An Efficient Self-stabilizing Distance-2 Coloring Algorithm

Jean Blair[1] and Fredrik Manne[2]

[1] Department of EE and CS, United States Military Academy West Point, NY, 10996, USA
Jean.Blair@usma.edu

[2] Department of Informatics, University of Bergen, N-5020 Bergen, Norway
fredrikm@ii.uib.no

Abstract. We present a self-stabilizing algorithm for the distance-2 coloring problem that uses a constant number of variables on each node and that stabilizes in $O(\Delta^2 m)$ moves using at most $\Delta^2 + 1$ colors, where Δ is the maximum degree in the graph and m is the number of edges in the graph. The analysis holds true both for the sequential and the distributed adversarial daemon model. This should be compared with the previous best self-stabilizing algorithm for this problem which stabilizes in $O(nm)$ moves under the sequential adversarial daemon and in $O(n^3 m)$ time steps for the distributed adversarial daemon and which uses $O(\delta_i)$ variables on each node i, where δ_i is the degree of node i.

1 Introduction

The problem of preventing potential interference when assigning frequencies to processes can be modeled as a graph coloring problem where nodes that are sufficiently close must have different colors. As frequencies (colors) are a scarce resource, it is also desirable to use as few colors as possible. A number of different objective functions and models have been studied for this problem; see [1] for a recent survey. In the current paper we study one such problem, that of assigning colors to nodes so that two nodes that are within distance two of each other are assigned different colors. We present and analyse an efficient self-stabilizing algorithm for this problem. The remainder of this section briefly surveys previous work on self-stabilizing coloring algorithms and then shows how the current paper extends and improves on that body of knowledge.

In 1993 Ghosh and Karaata [4] presented an algorithm for coloring planar graphs using at most 6 colors by transforming the graph into a directed acyclic graph, and assuming that all nodes have unique identifiers. This result was later improved to work with bounded variable values and without identifiers by Huang et al. [9] and finally was generalized to a wider class of graphs by Goddard et al. [5].

Also in 1993, Sur and Srimani [14] gave an algorithm for exact coloring of bipartite graphs. The algorithm assumes that a specific node is a root and then colors nodes based on the distance from the root. For this algorithm only finite stabilization was shown and there was no bound on the number of moves. This work was later extended by Kosowski and Kuszner [10] who presented a self-stabilizing algorithm that colors bipartite graphs using exactly two colors and using a polynomial number of moves. Their algorithm also relies on a distinguished root.

Shukla et al. [11] offered randomized self-stabilizing algorithms for coloring of anonymous chains and oriented rings. In [12] the same authors developed self-stabilizing algorithms for two-coloring several classes of bipartite graphs, namely complete odd-degree bipartite graphs and tree graphs.

The first self-stabilizing coloring algorithms for general graphs were given by Gradinariu and Tixeuil [7] in 2000. They presented three different algorithms based on a greedy assignment technique. These algorithms use at most $\Delta + 1$ colors and stabilize in $O(n\Delta)$ moves, where Δ is the maximum node degree in the graph. It is assumed that each node has knowledge of Δ. This result was later improved by Hedetniemi et al. [8] who gave two algorithms for coloring arbitrary graphs, respectively, also using $\Delta + 1$ colors. The moves complexity of these algorithms is $O(n)$ and $O(m)$, where the latter algorithm also guarantees that each node is assigned the smallest available color within its neighborhood.

Other types of coloring problems have also been studied using the self-stabilizing paradigm. For instance, [13] gives a self-stabilizing algorithm that tries to achieve a node coloring where the sum of the colors assigned to each node is minimum. [15] presents a self-stabilizing $\Delta + 4$ edge coloring algorithm for planar graphs in anonymous networks, while [2] describes a self-stabilizing algorithm for edge coloring general graphs.

In this paper we consider self-stabilizing algorithms for the distance-2 coloring problem. That is, one wants to assign colors to the nodes in such a way that each node receives a color different from its neighbors within distance 2 (i.e. different from all of the nodes neighbors and its neighbors' neighbors).

In [6] Gradinariu and Johnen describe a self-stabilizing algorithm for the problem of *unique naming*. This is essentially the same problem as is studied here in that it asks for an assignment of labels to nodes such that no two nodes who are distance-2 neighbors have the same label. They present a randomized scheme where the expected number of moves by each node is one. However, the scheme requires that every node knows n, the number of nodes in the network, and it assigns colors in the range $[1, 2n^2]$.

In [3] Gairing et al. introduce a general mechanism for allowing a node to obtain information at distance-2 from it. The idea is based on each node copying the states of its neighbors and thus making this information available to its own neighbors. It is shown how a distance-2 coloring can be obtained in $O(nm)$ moves under the sequential daemon model and in $O(n^3m)$ time steps under the distributed daemon model. In these algorithms the color of each node can easily be chosen in the range $[1, \delta2_i + 1]$ where $\delta2_i$ denotes the number of distance-2 neighbors of node i. We note that the algorithm requires that each node i maintain $O(\delta_i)$ variables where δ_i is the number of neighbors of i.

