Simplifying Round-Based Distributed Graph Coloring

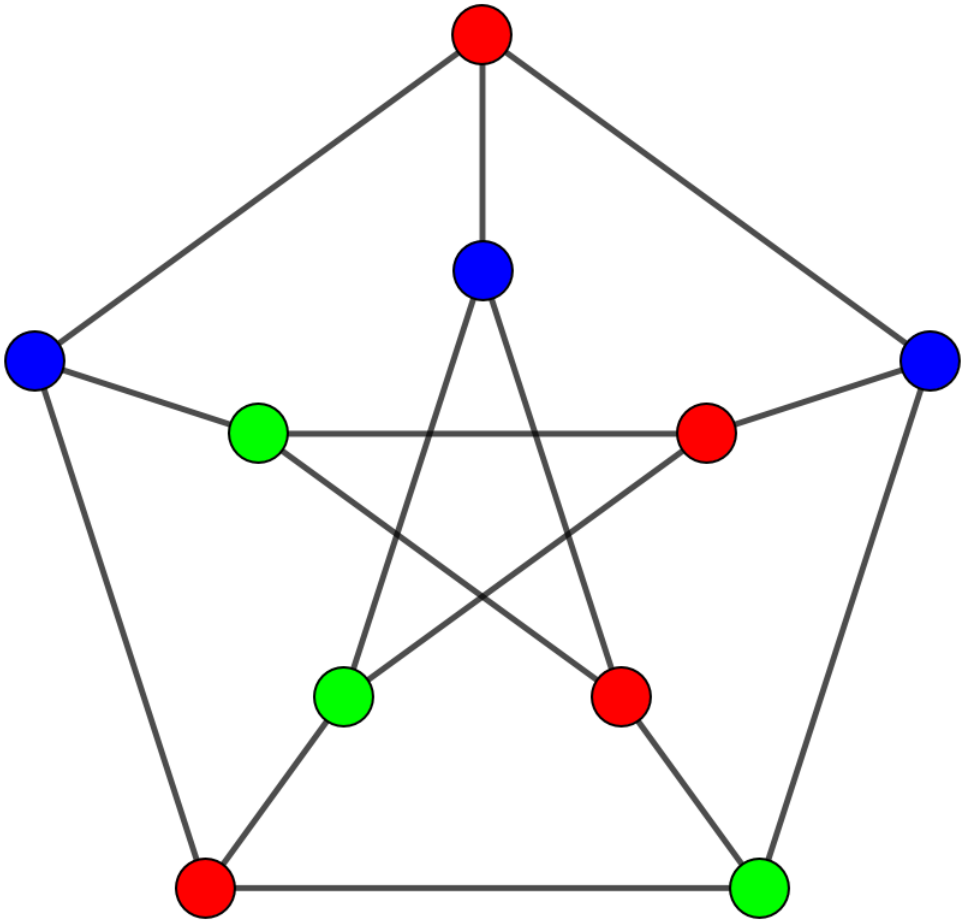
David Worley
School of Electrical Engineering and Computer Science
University of Ottawa, Ottawa, Canada
dworl020@uottawa.ca

What is Graph Coloring?

Graph Coloring is the process of taking a graph and applying a color to each vertex such that no two neighbouring vertices share a color.

The main focuses of this problem are finding the smallest number of colors possible for certain graph classes, or for designing algorithms to find good, but not necessarily optimal, colorings in fast runtime.

An Example Coloring



The image above shows a proper 3-coloring of a 10-vertex, cubic graph.

