w$. To see this, note that the neighbor v' of u_j immediately clockwise from v is adjacent to v, but v' was not assigned to y, nor was $u_j v'$ assigned to $u_j v_\ell$. Therefore $c[v'] = c_{P_2}$, and it must be that $v' = v_{i+1}$ and $v = v_i$.

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.

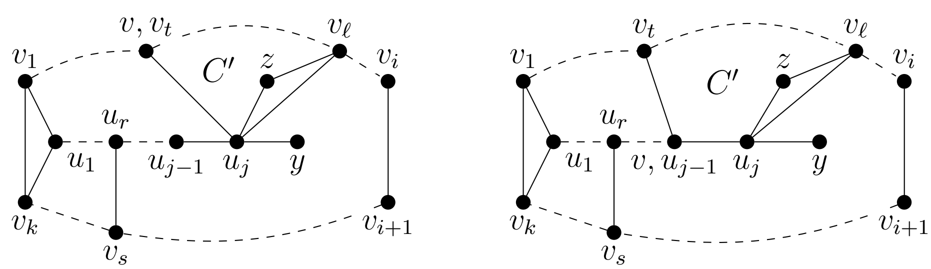

Figure 5: Algorithm 3.3, Case 1.2.2 (left) and Case 1.2.3 (right).

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

Case 1.2.2. Suppose that $c[v] = c_{P_1}$. Then $v = v_t \in P_1$ and $C' = u_j, v_t, v_{t+1}, \dots, v_\ell$ is an induced cycle. Each uncolored vertex in $N(P_3) \cap V(\mathrm{Int}(C'))$ has been marked with m_{P_3}. If $c[z] = 0$, make a recursive call with input (z, zv_ℓ) to path 3-color $\mathrm{Int}(C')$.

Case 1.2.3. Suppose that $c[v] \neq 0$ and $c[v] \neq c_{P_1}$. If $c[v] = c_{P_2}$, then $j = 1$, $v = v_k$, $u_j v_\ell = uv_1$, and there are no uncolored neighbors of u_j between v_ℓ and v.

Suppose that $c[v] = c_{P_3}$. Let t be largest such that $v_t \in P_1$ and $u_{j-1} v_t$ is an edge. Observe that $C' = u_j, u_{j-1}, v_t, v_{t+1}, \dots, v_\ell$ is an induced cycle. Moreover, each uncolored vertex in $N(P_3) \cap V(\mathrm{Int}(C'))$ has been marked with m_{P_3}. If $c[z] = 0$, make a recursive call with input (z, zv_ℓ) to color $\mathrm{Int}(C')$.

