Proceedings of the IEEE IPDPS-2002 Workshop on Advances in Parallel and Distributed Computational Models, Orlando, Florida, April 2002.

Fault Tolerant Distributed Coloring Algorithms That Stabilize in Linear Time

Stephen T. Hedetniemi
Department of Computer Science
Clemson University
Clemson, SC 29634–0974, USA

David P. Jacobs
Department of Computer Science
Clemson University
Clemson, SC 29634–0974, USA

Pradip K. Srimani
Department of Computer Science
Clemson University
Clemson, SC 29634–0974, USA

Abstract

We propose two new self-stabilizing distributed algorithms for proper $\Delta+1$ (Δ is the maximum degree of a node in the graph) coloring of arbitrary system graphs. Both algorithms are capable of working with multiple types of demons (schedulers) as is the most recent algorithm in [1]. The first algorithm converges in $O(m)$ moves while the second converges in at most n moves (n is the number of nodes and m is the number of edges in the graph) as opposed to the $O(\Delta \times n)$ moves required by the algorithm [1]. The second improvement is that neither of the proposed algorithms requires each node to have knowledge of Δ, as is required in [1]. Further, the coloring produced by our first algorithm provides an interesting special case of coloring, e.g., Grundy Coloring [2].

1 Introduction

Robustness is one of the most important requirements of modern distributed systems. Different types of faults are likely to occur at various parts of the system. These systems go through the transient states because they are exposed to constant change of their environment. In a distributed system the computing elements or nodes exchange information only by message passing. One of the goals of a distributed system is that the system should function correctly in spite of intermittent faults. In other words, the global state of the system should ideally remain in the legitimate state. Often, malfunctions or perturbations bring the system to some illegitimate state, and it is desirable that the system be automatically brought back to the legitimate state without the interference of an external agent. Systems that reach the legitimate state starting from any illegitimate state in a finite number of steps are called *self-stabilizing systems* [3, 4]. This kind of property is highly desirable for any distributed system, since without having a global memory, global synchronization is achieved in finite time and thus the system can correct itself automatically from spurious perturbation or failures.

2 Self-Stabilization – A Paradigm for Distributed Fault Tolerance

There are different existing approaches towards designing fault tolerant software like N-version programming, recovery blocks, consensus recovery blocks, etc. But design of fault tolerant software or algorithms has been traditionally investigated in the context of particular applications, system architectures as well as specific technologies. As a result we have different models and techniques for different applications and there is no simple way to verify these fault tolerant systems. Also, the basic approach in designing fault tolerant software has traditionally been to mask or tolerate design faults in the software itself.

The common approach to design the fault tolerant systems is to mask the effects of the fault; but, fault masking is not free; it requires additional hardware or software and it considerably increases the cost of the system. This additional cost may not be an economic option, especially when most faults are transient in nature and a temporary unavailability of a system service is acceptable. *Self-stabilization* is a relatively new way of looking at system fault tolerance. It provides a "built-in-safeguard" against "transient failures" that might corrupt the data in a distributed system.

In this paper we propose two simple, yet very efficient, self-stabilizing algorithms that find proper colorings in an arbitrary graph. Self-stabilizing algorithms for proper coloring of graphs has been studied in the literature

[8, 9, 10, 11, 1]. For example, [9] gives a self-stabilizing algorithm to 2-color any bipartite graph. And in [8], the authors describe a self-stabilizing algorithm to 6-color any planar graph. Obviously the bipartite algorithm is optimal and the planar algorithm is close to optimal, given that all planar graphs are 4-colorable. But, the authors do not provide any complexity analysis. Only a recent paper of Gradinariu and Tixeuil [1] gives a self-stabilizing algorithm that finds in any graph a $(\Delta + 1)$-coloring, and which has a worst-case analyses of $O(\Delta \times n)$ moves; one drawback of this algorithm is that each node must know the value of Δ. Our first $(\Delta + 1)$-coloring algorithm, Algorithm 3.1, has three advantages: (1) no node must know the value of Δ (as opposed to the requirement in the algorithm of [1]); (2) the algorithm converges in $O(m)$ moves (m is the number of edges in the graph) compared to $O(\Delta \times n)$ moves of [1]; (3) the coloring obtained by the algorithm is always a Grundy Coloring [2] of the graph. We then propose another $(\Delta + 1)$-coloring algorithm, Algorithm 3.2; this algorithm stabilizes in at most n moves. Algorithm 3.2 appears to be faster than other known self-stabilizing $(\Delta + 1)$-coloring algorithms.

