A Self-stabilizing Weighted Matching Algorithm

Fredrik Manne and Morten Mjelde

University in Bergen, Norway
{fredrik.manne, mortenm}@ii.uib.no

Abstract. The problem of computing a matching in a graph involves creating pairs of neighboring nodes such that no node is paired more than once. Previous work on the matching problem has resulted in several self-stabilizing algorithms for finding a maximal matching in an unweighted graph. In this paper we present the first self-stabilizing algorithm for the weighted matching problem. We show that the algorithm computes a $\frac{1}{2}$-approximation to the optimal solution. The algorithm is simple and uses only a fixed number of variables per node. Stabilization is shown under various types of daemons.

Keywords: self-stabilizing algorithms, weighted matching.

1 Introduction

Given a graph with n nodes and m edges, a matching is a set of edges in a graph such that no node is incident to more than one selected edge. In a distributed setting a matching can model a situation where each node must choose exactly one of its neighbors for communication. The associated optimization problem then becomes to choose a matching of maximum cardinality.

The matching problem lends itself well to distributed solutions since progress towards a maximal solution can be made by selecting any edge in any order and adding it to the current matching just as long as the selected edge is not incident to an edge already included in the matching. It is well known that any maximal matching (i.e. where no more edges can be added) is also a $\frac{1}{2}$-approximation to the maximum matching.

Figure 1a shows an example of a non-maximal matching, while Figure 1b illustrates a matching that is maximal, but not maximum. A maximum matching is shown in Figure 1c. Finally, Figure 1d shows a set of edges that is not a matching, since they are incident on the same node.

Previous work on the matching problem has resulted in several self-stabilizing algorithms. Hsu and Huang [7] gave the first such algorithm and proved a bound of $O(n^3)$ on the number of moves assuming a sequential model under an adversarial daemon. This analysis was later improved to $O(n^2)$ by Tel [9] and finally to $O(m)$ by Hedetniemi et al. [6].

Gradinariu and Johnen [5] employed a method of randomization to assign an ID to each node that is unique within distance 2, and used this to run Hsu and

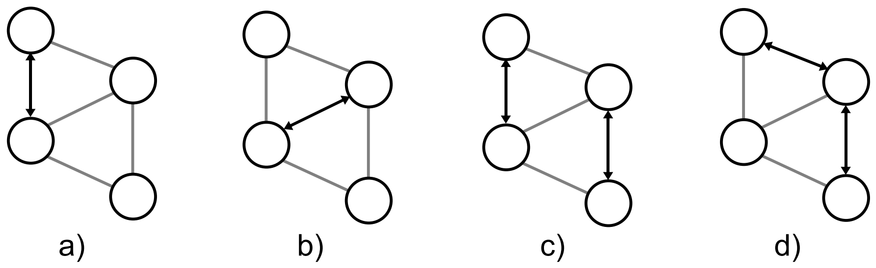

Fig. 1.

Huang's algorithm under an adversarial daemon. They show only finite stabilization time however. Using the same technique of randomized local symmetry breaking, Chattopadhyay et al. [1] later provided a maximal matching with $O(n)$ round complexity, but assuming the weaker fair distributed daemon.

In [4] Goddard et al. describe a synchronous version of Hsu and Huangs algorithm and show that it stabilizes in $O(n)$ time steps. Very recently, Manne et al. [8] presented an algorithm that stabilizes in $O(m)$ time steps in the more general distributed model. Goddard et al. [3] also gave a self-stabilizing algorithm for computing a maximal strong matching, however with exponential stabilization time.

In the current paper we present the first self-stabilizing algorithm for computing a *weighted* matching. As opposed to the unweighted case, we now assume an edge weighted graph and the objective is to compute a matching such that the sum of the weights of the edges in the matching is as large as possible. Again considering an example where every node in a network must choose exactly one neighbor to communicate with, the weighted matching problem can be used to model networks where not all lines of communication are equally desirable. For example, in a wireless system the weight assigned to a link might reflect the quality or bandwidth of the link. In this setting selecting a matching of maximum weight would ensure maximum information flow in the network.

We show that the presented algorithm computes a $\frac{1}{2}$-approximation to the optimal solution. As to speed of convergence, we show that the algorithm stabilizes in $O(n)$ rounds in the distributed model under a fair daemon. This implies that it also stabilizes in $O(n)$ time steps in the synchronous model which is the same as the algorithm by Goddard et al. [4] for the unweighted case. For an adversarial daemon we show that the algorithm stabilizes in a finite (exponential) number of steps both for the sequential and the distributed daemon.

