Simplifying Round-Based Distributed Graph Coloring

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

December 7, 2021

Abstract

Distributed graph coloring is a large and popular area of distributed algorithms research. With many applications in social network analysis [8] and job scheduling [10][6], it is also a very impactful area of research in industry. This has led to large research interest and, as a result, plenty of distributed graph coloring algorithms. These algorithms differ in their purposes, graph classes, or design ideologies, but one common aspect between most of these algorithms is that they are *round-based*. This means they function by coloring some vertices of the graph in each round, over a specified number of rounds, until a coloring is created. Unsurprisingly, the work done in a round can differ greatly between different algorithms and is frequently quite complex. This paper implements and discusses an algorithm designed by Maus [9] that simplifies many of these more complicated algorithms using a generic, round-based algorithm that can subsume the results of many other state of the art coloring algorithms through clever setting of parameters to guarantee a proper coloring when the algorithm terminates, regardless of how many rounds it runs.

1 Introduction

Graph coloring is a well-known problem in the area of graph theory that asks, "Given a graph G, how can we color the vertices of G such that no two adjacent vertices share the same color?" The optimization aspect of the problem is finding a proper coloring for the graph with the minimal number of colors possible.

Distributed graph coloring is the process of parallelizing graph coloring algorithms, typically by having processors attempt to color nodes or generate colors to be tested by a main processor, this allows much faster colorings on large graphs, typically at the cost of using more colors than are necessary. Thus, the goal of distributed graph coloring is to create algorithms that find colorings that may not be optimal, but are as fast as possible and use as few colors as possible.

These colorings can be found in many different ways for different classes of graphs, but one frequently overlooked aspect in algorithm design is simplicity. In this paper, the distributed graph coloring algorithm proposed by Maus in [9] will be discussed, compared with other results, and implemented and tested. This is because of the simplicity and flexiblity of Maus's algorithm in comparison to many other state of the art algorithms. This look will provide insight into how Maus's algorithm fits within the field of distributed graph coloring as well as looking at the merits of the algorithm in practice.

Section 2 will review the history of distributed graph coloring algorithms, discussing historic results, current state of the art research, and concluding with Maus's algorithm. Section 3 will motivate the importance of this algorithm further by highlighting the complexity of other state of the art algorithms in comparison to Maus's. Section 4 will highlight the theory behind Maus's algorithm and how it works to subsume so many critical results. In section 5, the algorithm implementation will be discussed to lead into section 6, covering the results from said implementation. Section 7 concludes the paper.

2 Literature Review

2.1 Models and Bounds for Distributed Graph Coloring

Currently, the majority of work in distributed graph coloring focuses on finding a k-coloring such that $\Delta + 1 \leq k \leq O(\Delta^2)$, where Δ is the maximum degree of a vertex. The lower bound results from the common fact that any graph where the maximum degree is Δ can be colored with $\Delta + 1$ colors, with simple greedy algorithms able to find such a coloring. The upper bound is a result of Linial who, in 1992, proposed a model for distributed graph algorithms called the LOCAL model [7]. This model is round-based, allowing for each vertex to transmit information to each of its neighbours at the end of each round. Using this model, Linial shows that an $O(\Delta^2)$ coloring can be generated in 1 round in $O(\log^* n)$ time, where $\log^* n$ is the iterative logarithm, or how many times the log function must be applied to n until the result is 1.

This LOCAL model proves to be the simplest to work with, though other common models are also important in the literature. These include the SET-LOCAL model, sometimes called the weak LOCAL model, in which vertices do not have IDs, and so cannot distinguish between the messages of its neighbours. The main assumption algorithms work off if within the SET-LOCAL model is that the algorithm *begins* with a proper coloring [5]. Another common model is the more restrictive CONGEST model, in which the vertices can only transmit $O(\log n)$ data to each neighbour per round. This is particularly restrictive as vertices are typically identified using bit IDs of size $O(\log n)$.