The objective of self-stabilization is (as opposed to mask faults) to recover from failure in a reasonable time and without intervention by any external agency. Self-stabilization is based on two basic ideas: first, the code executed by a node is re-entrant and incorruptible (as if written in a fault resilient memory) and transient faults affect only data locations; second, a fault free system behavior is usually checked by evaluating some predicate of the system state variables. Every node has a set of local variables whose contents specify the local state of the node. The state of the entire system, called the *global state*, is the union of the local states of all the nodes in the system. Each node is allowed to have only a partial view of the global state, and this depends on the connectivity of the system and the propagation delay of different messages. Yet, the objective in a distributed system is to arrive at a desirable global final state (legitimate state).

The set of all possible global states is divided into two disjoint sets: the error free or the *legitimate* states, and the erroneous states or the *illegitimate* states. The self-stabilization paradigm assumes that each node computes a predicate of its own local state and its neighbors' states (a predicate that would use other node states than the neighbor node states requires an underlying routing protocol to be implemented). When an inconsistent state is detected, a common approach in centralized systems is to force the system to a well defined state by a hardware reset or a power-cycle. This is often not an option in distributed systems that may cover a large geographical area. In a self-stabilizing system, when a node detects a local inconsistency, it takes a local action (a node can modify only its own states) in an attempt to correct the error. This node becomes locally consistent, but some of its neighbor nodes may become inconsistent (which were locally consistent before the action) and this ripple effect may propagate the entire system. A system state is globally legitimate when each node is locally legitimate or consistent. An algorithm is self-stabilizing if for any initial illegitimate state it reaches a consistent state after a finite number of node moves. A distributed system running a self-stabilizing algorithm is called a *self-stabilizing system*.

In general, a node is triggered into action when a local inconsistency is detected; hence, in a legitimate system state no node may move. However, there are many services (provided by distributed systems) that require the system to change its state continually. A classical example is the token circulation for a distributed mutual exclusion algorithm. In a legitimate state, a node with a token selects one of its neighbors to pass on the token. If the system is in a illegitimate state, at least one node detects the error and takes corrective action. Thus, the error recovery procedure is integrated in the normal algorithm function. An algorithm is then self-stabilizing if (i) for any initial illegitimate state it reaches a legitimate state after a finite number of node moves, and (ii) for any legitimate state and for any move allowed by that state, the next state is a legitimate state.

A self-stabilizing system does not guarantee that the system is able to operate properly when a node continuously injects faults in the system (Byzantine fault) or when the communication errors occur so frequently that the new legitimate state cannot be reached until the new communication error. While the system services are unavailable when the self-stabilizing system is in an illegitimate state, the repair of a self stabilizing system is simple; once the offending equipment is removed or repaired the system provides its service after a reasonable time.

In a self-stabilizing algorithm, a node may change its local state by making a *move* (specification of an action). Algorithms are given as a set of rules of the form $p(i) \Rightarrow M$, where $p(i)$ is a predicate and M is a move. A node i becomes *privileged* if $p(i)$ is true. When a node becomes privileged, it may execute the corresponding move. We assume a serial model in which no two nodes move simultaneously. A central daemon selects, among all privileged nodes, the next node to move. If two or more nodes are privileged, we cannot predict which node will move next. Multiple protocols exist [5, 6, 7] that provide such a scheduler; hence our algorithms can be easily combined with any of those protocols to work under different schedulers as well. An execution will be represented as a sequence of moves $M_1, M_2, \ldots$, in which M_s denotes the s-th move. The system's initial state is denoted by s_0, and for $t > 0$, the state resulting from M_t is denoted by s_t.

