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

June 1, 2017

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 of both algorithms are available.

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 expiremental benchmarks for implementations of both algorithms [3]. It also discusses how both algorithms 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 clockwise order around v; no additional data structures are required.

We will assume that both forward and backward iteration in adjacency lists is constant time. In terms of plane graphs: given the entry for an edge in a rotation scheme ordered adjacency list, we assume that it is a constant time operation to retrieve entries corresponding to the immediately clockwise and counter-clockwise edges. The provided C implementation uses adjacency arrays, but a doubly linked list would also suffice.

The input for each algorithm will be a 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 an efficient 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 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: the Poh Algorithm

We will first describe Poh's path 3-coloring algorithm. Given a cycle C in a plane graph G we define $\mathrm{Int}(C)$ to be the subgraph of G consisting of C and all vertices and edges interior to C. Equivalently, $\mathrm{Int}(C)$ is the maximal subgraph of G which has outer face C.

Algorithm 3.1 (Poh 1990)**.**
Input: Let G be a weakly triangulated plane graph with outer face a cycle $C = v_1, v_2, \ldots, v_k$. Let c be a 2-coloring of C such that each color class induces a path, $P_1 = v_1, v_2, \ldots, v_\ell$ and $P_2 = v_{l+1}, v_{l+2}, \ldots, v_k$ respectively.

Output: An extension of the 2-coloring c of C to a path 3-coloring of G such that no vertex in $G - C$ receives the same color as a neighbor of that vertex in C.

Step 1: If $G = C$ then G is already path 3-colored and we are done. Otherwise there are two cases to consider.

Case 1.1: Suppose C is an induced subgraph of G. Let $u, w \in V(G) - V(C)$ such that the cycles u, v_1, v_k and $w, v_\ell, v_{\ell+1}$ each exist and are faces of G; note that u and w are uniquely determined, but it may be the case 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 an induced u, w-path in $G - C$. Color each vertex of P_3 with the third color 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 $C_2 = u_1, u_2, \ldots, u_r, v_{\ell+1}, v_{l+2}, \ldots, v_k$.

Case 1.2: 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 $C_2 = v_i, v_{i+1}, \ldots, v_j$.

Step 2: Apply Algorithm 3.1 separately to $\mathrm{Int}(C_1)$ and $\mathrm{Int}(C_2)$. The resulting path 3-colorings agree and extend to a path 3-coloring of G.

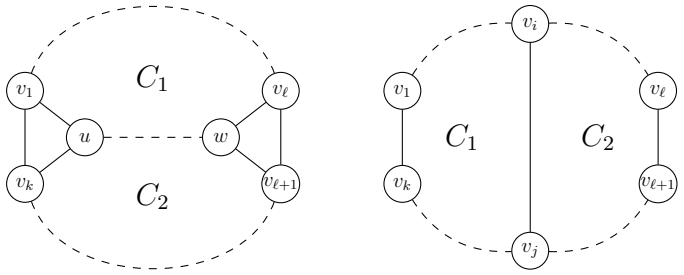

Figure 1: Algorithm 3.1 Case 1.1 (left) and Case 1.2 (right).

Note that the graph G is finite and the recursive step applies the algorithm to two proper subgraphs of G. Therefore Algorithm 3.1 must terminate.

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

In Poh's original proof he picked the induced u, w-path in Case 1.1 to be the shortest u, w-path. Thus a natural way to implement Poh's algorithm is to first locate u and w,

and then use a breadth-first search to to either construct a u, w-path or locate a chord edge if no such path is possible.

Algorithm 3.2.
Input: Let $C = v_1, v_2, \ldots, v_k$ be a cycle in a weakly triangulated plane graph G with adjacency list representation Adj. Let c be an array of colors repreenting a 2-coloring of C such that each color class induces a single 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 all $v \in \text{Int}(C) - C$.

Output: 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.

Step 1: Iterate through $\text{Adj}[v_\ell]$ to locate the vertex u immediately following v_k. Note that since G is triangulated, v_1, u, v_k is a face of G.

Case 1.1: If $u \in C$, then $u = v_{k-1}$, since G is triangulated, and C is not an induced cycle. We then apply Algorithm 3.2 to the cycle $C' = v_1, v_2, \ldots, v_{k-1}$.

Case 1.2: Perform a breadth-first of the maximal subgraph of G with outer face C, starting from the vertex u. Terminate the search upon locating a vertex $w \notin C$ 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 to construct a minimal u, w-path $P_3 = u_1, u_2, \ldots, u_r$. Let $C_1 = v_1, v_2, \ldots, v_i, u_r, u_{r-1}, \ldots, u_1$ and $C_2 = u_1, u_2, \ldots, u_r, v_j, v_{j-1}, \ldots, v_k$. Apply Algorithm 3.2 separately to C_1 and C_2. If $i = l$ and $j = \ell + 1$ then C was an induced cycle and we are done. Otherwise, also apply Algorithm 3.2 to $C_3 = v_i, v_{i+1}, \ldots, v_j$.

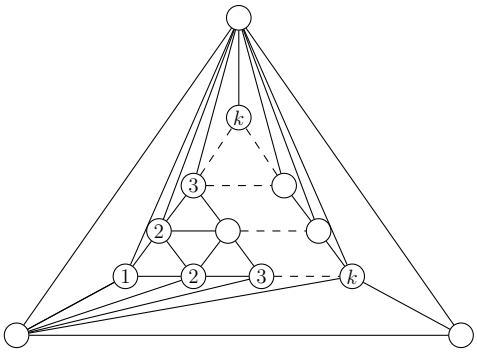

Figure 2: The collection of graphs $\{G_k\}_{k\in\mathbb{N}}$ on which Poh performs poorly.

Unfortunately Algorithm 3.2 is not linear. Consider the family of graphs $\{G_k\}_{k\in\mathbb{N}}$ depicted in Figure 2. 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 iteration i of Poh's algorithm the shortest path through the interior 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 with breadth-first search is $\Omega(n^{3/2})$.

