International Journal of Distributed Sensor Networks, 1: 329–344, 2005

ISSN: 1550-1329 print/1550-1477 online
DOI: 10.1080/15501320500330745

Taylor & Francis
Taylor & Francis Group

Self-Stabilizing Global Optimization Algorithms for Large Network Graphs†

WAYNE GODDARD, STEPHEN T. HEDETNIEMI,
DAVID P. JACOBS, and PRADIP K. SRIMANI

Department of Computer Science, Clemson University, Clemson, SC

The paradigm of self-stabilization provides a mechanism to design efficient localized distributed algorithms that are proving to be essential for modern day large networks of sensors. We provide self-stabilizing algorithms (in the shared-variable ID-based model) for three graph optimization problems: a minimal total dominating set (where every node must be adjacent to a node in the set) and its generalizations, a maximal k-packing (a set of nodes where every pair of nodes are more than distance k apart), and a maximal strong matching (a collection of totally disjoint edges).

Keywords Self-stabilization; Optimization Alogrithms; *k*-packing

1. Introduction

Most of the essential fundamental services for mobile networked distributed systems (ad hoc, wireless or sensor) involve maintaining a global predicate over the entire network (defined by some invariance relation on the global state of the network) by using local knowledge at each of the participating nodes. With the advent of large scale sensor networks, where a very large number of sensor nodes with limited computing and communication capabilities are involved to achieve a larger global task, scalability in coordinating the nodes and implementing fault tolerance in the network now require fundamentally new approaches [1]. The participating sensor nodes can no longer keep track of even a small fraction of the knowledge about the global network due to limited storage. Similarly, the traditional approach to building fault tolerant distributed system is no longer economically viable. The traditional approach of *fault masking* is *pessimistic* in the sense that it assumes a worst case scenario and protects the system against such an eventuality. Validity is guaranteed in the presence of faulty processes, which necessitates restrictions on the number of faults and on the fault model. But fault masking is not free; it requires additional hardware or software, and it considerably increases the cost of the system. This additional cost is not an economic option, especially when most faults are transient in nature and a temporary unavailability of a system service is acceptable for a short period of time. We need a new paradigm of *localized* distributed algorithms, where a node takes *simple actions based on local knowledge of only its immediate neighbors and yet the system achieves a global objective* [1].

†This work has been supported by NSF grant # ANI-0218495.
Address correspondence to Pradip K Srimani, Department of Computer Science, Clemson University, Clemson, SC 29634–0974. E-mail: srimani@cs.clemson.edu

Self-stabilization is a relatively new paradigm for designing such localized distributed algorithms for networks; it is an *optimistic* way of looking at system fault tolerance and scalable coordination, because it provides a built-in safeguard against transient failures that might corrupt the data in a distributed system. The concept was introduced by Dijkstra in 1974 [2], and Lamport [3] showed its relevance to fault tolerance in distributed systems in 1983; a good survey of early self-stabilizing algorithms can be found in [4] and Herman's bibliography [5] also provides a fairly comprehensive listing of most papers in this field. The system is able to adjust when faults occur, but 100% fault tolerance is not warranted. The promise of self-stabilization, as opposed to fault masking, is to recover from failure in a reasonable amount of time and without intervention by any external agency. Since the faults are transient (eventual repair is assumed), it is no longer necessary to assume a bound on the number of failures. The participating nodes communicate only with their immediate neighbors and require minimal storage to keep the local knowledge and yet a desired global objective is achieved. Since no communication is needed beyond a node's immediate neighborhood, communication overhead scales well with increase or decrease in the network size.

Another fundamental idea of self-stabilizing algorithms is that the distributed system may be started from an arbitrary global state. After a finite amount of time the system reaches a correct global state, called a *legitimate* or *stable* state. An algorithm is self-stabilizing if

i. for any initial illegitimate state it reaches a legitimate state after a finite number of node moves, and
ii. for any legitimate state and for any move allowed by that state, the next state is a legitimate state.

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

Graph theoretic optimization problems are useful for such dynamic networks; fault tolerant distributed protocols for such problems provide the key resources for designing such wireless, sensor and ad hoc networks and they offer new insight into the fundamental role of discrete distributed algorithms in developing these real life applications [6]. For example, a minimal spanning tree must be maintained to minimize latency and bandwidth requirements of multicast/broadcast messages or to implement echo-based distributed algorithms [7]; a minimal dominating set must be maintained to optimize the number and the locations of the resource centers in a network [8]; an (r,d) configuration must be maintained in a network where various resources must be allocated but all nodes have a fixed capacity r [9]. In some sense, the concept "graph problem" is not very restrictive as it can apply to any problem where there is some static global computation in the network. Our purpose in the present paper is to further explore the intrinsic algorithmic power of this paradigm to design protocols for maintaining global predicates in a network based on only local knowledge at nodes. We are interested in algorithms to implement global synchronization where the legitimacy specification in terms of local property. Specifically, we propose new self-stabilizing algorithms for generalizations of dominating sets, generalization of matching, and maximal k-packing and provide correctness proofs and complexity analyses.

Dominating sets with specific properties have been used in several routing proposals for mobile networks [10], or for server placement (see for example [11]). Natural generalizations of a dominating set S can be obtained by placing a lower bound on the

number of neighbours that each node (or each node not in S) must have in S, or by insisting that each node outside S be sufficiently close to at least one node in S. That is, the set S must be sufficiently dense and the goal is to find a minimal such set. We provide a self-stabilizing algorithm for a family of these generalizations. This includes what is called a ***total dominating*** set: a set S of nodes such that each node in the graph is adjacent to at least one node of S[1].

We then provide a self-stabilizing algorithm for the generalization of a maximal independent set S where every two nodes in S must be sufficiently far apart. That is, the set must be locally sparse and the goal is to find a maximal such set. A special case of this algorithm occurs when one specifies that no two nodes in S can be within distance k of each other; such a set is called a k-***packing***. A self-stabilizing algorithm for finding a maximal 2-packing was given by Karaata [13]; our protocol uses a different approach and generalizes the results in [13].

