Parallel Processing Letters
© World Scientific Publishing Company

SELF-STABILIZING GRAPH PROTOCOLS

WAYNE GODDARD
STEPHEN T HEDEDTNIEMI
DAVID P. JACOBS
PRADIP K. SRIMANI
ZHENYU XU
School of Computing, Clemson University
Clemson SC 29634, USA

Received May 2006
Revised October 2007
Communicated by J. Beauquier

ABSTRACT

We provide self-stabilizing algorithms to obtain and maintain a maximal matching, maximal independent set or minimal dominating set in a given system graph. They converge in linear rounds under a distributed or synchronous daemon. They can be implemented in an ad hoc network by piggy-backing on the beacon messages that nodes already use.

Keywords: Self-stabilizing protocol, distributed daemon, fault tolerance, graph algorithms

1. Introduction

Most services for networked distributed systems (mobile or wired) involve maintaining a global predicate over the entire network by using local knowledge at each node. 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 [1,2]; a minimal dominating set must be maintained to optimize the number and location of resource centers in a network [3]; or an (r, d) configuration must be maintained in a network where various resources must be allocated but all nodes have a fixed capacity r [4].

The traditional approach to building fault-tolerant, distributed systems uses *fault masking*. It is *pessimistic* in that it defines a worst case and protects 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 requires additional hardware or software, which might not be viable, especially when most faults are transient and the temporary unavailability of a service is acceptable. The paradigm of self-stabilization is *optimistic*, because it provides a safeguard against unlimited transient failures that might corrupt the data in a distributed system. Although the concept was introduced by Dijkstra in

1974 [5], serious work only began in the late nineteen-eighties. See Dolev's book [6] for an overview.

A self-stabilizing program consists of a collection of rules of the form "*if* condition *then* change state". A node is called *privileged* if the predicate of one of the rules is true. It *moves* by changing state. A fundamental idea of self-stabilizing algorithms is that the distributed system may be started from an arbitrary global state. After finite 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.

This paper considers two daemons. The adversarial *distributed daemon* chooses a subset of the privileged nodes to move at each time-step. We say that the daemon *taps* the nodes. A special case of this is the *synchronous daemon* where every privileged node moves at each time-step.

The running time of an algorithm under the distributed daemon can be measured in moves, time-steps or rounds. A *round* is a minimal sequence of time-steps where every node enabled at the start is either tapped or sees its move disabled by the move of a neighbor; if the daemon is *fair* a round is guaranteed to finish. For the synchronous daemon, the measures time-steps and rounds coincide. In general, the number of moves is an upper bound on the number of time-steps.

We provide self-stabilizing algorithms to maintain a maximal matching, a maximal independent set or a minimal dominating set in a given system graph. While the central-daemon algorithms for these problems (e.g. [7,8]) may be converted into a synchronous daemon protocol using the techniques of [9,10], the resulting protocols are not as fast. Synchronous algorithms have been considered in [9,11,12,8] inter alia.

We assume each node is assigned a unique numeric ID—synchronous or distributed-daemon algorithms must have some form of symmetry breaking. It is to be noted, however, that each algorithm actually only requires that IDs are "locally" distinct (within radius 1 suffices).

Preliminary versions of parts of this paper appeared as [13,14].

2. Ad hoc Implementation

These algorithms can be implemented on an ad hoc network by using the *beacon* ("keep alive") messages normally used by a node to inform its neighbors of its continued presence.

This assumes the following about the system. A link-layer protocol at each node i maintains the identities of its neighbors in some list $neigh(i)$. This data-link protocol resolves contention for the shared medium by supporting logical links between neighbors, and ensures that a message sent over a correct (or functioning) logical link is correctly received at the other end. The logical links between two neighboring nodes are assumed to be bounded, FIFO and bidirectional.

Each node periodically (at intervals of t_b) broadcasts a *beacon* message. When node i receives the beacon signal from a node j which is not in its neighbors list $neigh(i)$, it adds j to its neighbors list, thus establishing link (i, j). For each

link (i, j), node i maintains a timer t_{ij}. If node i does not receive a beacon signal from j in time t_b, it assumes that link (i, j) is no longer available and removes j from its neighbor set. Upon receiving a beacon signal from neighbor j, node i resets its appropriate timer. When a node j sends a beacon message, it includes the state of the node j as used in the algorithm. A beacon message provides information about its neighbor nodes synchronously (at specific time-intervals), and a node takes action only after receiving beacon messages (along with algorithm-related information) from all neighboring nodes.

