Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

A new self-stabilizing maximal matching algorithm[☆]

Fredrik Manne[a], Morten Mjelde[a,*], Laurence Pilard[b], Sébastien Tixeuil[c]

[a] *University of Bergen, Norway*
[b] *University of Franche Comté, France*
[c] *LIP6 & INRIA Grand Large, Université Pierre et Marie Curie - Paris 6, France*

A R T I C L E   I N F O

Keywords:
Self-stabilizing algorithms
Maximal matching

A B S T R A C T

The maximal matching problem has received considerable attention in the self-stabilizing community. Previous work has given several self-stabilizing algorithms that solve the problem for both the adversarial and the fair distributed daemon, the sequential adversarial daemon, as well as the synchronous daemon. In the following we present a single self-stabilizing algorithm for this problem that unites all of these algorithms in that it has the same time complexity as the previous best algorithms for the sequential adversarial, the distributed fair, and the synchronous daemon. In addition, the algorithm improves the previous best time complexities for the distributed adversarial daemon from $O(n^2)$ and $O(\delta m)$ to $O(m)$ where n is the number of processes, m is the number of edges, and δ is the maximum degree in the graph.

1. Introduction

A *matching* in an undirected graph is a subset of edges in which no pair of edges are adjacent. A matching M is *maximal* if no proper superset of M is also a matching. Matchings are typically used in distributed applications when pairs of neighboring processes have to be set up (e.g. between a server and a client). As current distributed applications usually run continuously, it is expected that the system is dynamic (processes may leave or join the network), so an algorithm for the distributed construction of a maximal matching should be able to reconfigure on the fly. *Self-stabilization* [4,5] is an elegant approach to forward recovery from transient faults as well as initializing a large-scale system. Informally, a self-stabilizing systems is able to recover from any transient fault in finite time, without restricting the nature or the span of those faults.

The environment of a self-stabilizing algorithm is modeled by the notion of a *daemon*. The use of a daemon allows for reasoning about how the algorithm behaves under different executions models, where various daemons correspond to different environments. There are two main characteristics for the daemon: it can be either *sequential* (or central, meaning that exactly one eligible process is scheduled for execution at a given time) or *distributed* (meaning that any subset of eligible processes can be scheduled for execution at a given time), and in an orthogonal way, it can be *fair* (meaning that in any execution, every eligible processor is eventually scheduled for execution) or *adversarial* (meaning that the daemon only guarantees global progress, i.e. some eligible process is eventually scheduled for execution). An extreme example of a fair daemon is the *synchronous* daemon, where all eligible processes are scheduled for execution at every time step. Of course, any algorithm that works under the distributed daemon will also work with the sequential daemon or the synchronous

[☆] Support for this work was given by the Aurora program for collaboration between France and Norway.
[*] Corresponding author. Tel.: +47 5 558 4294.
E-mail addresses: fredrikm@ii.uib.no (F. Manne), mortenm@ii.uib.no (M. Mjelde), laurence.pilard@iut-bm.univ-fcomte.fr (L. Pilard), tixeuil@lri.fr (S. Tixeuil).

0304-3975/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2008.12.022

daemon, and any algorithm that can handle the adversarial daemon can be used with a fair daemon, but the converse is not true in either case.

There exist several self-stabilizing algorithms for computing a maximal matching in an unweighted general graph. Hsu and Huang [13] gave the first such algorithm and proved a bound of $O(n^3)$ on the number of moves assuming an adversarial daemon. This analysis was later improved to $O(n^2)$ by Tel [15] and finally to $O(m)$ by Hedetniemi et al. [12]. The original algorithm assumes an anonymous network and operates therefore under the sequential daemon in order to achieve symmetry breaking. We will show in Section 2 that in some anonymous symmetric networks there exists no deterministic self-stabilizing solution to the maximal matching problem.

By using randomization, Gradinariu and Johnen [10] proposed a scheme to give processes a local name that is unique within distance 2, and used this scheme to run Hsu and Huang's algorithm under an adversarial distributed daemon. However, only a finite stabilization time was proved. Using the same technique of randomized local symmetry breaking, Chattopadhyay et al. [3] later gave a maximal matching algorithm with $O(n)$ round complexity, but assuming the weaker fair distributed daemon. It is straightforward to show that this algorithm could use $\Omega(n^2)$ moves on a path graph with n processes where the processes are numbered consecutively and where each process is initialized to attempt a matching with it highest numbered neighbor. Using a combination of other results we believe that it is possible to show that the correct upper bound for the step complexity of this algorithm under the adversarial distributed daemon is $O(mn)$. However, as this result is non-trivial we do not include it in the current paper.

In [8] Goddard et al. describe a synchronous version of Hsu and Huang's algorithm and show that it stabilizes in $O(n)$ rounds. Although not explicitly proved in the paper, it can be shown that their algorithm also copes with the adversarial distributed daemon using $\theta(n^2)$ steps. Here, symmetry is broken using unique identifiers at every process. In [11], Gradinariu and Tixeuil provide a general scheme to transform an algorithm using the sequential adversarial daemon into an algorithm that works under the distributed adversarial daemon. Using this scheme with Hsu and Huang's algorithm yields a step complexity of $O(\delta m)$, where δ denotes the maximum degree of the network. Recently, Manne and Mjelde [14] presented a self-stabilizing algorithm for computing a $\frac{1}{2}$-approximation to the maximum weighted matching problem in general graphs. The algorithm is analyzed both when run under the distributed adversarial daemon and the distributed fair daemon.

Our contribution is a new self-stabilizing algorithm that stabilizes after $O(m)$ steps under the distributed adversarial daemon. Under a distributed fair daemon the algorithm stabilizes after $O(n)$ rounds. Thus, this algorithm meets the step complexities of the previous best algorithms for the sequential daemon (which is a special case of the distributed adversarial daemon) and the round complexity for the distributed fair daemon. It also improves the previous best step complexity for the distributed adversarial daemon. As a side effect, we improve the best known algorithm for the adversarial daemon by lowering the environment requirements (distributed instead of sequential). To break symmetry, we assume that process identifiers are unique within distance two. That is, no process has two or more neighbors with identical identifier. The following table compares features of the aforementioned algorithms and ours (best feature for each category is underlined).

