Implementing algorithms for path 3-coloring and path 3-list-coloring plane graphs

Final Talk

Aven Bross

February 28, 2017

Committee

- Dr. Chappell (Chair)
- Dr. Lawlor
- Dr. Hartman

Project goal

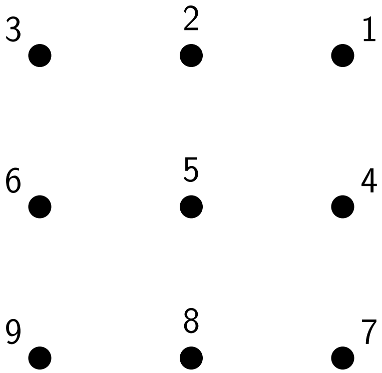
Produce a nice, documented implementation of two previously unimplemented algorithms for coloring planar graphs.

Overview

Graphs

A **graph** consists of:

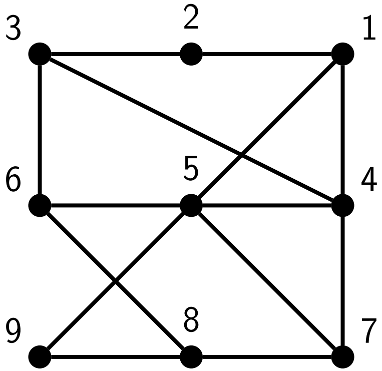
1. a set of objects called **vertices**;



Graphs

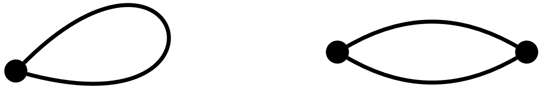
A **graph** consists of:

1. a set of objects called **vertices**;
2. a set of **edges** between pairs of vertices.



Simple graphs

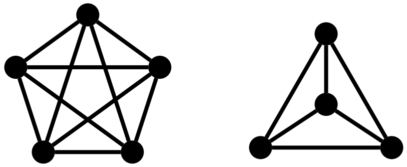
All graphs in this talk will be **simple** graphs which have no **loops** or **parallel edges**.



A loop (left) and two parallel edges (right).

Plane graphs

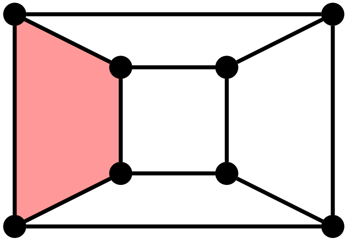
We are interested in **planar graphs** which are graphs that may be drawn in the plane without crossing edges. A planar graph along with a particular drawing is called a **plane graph**.



The nonplanar graph K_5, and the planar graph K_4.

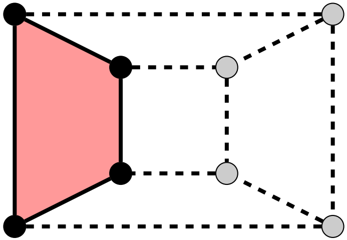
Faces

A **face** of a plane graph is a maximal region of the plane not containing any edges or vertices.



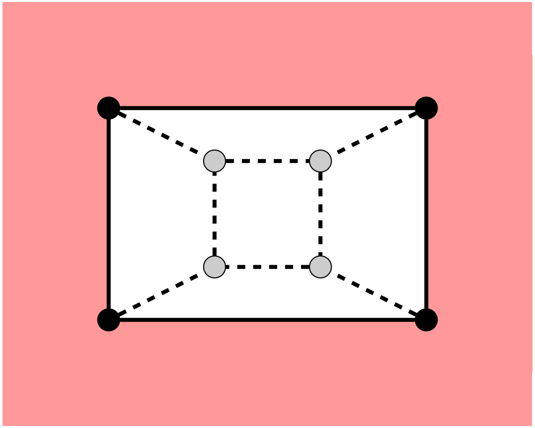
Faces

We will refer to a face by the subgraph of vertices and edges lying on its border.



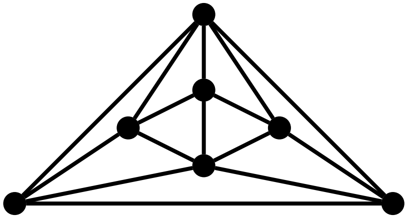
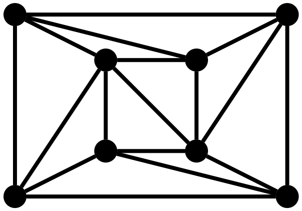
Faces

The unbounded region is also a face, known as the **outer face**.