In the current paper we present a self-stabilizing algorithm for the distance-2 coloring problem that uses at most $\Delta^2 + 1$ colors. The algorithm stabilizes in $O(\Delta^2 m)$ moves under the sequential daemon and also uses the same number of time steps for the distributed daemon model. For a fair daemon (sequential or distributed) our algorithm requires $O(\Delta m)$ rounds to stabilize. In addition, each node is only required to maintain a constant number of variables. Thus our algorithm improves the time step complexity for the distributed adversarial daemon by at least a factor of n and depending on how Δ^2 compares with n the algorithm might also improve the moves complexity for the sequential adversarial daemon. For instance, for a graph where the degree of each node is at most a constant, our algorithm improves the moves complexity by a factor of n for the sequential adversarial daemon and by a factor of n^3 for the distributed adversarial daemon. Moreover, our algorithm improves the overall memory consumption from $O(m)$ down to $O(n)$ variables.

The rest of this paper is organized as follows. In Section 2 we give a short introduction to the self-stabilizing model. In Section 3 we present and motivate our algorithm. In Section 4 we show that any stable configuration of the algorithm also gives a valid distance-2 coloring and in Section 5 we analyze the complexity of the algorithm. Finally, we conclude in Section 6.

2 Model

A system consists of a set of processes where two adjacent processes can communicate with each other. The communication relation is typically represented by a graph $G = (V, E)$ where $|V| = n$ and $|E| = m$. Each process corresponds to a node in V and two nodes i and j are adjacent if and only if $(i, j) \in E$. We assume that each node has a unique identifier. In the following we will not distinguish between a node and its identifier.

The set of neighbors of a node $i \in V$ is denoted by $N(i)$ and $N[i] = N(i) \cup \{i\}$. Similarly we define $N^2(i)$ as the set of neighbors of node i within distance 2 of i and $N^2[i] = N^2(i) \cup \{i\}$. Let $\delta_i = |N(i)|$ and $\Delta = \max_{i \in V} \delta_i$.

A node maintains a set of local variables which make up the local state of the node. Each variable ranges over a fixed domain of values. Every node executes the same algorithm, which consists of one or more rules. A rule has the form name : **if** *guard* **then** *command*. A guard is a boolean predicate over the variables of both the node and those of its neighbors. A command is a sequence of statements assigning new values to the variables of the node.

An assignment of a value to every variable of each node from its corresponding domain defines a configuration of the system. A rule is enabled in some configuration if the guard is true with the current assignment of values to variables. A node is eligible if it has at least one enabled rule. A computation is a maximal sequence of configurations such that for each configuration s_i, the next configuration s_{i+1} is obtained by executing the command of at least one rule that is enabled in s_i. (A node that executes such a rule makes a move or a step). A configuration is defined as stable if there are no eligible nodes in the system.

A daemon is a predicate on executions. We distinguish several kinds of daemons: the sequential daemon makes the system move from one configuration to the next by executing exactly one enabled rule, while the distributed daemon achieves this by executing any non-empty subset of enabled rules. Note that a sequential daemon is an instance of the distributed daemon. Also, a daemon is fair if any rule that is continuously enabled is eventually executed, and adversarial if it may execute any enabled rule at every step. Again, the adversarial daemon is more general than the fair daemon.

A system is self-stabilizing for a given specification if in finite time it converges to a stable configuration that conforms to this specification, independent of its initial configuration and without external intervention.

We consider two measures for evaluating complexity of self-stabilizing programs. A step is the minimum unit of time such that a process can perform any of its moves. For a sequential daemon exactly one process executes one eligible rule during each step, while for a distributed daemon there can be several processes that each makes one simultaneous move during a given step. Thus, the *step complexity* measures the maximum number of steps that are needed to reach a configuration that conforms to the specification (i.e. a legitimate configuration) for all possible starting configurations. The *round complexity* considers that executions are observed in rounds: a round is the smallest sub-sequence of an execution in which every process that was eligible at the beginning of the round either makes a move or has its guard(s) disabled since the beginning of the round. Note that both of these types of analysis focus on communication and not on computation, as it is assumed that a process can perform any type of necessary local computation during one move.

3 The Algorithm

In the following we motivate and describe the new algorithm. We begin by comparing the algorithm with previous self-stabilizing coloring algorithms. In doing so, we examine how coloring conflicts at distance-1 and distance-2 are handled.

For coloring conflicts between neighboring nodes any self-stabilizing algorithm must avoid the possibility of two adjacent nodes repeatedly changing their colors to the same color in a lockstep fashion. With a sequential daemon this is straight forward to handle [8]. For a distributed daemon one can solve this by using a randomized scheme if the network is anonymous [7], or if the nodes have unique identifiers by using the relative values of the identifiers to break ties [7].

For coloring conflicts between nodes at distance-2 there are two issues to consider: how to discover a coloring conflict and then how to resolve it. Even for a sequential daemon, resolving a conflict can be difficult, as information does not propagate immediately between distance-2 neighbors.

Gairing et al. [3] let each node maintain a local copy of the colors of its neighbors. Thus a node i has direct access to the colors of the nodes in $N^2[i]$ and can itself discover any coloring conflicts that it is involved in. A node that wants to change its color must then obtain permission from all of its distance one neighbors before doing so. This is achieved by using pointers. In this way no two nodes at distance two from each other can change color at the same time.

In the algorithm by Gradinariu and Johnen [6] coloring conflicts are detected by a node i that discovers that it has two neighbors with the same color (one "neighbor" may in fact be i itself). The node i then sets a flag value equal to the conflicting color. This signals that any node in $N[i]$ using this color should recolor itself. Nodes that are affected by this then choose a new color randomly from a predetermined interval.