3 Self-Stabilizing Coloring Algorithms

We model a distributed system with an undirected connected graph $G = (V, E)$, where the node set V represents the processors, and the edge set E represents the processor interconnections. Throughout this paper we assume $|V| = n$ and $|E| = m$. If i is a node, then $N(i)$, its *open neighborhood*, denotes the set of nodes to which i is adjacent. Every node $j \in N(i)$ is called a *neighbor* of node i. We let $d_i = |N(i)|$, the number of neighbors of node i, or its *degree*, and we let $\Delta = \max\{d_i \mid i \in V\}$.

Given a graph $G = (V, E)$, a *k-coloring* is a function $c : V \to \{1, 2, \ldots, k\}$, where the elements in the range are called *colors*. A node i is *properly colored* if for all $j \in N(i)$, $c(i) \neq c(j)$, and c is a *proper coloring* if all nodes are properly colored. A graph G can be properly colored with k colors if and only if its node set can be partitioned into k pairwise disjoint independent sets. The minimum number of colors needed to properly color a graph G is called its *chromatic number*, denoted $\chi(G)$, and is, in general, NP-hard to compute [12, 13].

3.1 Grundy Colorings

Given a coloring $c : V \to \{1, 2, \ldots, k\}$, a node i is called a *Grundy node* if

$$c(i) = \min\{\ell \geq 1 \mid (\forall j \in N(i))(c(j) \neq \ell)\}.$$

That is, i is colored with the smallest color not taken by any neighbor. Note that a Grundy node is by definition properly colored. If $c(i) = 1$, and if i is properly colored, then, trivially, it must be a Grundy node. A *Grundy coloring* is one in which every node is a Grundy node. While this idea seems to have originated in [14], a more recent discussion of Grundy colorings, with more references, can be found in [2].

If a Grundy coloring uses k colors, then any node i, colored k, must have exactly $k - 1$ neighbors colored $1, \cdots, k - 1$ and hence the degree of the node i is $k - 1$. It follows that

$$\chi(G) \leq k \leq \Delta + 1.$$

However, it is known that the number of colors in a Grundy coloring can be arbitrarily larger than $\chi(G)$. For example, all trees can be 2-colored, yet for any positive integer k, there exists a tree (whose order is exponential in k) that can be Grundy colored with k colors.

Despite this potential worst case behavior, there are good reasons to seek Grundy colorings. First, any graph G has a Grundy coloring which uses $\chi(G)$ colors. And on average, a Grundy coloring does fairly well. It is known that for random graphs in which each node pair is assigned an edge with probability $\frac{1}{2}$, a Grundy coloring will use about twice as many colors as are necessary [15].

We propose a self-stabilizing algorithm to produce a Grundy coloring for an arbitrary graph of order n. In this algorithm, each node i maintains a single integer variable $c(i)$, its color, where $1 \leq c(i) \leq d_i + 1$. The algorithm, given below, has a single rule, namely if a node's color is different than the first positive integer not taken by any neighbor, then it chooses that color instead.

Note that $c(i)$ is changed whenever a node i executes rule **R**, so we say a move is *increasing* if $c(i)$ increases, and *decreasing* otherwise. The following lemma is clear.

Lemma 1 *After any move made by node i, $1 \leq c(i) \leq d_i + 1$.*

A main idea in our analysis is to bound the number of decreasing moves that each node can make. In the next lemma, we speak of a node i making a sequence of *consecutive decreasing moves*. We mean here simply that there is no intermediate increasing move made by i, but there could be intermediate moves (either increasing or decreasing) made by other nodes.