Triangulation

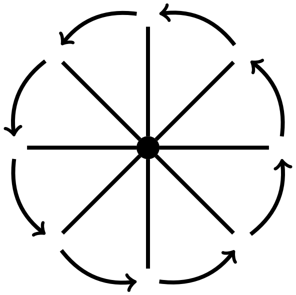
A graph is **triangulated** if all of its faces are triangles. If a graph has a single nontriangle face we say it is **weakly triangulated**.

A triangulated graph and a weakly triangulated graph.

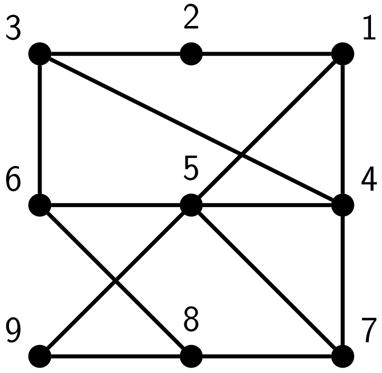
Rotation schemes

A plane graph naturally provides a cyclic ordering of the edges around each vertex, called a **rotation scheme**. In fact, the rotation scheme tells us everything we need to know about the plane graph.



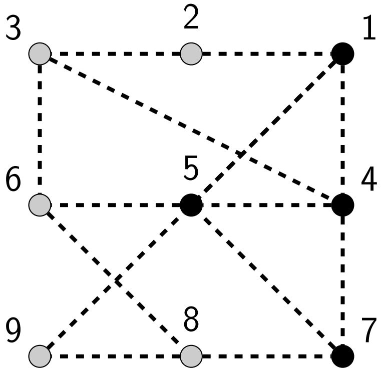
Induced subgraphs

Given a subset S of vertices of a graph G, the **induced subgraph** of S consists of all edges in G between vertices in S.



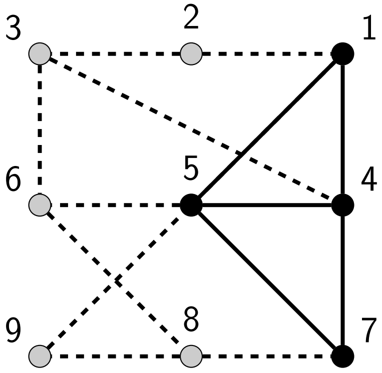
Induced subgraphs

Given a subset S of vertices of a graph G, the **induced subgraph** of S consists of all edges in G between vertices in S.



Induced subgraphs

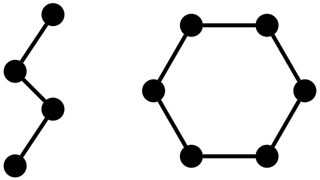
Given a subset S of vertices of a graph G, the **induced subgraph** of S consists of all edges in G between vertices in S.



Paths

A **path** is a sequence of distinct vertices with edges between consecutive vertices.

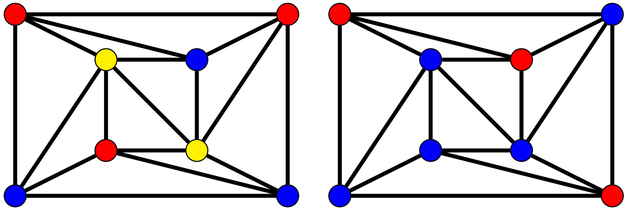
A **cycle** consists of a path and an edge between the first and last vertex.



A length 4 path, and a 6-cycle.

Coloring

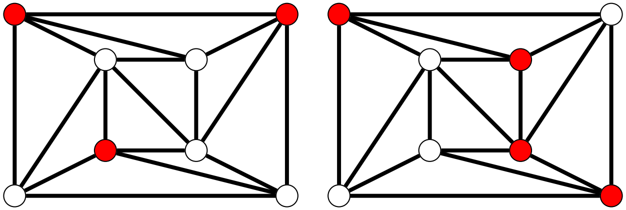
A (vertex) **coloring** of a graph assigns a color to each vertex. A k-coloring is a coloring that uses at most k colors.



Two different colorings.

Coloring

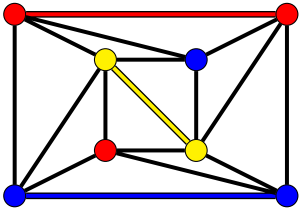
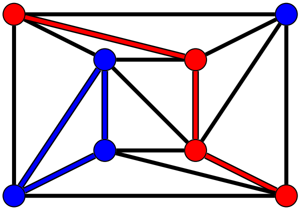
The set of all vertices of a particular color is called a **color class**.



The color class of red in each graph.

Path coloring

A **path coloring** is a coloring such that each color class induces a collection of disjoint paths.

A path 3-coloring and a 2-coloring that is not a path coloring.

Poh

Theorem (Poh, 1990)

All planar graphs admit a path 3-coloring.

In his proof Poh described a constructive procedure to produce such a coloring. We will describe an efficient implementation of Poh's algorithm.