In our algorithm we combine ideas from both [3] and [6]. A coloring conflict is detected by any node that is adjacent to the conflicting nodes. This node will then signal to exactly one of the conflicting nodes i that it should change its color. When i sees the signal it will put up a flag requesting to change its color. However, i can only recolor itself once all of its neighbors have acknowledged the flag by themselves pointing to i. In this way no other node in $N^2[i]$ can change its color at the same time as i. To select the appropriate color we use a novel deterministic scheme where i will perform a linear search starting from color 1 until it finds a valid color. Each possible color that i considers must either be accepted or rejected by the neighbors of i. If any neighbor rejects the suggested color, i will try the next possible color and repeat until it finds a color that is accepted by all of its neighbors.

A recoloring can either take place because of a distance-1 or a distance-2 coloring conflict. In addition we also force recoloring if the color of a node is higher than a reasonable upper bound on the size of its distance-2 neighborhood. This assures that the final coloring never uses more than $\Delta^2 + 1$ colors.

The following list gives the variables that are available on each node i.

- $dist1deg_i$, the size of $|N[i]|$.
- $dist2deg_i$, an upper bound on the number of nodes in $N^2[i]$. Every node should get a color in the range $[1, dist2deg_i]$.
- c_i, the color of node i.
- $flag_i$, true if node i wants to change its color, otherwise false.
- p_i, a pointer to a node $j \in N[i]$, signalling that j should change its color. If no such node exists then $p_i = null$.
- s_i, the current color of p_i.
- t_i, a color that p_i could change to.
- $coloring_i$, true if node i is in the process of recoloring itself. This requires that $p_j = i$ for all $j \in N[i]$.

Next, we describe two functions that are used by the algorithm. Here node i is the calling processor and in the *NextColor* function $q \in N[i]$.

NextColor(i, q) is used by node i for calculating which color node q could have. The function returns both the current color of q and the smallest color $\geq c_q$, that q can have without causing any coloring conflicts with nodes in $N[i] - \{q\}$.

```
NextColor(i, q):
    $w = \min\{a : a \geq c_q \wedge (\forall z \in N[i] - \{q\} : a \neq c_z)\}$
    return $(c_q, w)$
```

CorrectPointer(i) is used for determining the next node in $N[i]$ that should change its color (or at least have it verified). A node $j \in N[i]$ needs to attempt a recoloring if either $flag_j = true$ or if $\exists k \in N[i] - \{j\}$ such that $c_j = c_k$. If there are several candidates the one with the lowest ID is chosen. The function returns a triplet (q, c_q, w) where q is the next node in $N[i]$ that should attempt a recoloring and w is the smallest color $\geq c_q$ that does not cause a conflict with nodes in $N[i] - \{q\}$.

```
CorrectPointer($i$):
    $q = \min\{\, j \in N[i] : (flag_j = true \vee \exists k \in N[i] - \{j\} : (c_j = c_k))\}$
    if  $q \neq null$
    then return ($q$, NextColor($i, q$))
    return ($null, null, null$)
```

Before formally specifying the algorithm, we give the intuition for each rule.

Distance-1: Set $dist1deg_i$ to the size of $N[i]$.

Distance-2: Set $dist2deg_i$ to an upper bound on the size of $N^2[i]$. Note that this rule double counts two nodes in $N(i)$ if they are themselves neighbors or if they have a common neighbor.

Reset: Set $coloring_i$ to $false$ if it is incorrectly $true$. It could be that either $coloring_i$ was incorrectly $true$ in an initial configuration or that i has to abandon an attempt to recolor itself. This is detected if some node $j \in N[i]$ does not point to i (i.e. $p_j \neq i$) or if $flag_i \neq true$.

Notify neighbor: Set p_i to point to the lowest numbered node $j \in N[i]$ that either wants to recolor itself (i.e. $flag_j = true$) or needs to recolor itself because it has a color that conflicts with some node in $N[i]$. Also, set t_i to a suggested new color for $p_i = j$ and set $s_i = c_j$ to indicate that the values have been set in response to the current value of c_j. Note that once a node j has started to recolor itself, as indicated by $coloring_j = true$, no node i that is pointing to j can change its pointer-value. That is, p_i must continue to point to j as long as j is recoloring itself.

Respond to color: If the neighbor p_i is recoloring itself and has changed its color, acknowledge the color change in s_i and if the color s_i conflicts with a color in $N[i]$, use t_i to suggest the next higher possible color for p_i to use. Recall that if the node p_i is recoloring itself (indicated by $coloring_{p_i} = true$) then the node p_i will cycle through possible colors. For each such color, node i must acknowledge the color change (by setting s_i to the new color) and signalling if it accepts the new color (by setting $t_i = c_{p_j}$) or if p_i should change to a higher color ($t_i > c_{p_j}$).

Need new color: If i needs to recolor, set $flag_i = true$, signalling a request to recolor. If a node $j \in N[i]$ is pointing to i ($p_j = i$) while both acknowledging the current color of i ($s_j = c_i$) and requesting that i change its color ($t_j > c_i$), then node i must perform a recoloring. Node i signals to its neighbors that it wants to do so by setting $flag_i = true$. Alternatively, if i has $dist2deg_i < c_i$ then it should also set $flag_i = true$ to indicate that it needs to change its color. Note that the only place that i can later set $flag_i = false$ is in the *Done recoloring* method.

