Implementing Path Coloring Algorithms on Planar Graphs

Aven Bross

August 6, 2017

Abstract

A path coloring of a graph partitions its vertex set into color classes such that each class induces a disjoint union of paths. In this project we implement several algorithms to compute path colorings of graphs embedded in the plane.

We present two algorithms to path color plane graphs with 3 colors based on a proof by Poh in 1990. First we describe a naive algorithm that directly follows Poh's procedure, then we give a modified algorithm that runs in linear time.

Independent results of Hartman and Škrekovski describe a procedure that takes a plane graph G and a list of 3 colors for each vertex, and computes a path coloring of G such that each vertex receives a color from its list. We present a linear time implementation based on Hartman and Škrekovski's proofs.

A C++ implementation is provided for all three algorithms, utilizing the Boost Graph Library. Instructions are given on how to use the implementation to construct colorings for plane graphs represented by Boost data structures.

1 Plane Graphs

We will be concerned only with simple plane graphs. Informally, a plane graph is a network drawn in the plane consisting of a set of points, and a set of lines between points such that no lines cross.

Formally a *simple graph* is an ordered pair $G = (V, E)$ consisting of a finite set V of *vertices* and a set E of two element subsets of V known as *edges*. We will refer to the the vertex and edge sets of a graph G by $V(G)$ and $E(G)$, respectively. All graphs in this project are simple.

As shorthand we will denote an edge $\{u, v\} \in E(G)$ simply as uv or vu. Furthermore, if it is clear from context that $v \in V(G)$ is a vertex, or $uv \in E(G)$ an edge, then we will use the notation $v \in G$ or $uv \in G$.

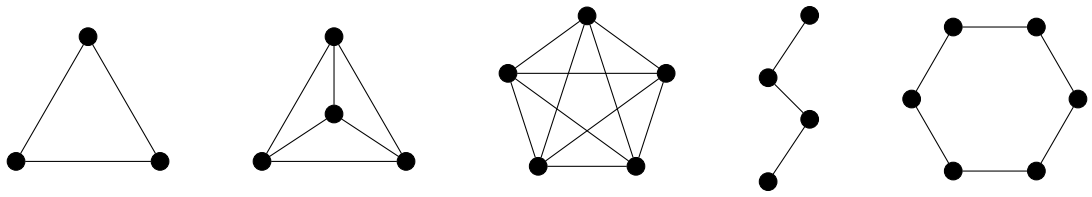

Figure 1.1: Drawings of K_3, K_4, K_5 (nonplanar), an 4-vertex path, and a 6-cycle.

Two vertices $u, v \in V(G)$ are *adjacent* if $uv \in E(G)$. Vertices u and v are known as the *endpoints* of uv. The edge uv is said to be *incident* to the vertices u and v. The vertices in G adjacent to a vertex v are known as the *neighbors* of v. The number of neighbors of a vertex v is its *degree*, denoted $\deg(v)$.

A graph H is a *subgraph* of G if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. If $S \subseteq V(G)$ then the *induced subgraph* of S on G is the subgraph H defined by $V(H) = S$ and $E(H) = \{uv \in E(G) \mid u, v \in S\}$. We say a subgraph H of a graph G is *induced* if it is the induced subgraph of its vertex set on G.

If $v \in V(G)$ then we will use $G - v$ to denote the subgraph obtained by removing v and its incident edges from G. Similarly, if H is a subgraph of a graph G, then we define $G - H$ to be the subgraph obtained by removing from G all vertices in H and all edges incident to a vertex in H.

A n-vertex path consists of the vertices $v_1, v_2, \ldots, v_n$ and the edges v_1v_2, v_2v_3, $\ldots$, $v_{n-1}v_n$. A length n cycle, or n-cycle, consists of an n-vertex path and the additional edge v_1v_n. We will often denote a path or cycle G by simply listing its vertices in order, i.e. $G = v_1v_2 \ldots v_n$.

If a path $P = v_1v_2 \ldots v_n$ is a subgraph of a graph G then we say P is a *v_1v_n-path* in G. A graph G is *connected* if for every $u, v \in V(G)$ there exists a uv-path in G. If any $k - 1$ vertices may be removed from a graph G with G remaining connected, then we say G is *k-connected*.

A *drawing* of a graph maps each vertex to a point in the plane and each edge to a curve connecting its endpoints. A *planar embedding* is a drawing where edge curves intersect only at their endpoints. We say a graph is *planar* if it admits a planar embedding. A planar graph together with a particular planar embedding is called a *plane graph*.

Let G be a plane graph. A *face* of G is a maximal region of the plane not containing any point used in the embedding. The unbounded face is known as the *outer face*. We will always refer to a face by the subgraph of vertices and edges that lie on its border.

For brevity, we have not fully formalized curves, regions, or borders. However, the above definitions and results are fairly standard and may be found in many graph theory texts, for example [27].

Theorem 1.1 (Euler's Formula)**.** *If G is a connected plane graph with n vertices, m*

edges, and f faces, then $n - m + f = 2$.

A simple corollary of Euler's Formula states that if $n \geq 3$, then $m \leq 3n - 6$. A plane graph is said to be *triangulated* if adding any new edge results in a nonplanar graph. Triangulated plane graphs with $n \geq 3$ vertices have exactly $3n - 6$ edges.

A face is said to be a *triangle* if it is a 3-cycle. It is easy to see that all faces in a triangulated plane graph are triangles: if any face has more than three vertices, then we may add an edge curve connecting two face vertices without crossing existing edges. Conversely if all faces in a plane graph are triangles, then it is triangulated.

If a plane graph has triangles for all but one face we shall say it is *weakly triangulated*. We will always assume that the non-triangle face is the outer face. A 2-connected weakly triangulated plane graph has a cycle for its nontriangle face.

Suppose C is a cycle in a weakly triangulated plane graph G. Then the subgraph consisting of C and all interior vertices and edges is denoted $\text{Int}(C)$. If $u, v \in V(C)$ then we denote the uv-path in C running clockwise around the cycle with $C[u, v]$. Finally, if $u, v \in V(C)$ we call any edge $uv \in E(G) \setminus E(C)$ a *chord* of the cycle C.

A *rotation scheme* for a graph is a cyclic ordering of the incident edges around each vertex. Planar embeddings naturally induce a rotation scheme by the order in which edge curves are positioned around each vertex. In fact, with respect to graph algorithms, the induced rotation scheme contains all the useful information of an embedding. Therefore, while we may often visualize plane graphs with drawings, planar embeddings will always be represented solely by their induced rotation scheme.

2 A Brief History of Coloring Plane Graphs

A *k-coloring* of a graph maps each vertex to one of k possible colors. Equivalently, a k-coloring partitions the vertices of a graph into k disjoint sets called *color classes*. A coloring is *proper* if no pair of adjacent vertices receive the same color, or equivalently if the color classes all consist of nonadjacent vertices.

It is clear that not all planar graphs admit a proper 3-coloring: the complete graph on four vertices is planar and requires 4 colors. Whether all planar graphs admit a proper coloring with 4 colors, the Four Color Problem, remained one of the premier open questions in graph theory until it was verified by Appel and Haken in 1976 [2, 3].

A (k, l)-coloring, or a k-coloring with defect l, is a k-coloring such that each vertex has at most l same color neighbors. Generalizations of proper colorings were first introduced in 1968 by Chartrand et al. in [9]. Defective colorings in particular were introduced about simultaneously around 1985 by Cowen et al. [11], Harary et al. [18], and Andrews et al. [1]. It was shown in [11] that all planar graphs admit a $(3, 2)$-coloring.

A *path k-coloring* is a k-coloring such that the induced subgraph of each color class consists of one or more disjoint paths. Note that path k-coloring is equivalent to $(k, 2)$-coloring with the added restriction that path coloring forbids cycles. It was

conjectured by Broere et al. [6] that all planar graphs may be path 3-colored. In 1990 Poh [21] and Goddard [15] independently proved the conjecture. Planar graphs that do not admit a path 3-coloring were described by Chartrand et al. [10], and thus the result is best possible.

Poh's proof is constructive and may easily be adapted to an algorithm for path 3-coloring plane graphs. We describe a naive version of Poh's algorithm, as well as a modified algorithm that runs in $\mathcal{O}(n)$ time.

Let G be a graph. A *list assignment* for G is a map L assigning each vertex $v \in V(G)$ a list of colors. Given a list assignment L an *L-list-coloring* of G, first introduced by Erdös et al. in [13], maps each $v \in V(G)$ to a color in $L(v)$. We say a graph G is *k-choosable* if given any list assignment L such that $\left|L(v)\right| \geq k$ for all $v \in V(G)$, G admits a proper L-list-coloring.

In 1994 Thomassen [25] proved that if G is planar, then G is 5-choosable. A planar graph that is not 4-choosable was described by Voigt [26] in 1993, so Thomassen's result is best possible.

We may equivalently define the properties *(k, l)-choosable* and *path k-choosable*. In 1997 Hartman [19] proved that all planar graphs are path 3-choosable. Hartman's result is best possible since path 3-coloring is a special case of path L-list-coloring with lists of size 3. In 1999 Hull and Eaton [12] and Škrekovski [24] independently proved that if G is a planar graph, then G is $(3, 2)$-choosable.

Hartman's proof provides a constructive procedure to find a path L-list-coloring for a plane graph that has been given a list assignment L with lists of size at least 3. Interestingly, the proofs of Hartman and Škrekovski follow the same coloring algorithm, and thus Škrekovski unknowingly showed the stronger path 3-choosability result. We describe an algorithm based on Hartman and Škrekovski's work and show it runs in $\mathcal{O}(n)$ time.

3 Graph Representations and Time Complexity

Let G be a connected plane graph. Vertices will be represented by integers, that is, we shall assume that $V(G) = \{0, 1, \ldots, n-1\}$. We will always denote the number of vertices in G by n and the number of edges by m.

The input size for each algorithm, given input graph G, will be the number of vertices n. However, since G is a connected plane graph, if $n \geq 3$ then $m \leq 3n - 6$. Thus $\mathcal{O}(m) = \mathcal{O}(n)$. Hence it is equivalent to take the input size to be the number of edges m.

We assume an integer RAM model of computation in which integers require fixed space and integer operations take constant time. The basic operation for all time complexity discussions will therefore be a single memory reference lookup, integer arithmetic operation, or integer comparison.

We will ignore the allocation of memory with respect to time complexity, such as in the creation of arrays or other data structures. The operations required to initialize

elements in a structure are counted. In accordance with these assumptions, inserting or removing an element in a linked list or at the back of an array will require $\mathcal{O}(1)$ time.