Reference	Daemon	Step complexity	Round complexity	Structural information
[12,13,15]	Sequential adversarial	$\underline{O(m)}$		Anonymous
[10]	Distributed adversarial	finite		Distance 2
[3]	Distributed fair	$\Omega(n^2)$	$\underline{O(n)}$	Distance 2
[8]	Synchronous	$O(n^2)$	$\underline{O(n)}$	Unique ID
[11]	Distributed adversarial	$O(\delta m)$		Unique ID
This paper	Distributed adversarial	$\underline{O(m)}$	$\underline{O(n)}$	Distance 2

As seen in the table, the algorithm presented in [13] has the currently best step complexity, while [3] gives the algorithm with the best round complexity with the most relaxed structural requirement (IDs unique at distance two).

Similar to the algorithms given in [3] and [13], the algorithm we present in this paper uses a variable on each process for pointing to a neighbor (requiring $\log \delta$ bits). By doing so a process indicates that it wants to match with the process being pointed to, and if this process is pointing back to the first process then the processes are considered matched. However, our algorithm has two distinctions compared to [3] and [13].

First, we employ a boolean variable at each process that is used to communicate to the neighbors of this process whether or not it is matched. This allows us, contrary to [13], to avoid cases where a process retracts its pointer before the neighbor to which it was pointing, has settled on a match. Note that the introduction of this variable does not significantly increase the memory requirement of our algorithm compared to either [3] and [13] (a single additional bit per process is sufficient).

The algorithm given in [3] uses a scheme in which every process tries to match with the neighbor with smallest ID. This has the effect that even if two neighboring processes agree that they are matched, one or both of them may chose to leave the matching at a later point in the execution. In the algorithm presented here, if two neighboring processes become matched, they remain matched for the duration of the algorithm.

The rest of this paper is organized as follows. In Section 2 we give a short introduction to self-stabilizing algorithms and the computational environment we use. In Section 3 we describe our algorithm and prove its correctness and speed of convergence in Section 4. Finally, in Section 5 we conclude.

2. Model

A system consists of a set of processes where two adjacent processes can communicate with each other. The communication relation is typically represented by a graph $G = (V, E)$ where $|V| = n$ and $|E| = m$. Each process corresponds to a node in V and two processes i and j are adjacent if and only if $(i, j) \in E$. As mentioned in Section 1, we assume that each process has an identifier that is unique at distance two. At the end of this section we will justify this requirement. In the following we will not distinguish between a process and its identifier.

The set of neighbors of a process $i \in V$ is denoted by $N(i)$. The neighbors of a set of processes $A \subseteq V$ is defined as $N(A) = \{j \in V - A, \exists i \in A \text{ s.t. } (i, j) \in E\}$. A process maintains a set of local variables that make up the local state of the process. Each variable ranges over a fixed domain of values. Every process executes the same algorithm, which consists of one or more rules. A rule has the form $\langle name \rangle$: **if** $\langle guard \rangle$ **then** $\langle command \rangle$. A *guard* is a boolean predicate over the variables of both the process and those of its neighbors. A *command* is a sequence of statements assigning new values to the variables of the process.

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

A *daemon* is a predicate on executions. We distinguish several kinds of daemons: the *sequential* daemon makes the system move from one configuration to the next by executing exactly one enabled rule, the *synchronous* daemon makes the system move from one configuration to the next one by executing exactly one rule on all eligible processes, the *distributed* daemon makes the system move from one configuration to the next one by allowing any non-empty set of eligible processes to execute exactly one rule. Note that the sequential and synchronous daemons are instances of the more general (i.e. less constrained) distributed daemon. Also, a daemon is *fair* if any rule that is continuously enabled is eventually executed, and *adversarial* if it may execute *any* enabled rule at every step. Again, the adversarial daemon is more general than the fair daemon.

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

We consider two measures for evaluating complexity of self-stabilizing algorithms. A step is the minimum unit of time such that a process can perform any one of its moves. For a distributed daemon there can be several processes that makes simultaneous moves during one step, thus the *step complexity* investigates the maximum number of steps that are needed to reach a configuration that conforms to the specification (i.e. a *legitimate* configuration), for all possible starting configurations. The *round* complexity considers that executions are observed in rounds: a round is the smallest sub-sequence of an execution in which every process eligible for at least one move at the start of a round has either executed one of these moves during the round, or has become ineligible to do so. Note that both of these types of analysis focus on communication and not on computation, as it is assumed that a process can perform any type of necessary local computation during one move.

We justify our assumption of process identifiers by observing that the simultaneous hypotheses of a distributed unfair daemon and anonymous processes results in a model where it is impossible to solve the maximal matching problem using a deterministic self-stabilizing algorithm. The proof technique is similar to that of [1].

Lemma 2.1. *There exists no deterministic self-stabilizing algorithm for the maximal matching problem that can simultaneously work under the synchronous daemon and perform in arbitrary anonymous networks.*

Proof. Assume that for every process in the network, the local state is either unmatched or proposed matched with one specific neighbor. A particular configuration is a valid matching if every pair of processes is consistent in their mutual relationship (i.e. every pair of processes either agrees on being matched or not). We now consider a network that is a ring of size at least 3. Initially, each process has the same state, and since the local view of every process is identical, it follows that their moves (if any) will also be identical. Observe that every process has exactly two neighbors, and a process' proposed matching may either be directed clockwise or counter-clockwise. Thus, each process is either *(1)* unmatched, *(2)* clockwise proposed matched, or *(3)* counter-clockwise proposed matched. Since the local state of every process is identical and no process is matched, the initial configuration is not a maximal matching.

Thus, the view of each process is identical and it follows that the processes will change between the three mentioned cases in a lockstep fashion and neither reach a stable state nor a maximal matching. ■

3. The algorithm

In the following we present and motivate our algorithm for computing a maximal matching. The algorithm is self-stabilizing and does not make any assumptions on the network topology. A set of edges $M \subseteq E$ is a *matching* if and only if edges $x, y \in M$ implies that x and y do not share a common end point. A matching M is *maximal* if no proper superset of M is also a matching.