Start recoloring: If every node in $N[i]$ agrees that i is the next to recolor, i begins the recoloring process by setting $coloring_i = true$ and starting with color 1. A node can only start to recolor itself when it has set $flag_i = true$ and each node $j \in N[i]$ is pointing to it ($p_j = i$), while at the same time acknowledging the current color of i (by setting $s_j = c_i$). The node i then sets $coloring_i = true$, locking all other nodes in $N[i]$ from changing their p-values until i has completed the recoloring.

Change color: If all neighbors have acknowledged the current color c_i and at least one neighbor knows of a conflict with c_i, then change i's color. Whenever node i has proposed a new color it must wait for this to be acknowledged by all nodes in $N[i]$ ($s_j = c_i$). If at least one $j \in N[i]$ indicates that there is a conflict with the current color choice (by setting $t_j > c_i$) then i must try the next possible color (i.e., the maximum color over all t_j values).

Done recoloring: If all neighbors have acknowledged the current color c_i and no neighbor knows of a conflict with c_i, set $flag_i = false$ and $coloring_i = false$, indicating that i has completed its recoloring process. Note that in this case there is no distance-2 conflict with c_i. Note also that this is the only routine that sets $flag_i$ to false.

The rules are executed in the given order, meaning that a rule is never executed unless all the previous rules cannot be executed.

Algorithm 1

```
Distance-1:
    if   $dist1deg_i \neq |N[i]|$
    then $dist1deg_i = |N[i]|$
Distance-2:
    if   $dist2deg_i \neq (\sum_{j \in N(i)} dist1deg_j) - dist1deg_i + 2$
    then $dist2deg_i = (\sum_{j \in N(i)} dist1deg_j) - dist1deg_i + 2$
Reset:
    if   $(coloring_i = true)$ and $((\exists j \in N[i] : p_j \neq i)$ or $flag_i = false)$
    then $coloring_i = false$
Notify neighbor:
    if   $(p_i = null$ or $coloring_{p_i} = false)$ and $((p_i, s_i, t_i) \neq$ CorrectPointer($i$))
    then $(p_i, s_i, t_i) =$ CorrectPointer($i$)
Respond to color:
    if   $(p_i \neq null)$ and $(coloring_{p_i} = true)$ and $((s_i, t_i) \neq$ NextColor($i, p_i$))
    then $(s_i, t_i) =$ NextColor($i, p_i$)
Need new color:
    if   $(flag_i = false)$ and $((\exists j \in N[i] : (p_j = i \wedge s_j = c_i \wedge t_j > c_i)) \vee$
         $(1 \leq dist2deg_i < c_i))$
    then $flag_i = true$
Start recoloring:
    if   $(flag_i = true)$ and $(\forall j \in N[i] : (p_j = i \wedge s_j = c_i))$ and $(coloring_i = false)$
    then $coloring_i = true$
         $c_i = 1$
```

```
Change color:
    if   $(coloring_i = true)$ and $(\forall j \in N[i] : (p_j = i \wedge s_j = c_i))$ and $(\exists j \in N[i] : t_j > c_i)$
    then $c_i = max\{t_j : j \in N[i]\}$

Done recoloring:
    if   $(coloring_i = true)$ and $(\forall j \in N[i] : (p_j = i \wedge s_j = c_i \wedge t_j = c_i))$
    then $coloring_i = false$
         $flag_i = false$
```

Figure 1 shows a possible execution of Algorithm 1. The initial graph consists of four nodes i, j, k, and l where $i > k$ and with colors as shown in Figure 1a. We assume that the *Distance-1*, *Distance-2*, and *Reset* rules have stabilized before our example starts. Since $c_i = c_k$ node j will first execute a *Notify neighbor* move and set $p_j = k$, $s_j = 2$, and $t_j = 3$. This will force node k to execute a *Need new color* move and set $flag_k = true$. This will again be followed by nodes k and l executing *Notify neighbor* moves giving the configuration shown in Figure 1b. At this point all nodes in $N[k]$ are pointing to k, each with an s-value equal to c_k. Since $t_j > c_k$ it follows that k now can execute a *Start recoloring* move, setting $coloring_k = true$ and $c_k = 1$. From this point no node in $N[k]$ can change its p-value until $coloring_k = false$.

All three nodes in $N[k]$ are now ready to respond to the current value of c_k through *Respond to color* moves. In doing so both nodes j and k will set their t-values $> c_k$ since both of them can see that $c_j = c_k$. This will give the configuration in Figure 1c.

Now k will execute two *Change color* moves, each followed by all nodes in $N[k]$ acknowledging the change in color by executing a *Respond to color* move. This will first increase the value of c_k to 3 (Figure 1d) and then to 4 (Figure 1e). At this point there are no conflicts between c_k and the nodes in $N^2(i)$. This