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

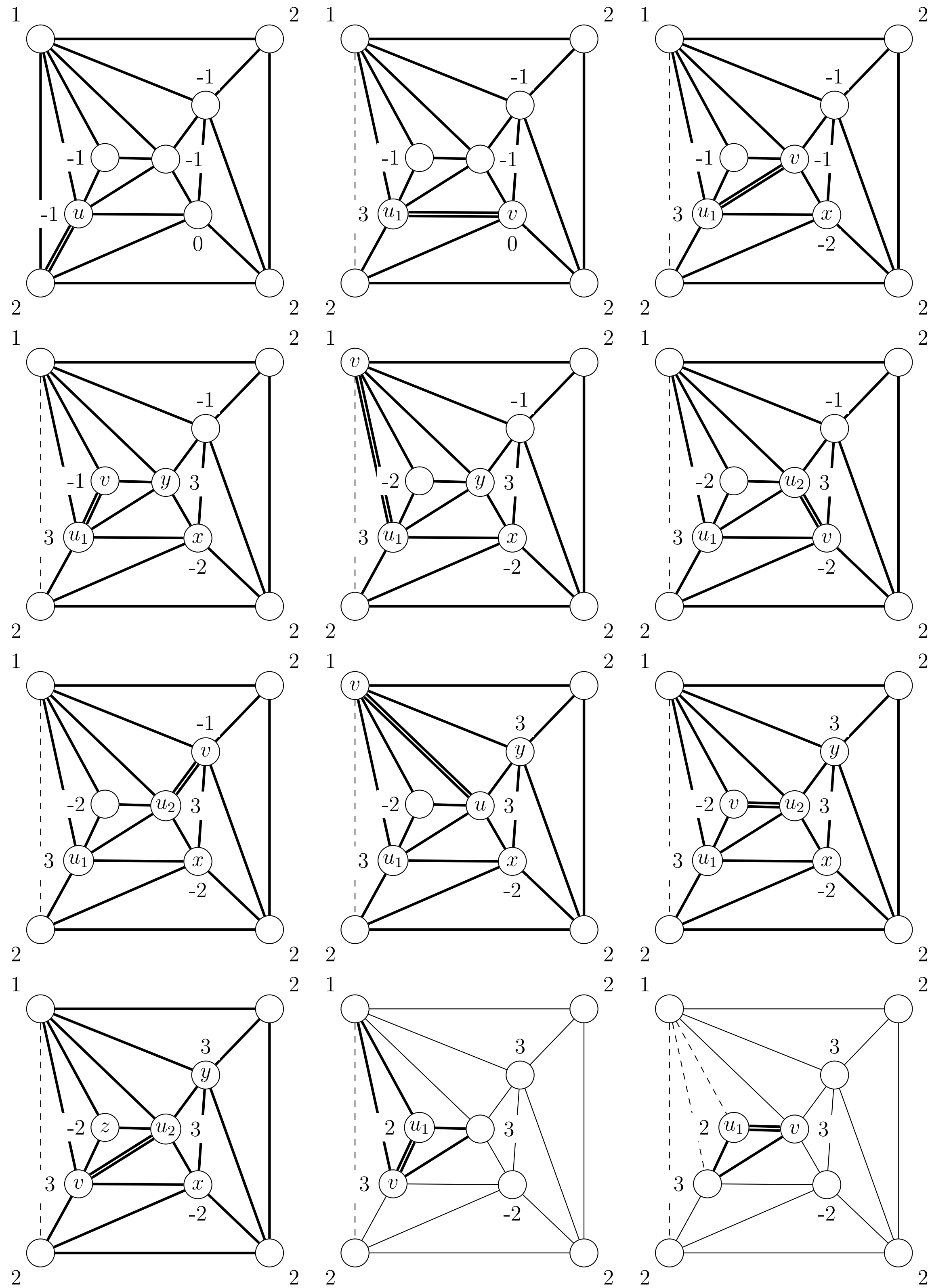

Figure 6: An example of Algorithm 3.3. Each vertex $v \in G$ is labeled with $c[v]$ if $c[v] > 0$, and labeled with $-S[v]$ otherwise.