Overview

Representing graphs

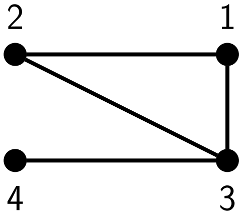
In order to work with graphs on computers we require an efficient data structure to represent a graph.

Suppose G is a graph with n vertices and m edges. We will assume the vertices of G are the integers $1, 2, \ldots, n$.

The input size will always be the number of vertices n. However, for plane graphs $\mathcal{O}(m) = \mathcal{O}(n)$, so it is equivalent to take the input size to be the number of edges.

Adjacency lists

For each vertex we define a linked list known as an **adjacency list** containing its neighbors. The full graph is then represented by a size n array Adj such that each vertex v has adjacency list Adj[v].



$\text{Adj}[1] = 2 \rightarrow 3$

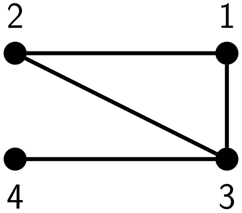
$\text{Adj}[2] = 1 \rightarrow 3$

$\text{Adj}[3] = 1 \rightarrow 4 \rightarrow 2$

$\text{Adj}[4] = 3$

Adjacency lists

Adjacency list representations may simultaneously store a rotation scheme for a plane graph by ordering the neighbors in each list.



$\text{Adj}[1] = 2 \to 3$

$\text{Adj}[2] = 1 \to 3$

$\text{Adj}[3] = 1 \to 4 \to 2$

$\text{Adj}[4] = 3$

Triangulations

Given an arbitrary planar graph with an adjacency list representation, linear time algorithms exist to embed it in the plane, i.e. order it's adjacency lists such that they correspond to a drawing of the graph with no edge crossings.

Moreover, given a plane graph, linear time algorithms exist to add edges until the graph is triangulated.

Vertex properties

Idea

We often wish to track various properties about the vertices of a graph, such as what color they have received, or whether they are in a particular subgraph.

Vertex properties

Idea

We often wish to track various properties about the vertices of a graph, such as what color they have received, or whether they are in a particular subgraph.

Implementation

Properties will be represented by constructing a size n array indexed by vertices. Thus accessing a vertex property will be a basic operation.

Vertex properties

Idea

We often wish to track various properties about the vertices of a graph, such as what color they have received, or whether they are in a particular subgraph.

Implementation

Properties will be represented by constructing a size n array indexed by vertices. Thus accessing a vertex property will be a basic operation.

Example

An adjacency list is a vertex property.

Vertex marking

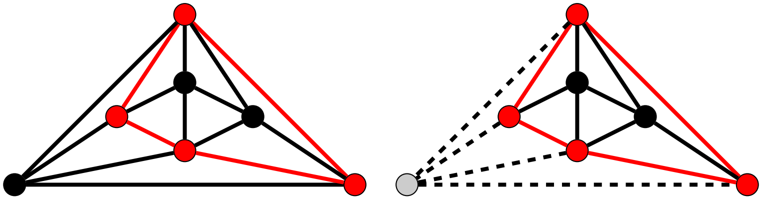
We will often use an integer vertex property called a mark.

To represent a path or cycle we mark all vertices on the path or cycle with a unique integer identifying the path.

To perform a breadth first search we mark vertices that have already been visited.

Cycles and plane graphs

Given a cycle in a plane graph we are guaranteed no interior vertices are connected with exterior vertices. Therefore by marking a cycle in our graph we simultaneously represent a subgraph.



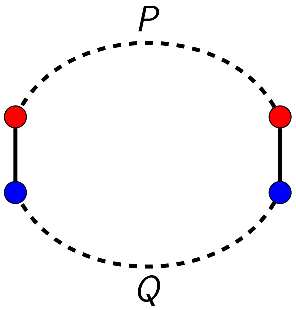
A cycle in a triangulated graph.

Overview

Poh's algorithm

Input: A weakly triangulated plane graph G such that it's outer face is a cycle C, and a 2-coloring of C such that each color class induces a path, denoted P and Q respectively.

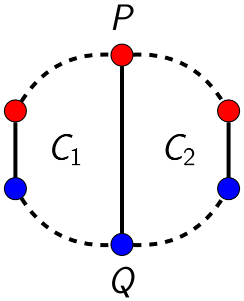
Output: An extension of the 2-coloring of C to a path 3-coloring of G such that no vertex in C receives a same color neighbor in $G - C$.



Poh's algorithm

Case (1)

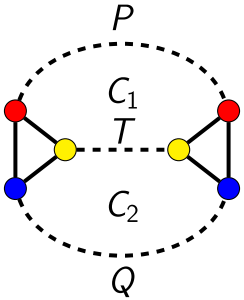
If there exists an edge between P and Q that is not in C we may apply Poh's algorithm to separately path 3-color the interior of the cycles C_1 and C_2 seen below.



