AUSTRALASIAN JOURNAL OF COMBINATORICS
Volume **38** (2007), Pages 237–247

A self-stabilizing distributed algorithm for edge-coloring general graphs

Pranay Chaudhuri Hussein Thompson

Department of Computer Science, Mathematics and Physics
University of the West Indies, Cave Hill Campus
PO Box 64, Bridgetown
Barbados
pchaudhuri@uwichill.edu.bb hthomp@acm.org

Abstract

An edge coloring of a graph G seeks to assign unique colors to each edge in G, such that adjacent edges do not receive the same color. Edge colorings are known to have many practical applications. In this paper, we present a self-stabilizing distributed algorithm which will $\{2\Delta - 1\}$-edge-color an arbitrary graph G in $O(\Delta^2 m)$ moves, where Δ is an upper bound on the degree of a node and m is the number of edges in G. The algorithm is resilient to transient faults and changes in the network topology. In addition, the algorithm does not require initialization of the nodes in the system in order to converge to a correct solution.

1 Introduction

Consider a graph $G = \langle V, E\rangle$, where V is a nonempty set of nodes and $E \subseteq V \times V$ is the set of edges. We denote the cardinality of the sets V and E as $\|V\| = n$ and $\|E\| = m$ respectively. A *k-coloring* is a function $f : V \to \{1, 2, \ldots, k\}$ from the vertex set V onto the set of positive integers $\{1, 2, \ldots, k\}$ having the property that $f(i) \neq f(j)$ for every edge $(i, j) \in E$. The minimum k for which G has a k-coloring is called the *chromatic number* and is denoted $\chi(G)$. A *k-edge-coloring* f', may be defined analogously. Here however adjacent edges receive different colors, i.e. $f'(i, j) \neq f'(u, v)$ whenever u (or v) $\in \{i, j\}$. The minimum k for which G has a k-edge-coloring is called the *edge chromatic number* or *chromatic index* of G and is denoted $\chi'(G)$. Edge colorings have applications to map coloring, matching theory, job shop scheduling, packet routing and resource allocation [19, 13]. Although not as extensively studied as the vertex coloring problem [13], there are a number of known results for the edge coloring problem in the literature. We mention some of them here.

It is obvious that at least Δ colors are needed to edge color a graph G, where Δ is the maximum degree of any node in G. In 1964 Vizing showed that $\chi' \leq \Delta + 1$ [9, 6].

This also leads to a ploynomial time sequential $\Delta+1$ -edge-coloring algorithm [9]. It is NP-complete to determine if an arbitrary graph is Δ-edge colorable or $\Delta+1$-edge colorable. Mitchell and Hedetniemi[18] present an $O(n)$ sequential algorithm for edge coloring trees. A slight modification is also presented to edge-color a unicyclic graph (after the underlying tree is colored). The tree algorithm uses Δ colors which is optimum since the edge-chromatic number of a bipartite graph G satisfies, $\chi'(G) = \Delta$ [13, Theorem 27]. It was also shown when $\chi'(H) = \Delta + 1$ for a unicyclic graph H.

In 1987 Karloff and Shmoys gave a RNC PRAM algorithm [15] that used $\Delta + \sqrt{\Delta}$ colors and $\log n$ time. It is possible to derandomize this algorithm to obtain a deterministic parallel version.

In 1992 Panconesi and Srinivasan gave two randomized distributed algorithms for edge coloring bipartite graphs [19] and a means to extend the algorithms to general graphs is given using an idea from [15]. Note that the same time complexities are achieved. In 1997 Grable and Panconesi present a distributed $(1+\varepsilon)\Delta(G)$ coloring randomized algorithm which executes in $O(\log\log n)$ time for large enough n (polylog $n \leq n \leq n^c$, for some positive c) [9]. This algorithm was modified later in [17, 16] to reduce the failure probability.

Recently Sakurai et al. [20] presented a self-stabilizing algorithm resilient to Byzantine fault and crash failures which edge colors a tree network. The algorithm uses $\Delta + 1$ colors (1 more color than is necessary) in order to achieve the constant containment radius of (2, 1) and assumes that the tree is rooted. There the model used was a shared memory model and both the central and distributed daemons were considered. In this paper we present a self-stabilizing distributed algorithm which will edge color an arbitrary graph G. The algorithm uses $2\Delta - 1$ colors, which appears to be the current known upperbound for deterministic distributed algorithms [19]. We show that our algorithm requires $O(\Delta^2 m)$ moves to stabilize.