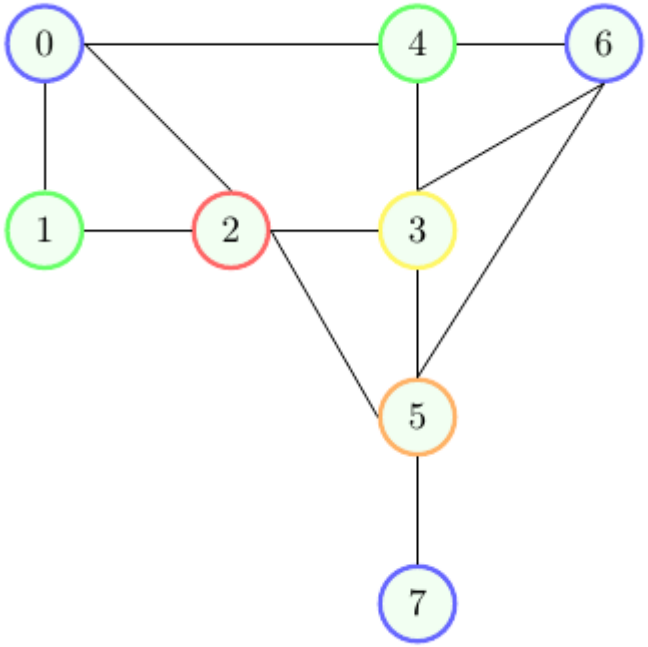
Distributed Graph Coloring

Distributed graph coloring is the process of parallelizing graph coloring algorithms to find proper colorings using as few colors as possible, as fast as possible. This is usually done with *round-based* algorithms to calculate a proper coloring after R rounds.

A coloring with a larger amount of colors can also be useful if we have ways to reduce the number of colors using a separate distributed algorithm that runs as fast or faster

A Lower Bound for Graph Coloring

Let Δ be the maximum degree of the graph, then a Δ +1 coloring can be generated for any graph using a simple greedy sequential coloring algorithm.



A greedy Δ+1 coloring on a graph with 8 vertices and Δ=4

This gives a baseline for a good coloring size on our distributed setting, as we know it is always obtainable.

An Upper Bound by Linial

In 1992, such a bound was established by Linial with an algorithm that generates an $O(\Delta^2)$ coloring in $O(\log^* n)$ time, where $\log^* n$ is the iterated log function.

This gives a suitable upper bound as any graph coloring algorithm running as fast, or slower, than Linial's algorithm can use Linial's algorithm to obtain an $O(\Delta^2)$ coloring.

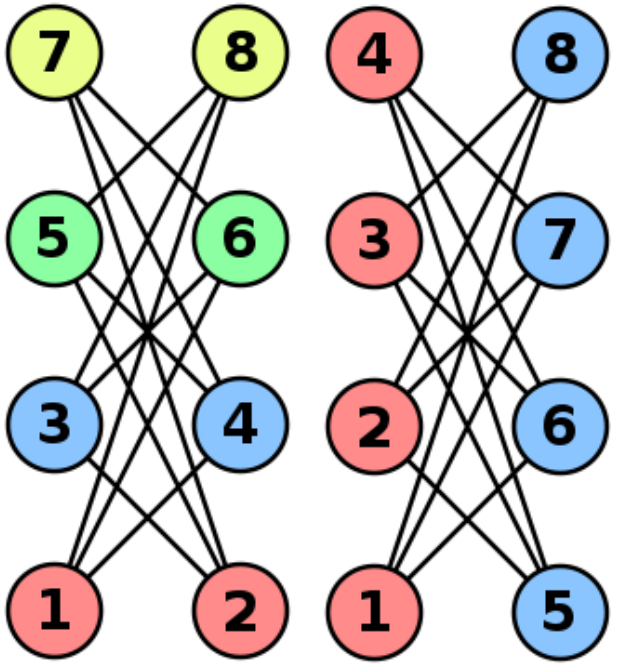
Algorithm Optimality and Runtime

Linial also proved in his paper that any graph coloring algorithm must use at least $\Omega(\log^* n)$ time to color even the simplest graphs.

Since many graph coloring algorithms are round-based, the complexity is expressed with respect to the number of rounds as opposed to runtime with respect to the number of vertices.