Algorithm 1 A self-stabilizing maximal matching algorithm

```
Variables of process i:
    $m_i \in \{true, false\}$
    $p_i \in \{null\} \cup N(i)$

Predicate:
    $PRmarried(i) \equiv \exists j \in N(i) : (p_i = j \textbf{ and } p_j = i)$

Rules:
    Update:
        if $m_i \neq PRmarried(i)$
        then $m_i := PRmarried(i)$

    Marriage:
        if $m_i = PRmarried(i) \textbf{ and } p_i = null \textbf{ and } \exists j \in N(i) : p_j = i$
        then $p_i := j$

    Seduction:
        if $m_i = PRmarried(i) \textbf{ and } p_i = null \textbf{ and } \forall k \in N(i) : p_k \neq i$
            and $\exists j \in N(i) : (p_j = null \textbf{ and } j > i \textbf{ and } \neg m_j)$
        then $p_i := \max\{j \in N(i) : (p_j = null \textbf{ and } j > i \textbf{ and } \neg m_j)\}$

    Abandonment:
        if $m_i = PRmarried(i) \textbf{ and } p_i = j \neq null \textbf{ and } p_j \neq i \textbf{ and } (m_j \textbf{ or } j \leq i)$
        then $p_i := null$
```

Each process i has a variable p_i pointing to one of its neighbors or to *null*. Processes i and j are *married* to each other if and only if i and j are neighbors and $p_i = j$ and $p_j = i$. In this case we will also refer to i as being married without specifying j. However, we note that if i is married then $j = p_i$ is well defined. A process that is not married is *unmarried*.

In addition to p_i, process i has a variable m_i that is used to let neighboring processes know if i is married or not. The value of m_i is determined by a predicate *PRmarried(i)*. This evaluates to true if and only if i is married. Thus, *PRmarried(i)* allows process i to know if it is currently married and the variable m_i allows neighbors of i to determine if i is married. Note that the value of m_i is not necessarily always equal to *PRmarried(i)*.

If $p_i \notin N(i) \cup \{null\}$, then we define $p_{p_i} \neq i$ and $m_{p_i} = true$. As we shall see later, this will cause i to become eligible for a move if it is pointing to a process not in its neighborhood.

Our self-stabilizing scheme is given in Algorithm 1. It is composed of four mutual exclusive guarded rules as described below.

The *Update* rule updates the value of m_i if this is incorrect, while the three other rules can only be executed if the value of m_i is correct. In the *Marriage* rule, an unmarried process that is currently being pointed to by a neighbor j tries to marry j by setting $p_i = j$. In the *Seduction* rule, an unmarried process that is not being pointed to by any neighbor, points to an unmarried neighbor with the objective of marriage. Note that the identifier of the chosen neighbor has to be larger than that of the current process. This is enforced to avoid the creation of cycles of pointer values. In the *Abandonment* rule, a process i resets its p_i value to *null*. This is done if the process j, to which i is pointing, does not point back to i and if either *(1)* j is apparently married, or *(2)* j has a lower identifier than i. Condition *(1)* allows a process to stop waiting for an already married process while the purpose of Condition *(2)* is to break a possible initial cycle of p-values.

We note that if *PRmarried(i)* is true at some point of time then it will remain true throughout the execution of the algorithm. Moreover, the algorithm will never actively create a cycle of pointing values since the *Seduction* rule enforces that $j > i$ before process i will point to process j. Also, all initial cycles are eventually broken since the guard of the *Abandonment* rule requires that $j \leq i$.

Fig. 1 gives a short example of the execution of the algorithm. The initial configuration is as shown in Fig. 1a, where $i > j > k$. Here, both processes j and k are attempting to become married to i. In Fig. 1b process i has executed a *Marriage* move, and i and j are now married. In Fig. 1c both i and j execute an *Update* move, setting their m-values to *true*. And finally, in Fig. 1d process k executes an *Abandonment* move.

4. Proof of correctness

In the following we will first show that when Algorithm 1 has reached a stable configuration then this also defines a maximal matching. We will then bound the number of steps the algorithm needs to stabilize both for the adversarial

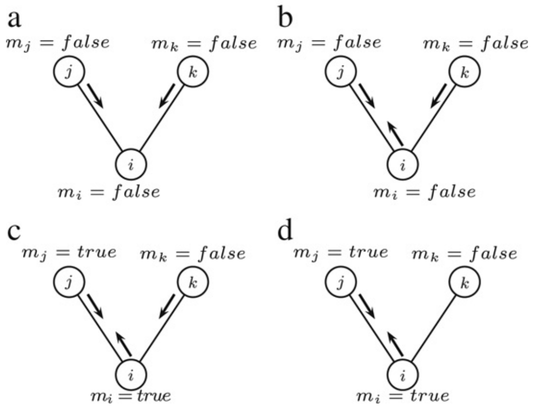

Fig. 1. Example of an execution of Algorithm 1.

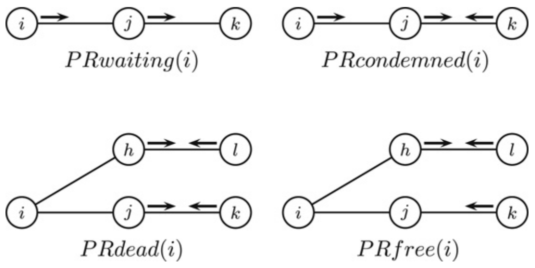

Fig. 2. Examples where the four predicates $PRwaiting(i)$, $PRcondemned(i)$, $PRdead(i)$, and $PRfree(i)$ are true.

distributed daemon and for the fair distributed daemon. Note that the sequential daemon is a subset of the distributed one, thus any result for the latter also applies to the former.