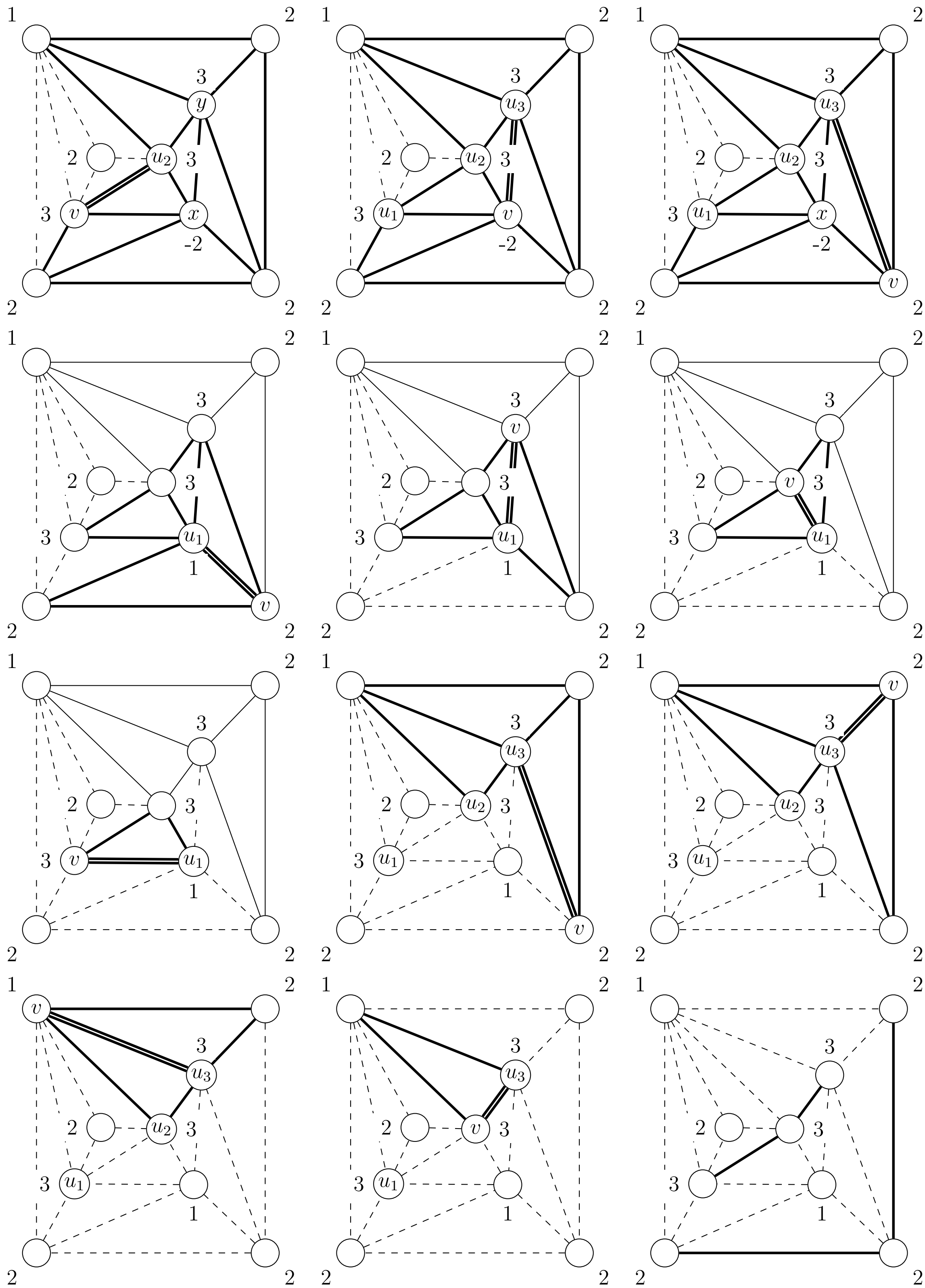

Figure 7: An Algorithm 3.3 example, continued from Figure 6.

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 $c[v_1] \leftarrow 1$, $c[v_2] \leftarrow 2$, and $c[v_3] \leftarrow 2$. Iterate through $\text{Adj}[v_1]$ 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 $\text{Adj}[v_1]$. Note that if $n(G) > 3$, then $u \notin C$. The arrays Adj, c, and S, together with (u, uv_k) form a valid input for Algorithm 3.3.

While executing Algorithm 3.3 we iterate through the adjacency list of each vertex $v \in G$ exactly twice: once to orient $\text{Adj}[v]$ around a particular edge when v is colored, and once to examine each neighbor in $\text{Adj}[v]$ 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.

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 algorithm 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 set 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 assignment of G such that*

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

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 $x = y$. 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 m, the number of edges. 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 clockwise from z around C.

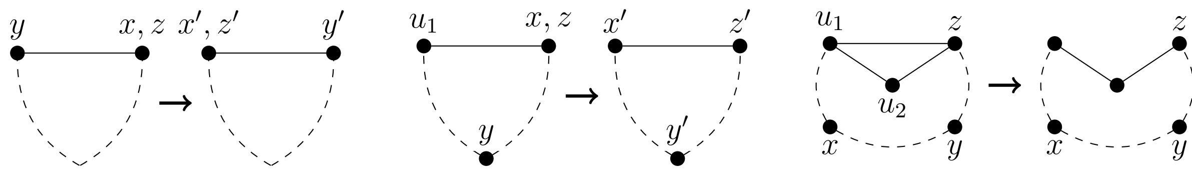

Figure 8: The proof of Lemma 4.1, Case 1 (left), Case 2 (middle), and Case 3.1 (right).

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 $u_1 \in C - C[x, z]$ 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 to path L'-color G with designated vertices $x' = u_1$, $y' = y$, 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.

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 $u_1 z$ 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 $C_1 = C[u_2, u_1] + u_1 u_2$ 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 $\deg_c(u_2) \leq 2$.

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 $x_2 = x$, $y_2 = y$,

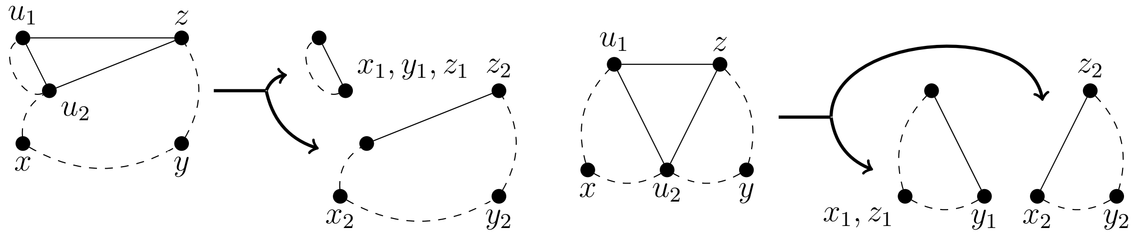

Figure 9: The proof of Lemma 4.1, Case 3.2.1 (left) and Case 3.2.2 (right).

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.