2.2 One Round Color Reductions

While the $O(\Delta^2)$ bound proposed above is large, this is acceptable due to the presence of color-reduction algorithms that can be used to reduce the amount of colors in the coloring. With Linial's algorithm acting as a preprocessing step, color reduction algorithms can focus on reducing the $O(\Delta^2)$ coloring instead. In fact, making this distinction between coloring algorithms and color reduction algorithms is unnecessary, as any coloring algorithm can be considered as a color reduction algorithm from an input $|V|$-coloring, where V is the set of vertices of the input graph (and so each vertex has its own color). These color reduction algorithms can make use of the reduced $O(\Delta^2)$ coloring well to create a smaller coloring in as few rounds as possible, making this output coloring from Linial's algorithm still valuable. Alternatively, color reduction algorithms can focus on specific settings in which a coloring can be reduced greatly in *one* round. Linial himself once again sets strong foundations for this area, presenting an algorithm to reduce a k-coloring of a graph to a $O(\Delta^2 \log m)$-coloring in a single round under his model [7].

With these foundations, much research focuses on one-round color reduction algorithms that can reduce the colors significantly for certain classes of graphs. For example, one-round

color reduction algorithms were developed for directed paths that can reduce k-colorings to 3-colorings in $\frac{1}{2}\log^* n + O(1)$ rounds [4]. This was later reduced into a tight bound of $\frac{1}{2}\log^* n$ [11]. This result is important due to its usefulness as a subroutine of other distributed graph coloring algorithms. Other improvements look towards specific classes of graphs or reductions from k-colorings under some set of assumptions about k. For example, Maus proposes a one-round coloring algorithm that reduces an k-coloring to a $m(\Delta - m + 2)$-coloring, given that $k \geq m(\Delta - m + 3)$, removing m colors from the coloring, where $1 \leq m \leq \Delta/2 + 3/2$ [9].

2.3 Improving Coloring Algorithms

With strong color reduction algorithms, efficient colorings can be obtained through the use of a coloring algorithm followed by repeated color reductions to said graph. Considering our lower bound of $\Delta+1$, its natural to ask how many rounds a coloring algorithm requires to result in a $(\Delta+1)$-coloring. In 1993, Szegedy and Vishwanathan showed that for algorithms that were locally iterative, a $(\Delta+1)$-coloring algorithm requires $\Omega(\Delta \log \Delta + \log^* n)$ rounds, barring the existence of a special type of coloring whose reduction could be done very efficiently [12]. A locally-iterative algorithm is one where each vertex chooses its next color based solely on the colors of its local neighbourhood, so this bound applied to a large class of the algorithms within the field. For over 25 years, no algorithm could make use of such a special coloring and thus could not beat the proposed *SV Barrier* lower bound. It wasn't until 2021 that this barrier was broken, with a locally iterative algorithm that could compute a $(\Delta + 1)$-coloring with runtime $O(\Delta + \log^* n)$ [2].

2.4 Maus's Algorithm

The algorithm proposed by Maus in [9] is a round based color reduction algorithm scaling between the two bounds presented above as follows. For a given integer $1 \leq k \leq O(\Delta)$, the algorithm generates an $O(\Delta k)$-coloring in $O(\Delta/k)$ rounds through a trial based reduction of an input coloring, like one given by Linial's algorithm. Each vertex of the graph will compute a sequence of k colors and attempt to color itself with this sequence, stopping on the first one that is found to be without conflict. If no colors in the first k are without conflict, then k more are tested in the subsequent round. This process repeats until all vertices are colored.

Since Δ is known and k can be chosen before the algorithm is run, k can always be selected proportionally to Δ to make the algorithm scale from a $O(1)$-round reduction algorithm that generates an $O(\Delta^2)$ coloring, subsuming Linial's algorithm, or an $O(\Delta)$-round algorithm to compute an $O(\Delta)$ coloring. The algorithm is also general enough for other types of graph coloring problems, such as d-defective colorings where vertices are allowed to have the same color as at most d of their neighbours.