Poh's algorithm

Case (2)

If no such edge exists and there are vertices left to color then we find the shortest path T through the interior. We may then color T with the remaining color and apply Poh's algorithm to separately color the interior of the cycles C_1 and C_2 seen below.



Procedure to path 3-color a planar graph

Given an arbitrary plane graph we may add edges until it is triangulated.

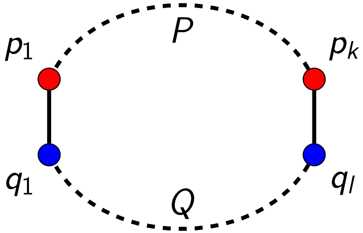
By path 2-coloring the outer triangle of the triangulated graph we may then apply Poh's algorithm to produce a path 3-coloring.

The resulting coloring is also a 3-coloring of the original graph, with the additional "triangulation edges" removed.

Implementing Poh's algorithm

Input: A triangulated graph, and the first and last vertex of two marked paths $P = p_1 p_2 \dots p_k$ and $Q = q_1 q_2 \dots q_l$ that satisfy the requirements of Poh's algorithm.

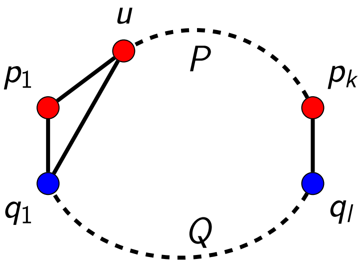
Output: A path 3-coloring of the interior of the cycle formed by P and Q such that none of the vertices in P or Q receive a new same color neighbor.



Implementing Poh's algorithm

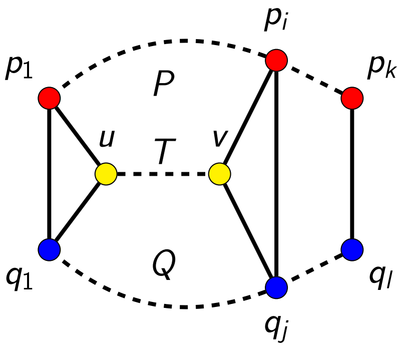
We begin by locating q_1 in Adj$[p_1]$. Let u be the vertex clockwise from q_1 in Adj$[p_1]$. Note the cycle $p_1 q_1 u$ is a triangle face.

If u is in P we apply the algorithm to $P - p_1$ and Q. Similarly, if u is in Q we apply the algorithm to P and $Q - q_1$.

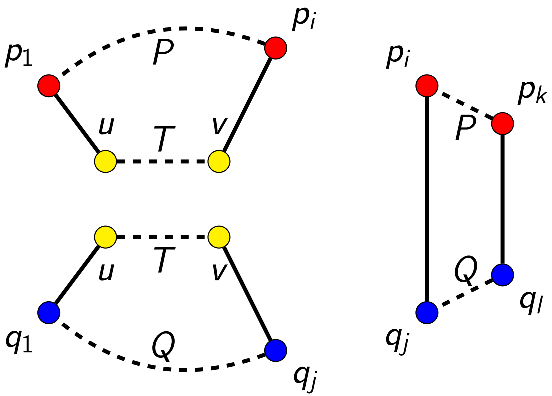


Implementing Poh's algorithm

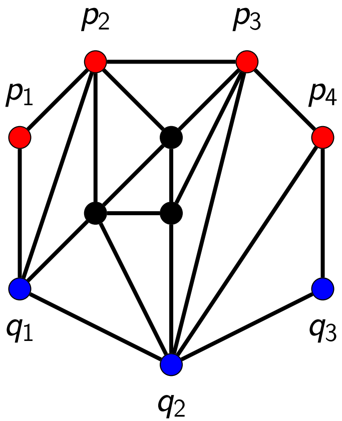
Otherwise, u is an interior vertex. Perform a breadth first search from u through the interior vertices until we find a vertex v with a neighbor in Q immediately clockwise from a neighbor in P. We may then backtrack along the search to color the path T.