A different class of graph problems involves finding a maximal ***matching***: this is a set of edges, no two of which are adjacent. A self-stabilizing algorithm for this problem was constructed by Hsu and Huang [14] (and shown to run in linear-time in [15]). A generalization is given in [16]. These matching algorithms work for anonymous networks. We consider here a variation called a ***strong*** or induced matching. This is a matching $\boldsymbol{M}$ with the added property that no two edges in $\boldsymbol{M}$ are joined by an edge. We provide a self-stabilizing algorithm to find a maximal strong matching[2]. We observe the following major characteristics of the proposed protocols.

- The protocols developed for various global predicates in a network are all localized in the sense that nodes need to know the states only of its immediate neighbors. Thus, the solutions are scalable for arbitrary number of nodes in the network and since the protocols are self-stabilizing, nodes can enter and leave the system as long as each node knows an upper-bound on the number of nodes in the network and the nodes are assigned unique IDs.
- The protocols show an unified way of utilizing ordered ID space of nodes to solve different graph optimization problems which can be used as building blocks for other related problems.
- Although the protocols have been described using the shared memory model, they can easily be implemented (by using the existing local mutual exclusion protocols at a lower layer of the networks) in a distributed system; in ad hoc or sensor networks where message exchange is essentially by broadcast, local mutual exclusion is already implemented by the very nature of transmission.
- The protocols are self-stabilizing and hence no system wide reset is necessary in presence of link and node failures and the protocols can tolerate an arbitrary number of such faults as long as the rate of failures is lower than the convergence time of the protocols.

1.1 Model, Notation, and Related Work

We restrict attention to bidirectional networks or symmetric graphs. We use the standard ***shared-variable*** model [17], in which a node sees the variables of its neighbors; this is the same model that has been used in earlier self-stabilizing protocol design. Since our objective is to explore the algorithmic aspects distributed protocols based on local

[1]A preliminary version of the total domination algorithm was presented in [12]
[2]A preliminary version was presented in [29]

knowledge, we do not consider implementation details of message communication. Our algorithms are deterministic. We measure time in the actual number of ***moves***.

For symmetry breaking we assume the nodes have ***identifiers*** (IDs). We assume there is a total ordering on the set of IDs, but like other results in the literature, we often actually only require a total ordering on the IDs in the neighborhood of each node.

We assume the algorithm is executed by a ***central daemon***, but we do not assume the daemon is fair [17]. This assumption is not restrictive and is made only for the sake of convenience; the proposed algorithms can be modified to run under a distributive daemon and read/write atomicity, either directly or using the general results of daemon transformers given in [18, 19].

Several authors have considered self-stabilizing algorithms for graph problems. For example, matchings are studied in [14, 16], maximal independent sets in [20], and domination in [21–24]. A more comprehensive list of references to past related works can be found in those cited works.

Given a node i in a graph, we denote by $N(i)$ its neighborhood, that is, the set of all nodes adjacent to i (its neighbors). The set $N[i] = N(i) \cup \{i\}$ is its closed neighborhood. Given a set D of nodes, we denote by $D_i^<$ the subset of D of those nodes having an ID smaller than i. Similarly, $D_i^>$ is the set of all nodes in D with ID greater than i. Also, note that we use i interchangeably to denote a node, and the node's ID.

2. Minimal Total Domination

2.1 Algorithm

The algorithm has a similar flavor to the one for minimal domination given in [HnJS03]. In the proposed algorithm, each node i has two variables: a pointer $p(i)$ (which may be null) and a boolean flag $x(i)$. If $p(i) = j$ then we say that *i points to j*. We will use i interchangeably to denote a node and its ID. At any given time, we will denote with D the current set of nodes i with $x(i) = true$.

Definition 2.1. For a node i, we define m(i) as its neighbor having the smallest ID.

Definition 2.2. We define a pointer expression q(i) for any node i as follows:

$$q(i) = \begin{cases} m(i) & \text{if } N(i) \cap D = \emptyset \\ j & \text{if } N(i) \cap D = \{j\} \\ null & \text{if } |N(i) \cap D| \geq 2. \end{cases}$$

Note that the value q(i) can be computed by i using only local information.

Definition 2.3. we define the boolean condition y(i) for a node i to be true if and only if some neighbor of i points to it.

The algorithm consists of one rule shown in Algorithm 1. Thus, a node i is privileged if $x(i) \neq y(i)$ or $p(i) \neq q(i)$. If it executes, then it sets $x(i) = y(i)$ and $p(i) = q(i)$.

Algorithm 1 Minimal Total Dominating Set
Variables: boolean $x(i)$, pointer $p(i)$

Rec: **if** $x(i) \neq y(i)$ **or** $p(i) \neq q(i)$
then set $x(i) = y(i)$ and $p(i) = q(i)$

2.2 Correctness and Convergence

Lemma 2.1. If Algorithm 1 stabilizes, then D is a minimal total dominating set.

Proof. First, we claim that D is a total dominating set. For suppose, that some node i is not totally dominated (that is, has no neighbor in D). Then $N(i) \cap D = \varnothing$. Since the system is stable, $p(i) = q(i) = m(i)$, and $m(i) \notin D$. But this implies $y(m(i)) = true$ and $x(m(i)) = false$, and so node $m(i)$ is privileged, a contradiction. Thus D is a total dominating set.

Next, we claim that D is minimal. For suppose there is some $j \in D$ for which $D - \{j\}$ is a total dominating set. Since $j \in D$, or $x(j) = true$, there is some node $i \in N(j)$ for which $p(i) = j$. But since $p(i) = q(i)$, node j must be the unique neighbor of i with membership in D. Thus the removal of j will leave i undominated, a contradiction.

Note. We say that node i *invites* node j if at some time t node i has no neighbor in D and then executes the rule, causing $p(i) = m(i) = j$. For a node to join D, it must either be pointed to from an initial erroneous state or be invited.

We now show our algorithm stabilizes. Observe that if D remains the same, then every node can execute at most once (to correct its pointer). So it suffices to show that D changes at most a finite number of times. We say a move is an *in-move* if it causes $x(i)$ to become true.