4.1. Correct stabilization

We now proceed to show that if Algorithm 1 reaches a stable configuration then the p- and m-values define a maximal matching M where $(i, j) \in M$ if and only if $(i, j) \in E$, $p_i = j$, and $p_j = i$ while both m_i and m_j are true. Recall from Section 2 that a configuration is stable if no process is eligible to execute a move. In order to perform the proof, we define the following six mutual exclusive predicates, similar to what was done in [13]:

$$\begin{aligned}
PRinvalid(i) &\equiv (p_i \neq null) \textbf{ and } (p_i \notin N(i)) \\
PRmarried(i) &\equiv \exists j \in N(i) : (p_i = j \textbf{ and } p_j = i) \\
PRwaiting(i) &\equiv \exists j \in N(i) : (p_i = j \textbf{ and } p_j \neq i \textbf{ and } \neg PRmarried(j)) \\
PRcondemned(i) &\equiv \exists j \in N(i) : (p_i = j \textbf{ and } p_j \neq i \textbf{ and } PRmarried(j)) \\
PRdead(i) &\equiv (p_i = null) \textbf{ and } (\forall j \in N(i) : PRmarried(j)) \\
PRfree(i) &\equiv (p_i = null) \textbf{ and } (\exists j \in N(i) : \neg PRmarried(j)).
\end{aligned}$$

Note first that each process will evaluate exactly one of these predicates to true. Moreover, also note that $PRmarried(i)$ is the same as in Algorithm 1.

Fig. 2 gives an example where each of the predicates $PRwaiting(i)$, $PRcondemned(i)$, $PRdead(i)$, and $PRfree(i)$ is true. An example where $PRmarried(i)$ is true was shown in Fig. 1d. $PRinvalid(i)$ is not shown.

We now show that in a stable configuration each process i evaluates either $PRmarried(i)$ or $PRdead(i)$ to true, and when this is the case, the p-values define a maximal matching. To do so, we first note that in any stable configuration the m-values reflects the current status of the processes.

Lemma 4.1. *In a stable configuration $m_i = PRmarried(i)$ for each $i \in V$.*

Proof. This follows directly since if $m_i \neq PRmarried(i)$ then i is eligible to execute the *Update* rule. ■

We next show in the following four lemmas that no process will evaluate neither $PRinvalid(i)$, $PRwaiting(i)$, $PRcondemned(i)$, nor $PRfree(i)$ to true in a stable configuration.

Lemma 4.2. *In a stable configuration $PRinvalid(i)$ is false for each $i \in V$.*

Proof. First note that if there exists a process i such that $PRmarried(i) \neq m_i$, then i is eligible for an *Update* move. Assume therefore that this is not the case, and that $PRinvalid(i) = true$. Recall that if $p_i \notin N(i) \cup \{null\}$, then by definition $p_{p_i} \neq i$ and $m_{p_i} = true$. Thus, it follows from the guard of the *Abandonment* rule that if $PRinvalid(i) = true$, then i must be eligible for an *Abandonment* move. ■

Lemma 4.3. *In a stable configuration $PRcondemned(i)$ is false for each $i \in V$.*

Proof. If there exists at least one process i in the current configuration such that $PRcondemned(i)$ is true then p_i is pointing to a process $j \in N(i)$ that is married to a process k where $k \neq i$. From Lemma 4.1 it follows that in a stable configuration we have $m_i = PRmarried(i)$ and $m_j = PRmarried(j)$. Thus, if this is the case then the predicate ($m_i = PRmarried(i)$ **and** $p_i = j$ **and** $p_j \neq i$ **and** m_j) evaluates to true. But then process i is eligible to execute the *Abandonment* rule and it follows that the current configuration cannot be stable. ■

Lemma 4.4. *In a stable configuration $PRwaiting(i)$ is false for each $i \in V$.*

Proof. Assume that the current configuration is stable and that there exists at least one process i such that $PRwaiting(i)$ is true. Then it follows that p_i is pointing to a process $j \in N(i)$ such that $p_j \neq i$ and j is unmarried. Note first that if $p_j = null$ then process j is eligible to execute a *Marriage* move. Also, if $j < i$ then process i can execute an *Abandonment* move.

Assume therefore that $p_j \neq null$ and that $j > i$. From Lemma 4.2 we have that $PRinvalid(j) = false$ and from Lemma 4.3 that $PRcondemned(j) = false$. Since j is not married we also have $PRmarried(j) = false$. Thus, $PRwaiting(j)$ must be true. By repeating the same argument for j as we just did for i, it follows that if both i and j are ineligible for any move then there must exist a process k such that $p_j = k$, $k > j$, and $PRwaiting(k)$ also evaluates to true. This sequence of processes satisfying the $PRwaiting$ predicate cannot be extended indefinitely since each process must have a higher identifier than the preceding one. Thus, there must exist some process in V that is eligible for a move and the assumption that the current configuration is stable is incorrect. ■

Lemma 4.5. *In a stable configuration $PRfree(i)$ is false for each $i \in V$.*

Proof. Assume that the current configuration is stable and that there exists at least one process i such that *PRfree(i)* is true. Then it follows that $p_i = null$ and that there exists at least one process $j \in N(i)$ such that j is not married.

Next, we look at the value of the different predicates for the process j. Since j is not married it follows that $PRmarried(j)$ evaluates to false. Also, from Lemmas 4.2–4.4 we have that $PRinvalid(j)$, $PRwaiting(j)$, and $PRcondemned(j)$ must evaluate to false. Finally, since i is not married we cannot have $PRdead(j)$. Thus, we must have $PRfree(j)$. But then the process with the smaller identifier of i and j is eligible to propose to the other by executing a *Seduction* move, contradicting the assumption that the current configuration is stable. ■

From Lemmas 4.2–4.5 we immediately get the following corollary.

Corollary 4.6. *In a stable configuration either $PRmarried(i)$ or $PRdead(i)$ holds for every $i \in V$.*

We can now show that a stable configuration also defines a maximal matching.

Theorem 4.7. *In any stable configuration the p- and m-values define a maximal matching.*

Proof. From Corollary 4.6 we know that in a stable configuration either $PRmarried(i)$ or $PRdead(i)$ holds for every $i \in V$. Also, from Lemma 4.1 it follows that m_i is true if and only if i is married. It is then straightforward to see that the p-values define a matching.