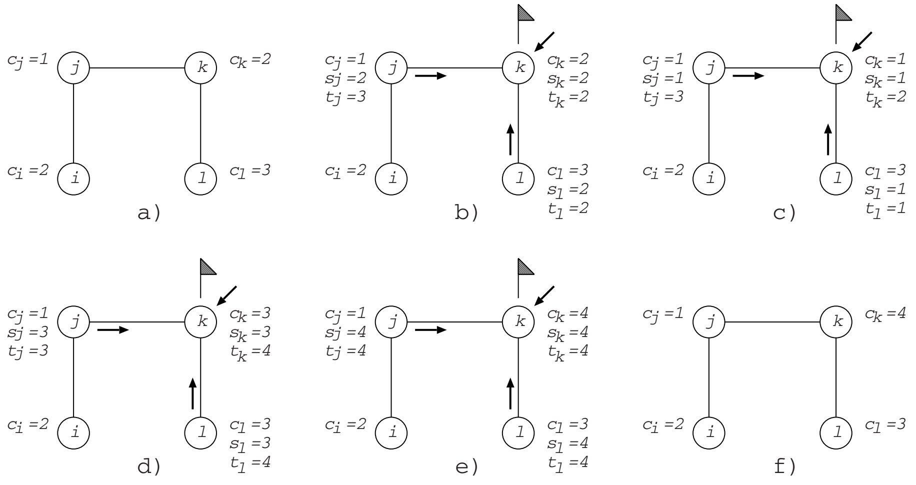

Fig. 1. A possible execution of Algorithm 1

is indicated by the fact that all t-values in $N[k]$ are equal to c_k. Thus node k can execute a *Done recoloring* move which will again be followed by each node in $N[k]$ executing a *Notify neighbor* move to set their p, s, and t values to *null*, finally giving the coloring shown in Figure 1f.

4 Correct Stabilization

In this section we show that when Algorithm 1 is stable the c_i values define a legal distance-2 coloring where no node has a color that is larger than $\delta_i\Delta+1 \leq \Delta^2+1$. We start by showing that each node has an effective bound on the size of $N^2[i]$.

Lemma 1. *In a stable configuration every node i has $dist2deg_i \leq \delta_i\Delta + 1$.*

Proof. Note first that in a stable configuration it follows from the *Distance-1* rule that every node must have $dist1deg_i = \delta_i + 1 \leq \Delta + 1$. The *Distance-2* rule then implies that $dist2deg_i = (\Sigma_{j\in N(i)} dist1deg_j) - dist1deg_i + 2 = (\Sigma_{j\in N(i)}(\delta_j + 1)) - \delta_i + 1 \leq \delta_i\Delta + 1$. ■

Since, for any node i, we have that $|N^2[i]| \leq \delta_i\Delta + 1$, it follows that it is possible to achieve a legal distance-2 coloring where i has a color in the range $[1, \delta_i\Delta + 1]$. To see this, it is sufficient to note that there must be a color in the range $[1, |N^2[i]|]$ not used by the nodes in $N^2(i)$. This color can always be assigned to node i.

Next, we show that when the algorithm is stable no node is actively trying to change color.

Lemma 2. *In any stable configuration, $coloring_i = false$ for every node i.*

Proof. If there exists a node i with $coloring_i = true$, then every node $j \in N[i]$ must have $p_j = i$, otherwise i could execute a *Reset coloring* move. Similarly, there must be at least one node $j \in N[i]$ with $s_j \neq c_i$ or $t_j \neq c_i$ (or both); otherwise i could execute a *Done recoloring* move. A node $j \in N[i]$ where $s_j \neq c_i$ is eligible for a *Respond to color* move, since c_i is *NextColor*(j, i)'s first return value. Thus we may assume that some j has $t_j \neq c_i$. If $t_j < c_i$ then again node j is eligible for a *Respond to color* move, while if $t_j > c_i$ then i is eligible for a *Change color* move. This is a contradiction. It follows that $coloring_i = false$ in a stable configuration. ■

Lemma 3. *In any stable configuration the following statements are true for every node i: (i) $flag_i = false$, (ii) For every pair of distinct nodes $j, k \in N[i], c_j \neq c_k$, and (iii) $c_i \leq dist2deg_i$.*

Proof. This proof is omitted due to space limitations.

We can now state the main result of this section.

Theorem 1. *In a stable configuration the c values define a legal distance-2 coloring where every node i satisfies $c_i \leq \delta_i\Delta + 1$.*

Proof. This follows directly from Lemmas 1 and 3. ■

5 Step Complexity

In this section we derive and prove a bound on the number of time steps needed for Algorithm 1 to stabilize, given an arbitrary initial configuration. The analysis assumes a distributed adversarial daemon. This means that in each time step a non-empty subset of eligible nodes makes one move each.

Table 1 is a summary of upper bounds on the number of time steps that might include a move of each type (i.e., each rule) before stabilization. The last column in the table references the result that proves the number of steps. The results and proofs follow the table.

Lemma 4. *There can be at most* $2(m+n)$ *time steps containing* Distance-1 *or* Distance-2 *moves.*

Proof. Each node can at most make one *Distance-1* move. After this move a node can make one initial *Distance-2* move and then only after each node in $j \in N(i)$ changes its $dist1deg_j$ value. Thus a node i can at most make a total of $\delta_i + 2$ *Distance-1* and *Distance-2* moves. Since $\sum_{i \in V}(\delta_i + 2) = 2n + 2m$ we get that the total number of *Distance-1* and *Distance-2* moves is bounded by $2(m+n)$. ■

Lemma 5. *There can be at most* n *time steps containing* Reset *moves that start with* `coloring = true` *and* `flag = false`*.*

Proof. Each node i can make one such initial *Reset* move. Any subsequent *Reset* move must follow a *Start recoloring* move and come before any *Done recoloring*