Color Reduction

This changes research focus from finding a coloring algorithm to finding a *color reduction* algorithm, that takes in a graph with an input coloring and outputs a graph with a smaller coloring within a certain amount of rounds



For example, the image to the right shows a color reduction from a 4-coloring to a 2-coloring

Maus's Paper

Maus introduces a new flexible coloring algorithm that generalizes many current state of the art algorithms while simplifying the ideas that they use.

This generalization is done using clever use of multiple parameters that allow the algorithm to scale from running 1 round up to Δ, with the size of the coloring scaling inversely

Parameters and Generality

Specifically, Maus's algorithm will take in a value, *k,* specified by the user, and generate an O(Δk)-coloring (16kΔ to be exact) in Δ/k rounds, assuming it was given a graph with an input *m*-coloring and maximum degree Δ.

To view just how flexible this is, consider setting k= Δ. this gives an $O(\Delta^2)$ coloring in 1 round, recreating the results of Linial's famous algorithm. We can similarly get an O(Δ) coloring in O(Δ) rounds by setting $k < \Delta$.

The algorithm is also adaptable enough to calculate a d-defective coloring as well, generalizing many relevant results in distributed graph coloring.

Maus's Algorithm

Algorithm 1: for vertex with color i. Parameters d, k, m, Δ.

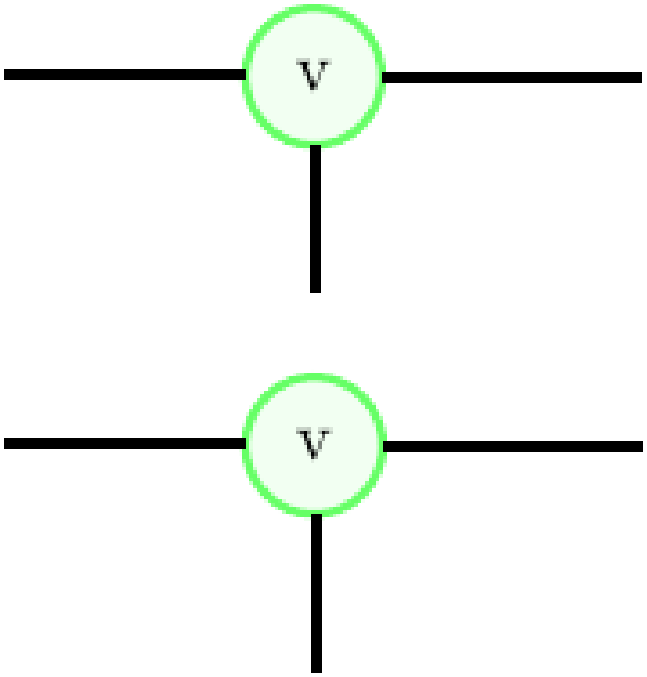
Locally compute:
polynomial $p_i : \mathbb{F}_q \to \mathbb{F}_q$ with q chosen by (1)
sequence s_i: $\left(x \mod k, p_i(x) \mod q\right), x = 0, \ldots, q-1$
Process s_i in disjoint batches B_j of size k, for $j = 1, \ldots, \lceil \frac{q}{k} \rceil$
Try the colors in batch B_j (in a single round)
if $\exists \left(d\text{-}proper\ c \in B_j\right)$ **then** adopt c, join P_j, and **return**;

The algorithm generates the above polynomials of degree f=ceil($\log_\Delta m$) from a prime field of size q, where q is the smallest prime > $2\Delta\log_\Delta m$.

This choice of f and q guarantees that the sequences for any two vertices will only conflict with f tuples, and since the algorithm will test more than f tuples, a proper coloring is guaranteed.