To see that this matching is also maximal, assume to the contrary that it is possible to add one more edge (i, j) to the matching while it still remains a legal matching. For this to be possible we must have $p_i = null$ and $p_j = null$. Thus, we have $\neg PRmarried(i)$ and $\neg PRmarried(j)$ which again implies that both $PRdead(i)$ and $PRdead(j)$ evaluates to true. But according to the $PRdead$ predicate two adjacent processes cannot be dead at the same time. It follows that the current matching is maximal. ■

4.2. Convergence for the distributed adversarial daemon

In the following we will show that Algorithm 1 will reach a stable configuration after at most $4 \cdot n + 2 \cdot m$ steps when executed under the distributed adversarial daemon.

First, we note that as soon as two processes are married they will remain so for the rest of the execution of the algorithm.

Lemma 4.8. *If processes i and j are married in a configuration C, i.e. $p_i = j$ and $p_j = i$, then they will remain married in any ensuing configuration C'.*

Proof. Assume that $p_i = j$ and $p_j = i$ in some configuration C. Then process i can neither execute the *Marriage* nor the *Seduction* rule since these require that $p_i = null$. Similarly, i cannot execute the *Abandonment* rule since this requires that $p_j \neq i$. The exact same argument for process j shows that j also cannot execute any of the three rules *Marriage*, *Seduction*, and *Abandonment*. Thus, the only rule that processes i and j can execute is *Update* but this will not change the values of p_i or p_j. ■

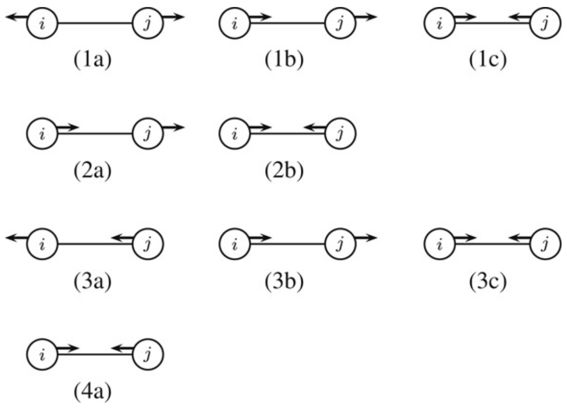

Fig. 3. The four series of i, j-moves used in Lemma 4.11.

A process signals that it is married through executing the *Update* rule. Thus, this is the last rule a married process will execute in the algorithm. This is reflected in the following.

Corollary 4.9. *If a process i executes an* Update *move and sets $m_i = true$ then i will not move again.*

Proof. From the predicate of the *Update* rule it follows that if process i sets $m_i =$ true then there must exist a process $j \in N(i)$ such that $p_i = j$ and $p_j = i$. From Lemma 4.8 process i cannot make any subsequent move that changes the value of p_i. Thus, the only subsequent move i can make is another *Update* move. But since the value of m_i is correct and since p_i and p_j will not change again, this will not happen. ■

Because a married process cannot become "divorced" we also have the following restriction on the number of times the *Update* rule can be executed by any process.

Corollary 4.10. *Any process executes at most two* Update *moves.*

We now progress to bound the number of moves from the set $MSA = \{Marriage, Seduction, Abandonment\}$. Recall that for the distributed adversarial daemon we measure complexity in steps, where one step is a non-empty set of moves executed simultaneously. Note that each MSA move is performed by a process i in relation to a neighbor j. We denote any such move made by either i or j with respect to the other as an *i, j-move*. Furthermore, we define an *i, j-step* as containing an i, j-move. Thus, for a particular edge $(i, j) \in E$ every i, j-step contains either one or two i, j-moves.

Lemma 4.11. *For any edge $(i, j) \in E$, there can at most be three i, j-steps.*

Proof. Let $(i, j) \in E$ be an edge such that $i < j$. Initially we can have $p_i = j$ or $p_i \neq j$ and similarly $p_j = i$ or $p_j \neq i$. In the following we consider each such combination, as illustrated in the left column of Fig. 3. In the figure, the case where the arrow of process i is pointing away from process j illustrates that $p_i \neq j$ (which also includes $p_i = null$).

Case (1): $p_i \neq j$ and $p_j \neq i$ (Fig. 3.1a). Since $i < j$ the first i, j-step cannot contain process j executing a *Seduction* move. Also, as long as $p_i \neq j$, process j cannot execute a *Marriage* move. Thus, process j cannot execute an i, j-move until after process i has first made an i, j-move. It follows that the first possible i, j-step consists of i executing a *Seduction* move (Fig. 3.1b). Note that at the starting configuration of this step we must have $\neg m_j$.

If the second i, j-step only involves j then this must be a *Marriage* move which results in $p_i = j$ and $p_j = i$ (Fig. 3.1c). Then by Lemma 4.8 there will be no more i, j-steps. However, if the second i, j-step contains process i executing an i, j-move (independently of what process j does) then this must be an *Abandonment* move (not shown in Fig. 3). But this requires that the value of m_j has changed from false to true. Then by Corollary 4.9 process j will not make any more i, j-moves and since $p_j \neq null$ and $p_j \neq i$ for the rest of the algorithm, it follows that process i cannot execute any future i, j-moves. Thus, at most two i, j-steps are performed.

Case (2): $p_i = j$ and $p_j \neq i$ (Fig. 3.2a). If the first i, j-step only involves process j then this must be a *Marriage* move resulting in $p_i = j$ and $p_j = i$ (Fig. 3.2b). Then from Lemma 4.8 neither i nor j will make any future i, j-moves. If the first i, j-step involves process i then this must be an *Abandonment* move. Thus, in the configuration prior to this step we must have $m_j =$ true. It follows that either $m_j \neq PRmarried(j)$ or $p_j \neq null$. In either case process j cannot make an i, j-move simultaneously as i makes its move. Thus, following the *Abandonment* move by process i we are at Case *(1)* and there can at most be two more i, j-steps. Hence, there can at most be a total of three i, j-steps.

Case (3): $p_i \neq j$ and $p_j = i$ (Fig. 3.3a). If the first i, j-step only involves process i then i must execute a *Marriage* move resulting in $p_i = j$ and $p_j = i$ and from Lemma 4.8 neither i nor j will make any future i, j-moves. If the first i, j-step involves process j then this must be an *Abandonment* move. If the same i, j-step does not involve process i then this will result in *Case (1)* and there can at most be two more i, j-steps for a total of three i, j-steps.