Table 1. Summary of Step Complexity

Move	# Steps (Upper Bound)	Complexity	Proof
Distance-1	n	$= O(m)$	Lemma 4
Distance-2	$n + 2m$	$= O(m)$	Lemma 4
Reset `coloring = false`	0	$= O(1)$	Lemma 5
Reset `coloring = true` `flag = false`	n	$= O(m)$	Lemma 5
Reset `coloring = true` `flag = true`	$n + 8m\Delta$	$= O(\Delta m)$	Lemma 9
Notify neighbor	$9n + 16m$	$= O(m)$	Lemma 12
Respond to color	$4n + 14m(\Delta^2 + \Delta + 1)$	$= O(\Delta^2 m)$	Lemma 14
Need new color	$3n$	$= O(m)$	Lemma 11
Start recoloring	$5n + 8m\Delta$	$= O(\Delta m)$	Lemma 10
Change color	$4n + 8m\Delta$	$= O(\Delta m)$	Lemma 13
Done recoloring	$4n$	$= O(m)$	Corollary 2

move, since this is the only occasion when *coloring*$_i$ $= true$. However, *Start recoloring* is only executed when $flag_i = true$ and the only move that can set $flag_i = false$ is *Done recoloring*, which also sets *coloring*$_i$ $= false$. Thus any subsequent *Reset* move cannot be triggered by $flag_i = false$. ■

Before investigating the step complexity of the remaining rules, we examine how each move can or cannot cause a transition between different states of a node i. The states we are interested in depend on the possible values of *coloring*$_i$ and $flag_i$. Figure 2 shows the state transition diagram. Note that four rules are not shown in the figure since they do not impact the analysis: *Distance-1*, *Distance-2*, *Respond to color*, and *Notify neighbor*.

The transitions in Figure 2 are defined by the predicates and commands of the rules. If *coloring*$_i$ $= true$ while $flag_i = false$ then i will execute a *Reset* move and set *coloring*$_i$ $= false$. From this configuration the only move that can affect the values of *coloring*$_i$ and $flag_i$ is a *Need new color* move that sets $flag_i = true$. From that state the only possible move is *Start recoloring*, which sets *coloring*$_i$ $= true$. From the configuration *coloring*$_i$ $= true$ and $flag_i = true$ node i can execute a number of *Change color* moves, but these do not change the values of either *coloring*$_i$ or $flag_i$. It is possible that i executes a *Reset* move, setting *coloring*$_i$ $= false$, if some $j \in N[i]$ has $p_j \neq i$. The other possibility is that i executes a *Done recoloring* move and sets *coloring*$_i$ $= false$ and $flag_i = false$. In addition to these moves, i can also execute a *Distance-1*, a *Distance-2*, a *Notify neighbor*, or a *Respond to color* move. These do not affect *coloring*$_i$ and $flag_i$ and are not shown in Figure 2.

A *recoloring sequence* by node i consists of a sequence of moves beginning with *Start recoloring* (the transition from state D to state C) and ending with *Done recoloring* (the transition from state C to state B). Note that i can abort an initiated recoloring sequence by executing a *Reset* move and transitioning from state C back to state D. This can only happen if some $j \in N(i)$ executes a *Notify neighbor* move, which will then set $p_j \neq i$, during the same step that

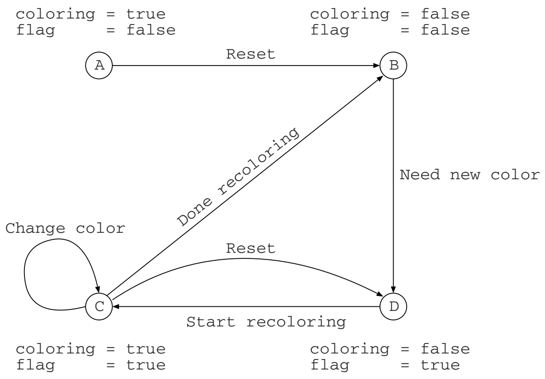

Fig. 2. States of Algorithm 1 with respect to *coloring* and *flag* values

i executes the initial *Start recoloring* move. Otherwise $p_j = i$ will remain true as long as $coloring_i = true$. If i does not abort, we call the recoloring sequence *complete*. A complete recoloring sequence is *correct* if i has been assigned a color $r \leq \delta_i \Delta + 1$ not used by any node in $N^2(i)$ when the recoloring sequence ends.

We now consider a complete recoloring sequence α executed by a node i where α is not the first complete recoloring sequence executed by i. Let g be the time step when i enters α by executing a *Start recoloring* move and let h be the time step when i executes its first *Change color* or *Done recoloring* move in α, whichever comes first. Also, for a particular $j \in N[i]$, let f be the last time step prior to g when j executes a *Notify neighbor* move. Note that j sets $p_j = i$ in time step f. Then $f < g < h$ and $p_j = i$ will remain true for at least the time span $[f, g-1]$ and also after time step $h-1$. In the same manner $coloring_i = true$ in the time span $[g, h]$ (and possibly longer).

Our next result considers the values that t_j can take on prior to time step h.

Lemma 6. *Let g be the time step when node i executes the* Start recoloring *move in a non-initial complete recoloring sequence α, and let h be the earliest time step in α that i executes a* Change color *or* Done recoloring *move. For any particular $j \in N[i]$, let f be the last time step prior to g when j executes a* Notify neighbor *move.*

After time step $h-1$ and before time step h: for every $j \in N[i]$ the following are true: $p_j = i$, $s_j = c_i$, and either $t_j = 1$ or t_j is equal to the lowest or second lowest unused color in $N[j] - \{i\}$.