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

Let $N(P_1)$ be the set of vertices with a neighbor in P_1. The strategy will be to construct a path $P_3 = u_1, u_2, \ldots, u_d$ consisting of vertices in $N(P_1)$ such that $C_1 = P_1 \cup P_3 \cup \{u_1v_1, u_dv_l\}$ is a cycle of minimal length.

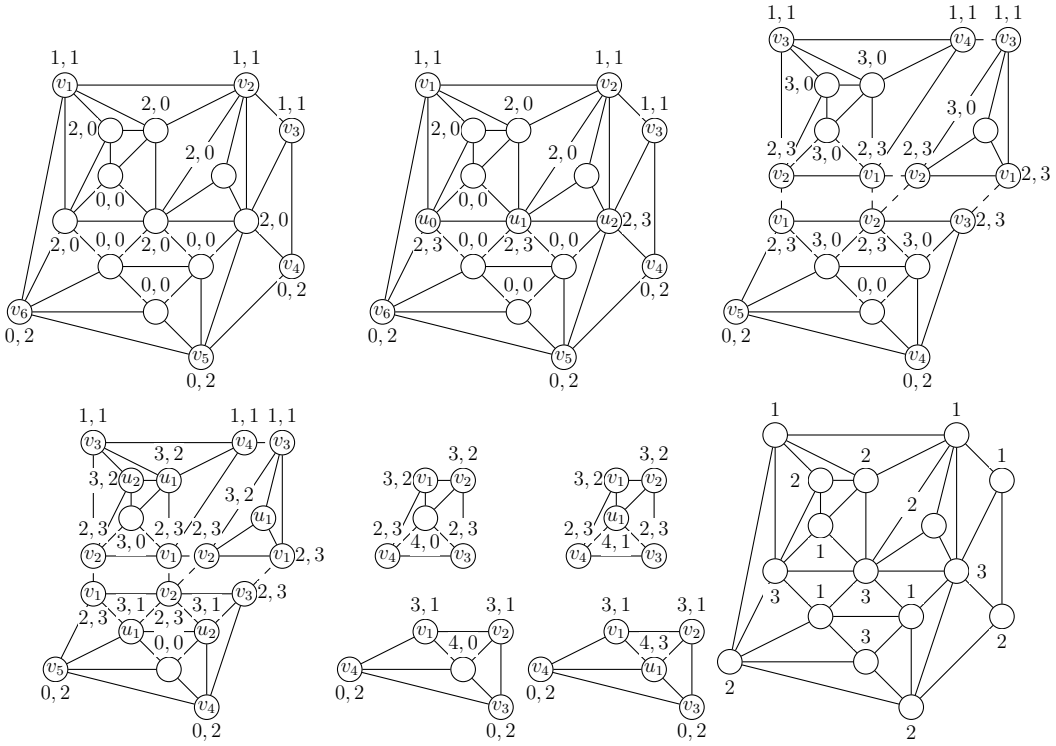

Figure 3: Algorithm 3.3 example, vertices labelled with $S[v], c[v]$.

Algorithm 3.3.
Input: Assume that $C = v_1, v_2, \ldots, v_k$ is an induced cycle in a 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 $P_1 = v_1, v_2, \ldots, v_i$ and $P_2 = v_{i+1}, v_{i+2}, \ldots, v_k$. Let c_{P_1} and c_{P_2} be the colors used on P_1 and P_2, respectively. Assume that $c[v] = 0$ for all $v \in \mathrm{Int}(C) - C$.

The vertex v_1 and the entry for the edge v_1v_k in $\mathrm{Adj}[v_1]$, together with the coloring c, serve as a representation of C, P_1, and P_2.

Finally, let S be an 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 $v \in N(P_1)$.

Output: Each vertex $v \in \mathrm{Int}(C) - C$, a nonzero color 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.

Step 1: Let u be the neighbor of v_1 immediately clockwise from v_k. If $c[u] \neq 0$ then $u \in C$. But C is an induced cycle, so it must be that $u = v_2$ and $\text{Int}(C) = C = v_1, v_2, v_3$ with $k = 3$ and $\text{Int}(C)$ is already colored. From now on we will assume that $c[u] = 0$ and $u \notin C$.

Step 2: Let us color an induced path P_3 in $\text{Int}(C) - C$ with the color c_{P_3} distinct from the two colors used on P_1 and P_2. The new path will start at u and end at the unique verterx $w \in \text{Int}(C) - C$ such that w, v_i, v_{i+1} is a face. Note that the vertex w is guaranteed to exist, but in practice we will not a priori know w before constructing the path P_3.

To start, color the vertex u by assigning $c[u] \leftarrow c_{P_3}$. We will now describe the procedure to color and add a vertex to P_3.

Suppose that we have colored an induced path $P_3 = u_1, u_2, \ldots, u_j$ such that $u_1 = u$ and $S[u_\ell] = m_{P_1}$ for each $\ell \in \{1, 2, \ldots, j\}$. Moreover, for each $\ell < j$ assume that no neighbor y of u_ℓ between $u_{\ell-1}$ and $u_{\ell+1}$ counter-clocwise satisfies $S[y] = m_{P_1}$ (consider $u_0 = v_k$). Iterate through the neighbors of u_j starting from u_{j-1} until encountering a neighbor y with $S[y] = m_{P_1}$ or $c[y] = c_{P_1}$.

If $c[y] = c_{P_1}$ then we claim that $y = v_i$ and $u_j = w$. Note that the neighbor x of u_j immediately clockwise from y is a neighbor of y and therefore satisfies either $c[x] \neq 0$ or $S[x] = m_{P_1}$. But we know that $c[x] \neq c_{P_1}$ and $S[x] \neq m_{P_1}$, so it must be that $c[x] = c_{P_2}$. Since C is an induced cycle and $y \in P_1$, $x \in P_2$, it must be that $y = v_i$ and $x = v_{i+1}$.