Vertex properties will be stored in size n arrays indexed by vertices. Thus accessing or comparing vertex properties shall, in general, be constant time. Colors are assumed to be integers. A coloring of G will thus be represented by an integer vertex property.

For each $v \in V(G)$ we define a linked list called an *adjacency list* containing the neighbors of v ordered according to the rotation scheme of the embedding. The full plane graph G may then represented by a vertex property Adj storing the adjacency list for each vertex. That is, each vertex $v \in V(G)$ has the adjacency list Adj$[v]$.

We will sometimes wish for the ability to quickly find a neighbor u in v's adjacency list directly from v's entry in u's list. To allow this lookup in $\mathcal{O}(1)$ time we will instead define a linked list of pairs Adj$[v]$ for each $v \in V(G)$ called an *augmented adjacency list*. Each node in the list Adj$[v]$ will store a neighboring vertex u as well as a reference to the node for v in Adj$[u]$.

An augmented adjacency list representation of a graph G may be constructed from a standard adjacency list representation in $\mathcal{O}(m)$ time via the following algorithm due to Glenn Chappell [8].

Algorithm 3.1. (Augment Adjacency Lists)
Input: An adjacency list representation Adj of a graph G.
Output: An augmented adjacency list representation Adj$'$ of G with the neighbors of each vertex listed in the same order as in Adj.
Description: We will begin by using Adj to construct an augmented adjacency list representation Adj$'$ of G with the reference portion of each node uninitialized. Next we construct an array Wrk$[v]$ of size $\deg(v)$ for each $v \in V(G)$.

We fill in Wrk as follows. For each v from 0 to $n-1$ let us walk through Adj$'[v]$. At each neighbor u in Adj$'[v]$ let $r_{v,u}$ be the reference to u's position in Adj$'[v]$ and append the pair $(v, r_{v,u})$ to Wrk$[u]$.

After this process finishes each $u \in V(G)$ will have an array Wrk$[u]$ containing the pairs $(v, r_{v,u})$ for each neighbor v, sorted in ascending order by the vertices v.

We will now initialize the references of each node of the augmented adjacency lists. Iterate through the vertices in descending order. Let v be the current vertex. For each $uw \in E(G)$ such that $u < w$ and $v < w$ we shall have initialized the reference for u in Adj$'[w]$ and the reference for w in Adj$'[u]$. We will also have removed the entry $(w, r_{w,u})$ from Wrk$[u]$. It remains to handle edges $uv \in E(G)$ with $v > u$.

For each v from $n-1$ to 0 let us walk through Wrk$[v]$. For i from 1 to $\deg(v)$ take $(u, r_{u,v}) =$ Wrk$[v][i]$. Note that $u < v$ by our assumptions above. Moreover, Wrk$[u]$ contains no entries for neighbors greater than v so $(v, r_{v,u})$ is the last element of Wrk$[u]$. Thus we may lookup $r_{v,u}$ to find u's node in Adj$'[v]$ and initialize the reference with $r_{u,v}$. We may similarly initialize the reference for v's node in Adj$'[u]$. Finally, we remove $(v, r_{v,u})$ from Wrk$[u]$.

Time Complexity: For each edge $uv \in E(G)$, $u < v$, we make a constant number of assignments to Adj′ and Wrk, two reference lookups, and one entry removal from the back of Wrk$[u]$. Therefore the overall complexity of the algorithm is $\mathcal{O}(m)$.

If G is a planar graph without a given embedding we may still construct an adjacency list representation of G, with neighbors simply listed in arbitrary order. There exist numerous $\mathcal{O}(n)$ time algorithms to then reorder the adjacency list representation of G so that it corresponds to a valid planar embedding of G [17, 20, 5, 4]. Additionally, there exist $\mathcal{O}(n)$ time algorithms to add edges to the adjacency list representation in order to connect, 2-connect, and triangulate G, while maintaining planarity [16, 22, 14]. Thus while the algorithms presented will often assume that input graphs are triangulated and plane embedded, arbitrary planar graphs may be modified in linear time to fit these criteria.

Some of the algorithms we discuss, for example *Poh 3-Coloring* (4.1), will describe procedures on abstract graphs. Others, for example *Augment Adjacency Lists* (3.1), will describe algorithms working with computer graph representations. We will provide time complexity analysis for all algorithms working with concrete representations.

Each algorithm presented in this project will allocate some fixed number of vertex properties, independent of the size of the graph. The size of all other data structures constructed will be $\mathcal{O}(n)$ at all points during the operation of each algorithm. Therefore the space complexity of every algorithm is $\mathcal{O}(n)$.

4 Path Coloring and the Poh Algorithm

In this section we detail two algorithms for path 3-coloring plane graphs. We begin by describing the general procedure proposed by Poh [21].

Algorithm 4.1. (Poh 3-Coloring)
Input: A 2-connected weakly triangulated plane graph G with outer cycle $C = v_1 v_2 \dots v_k$ and a 2-coloring of C such that the color classes induce the paths $P = v_1 v_2 \dots v_l$ and $Q = v_k v_{k-1} \dots v_{l+1}$.
Output: An extension of the 2-coloring of C to a path 3-coloring of G such that no vertex in C receives a same color neighbor in $G - C$.
Description: If $G - C$ is empty there are no vertices remaining to color. Otherwise the algorithm proceeds as follows.

Case 1: Suppose there is a chord of C, that is, an edge $v_i v_j \in E(G) \setminus E(C)$ with $i < j$. Since P and Q are induced paths it must be that $v_i \in P$ and $v_j \in Q$. Let C_1 by the cycle consisting of $C[v_j, v_i]$ and the edge $v_i v_j$, and C_2 the cycle consisting of $C[v_i, v_j]$ and the edge $v_i v_j$. Observe that C_1 and C_2 are each 2-colored such that each color class induces a path. Thus we may apply the algorithm to path 3-color Int(C_1)

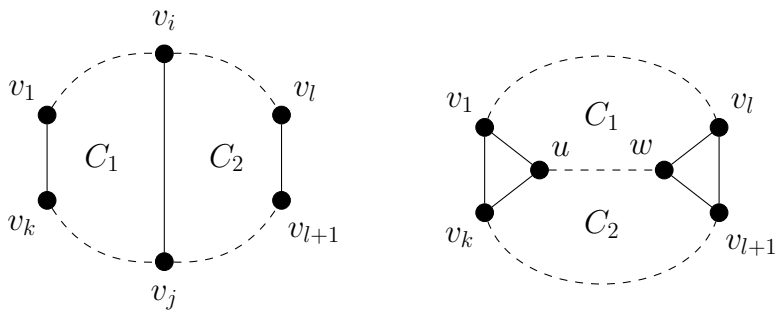

Figure 4.1: The case of a chord (left) and the case no chord exists (right).

and $\text{Int}(C_2)$. Since the subgraphs $\text{Int}(C_1)$ and $\text{Int}(C_2)$ have only the vertices of the chord v_iv_j in common, the combined coloring forms a path 3-coloring of G.

Case 2: Suppose no chords of C exist. Let u be the neighbor of v_k immediately clockwise from v_1 and let w be the neighbor of v_l immediately clockwise from v_{l+1}. That is, $u, w \in \text{Int}(C)$ are the unique, but possibly not distinct vertices such that the cycles uv_1v_k and wv_lv_{l+1} are each faces of G.

Since G is weakly triangulated, $G-C$ is nonempty, and C has no chords, it follows that $G-C$ is connected. Thus there exists a uw-path in $G-C$. Let T be the shortest such path, and note that therefore T is an induced path. Color T with the remaining color not used on P or Q.

Let C_1 be the cycle consisting of P, T, and the edges v_1u and v_lw. Similarly, let C_2 be the cycle consisting of T, Q, and the edges v_ku and $v_{l+1}w$. Then we may apply the algorithm to path 3-color $\text{Int}(C_1)$ and $\text{Int}(C_2)$. Since $\text{Int}(C_1)$ and $\text{Int}(C_2)$ have only the vertices of the path T in common, the combined coloring forms a path 3-coloring of G.

Given any plane graph G we may add edges until it is triangulated. Observe that any path coloring of G with the additional edges is also a path coloring of the original G. Therefore by path 2-coloring the outer triangle we may apply Poh's algorithm to path 3-color G. This observation yields the following result.

Theorem 4.1 (Poh [21] and Goddard [15])**.** *All planar graphs are path* 3*-colorable.*

The Poh Algorithm with Breadth First Search

In order to implement Poh's algorithm with adjacency lists there are two main obstacles. First, we must have a method to efficiently represent colored paths, as we will be recursively constructing paths and dividing the graph along them. Second, we will need an efficient algorithm to locate the chords of C and the uw-path.

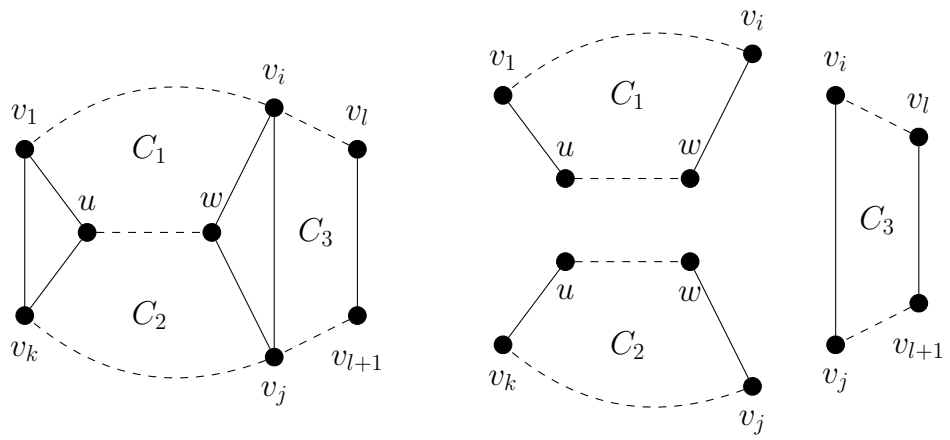

Figure 4.2: Dividing G along the edge v_iv_j and the uw-path.

Let G be a 2-connected weakly triangulated plane graph with an adjacency list representation. Each call of the algorithm will be provided with a cycle C in G and produce a path 3-coloring of $\text{Int}(C)$ according to the specifications of Poh's algorithm.