Lemma 2.2. If during some time-interval there is no in-move by a node bigger than node i, then during this time-interval node i can make at most two in-moves.

Proof. The first in-move made by i may have been because a neighboring node happened to initially point to i. The second in-move made by i must be by invitation. So suppose i is invited by node j. Then i is the smallest node in j's neighborhood, since $m(j) = i$, and at the time of invitation, all other nodes in j's neighborhood are out of D. By our assumption, their membership status does not change, so j remains pointing to i throughout, and i remains in D for the remainder of the time-interval.

Theorem 2.1. Algorithm 1 always stabilizes, and finds a minimal total dominating set.

Proof. It suffices to show that every node makes only a finite number of in-moves. By Lemma 2.2, node n, which has the largest ID, makes at most two in-moves. During each of the three time-intervals between such moves, using Lemma 2.2 again, node n–1 makes at most two in-moves. It is easy to show this argument can be repeated, showing that each node can make only finitely many in-moves during the intervals in which larger nodes are inactive.

2.3 Exponential Running Time

We briefly sketch how the algorithm can make an exponential number of moves. Consider the subdivision of the complete graph and add a leaf-node incident to each node of large degree. Assume all the large-degree nodes have the largest IDs. Initially all flags are clear and all pointers null.

Number the large-degree nodes $v_1,\ldots,v_s$ in increasing ID. Let w_{ij} be the node of degree two adjacent to v_i and v_j, and ℓ_i the leaf-node adjacent to v_i. Define the sequence $\mathcal{L}_i$ recursively in decreasing order. Sequence $\mathcal{L}_{s+1}$ is empty. Then to get $\mathcal{L}_i$ from $\mathcal{L}_{i+1}$ proceed as follows. Every time a node v_j (necessarily $j > i$) sets its flag, then immediately precede it with:

Fire u_{ij} (so it points to v_i) and fire v_i so it enters the set.

and immediately succeed with

Fire w_{ij} (so it points to *null*) and fire v_i so it exits the set.

Finally tack onto the end of the sequence that ℓ_i fires inviting v_i and v_i fires to end up in the set. The result is a sequence of moves exponential in s (and s is like the square-root of the number of nodes in the graph).

3. Minimal Extended Domination

We now show how to generalize the basic ideas of the previous section. We also show in the next section that the same ideas can be used to solve the opposite problem of maximal packing.

A dominating set is a set D in which, for all i, $|N[i] \cap D| \geq 1$, and a total dominating set satisfies $|N(i) \cap D| \geq 1$. We seek a common generalization.

Definition 3.1. Assume that for each node i, the set $\mathcal{N}(i)$ represents some fixed subset of its closed neighborhood $\mathcal{N}[i]$. Assume further that each node has a target integer t(i) (which is at most $|\mathcal{N}(i)|$), indicating how many elements of $\mathcal{N}(i)$ are required to dominate i. Note that in the case of total domination $\mathcal{N}(i)$ is precisely $\mathcal{N}(i)$ and t(i) is uniformly one. Given these assumptions we seek a minimal set D (called **minimal extended dominating set***) in which, for all i,*

$$|\mathcal{N}(i) \cap D| \geq t(i). \tag{1}$$

3.1 Algorithm

For the algorithm, we provide each node with a bag of pointers, denoted $P(i)$, whose cardinality is bounded by $t(i)$. (We allow $P(i)$ to contain i.) Each node also has a boolean flag $x(i)$. As before, $x(i)$ should be true if and only if some node points to i, and also as before, D will denote the set of nodes with true flags at any point in time.

At a given time, assume $|D \cap \mathcal{N}(i)| = k \leq t(i$. Then since $t(i) \leq |\mathcal{N}(i)|$, there are at least $t(i) - k$ members in $\mathcal{N}(i) - D$. Let M_i denote the (unique) set of those $t(i) - k$ nodes in $\mathcal{N}(i) - D$ having smallest IDs. Note that this set depends on the membership of D of the time.

Definition 3.2. We define a set of pointers Q(i) as follows.

$$Q(i) = \begin{cases} (D \cap \mathcal{N}(i)) \cup M_i & \text{if } |\mathcal{N}(i) \cap D| = k \leq t(i) \\ \varnothing & \text{if } |\mathcal{N}(i) \cap D| > t(i). \end{cases}$$

As before, we define the boolean condition $y(i)$ to be *true* if and only if some neighbor of i points to it. The algorithm consists of one rule shown in Algorithm 2. Thus, a node i is privileged if $x(i) \neq y(i)$ or $P(i) \neq Q(i)$. If it executes, then it sets $x(i) = y(i)$ and $P(i) = Q(i)$.

Remark 1. It is easy to see that Algorithm 2 reduces to Algorithm 1 when $\mathcal{N}(i) = N(i)$ and t(i) = 1 for all i.

Algorithm 2 Minimal Extended Dominating Set
Variables: bag $P(i)$ of pointers, flag $x(i)$

Rec: if $x(i) \neq y(i)$ **or** $P(i) \neq Q(i)$

then set $x(i) = y(i)$ and $P(i) = Q(i)$

Lemma 3.1. If Algorithm 2 stabilizes, then D is a minimal set satisfying (1).

Proof. Assume Algorithm 2 stabilizes, and suppose that for some i, $D \cap \mathcal{N}(i) < t(i)$. Then $M_i \neq \theta$, and so there is some neighbor $j \in P(i), j \notin D$. But $y(j)$ is true and $x(j)$ is false, a contradiction. We now claim D is minimal as well. For every node $j \in D$, there is some node i that points to it. Since $P(i) = Q(i)$, and since $P(i) \neq \phi$, we must have $|\mathcal{N}(i) \cap D| = k \leq t(i)$. Thus, the removal of j from D will leave $|D \cap \mathcal{N}(i) < t(i)$.

Again, we say that node i ***invites*** node j (with $j = i$ allowed) if at some time $|D \cap \mathcal{N}(i)| = k < t(i), j \in M_i$, and i executes a move. For a node to join D, it must be pointed to from an initial state or be invited.

