Self-stabilizing Algorithms for Graph Coloring with Improved Performance Guarantees

Adrian Kosowski and Łukasz Kuszner

Gdańsk University of Technology, Poland
Department of Algorithms and System Modeling
{kosowski,kuszner}@sphere.pl

Abstract. In the self-stabilizing model we consider a connected system of autonomous asynchronous nodes, each of which has only local information about the system. Regardless of the initial state, the system must achieve a desirable global state by executing a set of rules assigned to each node. The paper deals with the construction of a solution to graph coloring in this model, a problem motivated by code assignment in wireless networks.
A new method based on spanning trees is applied to give the first (to our knowledge) self-stabilizing algorithms working in a polynomial number of moves, which color bipartite graphs with exactly two colors. The complexity and performance characteristics of the presented algorithms are discussed for different graph classes.

Key words: self-stabilizing algorithm, graph coloring, bipartite graph

1 Introduction

Self-stabilizing algorithms are a fundamental branch of fault-tolerant computing, first introduced by Dijkstra [4] in 1974. The resilience of the system guarantees that the system will reach a desirable state even after a period of malfunction or an unexpected change of topology. These properties render self-stabilizing algorithms feasible in contexts where ordinary distributed algorithms prove insufficiently stable, especially when applied in ad-hoc wireless networks [11, 15] and in state-of-the-art systems of nano-electrical sensors placed in a hostile environment [3]. The most relevant self-stabilizing algorithms deal with assigning codes or values to nodes and the links between them, thus facilitating communication and role division of the respective units. In this paper we focus our attention on the fundamental problem of coloring the system graph in such a way that neighbouring nodes receive different colors and the number of colors used is as small as possible. This process of graph coloring in a distributed setting has numerous applications, including such areas as code assignment for wireless networks [1, 11, 13].

The model of execution. In all considerations it is assumed that the self-stabilizing system consists of nodes connected by communication channels. Each node maintains variables which determine its *local state*. The *global state* of the system is the union of all local states of its nodes. Thus we model a system by a connected graph $G = (V, E)$, where vertex set V corresponds to system nodes and the set of edges E denotes communication links between them. The algorithm for each vertex v is given as a set of rules of the form **if** $p(v)$ **then** A, where $p(v)$ is a predicate over local states of v and its neighbors, and A is an action changing a local state of v (a *move* of v). A vertex v becomes *active* when $p(v)$ is true, otherwise v is *stable*. The execution of the algorithm is controlled by a *scheduler* which allows some non-empty subset of active vertices to perform a simultaneous move defined by the rules for the respective nodes; this is referred to as a single *step*. If all vertices of the graph are stable, we say that the system is *stable* and the execution of the algorithm is complete. A *self-stabilizing system* is one which is able to achieve a legitimate global state starting from any possible global state [4].

The time complexity of such self-stabilizing algorithms is expressed in terms of the number of steps of the algorithm, or the number of moves performed by respective nodes. In this paper the total number of moves performed by all nodes is treated as the primary measure. This value is closely related to (and never less than) the number of steps of the system, and may moreover be regarded as the "energy cost" of performing the stabilization process.

Problem definition and related work. A self-stabilizing algorithm is said to perform *graph coloring* if any stable state reached by the system may be interpreted as an assignment of non-negative integer color values $c(v)$ to all system nodes $v \in V$ such that $c(v_1) \neq c(v_2)$ if $\{v_1, v_2\} \in E$. The goal of optimization is minimizing the largest color value used.

Self-stabilizing graph coloring algorithms have been intensively studied in literature. In 1993 Ghosh and Karaata [6] presented an algorithm for coloring planar graphs with at most 6 colors, assuming that all vertices have unique identifiers. This result was later improved to allow operation with bounded variable values and without identifiers in [12] and generalized for a wider class of graphs in [8]. Also in 1993, Sur and Srimani [16] gave an algorithm for exact coloring of bipartite graphs. All of the discussed papers prove the finite stabilization time of the algorithm, but leave bounds on time complexity as open problems. Later on three algorithms with improved time constraints for arbitrary graphs were presented in [9], based on a greedy assignment technique. More recently, a linear-time algorithm for coloring arbitrary graphs with $\Delta + 1$ colors (where Δ denotes the maximum vertex degree) was given in [10].