Proof. This proof is omitted due to space limitations.

Now we have established the different possible values that each t_j for $j \in N[i]$ can have just after time step $h-1$. The next two results are needed to make sure that i starts to select a new color once i has executed a *Start recoloring* move.

Corollary 1. *Let g be the time step when node i executes the* Start recoloring *move in a non-initial complete recoloring sequence α, and let h be the earliest time step in α that i executes a* Change color *or* Done recoloring *move.*

If the Need new color *move by i that set $flag_i = true$ prior to i entering α was caused by $dist2deg_i < c_i$ then for each $j \in N[i]$ the value of t_j will be pointing to the lowest unused color in $N[j] - \{i\}$ after time step $h-1$.*

Proof. The *Need new color* rule requires that $1 \leq dist2deg_i < c_i$. Thus since $1 < c_i$ the value of c_i will be reduced to 1 when i executes a *Start Recoloring* move in time step g. ■

Lemma 7. *Let g be the time step when node i executes the* Start recoloring *move in a non-initial complete recoloring sequence α, and let h be the earliest time step in α that i executes a* Change color *or* Done recoloring *move.*

If the Need new color *move by i that set $flag_i = true$ prior to i entering α was not caused by $dist2deg_i < c_i$ then there must be some $j \in N[i]$ that has $t_j > 1$ after time step $h-1$.*

Proof. Since i exited the previous recoloring sequence with every $j \in N[i]$ satisfying $t_j = c_i$ at the time step in which the *Done recoloring* move was executed, there must exist some node $j \in N[i]$ that executed a *Notify neighbor* move and set $p_j = i$, $s_j = c_i$, and $t_j > c_i$ prior to i executing the *Need new color* move to enter α. At the time j executed the *Notify neighbor* move, there must have existed at least one node $k \in N[j] - \{i\}$ such that $c_k = c_i$ and $i < k$. Note that k cannot have changed color between this point and time step h, because $p_j \neq k$. (As long as $c_i = c_k$ *CorrectPointer(j)* will never set $p_j = k$ in *Notify neighbor* since $i < k$.) Thus we can conclude that the coloring conflict between i and k still exists after time step $h - 1$. From this it follows that when p_j was last set to i the value of t_j must have been set to a value greater than c_i. ■

We can now show that α must be correct.

Lemma 8. *Let α be a recoloring sequence for node i. When i exits α there is no node in $N^2(i)$ with the same color as i and $c_i \leq \delta_i \Delta + 1$.*

Proof. This proof is omitted due to space limitations.

Note that every node $j \in N[i]$ must execute at least one *Respond to color* move before i executes a *Done recoloring* move. Thus the value of $dist1deg_j$ for each $j \in N[i]$ must be correct when i exits α. Similarly, $dist2deg_i$ must be correct when i exits α.

We have now shown that every complete recoloring sequence (except maybe the first) will result in a node i having a distinct color among all the nodes in $N^2[i]$. However, there is a possibility that i does not complete a recoloring sequence and this may result in a coloring conflict. But as the proof of the following result shows, the non-complete recoloring sequences can be subsumed in the complete recoloring sequences.

Theorem 2. *No node will perform more than three complete recoloring sequences.*

Proof. From Lemma 8 it follows that a non-initial complete recoloring sequence by a node i will result in c_i being unique relative to the colors used by the nodes in $N^2(i)$. An incomplete recoloring sequence by i will result in i executing a *Reset* move with $c_i = 1$. Thus if i has received a legal color such that $c_i > 1$, then no node in $N^2(i)$ will receive the same color as i.

Now assume a node i has executed its second complete recoloring sequence. If $c_i > 1$ then no node $k \in N^2(i)$ can exit a subsequent recoloring sequence with $c_k = c_i$. But if k, where $i < k$, performs a *Reset* move right after executing a *Start recoloring* move and if $c_i = 1$ then we get $c_i = c_k$. This would force i to perform a new recoloring sequence which would result in $c_i > 1$ (since k cannot change color until i has done so) and thus no further recoloring sequences would be needed by i. ■

Now that we have shown that each node can execute at most three complete recoloring sequences it is fairly straight forward to count the number of different moves each node can make. The following results state these counts without showing the straight-forward proofs, in the interest of saving space.

Corollary 2. *There can be at most* $4n$ *time steps containing* Done recoloring *moves.*

Lemma 9. *There can be at most* $n + 8m\Delta$ *time steps containing* Reset *moves that start with* `coloring = true` *and* `flag = true`*.*

Lemma 10. *There can be at most* $5n + 8m\Delta$ *time steps containing* Start recoloring *moves.*

Lemma 11. *There can be at most* $3n$ *time steps containing* Need new color *moves.*

Lemma 12. *There can be at most* $9n + 16m$ *time steps containing* Notify neighbor *moves.*

Lemma 13. *There can be at most* $4n+8m\Delta$ *time steps containing* Change color *moves.*

Lemma 14. *There can be at most* $4n + 14m(\Delta^2 + \Delta + 1)$ *time steps containing* Respond to color *moves.*

Theorem 3. *Algorithm 1 stabilizes after* $O(\Delta^2 m)$ *time steps.*

Proof. The result follows directly from Lemmas 4, 5, 9, 10, 11, 12, 13, 14 and Corollary 2. See Table 1 for a summary. ■

We note that the same time step analysis holds for a sequential adversarial daemon. The main difference between a distributed and sequential adversarial daemon is that with the sequential one, we can show that any node that has gone through at least two complete recoloring sequences will end up with the lowest color not used by any node in $N^2(i)$, as opposed to the second lowest for the distributed daemon. However, in both cases one cannot guarantee that each node has been assigned the lowest available color in a stable solution, as there might be nodes that do not change color during the execution of the algorithm.