If $S[y] = m_{P_1}$, then we claim that y has not already been added to P_3 and that $u_1, u_2, \ldots, u_j, y$ is an induced path. Since $u_j \in N(P)$ there exists $r \in \{1, 2, \ldots, i\}$ such that $u_j v_r$ is an edge of $\text{Int}(C)$. Note that $D = v_1, v_2, \ldots, v_r, u_j, u_{j-1}, \ldots, u_1$ is a cycle in $\text{Int}(C)$ and $y \notin \text{Int}(D)$. Thus if an edge yu_ℓ were to exist, it would have to be between $u_{\ell-1}$ and $u_{\ell+1}$ counter-clockwise in $\text{Adj}[u_\ell]$, a contradiction unless $\ell = j$. Therefore we may assign $u_{j+1} \leftarrow y$, color $c[y] = c_{P_3}$, and continue coloring the path P_3.

Step 3: So far we have colored an induced path $P_3 = u_1, u_2, \ldots, u_j$ such that $C_1 = v_1, v_2, \ldots, v_i, u_j, u_{j-1}, \ldots, u_1$ and $C_2 = u_1, u_2, \ldots, u_j, v_{i+1}, v_{i+2}, \ldots, v_k$ are cycles. It remains to break each cycle down into induced cycles that we may recursively apply the algorithm to.

To path 3-color $\text{Int}(C_2)$ let us iterate through the vertices of P_3 starting from u_1. We will keep track of an edge $u_r v_s$ by storing u_r and the entry for v_s in $\text{Adj}[u_r]$. Initialize $u_r = u_1$ and $v_s = v_k$. Additinally, pick m_{P_3} to be a new unique integer mark to identify vertices in $N(P_3)$.

Let u_ℓ be the current vertex and let $u_r v_s$ be the last edge between P_3 and P_2 that we have encounterd. Iterate through the neighbors of u_ℓ starting from $u_{\ell-1}$ (v_k when $\ell = 1$) and stopping at $u_{\ell+1}$ (v_{i+1} when $\ell = j$). For each neighbor y of u_ℓ we assign $S[y] = m_{P_3}$. Each time we encounter a neighbor y with $c[y] = c_{P_2}$, then we know that $y = v_t \in P_2$ and $C_{r,\ell,s,t} = u_r, u_{r+1}, \ldots, u_\ell, v_t, v_{t+1}, \ldots, v_t$ is an induced cycle. We have also marked all vertices in $\text{Int}(C_{r,\ell,s,t})$ in $N(P_3)$ with the unique mark m_{P_3}. Make a recursive call to path 3-color $\text{Int}(C_{r,\ell,s,t})$ and assign $u_r v_s \leftarrow u_\ell v_t$.

To path 3-color $\text{Int}(C_1)$ we can follow the same procedure we used to color $\text{Int}(C_2)$, but instead iterate through P_3 backwards from u_j. At each vertex $u_\ell \in P_3$ we will iterate

through the neighbors in counter-clockwise order from $u_{\ell+1}$ up to $u_{\ell-1}$.

Note that while executing Algorithm 3.3 we only iterate through the adjacency list of a vertex when it is colored and added to a path. Therefore the algorithm is linear in the number of vertices.

4 Path List Coloring: the Hartman-Škrekovski Algorithm

In this section we will discuss a linear time algorithm for path coloring 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 vertex and edge at a time. The resulting algorithm is similar, but will produce different colorings in some situations.

Let C be the outer face of a 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 path from u to v around C. If $n(G) < 3$ then we define $C(u, v) = C$ if $u \neq v$, and $C(u, u) = u$.

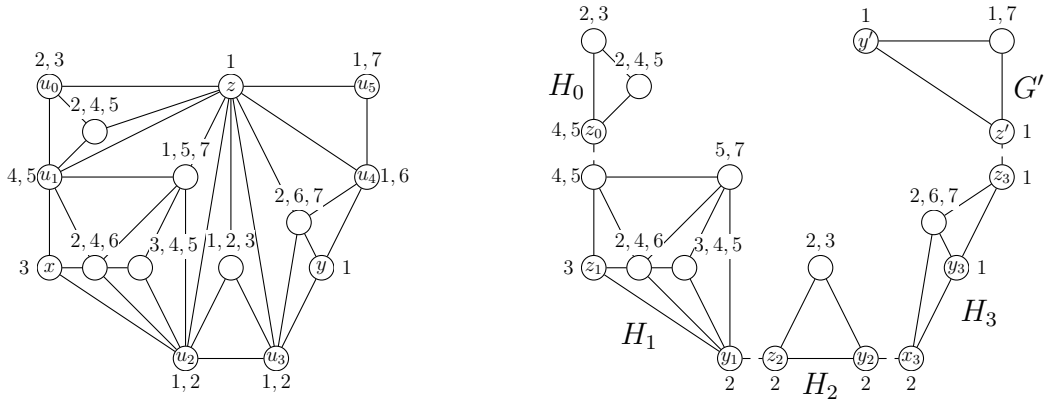

Figure 4: A step of Algorithm 4.1 (if $z_\ell = x_\ell$ and/or $z_\ell = y_\ell$, only z_ℓ is labelled).

Algorithm 4.1 (Hartman 1997, Škrekovski 1999)**.**
Input: Let G be a weakly triangulated plane graph with outer face C. Let $x, y, z \in C$ be vertices (not necessarily distinct) such that $z \in C(x, y)$. Finally, let $L : V(G) \to P_{<\aleph_0}(\mathbb{N})$ be a function assigning a finite list of colors to each vertex such that

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

and if $u \in C(x,z) - z$, then $L(u) \cap L(z) = \emptyset$.