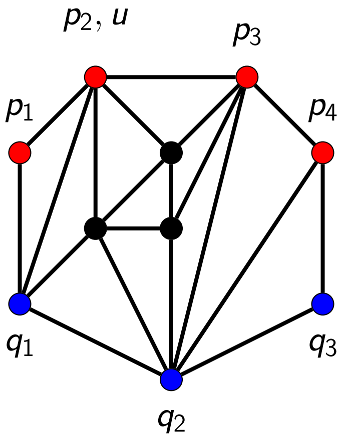
Implementing Poh's algorithm



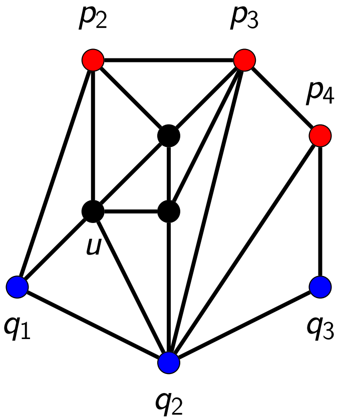
Poh coloring example



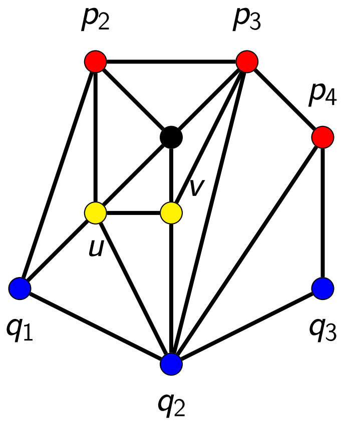
Poh coloring example



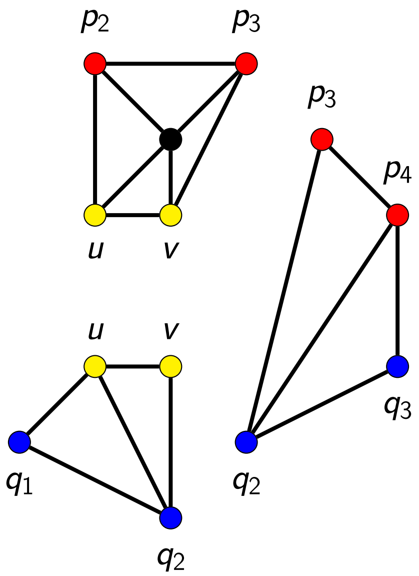
Poh coloring example



Poh coloring example



Poh coloring example



Poh time complexity

Poh's algorithm potentially performs a breadth first in each call. Unfortunately this results in a worst case running time that is not linear.

However, another way to find an induced path through the cycle is walking along the inside one of the paths P or Q. Using this method produces a similar algorithm that runs in linear time.

Overview

Path 3-list-coloring

Suppose L maps each vertex of a graph G to a list of colors. Then a **path list-coloring** of G from L is a path coloring such that each vertex v receives a color from $L(v)$.

Theorem (Hartman, 1997)

All planar graphs admit a path list coloring if each vertex is assigned a list of at least 3 colors.

The proof provides a constructive algorithm to produce a such a coloring.

Independently, around the same time Skrekovski proved a slightly weaker result using the same coloring procedure.

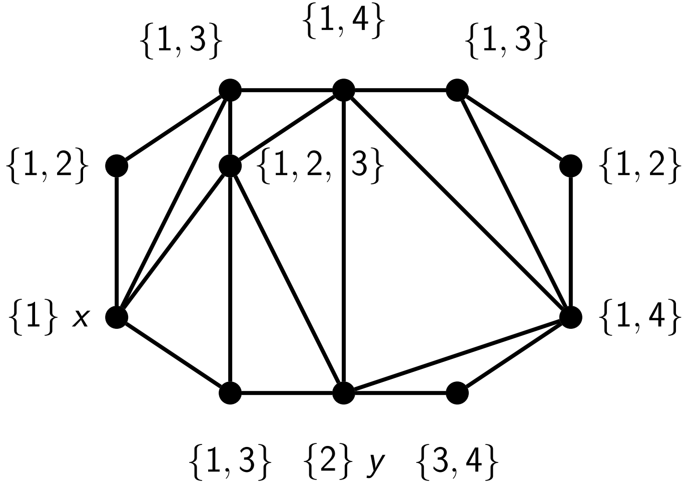
We will describe Hartman and Skrekovski's coloring algorithm, and then describe how it may be implemented in linear time.

Hartman-Skrekovski list-coloring

Input: A weakly triangulated plane graph G whose outer face is a cycle C, and a pair of vertices x, y in C. Also, a list assignment L such that x, y receive at least 1 color, other vertices in C receive at least 2 colors, and interior vertices receive at least 3 colors.

Output: A path list-coloring of G from L.

Hartman-Skrekovski list-coloring



Hartman-Skrekovski list-coloring

Select a color c from $L(x)$. We will now color an induced path with c as far as possible clockwise along C towards y.