The algorithms are fault tolerant (reliable) in that they can respond to occasional link failures and/or link creations (due to mobility of the hosts) and re-adjust the set. We assume that no node leaves the system and no new node joins the system (at least during the algorithm); we also assume that transient link failures are handled by the link-layer protocol by using time-outs, retransmissions, and per-hop acknowledgments.

As the stabilization time must be small with respect to the periodicity of change, the quickness of self-stabilization is important.

3. Maximal Independent Set

Given an undirected graph $G = (V, E)$, a set S of nodes is *independent* if no two members in S are adjacent. Ikeda et al. [15] presented a simple self-stabilizing protocol for finding a maximal independent set under a distributed daemon. We show here that the algorithm also stabilizes in linear rounds and hence under a synchronous daemon one obtains linear-time performance.

Algorithm MIS is shown in Figure 1. Each node maintains a flag x such that $x(i) = 1$ implies node i is in S and $x(i) = 0$ implies that it is not. The notation $N(i)$ means the neighbors of node i.

I1: if $(x(i) = 0) \wedge (\nexists j \in N(i) : x(j) = 1)$
    then $x(i) := 1$ [enter set]
I2: if $(x(i) = 1) \wedge (\exists j \in N(i) : j > i \wedge x(j) = 1)$
    then $x(i) := 0$ [leave set]

Fig. 1. Algorithm MIS: Maximal Independent Set [15]

Theorem 1 *[15] If Algorithm MIS stabilizes, then the set $S = \{ i : x(i) = 1 \}$ is a maximal independent set. Furthermore, starting from an arbitrary state, Algorithm MIS stabilizes in at most $O(n^2)$ moves, where n is the number of nodes.*

Ikeda et al. [15] constructed an example where Algorithm MIS takes $\Theta(n^2)$ time-steps. We determine the complexity in terms of rounds:

Theorem 2 *Starting from an arbitrary state, Algorithm MIS stabilizes in at most n rounds, where n is the number of nodes.*

Proof. We claim that in every round R there is a node v_R which moves and never moves again.

If some node executes rule **I1** in round R, then let v_R be such a node with maximum ID. Since v_R executed rule **I1**, before this time-step all of its neighbors were

out of the set. By the choice of v_R, any neighbor of v_R that enters simultaneously has smaller ID. After this, no new neighbor can enter, since v_R is in the set, and v_R will never leave, since it has larger ID than any neighbor in the set. Hence v_R will never move again.

If no node executes rule **I1** in round R, then let v_R be the node with maximum ID that executes rule **I2** during the round. Since it left the set, v_R has a larger-ID neighbor in S; let w_R be such a neighbor with the maximum ID. By the choice of v_R, w_R stays in S for the rest of the round. Since w_R did not execute rule **I1** during this round, it was in S at the start of the round; it follows that all neighbors of w_R in S are privileged at the start of the round, and hence execute rule **I2** by the end of the round. Thus the node w_R is isolated by the end of R and will not move again. Hence v_R will never move again.

It follows that the number of rounds is at most the number of nodes. □

We note that the protocol can stabilize with every possible maximal independent set. Indeed, with the distributed daemon one can obtain such a set starting from all nodes out.

Also, one can construct a graph where the synchronous version takes exactly n time-steps as follows. Take the path with n nodes, arrange the IDs in increasing order, and start with all nodes out of the set. In odd rounds nodes enter; in even rounds nodes leave. Further, note that one can modify the example so that one mistake triggers the $\Theta(n^2)$ moves. Add a new node x which is adjacent to the entire path, and a new node y adjacent only to x with a higher ID. If x and y start in the set, and the other nodes out, then the first move is for x to leave the set, after which the behavior is as before.