If the first i, j-step contains two i, j-moves then these must be an *Abandonment* move by process j and a *Marriage* move by process i (Fig. 3.3b). We are now at a similar configuration as Case *(2)* but with $\neg m_j$. If the second i, j-step involves process i

then this must be an *Abandonment* move implying that m_j has changed to true. It then follows from Corollary 4.9 that process j (and therefore also process i) will not make any future i, j-moves leaving a total of two i, j-steps. If the second i, j-step does not involve i then it must consist of a *Marriage* move performed by process j resulting in $p_i = j$ and $p_j = i$ (Fig. 3.3c) and from Lemma 4.8 neither i nor j will make any future i, j-moves.

Case (4): $p_i = j$ and $p_j = i$ (Fig. 3.4a). In this case it follows from Lemma 4.8 that neither process i nor process j will make any future i, j-moves. ■

It should be noted in the proof of Lemma 4.11 that only an edge (i, j) where we initially have either $p_i = j$ or $p_j = i$ (but not both) can result in three i, j-steps, otherwise the limit is two i, j-steps per edge. When we have three i, j-steps across an edge (i, j) we can charge these steps to the processor that was initially pointing to the other. In this way each process will at most be incident on one edge to which it is charged three steps for.

Furthermore, for a process i, the initial state may be such that $p_i \notin N(i) \cup \{null\}$. In such an event, i will become eligible for an *Abandonment* move after at most one *Update* move. Note that this type of move is not an i, j-move, as the edge $(i, p_i) \notin E$. Since there does not exist a move that can set $p_i \notin N(i) \cup \{null\}$, it follows that this case can occur at most once for every process. From these two observations we can now give the following bound on the total number of steps needed to obtain a stable solution.

Theorem 4.12. *Algorithm 1 will stabilize after at most $4 \cdot n + 2 \cdot m$ steps under the distributed adversarial daemon.*

Proof. From Corollary 4.9 we know that there can be at most $2 \cdot n$ *Update* moves, each which can occur in a separate step. In addition each process can execute at most one *Abandonment* move due to an invalid initial p-value. From Lemma 4.11 it follows that there can at most be three i, j-steps per edge. But as observed, there is at most one such edge incident on each process i for which process i is charged for, otherwise the limit is two i, j-steps. Thus, the total number of i, j-steps is at most $n + 2 \cdot m$ and the result follows. ■

From Theorem 4.12 it follows that Algorithm 1 will use $O(m)$ steps on any system when assuming a distributed daemon. Since the distributed daemon encompasses the sequential daemon this result also holds for the latter.

To see that $O(m)$ is a tight bound for the stabilization time, consider a complete graph in which each process $i_1, i_2, \ldots, i_n$ has a unique identifier such that $i_1 < i_2 < \cdots < i_n$. We will now show that there exists an initial configuration and a sequence of moves such that $\Omega(m)$ moves are executed before the system reaches a stable configuration.

Assume that every process is initially unmarried and pointing to *null*. Then the processes $i_1, \ldots, i_{n-1}$ are eligible to execute *Seduction* moves and point to i_n. Following this, i_n may now execute a *Marriage* move, and become married to i_{n-1}. Thus, the processes $i_1, \ldots, i_{n-2}$ are now eligible to execute *Abandonment* moves. Observe that following these moves, two moves have been executed for every edge incident to i_n, and the processes $i_1, \ldots, i_{n-2}$ are once again pointing to *null*. Furthermore, by Lemma 4.8 we know that neither i_{n-1} nor i_n will execute any further *MSA* moves (note that no moves were executed for any edge incident on i_{n-1}, with the exception of the edge (i_{n-1}, i_n)). In the same manner, we can now reason that in addition to the above, two moves are executed for every edge incident on i_{n-2} (with the exception of those incident on processes with larger identifiers than i_{n-2}). Repeating this argument for every i_{n-2k} where $1 < k < \frac{n}{2}$ shows that $\Omega(m)$ *MSA* moves can be executed before the system reaches a stable configuration.

Finally, we note that Algorithm 1 is very resilient to sporadic faults or to the introduction or deletion of a small number of processes. This follows from Lemma 4.8 which showed that two married processes will remain married for the duration of the algorithm. Thus, a fault in one particular process i will only effect i, the process j that i might have been married to, and also any unmarried neighbors of i and j. Together these processes induce a subgraph containing at most $|N(i)| + |N(j)|$ processes and $|N(i)| + |N(j)| - 1$ edges. From Theorem 4.12 it follows that the system will re-stabilize in $O(|N(i)| + |N(j)|)$ moves and without affecting any other married processes apart from i and j.

4.3. Convergence for the distributed fair daemon

Next we consider Algorithm 1 when analyzed under the distributed fair daemon. The execution of the algorithm remains unchanged but instead of counting steps we now count the number of rounds. One round may encompass several steps, and we only require that every process eligible for at least one move at the start of a round has either executed one of these moves during the round, or has become ineligible to do so (note however that a process may also execute moves for which it becomes eligible over the course of the round). Since we are assuming a distributed daemon, moves made in the same round may be simultaneous. The fair distributed daemon is a subset of the adversarial distributed daemon and therefore any results that were shown in Section 4.2 also apply here. We will now show that Algorithm 1 converges after at most $2 \cdot n + 1$ rounds for this daemon.

We define a process $i \in V$ as *inactive* if either $PRmarried(i)$ or $PRdead(i)$ is true, and *active* otherwise. From Corollary 4.9 it follows that any process $i \in V$ where $PRmarried(i)$ is true will not become active again for the remainder of the execution. This also applies to $PRdead(i)$, since if $PRdead(i) = true$ every process in $N(i)$ is married.

Let $A \subseteq V$ be a maximal connected set of active processes in some arbitrary configuration of the algorithm. We now show that if A contains at least two processes then the size of A must be reduced by at least two within at most four rounds. We show this by contradiction, i.e. by assuming that the size of A remains unchanged over the course of four rounds. Note that if A contains only a single process, it does not have any unmarried neighbors, and thus it may only execute a constant number of moves.