Coloring a Vertex With Maus's Algorithm

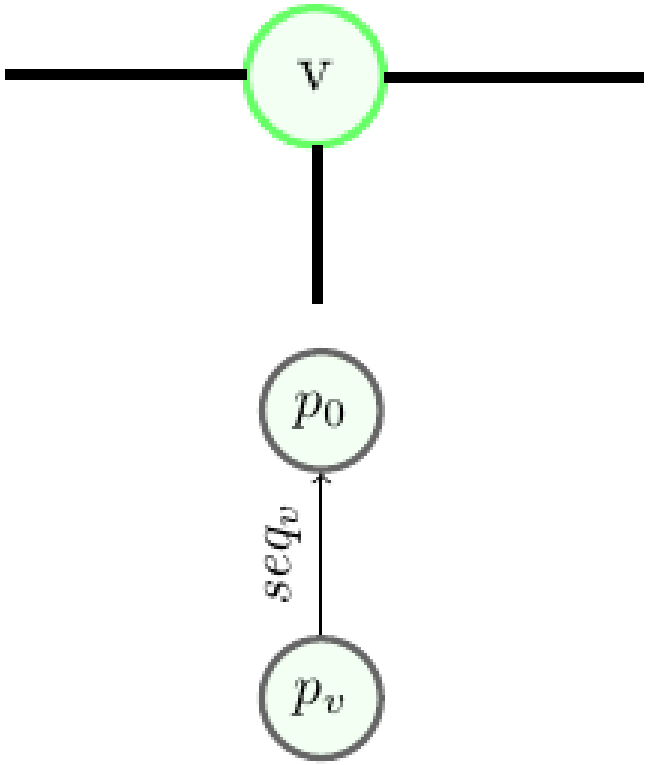
If a vertex *v* has input color m, we sample polynomial p_m for *k* values (p(x) for x=0,…,k-1)



$Seq_v = (p_m(0)\%q, p_m(1)\%q, \ldots, p_m(k-1)\%q)$

So take $p_m = a_1x_1 + a_2x_2 + \ldots + a_fx_f$ where $f=\log_\Delta m$

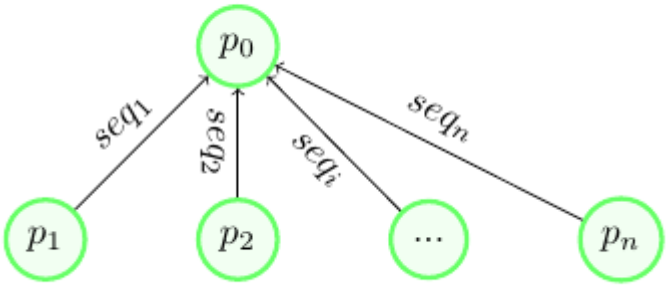
and $(a_0, a_1, \ldots, a_f)$ corresponds to the m^{th} polynomial



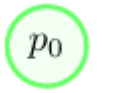
Calculate the sequence of *v,* modding the elements by q as p_m is a polynomial from a finite field.

The processor working on vertex *v* will send this sequence to the main processor for conflict checking.

A Round of Maus's Algorithm

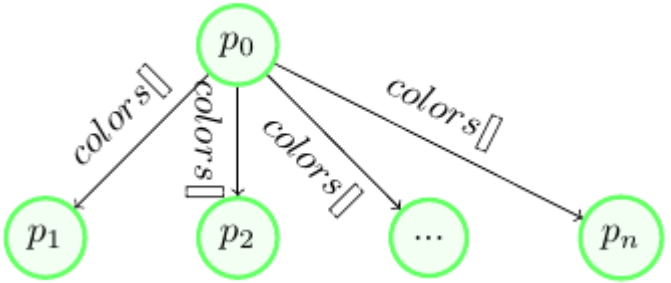


Step 1: Each processor calculates the sequence for its nodes and sends it to the main processor



working

Step 2: Main processor uses the sequences to check for conflicts and create the new coloring



Step 3: Main processor redistributes the new coloring so each processor can calculate new sequences

Algorithm Implementation

The algorithm was implemented in C++ using MPI and involved generating the polynomials for each input color.