To represent induced paths in G we will simply use the color vertex property. Suppose $P = v_1v_2 \dots v_k$ is an induced path in G, and each vertex of P has been assigned c_P. Assume that the coloring constructed so far is a path coloring. If $v_i \in P$ then a neighbor u of v_i will have the color c_P if and only if $u \in P$, that is, $u = v_{i-1}$ or $u = v_{i+1}$. Therefore we may represent the entire path by storing just the vertices v_1 and v_k.

We will now describe the first version of Poh's algorithm on adjacency list graphs, using a breadth first search to find induced paths and chords.

Algorithm 4.2. (Poh – BFS)
Assumptions: Suppose $P = v_1v_2 \dots v_l$ and $Q = v_kv_{k-1} \dots v_{l+1}$ are induced paths such that $C = v_1v_2 \dots v_k$ is a cycle. Additionally, assume that each path has been colored with a distinct color.
Input: The paths P and Q, each represented by their endpoints as described above.
Output: An extension of the 2-coloring of C to a path 3-coloring of $\text{Int}(C)$ such that no vertex in C receives a same color neighbor in $\text{Int}(C) - C$.
Description: Locate the position of v_k in $\text{Adj}[v_1]$. Proceeding one vertex further in $\text{Adj}[v_1]$ gives us a vertex u such that the cycle uv_1v_k is a triangle.

Case 1: Suppose $u \in C$. If u is in P, i.e. $u = v_2$, we apply the algorithm to the paths $P - u$ and Q. Similarly if w is in Q we apply the algorithm to P and $Q - u$. In either case, if the two remaining paths each consist of single vertex then there are no remaining uncolored vertices and we terminate the algorithm.

Case 2: Suppose $u \notin C$. Perform a breadth first search from u in $\text{Int}(C) - C$,

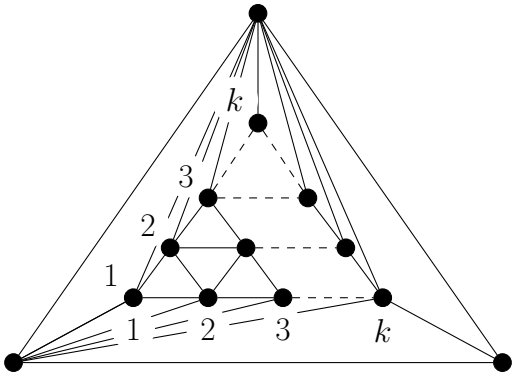

Figure 4.3: The collection of graphs $\{G_k\}_{k\in\mathbb{N}}$ on which Poh performs poorly.

that is, ignoring vertices in C. Terminate the search when we reach a vertex w with neighbors $v_i \in P$ and $v_j \in Q$ such that v_i is immediately past v_j in Adj[w]. Such a vertex must exist by the same argument as in *Poh 3-Coloring* (4.1). Backtracking from w along the breadth first search and coloring vertices produces an induced uw-path T, colored with the remaining color not used on P or Q.

Define the paths $P_1 = v_1 v_2 \dots v_i$, $P_2 = v_i p_{i+1} \dots v_l$, $Q_1 = v_k v_{k-1} \dots v_j$, and $Q_2 = v_j q_{j-1} \dots v_{l+1}$. Observe that we have a cycle C_1 consisting of P_1, T, and the edges $v_1 u$ and $v_i w$. Similarly we have a cycle C_2 consisting of T, Q_1, and the edges $v_k u$ and $v_j w$. We apply the algorithm to P_1 and T to color Int(C_1) and similarly to T and Q_1 to color Int(C_2).

If $i = l$ and $j = l + 1$ we are done. Otherwise, we have the cycle C_3 consisting of P_2, Q_2 and the edges $v_i v_j$ and $v_l v_{l+1}$, and we may apply the algorithm to color Int(C_3).

Note that the combined coloring forms a path 3-coloring of Int(C) by the same argument as in *Poh 3-Coloring* (4.1).

Complexity: In the first step we rotate through Adj[v_1] to find v_k and get an orientation within the graph. This orientation must be performed at most once for each vertex, for a total of at most $\sum_{v=0}^{n-1} \deg(v) = 2m$ operations.

In the next step we perform a breadth first search from the vertex u. A breadth first search requires at most m lookups. Moreover, the vertex will u will be colored following the search. Thus we perform at most one breadth first search from each vertex, requiring at most nm operations. Therefore the complexity of the algorithm is $\mathcal{O}(2m + nm) = \mathcal{O}(n^2)$.

We define the collection of graphs $\{G_k\}_{k\in\mathbb{N}}$, depicted in Figure 4.3. Let us fix $k \in \mathbb{N}$ and note that G_k has $n = \frac{k^2+k}{2} + 3$ vertices.

Suppose we apply *Poh BFS* (4.2) to path 3-color G_k. Let the initial 2-coloring of the outer triangle of G_k assign the top vertex a color distinct from the bottom two. The $(k-i+1)$th recursive call will perform a breadth first search visiting each vertex

in a subgraph of size $\frac{i^2+i}{2}$, hence requiring $\Theta(\frac{i^2+i}{2})$ operations. The total number of operations required to path 3-color G_k is therefore

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

Thus the time complexity of the algorithm is $\Omega(n^{3/2})$. In particular the algorithm is not linear.

The Poh Algorithm in Linear Time

Poh's proof requests that we find the shortest uv-path in $\text{Int}(C)$. Therefore Poh's algorithm as written does not appear to admit a linear time algorithm.

However, the correctness of Poh's algorithm does not require that T be the shortest uw-path, only that T be an induced uw-path. We will show that Poh's algorithm becomes linear if we instead construct an induced uw-path consisting of vertices in $G - C$ with at least one neighbor in the colored path P.

Algorithm 4.3. (Poh – Path Walk)
Assumptions: Assume that $P = v_1 v_2 \dots v_l$ and $Q = v_k v_{k-1} \dots v_{l+1}$ are induced paths, each colored with a distinct color, such that $C = v_1 v_2 \dots v_k$ is a cycle.
Input: The path P, represented by its endpoints, and the color of the path Q.
Output: An extension of the 2-coloring of C to a path 3-coloring of $\text{Int}(C)$ such that no vertex in C receives a same color neighbor in $\text{Int}(C) - C$.
Description: We will iterate through the vertices of P until we find a chord. All interior vertices visited will be marked to indicate they have a neighbor in P. For each i from 1 to l let us walk through $\text{Adj}[v_i]$ from v_{i-1} to v_{i+1}, not including v_{i-1} and v_{i+1}.

Let v be the current neighbor. If $v = v_j \in Q$ then $v_i v_j$ is a chord of C and we stop. Otherwise $v \in \text{Int}(C) - C$ with the neighbor $v_i \in P$ and we mark it.

Define the cycles C_1 and C_2 as usual by dividing C along the chord $v_i v_j$. Note that C_1 is chordless as P and Q are induced paths and $v_i v_j$ is the first chord of C encountered. Apply *Path Walk* (4.3) to path 3-color $\text{Int}(C_2)$. It remains to color $\text{Int}(C_1)$.

If we never encountered a vertex in $\text{Int}(C) - C$ during our walk through the neighbors of $v_1, \dots, v_i$, then $\text{Int}(C_1) - C_1$ is empty and thus $\text{Int}(C_1)$ is already colored. Otherwise let u be the first such neighbor encountered. Note that u is the unique vertex such that uv_1v_l is a face. We may therefore apply *Path Trace* (4.4) to path 3-color $\text{Int}(C_1)$.

The combined coloring is a path 3-coloring of $\text{Int}(C)$ by the same argument as in *Poh 3-Coloring* (4.1).

Complexity: See *Path Trace* (4.4).

Algorithm 4.4. (Poh – Path Trace)
Assumptions: Let $P = v_1v_2 \dots v_l$ and $Q = v_kv_{k-1} \dots v_{l+1}$ be induced paths, each colored with a distinct color, such that $C = v_1v_2 \dots v_k$ is a chordless cycle. In addition, suppose $\text{Int}(C) - C$ is nonempty and all vertices in $\text{Int}(C) - C$ with at least one neighbor in P have been marked.
Input: The vertex $u \in \text{Int}(C) - C$ such that the cycle uv_1v_k is a face, as well as the respective colors of the paths P and Q.
Output: An extension of the 2-coloring of C to a path 3-coloring of $\text{Int}(C)$ such that no vertex in C receives the same color as a neighbor of that vertex in $\text{Int}(C) - C$.
Description: Initialize T as the path consisting of the single vertex u, coloring u with the remaining color. We will recursively add vertices to T until we reach the unique vertex w such that wv_lv_{l+1} is a face.

Suppose we have constructed the induced path $T = t_1t_2 \dots t_d$, such that $t_1 = u$ and each t_i has at least one neighbor in P. Iterate through $\text{Adj}[t_d]$, starting from t_{d-1}. Let v be the current neighbor. If v has a neighbor in P we color v, assign $T = t_1 \dots t_dv$, and repeat the process with v as the new end vertex. If $v \in P$ it must be that $t_d = w$ and we are finished constructing T. Otherwise ignore v and move to the next neighbor.

Note that the process above must terminate since each vertex in T has at least one neighbor in P.

Let $T = t_1t_2 \dots t_d$ be the path constructed above. Suppose t_it_j is an edge with $t_i, t_j \in T$, $i < j$. If $i = 1$ let us define the vertex t_0 to be v_1. Since each vertex in T has a neighbor in P, by planarity it must be that t_j is between t_{i-1} and t_{i+1} counterclockwise in $\text{Adj}[t_i]$. But by the construction of T, t_{i+1} is the first neighbor of t_i counterclockwise from t_{i-1}. Thus $j = i + 1$.

Therefore the only edges between vertices in T are the edges $t_1t_2, \dots, t_{d-1}t_d$. So T is an induced path.

We apply *Path Walk* (4.3) to path 3-color $\text{Int}(C_2)$. It remains to color any uncolored vertices in $\text{Int}(C_1)$.

All vertices in T have at least one neighbor in P. Therefore any uncolored vertex in $\text{Int}(C_1)$ must lie in a path 2-colored chordless cycle of the form $v_iv_{i+1} \dots v_jt_{p+1}t_p$ or $v_iv_{i+1} \dots v_jt_p$. We will use the following procedure to locate all such cycles that contain uncolored vertices and color them using *Path Trace* (4.4).

For each p from 1 to d let us iterate through $\text{Adj}[t_p]$, starting with t_{p+1}. In the case $p = d$ we will define $t_{p+1} = v_l$. Suppose we visit a neighbor $y \in \text{Int}(C_1) - C_1$ followed counterclockwise by a neighbor $v_i \in P$. There are two possible cases.