Theorem 3.1. Algorithm 2 always stabilizes, and finds a minimal extended dominating set.

Proof. In light of Lemma 3.1 we need only show stabilization. As before, observe that if D remains the same, then every node can make at most one move (to correct its pointers). So it suffices to show that D changes at most a finite number of times. In particular, it suffices to show that if during some time interval, $x(k)$ remains unchanged for all nodes $k > i$, then during this interval node i can make at most two in-moves.

If i is never invited during this interval, then once i leaves D, it cannot rejoin. So suppose that during this interval i is invited by node j, allowing i to make an in-move. Once i enters D it must remain there if j continues pointing at it. And this is ensured, provided $|D \cap \mathcal{N}(j)| \leq t(j)$. Suppose at the time of invitation $|D \cap \mathcal{N}(j)| = k$. Nodes having IDs larger than i do not move during this interval, but the smaller nodes can. At the time of invitation, i is among the $t(j) - k$ smallest nodes in $\mathcal{N}(j) - D$. Even if all nodes smaller than i were to enter D, we would still have $|D \cap \mathcal{N}(j) \leq t(j)$. It follows that j will remain pointing to i throughout and i will remain in D. Hence, $x(i)$ can make at most two in-moves during this interval.

3.2 Applications

Signed Domination. With this extension, one can, for example, find a minimal weak dominating set: here every node must be dominated by a node with degree at most its own. (For more details, see [25].) We also observe that Algorithm 2 gives one a self-stabilizing algorithm for finding a minimal ***signed dominating functions*** (For references, see [25]). An assignment $f : V \rightarrow \{-\mathbf{1}, 1\}$ is a signed dominating function if, for every node i, the sum of the values in $N[i]$ is positive. Equivalently, f is signed dominating if a strict majority of the values in every closed neighborhood are positive. The function f is minimal if the function f' obtained by reducing the value at any positive node, is never signed dominating. It is easy to see that minimal signed dominating functions correspond to certain minimal extended dominating sets. In particular, $f \rightarrow \{-1, 1\}$ is a minimal signed dominating function if and only if the set $D\{i | f(i) = 1\}$ is a minimal extended dominating set in which for all i,

$$\mathcal{N}(i) = N(i) \qquad \text{and} \qquad t(i) = \left\lceil \frac{|N[i]|}{2} \right\rceil + 1.$$

Packings. The generalized domination algorithm above solves the problem of a maximal packing. Another way of defining a packing is to say that it is a set such that in any closed

neighborhood there is at most one node of the set. The set that is constructed is the set of nodes *not* in the packing. The requirement is that for every closed neighborhood of cardinality N, at least N–1 nodes must be outside the packing. Further, it is easy to see that the packing is maximal iff the complement is minimal with respect to that property.

One can generalize packing to allow denser sets. We define a $\{k\}$-packing as a set S of nodes such that in every closed neighborhood there are at most k members of S. So a normal packing is a $\{1\}$-packing. Such a packing is maximal if no node can be added to it. By the same argument as above, the extended domination algorithm solves this problem.

Weighted Extended Domination. The techniques used above can be extended further. Consider the problem of ***weighted extended domination***. Here each node i has an allowable range of values $\{0,1,\ldots, b(i)\}$. Each node also has a target $t(i)$ for the sum of the values of the elements in $\mathcal{N}(i)$. We want a minimal assignment of values that satisfy the constraints.

One way to handle this is simply to think of a node i as having $b(i)$ flags that operate independently and have unique IDs. However, for efficiency, instead of a flag $x(i)$ a node could have a value $X(i)$ restricted to the allowed range. The bag of pointers is replaced by an array of counters, one for each neighbor. The weight that i places in that register is how much weight it needs from that neighbor. The rule for $X(i)$ for consistency is that it is the minimum of the weight-needs of its neighbors. We omit the details which are similar.

4. Maximal k-Packings

We now consider the problem of finding a set such that every pair of nodes are far apart. Recall that a k-packing is a set such that every pair of nodes are strictly more than k apart. We present next an algorithm for finding a maximal k-packing. This also provides a minimal set such that every node is within distance k of the set (called distance k-domination in the literature). Distributive algorithms for this problem were given by Kutten and Peleg [26].

4.1 Algorithm

In the algorithm, each node i has a vector R_i (for ruler) of length k (indexed $0,\ldots, k$–1) whose entries are IDs or null. First, We define the set D of ***despots***.

Definition 4.1. D is the set of all nodes i such that R_i contains only the value i (written $R_i \equiv i$). The intended meaning of the entry $R_i[d]$ is that it gives the smallest ID of a despot within distance d of the node.

The algorithm consists of three rules as shown in Algorithm 3. The idea is that a node becomes a despot if (as far as the node can tell) there is no other despot within distance k, and retire if (as far as the node can tell) there is a despot with a smaller ID within distance k. Otherwise the node updates its ruler-vector to have the desired meaning.

Algorithm 3 Maximal Packing
Variables: an array R_i of k pointers
Add: **if** $R_j[k-1] = null$ for all $j \in N(i)$
then fill R with i.
Ret: **if** $R_i[0] = i$ and $R_j[k-1] < i$ for some $j \in N(i)$
then set $R_i[0] = null$ and call Rule Upd below.

Upd: Calculate array S as follows: Set $S[0] = null$. For $d \geq 1$, set $S[d] = \min\{R_j[d-1]: j \in N(i)\}$. If $R_i \neq i$ and $R_i \neq S$, then set $R_i = S$.

4.2 Correctness and Convergence

Lemma 4.1. If Algorithm 3 stabilizes, then for each node i the value $R_i[d]$ does indeed give the smallest ID of a despot within distance d from i

Proof. Proof by induction on d. True for $d = 0$: if i is a despot then $R_i[0] = i$ and otherwise $R_i[0] = null$ (any other possibility and the node would be privileged for Rule Upd). So assume true for indices less than d.