Output: A path 3-coloring $c : V(G) \to \mathbb{N}$ of G such that $c(v) \in L(v)$ for all $v \in G$ and $\deg_c(x) \leq 1$, $\deg_c(y) \leq 1$, and $\deg_c(z) \leq 1$. Furthermore, if $v \in G - C(z,y)$ is a neighbor of z, then $c(z) \neq c(v)$.

Step 1: For each $v \in \{x,y,z\}$ assign $c(v)$ to be a color in $L(v)$. If x,y,z are the only vertices in G, then we are done. If $n(G) = 2$ and $x = y = z$, then the remaining vertex $v \in G - x$ satisfies $|L(v) - c(z)| \geq 2$ and we may choose $c(v)$ to be an arbitrary color in $L(v) - c(z)$. From now on assume that $n(G) > 2$ and the outer face C is a cycle.

Step 2: We will recursively apply the algorithm to path L-choose the 2-connected components of $G - z$. Note that each cut-vertex of $G - z$ must be a neighbor of z in C. Therefore each cut-vertex will exist in at most two 2-connected components and the "graph of 2-connected components" of $G - z$ is a path. Let us label these 2-connected components $H_1, H_2, \ldots, H_{k-1}$ in counter-clockwise order from z around C. Let us similarly label the neighbors of z in counter-clockwise order as $u_1, \ldots, u_k$. Note that the neighgors $u_2, \ldots, u_{k-1}$ are the cut-vertices of $G - z$ and each u_ℓ is contained in $H_{\ell-1}$ and H_ℓ.

Suppose that $u_1 = y$. Then since $z \in C(x,y)$, it must be that $x = z$. We may re-label $x' = y$, $z' = y$, $y' = z$ and apply the algorithm to path L-choose G with the same color degree guarantees on x, y, and z. Thus we may assume that $u_1 \neq y$.

If $z \neq x$ then let H_i be the 2-connected component containing x; if $x = u_i$ is a cut-vertex, and is therfore contained two components H_i and H_{i+1}, we single out the component H_i. If $z = x$, define $i = 1$.

Similarly, if $z \neq y$ define H_j to be the component containing y; if $y = u_{j+1}$ is a cut-vertex, then pick the component H_j ($j > 0$ since $u_1 \neq y$). If $z = y$ then define $j = k$. Observe that $1 \leq i \leq j \leq k$.

For any $a, b \in \{1, 2, \ldots, k\}$ such that $a \leq b$ define

$$G_{a,b} = G\left[\{z\} \cup V(H_a) \cup V(H_{a+1}) \cup \ldots \cup V(H_b)\right].$$

We will L-choose the 2-connected components of $G - z$ in the order

$$H_i, H_{i+1}, \ldots, H_j, H_{i-1}, H_{i-2}, \ldots, H_1$$

to produce a path L-choosing of $G_{1,j}$. Finally, if $j < k$, we will path L-choose $G_{j+1,k}$ as a separate case.

Step 2.1: To start we will path L-choose $G_{i,i}$.

Case 2.1.1: Suppose that $i = j$ and therefore $G_{i,i} = G_{i,j}$.

If $c(z) \in L(u_{i+1})$ then define $L_i(u_{i+1}) = \{c(z)\}$ and $L(v) = L(v) \setminus \{c(z)\}$ for $v \in H_i - u_{i+1}$. Recursively path L_i-choose H_i with $x_i = x$, $y_i = y$, and $z_i = u_{i+1}$. The L_i-choosing of H_i extends to a path L-choosing of $G_{i,j}$ such that $\deg_c(x) \leq 1$, $\deg_c(y) \leq 1$, and $\deg_c(z) = 1$.

Otherwise, $c(z) \notin L(u_{i+1})$. Define $L_i(v) = L(v) \setminus \{c(z)\}$ for all $v \in H_i$ and note that $|L_i(u_{i+1})| \geq 2$. Path L_i-choose H_i with $x_i = x$, $y_i = y$, and $z_i = x$. This L_i-choosing extends to a path L-choosing of $G_{i,j}$ with $\deg_c(x) \leq 1$, $\deg_c(y) \leq 1$, and $\deg_c(z) = 0$.

Case 2.1.2: Suppose that $i \neq j$. Define $L_i(v) = L(v) \setminus \{c(z)\}$ for all $v \in H_i$. Recall that $u_i \in C(x,z) - z$, thus $L(u_i) \cap L(z) = \emptyset$ and $L_i(u_i) = L(u_i)$.

If $z = x$ or if $x = u_i$ is a cut-vertex, then we define $x_i = u_i$, $z_i = u_i$, and $y_i = u_{i+1}$ and apply the algorithm to path L_i-choose H_i. Note that if $z = x$ then $i = 1$ and $x_1 = u_1$ is the vertex immediately counter-clockwise to z around C.

Otherwise x is not a cut-vertex. Note that $u_i \in C(x,z) - z$ and therefore $|L'(u_i)| \geq 2$. Thus we may define $x_i = x$, $z_i = x$, $y_i = u_{i+1}$ and path L_i-choose H_i.

In both cases the path L_i-choosing of H_i extends to a path L-choosing of $G_{i,i}$ such that $\deg_c(u_{i+1}) \leq 1$, $\deg_c(x) \leq 1$, and $\deg_c(z) = 0$.

Step 2.2: Suppose that $\ell \in \{i+1, i+2, \ldots, j-1\}$ and we have computed a path L-choosing of $G_{i,\ell-1}$ such that $\deg_c(u_\ell) \leq 1$, $\deg_c(x) \leq 1$, and $\deg_c(z) = 0$.

Define $L_\ell(u_\ell) = \{c(u_\ell)\}$ and $L_\ell(v) = L(v) \setminus \{c(z)\}$ for all $v \in H_\ell - u_\ell$. Path L_ℓ-choose H_ℓ with $x_\ell = u_\ell$, $z_\ell = u_\ell$, and $y_\ell = u_{\ell+1}$. We are guaranteed that the path L-choosing of $G_{i,\ell-1}$ and the path L_ℓ-choosing of H_ℓ agree at the shared vertex u_ℓ, and that $\deg_c(u_\ell) \leq 1$ and $\deg_c(u_{\ell+1}) \leq 1$ in H_ℓ. Taken together they form a path L-list coloring of $G_{i,\ell}$ with $\deg_c(u_{\ell+1}) \leq 1$, $\deg_c(x) \leq 1$, and $\deg_c(z) = 0$.