Our results. In this paper we present new graph coloring algorithms which stabilize in a polynomial number of moves and find optimal colorings for bipartite (2-colorable) graphs and cacti, thus improving the results of [16] and [9]. To achieve the results we a use recent algorithm for spanning tree construction [14].

We give different versions of the graph coloring algorithm. Depending on the configuration of the scheduler, the algorithm either operates without vertex identifiers (for a system with a *central daemon*, i.e. when the scheduler allows the move of at most one vertex at a time, Section 3) or requires unique vertex identifiers (for a system with a *distributed daemon*, i.e. when no additional assumptions are made about the scheduler, Section 4). For these cases the number of moves performed by the algorithms is bounded by $O(mn^3 \operatorname{diam}(G))$ and $O(mn^3 \Delta \operatorname{diam}(G))$ respectively, where $\operatorname{diam}(G) < n$ is the distance between the furthest pair of nodes of the system. Finally, in Section 5 we analyze the performance of the presented algorithms color for arbitrary graphs, stating that they use not more than 2Δ colors, and that this bound is tight even for series-parallel graphs.

2 Preliminaries: Local state definition and assumptions

Let $G = (V, E)$ be the system graph with vertex set V and edge set E. By $n = |V|$ and $m = |E|$ we denote the number of vertices and the number of edges, respectively. In addition, let $N(v) = \{u : (u, v) \in E\}$ be the *open neighborhood* of vertex $v \in V$, and let $\deg(v) = |N(v)|$ be the *degree* of v. Each vertex has constant read-only access to the local states of all vertices in its neighbourhood throughout the operation of the algorithm. When performing a move, a vertex may write to the variables of its local state. The entire move of a vertex is seen as an atomic operation by its neighbours.

In our approach we provide a *semi-uniform* algorithm, which means that exactly one of the nodes of set V, called a *root* and denoted r, needs to be distinguished and operate using a different set of rules than the other vertices. Each node has two local variables f and c, both non-negative integers, which reflect the two parts of coloring algorithm. State variable c is simply the sought color of a given vertex.

Variable f is used during the construction of a spanning tree of graph G. A *spanning tree* $T = (V, E')$ of $G = (V, E)$ is a subgraph of G consisting of the same set of nodes V, but only a subset $E' \subseteq E$ of edges such that there exists exactly one path between every pair of nodes in T. In particular, this means that each vertex $v \in V$ other than the root r has exactly one neighbor on the path connecting v and r in T, known as the *parent* of v in T. In this context, the interpretation of variable f is as follows. Consider vertex v and let us choose u such that $f(u) = \min_{w \in N(v)} f(w)$ and u is the first [1] vertex among the neighbors of v with such a property. If $f(u) < f(v)$ then we say that u is the parent of v and denote $p(v) = u$. Other case we say that a parent vertex does not exist for v in the current system configuration and denote it by $p(v) = \text{null}$. If all vertices save the root have parent vertices, then the parenthood relation defines the sought spanning tree T of G. For the distinguished vertex we set permanently $f(r) := 0$ and $c(r) := 0$.

[1] To be able to say "first" we must assume that the neighbors are somehow ordered. This is not a strong assumption as long as a node is able to distinguish between its neighbors. For example, if the neighbors of v are stored in the form of a list, the order can be given according to the list sequence.