Case 1: Suppose none of the neighbors of t_p between t_{p+1} and v_i counterclockwise are in P. Let j be the smallest integer such that $t_{p+1}v_j$ is an edge, noting again that $i < j$ by planarity. Note that since P is an induced path the cycle $C_y = t_pv_iv_{i+1} \dots v_jt_{p+1}$ is chordless by the our selection of v_j. Thus we may apply *Path*

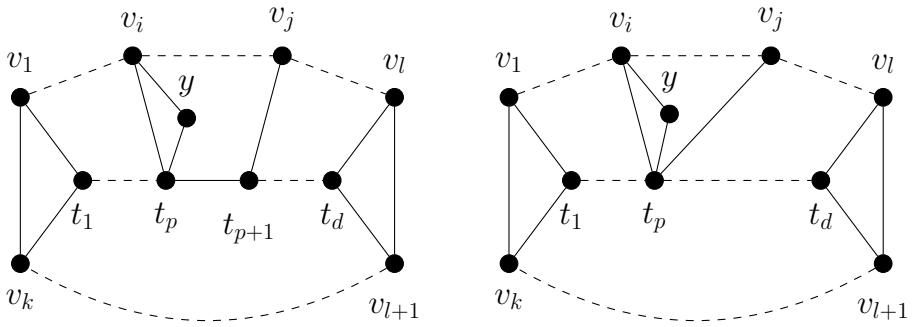

Figure 4.4: Coloring vertices above T in *Path Trace* (4.4) case 1 (left), case 2 (right).

Trace (4.4) to color $\text{Int}(C_y)$ with the vertex y forming the face yt_pv_i.

Case 2: Suppose we have previously visited a neighbor of t_p in P, and let $v_j \in P$ be the most recent such neighbor visited. Note that by planarity it must be that $i < j$. Thus the cycle $C_y = v_iv_{i+1} \dots v_jt_p$ is chordless since P is an induced path. We may therefore apply *Path Trace* (4.4) to color $\text{Int}(C_y)$ with the vertex y forming the face yt_pv_i.

Complexity: Let G be a plane graph that has been colored with Poh. Let P be a path induced by the path 3-coloring of G. Note that each vertex is in exactly one such path.

Let $v \in P$. We iterated through $\text{Adj}[v]$ exactly once during *Path Walk* (4.3). We iterated through $\text{Adj}[v]$ at most twice during *Path Trace* (4.4): once to locate the starting neighbor, and once to find the next vertex to add to the path and find uncolored vertices above T.

Thus the time complexity of the algorithm is

$$\mathcal{O}\left(\sum_{v=0}^{n-1} 3 \cdot \deg(v)\right) = \mathcal{O}(6m) = \mathcal{O}(n).$$

More specifically, it is $\Theta(n)$.

5 Path List-Coloring and the Hartman-Škrekovski Algorithm

In this section we describe an algorithm for path list-coloring plane graphs with lists of size 3. The following algorithm on abstract graphs is due to the independent work of Hartman [19] and Škrekovski [24].

Note that the description of the algorithm given below is structured differently from the descriptions given by both Hartman and Škrekovski. This restructuring is,

in many ways, less elegant than both original proofs, but helps illuminate how the algorithm will operate on a graph with an adjacency list representation.

Algorithm 5.1. (Hartman-Škrekovski – Path Color)
Input: Let G be a 2-connected weakly triangulated plane graph with outer cycle $C = v_1 v_2 \dots v_k$. Let $x = v_1$ and $y \in C - x$. Suppose L is a list assignment for G such that for each vertex $v \in G$

$$\begin{vmatrix} L(v) \\ L(v) \\ L(v) \end{vmatrix} \geq \begin{matrix} 1 & \text{if } v = x \text{ or } v = y; \\ 2 & \text{if } v \in C - x - y; \\ 3 & \text{otherwise.} \end{matrix}$$

We will call x and y the *fixed* vertices. Assume that all vertices are uncolored except for potentially x and y. If x or y are colored, assume that $L(x)$, $L(y)$ contain only the color they have been assigned.
Output: A path L-list-coloring of G such that the fixed vertices x and y each receive at most one same color neighbor.
Description: If x is already colored let c be the color of x. Otherwise select arbitrary $c \in L(x)$. We will construct an induced path P, colored with c, and consisting of vertices from C. The path will begin at x and proceed clockwise along the outer face a far as possible towards y. Initialize P to consist of the single vertex x.

Suppose we have constructed an induced path $P = v_{j_1} v_{j_2} \dots v_{j_l}$ with $1 = j_1 < j_2 < \dots < j_l < k$. Let us select the largest integer i such that $v_i \in C[v_{j_l}, y]$ and $c \in L(v_i)$. If no such i exists we have finished constructing P. Otherwise we append v_i to P and repeat.

Let L' be a list assignment for G defined such that for $v \in G$

$$L'(v) = \begin{cases} \{c\}, & \text{if } v \in P; \\ L(v), & \text{otherwise.} \end{cases}$$

Let $P = v_{j_1} v_{j_2} \dots v_{j_l}$ be the path constructed above. For each vertex in $v_{j_i} \in P$, color v_i with c.

Suppose $i \in \{1, \dots, l-1\}$ such that $j_i + 1 < j_{i+1}$. We apply *Remove Path* (5.2) to path L'-list-color the subgraph bounded by the cycle consisting of $C[v_{j_i}, v_{j_{i+1}}]$ and the edge $v_{j_i} v_{j_{i+1}}$, with colored path $v_{j_{i+1}} v_{j_i}$, and fixed vertices $v_{(j_i)+1}$ and $v_{(j_{i+1})-1}$.

If $y \in P$ let us define $y' = v_{j_l+1}$, otherwise $y' = y$. We may apply Remove Path (5.2) to path L'-list-color the subgraph bounded by the cycle consisting of P and $C[v_{j_l}, v_{j_1}]$, with colored path P, and fixed vertices v_k and y'.

Pairwise all subgraphs above have only vertices in the path P in common. By *Remove Path* (5.2), no vertex with a neighbor in P will receive the color c. Therefore the combined coloring is a path L-list-coloring of G such that x, y receive at most one same color neighbor.

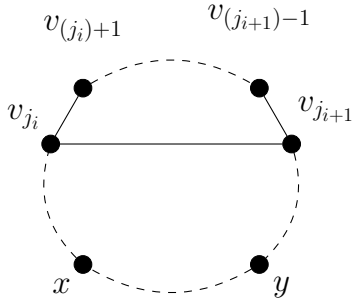

Figure 5.1: The case the path P uses a chord of C.

Algorithm 5.2. (Hartman-Škrekovski – Remove Path)
Input: Let G be a 2-connected weakly triangulated plane graph with outer cycle $C = v_1v_2 \dots v_k$. Let $P = v_1v_2 \dots v_l$ be an induced path in C. Let $x = v_k$ and $y \in C - P$. Let L be a list assignment for G such that for $v \in G$

$$\begin{array}{ll} L(v) = \{c\} & \text{if } v \in P; \\ \left|L(v)\right| \geq 1 & \text{if } v = x \text{ or } v = y; \\ \left|L(v)\right| \geq 2 & \text{if } v \in C - P - x - y; \\ \left|L(v)\right| \geq 3 & \text{otherwise.} \end{array}$$

Assume that if $\left|L(x)\right| = 1$ then $c \notin L(x)$. Additionally, assume that for every $v \in C[v_{l+1}, y]$, if v has a neighbor in P then $c \notin L(v)$.

We will once again refer to x and y as fixed vertices, although in this algorithm it may be the case that $x = y$. Assume that all vertices are uncolored except for potentially x and y. If x or y are colored assume that $L(x)$, $L(y)$ contain only the color they have been assigned.
Output: A path L-list-coloring of G such that x and y each receive at most one same color neighbor, and no vertex in $G - P$ with a neighbor in P receives the color c. If $x = y$ then x will receive no same colored neighbors in G.
Description: Note that G is 2-connected and weakly triangulated. Thus to disconnect G by removing vertices from C we would need to remove vertices $v_i, v_j \in C$ such that v_iv_j is a chord of C. Observe that P is a subgraph of C and an induced path in G, so no vertices in P induce a chord of C. So $G - P$ is connected.

Case 1: Suppose there is a chord of C with an endpoint in P. Let us select the smallest $i \in \{1, \dots, l\}$ and largest $j \in \{l+2, \dots, k-1\}$ such that $v_i \in P$ and v_iv_j is a chord of C. Let C_1 be the cycle consisting of $C[v_j, v_i]$ and the edge v_iv_j. Similarly, let C_2 be the cycle consisting of $C[v_i, v_j]$ and the edge v_iv_j. Let $P_1 = v_1v_2 \dots v_i$ and $P_2 = v_iv_{i+1} \dots v_l$.

Case 1.1: Suppose $y \in C[v_{l+2}, x]$ and $v_j \in C[v_{l+2}, y]$. Then $x, y \in C_1$. We will first apply *Remove Path* (5.2) to path L-list-color $\text{Int}(C_1)$ with the colored path P_1,

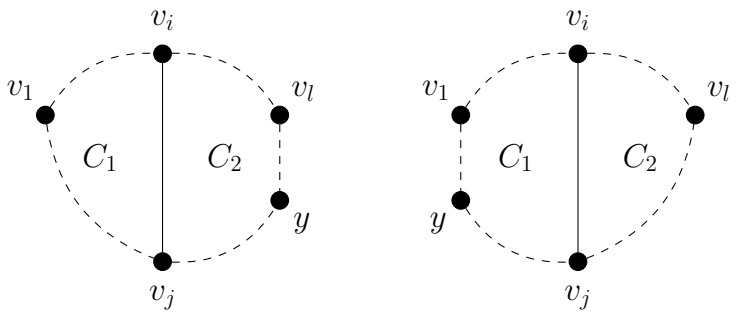

Figure 5.2: Hartman-Škrekovski case 1.1 (left) and case 1.2 (right).

and fixed vertices x and y. We then apply *Remove Path* (5.2) to path L-list-color $\text{Int}(C_2)$ with colored path P_2, and the single fixed vertex v_j.

The subgraphs $\text{Int}(C_1)$ and $\text{Int}(C_2)$ have only the chord v_iv_j in common. The vertex v_i is an endpoint of the colored path in both $\text{Int}(C_1)$ and $\text{Int}(C_2)$. Thus v_i will receive at most one same color neighbor in each. Since v_j is the single fixed vertex in $\text{Int}(C_2)$, v_j will receive no same color neighbors in $\text{Int}(C_2)$. Thus the combined coloring is a path L-list-coloring of G with x, y receiving the correct number of shared color neighbors.