Step 2.3: Suppose that $i \neq j$ and we have computed a path L-list-colroing of $G_{i,j-1}$ such that $\deg_c(u_j) = 1$, $\deg_c(x) \leq 1$, and $\deg_c(z) = 0$.

If $z = y$, then define $x_j = u_j$, $z_j = u_j$, $y_j = u_{j+1}$, and $L_j(v) = L(v) \setminus \{c(z)\}$ for all $v \in H_j$. Path L_j-choose H_j and note that it extends to form a path L-choosing of $G_{i,j}$ with $\deg_c(z) = 0$ via the same argument from Step 2.2.

Otherwise $z \neq y$ and u_{j+1} is the furthest clockwise neighbor of z around C in $C(z,y)$. If $c(z) \in L(u_{j+1})$, then define $L_j(u_{j+1}) = \{c(z)\}$ and $L_j(v) = L(v) \setminus \{c(z)\}$ for all $v \in H_j - u_{j+1}$. Path L_j-choose H_j with $x_j = u_j$, $z_j = u_{j+1}$, and $y_j = y$. Again the path L_j-choosing of H_j agrees with the path L-choosing of $G_{i,j-1}$ and both combine to a path L-choosing of $G_{i,j}$ such that $\deg_c(z) = 1$.

If $c(z) \notin L(u_{j+1})$ then define $L_j(v) = L(v) \setminus \{c(z\}$ for all $v \in H_j$. Note that $|L_j(u_{j+1})| \geq 2$, thus we may path L_j-choose H_j with $x_j = u_j$, $z_j = u_j$, and $y_j = y$. The L_j-choosing extends to path L-choosing of $G_{i,j}$ with $\deg_c(z) = 0$.

Step 2.4: We will now color any remaining components H_ℓ with $\ell < i$. Suppose that $\ell \in \{1, 2, .., i-1\}$ and that we have computed a path L-choosing of $G_{\ell+1,j}$ such that $\deg_c(z) \leq 1$. Define $L_\ell(u_{\ell+1}) = \{c(u_{\ell+1})\}$ and $L_\ell(v) = L(v) \setminus \{c(z)\}$ for all $v \in H_\ell - u_{\{}\ell + 1\}$. Path L_ℓ-choose H_ℓ with $x_\ell = u_{\ell+1}$, $y_\ell = u_{\ell+1}$, and $z_\ell = u_{\ell+1}$. Note that $\deg_c(u_{\ell+1}) = 0$ in H_ℓ and therefore the L_ℓ-choosing extends to a path L-choosing of $G_{\ell,j}$ with $\deg_c(z) \leq 1$.

Step 2.5: If $j = k$, then the above steps have computed a path L-choosing of G. Otherwise it remains to extend the L-choosing to $G_{j+1,k}$.

Define $L'(u_{j+1}) = \{c(u_{j+1})\}$, $L'(z) = \{c(z)\}$, and $L'(v) = L(v)$ for $v \in G_{j+1,k} - \{u_{j+1}, z\}$. We then path L'-choose $G' = G_{j+1,k}$ with $x' = u_{j+1}$, $z' = u_{j+1}$, and $y' = z$. Taking C' to be the outer face of G' note that $C'(x', y')$ is the two vertex path x', y'. Therefore no neighbor of x' other than y' will be assigned the color $c(x')$.

By Step 2.4, either $\deg_c(z) = 0$ in $G_{1,j}$ or $c(u_{j+1}) = c(z)$ and $\deg_c(z) = 1$. Because

$$V(G_{1,j}) \cap V(G_{j+1,k}) = \{z, u_{j+1}\} = \{x', y'\},$$

the path L-choosing of $G_{1,j}$ and the path L'-choosing of $G_{j+1,k}$ agree and extend to a path L-choosing of G such that $\deg_c(x) \leq 1$, $\deg_c(y) \leq 1$, and $\deg_c(z) \leq 1$.

Algorithm 4.1 may be implemented for plane graphs represented by rotation scheme ordered augmented adjacency lists in $\mathcal{O}(n)$ time. For each vertex $v \in G$ the implementation will track a list $L[v]$ of up to three colors, a integer location mark $S[v]$, and a pair $N[v] = (r_1, r_2)$ of refernces to neighbor entries in v's augmented adjacency list.

The pair of references to the adjacency list of $v \in C$ will be the entries for the vertices preceeding and succeeding v clockwise around C. The references provide a representation of $\mathrm{Int}(C)$ and a quick way to walk around C.

We can track vertex locations on the outer face by marking each vertex in $C_\ell(u_\ell, u_{\ell+1}) - u_{\ell+1}$ with a new unique mark. In Step 2.1 when $z \neq x$ we must instead mark the vertices in $C_i(u_i, u_{i+1}) - u_{i+1}$ with the mark $S[x]$ already assigned to the vertices in $C(x, u_i) = C_i(x, u_i)$.

In Step 2.1.1 or Step 2.3 when $z \neq y$ and $c(z) \notin L(u_{j+1})$, the segment $C_j(z_j, y) = C_j(x_j, y)$ will consist of vertices in $G - C$ marked with the new mark $S[u_j]$, as well as vertices in $C(u_{j+1}, y)$ marked with $S[y]$. To proceed with the algorithm, we need these two distinct marks to compare equal and represent the single face segment $C(x_j, y)$. To accomplish this re-marking efficiently, we will track a separate array M initialized such that $M[m] = m$ for each integer mark m. Whenever a mark $S[v]$ is compared with the mark $S[y]$, we will instead compare $M[S[v]]$ with $S[y]$. Then to join the segment $C_j(u_j, u_{j+1})$ marked with $S[u_j]$ and the segment $C_j(u_{j+1}, y)$ marked with $S[y]$ we can simply assign $M[S[u_j]] = S[y]$.