3 A graph coloring algorithm under a central daemon optimal for bipartite graphs

The concept of the graph coloring algorithm presented in this section is based on a simple observation. We use the *parity function* $A : \mathbb{N} \cup \{0\} \to \{0, 1\}$, defined as $A(c) \equiv c \mod 2$, to partition the space of colors available for each vertex into two subsets — the set of *even colors* $\{c \in \mathbb{N} \cup \{0\} : A(c) = 0\}$ and the set of *odd colors* $\{c \in \mathbb{N} \cup \{0\} : A(c) = 1\}$.

The constructed algorithm will attempt to assign to each vertex the smallest possible odd color if the color of its parent is even and the smallest possible even color when the color of its parent is odd. For a given vertex v such a color is denoted as $\gamma(v)$:

$$\gamma(v) = \min\{k \in \mathbb{N} \cup \{0\} \,:\, A(k) \neq A(c(p(v))) \wedge \forall_{u \in N(v)} k \neq c(u)\}.$$

Observe that $\gamma(v)$ is properly defined iff the parent of v exists; otherwise, we assume $\gamma(v) = c(v)$.

We can now write the graph coloring algorithm for the case when the scheduler selects exactly one active node at a time to make a move, which is often referred to in literature as self-stabilization under a *central daemon.* At first glance such a model can be seen as very strong, however it is equivalent to one where only local mutual exclusion of neighboring nodes is guaranteed. Moreover, there exist protocols to convert algorithms designed for the central daemon model to weaker ones [2, 5].

Algorithm 1: Graph coloring under a central daemon

`F`: **if** $v \neq r \wedge f(v) \leq \min_{u \in N(v)} f(u)$
then $f(v) = \max_{u \in N(v)} f(u) + 1$

`C`: **if** $p(v) \neq \text{null} \wedge c(v) \neq \gamma(v)$
then $c(v) := \gamma(v)$

Rule `F` is the same as that used for constructing a spanning tree in Algorithm 1 [14] and is not affected by other rules, so using results from [14] at most $n \operatorname{diam}(G)$ moves using rule `F` are possible. Now we will establish a bound on the number of moves using rule `C` in between two consecutive moves using rule `F`.

Let us consider an arbitrarily chosen vertex v. We can distinguish between two types of moves according to rule `C`, namely: moves changing the parity of $c(v)$ which we call *alternating moves*, and other moves referred to as *non-alternating.*

Lemma 1 *The number of alternating moves is at most n^2 in between two consecutive moves performed according to rule* `F`*.*

Proof. Let $T = (V, E')$ be the forest in G induced the by parent relation p in such a way that $u, v \in E'$ iff $u = p(v)$ or $v = p(u)$. Observe that in each component of T there is exactly one vertex u such that $p(u)$ = null. Let r_v be such a vertex that $p(r_v)$ = null and both r_v and v belong to the same component of T, and let $[u, v]_T$ denote the only possible path from u to v in T, say $[u, v] = (u = u_0, u_1, \dots, u_k = v)$. Let us consider the following function:

$$S_T(v) = \begin{cases} S_T(p(v)) + 1, & p(v) \neq \text{null} \wedge A(c(p(v)) \neq A(c(v)) \\ S_T(p(v)), & p(v) \neq \text{null} \wedge A(c(p(v)) = A(c(v)) \\ 0, & p(v) = \text{null} \end{cases}$$

Intuitively, $S_T(v)$ can be considered as the number of edges $\{w, w'\}$ in the path $[r_v, v]_T$ for which color variable c changes parity while moving from w to w'. Obviously $0 \leq S_T(v) < n$. Observe that $S_T(v)$ does not depend on $f(x)$ for $x \notin [r_v, v]_T$. Now, let us consider the influence of an alternating move `C` performed by $x \in [r_v, v]_T$ on $S_T(v)$. As r_v cannot make move `C`, thus $x \neq r_v$. There are two possibilities. First, if $v = x$, then $S_T(v)$ increases by 1. Secondly, if $v \neq x$, then $S_T(v)$ does not change or increases by 2.