Lemma 2 *A node i can make at most $d_i + 1$ consecutive decreasing moves.*

Proof: This follows from Lemma 1. □

Lemma 3 *A node can make an increasing move M_t only if it is not properly colored in state s_{t-1}.*

Proof: Node i can execute an increasing move if and only if for each $k \in \{1, 2, \ldots, c(i)\}$, there exists a $j \in N(i)$ with $c(j) = k$. In particular, i has a neighbor colored $c(i)$. □

Lemma 4 *After node i executes rule **R**, it becomes properly colored and remains so.*

Proof: It is clear that by executing **R**, i becomes properly colored. That it remains in this condition follows from the fact that by executing rule **R**, no node can ever destroy the proper coloring of another node. □

We mention that although the execution of rule **R** by node i cannot destroy the proper coloring of another node, it *can* destroy the Grundy coloring of another node.

Lemma 5 *Each node i can make at most one increasing move, and that can only occur on its first move.*

Proof: This follows from Lemmas 3 and 4. □

Lemma 6 *Each node i can make at most $d_i + 1$ moves.*

Algorithm 2.1: Grundy Coloring ()

R: If $c(i) \neq \min\{\ell \geq 1 \mid (\forall j \in N(i))(c(j) \neq \ell)\}$
then set $c(i) = \min\{\ell \geq 1 \mid (\forall j \in N(i))(c(j) \neq \ell)\}$

Proof: If node i never makes an increasing move, then it can make at most $d_i + 1$ moves by Lemma 2. If it does make an increasing move, by Lemma 5 this occurs only once as its first move. By Lemma 1, after this first move, $1 \leq c(i) \leq d_i + 1$. Its remaining moves must be decreasing, and there can be at most d_i of these.
□

Theorem 1 *Given a graph with n nodes and m edges, Algorithm 3.1 finds a Grundy coloring in at most $n + 2m$ moves.*

Proof: Using Lemma 6 and summing over every i, we see that any sequence of moves can have length at most $\sum_{i=1}^{n}(d_i + 1) = n + 2m$. □

Corollary 1 *For planar graphs, Algorithm 3.1 constructs a Grundy coloring in at most $7n - 12$ moves.*

Proof: It is well-known that for planar graphs, $m \leq 3n - 6$.
□

It should be noted that an algorithm, which at first glance appears similar to Algorithm 3.1, is given in [1], and has an accompanying worst case analysis of $O(\Delta n)$. This algorithm properly colors nodes with values in the range $\{0, 1, \ldots, \Delta\}$, always choosing the *largest* possible color. Our algorithm and analysis have two advantages. First, the bound of $n + 2m$ steps is an improvement. And second, in Algorithm 3.1, nodes are not required to know Δ, as in [1], nor any other global property.

3.2 $(\Delta + 1)$-Coloring in n Moves

Algorithm 3.1 constructs a Grundy coloring with at most $\Delta + 1$ colors in at most $n + 2m$ moves. In this section we present a simple algorithm that also constructs a proper $\Delta + 1$-coloring, but which stabilizes in at most n moves. This appears to be the first $O(n)$ self-stabilizing proper coloring algorithm.

The following lemma is self-evident.

Lemma 7 *After any move made by node i, $1 \leq c(i) \leq d_i + 1$.*

Lemma 8 *After node i executes rule* **R**, *it becomes properly colored and remains so.*

Proof: Clearly i becomes properly colored. It remains so because a move by another node can not destroy this property. □

Lemma 9 *Each node can move at most once.*

Proof: A node i is privileged if and only if it is not properly colored or $c(i) > d_i + 1$. By Lemma 7 and Lemmas 8, after moving, it can not become privileged again. □

Theorem 2 *For any graph with n nodes, Algorithm 3.2 finds a $\Delta + 1$-coloring in at most n moves.*

Proof: By Lemma 9, each node will move at most once, and clearly this will stabilize in a proper coloring. This coloring must use at most $\Delta + 1$ colors, or else some node would be privileged. □

A path P_n with n nodes, each of which is initially colored 1, is an example of a graph for which Algorithm 3.2 can make $n - 1$ moves, if the nodes move in order from left to right.