The rest of the paper is organized as follows. Section 2 introduces the model of computation and the self-stabilizing algorithm is presented in Section 3. The proof of correctness and the complexity related issues are provided in Section 4 and finally Section 5 concludes the paper.

2 Model of Computation

We define a *distributed system* to be a collection of processing elements (or nodes) which do not share a global memory. Every node has a distinct set of local variables, whose contents specify the *local state* of that node. Note that each node is permitted to read its own state and the state of its neighbors (adjacent nodes) but may modify only its local state. The union of the local states of all nodes in the system specifies the *global state* of the system. Two classes of global states exist. These are (i) legitimate state, and (ii) illegitimate state. The goal in a self-stabilizing distributed system is to start from an arbitrary (possibly illegitimate) initial state and reach a legitimate state after a finite number of moves (steps). Self-stabilizing algorithms are resilient to *transient faults*. A transient fault is defined to be a fault that perturbs the state of the system but not its program. If any transient fault occurs, the algorithm

is automatically reactivated and brings the system back to the legitimate state within a finite time. Dijkstra first introduced the concept of self-stabilization in distributed systems [7]. Since then, self-stabilizing algorithms for many fundamental problems, including some graph theory problems have been reported in the literature (see e.g., [1, 3, 4, 5, 12, 14, 11]).

The system is modeled as an asynchronous network of processors with their own local memory, such that each processor corresponds to a node of the given graph $G = \langle V, E \rangle$. The set of edges E correspond to bidirectional communication links. Therefore, adjacent nodes $i, j \in V$ (i.e. $(i, j) \in E \, or \, (j, i) \in E$) can access each others data. The algorithm for each node $i \in V$ can be expressed as

$$*[G[1] \rightarrow M[1] \, \square \, G[2] \rightarrow M[2] \square \ldots \square \, G[k] \rightarrow M[k]],$$

where $*[S]$ refers to the repeated execution of the statement S until all guards are false and $\square$ denotes the nondeterminism symbol. Each *action* is of the form $G[x] \rightarrow M[x]$ for some $x, 1 \leq x \leq k$. The *guard* of an action $G[\,]$, is a Boolean expression over the variables of node i and those of its adjacent neighbors. A *move* $M[\,]$, is defined to be the execution of an assignment statement of an action and is atomic in nature. A node i is said to *enjoy the privilege* when any one of its guards evaluates to true. Such a guard and the corresponding move is said to be *enabled*. Only a privileged node can make a move. If more than one node enjoys the privilege, then one of the privileged nodes is nondeterministically selected to make the move. Further, if the selected node has more than one enabled guard, then one of the enabled actions is nondeterministically selected for execution. For successful termination of an algorithm the following global criteria must be satisfied.

GC1 In every illegitimate state, at least one node enjoys the privilege.

GC2 In a legitimate state, no node enjoys the privilege. Therefore, once the system reaches a legitimate state, no further move is possible.

GC3 Regardless of the initial state and regardless of which node out of a set of nodes enjoying the privilege is selected each time for the next move, the system is guaranteed to reach a legitimate state after a finite number of moves.

In self-stabilizing systems, two types of *schedulers* (*daemons*) exist. These are *distributed* and *centralized daemons*. Under the centralized daemon, only one enabled node is selected to execute a move at any given time, whereas under the distributed daemon, an arbitrary subset of the enabled nodes are allowed to execute at any given time. Therefore the centralized daemon may be considered as a special instance of the distributed daemon. We initially assume a central daemon, but later show that the proposed algorithm also works under a distributed daemon.

In Dijkstra's definition of a self-stabilizing system [7], the system can make a move from one legitimate state to another legitimate state and therefore algorithm executions are infinite. However, for self-stabilizing graph algorithms (i.e. algorithms which compute some relation on the current network topology), once the system has reached a legitimate state, each state is merely repeated. Such algorithms have

also been called *static* [20, 8], *silent* [8] or *noninteractive* [21]. As a consequence, we assume that once the system is in the legitimate state the algorithm is *terminated*. In this case, if any transient fault occurs that takes the system back into an illegitimate state, the algorithm is automatically reactivated and brings the system back to the legitimate state within a finite time.