Then, each processor will calculate the color sequence for each vertex it was assigned and send these sequences to a main processor. This main processor will attempt to recolor vertices with their sequences, marking them as inactive when they've been successfully colored.

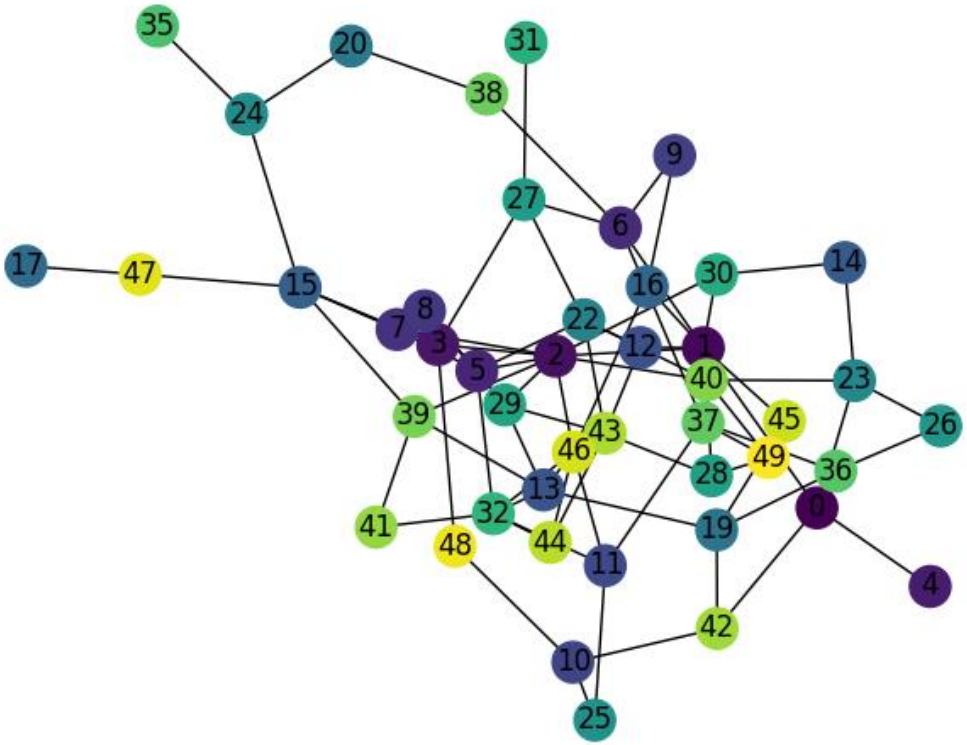
This process is repeated for the required number of rounds, and the unique properties of polynomials from a prime field guarantee that all vertices will be colored upon completion.