Algorithm 4.2.
Input: Let C be a cycle in a 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 $v \in \mathrm{Int}(C) - C$ the list $L[v]$ has length three. Furthermore, for all $v \in C(x, z) - z$ 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 $N[v] = (r_1, r_2)$ has r_1 as a reference to the neighbor the immediately counter-clockwise from v around C, and r_2 as a reference to the neighbor immediately clockwise from v around C.

Let S be an array of integers such that for all $v \in C(x, z) - z$ 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 all vertices $v \in C$ we $S[v] \neq 0$ and for all $v \in \mathrm{Int}(C) - C$ we have $S[v] = 0$. Finally, let M be an array of integers such that for all vertices $v \in C(z, y) - z$ 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 $v \in \mathrm{Int}(C)$ the list $L[v]$ consists of a single color. The coloring of $\mathrm{Int}(C)$ corresponding to the remaining colors will be a path coloring c such that $\deg_c(x) \leq 1$, $\deg_c(y) \leq 1$, and $\deg_c(z) \leq 1$.

Procedure: Let $N[z] = (r_1, r_2)$. Define the vertex u_0 to be the neighbor of z corresponding to r_1, and define v to be the neighbor of z immediately clockwise from u.

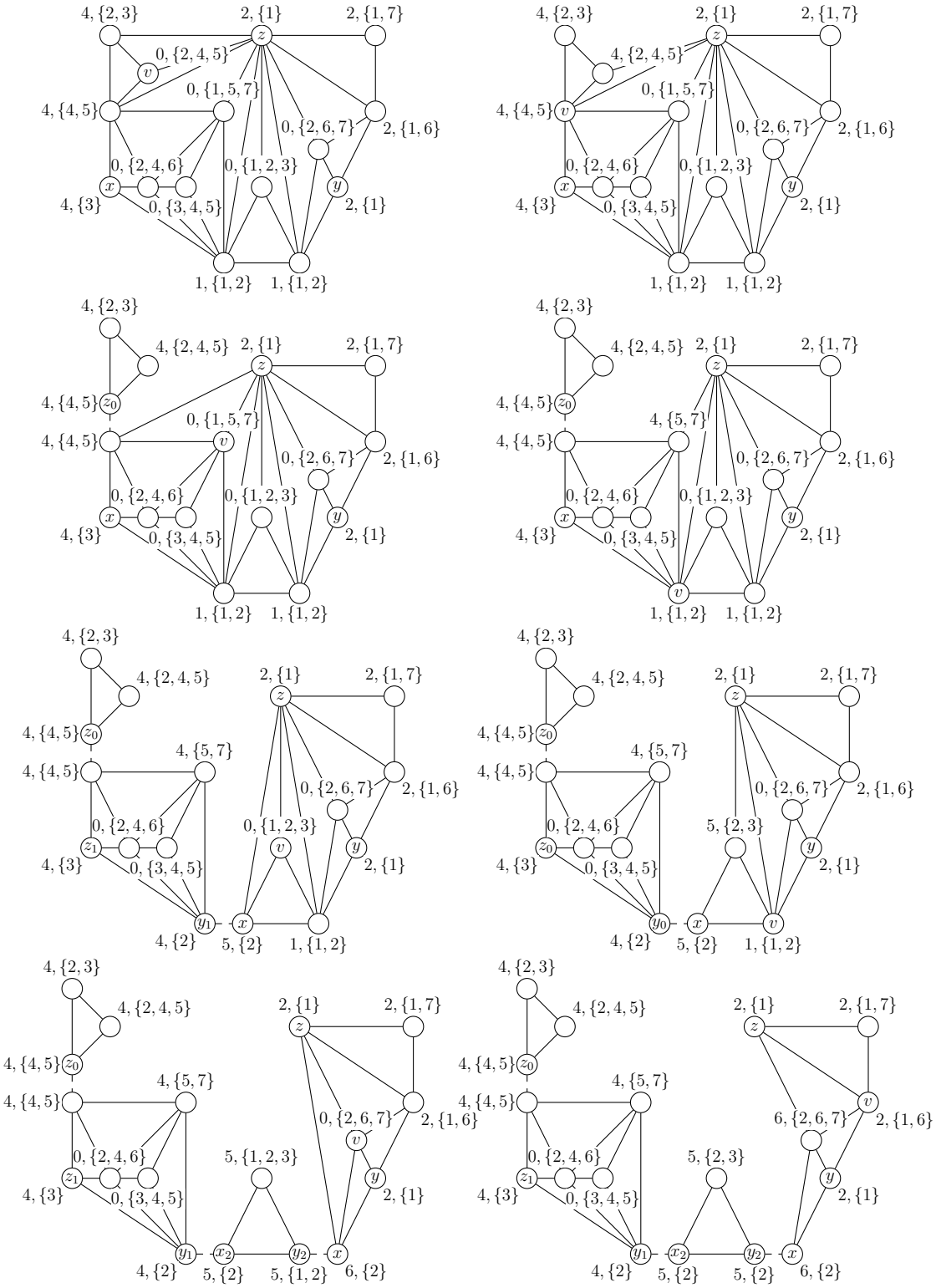

Figure 5: Algorithm 4.2 example, vertices labelled with $S[v], L[v]$.