3 The Self-stabilizing Algorithm

3.1 Outline of the Algorithm

In this section we present ALGORITHM-SEC, a self-stabilizing algorithm for producing an edge coloring of any graph. It is known that the problem of edge coloring a given graph G is equivalent to node coloring the line graph of G (denoted by $\mathcal{L}(G)$). A $\Delta + 1$ node coloring of $\mathcal{L}(G)$ translates to a $2\Delta - 1$ edge coloring of G [19]. Although it appears possible to allow the actual nodes in V to emulate virtual nodes in $\mathcal{L}(G)$, and then run either of the node coloring algorithms in [10] *on top of* the virtual network $\mathcal{L}(G)$, we run into a problem since a pair of nodes in $\mathcal{L}(G)$ may actually be a physical distance of two away in G. Note that in [8] a generic scheme is provided which allows self-stabilizing algorithms to use distance-two information, but the presented scheme requires distinct node IDs which are totally ordered. In addition, algorithms `GRUNDY-COLORING()` and `FAST-COLORING()` in [10] would run in $O(m^2)$ and $O(n \times m)$ time respectively, under the scheme described above [8, Theorem 1]. Nonetheless, we still make use of the self-stabilizing node coloring algorithm presented by Hedetniemi et al. [10].

ALGORITHM-SEC is fairly simple and works as follows. For each edge, a leader must be chosen (and we refer to non-leaders as *gatherers* rather than followers). The leader u of an edge $uv \in E$ will color that edge based on the color of all incident edges uv', where $v' \neq v$ and the color of all edges $u'v$, where $u' \neq u$, incident to the gatherer v. The gatherer v of an edge uv will collect the colors in use by all of its incident edges except u and make this information available to u through the variable $N\_Color(v)(u)$. These concepts are illustrated further in Section 5. Note that node v maintains $N\_Color(v)(u)$ only if node u is the leader of edge uv.

3.2 The Algorithm

We now present the algorithm more formally in the figures that follow. All terminology used in the algorithm is shown in Table 1. The procedures which the algorithm makes use of are provided in Figure 1 and ALGORITHM-SEC is shown in Figure 2.

4 Correctness and Complexity

In this section of the paper, we provide a correctness proof of ALGORITHM-SEC. Note that from [10], only $O(n)$ moves are required to elect a leader for each edge and that if no leader has been elected for an edge, it will not be colored.

```
1: procedure ELECT( )/*FAST-COLORING[11]*/
2:     if {c(j) : j ∈ N(i)} = {1, 2, ..., deg(i)} then
3:         c(i) := deg(i) + 1
4:     else
5:         c(i) := x ∈ {1, 2,..., deg(i)} − {c(j) : j ∈ N(i)}
6:     end if
7: end procedure

8: procedure GATHER(i, j)/*NB: j in N(i)*/
9:     T := {}
10:    for all k ∈ N(i) − {j} do
11:        if ELECTED(i, k) then
12:            T := T ∪ Color(i)(k)
13:        else
14:            T := T ∪ Color(k)(i)
15:        end if
16:    end for
17:    return T
18: end procedure

19: procedure COLOREDGE(i, j)/*NB: j in N(i)*/
20:    T := GATHER(i, j) ∪ N_Color(j)(i)
21:    x := min{1, 2, ..., deg(i) + deg(j) − 1} − T
22:    for all k ∈ N(i) − {j} do
23:        if GATHERER(i, k) then
24:            N_Color(i)(k) := GATHER(i, k) /*Allow neighbors to see new color*/
25:        end if
26:    end for
27:    return x
28: end procedure
```