Case 1.2: Otherwise $v_j \in C[y, v_{k-1}]$, $v_j \neq y$. Again, we first apply *Remove Path* (5.2) to path L-list-color $\text{Int}(C_1)$ with the colored path P_1, and fixed vertices x and v_j. We may then apply *Remove Path* (5.2) to path L-list-color $\text{Int}(C_2)$ with colored path P_2, and fixed vertices v_j and y.

Again $\text{Int}(C_1)$ and $\text{Int}(C_2)$ have only the chord v_iv_j in common. In both $\text{Int}(C_1)$ and $\text{Int}(C_2)$ the vertex v_j is fixed vertex, and v_i is an endpoint of the colored path. Therefore v_i and v_j will both have at most one same color neighbor in each subgraph. Thus the combined coloring is a path L-list-coloring of G with x, y receiving the correct number of shared color neighbors.

Case 2: Suppose there are no chords of C with endpoints in P. Let L' be a list assignment for $G - P$ defined by

$$L'(v) = \begin{cases} L(v) \setminus \{c\}, & \text{if } v \text{ has at least one neighbor in } P; \\ L(v), & \text{otherwise.} \end{cases}$$

Case 2.1: Suppose $G - P$ is 2-connected. Let $v \in G - P$.

Suppose v is not on the outer face of $G - P$. Then v has no neighbors in P and $|L'(v)| = \big|L(v)\big| \geq 3$.

Suppose v is on the outer face of $G - P$, $v \notin C$. Then v has at least one neighbor in P and $|L'(v)| \geq \big|L(v)\big| - 1 \geq 2$.

Finally, suppose $v \in C$. Since there are no chords of C with endpoints in P the only vertices in $C - P$ with neighbors in P are x and v_{l+1}. Recall we assumed that if

$v \in C[v_{l+1}, y]$ and v has at least one neighbor in P then $c \notin L(v)$. Thus if $v \neq x$ and $v \neq y$, then $|L'(v)| = \big|L(v)\big| \geq 2$. We ensured $c \notin L(x)$ if $\big|L(x)\big| = 1$. Thus if $v = x$ or $v = y$, $|L'(v)| \geq 1$.

Therefore L' meets the requirements of *Path Color* (5.1) with fixed vertices x and y. Moreover, by the definition of L', in a path L'-list-coloring of $G - P$ no vertex with a neighbor in P will receive the color c.

Case 2.1.1: Suppose $x \neq y$. Then we may apply *Path Color* (5.1) to path L'-list-color $G - P$ with fixed vertices x and y, the new path starting at x.

Case 2.1.2: Suppose $x = y$. If x is uncolored select arbitrary $c_x \in L'(x)$, color x with c_x, and define $L'(x) = \{c_x\}$. Apply *Remove Path* (5.2) to path L'-list-color $G-P$ with the colored path consisting of the single vertex x, and the vertices adjacent to x on the outer cycle of $G - P$ as fixed vertices. This ensures x receives no same colored neighbors in G.

Case 2.2: Finally, if $G - P$ is not 2-connected, then $G - P$ must be a complete graph on one or two vertices. It is simple to check we may L'-list-color $G - P$ such that the requirements hold.

Let G be a plane graph and L a list assignment such that $\big|L(v)\big| \geq 3$ for all $v \in G$. We may add edges to G until it is triangulated. Then we may apply *Path Color* (5.1), with arbitrary fixed vertices, to construct a path L-list-coloring. This yields the following result.

Theorem 5.1 (Hartman [19])**.** *All planar graphs are path* 3*-choosable.*

The Hartman-Škrekovski Algorithm with Adjacency Lists

In order to implement Hartman and Škrekovski's algorithm with adjacency lists there are two main challenges. First, we must be able to remove paths and locate the subgraphs for recursive calls. Second, we must be able to track the location of vertices on the outer face with respect to the fixed vertices x, and y. For example: when adding a vertex to the path $P = v_{j_1} v_{j_2} \dots v_{j_l}$ in *Path Color* (5.1), we need to know which neighbors of v_{j_l} lie in $C[v_{j_l}, y]$.

For now, let us assume that we have solved the second challenge described above. That is, given vertices $u, v, w \in C$, assume that we can determine whether $v \in C[u, w]$ in $\mathcal{O}(1)$ time.

Let G be a 2-connected weakly triangulated plane graph with an augmented adjacency list representation. Just as in Poh's algorithm, each call will be provided with a cycle $C = v_1 v_2 \dots v_k$ in G. The job of a particular recursive call is then to color the subgraph $\mathrm{Int}(C)$ such that the requirements of the Hartman-Škrekovski algorithm hold.

We will provide each vertex in G with a boolean vertex property to represent its state. All vertices in C will be have a state indicating they are on the outer face, and likewise vertices in $\mathrm{Int}(C) - C$ will have a state indicating they are not in C.

The list assignment L will be represented by vertex property storing a linked list of colors $L[v]$ for each $v \in G$. We will denote the number of colors in the linked list by $\big|L[v]\big|$. We will produce a coloring of G by reducing the size of each color list to one. Thus we consider a vertex v colored if $\big|L[v]\big| = 1$.

For each vertex $v_i \in C$ we will store a vertex property $\mathrm{Nbr}[v_i]$ called a *neighbor range*. The neighbor range of v_i will contain a pair of references to nodes in $\mathrm{Adj}[v_i]$, that is $\mathrm{Nbr}[v_i] = (r_1, r_2)$. The first reference r_1 will point to the node for v_{i-1} in $\mathrm{Adj}[v_i]$ and the reference r_2 will point to the node for v_{i+1}.

Neighbor ranges provide immediate access to the preceding and subsequent vertices of v_i in C. Additionally, they give start and stop nodes in $\mathrm{Adj}[v_i]$ for the list of neighbors of v_i that are contained in the subgraph $\mathrm{Int}(C)$.

Algorithm 5.3. (Hartman-Škrekovski – Path Color)
Assumptions: Suppose $C = v_1 v_2 \dots v_k$ is a cycle, $x = v_1$, and $y \in C - x$. Assume that for each $v \in \mathrm{Int}(C)$

$$\begin{vmatrix} L[v] \end{vmatrix} \geq 1 \quad \text{if } v = x \text{ or } v = y;$$
$$\begin{vmatrix} L[v] \end{vmatrix} \geq 2 \quad \text{if } v \in C - x - y;$$
$$\begin{vmatrix} L[v] \end{vmatrix} \geq 3 \quad \text{otherwise.}$$

Assume that the vertices of $\mathrm{Int}(C)$ have been marked according to whether they are in C or $\mathrm{Int}(C) - C$. Finally, assume that for each $v_i \in C$ we have constructed $\mathrm{Nbr}[v_i]$ as described above.
Input: The fixed vertices x and y.
Output: A path L-list-coloring of $\mathrm{Int}(C)$ such that x and y each receive at most one same color neighbor.
Description: If x is colored let c be the color of x. Otherwise let c be the first color in $L[x]$ and assign $L[x] = \{c\}$.

Initialize P to contain the single vertex x. We will now append vertices to P following the procedure of *Path Color* (5.1).

Suppose we have constructed an induced path $P = v_{j_1} v_{j_2} \dots v_{j_l}$ with $1 = j_1 < j_2 < \dots < j_l < k$. Let $v = v_{j_l}$ be the last vertex added to P. Let us iterate through $\mathrm{Adj}[v]$ counterclockwise from $v_{j_{l-1}}$, if $l = 1$ start from v_k. Let u be the current neighbor.

Case 1: If $u \notin C[v, y]$ or $c \notin L(u)$ then we ignore u and continue to the next vertex in $\mathrm{Adj}[v]$.

Case 2: Suppose $u = v_i \in C$, $u \in C[v, y]$, and $c \in L(u)$, that is, suppose we may add u to P. There are two cases to consider.

Case 2.1: Suppose the start node of $\mathrm{Nbr}[u]$ is not v. Then $u \neq v_{j_l+1}$. Let $\mathrm{Nbr}[u] = (r_1, r_2)$ and $\mathrm{Nbr}[v_{j_l}] = (s_1, s_2)$. Let r_v be the reference to the node for v in $\mathrm{Adj}[u]$ and s_u be the reference to the node for u in $\mathrm{Adj}[v]$.

Let us assign $\mathrm{Nbr}[u] = (r_1, r_v)$ and $\mathrm{Nbr}[v] = (s_u, s_2)$. We will then call *Remove Path* (5.4) on the cycle consisting of $C[v, u]$ and the edge uv, with colored path uv, and fixed vertices v_{i-1} and v_{j_l+1}.

We then assign $\text{Nbr}[u] = (r_v, r_2)$ and $\text{Nbr}[v] = (s_1, s_u)$. Finally, color u with c, assign $L[u] = \{c\}$, append u to P, and attempt to continue the path from u.

Case 2.2: Suppose the start node of $\text{Nbr}[u]$ is v. Then we may color u with c, assign $L[u] = \{c\}$, append u to P, and attempt to continue the path from u.

Let $P = v_{j_1}v_{j_2}\dots v_{j_l}$ be the path constructed above. If $y \in P$ let us define $y' = v_{j_l+1}$, otherwise $y' = y$. We may finally apply Remove Path (5.4) to the cycle formed by P and $C[v_{j_l}, v_{j_1}]$, with colored path P, and fixed vertices v_k and y'.
Complexity: See *Remove Path* (5.4).

Algorithm 5.4. (Hartman-Škrekovski – Remove Path)
Assumptions: Suppose $C = v_1v_2\dots v_k$ is a cycle. Let $P = v_1v_2\dots v_l$ be an induced path in C colored with some color c. Let $y \in C - P$ and $x \in C[y, v_k]$. Let L be a list assignment for G such that for each $v \in G$

$$\begin{array}{ll} L[v] = \{c\} & \text{if } v \in P; \\ \left|L[v]\right| \geq 1 & \text{if } v = x \text{ or } v = y; \\ \left|L[v]\right| \geq 2 & \text{if } v \in C - P - x - y; \\ \left|L[v]\right| \geq 3 & \text{otherwise.} \end{array}$$

Assume that for every $v \in C[x, v_k]$, $c \notin L(v)$. Additionally, assume that for every $v \in C[v_{l+1}, y]$, if v has a neighbor in P then $c \notin L(v)$.
Input: The vertices v_1, x, and y.
Output: A path L-list-coloring of G such that x and y each receive at most one same color neighbor, and no vertex in $G - P$ with a neighbor in P receives the color c. If $x = y$ then x will receive no same colored neighbors in G.
Description: We will remove the path P one vertex at a time. In this call we will be "removing" edges v_1u around v_1, by updating $\text{Nbr}[u]$ to exclude the edge. Note if $u \notin C$ we must first construct $\text{Nbr}[u]$. We will completely remove v_1 from $\text{Int}(C)$ if there are no chords v_1v_i of C.