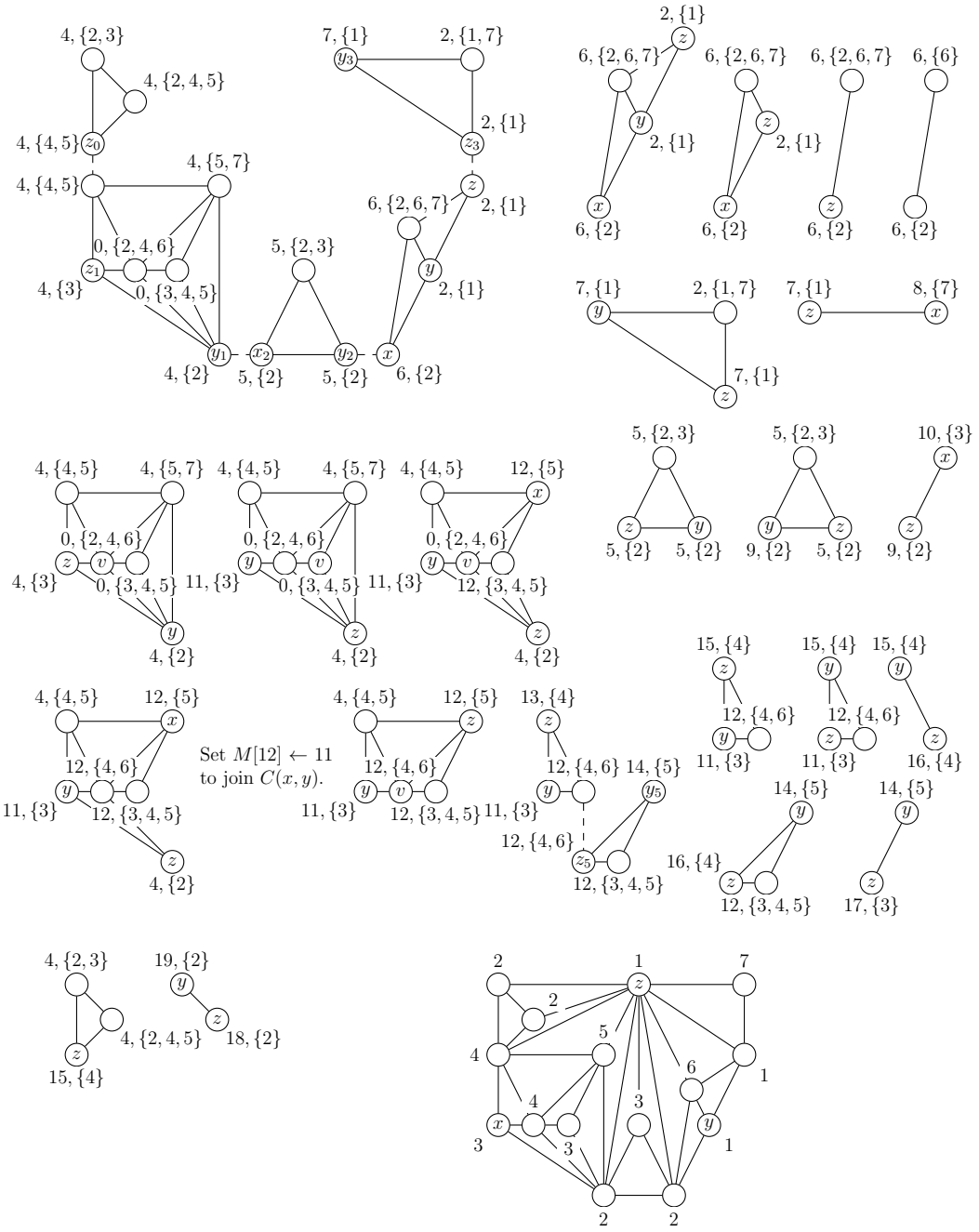

Figure 6: Algorithm 4.2 example continued.

Case 1 (Base Case): If $r_1 = r_2$ then $C = \{z, u_0\}$ is a 2-vertex path. If $u_0 \neq x$ and $u_0 \neq y$, remove color remaining in $L[z]$ from $L[u_0]$. If more than one color still remains in $L[u_0]$, remove arbitrary colors until a single color remains.

Case 2 (Recursive Step): Suppose that $r_1 \neq r_2$, and therefore C is a cycle.

Case 2.1: Suppose that $z = x$ and $u_0 = y$. Make a recursive call after assigning $S[x]$ and $S[y]$ new unique marks and re-assigning $y' \leftarrow z$, $z' \leftarrow u_0$, $x' \leftarrow u_0$.

Case 2.2: Suppose that $z = x$ and $u_0 \neq y$. Make a recursive call after removing the color in $L[z]$ from $L[u_0]$, assigning $x' \leftarrow u_0$, $y' \leftarrow y$, $z' \leftarrow z$, and setting $S[u_0]$ to be a new unique mark.

Case 2.3: Suppose that $z \neq x$ and $S[v] = 0$, that is, $v \in \text{Int}(C)$. Assign $S[v] = S[x]$, remove the color in $L[z]$ from $L[v]$, and remove the edge zu_0 by adjusting $N[u_0]$ and $N[z]$. Make a recursive call and note that the new "u_0" in this call will be the current vertex v.

Case 2.4: Suppose that $z \neq x$ and $S[v] = S[x]$, in other words $v \in C(x, z)$. Then we remove the edge zu_0 and make two recursive calls. The first call is with $x_1 \leftarrow x$, $y_1 \leftarrow y$, and $z_1 \leftarrow z$, but with $N[v]$ split at the edge vz to only include neighbors clockwise from z. The second recursive call is with x_2, y_2, z_2 all assigned equal to v and $N[v]$ split at the edge vz to contain only the neighbors of v counterclockwise from z.

Case 2.5: Suppose that $z \neq x$ and $M[S[v]] = S[y]$, in other words $v \in C(z, y)$. There are two different cases to consider.

Case 2.5.1: Suppose that the color in $L[z]$ is in $L[v]$. We remove all other colors from $L[v]$, remove the edge zu_0, and make two recursive calls. The first call will be made with $x_1 \leftarrow x$, $y_1 \leftarrow y$, and $z_1 \leftarrow v$, splitting $N[v]$ at the edge vz to contain only neighbors counter-clockwise from z. The second recursive call will be made with $x_2 \leftarrow v$, $y_2 \leftarrow z$, and $z_2 \leftarrow v$, splitting $N[v]$ at the edge vz to include z and all neighbors clockwise from z.