3 Problem Statement

The algorithms mentioned above, as well as the many more discussed by Barenboim and Elkin in their book [1], are all fast and effective algorithms, but many suffer from inflexibility and complex implementations. Since they are solving specific problems, the former is not a major issue, but the complexity of the algorithms can make the research inaccessible, hard to advance from, or difficult to make use of in practice. For a concrete

example consider the algorithm by Barenboim, Elkin, and Kuhn from [3], a snippet of the algorithm's psuedocode is presented below for the sake of comparison. This procedure is part of an algorithm used to generate a $\Delta + 1$ coloring in $O(\Delta)$ time. While the lack of context may make the comparison unfair, following through the psuedocode of Maus's algorithm presented in section 4 should highlight well how they differ in complexity.

Algorithm 1 Procedure Defective-Color(p, q) (algorithm for a vertex v).

Input: A graph G, and two parameters p, q, such that $p^2 < q$.
Output: $O(\frac{\log \Delta}{\log(q/p^2)} \cdot \Delta/p)$-defective p^2-coloring of G

1: $\varphi :=$ color G with $(c \cdot \Delta^2)$ colors
2: $\chi := c \cdot \Delta^2$ /* the current number of colors */
3: $i = 0$ /* the index of the current iteration */
4: **while** $\chi > p^2$ **do**
5: **if** $\chi < q$ **then**
6: $j := 1$
7: **else**
8: $j := \min\{\lceil \varphi(v)/q \rceil, \lfloor \chi/q \rfloor\}$
9: **end if**
10: set V_j to be the set of v
11: $\psi_j(v) := \varphi(v) - (j-1) \cdot q$ /* $\psi_j(\cdot)$ is an $(i \cdot \lfloor \Delta/p \rfloor)$-defective $(2q)$-coloring of $G(V_j)$ */
12: $\varphi'_j :=$ Refine$(G(V_j), \psi_j, p)$
13: $\varphi(v) := \varphi''(v) := \varphi'_j(v) + (j-1) \cdot p^2$
14: $\chi := (\max\{\lfloor \chi/q \rfloor, 1\}) \cdot p^2$ /* $\varphi(\cdot)$ is an $(i \cdot \lfloor \Delta/p \rfloor)$-defective χ-coloring of G */
15: $i := i + 1$
16: **end while**
17: return φ

While these algorithms are correct, efficient, and effective, the ease of use for an algorithm must also be considered, especially when considering algorithms for use in industry. When given two algorithms solving the same problem with approximately the same runtime, any sensible programmer will choose the simpler one. This means that a simpler algorithm is still of great practical value, especially when it can also provide flexibility.

4 Maus's Algorithm

Maus's Algorithm solves this problem by proposing a simple, general round-based algorithm. This algorithm is flexible in the size of the coloring it creates, the number of rounds achieved by the coloring, and the type of coloring it can generate. It accomplishes all of this without sacrificing runtime efficiency as well. The simplicity of the algorithm can be seen from its psuedocode for coloring a single vertex, shown below.

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: $(x \mod k, p_i(x) \mod q)$, $x = 0, \ldots, q-1$
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 $\exists$ (d-*proper* $c \in B_j$) **then** adopt c, join P_j, and **return**;

4.1 Parameters

The algorithm's flexibility is largely a result of its parameters. The algorithm requires parameters k, defining the tradeoff between coloring size and number of rounds, and d, representing how many adjacent nodes can have the same color (so a proper coloring has $d = 0$). Both of these parameters are chosen by the user before the algorithm is run, and the algorithm determines other parameters based off of the size of the graph's input coloring, m, and maximum degree, Δ.

With k, d, Δ, and m known before the algorithm begins processing the graph, more parameters are calculated. X is the other parameter controlling the tradeoff between rounds and coloring size, as the algorithm will run $R = \lceil X/k \rceil$ rounds to compute a $X * k$ coloring, with $X = 4(\Delta/(d+1))\lceil \log_{\Delta/(d+1)} m \rceil$. Note that this is calculated independently of k, and relies only on the input graph's data, this leaves full flexibility in the tradeoff of coloring size versus rounds in the user's hands through careful choice of k.