Consequently, since the initial value of $S_T(v)$ is never smaller than its final value by more than n and the value $S_T(v)$ increases each time when v makes an alternating move thus v cannot perform more than n alternating moves. The total number of moves performed is thus not more than n^2. □

Theorem 2 *Algorithm 1 stabilizes in $O(mn^3 \; diam(G))$ moves under a central daemon.*

Proof. We can now find a bound on the number of non-alternating moves in between two consecutive alternating moves. Again, non-alternating moves can be of two types: those increasing the value $c(v)$ (called *increasing moves*) and all other moves (*decreasing moves*). In between two alternating moves vertex v can perform at most one increasing move (as the first non-alternating move only). All following moves are decreasing moves. After the first move the value $c(v)$ belongs to the set of the smallest $\deg(v)$ color values of the given parity; the number of subsequent decreasing moves is obviously not more than $\deg(v) - 1$. Consequently, between two alternating moves each vertex performs not more than $\deg(v)$ non-alternating moves, thus the number of all non-alternating moves between alternating moves is not more than $2m$.

Finally, using Theorem 2 and Lemma 1 we have that the algorithm performs at most $2m$ non-alternating moves for each of at most n^2 alternating moves of rule `C`, for each of at most $n \, \mathrm{diam}(G)$ moves of rule `F`. By multiplying these values we obtain an upper bound of $O(mn^3 \, \mathrm{diam}(G))$ moves, which completes the proof. □

Bearing in mind Theorem 2, we know that the system stabilizes, thus let consider the result of the process.

Theorem 3 *Algorithm 1 finds a legal coloring of graph G.*

Proof. Suppose that the system is stable, so each node is stable. Hence, according to rule `C`, the colors of any two neighboring vertices are different. □

Theorem 4 *Algorithm 1 finds a legal coloring of an arbitrary graph and uses not more than two colors when applied to bipartite graph G.*

Proof. Suppose that the system is stable and consider obtained coloring. Legality is obvious. It remains to show that if G is bipartite no more than two colors has been used.

Now, suppose to the contrary, that G is bipartite and Algorithm 1 stabilizes using more than two colors. Consider a vertex v arbitrarily chosen among of vertices with the largest value of color c. Observe that for each neighbor $u \in N(v)$ the parity of $c(v)$ and $c(u)$ is different (otherwise G would have to contain an odd cycle). Thus by property 2 of function A we have $c(v) < 2$, since otherwise v would be active due to rule C, a contradiction. □