Case 2.5.2: Suppose that the color in $L[z]$ is not in $L[v]$. We remove the edge zu_0, assign $M[S[x]] = S[y]$, and make two recursive calls. The first call will be made with $x_1 \leftarrow x$, $y_1 \leftarrow y$, and $z_1 \leftarrow x$, splitting $N[v]$ at the edge vz to contain only neighbors counter-clockwise from z. The second recursive call will be made with $x_1 \leftarrow v$, $y_1 \leftarrow z$, and $z_1 \leftarrow v$, splitting $N[v]$ at the edge vz to include z and all neighbors clockwise from z.

Case 2.6: Suppose that $z \neq x$ and $S[v] \neq 0$, but $S[v] \neq S[x]$ and $M[S[v]] \neq S[y]$, in other words $v \in C(y, x) - x - y$. First remove the color in $L[z]$ from $L[v]$, if it exists, and then remove arbitrary colors until $L[v]$ is of length one. Then remove the edge zu_0 and make two recursive calls. The first call is made with $x_1 \leftarrow x$, $y_1 \leftarrow v$, and $z_1 \leftarrow x$, splitting $N[v]$ at the edge vz to contain only neighbors counter-clockwise from z. The second call is made with $x_2 \leftarrow v$, $y_2 \leftarrow y$, and $z_2 \leftarrow z$, splitting $N[v]$ at the edge vz to contain z and all neighbors clockwise from z.

Every recursive case except for Case 2.1 and Case 2.2 removes an edge from the representation and operates on one or two strictly smaller subgraphs. In Case 2.1 a single recursive call is made with an input that will not itself satisfy Case 2.1. In Case 2.2 a recursive call is made with an input that will not itself satisfy Case 2.1 or Case 2.2. Therefore the number of operations performed is $\mathcal{O}(m) = \mathcal{O}(n)$.

Each operation performed is either integer arithmetic, an array lookup, or a walk through a list of length three or less. As each of these operations is constant time, the

algorithm has a time complexity of $\mathcal{O}(n)$.

5 Experimental performance and parallelism

A C implementation for both algorithms was run on randomly generated triangulated plane graphs up to $10,000,000$ vertices. Benchmark timings for both algorithms are shown below. Timings for a simple breadth-first search (BFS) algorithm are also shown as baseline performance for an algorithm that hits every half-edge. All benchmarks were run on an x86_64 Intel N100 processor.

	$n = 10^3$	$n = 10^4$	$n = 10^5$	$n = 10^6$	$n = 10^7$
BFS	$2.39 \cdot 10^4$	$3.03 \cdot 10^5$	$3.85 \cdot 10^6$	$1.16 \cdot 10^8$	$1.48 \cdot 10^9$
Poh	$7.10 \cdot 10^4$	$8.12 \cdot 10^5$	$1.00 \cdot 10^7$	$2.16 \cdot 10^8$	$2.69 \cdot 10^9$
Poh (3 threads)	$1.25 \cdot 10^5$	$1.17 \cdot 10^6$	$1.20 \cdot 10^7$	$1.47 \cdot 10^8$	$1.69 \cdot 10^9$
Hartman	$1.04 \cdot 10^5$	$1.23 \cdot 10^6$	$2.93 \cdot 10^7$	$6.44 \cdot 10^8$	$7.73 \cdot 10^9$
Hartman (3 threads)	$2.18 \cdot 10^5$	$2.19 \cdot 10^6$	$3.30 \cdot 10^7$	$5.84 \cdot 10^8$	$6.71 \cdot 10^9$

Figure 7: Benchmarks (ns / graph) for Algorithm 3.3 and Algorithm 4.2.

The Poh and Hartman-Škrekovski algorithms color graphs by operating on weakly-triangulated subgraphs that are disjoint except for 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 in parallel.

In the case of Poh's algorithm this can be achieved trivially as each frame writes only to vertices interior to the outer face. The algorithm may use a shared mutex or spin-lock guarded stack of frames from which all threads will push and pop frames. If no frames are available, a thread must idle until either a new frame is pushed to the stack or all other threads are idle.

Benchmarks showed the best performance improvements for Poh's algorithm when using three threads to color plane graphs on the order of 10^6 vertices or more.

The Hartman-Škrekovski algorithm is tricker to adapt to parallel execution. The first diffuculty to consider is that certain recursive frames are dependent on each other: the recursive calls made in Algorithm 4.2 cannot be re-ordered, with the exception of in Case 2.4.1 and Case 2.5. For a concrete example, in Figure 5 the first recursive frame produced cannot start until the vertex z_0 has it's list $L[z_0]$ reduced from $\{4, 5\}$ down to $\{4\}$, which does not happen until much later in Figure 6. A smaller, but no less important detail is that if two threads ever use the same face mark then we risk data races against the array M.

The first hurdle of dependent frames can be solved 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 only pushed to the stack when they are ready to be colored. For each vertex v we maintain an optional

reference to a frame that will become ready when v has been colored. Frames in the object pool will store intrusive linked list references so that multiple frames can await the coloring of a single vertex v. Whenever a vertex v is colored we simply walk the linked list of frames that were waiting on v and push references to those frames onto the stack.

The issue of unique marks can be solved by using an atomic mark counter shared between all threads. Contention can be minimized by having threads reserve marks in large chunks, rather than performing an atomic fetch-add operation for each new mark.

Benchmarks for the parallelized Hartman-Škrekovski algorithm showed the best performance benefits when using three threads to color plane graphs on the order of 10^6 vertices or more. The gains observed were less significant than those observed for the Poh algorithm, but that was to be expected as the recursive frames are often dependent and thread coordination is more complicated.

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`.

[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.