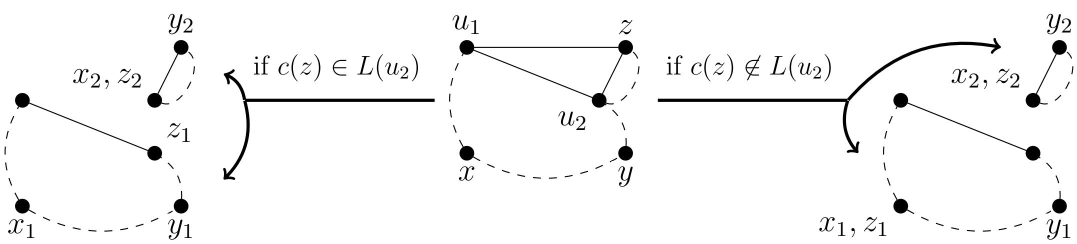

Figure 10: The proof of Lemma 4.1, Case 3.2.3.

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.

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 constructive and may be implemented as a linear time algorithm 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, but e.g. a doubly linked list would also suffice [3].

Algorithm 4.2.
Input. Let C be a cycle or length 2 path in a 2-connected, weakly triangulated plane graph G with n vertices. Let $x, y, z \in C$ be vertices such that $z \in C[x, y]$. Let Adj be an augmented adjacency list representation of G.

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 $v \in \mathrm{Int}(C) - C$ 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 Adj$[v]$ entry for neighbor the immediately counter-clockwise from v around C, and a reference r_2 to the entry for the neighbor immediately clockwise from v around C.

Let S be an array of integers such that if $v \in C$, then $S[v] \neq 0$, and if $v \in \mathrm{Int}(C) - C$, then $S[v] = 0$. Also, if $v \in C$, then $S[v] = S[x]$ if and only if $v \in C[x, z] - z$. Let M be an array of integers such that if $v \in C$, then $M[S[v]] = S[y]$ if and only if $v \in C[z, y]$.

The concrete input will be the arrays Adj, L, N, S, and M, alongside the tuple (x, y, z).

Output. For each $v \in \mathrm{Int}(C)$, all but one color will be removed from the list $L[v]$. The remaining color in each list will represent a path L-coloring of $\mathrm{Int}(C)$ 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 $L[z]$ did not 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 $u_1 = y$. Then $z = x$, since $z \in C[x, y]$. Assign $S[x]$ and $S[y]$ the same new unique mark. Assign $M[S[x]] \leftarrow S[x]$. Make a recursive tail call with the input (y, x, 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. Set $S[u_1]$ to be a new unique mark, assign $M[S[u_1]] \leftarrow S[u_1]$, and make a recursive call with input (u_1, y, z).

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, $u_2 \in \mathrm{Int}(C) - C$. 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 $\mathrm{Adj}[u_2]$, 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 the same input (x, y, z).

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 $\mathrm{Int}(C) - zu_1$.

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 $\mathrm{Adj}[u_2]$ and s_3+1 a reference to the next entry counter-clockwise from z in $\mathrm{Adj}[u_2]$. To represent the subgraph G_1, assign $N[u_2] \leftarrow (s_3 + 1, s_2)$. To represent the G_2, 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 input (x, y, 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 input (u_2, u_2, 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 input (x, u_2, 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 input (u_2, y, 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 from $L[u_2]$ except for the color in $L[z]$. Make a recursive call on G_1 with input (x, y, 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 input (u_2, z, 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 input (x, y, 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 input (u_2, z, u_2).

Let G be a triangulated plane graph with n vertices and $m = 3n-6$ 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