It should be noted that computing a $\frac{1}{2}$-approximation for the weighted matching problem is an inherently more difficult problem than for the unweighted case. The reason for this is that in the weighted case any maximal solution can be arbitrarily bad compared to the optimal solution.

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 assume. In Section 3 we present our new algorithm while we show correctness and speed of stabilization in Section 4 before concluding in Section 5.

2 The Self-stabilizing Paradigm

A self-stabilizing algorithm is a distributed system where each node is an independent entity with knowledge only of itself and its neighbors. Unlike many other distributed systems, a self-stabilizing algorithm is not initialized. Instead the algorithm has to be able to reach a stable, or terminal, configuration regardless of its starting configuration. In this sense, a self-stabilizing algorithm is very resistant to transient faults and can also handle a dynamic environment where the structure of the underlying graph is changing.

A self-stabilizing algorithm is comprised of a set of rules, where each rule is made up of a predicate and a move. If a predicate evaluates to true for a node, the node is referred to as privileged and only then may it execute the corresponding move. An algorithm is stable if there are no privileged nodes in the graph.

The general distributed model allows a non-empty subset of the privileged nodes to perform one move each during a time step in the execution of the algorithm. The synchronous model is a sub-variant of this model, and requires that every privileged node executes a move in each step. Another sub-variant of the distributed model, the often-used sequential model, allows only a single node to make a move during each step.

If more than one node in the graph is privileged at that start of a particular time step there are several models that govern which nodes will perform a move. Common to all of these models is the notion of a daemon that makes the actual choice as to which subset of privileged nodes are selected for a move. We distinguish between a fair and an adversarial daemon. Under a fair daemon a privileged node will never have to wait an infinite number of time steps before it is permitted to make a move, while an adversarial daemon can select any privileged node for a move.

With the adversarial daemon, the moves complexity of an algorithm is measured in time steps for the distributed and synchronous model and in single moves for the sequential model. Under a fair daemon we measure moves complexity in rounds, where one round is a minimal sequence of time steps during which every node privileged at the start of the round has either made a move or become non-privileged. For further reading about self-stabilization algorithms see [2].

3 The Algorithm

In the following we present and motivate our self-stabilizing weighted matching algorithm. Note that at this stage we do not make any assumptions as to which model or which daemon the algorithm will execute under.

3.1 The Graph Model

Given an undirected weighted graph $G = (V, E)$ where $|V| = n$ and $|E| = m$. We denote $w_{v,u} > 0$ as the weight of the edge $(v, u) \in E$. Furthermore, we assume that every node $v \in V$ has a unique, comparable, ID, denoted by ID_v. To ensure uniqueness of the edges we define a function $w(v, u) = (w_{v,u}, \max\{ID_v, ID_u\}, \min\{ID_v, ID_u\})$. This triplet, referred to as the *effective weight* of an edge, is used to define a total ordering of the edges. That is, the edges are first ranked by their weight and if two edges have the same weight they are ranked by the highest ID of the two nodes incident on that edge. If these are also equal the edges are ranked by the lowest ID of the two incident nodes. In this way, any node can compute the effective weight of all edges incident on it, and no two edges in the graph will ever be considered to be of equal weight (note that $w(v, null)$ is by definition 0). Thus the heaviest edge in any subset of E is the edge with greatest effective weight. For ease of presentation we will not distinguish between weight and effective weight in the rest of the presentation but merely assume that the weight of each edge is unique.

We also use the notation $N(v)$ for the neighborhood of v. That is, for a node $v \in V$, $u \in N(v) \Leftrightarrow (v, u) \in E$. We say that two edges are incident if they share at least one common end point. Note that an edge is incident to itself.

3.2 Variables

Every node $v \in V$ has two variables, m_v and h_v. The intent being that in a stable configuration m_v should point to the neighbor of v that v is matched with, while h_v is the weight of the edge (v, m_v). If the node v is not matched then m_v should be set to *null* and h_v to 0. During the execution of the algorithm a node v will use m_v and h_v to propose a matching with one of its neighbors by pointing to it. However, two neighbors v and w are only considered to be matched with each other if both $m_v = w$ and $m_w = v$. A matching M consists of all the matched edges in the graph, while the weight of M is the sum of the weights of the edges in M.