4. Minimal Dominating Set

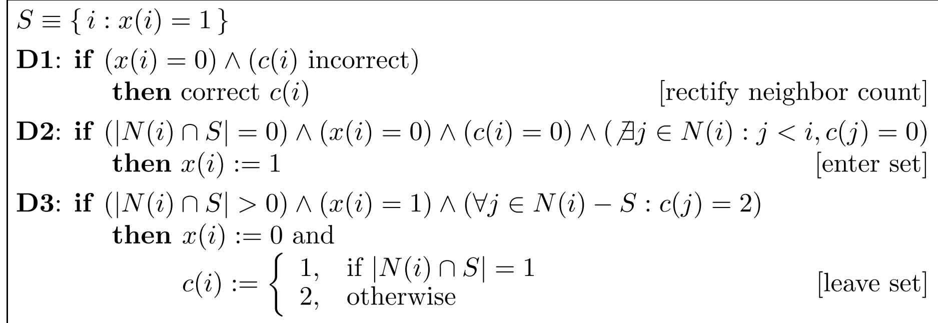
Given a node i we use the notation $N[i] = \{i\} \cup N(i)$; that is $N[i]$ is node i and its neighbors. Given an undirected graph $G = (V, E)$, a *dominating set* is a subset S of nodes such that $\forall i \in V : N[i] \cap S \neq \emptyset$; and a dominating set S is *minimal* if there does not exist another dominating set S' such that $S' \subset S$. In this section we present a self-stabilizing protocol for finding a minimal dominating set under a distributed daemon.

Algorithm MDS is shown in Figure 2. Each node maintains a flag x and an integer variable c. The value $x(i) = 0$ indicates that $i \notin S$, while the value $x(i) = 1$ indicates that $i \in S$. The variable $c(i)$ is designed to count $N(i) \cap S$. It can have one of three values: $c(i) = 0$ means that i has no neighbor in S, $c(i) = 1$ that i has exactly one neighbor in S, and $c(i) = 2$ that i has two or more neighbors in S. However, The value of $c(i)$ is ignored if $x(i) = 1$.

Rule **D1** tries to ensure that a node $i \notin S$ has the correct value for $c(i)$. Rule **D2** says that a node i should enter S if it is undominated, has previously declared this by setting $c(i) = 0$, and has ID smaller than any neighbor which has declared itself undominated. Rule **D3** says that a node i should leave S if as far as it can tell it is the unique dominator of neither itself nor any of its neighbors.

We first show the correctness.

Theorem 3 *If Algorithm MDS stabilizes, then the set $S = \{\, i : x(i) = 1 \,\}$ is a minimal dominating set of the network graph.*



$S \equiv \{\, i : x(i) = 1 \,\}$

D1: **if** $(x(i) = 0) \wedge (c(i)$ incorrect$)$
then correct $c(i)$ [rectify neighbor count]

D2: **if** $(|N(i) \cap S| = 0) \wedge (x(i) = 0) \wedge (c(i) = 0) \wedge (\nexists j \in N(i) : j < i, c(j) = 0)$
then $x(i) := 1$ [enter set]

D3: **if** $(|N(i) \cap S| > 0) \wedge (x(i) = 1) \wedge (\forall j \in N(i) - S : c(j) = 2)$
then $x(i) := 0$ and

$$c(i) := \begin{cases} 1, & \text{if } |N(i) \cap S| = 1 \\ 2, & \text{otherwise} \end{cases}$$ [leave set]

Fig. 2. Algorithm MDS: Minimal Dominating Set

Proof. Suppose that the set S is not dominating when the protocol has terminated. That is, there is a node i such that $S \cap N[i] = \emptyset$. Let i be such a node with minimum ID. Then, $x(i) = 0$. Further, by rule **D1**, $c(i) = 0$. Consider any neighbor $j \in N(i)$ with ID smaller than i. Then, $x(j) = 0$, but by the choice of i, j is dominated. So, by rule **D1**, $c(j) > 0$. It follows that node i is privileged for rule **D2**, a contradiction. Therefore, S is dominating.

Suppose that S is dominating but not minimal. Then there is a node $i \in S$ such that $S - \{i\}$ is a dominating set. It follows that, for each $j \in N[i]$, we have $|N[j] \cap S| > 1$. If $j \in N(i) - S$, then by rule **D1**, since $x(j) = 0$, $c(j) = 2$. Hence, rule **D3** must apply to node i, a contradiction. Thus, S is a minimal dominating set. □

We second show the convergence.