Procedure:

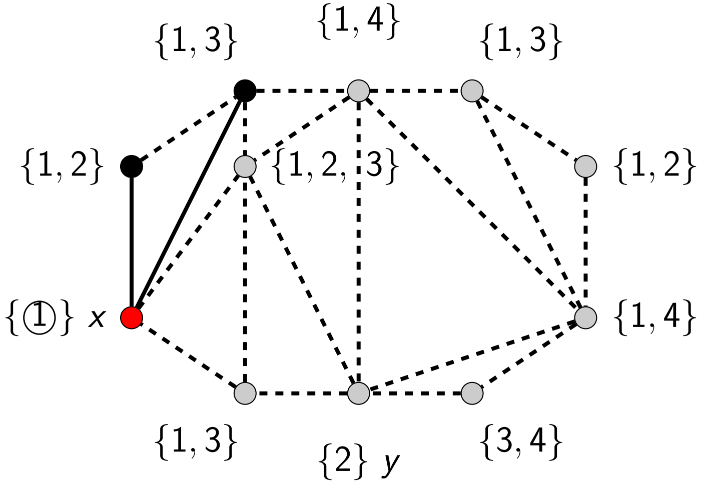
Initialize the path P to contain the single vertex x.

Let v be the last vertex of P. Let u be the furthest vertex clockwise from v along C such that

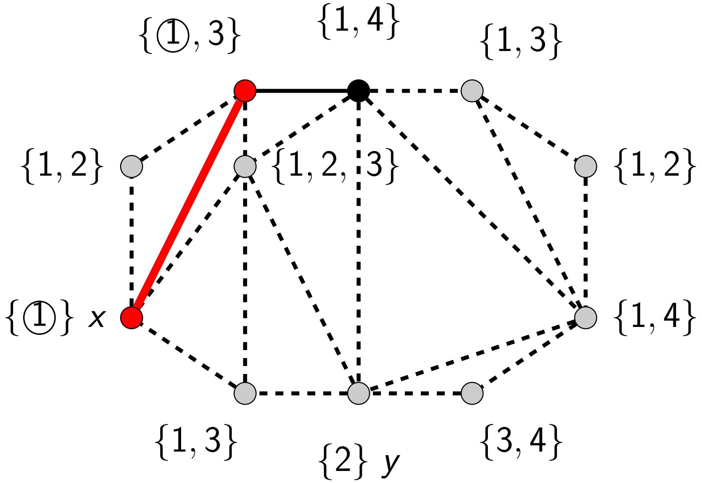
1. u is adjacent to v;
2. u lies between v and y clockwise along C;
3. $c \in L(u)$.

If such a u exists, append u to P and repeat. Otherwise, we are done.