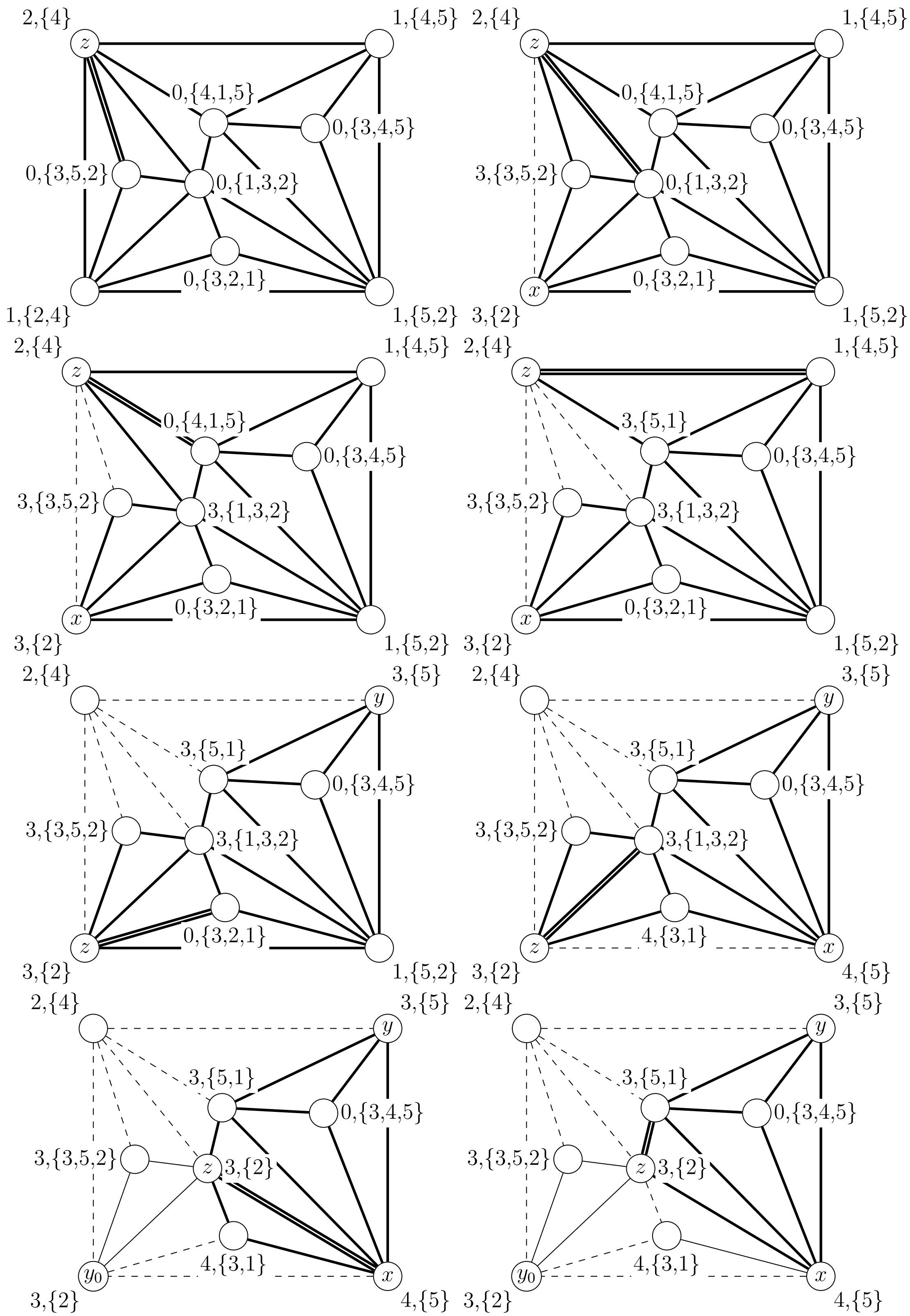

Figure 11: An example of Algorithm 4.2. Each vertex $v \in G$ is labeled with $S[v], L[v]$.