Lemma 4.1 *If $x(i)$ changes from 0 to 1, then $x(i)$ will not change again.*

Proof. Since $x(i)$ changes from 0 to 1, by rule **D2**, all nodes j in the neighborhood $N(i)$ have $x(j) = 0$. By rule **D2**, only the node of i and j with smaller ID is able to enter, so $x(j)$ does not change in the same time-step. So after the time-step, no neighbor of i is in S. After that, no neighbor j will enter S since there is at least one node (namely i) in $S \cap N(j)$, and i will not leave S since none of its neighbors is in S. □

Theorem 4 *Starting from any arbitrary state, Algorithm MDS stabilizes in at most $4n + 1$ time-steps under the synchronous daemon.*

Proof. By Lemma 4.1, each node will change its x-value at most twice. Therefore, there can be at most $2n$ changes of x-values on all nodes in all the time. If there is no change in x-value of any node in a time-step, then the time-step involves only changes in c-values. The change in a c-value is determined only by x-values. Since we are working with the synchronous daemon, there cannot be two consecutive time-steps without a change in x-value. Therefore, the upper bound of execution time is $4n + 1$ time-steps. □

The convergence for a general distributed daemon is surprisingly fast:

Theorem 5 *Starting from any arbitrary state, Algorithm MDS stabilizes in at most $5n$ moves under the distributed daemon.*

Proof. We claim that every node i moves at most 5 times.

Assume first that $c(i)$ never changes to 0. By Lemma 4.1, if i executes rule **D2**, then it does not move again. Thus we may assume that after its first move, $x(i) = 0$. So, apart from possibly its first move being rule **D3**, node i makes only rule **D1** moves. Each such move changes the value of $c(i)$, which must oscillate between 1 and 2. Each 1–2 move is due to a neighbor entering; once two neighbors have entered, i has two neighbors in S until the end of the algorithm, and so cannot move again. It follows that the longest possible sequence of changes for $c(i)$ is ?–2–1–2–1–2.

Assume second that $c(i)$ changes to 0 at some point. No neighbor enters before i goes to 0. So before the move $c(i) \leftarrow 0$, i may make at most two moves (a leave move or a $c(i) \leftarrow 2$ move perhaps followed by a $c(i) \leftarrow 1$ move). After $c(i)$ becomes 0, i may make either an enter move, or a $c(i) \leftarrow 1$ move perhaps followed by a $c(i) \leftarrow 2$ move. This proves the claim. □

Synchronous example. We show next how to construct a network where Algorithm MDS takes $\Theta(n)$ rounds.

Let m be a positive integer. The network $\mathcal{G}_m$ has $5m+1$ nodes. These nodes are labeled, in increasing ID, $A_0, \ldots, A_m$, $B_0, \ldots, B_{m-1}$, $B'_0, \ldots, B'_{m-1}$, $C_0, \ldots, C_{m-1}$ and $C'_0, \ldots, C'_{m-1}$. The edges are A_iB_i, $A_iB'_i$, B_iC_i, $B'_iC'_i$, $C_iC'_i$, C_iA_{i+1} and C'_iA_{i+1} for $0 \leq i \leq m-1$. The network starts with all C_i and C'_i in S, and the remaining nodes with correct c-values.

The network may be thought of a collection of blocks: block i is the subgraph induced by the set $\{A_i, B_i, B'_i, C_i, C'_i, A_{i+1}\}$. We depict this in Figure 3. We use a shaded circle to represent a node in S (i.e. x-value equals 1), and an unshaded circle to represent a node not in S (i.e. x-value equals 0). Recall that the value of c is ignored if a node is in S.