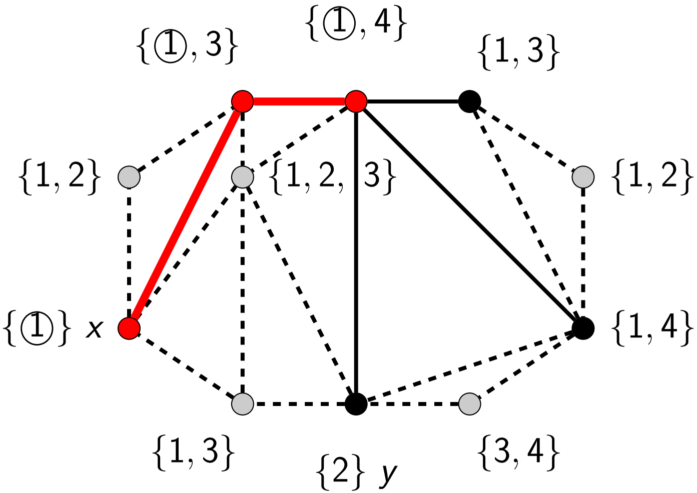
Hartman-Skrekovski list-coloring



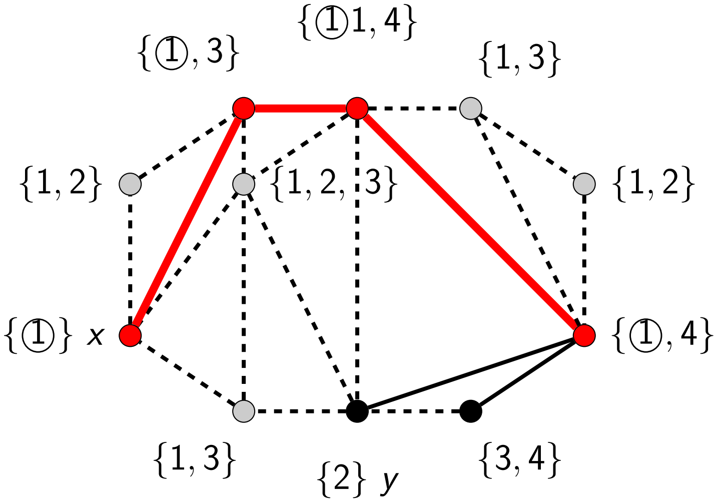
Hartman-Skrekovski list-coloring



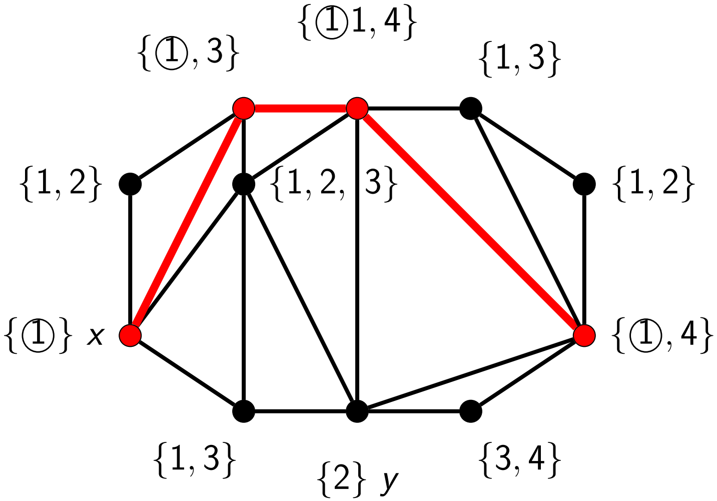
Hartman-Skrekovski list-coloring



Hartman-Skrekovski list-coloring



Hartman-Skrekovski list-coloring

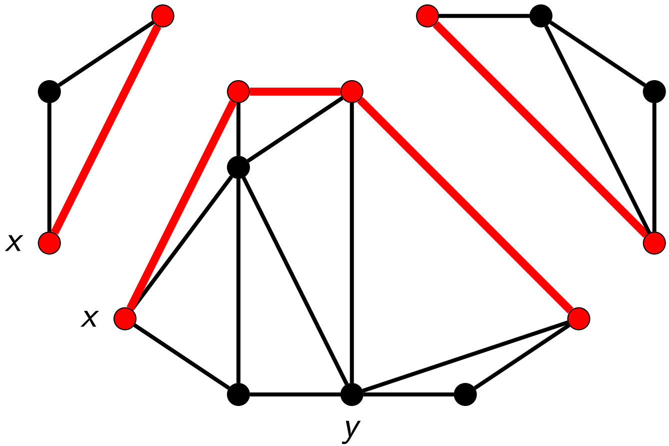


Hartman-Skrekovski list-coloring

If an edge uv of the path P is not an edge of C we will then separately consider the two subgraphs formed by dividing along uv. The section not containing x and y will be called a **lobe**.

We now have several weakly triangulated subgraphs, each with a path colored c along their outer cycle.

Hartman-Skrekovski list-coloring



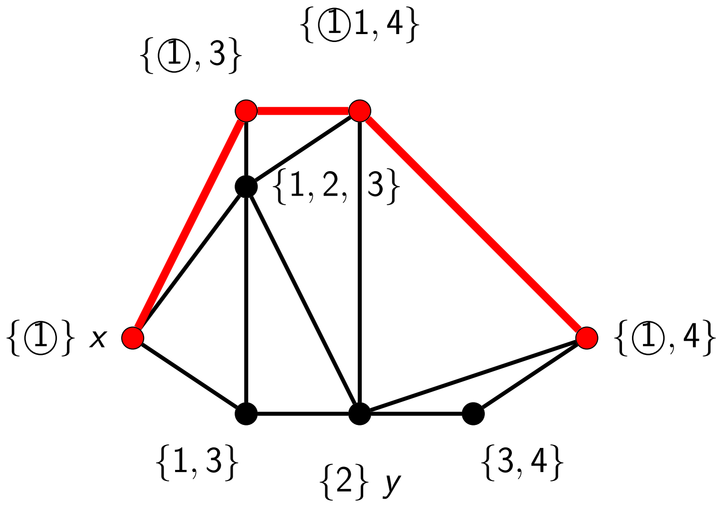
Hartman-Skrekovski list-coloring

We will now proceed to remove the colored path P and update the color lists of the remaining vertices so vertices with neighbors in P will be colored c.

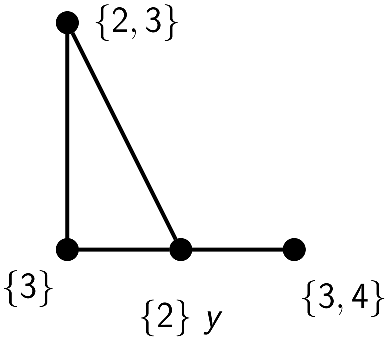
What remains will be several subgraphs with list assignments such that we may separately apply the algorithm to color each.

We will now remove the colored path P. Also, for all vertices v adjacent to a vertex in P we will remove c from $L(v)$.