Let us iterate counterclockwise through $\text{Adj}[v_1]$ beginning from v_k. Let u be the current neighbor of v_1.

Case 1: Suppose $u \notin C$. Look through $L[u]$ and remove the color c if it exists. After removing v_1, u will be on the outer face. Thus we set the state of u to indicate it is on the outer face. Construct $\text{Nbr}[u] = (r_1, r_2)$ such that r_1 is a reference to the node immediately prior to v_1 in $\text{Adj}[u]$ and r_2 is a reference to the node immediately subsequent to v_1.

Case 2: Suppose $u \in C$. There are several cases to consider.

Case 2.1: Suppose $u = v_k$. Let $\text{Nbr}[u] = (r_1, r_2)$. By our assumptions r_2 is a reference to the node for v_1 in $\text{Adj}[u]$. Reassign r_2 to be a reference to the node immediately prior to v_1 in $\text{Adj}[u]$. This removes v_1 from the set of neighbors of u contained in the cycle.

Case 2.2: Suppose $u \neq v_k$. In this case the edge v_1u is either a chord of C or $u = v_2$. Let N be the v_ku-path consisting of the neighbors of v_1. Let C_1 be the cycle

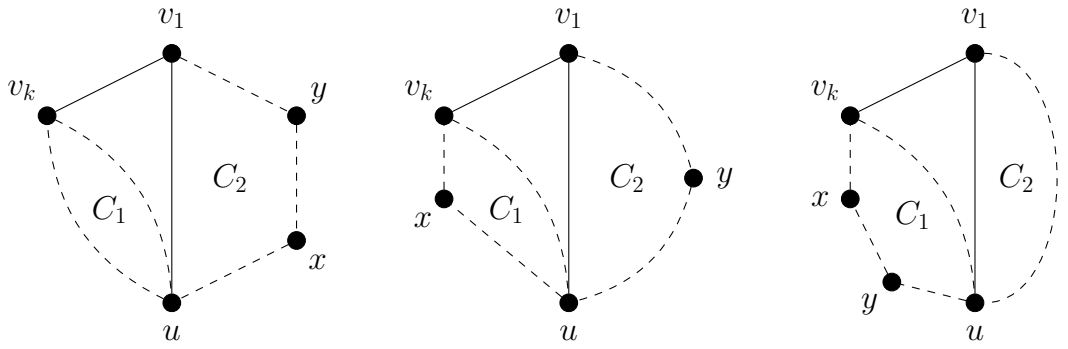

Figure 5.3: From left to right: case 2.2.1, case 2.2.2, and case 2.2.3.

consisting of N and $C[u, v_k]$. If $u \neq v_2$ let C_2 be the cycle consisting of $C[v_1, u]$ and the edge v_1u.

Let $\text{Nbr}[u] = (r_1, r_2)$ and $\text{Nbr}[v] = (s_1, s_2)$. Let r_v be the reference to the node for v in $\text{Adj}[u]$ and s_u be the reference to the node for u in $\text{Adj}[v]$.

In all cases below, before we apply an algorithm to color $\text{Int}(C_1)$ we will assign $\text{Nbr}[u] = (r_v, r_2)$ and $\text{Nbr}[v] = (s_1, s_v)$. Similarly, before applying an algorithm to color $\text{Int}(C_2)$ we will assign $\text{Nbr}[u] = (r_1, r_v)$ and $\text{Nbr}[v] = (s_u, s_2)$.

We will now path L-list-color $\text{Int}(C_1)$ and, if $u \neq v_2$, $\text{Int}(C_2)$. There are several cases to consider.

Case 2.2.1: Suppose $u \in C[x, v_1]$. Note that in this case it must be that $u \neq v_2$. We will first apply *Remove Path* (5.4) to path L-list-color $\text{Int}(C_2)$ with fixed vertices x and y. Note that this colors the vertex u. We will apply *Remove Path* (5.4) with colored path consisting of just the vertex u, and the vertices immediately adjacent to u on C_1 as the fixed vertices. This ensures u receives no same color neighbors in $\text{Int}(C_1)$.

Case 2.2.2: Suppose $u \in C[y, x]$, $u \neq y$. Again it must be that $u \neq v_2$. We apply *Color Path* (5.3) to path L-list-color $\text{Int}(C_1)$ with fixed vertices x and u, the new path starting at x. Next we apply *Remove Path* (5.4) to $\text{Int}(C_2)$ with fixed vertices u and y.

Case 2.2.3: Suppose $u \in C[v_1, y]$, $u \neq v_2$. We apply *Color Path* (5.3) to path L-list-color $\text{Int}(C_1)$ with fixed vertices x and y, the new path starting at x. We then apply *Remove Path* (5.4) to path L-list-color $\text{Int}(C_2)$ with the single fixed vertex u. This ensures u receives no same color neighbors in $\text{Int}(C_2)$.

Case 2.2.4: Suppose $u = v_2$. If $c \in L[u]$ then u is a path vertex and we apply *Remove Path* (5.4) to path L-list-color $\text{Int}(C_1)$ with fixed vertices x and y. Otherwise, we have reached the end of the path. We apply *Color Path* (5.3) to $\text{Int}(C_1)$ with fixed vertices x and y, the new path starting from x.

Complexity: Let $v \in G$. We iterate through $\text{Adj}[v]$ at most once during *Color Path* (5.3) when looking for the next vertex to add to the path containing v. In *Remove*

Path (5.4) we iterate through Adj[v] exactly once. We also iterate through Adj[v] once when we initially construct *Nbr*[v] to locate start and stop nodes. Therefore the overall complexity of the algorithm is

$$\mathcal{O}\left(\sum_{v=0}^{n-1} 3 \cdot \deg(v)\right) = \mathcal{O}(6m) = \mathcal{O}(n).$$

More specifically, it is $\Theta(n)$.

Tracking Vertices on the Outer Cycle

The Hartman-Škrekovski algorithm described in the previous section relied on the assumption that we could immediately know the relative location of vertices on the outer cycle. In *Path Color* (5.3) we assumed that we could determine whether a given vertex $u \in C$ was in the path $C[v, y]$, where v was the last vertex added to our colored path. Additionally, in *Remove Path* (5.4) we assumed that for $u \in C$ we could determine whether u was in $C[x, v_l]$, $C[y, x]$, or $C[v_1, y]$. We will now describe how this check may be accomplished in $\mathcal{O}(1)$ time.

Let us define an integer vertex property to store a location mark for each vertex on the outer cycle. Assume that we are given the input for *Remove Path* (5.4). Also, assume that vertices in $C[v_1, y]$ have been assigned the mark n_1, vertices in $C[y, x]$ have the mark n_2, and vertices in $C[y, v_k]$ have the mark n_3.

Let us iterate through Adj[v_1] starting from v_k as in (5.4). Let u be the current neighbor. If $u \notin C$ we will assign u the mark n_1. This is because u will be in $C_1[x, v_2]$ if there are no chords v_1v_i.

Now suppose we reach the end of the colored path, or we hit a chord v_1u with $u \in C[v_1, x]$. Then in the subsequent call to *Color Path* (5.3) on Int(C_1) we will need to treat vertices marked n_1 and n_2 as the same segment, since $C_1[x, y]$ consists of both $C_1[x, v_k]$ and $C_1[u, y]$. One solution is to walk along C_1 and remark vertices, but this is very inefficient. Another solution is to simply compare with both marks to check whether a vertex is in $C_1[x, y]$. However, we will be drawing further colored paths and generating further marks, hence the collection of marks to compare may grow very large.

Our solution is to use a disjoint set structure to compare location marks. All marks begin as singleton sets. To join the segments marked with n_1 and n_2 above we may simply perform a union operation in the disjoint set structure.

The mark n_1 for the segment $C_1[x, v_k]$ will always be a singleton set in the disjoint set structure. This is because the only vertices marked with n_1 are vertices that have been added to the outer face while removing vertices from the colored path. Thus in each union operation performed at least one of the two sets is always a singleton. Because of this, standard disjoint set optimizations allow set lookups in constant time. Therefore performing $\mathcal{O}(n)$ make set, union, and lookup operations in the disjoint set

structure requires $\mathcal{O}(n)$ time. Hence the overall performance of the algorithm remains linear.

For full details on managing location marks and disjoint set operations, see the provided C++ implementation.

6 Path 3-Coloring and Path List-Coloring in C++

In this section we detail the C++ implementation of each algorithm above. Instructions for using each algorithm are provided, as well as brief examples.

The Boost Graph Library (BGL) [23] details a generic interface for working with graphs, as well as numerous data structures and algorithms. We will begin with a brief introduction to the BGL. Then we will discuss implementing the Poh and Hartman-Škrekovski algorithms using BGL abstractions.

We will assume familiarity with the C++ language and the C++ Standard Template Library (STL). Full hyperlinked documentation is available, with links to the relevant Boost and STL concepts [7].

The Boost Graph Library

The BGL provides several abstract concepts for graph data structures. The basic `Graph` concept requires that vertex and edge types are defined, as well as a few other properties such as whether the graph is directed or undirected. The `VertexListGraph` and `EdgeListGraph` refine this concept to additionally require an interface to iterate over the vertex and edge sets, respectively. The `VertexAndEdgeListGraph` concept simply combines the two refinements.

Although other concepts exist such as `AdjacencyGraph`, we will only require that input graphs model `VertexListGraph` or `VertexAndEdgeListGraph`. The `AdjacencyGraph` concept might seem like an obvious choice, but we follow the BGL's decision and represent the rotation scheme for planar embeddings as an exterior property map. Thus the graph data structure itself remains fairly simple.

There are two different types of vertex and edge properties in the BGL: interior properties and exterior properties. Interior properties are properties that are are stored within the graph data structure. They are accessed or assigned via `get` or `put` functions, respectively, on the graph structure itself. Exterior properties are properties stored in a separate data structure. Calls to `get` and/or `put` on the property map structure then allow reads and/or writes to individual vertex or edge properties.