The algorithm also makes use of the set $N'(v)$ defined as follows: $N'(v)$ is a subset of $N(v)$ such that $u \in N'(v) \Leftrightarrow (u \in N(v) \wedge w(v, u) \geq h_u)$. That is, $N'(v)$ consists of all the neighbors of v that could achieve a match of equal or higher weight than their current one if they were to match with v. Note that $N'(v)$ is not a variable, but rather a set that v can compute during the execution of the algorithm.

3.3 The Algorithm

In this section we present our algorithm. It is quite simple, consisting of one function and one rule:

```
BestMatch(v)
    return $u : \max_{u \in N'(v) \cup \{null\}} w(v,u)$

SetMatch
    if $m_v \neq BestMatch(v) \bigvee h_v \neq w(v, m_v)$
    then
        $m_v = BestMatch(v)$
        $h_v = w(v, m_v)$
```

Algorithm 1.

The function **BestMatch(v)** returns the neighbor $u \in N'(u)$ such that $w(v,u)$ is maximal among all nodes in $N'(v)$, while the rule **SetMatch** sets m_v to point to the node returned by **BestMatch(v)** and also updates the value of h_v accordingly. Thus a node will always strive to match with the node in $N'(v)$ so that the resulting matched edge has maximal weight.

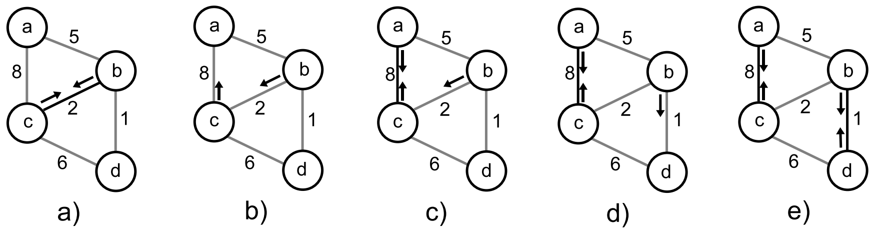

Fig. 2.

Figure 2 shows a possible execution of Algorithm 1. Starting from the configuration in Figure 2a, we observe that nodes b and c are matched. However, for both node b and c there exists an unmatched neighbor such that both the edge (a,b) and the edge (a,c) has a greater weight than (b,c). Assume now that c makes a move first and points to a (Figure 2b). Since (a,c) is the heaviest edge incident on a, the node a can now execute a move and point to c (Figure 2c) (note that the moves executed up till this point could have been done in any order). At this point b can no longer point to c, and since a is matched to c, it is left with d as its only unmatched neighbor. Thus b points to d (Figure 2d), and d, having now become privileged, points back (Figure 2e). Thus we have two pairs of matched nodes in the graph.

4 Proof of Correctness

In the following we will first show that when **Algorithm 1** has reached a stable configuration it also defines a matching that is a $\frac{1}{2}$-approximation to the maximum weight matching. We will then bound the number of steps the algorithm needs to stabilize both for the fair and for the adversarial distributed daemon. Note that the fair daemon is a subset of the adversarial one, thus any result for the latter also applies to the former.

4.1 Correct Stabilization

We now show that the algorithm, once stable, has found a $\frac{1}{2}$-approximation to the maximum weight matching problem. To do so, we first need the following observation which follows from the **BestMatch** function and from the predicate of **SetMatch**.

Observation 1. *In a stable configuration* $m_v \in N'(v) \cup \{null\}$ *and* $h_v = w(v, m_v)$ *for every node* $v \in V$.

The next step is to show that when stable, there is consensus in the graph as to which pairs of nodes are matched.

Lemma 1. *In a stable configuration* $m_v = u \Leftrightarrow m_u = v$ *for every edge* $(v, u) \in E$.

Proof. We note from Observation 1 that in a stable configuration $m_v \in \{N'(v), null\}$, $m_u \in \{N'(u), null\}$, $h_v = w(v, m_v)$, and $h_u = w(u, m_u)$. The rest of the proof is by contradiction.