Hartman-Skrekovski list-coloring

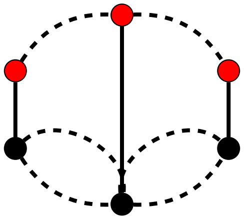


Hartman-Skrekovski list-coloring



Hartman-Skrekovski list-coloring

The key idea that makes this work is that a vertex only ends up with one remaining color in its list when it is in a position to be made x or y in the recursive call.



Implementing Hartman-Skrekovski

There are two main challenges faced in implementing Hartman and Skrekovski's algorithm:

1. removing paths and locating remaining components;
2. tracking where vertices are on the outer face.

Implementing Hartman-Skrekovski

Problem

We must remove a path and locate all the remaining subgraphs in order to make recursive calls.

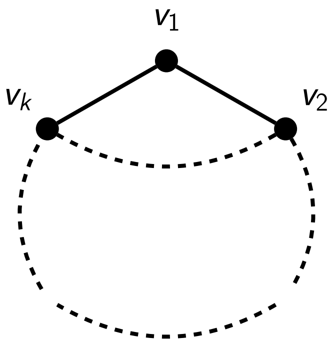
Idea

We remove the path one vertex at a time and immediately make any recursive calls as they become available.

Implementing Hartman-Skrekovski

Suppose we are at the point in the algorithm where we have a cycle $C = v_1 v_2 \ldots v_k$, and a colored path along the cycle.

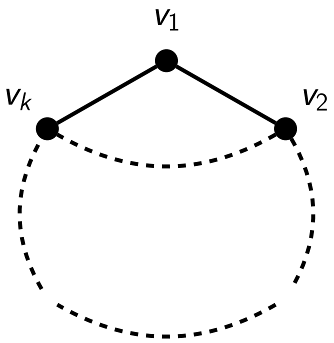
Suppose v_1 is a path vertex we wish to remove. We will iterate through its neighbors counterclockwise, starting with the neighbor v_k and ending with the neighbor v_2.



Implementing Hartman-Skrekovski

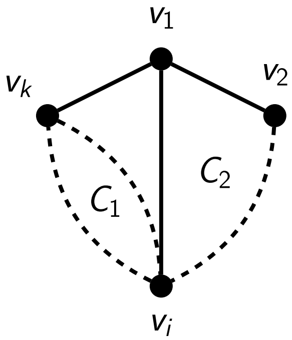
At each neighbor we will remove the given color from their list if necessary.

If a neighbor is an interior vertex we will update vertex properties to note that it is now on the outer face.



Implementing Hartman-Skrekovski

If we hit a neighbor v_i on the outer face, observe v_1 has been completely removed from C_1 then we make the recursive call on the cycle C_1. Since v_1 has been completely removed from C_1 we will continue the process of removing v_1 from the cycle C_2.



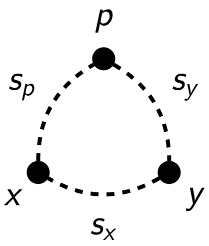
Implementing Hartman-Skrekovski

Problem

We must track where all vertices on the outer face are in relation to x, y, and the path P.

Idea

We mark vertices along the outer cycle to denote which “region” they are in.



Implementing Hartman-Skrekovski

When the path has been entirely removed the segments s_p and s_y are no longer separated by the colored path P.

Moreover, we are about to start coloring a new path along the outer face and must treat s_p as a part of s_y, that is, the segment between x and y clockwise along the outer face.

Walking along and remarking vertices is slow. Instead, we use a disjoint set (or union find) structure to store the segment marks. Then, as two segments are merged we may simply perform a union so both marks are treated the same.

Hartman-Skrekovski time complexity

These techniques combine into an algorithm where we walk through the adjacency list of each vertex a fixed number of times, and perform a fixed number of operations per neighbor.

Therefore the resulting implementation is $\mathcal{O}(n)$.

Boost

For the deliverable implementation I chose to use the C++ Boost Graph Library:

1. Boost is well maintained and kept up to date with the C++ standard;
2. the Boost Graph Library is designed for the implementation of graph algorithms;
3. Boost implements all the standard algorithms for embedding, drawing, and triangulating planar graphs.

Deliverables

Documented implementations of the following algorithms in the Boost Graph Library

1. Poh path 3-coloring with breadth first search;
2. Poh path 3-coloring with faster path finding;
3. Hartman-Skrekovski path 3-list-coloring.

In addition, a writeup describing the correctness and complexity of the implementation details for each algorithm.

Timeline

December 2015: Began reading papers and thinking about planar graphs

January 2016: Talk 1

February 2016: Completed a “first draft” implementation of Poh’s algorithm in Boost

March 2016: Talk 2

April - August 2016: Completed Boost implementations of the Poh and Hartman-Skrekovski algorithms

February 2017 (today!): Talk 3

March 2017: Finalize writeup

April - May 2017: Turn in paperwork and graduate