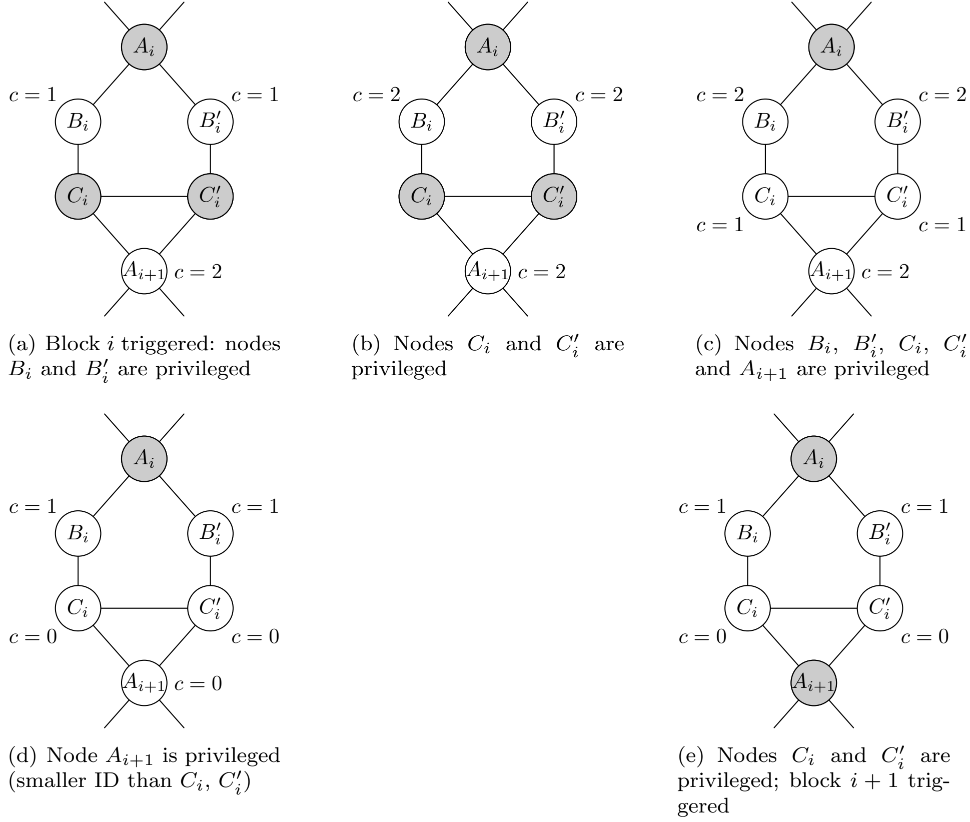
Lemma 4.2 *Algorithm MDS run on $\mathcal{G}_m$ in the starting configuration described takes $4m+2$ rounds.*

Proof. At the start, only node A_0 is privileged. The first 5 steps are as follows: (a) A_0 enters S; (b) B_0 and B'_0 correct their c-value; (c) C_0 and C'_0 leave S; (d) B_0, B'_0, C_0, C'_0 and A_1 correct their c-value; and (e) A_1 enters S (it has lower ID than C_0 and C'_0).

On the next round, C_0 and C'_0 correct their c-values. At the same time, B_1 and B'_1 correct their c-values, and a similar sequence of moves ensues in block 1, culminating in A_2 entering S. This then triggers similar moves inside block 2. And so on. As the algorithm runs, the set will slowly change to where all the A_i nodes are in the set; node A_m entering S in step $4m+1$, and C_{m-1} and C'_{m-1} make the final moves the following round. □

5. Maximal Matching

Given an undirected graph $G = (V, E)$, a *matching* is defined to be a subset $M \subseteq E$ of pairwise disjoint edges. That is, no two edges in M are incident with the same node. A matching M is *maximal* if there does not exist another matching M' such that $M' \supset M$. In this section, we present a self-stabilizing protocol for finding a maximal matching in an arbitrary network graph under a distributed daemon. The



(a) Block i triggered: nodes B_i and B'_i are privileged

(b) Nodes C_i and C'_i are privileged

(c) Nodes B_i, B'_i, C_i, C'_i and A_{i+1} are privileged

(d) Node A_{i+1} is privileged (smaller ID than C_i, C'_i)

(e) Nodes C_i and C'_i are privileged; block $i+1$ triggered

Fig. 3. Synchronous execution of Algorithm MDS through block i

protocol is similar to the central-daemon algorithm for maximal matching given by Hsu and Huang [7].

Algorithm MM is shown in Figure 4. Each node i maintains a single pointer variable which is either null, denoted by $i \downarrow$, or points to one of its neighbors j, denoted by $i \rightarrow j$. We say a node i is *matched* if there exists another node $j \in N(i)$ such that $i \rightarrow j \wedge j \rightarrow i$ (denoted by $i \leftrightarrow j$); it is *idle* if its pointer is null. In rule **M1**, an idle node i may select a node $j \in N(i)$, among those that are pointing to it, and become matched with j. Informally, we say that i *accepts* a proposal made by j. Rule **M2** allows a node i (which is idle and has no neighbors currently pointing to it), to point to an idle neighbor j. Note that i may not select an arbitrary neighbor, but rather the idle neighbor with minimum ID. Informally, we say that i *proposes* to j. Rule **M3** is executed by i when i is pointing to a neighbor j which in turn is pointing to another node $k \neq i$. In this case, i sets its pointer to null, and we say that it *backs off*.

Define M as the set of edges $\{i, j\}$ such that $i \leftrightarrow j$.