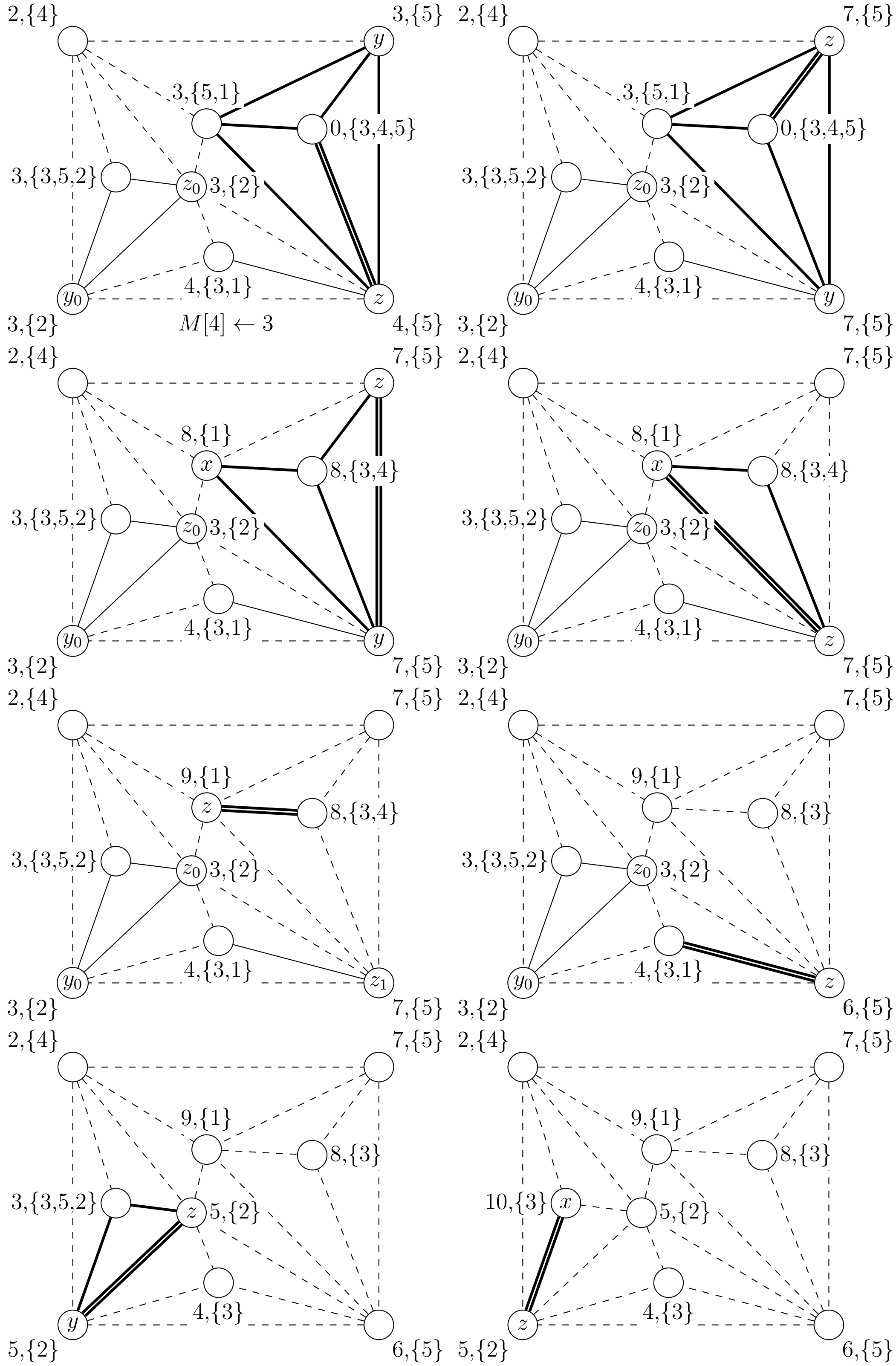

Figure 12: An Algorithm 4.2 example, continued from Figure 11.

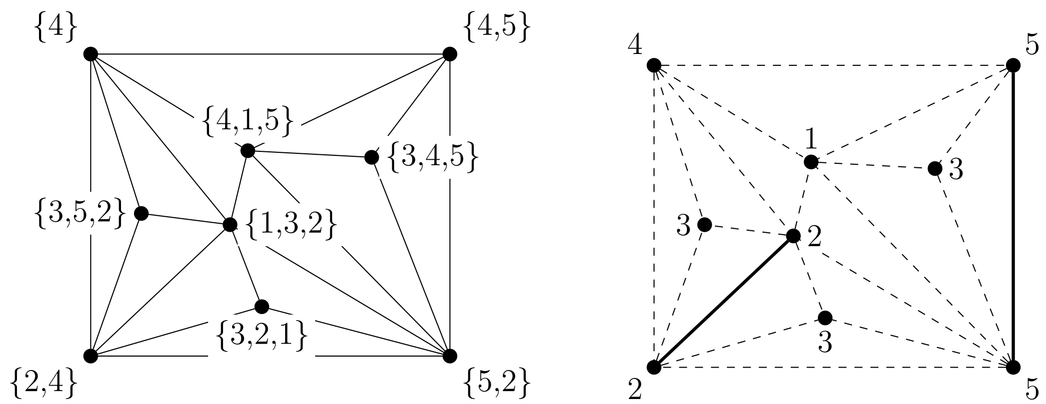

Figure 13: The initial list assignment L (left) and the induced paths of the resulting path L-coloring (right) from the Algorithm 4.2 example shown in Figure 11 and Figure 12.

of integers S, and initialize $S[v] \leftarrow 1$ for each $v \in C$ and $S[v] \leftarrow 0$ for each $v \in G - C$. Pick an arbitrary $u \in C$, assign $S[u] \leftarrow 2$, and remove all but one color from $L[u]$. Create a size $m + 1$ array of integers M and initialize $M[i] \leftarrow i$ for each $i \in \{0, 1, 2\}$. Finally, apply Algorithm 4.2 with the input Adj, L, N, S, M, and (u, u, u). The remaining color in each list in L will 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 11 and Figure 12 for a concrete example of Algorithm 4.2.