Data and Parameters

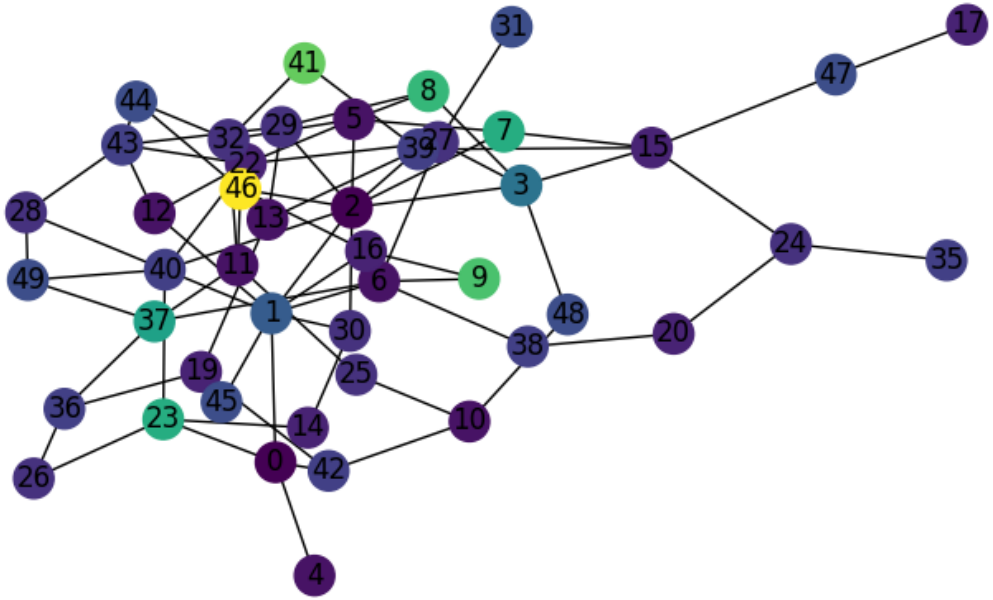
A small Python program to generate random graphs was used to generate input colorings for the implementation. These input colorings would randomly generate an edge set, calculate Δ, and generate a coloring, then output the contents for testing.

The coloring initially chosen was to use each vertex's ID as its color for an input |V|-coloring and k was chosen such that $1 <= k <= 4\Delta$

Before and After Coloring



A graph with an input 46-coloring



The graph after 6 rounds of Maus's Algorithm, now with a 15-coloring

The above graphs are isomorphic, though the graph visualization library used displays them slightly differently

Experimental Results

Number of Vertices	Max Degree (Δ)	Number of Rounds	Colors in Input Coloring	Colors in Output Coloring	Time Elapsed (s)
10	6	4	10	8	0.115
100	13	10	100	26	0.692
250	10	16	250	55	1.553
500	12	19	500	76	1.859
1000	13	20	1000	91	53.562*

*The long runtime of the 1000 vertex graph is likely due to the implementation of the graph as an adjacency matrix as opposed to an adjacency list

Future Work

1. The runtime of the algorithm could be improved by using an adjacency list instead of an adjacency matrix to minimize the time it takes to check for conflicts with neighbours

2. Due to needing the most up to date coloring during its conflict checking, the algorithm works off of a main processor that collects the color sequences after each round and checks conflicts, this creates a bottleneck that will be especially noticeable as the number of processors increases.

3. Adapting the implementation to handle d-defective colorings (where the current algorithm uses d=0) would add extra flexibility and generality to the algorithm. This involves significant changes to the conflict checking methodology.

Summary

Overall Maus's algorithm shows strong results in reducing large input colorings on graphs. The algorithm can recreate the results of various algorithms using a clever choice of parameters and is a strong, flexible algorithm that simplifies many state of the art results in distributed graph coloring.

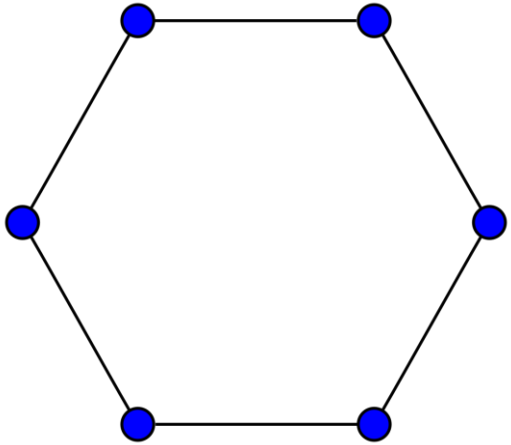
Further, additional work to improve the implementation should show strong runtime speedups for large graphs as well.

Questions For Audience

1. Why is research focusing on color reduction algorithms instead of coloring algorithms?

2. Why do we have a lower bound of $\Delta+1$ for colorings instead of something smaller/bigger?

3. Is it possible to color an n-cycle in constant time?



Thanks!

Any Questions?