Two more parameters, f and q, are calculated as follows.

$$f = \lceil \log_{\Delta/(d+1)} m \rceil \text{ and } 2f < q < 4f\Delta/(d+1)$$

where q is selected to be prime. This q always exists due to Bertrand's postulate, a result from number theory stating there always exists a prime with $n < p < 2n - 2$ for any $n > 3$. These latter two parameters are used in calculating the color sequence for each vertex to try.

4.2 Coloring

With all parameters defined, the algorithm achieves its coloring by generating m distinct polynomials of degree f from the field $\mathbb{F}_q$. The simplest way to do this is to represent a polynomial of degree f by a tuple of it's coefficients. Explicitly, we model the polynomial $p(x) = a_0 + a_1 x + .. + a_f x^f$ as $(a_0, a_1, ..., a_f)$. Then, for each color i in our input m-coloring, we choose $p_i(x)$ be the ith tuple in lexicographical order, excluding the zero polynomial. Each vertex with color i will then sample $p(x)\%q$ for $x = 0, ..., k-1$ and save the values as its *color sequence*.

Once each vertex of the graph has it's color sequence, it samples the values in its sequence, checking for conflicts with its neighbours. If the vertex finds a color in it's sequence that does not conflict with the color of it's neighbour (or with $d + 1$ of its neighbours in the $d > 0$ case), it permanently colors itself with that node and is not processed again. If a vertex goes through it's entire sequence without being colored, it keeps the color it had going into the round and tries again next round.

This coloring sequence is guaranteed to result in a valid coloring due to the distinct m polynomials with which the sequences are calculated. Given two polynomials of degree f from a finite prime field $\mathbb{F}_q$, f_1 and f_2 intersect no more than f times. So any two color sequences will only conflict in at most f locations. A case analysis shows that for two fixed vertices u and v, there are at most f tuples causing a conflict with u and v, since there are more unique tuples than this over the course of the algorithm's runtime due to our choice of f, there will be a round in which u and v can be colored without conflict. For more details on this case analysis see [9].

4.3 Generalizing Other Results

The generality of Maus's algorithm has been highlighted throughout this paper, and in this section two specific examples of this generality will be highlighted. First, consider Linial's famous algorithm to generate an $O(\Delta^2)$ coloring in one round. Now, consider setting $d = 0$ in Maus's algorithm, this forces $X = 16\Delta$. Setting $k = X$ now grants an algorithm that can generate an $X * k = 256\Delta^2$-coloring in $X/k = 1$ round. Thus, Maus's algorithm can also generate an $O(\Delta^2)$ coloring in one round.

Similarly, consider the psuedocode outlined in section 3 to calculate a $\Delta + 1$ coloring in $O(\Delta)$ time. This algorithm involves the calculation of defective colorings for every node which can then be used to find the desired $\Delta + 1$ coloring. We instead could set $d = 0$ once more, giving the same $X = 16\Delta$. Now, for any value of k we obtain a $16\Delta k$-coloring in $16\Delta/k$ rounds, giving the same $O(\Delta)$ coloring in $O(\Delta)$ rounds.

More results regarding algorithms that can be generalized in the case of $d > 0$ are discussed by Maus in [9].

5 Implementation

Maus's Algorithm was implemented using C++ and MPI using a main/worker architecture for processing the sequences. The main processor read in the graph, its parameters, and the initial coloring and distributes this info to the other processors. Each worker processor p handled the vertices in its mod class, i.e. processor 3 handled vertex 3, $3+p$, $3+2p$, etc. Since processor 0 was chosen to be the main processor, the vertices in its mod class were split between the worker processors.

Once each processor had the graph and coloring, the m polynomials were generated so that every worker processor had access to them. With all needed information, each processor goes over all of the vertices it was assigned. For each vertex v with input color i, the processor would choose the polynomial p_i, generate a sequence of k colors to test as described above, and send the sequence back to the main processor.