Figure 1: Procedures used in ALGORITHM-SEC

```
*[
/*R1:*/   CORRUPT(i) → ELECT()
/*R2:*/   □ There exists j ∈ N(i) AND GATHERER(i, j) AND
                N_Color(i)(j) ≠ GATHER(j) → N_Color(i)(j) := GATHER(i, j)
/*R3:*/   □ There exists j ∈ N(i) AND ELECTED(i, j) AND
                Color(i)(j) ≠ COLOREDGE(i, j) → Color(i)(j) := COLOREDGE(i, j)
]
```

Figure 2: ALGORITHM-SEC

Terminology	Definition
$N(i)$	The set of neighbors of node i
$c(i)$	The color assigned to the node i by algorithm `FAST-COLORING()`
$Color(i)$	The set of colors assigned to the edges incident to node i
$Color(i)(j)$	The color assigned to edge ij, where $j \in N(i)$
$N\_Color(i)(j)$	The set of colors assigned to all the neighbors of node i, except j
$CORRUPT(i)$	**There exists** a neighbor $j \in N(i)$ with $c(i) = c(j)$
$ELECTED(i,j)$	NOT $CORRUPT(i)$ AND $j \in N(i)$ AND $c(i) < c(j)$
$GATHERER(i,j)$	NOT $CORRUPT(i)$ AND $j \in N(i)$ AND $c(i) > c(j)$

Table 1: Terminology used in ALGORITHM-SEC

Lemma 1. *In any legitimate state of* ALGORITHM-SEC, *the color assigned to any edge is at most* $2\Delta - 1$.

Proof. Consider a node u which is elected to color edge uv. In one worst case, all of the edges incident to v, except edge uv, are assigned the colors $\{1, 2, \ldots, deg(v) - 1\}$ and all of the edges incident to u, except edge uv, are assigned the colors $\{deg(v), deg(v) + 1, \ldots, deg(u) + deg(v) - 2\}$. In this case, u must assign to the edge uv the color $deg(u) + deg(v) - 1$. Since $deg(u) + deg(v) - 2 \leq 2\Delta - 2$, the lemma holds. □

By the definition of ALGORITHM-SEC, the following lemma holds.

Lemma 2. *In any legitimate state of* ALGORITHM-SEC, *the coloring of edges is a proper edge coloring.*

We now provide the move complexity of ALGORITHM-SEC. To simplify the complexity analysis we assume that each move takes constant time. Note that this assumption does not affect the correctness proof of ALGORITHM-SEC.

Lemma 3. ALGORITHM-SEC *requires at most* $O(\Delta^2 m)$ *moves to converge.*

Proof. Consider an arbitrary directed edge (s_0, t) in G', as shown in Figure 3. Recall that the directed graph G' shows the elected node for each edge in G, and that $(u, v) \in G'$ implies that node u will color the edge $uv \in G$. We note that the number of times a node can change any of its N_Color-values is bounded (to within a constant factor) by the number of times its neighbors can change their color, therefore we count only the maximum number of times a node s_0 will color the edge $\{s_0, t\} \in G$.

Whenever s_0 colors $\{s_0, t\}$ with say x, the nodes in A will not choose x for any of their incident edges and s_0 will not choose x when coloring any of the edges with endpoints in B. Similarly, node t will not choose x when coloring edges in C. Only a change in the color of the edges $\{s_i, t\}$, $1 \leq i \leq k$, can cause node s_0 to recolor $\{s_0, t\}$. Without loss of generality assume that each s_i colors the edge $\{s_i, t\}$ for the first time only after s_{i-1} has colored the edge $\{s_{i-1}, t\}$, for $1 \leq i \leq k$. Then, clearly

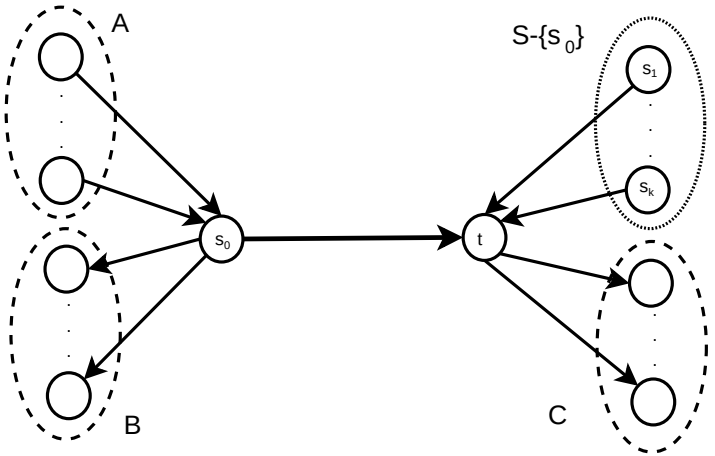


Figure 3: An arbitrary edge (s_0, t) in G' and its neighboring edges.