Theorem 6 *If Algorithm MM stabilizes, then the set M defined above is a maximal matching of the network graph.*

M1: if $(i\downarrow) \wedge (\exists j \in N(i) : j \to i)$	
then $i \to j$	[accept proposal]
M2: if $(i\downarrow) \wedge (\forall k \in N(i) : k \not\to i) \wedge (\exists j \in N(i) : j\downarrow)$	
then $i \to \min\{j \in N(i) : j\downarrow\}$	[make proposal]
M3: if $(i \to j) \wedge (j \to k$ where $k \neq i)$	
then $i\downarrow$	[back off]

Fig. 4. Algorithm MM: Maximal Matching

Proof. It is clear that M is always a matching. Suppose that the protocol has terminated but M is not maximal. Then, since no node is privileged for rule **M3**, every node is either matched or idle. Since M is not maximal, there are two idle nodes that are adjacent. But then both are privileged for rule **M2**, a contradiction. □

Lemma 5.3 *Once a node is matched it remains matched.*

Proof. If node i is matched, then $i \leftrightarrow j$ for some neighbor j. Clearly, neither i nor j is privileged, so they remain matched. □

Lemma 5.4 *Consider a time-step where at least one node is tapped for rule **M2** but no new match occurs. Then there exists some node that is proposed to but is not tapped.*

Proof. Suppose otherwise. Then one can follow the proposals: node i proposes to node j which proposes to node k and so on. Since the graph is finite, there must exist a cycle.

But consider the node in the cycle with the largest ID, say u. Say node v proposes to u, and node w proposes to node v. Then, since w has smaller ID than node u, node v should propose to w, a contradiction. □

Lemma 5.5 *There cannot be two consecutive rounds without a new match.*

Proof. Consider a round R without a new match. We show that the next round, if any, must create a new match.

Assume first that during round R some node executes rule **M2**. Then, by Lemma 5.4, at that time-step some node x is proposed to which is not tapped. It follows that x is privileged at the end of R, and must execute rule **M1** by the end of the following round, creating a new match.

Assume second that during round R no node executes rule **M2**. That is, all moves in R are due to rule **M3**. It follows that by the end of round R, every node is either matched, idle or points to a neighbor that is idle.

So, the first time-step of the next round is an execution of rule **M1** or rule **M2**. If no new match occurs, this must be rule **M2**. Then, by Lemma 5.4, some node x must be proposed to. But x was privileged at the start of the round, and so must accept by the end of the round, creating a new match. □

Theorem 7 *Starting from an arbitrary state, Algorithm MM stabilizes in at most n rounds, where n is the number of nodes.*

Proof. By Lemma 5.3, matched nodes remain matched. By Lemma 5.5, there cannot be two consecutive rounds without a new match. Since every new match matches two nodes, the theorem follows. □

The number of time-steps is a bit larger than for the previous algorithms:

Lemma 5.6 *Algorithm MM stabilizes in at most $O(n^3)$ time-steps under a distributed daemon.*

Proof. We know there are $O(n)$ time-steps where a new match occurs.

We claim that there are at most n^2 time-steps in which some node executes rule **M2** but no new match occurs. By Lemma 5.4, in each such time-step there is some node that is pointed to but not tapped. Each node can be pointed to only $n-1$ times. The claim follows.

In between the above time-steps, each node can execute rule **M3** at most once. Thus the total number of time-steps between two matches is at most $O(n^2)$. Since there can be at most $n/2$ matches, it follows that the total number of time-steps is at most $O(n^3)$. □

Here is an example where Algorithm MM takes $\Theta(n^3)$ time-steps. Start with an independent set A of m nodes and an independent set B of m nodes. Join all nodes in A to all nodes in B. Then add a path C of m nodes and join one end of C to all nodes in B. Number the nodes in increasing order starting with A, then B, then the nodes of C on the path starting with the node of attachment and moving away. See Figure 5.

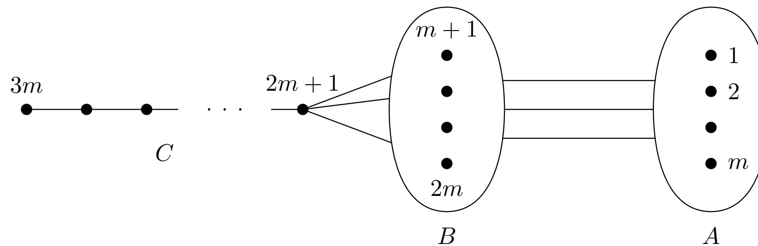


Fig. 5. An example graph for Algorithm MM

The nodes all start idle. The daemon starts by simultaneously tapping all of C and the smallest node of B. This creates a directed path of proposals from the leaf of C to the smallest node of A. Then the daemon taps the nodes of C one at a time, highest ID first, to back off. Then it taps all of C and the second-smallest node of B. And then it taps C again one at a time to back off. And repeats. Once this has been done m times, the daemon taps the smallest node in A, which matches to some node in B. Then the daemon backs off the remaining nodes in B, and we repeat the whole process from scratch. There are $O(m^2)$ time-steps between new matches.