All our properties will be stored in exterior property maps that satisfy the `LvaluePropertyMap` concept. The `LvaluePropertyMap` concept requires that `get` calls on the property map return values by reference. The BGL defines the `PlanarEmbedding` concept to refine `LvaluePropertyMap` to require that each vertex is assigned a range of edges, representing the embedding ordered rotation scheme.

```
// Define our the graph, edge, and vertex types
typedef adjacency_list<
        vecS,
        vecS,
        undirectedS,
        property<vertex_index_t, size_t>,
        property<edge_index_t, size_t>
    > graph_t;
typedef typename graph_traits<graph_t>
    ::vertex_descriptor vertex_t;
typedef typename graph_traits<graph_t>
    ::edge_descriptor edge_t;

// Construct a simple planar graph on 5 vertices
graph_t graph(5);
add_edge(0, 1, graph);
add_edge(1, 2, graph);
add_edge(2, 0, graph);
add_edge(1, 3, graph);
add_edge(0, 3, graph);
add_edge(2, 3, graph);
add_edge(0, 4, graph);
add_edge(2, 4, graph);
add_edge(3, 4, graph);
```

Figure 6.1: Example code to construct a graph in the BGL.

The BGL provides the concrete `boost::adjacency_list` graph data structure that models `VertexAndEdgeListGraph`, among other concepts. We provide wrapper functions that will construct fast property maps (`boost::iterator_property_map`) for all the property maps just used as working space for the algorithms. In the examples here we will only discuss `boost::adjacency_list` structures, but functions are available in the library to allow the algorithms to work on arbitrary data structures modeling the necessary BGL concepts.

Code to construct a simple triangulated planar graph may be seen in Figure 6.1. A planar embedding for the graph is constructed in Figure 6.3. This graph and embedding will be used as an example input in the later sections.

We will use `graph_t` to refer to some definition of `boost::adjacency_list`. We will use `vertex_t` and `edge_t` to refer to the vertex and edge types of `graph_t` (see the type definitions in Figure 6.1 for an example).

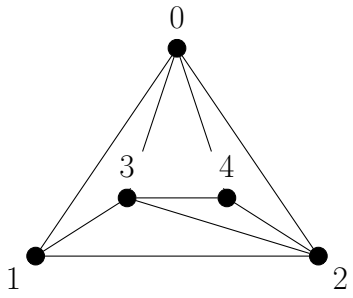

Figure 6.2: A drawing of the graph constructed in Figure 6.1.

```
// A vector to store embedding ordered incidence lists
vector<vector<edge_t>> planar_embedding_storage(
        num_vertices(graph)
    );

// Map each vertex to its incidence list
iterator_property_map<
        typename vector<vector<edge_t>>::iterator,
        property_map<graph_t, vertex_index_t>::const_type
    > planar_embedding(
        planar_embedding_storage.begin(),
        get(vertex_index, graph)
    );

// Reserve space so push_back is O(1)
for(size_t v = 0; v < num_vertices(graph); ++v) {
    planar_embedding[v].reserve(out_degree(v, graph));
}

// Construct the planar embedding
boyer_myrvold_planarity_test(
        boyer_myrvold_params::graph = graph,
        boyer_myrvold_params::embedding = planar_embedding
    );
```

Figure 6.3: Example code to construct a planar embedding in the BGL.

```
template<
        typename graph_t,
        typename planar_embedding_t,
        typename color_map_t,
        typename vertex_iterator_t,
        typename color_t
    >
void poh_color(
        const graph_t & graph,
        const planar_embedding_t & planar_embedding,
        vertex_iterator_t p_begin, vertex_iterator_t p_end,
        vertex_iterator_t q_begin, vertex_iterator_t q_end,
        color_t c_0, color_t c_1, color_t c_2,
        color_map_t & color_map
    );
```

Figure 6.4: Publicly visible function prototype for Poh with Path Walking.

Type	Concept	Additional Requirements
`graph_t`	none	must be `adjacency_list`
`color_t`	`EqualityComparable`, `CopyAssignable`	none
`embedding_t`	`PlanarEmbedding`	none
`vertex_iterator_t`	`InputIterator`	`value_type` is `vertex_t`
`color_map_t`	`LvaluePropertyMap`	`value_type` is `color_t`

Figure 6.5: Template requirements for Poh with Path Walking.

Poh's Algorithm

Here we describe our implementation of the linear time Poh algorithm described in (4.3) and (4.4). The function prototype, type definitions, and template requirements are shown in Figures 6.4 and Figure 6.5, respectively.

We assume that the provided graph is simple and weakly triangulated and the given planar embedding structure represents a valid planar embedding of the graph. We assume that the two ranges of vertices map are paths in the graph satisfying the requirements of *Poh – Path Walk* (4.3). Finally, we assume that no vertex is already colored any of `c_0`, `c_1`, or `c_2`.

When the algorithm is complete `color_map` will be assigned such that it represents a valid path 3-coloring of the subgraph bounded by the cycle formed by the two provided paths, $\text{Int}(C)$. The coloring will also satisfy the output requirements of (4.4).

```
// Create a vertex property map to store the coloring
vector<int> color_map_storage(num_vertices(graph));
iterator_property_map<
        vector<int>::iterator,
        typename property_map<
                graph_t, vertex_index_t
            >::const_type
    > color_map(
        color_map_storage.begin(), get(vertex_index, graph)
    );

// Construct the paths P and Q for the example graph
vector<vertex_t> path_p = { 0 };
vector<vertex_t> path_q = { 1, 2 };

// Color the graph with Poh
poh_color(
        graph,
        planar_embedding,
        path_p.begin(), path_p.end(),
        path_q.begin(), path_q.end(),
        1, 2, 3,
        color_map
    );
```

Figure 6.6: Example code to color a graph with Poh.

The implementation follows algorithms (4.3) and (4.4) with the following implementation decisions an modifications. We use a property map to track start and stop points in the cyclic ordering of neighbors provided by `planar_embedding`, similar to the neighbor range vertex property described in section 5. This allows us to do a single iteration through the neighbors of a vertex to get an "orientation", then remember this orientation for the remainder of the algorithm.

Additionally, we optimize the implementation by combining the steps of (4.3) and (4.4). Notice the path that is colored in (4.4) is the same path we will walk through in a subsequent call to (4.3). We may therefore perform the marking and chord finding operations of (4.3) as the path is colored in (4.4). This reduces the number of times we visit a particular edge by a factor of 2.

A brief code snippet in Figure 6.6 shows how to apply Poh to the graph constructed earlier in Figure 6.1. Source code, documentation, and complete examples are available online [7].

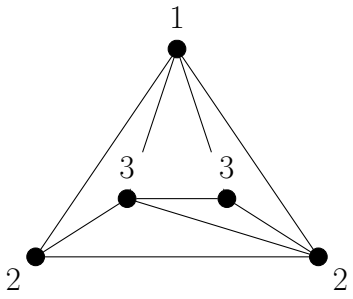

Figure 6.7: The coloring produced by the code in Figure 6.6.

```
template<
        typename graph_t,
        typename planar_embedding_t,
        typename vertex_iterator_t,
        typename color_t,
        typename color_map_t
    >
void poh_color_bfs(
        const graph_t & graph,
        const planar_embedding_t & planar_embedding,
        vertex_iterator_t p_begin, vertex_iterator_t p_end,
        vertex_iterator_t q_begin, vertex_iterator_t q_end,
        color_t c_0, color_t c_1, color_t c_2,
        color_map_t & color_map
    );
```

Figure 6.8: Publicly visible function prototype for Poh with BFS.

Poh with BFS

We also provide an implementation of of *Poh – BFS* (4.2). Note that the BFS algorithm is slower than the algorithm described in (4.3, 4.4). The function prototype, type definitions, and template requirements are shown in Figure 6.8 and Figure 6.9, respectively.

We make the exact same assumptions about input structures as in our implementation of (4.3, 4.4) in the previous section.

When the algorithm is complete `color_map` will be assigned such that it represents a valid path 3-coloring of the subgraph bounded by the cycle formed by the two provided paths (Int(C)). The coloring will also satisfy the output requirements of (4.2).

Type	Concept	Additional Requirements
graph_t	none	must be adjacency_list
color_t	EqualityComparable, CopyAssignable	none
embedding_t	PlanarEmbedding	none
vertex_iterator_t	InputIterator	value_type is vertex_t
color_map_t	LvaluePropertyMap	value_type is color_t

Figure 6.9: Template requirements for Poh with BFS.

Type	
augmented_embedding_t	a type modeling AugmentedEmbedding
node_t	boost::property_traits<augmented_embedding_t> ::value_type::value_type
iterator_t	boost::property_traits<augmented_embedding_t> ::value_type::iterator
graph_t	the type of the underlying graph
vertex_t	boost::graph_traits<graph_t>::vertex_descriptor

Figure 6.10: Types for the AugmentedEmbedding concept.

The implementation almost directly follows the description of (4.2), although some decisions had to be made. In order to keep the algorithm $\mathcal{O}(n^2)$, we must ensure we iterate through the adjacency list of a given vertex precisely once during the orientation phase of (4.2). This is the step where we would locate the position of v_k in Adj$[v_1]$. To do this we note that at least one of the path endpoint vertices v_1 or v_k has never been an path endpoint vertex in any call before. We then always choose this endpoint as the vertex whose adjacency list we search through.

Source code, documentation, and complete examples are available online [7].

Augmented Embeddings

In this section we describe the AugmentedEmbedding concept used to store embedding ordered augmented adjacency lists for a graph.

The AugmentedEmbedding concept refines LvaluePropertyMap, placing additional restrictions on the value_type of the map. The types are described in Figure 6.10. Valid expressions are described in Figure 6.12.

The object embedding will assign a range of objects of type node_t to each vertex v in the underlying graph. There will be exactly one node in this range for each neighbor of v in the underlying graph. We will call this range of nodes the augmented adjacency list for v.

Object(s)	Description
u,v	objects of type vertex_t
embedding	an object of type augmented_embedding_t
n	an object of type node_t

Figure 6.11: Notation for our discussion of augmented embeddings.

Expression	Type	Description
n.vertex	vertex_t	vertex member for the node n
n.iterator	iterator_t	iterator member for the node n
embedding[v].begin()	iterator_t	beginning of the range of nodes
embedding[v].end()	iterator_t	end of the range of nodes
embedding[v].push_back(n)	void	append n to the range of nodes
embedding[v].clear()	void	clear the range of nodes

Figure 6.12: Valid expressions for an object modeling AugmentedEmbedding.