The analysis for a fair daemon (sequential or distributed) is not much different from the one presented here and gives a round complexity of $O(\Delta m)$. Although we omit the details due to space considerations it is not hard to see that when a node i has executed a *Change color* move, all nodes in $N[i]$ can respond to this in one round. Thus the complexity of the *Respond to color* moves, which are the most frequent moves, are lowered from $O(\Delta^2 m)$ moves for the adversarial daemon to $O(\Delta m)$ rounds for the fair daemon. To see that this is also a lower bound it is sufficient to consider a complete graph where every node starts with the same initial color.

6 Concluding Remarks

We note that Algorithm 1 can easily be modified to solve various other restricted coloring problems. For instance, colors could be selected from a finite list of available colors (a so called *list coloring*) or it could be required that $|c_i - c_j| > a$ when i and j are distance-2 neighbors, where a is some positive constant.

However, Algorithm 1 cannot in its current form produce a Grundy coloring (i.e. where each node i has the lowest available color in $N^2(i)$) as it cannot detect available free colors that are smaller than the current (correct) color. One solution to this could be to let each node set $flag_i = true$ with some small probability.

We also note that although we require that every node has a unique identifier, it is not hard to show that it suffices that each identifier is unique within distance-2 for the algorithm to run correctly.

References

1. Aardal, K.I., van Hoesel, S.P.M., Koster, A.M.C.A., Mannino, C., Sassano, A.: Models and solution techniques for frequency assignment problems. Ann. Op. Res. 153, 79–129 (2007)
2. Chaudhuri, P., Thompson, H.: A self-stabilizing distributed algorithm for edge-coloring general graphs. Aust. J. Comb. 38, 237–248 (2007)
3. Gairing, M., Goddard, W., Hedetniemi, S.T., Kristiansen, P., McRae, A.A.: Distance-two information in self-stabilizing algorithms. Par. Proc. L. 14, 387–398 (2004)
4. Ghosh, S., Karaata, M.H.: A self-stabilizing algorithm for coloring planar graphs. Dist. Comp. 7, 55–59 (1993)
5. Goddard, W., Hedetniemi, S.T., Jacobs, D.P., Srimani, P.K.: Self-stabilizing algorithms for orderings and colorings. Int. J. Found. Comp. Sci. 16, 19–36 (2005)
6. Gradinariu, M., Johnen, C.: Self-stabilizing neighborhood unique naming under unfair scheduler. In: Sakellariou, R., Keane, J.A., Gurd, J.R., Freeman, L. (eds.) Euro-Par 2001. LNCS, vol. 2150, pp. 458–465. Springer, Heidelberg (2001)
7. Gradinariu, M., Tixeuil, S.: Self-stabilizing vertex coloring of arbitrary graphs. In: OPODIS 2000, pp. 55–70 (2000)
8. Hedetniemi, S.T., Jacobs, D.P., Srimani, P.K.: Linear time self-stabilizing coloring. Inf. Proc. Lett. 87, 251–255 (2003)
9. Huang, S.-T., Hung, S.-S., Tzeng, C.-H.: Self-stabilizing coloration in anonymous planar networks. Inf. Proc. Lett. 95, 307–312 (2005)
10. Kosowski, A., Kuszner, L.: Self-stabilizing algorithms for graph coloring with improved performance guarantees. In: Rutkowski, L., Tadeusiewicz, R., Zadeh, L.A., Żurada, J.M. (eds.) ICAISC 2006. LNCS (LNAI), vol. 4029, pp. 1150–1159. Springer, Heidelberg (2006)
11. Shukla, S., Rosenkrantz, D., Ravi, S.: Developing self-stabilizing coloring algorithms via systematic randomization. In: Proc. Int. Workshop on Par. Process., pp. 668–673 (1994)
12. Shukla, S.K., Rosenkrantz, D., Ravi, S.S.: Observations on self-stabilizing graph algorithms for anonymous networks. In: Proc. of the Second Workshop on Self-stabilizing Systems, pp. 7.1–7.15 (1995)
13. Sun, H., Effantin, B., Kheddouci, H.: A self-stabilizing algorithm for the minimum color sum of a graph. In: Rao, S., Chatterjee, M., Jayanti, P., Murthy, C.S.R., Saha, S.K. (eds.) ICDCN 2008. LNCS, vol. 4904, pp. 209–214. Springer, Heidelberg (2008)
14. Sur, S., Srimani, P.K.: A self-stabilizing algorithm for coloring bipartite graphs. Inf. Sci. 69, 219–227 (1993)
15. Tzeng, C.-H., Jiang, J.-R., Huang, S.-T.: A self-stabilizing $(\delta + 4)$-edge-coloring algorithm for planar graphs in anonymous uniform systems. Inf. Proc. Lett. 101, 168–173 (2007)