4 Conclusion

It is interesting to note that the set of nodes colored 1 by Algorithm 3.1 forms a maximal independent set. Algorithm 3.2 is inspired by the well-know Brooks' Theorem [16] which asserts that the chromatic number of a graph is at most $\Delta + 1$, and in fact is at most $k = \Delta$, unless $k = 2$ and G has a component which an odd cycle, or $n > 2$ and K_{n+1} is a component of G. It remains an interesting question whether one can construct a self-stabilizing algorithm for coloring a graph with at most Δ colors. Unlike Algorithm 3.1, Algorithm 3.2 does not necessarily find a maximal independent set.

5 Acknowledgement

Finally, the authors wish to thank D. Rall and M. Gairing for helpful comments during the preparation of this paper. The last author, Srimani's work was partially supported by a NSF grant # ANI-0073409.

Algorithm 2.1: Fast Coloring()

R: if $c(i) \in \{c(j) \mid j \in N(i)\} \vee c(i) > d_i + 1$

then
- /* recolor node i
- if $\{c(j) \mid j \in N(i)\} = \{1, 2, \ldots, d_i\}$
  - then $c(i) = d_i + 1$
  - else set $c(i) \in \{1, 2, \ldots, d_i\} - \{c(j) \mid j \in N(i)\}$

References

[1] M Gradinariu and S Tixeuil. Self-stabilizing vertex coloration and arbritary graphs. In *4th International Conference On Principles Of DIstributed Systems, OPODIS'2000*, pages 55–70, 2000.

[2] T. R. Jensen and B. Toft. *Graph coloring Problems*. John Wiley & Sons, New York, 1995.

[3] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

[4] E. W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1(1):5–6, 1986.

[5] M Nesterenko and A Arora. Stabilization-preserving atomicity refinement. In *DISC99 Distributed Computing 13th International Symposium, Springer-Verlag LNCS:1693*, pages 254–268, 1999.

[6] G Antonoiu and PK Srimani. Mutual exclusion between neighboring nodes in an arbitrary system graph tree that stabilizes using read/write atomicity. In *Europar'99 Parallel Processing, Proceedings LNCS:1685*, pages 823–830, 1999.

[7] J Beauquier, AK Datta, M Gradinariu, and F Magniette. Self-stabilizing local mutual exclusion and daemon refinement. In *DISC00 Distributed Computing 14th International Symposium, Springer-Verlag LNCS:1914*, 2000.

[8] S Ghosh and MH Karaata. A self-stabilizing algorithm for coloring planar graphs. *Distributed Computing*, 7:55–59, 1993.

[9] S Sur and PK Srimani. A self-stabilizing algorithm for coloring bipartite graphs. *Information Sciences*, 69:219–227, 1993.

[10] SK Shukla, DJ Rosenkrantz, and SS Ravi. Observations on self-stabilizing graph algorithms for anonymous networks. In *Proceedings of the Second Workshop on Self-Stabilizing Systems*, pages 7.1–7.15, 1995.

[11] F Petit and V Villain. A space-efficient and self-stabilizing depth-first token circulation protocol for asynchronous message-passing systems. In *Europar'97 Parallel Processing, Proceedings LNCS:1300*, pages 476–479. Springer-Verlag, 1997.

[12] M. R. Garey and M. R. Johnson. *Computers and Intractability*. Freeman, New York, 1979.

[13] R. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–104. Plenum Press, 1972.

[14] P. M. Grundy. Mathematics and games. *Eureka*, 2:6–8, 1939.

[15] G. R. Grimmett and C. J. H. McDiarmid. On coloring random graphs. *Mathematical Proceedings of the Cambridge Philosophical Society*, 77:313–324, 1975.

[16] R. L. Brooks. On coloring the nodes of a network. *Proceedings Cambridge Philos. Society*, 37:197–227, 1941.