Assume i is not a despot. By Rule Upd, the value of $R_i[d]$ is equal to the minimum of $R_j[d-1]$ for some neighbor j. By the induction hypothesis, $R_j[d-1]$ is the smallest despot within distance $d-1$ of j. In particular, $R_i[d]$ is a despot within distance d. But suppose z is the smallest despot within distance d. Then $z < R_i[d]$. But then if u is the neighbor of i on a shortest $i - z$ path, it holds that $R_u[d-1] \leq z$, a contradiction.

A similar argument holds if i is a despot, except that then if $R_i[d]$ (which equals i) is not the smallest despot within distance d, them $R_j[d-1] < i$ for some neighbor j, and so i is privileged for Rule Ret, a contradiction.

It follows that:

Lemma 4.2. If Algorithm 3 stabilizes, then D is a maximal k-packing

Proof. Suppose two despots $i < j$ are within distance k of each other. Let u be the first node on a shortest path from j to i (possibly $u = i$). Then by the above lemma, $R_u[k-1] \leq i$; but then j is privileged for Rule Ret, a contradiction. So D is a k-packing.

On the other hand, suppose that i can be added to D and it remain a k-packing. Then i is distance more than k from every despot. So every neighbor of i is distance at least k from every despot, and by the above lemma has the all-null vector. But this means that i is privileged for Rule Add, a contradiction.

Lemma 4.3. If the set D does not change over some time-interval, then there can be at most a finite number of moves

Proof. By assumption, during this time-interval neither Rule Add nor Rule Ret is ever executed. Thus, while R_i might be corrupt due to erroneous initialization, after one pseudoround $R_i[0]$ has stabilized for all nodes.

After another pseudoround, $R_i[1]$ has stabilized for all nodes. For, if node i executes Rule Upd, then $R_i[1]$ is the minimum of $R_j[0]$ over all its neighbors, and these values have stabilized. Similarly, after another pseudoround, $R_i[2]$ has stabilized. After k pseudorounds, there can be no further move.

Now we define an ***i-reversal*** as any move in which the value $R_v[d]$ for some node and $d \geq 0$ changes from being smaller than i to being null or at least v (or vice versa).

Lemma 4.4. If the set $D_i^{<}$ *does not change, then there are a finite number of i-reversals*

Proof. After one pseudoround, there obviously cannot be another reversal of $R_v[0]$ for any v. Similarly, after another pseudoround, there cannot be another reversal of $R_v[1]$. For, if $R_w[0] < i$ for some $w \in N(i)$, then $R_v[1]$ is fixed to the smallest value of $R_v[0]$ (as these values never reverse). On the other hand, if $R_w[0]$ is null or at least i for every neighbor, then $R_w[1]$ can never become a value less than i.

Similarly, after k pseudorounds, there can be no further i-reversal.

Theorem 4.1. Algorithm 3 always stabilizes, and finds a maximal k-packing.

Proof. In light of Lemma 4.2 we need only show stabilization. We argue that a node can enter D at most a finite number of times. The proof is by induction on i.

We first observe that between consecutive entrances of a node i, there must be an i-reversal. For, when i executes Rule Add, all of its neighbors must have all-null arrays. In order for it to execute Rule Ret, some neighbor of i must have $R_j[k-1]$ smaller than i. That is, there must have been an i-reversal.

For the smallest node $i = 1$, there can be no 1-reversal. Hence it can enter at most once. For a general node i, between changes in $D_i^{<}$ there is a bounded number of i-reversals, and hence a bounded number of entrances by i. This shows that D can change at most a finite number of times.

Putting this together with Lemma 4.3, it follows that the algorithm always terminates.

5. Strong Matching

In our proposed algorithm, each node i maintains only a pointer $P(i)$: the value of $P(i)$ is either a neighbor of i, or one of two special values: **Open** or **Unav**. The idea is that in general the pointers form the matching. If a node is not currently matched, it will declare itself ***open*** (set its pointer to Open) if it can be matched; and declare itself ***unavailable*** (set its pointer to Unav) if a neighbor is already matched and that match has higher precedence (see below). If a node's pointer does not point to Unav, then we say it is ***available***.

The key is to use the total ordering of the nodes to provide an ordering (sometimes called the lexicographic ordering) on the set of edges (or more precisely on the set of pairs of nodes).

Definition 5.1

If e and f are pairs of nodes, then $e \preceq f$ iff either $\min(e) < \min(f)$ *or* $\min(e) = \min(f)$ *and* $\max(e) \leq \max(f)$. *That is, if e and f do not have a node in common, then the one with the smallest node is the smaller; if they share a node, then they are ranked by the other node. If $e \preceq f$ and $e \neq f$ then we will write $e \prec f$.*

For a node i, we define $A(i)$ to be the smallest matched edge incident with one of its neighbors (as shown by their pointers).

Definition 5.2

For any node i, we define a special edge as

$$A(i) = \min\{\{j, P(j)\} : j \in N(i) \wedge P(j) \notin \{\text{Open, Unav}\}\},$$

if it exists. (The edges are compared using the ordering $\preceq$ *given above.)*

We further define $B(i)$ as the minimum available neighbor of node i.

Definition 5.3

For any node i, we define a special neighbor as

$$B(i) = \min\{j : j \in N(i) \wedge P(j) \neq \text{Unav}\}$$

if it exists.

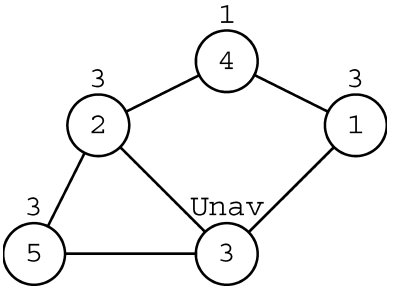


FIGURE 1 An example graph for illustration.

For example, consider the graph shown in Fig. 1 (where the values inside the nodes give the nodes' IDs and the values outside the value of P). For the node 2, $A(2) = \{1,4\}$ and $B(2) = 4$.