$O(k^2)$ such moves are possible.Note that because of the automatic neighbor update in procedure *ecolor*(), each s_i cannot be forced to recolor $\{s_i, t\}$ because of changes in the color of $\{s_i, w\}, for\ w \in N(s_i) - \{t\}$.

Since there are m edges in G, at most $O(\sigma^2 m)$ moves are possible, where σ is an upperbound on the size of any S in G. Clearly $\sigma \leq \Delta$, thus completing the proof. □

By the above lemma, it can be seen that each node can execute a finite number of moves, therefore ALGORITHM-SEC satisfies total correctness. Lemma 2 shows that partial correctness is also satisfied, therefore we get the following theorem.

Theorem 1. ALGORITHM-SEC *is correct.*

We now show that ALGORITHM-SEC is self-stabilizing. Let GP be a global predicate defined over the set of global states GS. Then ALGORITHM-SEC is self-stabilizing with respect to GP if it satisfies:

(i) *Closure:* If a global state $q \in GS$ satisfies GP, then any global state reachable from q also satisfies GP

(ii) *Convergence:* Starting from any arbitrary state, the system is guaranteed to reach a global state satisfying GP in a finite number of steps.

Lemma 4. ALGORITHM-SEC *satisfies both closure and convergence properties.*

Proof. Let GP be defined as follows. $GP = (\neg G1 \wedge \neg G2 \wedge \neg G3)$ where Gi is the guard of the i-th rule. Then, (i) by definition, if GP is satisfied, then no node enjoys the privilege and hence the algorithm returns the color for each edge $uv \in G$ and terminates. Since no other global state is reachable after ALGORITHM-SEC terminates, closure is trivially proved. (ii) by Theorem 1, predicate GP is eventually satisfied. Hence convergence is also satisfied. □

We may now state the following theorem.

Theorem 2. ALGORITHM-SEC *is self-stabilizing and computes a* $\{2\Delta - 1\}$*-edge-coloring of an arbitrary graph* G *after* $O(\Delta^2 m)$ *moves.*

5 Illustrative Example

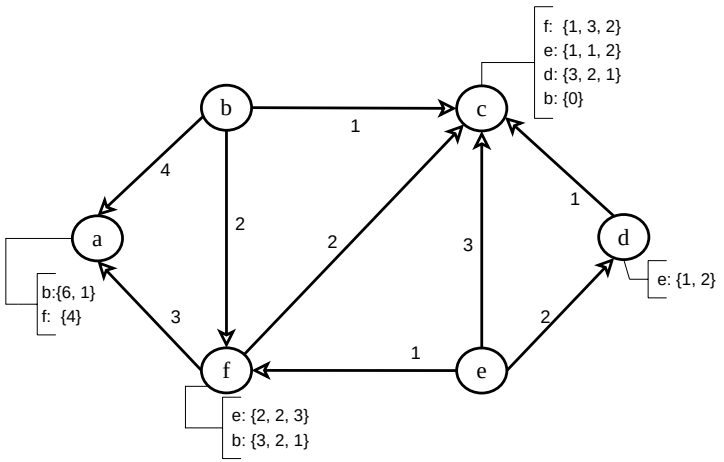
Figure 4(a) shows a system which has a leader for each edge and the contents of the remaining variables are arbitrary. The directed edges (u, v) indicate that node u is the leader of the undirected edge uv. The contents of the variables $Color(u)(v)$ are shown adjacent to the corresponding directed edge and the contents of $N\_Color(u)(v)$ are shown within the square brace connected to u. For instance, node a is not the leader for any incident edge and $N\_Color(a)(b) = \{6, 1\}$. Also, node f is the leader of edge cf, $Color(f)(c) = 2$ and $N\_Color(f)(e) = \{2, 2, 3\}$. In Figure 4(b), we assume the scheduler applies some rules of ALGORITHM-SEC to Figure 4(a) as follows. For node a, rule *R2* is applied to neighbor b, then node a changes $N\_Color(a)(b)$ to reflect the color of the edge af, namely $\{3\}$. Similarly if node c applies *R2* to neighbor d, its $N\_Color(c)(d)$ variable is changed to $\{2, 3, 1\}$ as shown. Assume further that node d executes rule *R3* for neighbor c. Then it calls COLOREDGE(d, c), which in turn calls GATHER(d, c) and T is set to $\{2\} \cup \{3, 2, 1\}$ as a result of (Figure 1) line 20. The value of the variable x is then set to $\min\{1, 2, 3, 4, 5\} - \{3, 2, 1\} = 4$. Then node d updates its $N\_Color(d)(e)$ value as per lines 22–26. Similarly, node f updates the color of edge cf to 4 and allows the neighbors b and e to see this new color by updating $N\_Color(f)(b)$ and $N\_Color(f)(e)$.