We first show that $m_v = u \Rightarrow m_u = v$. Assume that $m_v = u$ while $m_u = y$ where $y \neq v$. Depending on the weights of (v, u) and (u, y) we have the following two possibilities: *i)* $w(v, u) > w(u, y)$, in which case u would be privileged, since v would be a better match for u than y. *ii)* $w(v, u) < w(u, y)$ in which case v is not a better match for u, thus $u \notin N'(v)$ and v is privileged. In either case, the algorithm is not stable. Using the same argument it also follows that $m_v = u \Leftarrow m_u = v$, thus proving the lemma. □

In the following we refer to a *stable matching* as a set of edges M in a stable configuration such that for every edge $(x, y) \in E$, $(x, y) \in M \Leftrightarrow m_x = y$ and $m_y = x$. The next lemma shows that we cannot have a stable configuration where two adjacent nodes each have a matching of lower weight than that of the edge joining them.

Lemma 2. *In a stable configuration, for every edge* $(v, u) \in E$ *we have* $w(v, u) \leq \max(h_v, h_u)$.

Proof. From Observation 1 we know that in a stable configuration $h_x = w(x, m_x)$ for any node $x \in V$. The rest of the proof is by contradiction.

Assume that there exists an edge $(v, u) \in E$ in a stable configuration such that $w(v, u) > \max(h_v, h_u)$. Then since $w(v, u) > h_u$ we have $u \in N'(v)$. Also, since the current configuration is stable **BestMatch**(v) returns a node $x \in N'(v)$ such that $m_v = x$ and $w(v, x) > w(v, u)$. But since $h_v < w(v, u)$ it follows that $w(v, m_v) < w(v, u)$ and we must have $m_v \neq$ **BestMatch**(v) contradicting that the solution is stable. The same argument can also be used to show that u is privileged. □

Corollary 1. *Let M be any stable matching given by Algorithm 1. Then every edge $(v, u) \in E$ is incident on at least one edge $(x, y) \in M$ such that $w(v, u) \leq w(x, y)$.*

Proof. Let (v, u) be any edge in E in a stable configuration. From Lemma 2 we know that $w(v, u) \leq \max(h_v, h_u)$. Assume (without loss of generality) that $h_u = \max(h_v, h_u)$. Then since $w(v, u) > 0$ we must also have $h_u > 0$ and it follows that u is matched in M. From Observation 1 we know that $h_u = w(u, m_u)$ implying that $w(v, u) \leq w(u, m_u)$ and the result follows. □

This enables us to show the main result of this section.

Theorem 1. *Any stable matching M given by Algorithm 1 is a $\frac{1}{2}$-approximation to the maximum weighted matching problem.*

Proof. Let M^* be a maximum weighted matching for G. From Corollary 1 we know that it is possible to associate every edge $(v, u) \in M^*$ with exactly one incident edge $(x, y) \in M$ such that $w(v, u) \leq w(x, y)$. Since at most two edges from M^* can be associated with each edge of M we have that $2w(M) \geq w(M^*)$ and the result follows. □

We note that it is fairly straight forward to show that the matching produced by Algorithm 1 is in fact exactly the same matching as the sequential greedy algorithm would give.

4.2 Convergence

We now show that Algorithm 1 stabilizes from any given starting configuration. Specifically, we will be looking at the rate of convergence first using the distributed adversarial model and then using the distributed fair model. The distributed model is the most general model, where a non-empty subset of the privileged nodes makes a move during each time step. Note that if at each time step only one node is allowed to make a move, this model is identical to the sequential model.

4.3 The Distributed Adversarial Model

We proceed to bound the number of time steps needed before Algorithm 1 stabilizes under an adversarial daemon. The proof is based on counting the number of moves needed before at least one node v stabilizes permanently. We then repeat the argument recursively for the remaining set of nodes $A = V - \{v\}$ obtaining our desired bound. First we show that if parts of the graph has stabilized permanently then there exists at least one node that can at most make two more moves.

Lemma 3. *Given a set $A \subseteq V$ where the nodes in A are the only nodes in the graph permitted to move. Then there exists at least one node in A that can make at most two moves.*

Proof. Let (v, u) be the heaviest edge in the set $\{(x, y) \, \forall x, y \in A : (x, y) \in E\} \cup \{(a, b) \, \forall a \in A, \; b \in V \backslash A : (a, b) \in E \wedge w(a, b) \geq h_b\}$. We assume without loss of generality that at least $v \in A$.