The main processor waits to receive the sequence for each vertex, and then iterates over every vertex in the graph and attempts to color it with a value from its color sequence. If there is a conflict, it moves on to the next value in the sequence and repeats until either a valid color is found or all values in the vertex's sequence have been tried. If a valid color is found, the vertex is given that color and marked as inactive so it is not processed again, if the end of the sequence is reached, the node retains its old color and is tested again in the next round.

This process is repeated for $\lceil X/k \rceil$ rounds, and once done the final round concludes the main processor outputs the resulting coloring from the graph.

6 Experimental Evaluation

6.1 Input and Output Processing

The algorithm was tested on graphs of numerous sizes. These graphs were generated using a custom Python script that, given integers V and E, generates a connected graph with V vertices and at least E edges. This script then calculates the maximum degree, Δ, of the graph and outputs the graph information so that it can be read in by the main algorithm.

Once the algorithm is run and the coloring is outputted, a separate Python script loads the information outputted by Maus's algorithm and verifies that the resulting coloring is valid. For graphs under a specific size, a picture of the graph with its reduced coloring is also produced.

6.2 Results

The table below displays the results of the algorithm on graphs of various sizes. All tests below were run on 6 processors, meaning for a graph with V vertices, each processor handled $V/5$ of the vertices in the graph, with the other processor being the main processor doing the conflict checking. For each test case below, k is chosen to be the minimum value between 5 and $X/2$.

Number of Vertices	Max Degree (Δ)	Number of Rounds	Input Coloring Size	Output Coloring Size	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

Results from the graph visualizer on a small graph can be seen in fig.4 as well. This figure displays a 46 vertex graph with an input 46-coloring and maximum degree 8. Maus's algorithm was run on this graph for 6 rounds to reduce it to a valid 16-coloring in 0.63 seconds.

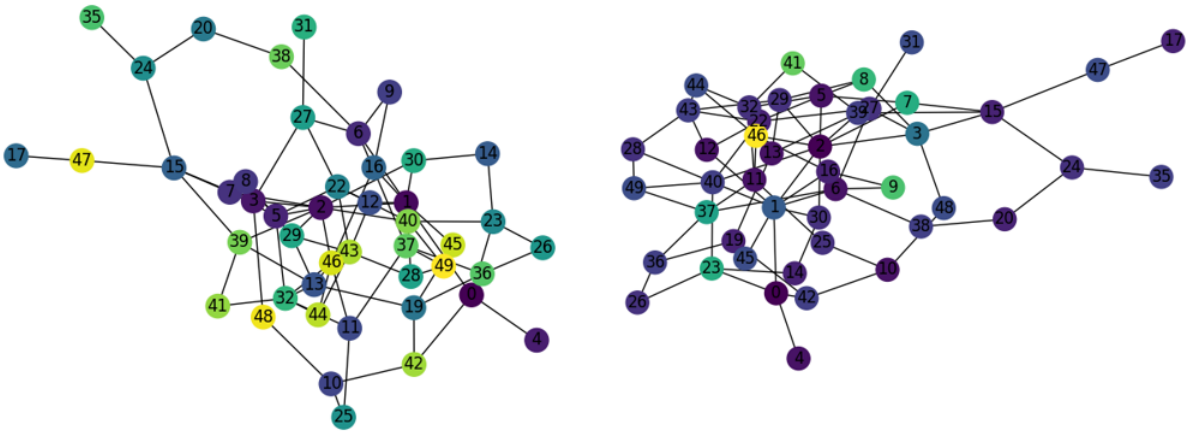

Figure 1: A 46-vertex graph (left) and its reduced 16-coloring (right) after running Maus's Algorithm.

6.3 Runtime Considerations

This shows strong reductions from the input colorings for graphs of various size. For the smaller test cases the algorithm ran very quickly, but the larger test cases show that

this runtime quickly increases with the size of the graph. This is mainly due to the specific implementation. The program models the graph as an adjacency matrix, meaning conflict checking is much slower as the algorithm must look through the adjacency matrix instead of an adjacency list. This could be offset by using an adjacency list representation and careful memory management so this graph representation can be sent over MPI without memory related issue.