Note that although discussed sequentially, the same result after the application of the above moves is achieved if all the nodes were allowed to execute concurrently provided that neighboring nodes active at the same time read the previous state. Note further, that the system shown in Figure 4(b) is not yet in a legitimate state since node c can still execute rule *R2*.

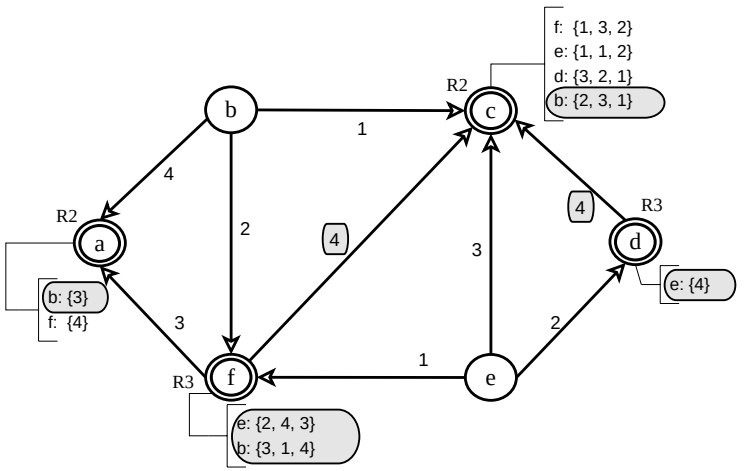
6 Conclusion

In this paper, we presented the self-stabilizing algorithm, ALGORITHM-SEC, which computes a $\{2\Delta - 1\}$-edge-coloring of an arbitrary graph G. The algorithm does not require initialization of the local variables kept at each node in the system. Starting from any arbitrary state, ALGORITHM-SEC is guaranteed to produce the correct output after $O(\Delta^2 m)$ moves. If the system is in the legitimate state and a transient fault occurs, or a change in the network topology changes the state of the system to an illegitimate state, the algorithm is automatically reactivated and brings the system back to a legitimate state after a finite time.

In the algorithm, each node indexed its neighbors by their *labels* (IDs). We note that this does not require the system to have globally unique IDs as in [8]. For instance, each node could maintain an arbitrary ordering of its edges, and use this ordering to index its neighbors instead.



(a)



(b)

Figure 4: Partial execution sequence of ALGORITHM-SEC on an arbitrary graph

In our model, we assumed the existence of a central daemon. Burns et al. [2] shows that a solution which is correct under a central demon is also correct with a distributed daemon provided two conditions hold. These are,

1. There are no multiwrite variables
2. There is no infinite sequence of moves involving a proper subset of nodes consisting of a pair of neighbors only.

The first condition follows since only a given node can modify its own state. The second condition holds since the number of moves the system can make is finite. Therefore, the proposed solution also works with distributed daemons.

We note here that the presented algorithm produces an *edge Grundy coloring*. A k-edge Grundy coloring f', is an edge coloring such that for every edge $e \in E$, the color of e, $f'(e)$ is the smallest positive integer not used as a color on any neighboring edge of e. It is known that Grundy colorings produce fairly good colorings on average and use twice as many colors as necessary on random graphs with edge probability 1/2 [10].