If $u \in V \backslash A$ then $w(v, u) \geq h_u$ and the only move v can make is one that sets $m_v = u$ (provided that this is not already the case). Since there does not exist any edge incident on v in A that is heavier than (v, u) and since u cannot make a move, it follows that v will not move again.

If $u \in A$ then there are two possibilities: *i)* $h_u \leq w(v, u)$ or *ii)* $h_u > w(v, u)$. In the first case it follows that the only move v can make is to match with u before becoming permanently stable (again assuming that only nodes in A may execute moves).

In the second case, u will need to make one move to correct its h-value (which is incorrect). During this time step v can also make a move. Following this move $h_u \leq w(v, u)$, and as in case *i)*, the only move v can make is to match to u, again becoming permanently stable. Thus v has executed at most two moves not counting any moves executed before u's first move. If v executes any move before u, we can simply switch the roles of v and u and repeat the argument.

In either of the above cases we see that there has to exist at least one node in A that can move at most twice. □

Based on Lemma 3 we can now give a recursive formula for the number of moves that Algorithm 1 can execute on the remaining nodes that have not yet stabilized permanently.

Lemma 4. *Let $A \subseteq V$ be a set where $|A| = k$ and let $t(k)$ be the maximum number of moves needed for A to stabilize given that only nodes in A are permitted to move. Then $t(k) \leq 3 \cdot t(k-1) + 2$.*

Proof. Recall from Lemma 3 that there exists at least one node $v \in A$ that can execute at most two moves. From the premise of the lemma we know that at most $t(k-1)$ moves can be made by the nodes in $A \backslash \{v\}$ before v makes its first move. Following this, another $t(k-1)$ moves can be made before v's second move. And finally at most $t(k-1)$ subsequent moves can be made as a result of v's second and final move. Thus at most $3 \cdot t(k-1) + 2$ moves can be made in a set of size k. □

Theorem 2. *Algorithm 1 stabilizes after $O(3^n)$ time steps under the distributed adversarial model.*

Proof. From Lemma 4 we know that the time needed for a subset of nodes of size k to become stable is $t(k) \leq 3 \cdot t(k-1) + 2$. Since $t(1) = 1$ it follows that the maximum number of moves needed to ensure stabilization is $t(n) \leq 2 \cdot 3^{n-1} - 1$. Thus the number of time steps used by the algorithm is $O(3^n)$. □

For the sequential adversarial model the bound from Theorem 2 can be improved to $O(2^n)$. This follows by noting that for any unstable configuration there exists at least one node that can make at most one more move before becoming permanently stable. We omit the details.

4.4 The Distributed Fair Model

We now look at the convergence rate of Algorithm 1 assuming a distributed model under a fair daemon, and prove that the algorithm stabilizes after at most $2 \cdot |M| + 1$ rounds where M is the final matching found by the algorithm. We remind the reader that in the distributed fair model, complexity is measured in rounds, where one round is a minimum period of time during which every node that was privileged at the start of the round has either made at least one move or at some point become non-privileged.

We first note that since the distributed fair model is a subset of the distributed adversarial model it follows from Theorem 2 that Algorithm 1 will eventually stabilize with a matching M that is a $\frac{1}{2}$-approximation to the maximum weighted matching problem. Thus it is meaningful to refer to the resulting matching M. We now proceed to bound the number of rounds before Algorithm 1 stabilizes.

Lemma 5. *After at most one round $m_v \in N(v) \cup \{null\}$ and $h_v = w(v, m_v)$ for every $v \in V$.*

Proof. Recall from the predicate of **SetMatch** that a node v is privileged if its m-value is incorrect or if $h_v \neq w(v, m_v)$. In either case **SetMatch** will set m_v to some node $u \in N'(v) \cup \{null\}$ and h_v to $w(v, m_v)$. Since $N'(v) \subseteq N(v)$ the result follows. □

Note that one cannot guarantee that $m_v \in N'(v) \cup \{null\}$ after the first round as $N'(v)$ might change after v has made its move.