The type node_t will represent a neighboring vertex u in the augmented adjacency list for a vertex v. The type iterator_t will be an iterator for the range of node_t objects for a vertex v.

For a vertex v each node n in the range embedding[v].begin() to embedding [v].end() will have n.vertex be a neighboring vertex u and n.iterator be the unique iterator in the range embedding[u].begin() to embedding[u].end() such that n.iterator->vertex is equal to v.

We implement an algorithm to construct a data structure modeling Augmented Embedding from a structure modeling PlanarEmbedding based on *Augment Embedding* (3.1).

A code snippet in Figure 6.13 shows how to construct an augmented embedding structure for the graph from Figure 6.1.

Hartman-Škrekovski in the BGL

Here we detail our implementation of the Hartman-Škrekovski algorithm described in (5.3, 5.4). The function prototype and template requirements are shown in Figure 6.14 and Figure 6.15, respectively.

We assume that the provided graph is simple and weakly triangulated and the given augmented_embedding structure represents a valid planar embedding of the graph. We assume that the range of vertices is a cycle in the provided plane graph, with vertices listed in clockwise order. We finally assume that color_list_map assigns each vertex in the cycle a sequence of at 2 or more colors, and each vertex interior to the cycle a sequence of 3 or more colors.

```
// Struct to store (v,r) pairs for augmented adjacency list
struct adjacency_node_t {
    vertex_t vertex;
    typename vector<adjacency_node_t>::iterator iterator;
};

// Create a vector to store augmented adjacency lists
vector<vector<adjacency_node_t>> augmented_embedding_storage(
        num_vertices(graph)
    );

// Map each vertex to its augmented adjacency list
iterator_property_map<
        vector<vector<adjacency_node_t>>::iterator,
        typename property_map<
                graph_t, vertex_index_t
            >::const_type
    > augmented_embedding(
        augmented_embedding_storage.begin(),
        get(vertex_index, graph)
    );

// Reserve space so push_back is O(1)
for(size_t v = 0; v < num_vertices(graph); ++v) {
    augmented_embedding[v].reserve(out_degree(v, graph));
}

// Fill in the augmented embedding structure
augment_embedding(
        graph, planar_embedding, augmented_embedding
    );
```

Figure 6.13: Example code to construct an augmented embedding.

```
template<
        typename graph_t,
        typename augmented_embedding_t,
        typename color_list_map_t,
        typename face_iterator_t
    >
void hartman_skrekovski_color(
        const graph_t & graph,
        const augmented_embedding_t & augmented_embedding,
        face_iterator_t face_begin, face_iterator_t face_end,
        color_list_map_t & color_list_map
    );
```

Figure 6.14: Publicly visible function prototype for Hartman-Škrekovski.

Type	Concept	Additional Requirements
`graph_t`	none	must be `adjacency_list`
`color_t`	`EqualityComparable`, `CopyAssignable`	none
`embedding_t`	`PlanarEmbedding`	none
`vertex_iterator_t`	`InputIterator`	`value_type` is `vertex_t`
`color_list_t`	`SequenceContainer`	`value_type` is `color_t`
`color_list_map_t`	`LvaluePropertyMap`	`value_type` is `color_list_t`

Figure 6.15: Template requirements for Hartman-Škrekovski.

```
// Vector listing vertices on the outer cycle
vector<vertex_t> cycle = { 0, 1, 2 };

// Create a structure to store the list assignment
vector<list<int>> color_list_storage(num_vertices(graph));
iterator_property_map<
        vector<list<int>>::iterator,
        typename property_map<
                graph_t, vertex_index_t
            >::const_type
    > color_list_map(
        color_list_storage.begin(), get(vertex_index, graph)
    );

// Assign each vertex a list of the appropriate size
color_list_map[0] = { 1, 2 };
color_list_map[1] = { 2, 3 };
color_list_map[2] = { 1, 4 };
color_list_map[3] = { 1, 3, 4 };
color_list_map[4] = { 1, 2, 4 };

// Construct the path list-coloring
hartman_skrekovski_color(
        graph, augmented_embedding,
        cycle.begin(), cycle.end(),
        color_list_map
    );
```

Figure 6.16: Example code to color a graph with Hartman-Škrekovski.

When the algorithm is complete `color_list_map` will have been modified such that a single color remains in each list. The remaining colors will represent a valid path coloring of the subgraph bounded by the provided cycle ($\text{Int}(C)$).

The implementation follows algorithms (5.3) and (5.4) with the following optimization. By combining the cases of (5.3) and (5.4) we may draw and remove the path simultaneously, one vertex at a time. In the implementation therefore perform the operations of both (5.3) and (5.4) simultaneously as each vertex is colored. This reduces the number of times we visit a particular edge by a factor of 2.

A brief code snippet in Figure 6.16 shows how to apply Hartman-Škrekovski to the graph constructed earlier in Figure 6.1. Source code, documentation, and complete examples are available online [7].

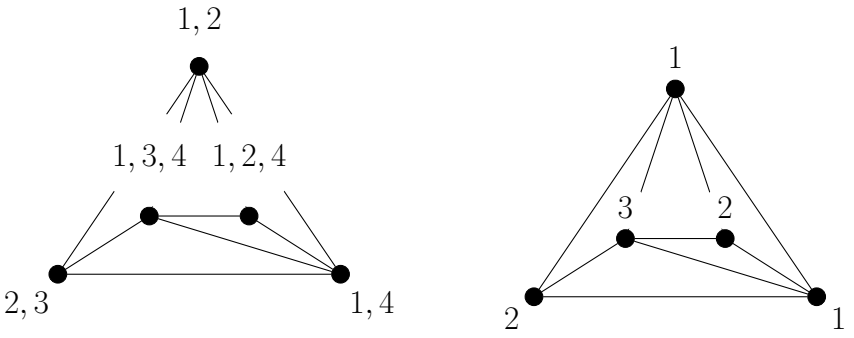

Figure 6.17: The coloring produced by the code in Figure 6.16.

7 Conclusion

In this project we considered two inductive procedures on plane graphs: one computing a path 3-coloring, and one computing a path list-coloring given lists of size at least 3. We adapted each procedure to an algorithm for finding path colorings of graphs with adjacency list representations. Additionally, we showed each procedure admits an algorithm that runs in linear time. Finally, we provided a documented implementation of each algorithm in C++.

Future work in this area might consider Hartman's procedure for path 4-coloring torus graphs, also found in [19]. The procedure first cuts and collapses a noncontractible cycle in the torus graph to form a plane graph. It then divides the resulting plane graph into several subgraphs which are individually colored with Poh's algorithm. The combined coloring is then adapted to a path 4-coloring of the original torus graph. It would be interesting to see if this procedure can be adapted to a linear time algorithm.

References

[1] Andrews, J. A. and Michael S. Jacobson, On a generalization of chromatic number, *Congressus Numerantium* **47** (1985), 33-48.

[2] Appel, K. and W. Haken, Every planar map is four colorable. I. Discharging, *Illinois J. Math* **21** (1991), no. 3, 429-490.

[3] Appel, K., W. Haken, J. Koch, Every planar map is four colorable. II. Reducibility, *Illinois J. Math* **21** (1991), no. 3, 491-567.

[4] Booth, K. and C. Lueker, Testing for the consecutive ones property interval graphs and graph planarity using *PQ*-tree algorithms, *J. Comput. System Sci.* **13** (1976), 335-379.

[5] Boyer, J. and W. Myrvold, On the cutting edge: simplified $\mathcal{O}(n)$ planarity by edge addition, *J. Graph Algorithms Appl.* **8** (2004), 241-273.

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

[7] Bross, A., 2017: Path coloring algorithms for plane graphs. [available from `http://github.com/permutationlock/path_coloring_bgl`.]

[8] Bross, A., G. G. Chappell, and C. Hartman, Path coloring algorithms for plane graphs, in preparation.

[9] Chartrand, G., D. P. Geller, and S. Hedetniemi, A generalization of the chromatic number, *Proc. Cambridge Philos. Soc.* **64** (1968), 265-271.

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

[11] Cowen, L., R. Cowen, and D. Woodall, Defective colorings of graphs in surfaces: partitions into subgraphs of bounded valency, *J. Graph Theory* **10** (1986), 187-195.

[12] Eaton, N. and N. Hull, Defective list colorings of planar graphs, *Bull. Inst. Combin. Appl.* **25** (1999), 79-87.

[13] Erdös, P., A. Rubin, H. Taylor, Choosability in graphs, *Congressus Numerantium* **26** (1980), 125-157.

[14] Eswaran, K. and R. Tarjan, Augmentation problems, *SIAM J. Comput.* **5** (1976), 653-665.

[15] Goddard, W., Acyclic colorings of planar graphs, *Discrete Math* **91** (1991), 91-94.

[16] Hagerup, T. and C. Uhrig, Triangulating a planar graph, *Library of Efficient Datatypes and Algorithms*, software package, Max Planck Institute for Informatics, Saarbrücken, 1991.

[17] Hopcroft, J. and E. Tarjan, Efficient planarity testing, *J. Assoc. Comput. Mach.* **21** (1974), 549-568.

[18] Harary, F. and K. Jones, Conditional colorability II: bipartite variations, Proceedings of the Sundance conference on combinatorics and related topics (Sundance, Utah, 1985), *Congressus Numerantium* **50** (1985), 205-218.

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

[20] Lempel, A., S. Even, and I. Cederbaum, An algorithm for planarity testing of graphs, *Theory of graphs (Internat. Sympos., Rme, 1966)*, 215-232, Gordon and Breach, New York; Dunod, Paris, 1967.

[21] Poh, K. S., On the linear vertex-arboricity of a planar graph, *J. Graph Theory* **14** (1990), 73-75.

[22] Reed, R., A new method for drawing a planar graph given the cyclic order of the edges at each vertex, Sixteenth Manitoba conference on numerical mathematics and computing (Winnipeg, Man., 1986), *Congressus Numerantium* **56** (1987), 31-44.

[23] Siek, J., L. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual*, Pearson Education, 2001.

[24] Škrekovski, R., List improper colourings of planar graphs, *Combin. Probab. Comput.* **8** (1999), 293-299.

[25] Thomassen, C., Every planar graph is 5-choosable, *J. Combin. Theory Ser. B* **62** (1994), no. 1, 180-181.

[26] Voigt, M., List colorings of planar graphs, *Discrete Mathematics* **120** (1993), 215-219.

[27] West, D., *Introduction to Graph Theory*, 2nd ed., Pearson, 2001.