5 Experimental performance and parallelism

A C implementation of each algorithm is available [3]. For each i in $3 \ldots 7$, a set of 100 random plane triangulations of order 10^i was generated. Each algorithm was then run on each set of plane graphs, see the timings in Figure 14. A breadth-first search implemented with a ring buffer FIFO queue was run on the same set of graphs as a baseline algorithm that hits every half-edge of a graph exactly once. All binaries were compiled with Clang 19.1.7 and run on a Linux 6.12.9 machine with an x86_64 Intel N100 processor. The benchmark source code is included with the implementation.

The list assignments for Algorithm 4.2 were randomly drawn from a set of eight colors. Experiments drawing from larger color sets showed slightly better performance for Algorithm 4.2, but also resulted in boring colorings inducing mostly short paths.

The performance of Algorithm 4.2 was surprisingly close to Algorithm 3.3 on the set of random graphs, despite the fact that Algorithm 4.2 is more complex and has a larger memory footprint. However, Algorithm 4.2 required an augmented adjacency list representation, which the benchmarks for Algorithm 2.1 showed was relatively expensive to construct from a standard adjacency list graph.

	$n = 10^3$	$n = 10^4$	$n = 10^5$	$n = 10^6$	$n = 10^7$
Breadth-first search	$1.95 \cdot 10^4$	$2.64 \cdot 10^5$	$3.72 \cdot 10^6$	$1.01 \cdot 10^8$	$1.26 \cdot 10^9$
Algorithm 2.1	$4.41 \cdot 10^4$	$6.69 \cdot 10^5$	$1.83 \cdot 10^7$	$4.17 \cdot 10^8$	$4.77 \cdot 10^9$
Algorithm 3.2	$2.31 \cdot 10^5$	$3.48 \cdot 10^6$	$5.29 \cdot 10^7$	$1.17 \cdot 10^9$	$1.82 \cdot 10^{10}$
Algorithm 3.3	$5.13 \cdot 10^4$	$5.96 \cdot 10^5$	$7.12 \cdot 10^6$	$1.30 \cdot 10^8$	$1.50 \cdot 10^9$
Algorithm 4.2	$6.02 \cdot 10^4$	$7.17 \cdot 10^5$	$1.05 \cdot 10^7$	$1.92 \cdot 10^8$	$2.19 \cdot 10^9$

Figure 14: Average time (ns) per triangulated plane graph (sample size 100).

A benchmark was also run to test the asymptotic performance of Algorithm 3.2 and Algorithm 3.3 on the collection of pyramid graphs $\{A_k\}_{k\in\mathbb{N}}$ from Figure 3; see the timings in Figure 15. Values of k were selected such that $n(A_k) \approx 10^i$. Not only did Algorithm 3.2 scale very poorly on pyramids, but Algorithm 3.3 with a "flipped" input scaled almost $1 : 1$ with n.

The representation for each graph A_k was generated such that adjacency lists for vertices along a horizontal line were stored consecutively in memory. The standard input had P_1 consist of the top vertex of the outer triangle, and P_2 the bottom two vertices, while the "flipped" case swapped P_1 and P_2. On the flipped input each recursive call of Algorithm 3.3 colored a horizontal line and the memory access pattern was nearly linear.

	$n \approx 10^3$	$n \approx 10^4$	$n \approx 10^5$	$n \approx 10^6$	$n \approx 10^7$
Algorithm 3.2	$2.69 \cdot 10^5$	$8.63 \cdot 10^6$	$3.59 \cdot 10^8$	$1.27 \cdot 10^{10}$	$6.77 \cdot 10^{11}$
Algorithm 3.3	$2.14 \cdot 10^4$	$2.34 \cdot 10^5$	$2.82 \cdot 10^6$	$4.42 \cdot 10^7$	$1.13 \cdot 10^9$
Algorithm 3.2 (flipped)	$2.66 \cdot 10^5$	$8.44 \cdot 10^6$	$3.79 \cdot 10^8$	$3.06 \cdot 10^{10}$	$1.08 \cdot 10^{12}$
Algorithm 3.3 (flipped)	$1.98 \cdot 10^4$	$1.98 \cdot 10^5$	$1.99 \cdot 10^6$	$2.07 \cdot 10^7$	$2.05 \cdot 10^8$

Figure 15: Average time (ns) per pyramid graph from Figure 3.

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 theoretically possible to maintain a stack of independent recursive frames and have a pool of threads operate on the frames concurrently. The amount of parallelism possible with this method is highly dependent on the structure of the graph. E.g. when Algorithm 3.3 is run on a pyramid graph from Figure 3 there will never be more than one recursive frame on the stack, and thus no opportunity for parallel execution.

In the case of Algorithm 3.3 such parallelism can easily be organized since each frame writes only to vertices interior to the outer face of its associated subgraph. Therefore the implementation 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.