Acknowledgement

The authors wish to thank the anonymous referee for comments and suggestions on an earlier version of the manuscript which have greatly enhanced the readability of the paper. We also wish to thank the referee for pointing out that an edge Grundy coloring is achieved.

References

[1] G. Antonoiu and P. K. Srimani, Distributed self-stabilizing algorithm for minimum spanning tree construction, *Euro-Par '97 Parallel Processing*, Proc. LNCS: 1300, Springer-Verlag, 1997, 480–487.

[2] J. E. Burns, M. Gouda and R. Miller, On relaxing interleaving assumptions, In *Proc. MCC Workshop on Self-stabilizing Systems*, MCC Technical Report No. STP-379-89, 1989.

[3] P. Chaudhuri, A self-stabilizing algorithm for detecting fundamental cycles in a graph, J. Comp. System Sc. 59 (1999), 84–93.

[4] S. Chen, H. P. Yu and S. T. Huang, A self-stabilizing algorithm for constructing spanning trees, *Information Processing Letters* 39 (1991), 147–151.

[5] Z. Collin and S. Dolev, Self-stabilizing depth-first search, *Information Processing Letters* 49 (1994), 297–301.

[6] R. Diestel, *Graph Theory*, Springer-Verlag, New York, 2002.

[7] E. W. Dijkstra, Self-stabilization in spite of distributed control, *Communic. ACM* 17 (1974), 643–644.

[8] M. Gairing, W. Goddard, S. T. Hedetniemi, P. Kristiansen and A. A. McRae, Distance-Two Information in Self-stabilizing Algorithms, *Parallel Processing Letters* 14 (2004), 387–398.

[9] D. A. Grable and A. Panconesi, Nearly optimal distributed edge colouring in $O(\log \log n)$ rounds, *Proc. Eighth Annual ACM-SIAM Symp. Discrete Algorithms*, Jan. 1997, New Orleans, Louisiana. ACM/SIAM, 1997, 278–285.

[10] S. T. Hedetniemi, D. P. Jacobs and P. K. Srimani, Linear time self-stabilizing colorings, *Information Processing Letters* 87 (2003), 251–255.

[11] S. C. Hsu and S. T. Huang, A self-stabilizing algorithm for maximal matching, *Information Processing Letters* 43 (1992), 77–81.

[12] S. T. Huang and N. S. Chen, A self-stabilizing algorithm for constructing breadth-first trees, *Information Processing Letters* 41 (1992), 109–117.

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

[14] M. H. Karaata and P. Chaudhuri, A self-stabilizing algorithm for bridge finding, *Distributed Computing* 12 (1999), 47–53.

[15] H. J. Karloff and D. B. Shmoys, Efficient Parallel Algorithms for Edge Coloring Problems, *J. Algorithms* 8 (1987), 39–52.

[16] M. V. Marathe, A. Panconesi and L. D. Risinger, An experimental study of a simple, distributed edge-coloring algorithm, *J. Experimental Algorithmics* 9 (2004), 1–22.

[17] M. V. Marathe, A. Panconesi and L. D. Risinger, An experimental study of a simple, distributed edge coloring algorithm , *ACM Symp. Parallel Algorithms and Architectures*, 2000, 166–175.

[18] S. Mitchell and S. Hedetniemi, Linear Algorithms for edge-coloring trees and unicyclic graphs, *Information Processing Letters* 9 (1979), 110–112.

[19] A. Panconesi and A. Srinivasan, Fast Randomized Algorithms for Distributed Edge Coloring (Extended Abstract), In *Proc. 11th Annual ACM Symposium on Principles of Distributed Computing*, 1992, 251–262.

[20] Y. Sakurai, F. Ooshita and T. Masuzawa, A self-stabilizing link-coloring protocol resilient to Byzantine faults in tree networks, *8th Internat. Conf. Principles of Distributed Systems* (OPODIS 2004), Dec. 2004, Grenoble, France, 2004.

[21] M. Schneider, Self-stabilization, *ACM Computing Surveys* 25 (1993), 45–67.

(Received 22 May 2006; revised 22 June 2006)