In the following we let $\mathcal{E} = \{\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3, \mathcal{E}_4\}$ denote four consecutive rounds. We assume that $|A| \geq 2$ prior to $\mathcal{E}$ and that A remains unchanged throughout $\mathcal{E}$, implying that no processes in A become married.

Lemma 4.13. *Following $\mathcal{E}_1$ we have for every $i \in A$ and $j \in N(A)$:*

(i) $m_i = PRmarried(i) = false$;
(ii) $m_j = PRmarried(j) = true$;
(iii) *No m-value will change during $\mathcal{E}$;*

Proof. If a process $i \in A$ has $m_i = true$ prior to $\mathcal{E}$, then since $PRmarried(i) = false$ (by definition) i will execute an *Update* move during $\mathcal{E}_1$ and set $m_i = false$.

A process $j \in N(A)$ cannot have $PRdead(j) = true$ since it must have a neighbor $i \in A$ where $PRmarried(i) = false$. Thus, $PRmarried(j) = true$ and if $m_j = false$ prior to $\mathcal{E}$ then it will be set to *true* in $\mathcal{E}_1$.

Since no process in A will become married during $\mathcal{E}$ (by assumption) and no process in $N(A)$ will become "divorced" (from Corollary 4.9) it follows that the m-values of processes in A and $N(A)$ will not change during $\mathcal{E}_2$, $\mathcal{E}_3$, and $\mathcal{E}_4$. ■

From Lemma 4.13 we get the following corollary.

Corollary 4.14. *Following $\mathcal{E}_1$, no inactive process executes a move.*

In the following, we consider when the *Abandonment* moves can be executed in $\mathcal{E}$.

Lemma 4.15. *No Abandonment moves can be executed in $\mathcal{E}_3$ or $\mathcal{E}_4$.*

Proof. Assume that $i \in A$ is eligible for an *Abandonment* move in either round $\mathcal{E}_3$ or $\mathcal{E}_4$. Then prior to this round $p_i = j \neq null$, $p_j \neq i$, and either *(1)* $m_j = true$ or *(2)* $j < i$. We look at each case separately.

In *Case (1)* if $m_j = true$ then it follows from Lemma 4.13 that $j \in N(A)$. Thus, in order for p_i to point to j, this must either have been true before $\mathcal{E}$, or i has executed a *Marriage* or *Seduction* move in $\mathcal{E}$. However, for i to execute a *Marriage* move we must have $p_j = i$ and for i to execute a *Seduction* move $p_j = null$ is required. Since $PRmarried(j) = true$ during $\mathcal{E}$, neither of these can be true and we must have $p_i = j$ prior to $\mathcal{E}$. With $m_j = true$, at least after $\mathcal{E}_1$, it follows that i would have been eligible for the *Abandonment* move no later than at the start of $\mathcal{E}_2$, and the move should have been executed no later than during this round.

For *Case (2)*, where $j < i$, then if the predicate is true prior to $\mathcal{E}$, i would have been eligible for an *Abandonment* move no later than at the start of $\mathcal{E}_2$. Thus, we assume that i sets $p_i = j$ in $\mathcal{E}_1$, $\mathcal{E}_2$, or $\mathcal{E}_3$ which can only happen if i executed a *Marriage* move. But then we must have $p_j = i$ prior to this event, and since i and j do not become married (by assumption), j must have executed a simultaneous *Abandonment* move and set $p_j \neq i$. Process j can only execute an *Abandonment* move if $m_i = true$ (since $j < i$). This can only happen during $\mathcal{E}_1$ and would force i to execute an *Update* move instead of a *Marriage* move. ■

As was observed in the proof of Lemma 4.15, if we assume that no processes become married, then any *Marriage* move must be executed simultaneously with a corresponding *Abandonment* move. From Lemma 4.15 we know that no *Abandonment* moves are executed after $\mathcal{E}_2$, and thus we get the following corollary.

Corollary 4.16. *If no processes in A become married, then no process can execute a Marriage move in $\mathcal{E}_3$ or $\mathcal{E}_4$.*

At this point we have shown that *Update*, *Abandonment*, and *Marriage* moves cannot occur after $\mathcal{E}_2$. Now consider a maximal path of processes $i_1, i_2, \ldots, i_k$ (where $k > 0$) such that $p_{i_x} = i_{x+1}$ and $i_x < i_{x+1}$ for $1 \leq x < k$. Since each process can only occur once on this path, it must be of finite length. If such a path exists at the start of $\mathcal{E}_3$ it follows that i_k must be eligible for a *Marriage* move. This move must then be executed in $\mathcal{E}_3$, after which i_k and one of the processes that points to it are married.

Assume therefor that such a path does not exist at the start of $\mathcal{E}_3$. Since no process is eligible for an *Abandonment* move following $\mathcal{E}_2$, this implies that $p_i = null$ for every process $i \in A$. Hence, every process in A that has a neighbor in A with a larger identifier than itself must be eligible for a *Seduction* move. There must exist at least one such process since $|A| \geq 2$. These *Seduction* moves will then be executed in $\mathcal{E}_3$, after which the above path must exist, and it follows that at least two processes become married in $\mathcal{E}_4$.

This contradicts our initial assumption that no processes become married in $\mathcal{E}$, and we get the following lemma.

Lemma 4.17. *Let $A \subseteq V$ be a maximal connected set of active processes in some configuration of the algorithm. If $|A| \geq 2$ then after at most four rounds the size of A has decreased by at least* 2.

Obviously $|A| \leq |V|$, and from Lemma 4.8 we know that once married, a process will remain married for the remainder of the execution of Algorithm 1. From this we get that at most $2 \cdot n$ rounds are needed to compute the maximal matching. However, after the matching has been found every process that was married in the last round may execute an *Update* move, and every remaining unmarried process may execute an *Abandonment* move. Both of these will be performed in the same round. Note that it is not necessary for a process i that is unmarried when the algorithm terminates to execute a final *Update* move as $m_i = false$ after the first round and remains false throughout the algorithm. From this we get the following theorem.