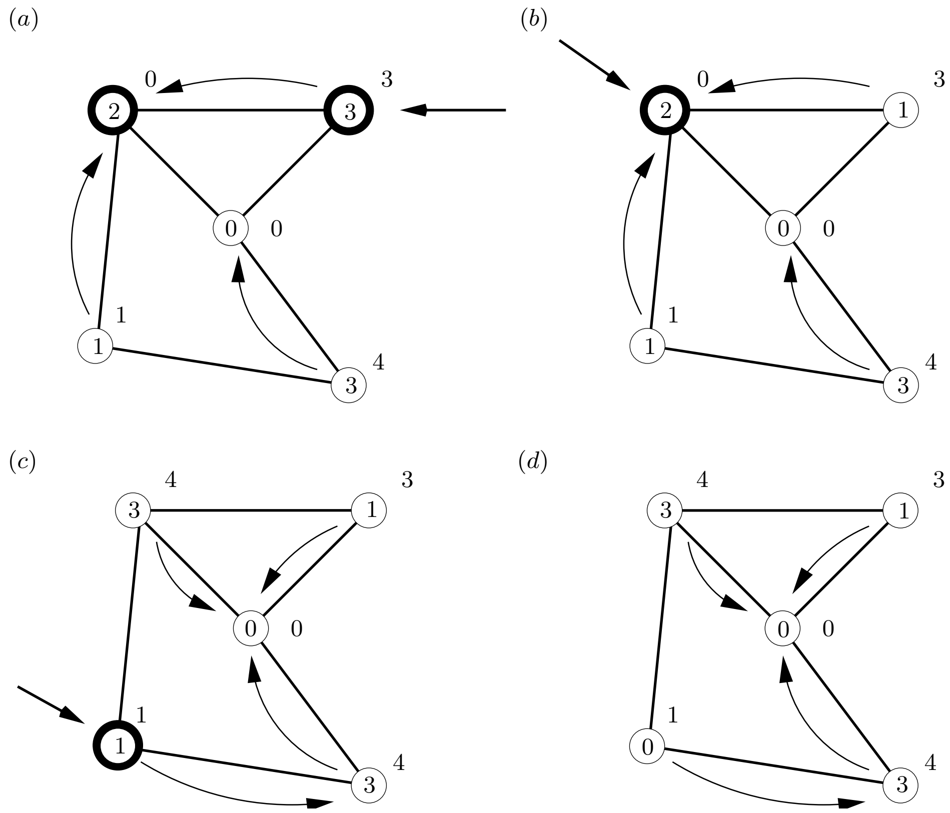

Fig. 1. Illustration of a process of Algorithm 1 (consult Subsection 3.1)

3.1 Example

In Figure 1 each of the pictures illustrates the states of the vertices in successive moves. The root vertex is placed in the middle of each picture and active vertices are marked with bold circles. The state value f of each vertex is given close to the circle, while the color value is denoted inside it.

Additional arcs show the parenthood relation between vertices and a straight pointer indicates the vertex to perform the next move. The algorithm computes a coloring in three steps and the result is visible in Figure 1(d). It can be observed that in the case of non-bipartite graphs, as in the example, we cannot expect a *Grundy coloring* [7] of G, i.e. such a coloring, that no single vertex may have its color value decreased without affecting the color other vertices.

4 A graph coloring algorithm under a distributed daemon optimal for bipartite graphs

When allowing adjacent nodes to perform moves in parallel we introduce vertex identifiers, i.e. we assume that each vertex $v \in V$ is assigned an identifier $Id(v)$ which is fixed throughout the operation of the algorithm and known to vertex v and its neighbors. Additionally, in order to allow mutual exclusion between moves of neighbouring vertices, we add a new local variable $s(v) \in \{on, off\}$ to the state of each vertex, which may be regarded as a generalisation of the concept of a semaphore.

Algorithm 2: Graph coloring under a distributed daemon

`F`: **if** $v \neq r \wedge f(v) \leq \min_{u \in N(v)} f(u)$
then $f(v) = \max_{u \in N(v)} f(u) + 1$

`C1`: **if** $p(v) \neq \text{null} \wedge c(v) \neq \gamma(v)$
then $s(v) := on$

`C2`: **if** $s(v) = on \wedge Id(v) < \min_{u \in N(v)}\{Id(u) \mid s(u) = on\}$
then $c(v) := \gamma(v)$; $s(v) := off$

Theorem 5 *Algorithm 2 determines a legal coloring of graph G in at most $O(mn^3 \Delta \operatorname{diam}(G))$ moves under a distributed daemon.*

Proof. By analogy to the correctness proof of Algorithm 1 we bound the number of moves using rule `F` by $n \log n$. Now, observe that rules `C1` and `C2` from Algorithm 2 are derived from rule `C` of Algorithm 1, and may be treated as two consecutive stages of its execution. First, in rule `C1` any vertex v whose color value is incorrect with respect to any legal state of the algorithm signals its intention of performing a color change by setting $s(v) := on$. The scheduler may then select a subset of vertices with a set value $s(v)$ to perform rule `C2`, which results in the desired change of color value $c(v)$ and unsets $s(v)$. Observe that the additional clause $Id(v) < \min\{Id(u) \mid u \in N(v) \wedge c(u) = c(v)\}$ in the condition of rule C2 serves to prevent parallel moves of neighboring vertices using this rule. Thus, the proposed algorithm behaves similarly to its counterpart with a central daemon (Algorithm 1) and we can repeat the reasoning from Lemma 1 and Theorem 2. Observe that the additional Δ-factor in the stabilization time of algorithm Algorithm 2 results from the fact that certain vertices may have a set value $s(v)$ and perform rule C2 even when their color value $c(v)$ is legitimate (since the value $\gamma(v)$ may have changed after the prior execution of rule C1 for vertex v due to a move made by one of its neighbours). □