This would not be a perfect solution to this issue however, as the graph continues to scale in size the conflict checking done by the main processor would still be the bottleneck of the algorithm. If this process could be distributed as well, so that each processor can process nodes and check for color conflicts, then the runtime of the algorithm would dramatically increase. This seems difficult to accomplish in practice though, as whenever a vertex is colored, the other processors would need to be notified so that they are working with the most up to date coloring.

7 Conclusions

7.1 Conclusion

The algorithm proposed by Maus in [9] was introduced to simplify current algorithms within the field by providing a general and flexible distributed graph coloring algorithm. The implementation of this algorithm, discussed above, does this with strong results and small colorings are obtained from this algorithm, even when given large input colorings. Even with a non-optimal implementation, the runtime of the algorithm is sufficiently short for small and medium size graphs.

7.2 Summary of Contributions

This paper sought to implement the algorithm designed proposed by Maus in [9], test it to verify the correctness of its result, and thoroughly understand and explain the value of its contributions to the field of distributed graph coloring. The strengths and effectiveness of this algorithm are also displayed through results of the implementation, as well as comparing it to others within the field of distributed graph coloring.

7.3 Future Research

Future research and improvements could take on of two forms. First, theoretical research could be done to improve Maus's base algorithm further, either generalizing to subsume even more results or finding different ways to resolve color checking to achieve faster runtime. The other area of improvement would be to resolve issues in the current implementation. Adapting the algorithm to use a more space and time efficient graph representation would provide a speedup on larger graphs. Further improvements could be done by identifying ways to change the communication rounds and offload work from the main processor by having every processor be involved in calculating color sequences and testing for color conflicts with those sequences. This would allow the algorithm to scale much better as the number of processors increases due to removing the concept of a main node as well as the bottleneck that comes with it.

References

[1] L. Barenboim and M. Elkin. Distributed graph coloring: Fundamentals and recent developments. *Synthesis Lectures on Distributed Computing Theory*, 4, 07 2013.

[2] L. Barenboim, M. Elkin, and U. Goldenberg. Locally-iterative distributed $(\delta + 1)$-colouring below szegedy-vishwanathan barrier, and applications to self-stabilization and to restricted-bandwith models. *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 437–446, 2018.

[3] L. Barenboim, M. Elkin, and F. Kuhn. Distributed $(\delta + 1)$-coloring in linear (in δ) time. *SIAM Journal on Computing*, 43:72–95, 01 2014.

[4] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, 1986.

[5] D. Hefetz, F. Kuhn, Y. Maus, and A. Steger. Polynomial lower bound for distributed graph colouring in a weak local model. *Proc. of the 30th International Symp. on Distributed Computing*, pages 99–113, 2016.

[6] R. Lewis and J. Thompson. On the application of graph colouring techniques in round-robin sports scheduling. *Computers and Operations Research*, 38(1):190–204, 2011. Project Management and Scheduling.

[7] N. Linial. Locality in distributed graph algorithms. *Society for Industrial and Applied Mathematics*, 21(1):193–201, 1992.

[8] M. Marei M. Mosa, A. Hamouda. Graph coloring and aco based summarization for social networks. *Expert Systems with Applications*, 74:115–126, 2017.

[9] Y. Maus. Distributed graph colouring made easy. *SPAA '21*, pages 362–372, 2021.

[10] P. Giaccari N. Zufferey, P. Amstutz. Graph colouring approaches for a satellite range scheduling problem. *Journal of Scheduling*, 11(4):263–277, Aug 2008.

[11] J. Rybicki and J. Suomela. Exact bounds for distributed graph colouring. *SIROCCO 2015*, 9439:46–60, 2015.

[12] M. Szegedy and S. Vishwanathan. Locality based graph coloring. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '93, page 201–207, New York, NY, USA, 1993. Association for Computing Machinery.