Threads	$n = 10^3$	$n = 10^4$	$n = 10^5$	$n = 10^6$	$n = 10^7$
1	$5.13 \cdot 10^4$	$5.96 \cdot 10^5$	$7.12 \cdot 10^6$	$1.30 \cdot 10^8$	$1.50 \cdot 10^9$
2	$5.12 \cdot 10^4$	$4.26 \cdot 10^5$	$4.30 \cdot 10^6$	$7.51 \cdot 10^7$	$8.53 \cdot 10^8$
3	$4.86 \cdot 10^4$	$3.20 \cdot 10^5$	$3.62 \cdot 10^6$	$6.01 \cdot 10^7$	$6.41 \cdot 10^8$
4	$4.89 \cdot 10^4$	$2.68 \cdot 10^5$	$3.12 \cdot 10^6$	$5.03 \cdot 10^7$	$5.37 \cdot 10^8$

Figure 16: Parallel Algorithm 3.3 average time (ns) per graph (sample size 100).

Benchmarks in Figure 16 showed performance improvements for graphs with 10^4 vertices or more when applying the parallel Algorithm 3.3 to the same set of random plane triangulations used in Figure 14.

Algorithm 4.2 is trickier to adapt to parallel execution. The first difficulty to consider is that the recursive calls made in Algorithm 4.2 relied on a sequential execution order. If we store the mark and neighbor range ($S[v]$ and $N[v]$) for x, y, and z alongside each frame, then the recursive calls in Case 3.2.1 with $u_2 = x$, Case 3.2.2, Case 3.2.3.1, and Case 3.2.3.2 with $u_2 = y$ become order independent. Unfortunately, in Case 3.2.1 with $u_2 \neq x$ and Case 3.2.3.2 with $u_2 \neq y$, the second recursive call will always depend on the reduced color list $L[u_2]$ produced by the first call.

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 will be added to the pool when they appear in the usual course of the algorithm, but will only be pushed to the shared stack when they are ready to be executed. For each vertex $v \in G$, maintain an optional reference to a frame that will become ready when v has been colored, i.e. has had its color list reduced to length 1. To allow for multiple frames to await the coloring of a single vertex, each frame in the 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, walk the linked list of frames waiting on v and push a reference to each frame onto the shared stack.

Another important detail is that the new unique face marks assigned to S must be unique across all threads, or else we risk corruption and data races against the array M. Uniqueness across threads can be ensured by using a shared atomic mark counter. Contention can be minimized by having threads reserve marks in chunks, rather than performing an atomic fetch-add operation for each new mark.

Technically, if we want to avoid even benign data races that don't affect correctness, one more small modification must be made to the algorithm. In Case 3.2.1, if $u_2 = x$, then $S[x]$ must be assigned a new unique mark in the recursive frame handling the G_2 subgraph. This will prevent future writes to $M[S[x]]$ in the G_2 frame from interfering with reads to $M[S[x]]$ that may result from the vertices on the outer face of G_1 marked with $S[x]$. In Figure 11 the mark assigned to the bottom right vertex x of the graph in the step shown at the bottom right of the page would change from 4 to 7 in the frame handling the G_2 block. The change would then ripple forward as the next unique mark would be have to be 8 rather than 7, and so on.

Lock contention can be reduced in a similar manner to Algorithm 3.3 by having each thread maintain a local stack of frames. Unfortunately, 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 another frame waiting on it, then the lock must be acquired.

Threads	$n = 10^3$	$n = 10^4$	$n = 10^5$	$n = 10^6$	$n = 10^7$
1	$6.02 \cdot 10^4$	$7.17 \cdot 10^5$	$1.05 \cdot 10^7$	$1.92 \cdot 10^8$	$2.19 \cdot 10^9$
2	$7.56 \cdot 10^4$	$6.84 \cdot 10^5$	$8.72 \cdot 10^6$	$1.33 \cdot 10^8$	$1.48 \cdot 10^9$
3	$7.51 \cdot 10^4$	$6.15 \cdot 10^5$	$7.63 \cdot 10^6$	$1.08 \cdot 10^8$	$1.14 \cdot 10^9$
4	$8.18 \cdot 10^4$	$6.25 \cdot 10^5$	$7.32 \cdot 10^6$	$9.60 \cdot 10^7$	$9.85 \cdot 10^8$

Figure 17: Parallel Algorithm 4.2 average time (ns) per graph (sample size 100).

Benchmarks in Figure 17 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. Once again, the same set of random plane triangulations was used as in Figure 14.

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, *The Electronic Journal of Combinatorics* **25** (2018), P4.33.

[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.