5 Final remarks

It is important to observe that Algorithms 1 and 2 can be applied to color not only bipartite graphs, and the established bounds on the number of performed moves do in fact hold for arbitrary graphs G. Furthermore, careful analysis of the presented algorithms leads to the following statement.

Corollary 6 *Any graph G is colored by Algorithm 1 in at most $O(mn^3 \operatorname{diam}(G))$ moves, and by Algorithm 2 in at most $O(mn^3 \Delta \operatorname{diam}(G))$ moves, using not more than 2Δ colors.*

By comparison, general-purpose self-stabilizing algorithms for graph coloring (e.g. [10]) have been shown to use not more than $\Delta + 1$ colors for any graph; however, it is possible to construct bipartite graphs which are sometimes colored by exactly $\Delta + 1$ colors. Obviously, no similar pathological cases exist for Algorithms 1 and 2.

In general, Algorithms 1 and 2 do not color tripartite graphs optimally and it is possible to construct examples of 3-colorable series-parallel graphs for which Algorithms 1 and 2 may stabilize in a state using 2Δ colors.

Property 7 *There exists a family of series-parallel graphs $\{G_k\}$ such that for each G_k we have $\Delta = k$, and Algorithms 1 and 2 may lead to a coloring of G_k using 2Δ colors.*

Proof. Let $\{T_k\}$ be a family of trees with one selected vertex $s(T_k)$ in each, defined constructively by the following procedure:

1. Tree T_0 consists of one vertex v; $s(T_0) = v$.
2. Tree T_{k+1} is formed by adding a new vertex v to the disjoint union of trees $\bigcup_{i=0}^{k} T_i$, and inserting additional edges $\{v, s(T_0)\}, \ldots, \{v, s(T_k)\}$ to obtain a tree; $s(T_{k+1}) = v$.

Now, series-parallel graph G_k is obtained by adding the root vertex r to tree T_k, and inserting edges connecting r with all the vertices of T_k. Observe that in some executions of Algorithms 1 and 2 the spanning tree of G_k will be the star with center r, and that the algorithm may reach a legal state in which vertex $s(T_k)$ uses color $2k - 1$, thus leading to a 2Δ-coloring of G_k. □

However, it has to be noted that graphs known as *cacti* (connected graphs such that each edge belongs to at most one cycle) constitute a highly relevant class of tripartite graphs for which Algorithms 1 and 2 always determine a nearly-optimal coloring.

Corollary 8 *A cactus G is always colored by Algorithms 1 and 2 using at most 4 colors.*

Proof. It has been shown that both algorithms stabilize, so it is enough to show that the number of colors used for cacti graphs is at most 4. Let G be a cactus and T be a spanning tree computed by the algorithm. Consider an arbitrarily chosen non-root vertex $v \in G$. Let $N_A(v) = \{u \in N(v) \mid \mathrm{A(u)=A(v)}\}$, clearly $c(v) < 2(|N_A(v)| + 1)$. Suppose that $u \in N_A(v)$, then $u \neq p(v)$ and $v \neq p(u)$.

Moreover u does not belong to path $[v, r]_T$, otherwise we would have a path $v = v_0, v_1, \ldots, v_k = u$, where $v_i = p(v_{i-1})$ for $i = 1, 2, \ldots, k$ and $f(v_i) < f(v_{i-1}) < \ldots < f(v_0)$, a contradiction with $p(v)$ being a neighbour of v such that $f(p(v)) = \min\{f(w) \mid w \in N(v)\}$. Similarly, v does not belong to path $[u, r]_T$.