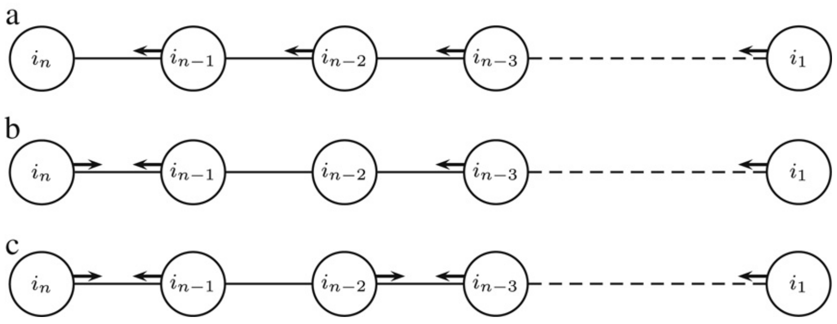

Fig. 4. Example execution of Algorithm 1.

Theorem 4.18. *Algorithm 1 will stabilize after at most $2 \cdot n + 1$ rounds under the fair distributed daemon.*

To see that $O(n)$ is a tight bound, consider a graph that is a path as illustrated in Fig. 4. We assume that each process has a unique identifier, such that $i_1 < i_2 < \cdots < i_n$. If initially no processes are married or pointing to any other process, then following the first round, i_x will be pointing to i_{x+1} for every $1 \leq x < n - 1$, as shown in Fig. 4a. In the second round, i_n will execute a *Marriage* move and become married to i_{n-1}. In round three i_{n-2} will execute an *Abandonment* move (Fig. 4b), and then a *Marriage* move in round four (Fig. 4c). Repeating this argument, we see that one pair of processes are married every other round, which implies that the algorithm needs $\Omega(n)$ rounds to stabilize.

5. Conclusion

We have presented a new self-stabilizing algorithm for the maximal matching problem that improves the step complexity of the previous best algorithm for the distributed adversarial daemon, while at the same time meeting the bounds of the previous best algorithms for the sequential and the distributed fair daemons.

It is well known that a maximal matching is a $\frac{1}{2}$-approximation to the *maximum* matching, where the maximum matching is a matching such that no other matching with strictly greater size exists in the network. In [9], Goddard et al. provide a $\frac{2}{3}$-approximation for a particular class of networks (trees and rings of size not divisible by 3). Also, in particular networks such as trees [2,7] or bipartite graphs [3], self-stabilizing algorithms have been proposed for computing a maximum matching. However, no self-stabilizing solution with a better approximation ratio than $\frac{1}{2}$ currently exists for general graphs. It is possible to collect the graph topology on each process using a self-stabilizing topology update protocol [6] and then run a sequential maximum matching algorithm on each process. This would yield a self-stabilizing maximum matching protocol for general graphs, but at the expense of huge memory and communication consumption. Thus, it would be of interest to know if it is possible to create a memory and time efficient self-stabilizing algorithm for general graphs that achieves a better approximation ratio than $\frac{1}{2}$.

References

[1] Dana Angluin, Local and global properties in networks of processors (extended abstract), in: STOC'80: Proceedings of the twelfth annual ACM symposium on Theory of computing, ACM Press, New York, NY, USA, 1980, pp. 82–93.
[2] Jean R.S. Blair, Fredrik Manne, Efficient self-stabilizing algorithms for tree networks, in: ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems, IEEE Computer Society, Washington, DC, USA, 2003, p. 20.
[3] Subhendu Chattopadhyay, Lisa Higham, Karen Seyffarth, Dynamic and self-stabilizing distributed matching, in: PODC'02: Proceedings of the twenty-first annual symposium on Principles of distributed computing, ACM, New York, NY, USA, 2002, pp. 290–297.
[4] Edsger W. Dijkstra, Self-stabilizing systems in spite of distributed control., Commun. ACM 17 (11) (1974) 643–644.
[5] Shlomi Dolev, Self-stabilization, MIT Press, 2000.
[6] Shlomi Dolev, Ted Herman, Superstabilizing protocols for dynamic distributed systems, Chicago J. Theor. Comput. Sci. 1997 (1997).
[7] Sukumar Ghosh, Arobinda Gupta, Mehmet H. Karaata, Sriram V. Pemmaraju, A self-stabilizing algorithm for maximal matching on trees, Technical Report TR-94-06, Department of Computer Science, The University of Iowa, Iowa City, 1994.
[8] Wayne Goddard, Stephen T. Hedetniemi, David P. Jacobs, Pradip K. Srimani, Self-stabilizing protocols for maximal matching and maximal independent sets for ad hoc networks, in: IPDPS'03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing, IEEE Computer Society, Washington, DC, USA, 2003, p. 162.2.
[9] Wayne Goddard, Stephen T. Hedetniemi, Zhengnan Shi, An anonymous self-stabilizing algorithm for 1-maximal matching in trees, in: PDPTA 2006: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications & Conference on Real-Time Computing Systems and Applications, vol. 2, CSREA Press, 2006, pp. 797–803.
[10] Maria Gradinariu, Colette Johnen, Self-stabilizing neighborhood unique naming under unfair scheduler., in: Euro-Par, in: Lecture Notes in Computer Science, vol. 2150, Springer, 2001, pp. 458–465.
[11] Maria Gradinariu, Sébastien Tixeuil, Conflict managers for self-stabilization without fairness assumption, in: ICDCS 2007: Proceedings of the International Conference on Distributed Computing Systems, IEEE, 2007.
[12] Stephen T. Hedetniemi, David Pokrass Jacobs, Pradip K. Srimani, Maximal matching stabilizes in time o(m)., Inf. Process. Lett. 80 (5) (2001) 221–223.
[13] Su-Chu Hsu, Shing-Tsaan Huang, A self-stabilizing algorithm for maximal matching., Inf. Process. Lett. 43 (2) (1992) 77–81.
[14] Fredrik Manne, Morten Mjelde, A self-stabilizing weighted matching algorithm, in: 9th International Symposium on Stabilization, Safety, and Security of Distributed Systems, SSS'07, in: Lecture Notes in Computer Science, vol. 4838, Springer, 2007, pp. 383–393.
[15] Gerard Tel, Maximal matching stabilizes in quadratic time., Inf. Process. Lett. 49 (6) (1994) 271–272.