Definition 5.4

We define the value $Q(i)$ for a node i as follows:

$$Q(i) = \begin{cases} \text{Unav} & \text{iff } A(i) \text{ exists and either } B(i) \text{ does not exist or } A(i) < \{i, B(i)\}, \\ B(i) & \text{iff } B(i) \text{ exists and either } A(i) \text{ does not exist or } \{i, B(i) \le A(i) \\ \text{Open} & \text{iff neither } A(i) \text{ nor } B(i) \text{ exists.} \end{cases}$$

*We say that a node i is **consistent** iff $P(i) = Q(i)$. That is, for a node to be consistent, it must be unavailable if there is a matched edge incident with a neighbor that is smaller than any that it could be in; failing which it must point to the smallest available neighbor if there is one; failing which it must be open.*

Note that the value $Q(i)$ can be computed by the node i (i.e., it uses only local information).

The self-stabilizing algorithm for maximal strong matching consists of one rule as shown in Algorithm 4. Thus, a node is ***privileged*** if $P(i) \neq Q(i)$. If a node executes, then it sets $P(i) = Q(i)$.

Algorithm 4 Maximal Strong Matching
Variables: a pointer $P(i)$

Correct: if $P(i) \neq Q(i)$
then set $P(i) = Q(i)$

5.1 Correctness at Convergence

Lemma 5.1. If Algorithm 4 stabilizes, then for any node i, if $P(i) = j$ then $P(j) = i$.

Proof. We claim that if $P(j) = \text{Unav}$ or $P(j) < i$, then i is privileged. For, if $P(j) = \text{Unav}$ then $B(i) \neq j$ so that $P(i) \neq Q(i)$; while if $P(j) < i$ then since $A(i) \preceq \{j, P(j)\}$ it follows that $Q(i) = \text{Unav}$ or $Q(i) \le P(j)$, so that $P(i) \neq Q(i)$.

Further, we claim that if $P(j) = \text{Open}$ or $P(j) > i$, then j is privileged. For, since $B(j) \le i$, it follows that $Q(i) = \text{Unav}$ or $Q(j) \le i$.

We define a special set of edges as follows.

Definition 5.5.

$$\mathfrak{M} = \{\{i, j\} : i, j \in V \wedge P(i) = j \wedge P(j) = i\}.$$

Lemma 5.2. If Algorithm 4 stabilizes, then the set $\mathfrak{M}$ is a maximal strong matching.

Proof. By the above lemma, the set $\mathfrak{M}$ is a matching.

Suppose adjacent nodes i and j are both in the matching, but the edge between them is not part of the matching. That is, edges $e = \{i, P(i)\}$ and $f = \{j, P(j)\}$ are disjoint. Without loss of generality, $e \prec f$. Then j is privileged, a contradiction. So $\mathfrak{M}$ is a strong matching.

Finally, suppose an edge $e = \{i, j\}$ can be added to $\mathfrak{M}$ and it still be a strong matching. Then neither i nor j has a neighbor in $\mathfrak{M}$. So neither $A(i)$ nor $A(j)$ exists. This means that since i and j are consistent, neither is unavailable; but then $B(i)$ and $B(j)$ exist, and hence both nodes are inconsistent, a contradiction.

5.2 Termination

In this section we show that Algorithm 4 terminates in a finite number of moves when it starts from an arbitrary initial state.

Definition 5.6. Let X be a totally ordered set. Consider a sequence $\mathfrak{S}$ of subsets S(t) of X, t = 1, 2,..., such that each subset is obtained from the previous one by either the addition or deletion of one element. We say such a sequence is ***downward*** *if for all elements i, if i is added at time t and then deleted at time t′, then some element smaller than i is added between time t and t′. That is, if $i \in S(\tau)$ for $t \le \tau < t'$ but $i \notin S(t-1), S(t')$, then there exists $j < i$ and τ with $t < \tau < t' - 1$ such that $S(\tau) = S(\tau - 1) \cup \{j\}$.*

Lemma 5.3. If X is finite, then a downward sequence $\mathfrak{S}$ on X is finite.

Proof. We prove by induction on $|X|$. Consider the minimum element of X; call it 0. It might be in or out of $S(1)$. But once added, it cannot be deleted. So define $\mathfrak{S}'$ as the subsequence up to 0's addition, if it exists, and $\mathfrak{S}($ as the subsequence after 0's addition. (If 0 is never added, then set $\mathfrak{S}' = \emptyset$ and $\mathfrak{S}(= \mathfrak{S}$.) Then ignore the transition where 0 is deleted, if this occurs, and restrict both subsequences to the set $X - \{0\}$. The result is two downward sequences on the set $X - \{0\}$. Hence, if $M(m)$ denotes the maximum length of a downward sequence for a set of cardinality m, it follows that $M(m) \le 2 + 2M(m - 1)$.

Definition 5.7. For any node k, we define the set of edges

$$S_k = \{\{j, P(j)y : j \ge k \wedge P(j) < k\}.$$

Note that S_k exists for a node k only when $P(k) \notin \{\text{Open, Unav}\}$

Lemma 5.4. Let k be a node, and consider a period when no node less than node k executes.

(i) *Then S_k can change only a finite number of times during that period.*
(ii) *If node k is at some stage consistent and available, and later declares itself unavailable, then in-between there was an addition to Sk.*

(iii) *If k is available throughout the period, and v is some neighbor of k that is at some stage consistent and available, and later declares itself unavailable, then in-between there was an addition to S_k*

Proof.

(i) Suppose at some stage $P(i)$ becomes j, with $i \geq k > j$. Since the nodes less than k are stable, it follows that $B(i) = j$ throughout. So for i to change again, it must happen that $A(i)$ changes to a value smaller than $\{i, j\}$.

(ii) That is, if at some stage $\{i, j\}$ is added to S_k, then before it is deleted some smaller edge must be added to S_k. (Note that if at the start $P(i) = j'$ with $j' < k$, then we assume the change in S_k occurs in two steps: add $\{i, j\}$ and then delete $\{i, j\}$.) Thus, the sequence of S_k is a downward sequence. By Lemma 5.3, S_k can change only a finite number of times.