6. Conclusions

We have proposed self-stabilizing distributed algorithms for maintaining three sets in a system graph. The algorithms (protocols) stabilize in $O(n)$ rounds under a distributed daemon and hence in $O(n)$ time-steps under the synchronous daemon. They can be implemented on ad hoc networks. While it is known that problems that are solvable with self-stabilizing algorithms using the central daemon are solvable under the distributed or synchronous daemon, there is no guarantee that the

resulting algorithm will be fast. It is an open question which other graph problems (such as maximal packing, minimal total dominating set or spanning tree creation) have fast distributed or synchronous algorithms.

Acknowledgments

We would like to greatly thank the referees for their comments and suggestions, including ways to simplify Algorithm MDS and pointing out the validity of the algorithms under the distributed daemon. We would also like to thank Dr Volker Turau for his comments and for drawing our attention to reference [15]. This work has been supported by NSF grant # ANI-0218495.

References

[1] H. Abu-Amara, B. Coan, S. Dolev, A. Kanevsky, and J.L. Welch. Self-stabilizing topology maintenance protocols for high-speed networks. *IEEE/ACM Transactions on Networking* **4** (1996), 902–912.
[2] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics.* (McGraw Hill, 1998).
[3] T.W. Haynes, S.T. Hedetniemi, and P.J. Slater. *Fundamentals of Domination in Graphs*, volume 208 of *Monographs and Textbooks in Pure and Applied Mathematics.* (Marcel Dekker Inc., New York, 1998).
[4] S. Fujita, T. Kameda, and M. Yamashita. A resource assignment problem on graphs. In *Proceedings of the 6th International Symposium on Algorithms and Computation* (Cairns, Australia, December 1995), 418–427.
[5] E.W. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, **17** (1974), 643–644
[6] S. Dolev. *Self-Stabilization.* (MIT Press, 2000).
[7] S.C. Hsu and S.T. Huang. A self-stabilizing algorithm for maximal matching. *Information Processing Letters* **43** (1992), 77–81.
[8] 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* (1995), 7.1–7.15.
[9] Y. Afek and S. Dolev. Local stabilizer. In *Proceedings of the 5th Israeli Symposium on Theory of Computing and Systems* (1997), 74–84.
[10] J. Beauquier, A.K. Datta, M. Gradinariu, and F. Magniette. Self-stabilizing local mutual exclusion and daemon refinement. In *DISC00 Distributed Computing 14th International Symposium* (Springer LNCS:1914, 2000), 223–237.
[11] W. Goddard, S.T. Hedetniemi, D.P. Jacobs, and P.K. Srimani. Self-stabilizing algorithms for orderings and colorings. *Int. J. Found. Comput. Sci.* **16** (2005), 19–36.
[12] S. Shukla, D. Rosenkrantz, and S. Ravi. Developing self-stabilizing coloring algorithms via systematic randomization. In *Proceedings of the International Workshop on Parallel Processing* (1994), 668–673.
[13] W. Goddard, S.T. Hedetniemi, D.P. Jacobs, and P.K. Srimani. Self-stabilizing protocols for maximal matching and maximal independent sets for ad hoc networks. In *Proceedings of 5th IPDPS Workshop on Advances in Parallel and Distributed Computational Models* (Nice 2003), 162.
[14] Z. Xu, S.T. Hedetniemi, W. Goddard, and P.K. Srimani. A synchronous self-stabilizing minimal domination protocol in an arbitrary network graph. In *Distributed Computing—IWDC 2003, 5th International Workshop* (Kolkata, India, Springer

LNCS:2918, 2003), 26–32.

[15] M. Ikeda, S. Kamei, and H. Kakugawa. A space-optimal self-stabilizing algorithm for the maximal independent set problem. In *PDCAT Proc. 3rd Int. Conf. Parallel and Distributed Computing, Applications and Technologies* (2002), 70–74.