Thus, we have a cycle in G involving u,v,$p(v)$, but G is a cactus, thus the edge $\{v, p(v)\}$ is involved in at most one cycle, so there is at most one vertex in $N_A(v)$ and consequently $c(v) < 4$. Remembering that $c(r)$ is equal to 0 and v was an arbitrarily chosen vertex, the proof is complete. □

It is difficult to say whether it is possible to construct a single polynomial-time self-stabilizing algorithm which optimally or near-optimally colors significantly wider graph classes than Algorithms 1 and 2. Due to the inapproximability of the 3-coloring problem in the classical model, no such algorithm may be constructed for general tripartite graphs (unless $P = NP$). However, for the relatively narrow 3-colorable classes of outerplanar graphs and series-parallel graphs the question remains open.

From the practical point of view, the algorithms presented in this paper combine the ability to take advantage of bounds on vertex degree Δ and of the 2-colorability of the system graph or its large portions. Colorings obtained for bipartite graphs, cacti and many other subclasses of planar graphs use a sufficiently small number of colors to merit application in code assignment problems in real-world conditions.

References

1. R. Battiti, A.A. Bertossi, M.A. Bonucceli, *Assigning code in wireless networks: bounds and scaling properties*, Wireless Networks, **5**, 195–209, 1999.
2. J. Beauquier, A. Kumar Datta, M. Gradinariu, F. Magniette, *Self-Stabilizing Local Mutual Exclusion and Definition Refinement*, Chicago Journal of Theoretical Computer Science, 2002.
3. G.C. Chachis, *If it walks like a duck: nanosensor threat assessment*. Proc. SPIE **5090**, 341–347, 2003.

4. E.W. Dijkstra, *Self-stabilizing systems in spite of distributed control*, Communications of the ACM **17**, 643–644, 1974.
5. S. Dolev, *Self-stabilization*, MIT Press, 2000.
6. S. Ghosh, M.H. Karaata, *A Self-Stabilizing Algorithm for Coloring Planar Graphs*, Distributed Computing **7**, 55–59, 1993.
7. Grundy, P. M.: Mathematics and games. Eureka **2** (1939), 6–8.
8. W. Goddard, S.T. Hedetniemi, D.P. Jacobs, P.K. Srimani, *Fault Tolerant Algorithms for Orderings and Colorings*, Proc. IPDPS'04, 2004.
9. M. Gradinariu, S. Tixeuil, *Self-stabilizing Vertex Coloring of Arbitrary Graphs*, Proc. OPODIS'00, 55–70, 2000.
10. S.T. Hedetniemi, D.P. Jacobs, P.K. Srimani, *Linear time self-stabilizing colorings*, Information Processing Letters **87**, 251–255, 2003.
11. L. Hu, *Distributed code assignments for CDMA Packet Radio Network*, IEEE/ACM Transactions on Networking **1**, 668–677, 1993.
12. S.T. Huang, S.S. Hung, C.H. Tzeng, *Self-stabilizing coloration in anonymous planar networks*, Information Processing Letters **95**, 307–312, 2005.
13. K.W. Hung T.S. Yum *An efficient code assignment algorithm for multihop spread spectrum packet radio networks*, Proc. GLOBECOM'90, 271–274, 1990.
14. A. Kosowski, Ł. Kuszner, *A self-stabilizing algorithm for finding a spanning tree in a polynomial number of moves*, Proc. PPAM'05 (to appear).
15. J. Li, C. Blake, D.S. De Couto, H.I. Lee, R. Morris, *Capacity of Ad Hoc wireless networks*. Proc. MobiCom'01. ACM Press, 61–69, 2001.
16. S. Sur, P.K. Srimani, *A self-stabilizing algorithm for coloring bipartite graphs*, Information Sciences **69**, 219–227, 1993.