(iii) When node k declares itself unavailable, a better edge must have been created in its neighborhood since node k was last consistent. Hence there was an addition to S_k.

(iv) At the moment v declares itself unavailable, it must see an edge smaller than $\{v, k\}$, which did not exist when it was consistent. Hence there must have been an addition to S_k.

Lemma 5.5. Let k be a node, and consider a period when no node less than k executes. Then node k moves a finite number of times.

Proof. There cannot be an infinite cycle of node k doing just the following types of moves: Unav to **Open**; Open to a value and/or decrease. So consider each time that k declares itself unavailable, increases value or changes from a value to being open. The latter two can only occur if some neighbor declares itself not available. Hence each such move involves either k or one of its neighbors declaring itself unavailable. By Lemma 5.4, this can occur only a finite number of times. Hence k can be privileged only a finite number of times.

Theorem 5.1. Starting from an arbitrary initial state, the algorithm terminates in finite time.

Proof. We use induction and Lemma 5.5; it follows that nodes $\{1,\ldots,k\}$ can be privileged (or make a move) only a finite number of times. Hence the algorithm terminates.

5.3 Complexity Analysis

We note that in our application, the downward sequences have the additional property that if i is deleted because of j's addition, then i cannot be re-added until j is deleted. It can readily be shown that such sequences have length $O(|X|^2)$. However, this provides no improvement in the bound on the running time of the overall algorithm—which is $O(n^n)$ since no state can repeat.

Algorithm 4 does indeed have exponential running time. Consider the following example. Take any graph G and add three new nodes x_3, x_2, x_1 such that x_3 is adjacent to all nodes, while x_1 and x_2 are adjacent only to each other and to x_3, and such that these three nodes have the smallest IDs and $x_3 > x_2 > x_1$. Call the resulting graph G'.

Assume all nodes start as Unav. Then assume the demon proceeds as follows:

(1) fire G until it stabilizes;
(2) fire x_1 (so goes to Open) and then x_3 (so points to x_1);
(3) fire all nodes in G (so go to Unav);
(4) fire x_2 (so points to x_1) and then x_3 (so goes to Unav);
(5) fire G until it stabilizes.

If $M(G)$ denotes the maximum number of steps on graph G assuming all nodes start as unavailable, then it follows that $M(G') \geq 2M(G) + 4$. By repeating this construction, it follows that running time can be at least $2^{n/3}$.

6. Conclusions

In this paper, we have proposed several self-stabilizing distributed algorithms for total domination, maximal k-packing and strong matching in large networks. We have assumed an ID based network to prove the correctness and the convergence of the algorithms. The underlying concepts are shown to be general and they are useful in symmetry breaking which is ever so important in designing distributed algorithms to achieve global objectives based on local knowledge. We did not address the issues of implementation of message exchanges in the network; this will be needed to be done before the protocols can be useful in real life networks; one possible implementation mechanism can be found in [27]. Also, we have considered only worst case performance analysis of the protocols; experimental investigations of average run-time behavior will be interesting and useful. We expect the inherent algorithmic power of the self-stabilization paradigm to achieve global objectives based on localized computations will be useful in designing algorithms for similar applications in ad hoc and sensor networks.

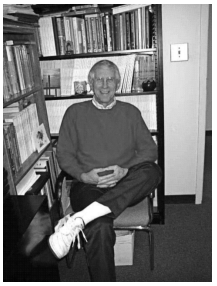
It is to be noted that in designing self-stabilizing algorithm, we need to define some kind of an invariant at each node that depends on the states of the neighboring nodes (since only the states of the immediate neighbor nodes are available to any node). Most of the self-stabilizing protocols for various graph theoretic problems follow this approach. On the other hand, there are problems for which designing a distributed algorithm becomes easier and more intuitive if distance-two knowledge was available at each node (this does not necessarily mean that the distributed protocol cannot be designed without distance-two knowledge). Authors in [28] have proposed a very elegant self-stabilizing protocol to collect distance-two knowledge at each node (invariant at each node still depends on strictly local knowledge). The underlying principle of the approach is to utilize some embedded lock mechanism to make sure the correct values of variables in distance-2 neighborhood are up to date before each node makes a move (which explicitly depends only on distance-one information). It has also been shown in [28] that any self-stabilizing algorithm that is designed using distance-2 knowledge can be simulated by their protocol to run on distance-1 knowledge at nodes with a slowdown factor of $O(n^2)$. It'd be interesting to if it is possible to design polynomial time self-stabilizing algorithms for the problems studied in this paper using the new approach.

References

1. D. Estrin, R. Govindan, J. S. Heidemann, and S. Kumar, "Next century challenges: Scalable coordination in sensor networks," in *Mobile Computing and Networking,* 263–270, 1999.
2. E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Communications of the ACM*, **17**, 643–644, 1974.
3. L. Lamport, "Solved problems, unsolved problems, and non-problems in concurrency," in *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing*, 1–11, 1984.
4. M. Schneider, "Self-stabilization," *ACM Computing Surveys*, **25**, 1, 45–67, March, 1993.
5. Herman. T., "A comprehensive bibliograph on self-stabilization, a working paper," *Chicago J. Theoretical Comput. Sci*. http://www.cs.uiowa.edu/ftp/selfstab/bibliography.
6. C. Boulinier, F. Petit, and V. Villain, "When graph theory helps self-stabilization," in *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing (PODC 2004, St. John's)*, 150–159, 2004.

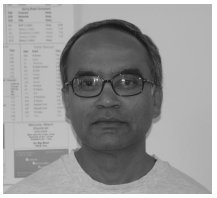
7. H. Attiya, and J. Welch, *Distributed computing: fundamentals, simulations, and advanced topics.* Mc Graw Hill, 1998.
8. T. W. Haynes, S. T. Hedetniemi, and P. J. Slater, *Fundamentals of domination in graphs.* New York: Marcel Dekker, 1998.
9. S. Fujita, T. Kameda, and M. Yamashita, "A resource assignment problem on graphs," *Proceedings of the 6th International Symposium on Algorithms and Computation*, (Cairns, Australia) 418–427, December, 1995.
10. F. Dai, and J. Wu, "An extended localized algorithm for connected dominating set formation in ad hoc wireless networks," *IEEE Trans. Parallel Distrib. Syst.*, **15**, 10, 908–920, 2004.
11. D. Peleg, "Distributed data structures: A complexity oriented view," in *Proceedings of the Fourth International Workshop on Distributed Algorithms*, 71–89, 1990.
12. W. Goddard, S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani, "A self-stabilizing distributed algorithm for minimal total domination in an arbitrary system graph," *Proceedings of the 8thIPDPS Workshop on Formal Methods for Parallel Programming*, (Nice, France), April, 2003.
13. M. H. Karaata, "Self-stabilizing strong fairness under weak fairness," *IEEE Transactions on Parallel and Distributed Systems*, **12**, 4, 337–345, 2001.
14. S. C. Hsu and S. T. Huang, "Analyzing self-stabilization with finite-state machine model," in *Proceedings of the 12th International Conference on Distributed Computing Systems*, 624–631, 1992.
15. S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani, "Maximal matching stabilizes in time o(m)," *Information Processing Letters*, **80**, 5, 221–223, 2001.
16. W. Goddard, S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani, "The b-matching paper," Preprint.
17. S. Dolev, *Self-Stabilization.* MIT Press, 2000.
18. J. Beauquier, M. Gradinariu, and C. Johnen, "Cross-over composition — enforcement of fairness under unfair adversary," in *WSS01 Proceedings of the Fifth International Workshop on Self-Stabilizing Systems, Springer LNCS, 2194*, 19–34, 2001.
19. M. Nesterenko, and A. Arora, "Stabilization-preserving atomicity refinement," *Journal of Parallel and Distributed Computing*, **62**, 5, 766–791, 2002.
20. S. K. Shukla, D. J. Rosenkrantz, and S. S. Ravi, "Observations on self-stabilizing graph algorithms for anonymous networks," in *Proceedings of the Second Workshop on Self-Stabilizing Systems*, 7.1–7.15, 1995.
21. J. R. S. Blair and F. Manne, "Efficient self-stabilizing algorithms for tree networks," in *Proceedings of ICDCS-2003,* Island, 2003.
22. S. T. Hedetniemi, D. Pokrass Jacobs, and P. K. Srimani, "Maximal matching stabilizes in time O(*m*)." *Information Processing Letters,* **80**, 221–223, 2001.
23. W. Goddard, S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani, "Fault tolerant algorithms for orderings and colorings," in *18th International Parallel and Distributed Processing Symposium (IPDPS 2004),* 2004.
24. S. M. Hedetniemi, S. T. Hedetniemi, D. P., Jacobs, P. K., Srimani, "Self-stabilizing algorithms for minimal dominating sets and independent sets," C*omputers & Mathematics with Applications* **46**, 5–6, 2003.
25. T. W. Haynes, S. T. Hedetniem, P. J. Slater, "Fundamentals of domination in graphs," *Monographs and Textbooks in Pure and Applied Mathematics,* Marcel Dekker: New York, 1998.
26. S. Kutter, and D. Peleg, "Fast distributed construction of small *k*-dominations sets and applications," *Journal of Algorithms,* **28**, 40–66, 1998.
27. S. K. S. Gupta, P. K. Srimani, "Self-stabilizing multicast protocols for ad hoc networks," *Journal of Parallel and Distributed Computing*, **63**, 1, 87–96, 2003.
28. M. Gairing, Goddard, W. Hedetniemi, S. T. and P. Kristiansen, "Distance-two information in self-stabilizing algorithms," To appear in *Parallel Processing Letters,* 2005.
29. W. Goddard, T. St. Hedetniemi, D. P. Jacobs, and P. K. Srimani, "Self-stabilizing distributed algorithm for strong matching in a system graph," *Springer-Verlag Lecture Notes in Computer Science 2913*, December, 2003.

Wayne Goddard is currently an Associate Professor in the Department of Computer Science having previously taught at the Universities of KwaZulu-Natal and University of Pennyslvania. His research interests are graph theory, algorithms, and networks. He received Ph.D.s from both the University of KwaZulu-Natal and the Massachusetts Institute of Technology. Goddard has published over 100 journal and conference papers in many areas of graph theory as well as in graph algorithms, self-stabilizing algorithms, game-playing, and ad hoc networks, and is co-author of a textbook on Research Methodology. He is currently a managing editor for the journal Utilitas Mathematica.



Stephen T Hedetniemi, Professor in the Dept of Computer Science, is a researcher in graph theory and algorithms. He has published over 180 journal and conference papers in many areas of graph theory, especially in domination and colorings of graphs. He is co-author of the book Fundamentals of Domination in Graphs and co-editor of a companion book. He is founder and co-organizer of the Clemson mini-Conference on Discrete Mathematics, now in its 20th year. He served for 17 years on the ABET Computing Accreditation Commission, and has advised or jointly advised 11 Ph.D.s. He was educated at the University of Michigan and had taught at the Universities of Oregon, Virginia, Victoria, and Iowa.

David P Jacobs is currently a professor of computer science at Clemson University. He was born in Chicago, and received his Ph.D. from this University of Missouri in 1976. He works mostly in the design and analysis of algorithms, but has also published papers on decidability, computer algebra, and ring theory. In work funded by the National Science Foundation, around 1988 he designed the computer algebra program Albert which has since led to dozens of discoveries in non-associative algebra. His current research focus is on self-stabilizing algorithms. David enjoys playing blues harmonica.



Pradip K Srimani is currently a professor and chair of computer science at Clemson University, South Carolina, USA. His research interests include parallel and distributed computing, mobile computing, and graph theory applications. His research has been supported by the National Science Foundation and others. Srimani received his BTech, MTech and Ph.D. from the University of Calcutta, India. He has published over 200 papers in journals, conference proceedings, and books. He co-edited two books for the Computer Society Press. A Fellow of the IEEE and a member of the ACM, he has served on editorial boards and as special issue guest editor for a number of journals.