Lemma 6. *After at most two rounds, the heaviest edge $(v, u) \in E$ is part of M. Furthermore, the algorithm will never cause (v, u) to leave M.*

Proof. From Lemma 5 we know that after the first round every node has a correct h-value and $m_v \in N(v) \cup \{null\}$. If (v, u) is the heaviest edge in G it follows that $h_v \leq w(v, u)$ and $h_u \leq w(v, u)$, implying that $u \in N'(v)$ and $v \in N'(u)$. Thus if $m_v \neq u$ then v is privileged and if $m_u \neq v$ then u is privileged. In either case, at most one more round is needed before v and u are matched. This will happen since neither of the two nodes has a neighbor that can give a better matching than the other node.

Since there does not exist any edge in the graph with weight greater than (v, u) neither v nor u will become privileged again, and thus the edge (v, u) will never leave the matching. □

We can now give the final bound on the number of rounds needed before Algorithm 1 stabilizes.

Theorem 3. *Algorithm 1 converges after at most $2 \cdot |M| + 1$ rounds.*

Proof. Let $e_1, e_2, \ldots, e_{|M|}$ be the edges in M sorted in descending order. We show by induction that $e_1, e_2, \ldots, e_i$ are all part of the matching after at most $2i$ rounds, and that they will not leave M in subsequent rounds.

The base case is covered by Lemma 6 and it follows that e_1 must be the heaviest edge in E.

For the induction step assume that the algorithm has run for at most $2 \cdot (i-1)$ rounds and that the edges $M_{i-1} = \{e_1, e_2, \ldots, e_{i-1}\}$ have been permanently added to M. It follows that we do not need to consider any edge incident on M_{i-1} for future inclusion in M.

Let (v, u) be the heaviest edge not incident on M_{i-1}. Then by the same argument as in the proof of Lemma 6 it follows that within the next two rounds (v, u) will be permanently added to M. Since no edge of weight greater than $w(v, u)$ will be added to M in subsequent time steps it follows that $e_i = (v, u)$.

From the above is follows that at most $2 \cdot |M|$ rounds are needed to find the matching. However, since some nodes may not be part of the matching, one more round may be needed for these nodes to right any incorrect variables they may have. Thus the algorithm requires at most $2 \cdot |M| + 1$ rounds. □

It should be noted that the size of a matching in a graph $G = (V, E)$ cannot exceed $|V|/2$. Since this would imply that every node is matched, the number of rounds needed for the algorithm to stabilize in this case is at most $2 \cdot |M| = |V|$, not $2 \cdot |M| + 1$.

As was noted in Section 2, both the synchronous and sequential fair models are sub variants of the distributed fair model. Thus the bound from Theorem 3 also holds for either of these models.

5 Conclusion

We have presented the first self-stabilizing algorithm for computing a $\frac{1}{2}$- approximation for the maximum weighted matching problem. In addition to being short and simple, the complexity of the algorithm is linear over the number of nodes in the graph when using a distributed fair daemon. Furthermore the algorithm requires only two variables per node.

It is worth noting that while we in Section 3.1 require that all IDs are unique, this is in fact not needed. The algorithm requires only that every ID is unique within distance 2. That is, no node can have two or more neighbors with the same ID. On the same note, we do not need to create a global ordering of the edges in the graph. While a global ordering was used to make the proofs more understandable, a local ordering is sufficient for the algorithm.

One common method for improving the approximation ratio of a matching is by the use of augmenting paths. An augmenting path is a path such that exactly every other edge in the path is part of the current matching. The length of an augmenting path is the number of unmatched edges in it. It is well known that if a matching does not contain an augmenting path of length i then the matching is a $\frac{i}{i+1}$-approximation. Thus it would be of interest to see if it is possible to design self-stabilizing algorithms that can detect and correct augmenting paths of length larger than $i = 1$ as is done in the current paper, while at the same time limiting the number of variables and stabilization time. One possible way of doing this could be to use the same kind of augmentations as is used in the sequential linear time algorithm by Vinkelmeier and Hougardy [10] to produce a solution with approximation ratio arbitrarily close to 2/3.

References

1. Chattopadhyay, S., Higham, L., Seyffarth, K.: Dynamic and self-stabilizing distributed matching. In: PODC 2002. Proceedings of the twenty-first annual symposium on Principles of distributed computing, pp. 290–297. ACM Press, New York (2002)
2. Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)
3. Goddard, W., Hedetniemi, S.T., Jacobs, D.P., Srimani, P.K.: Self-stabilizing distributed algorithm for strong matching in a system graph. In: HiPC 2003. LNCS (LNAI), vol. 2913, pp. 66–73. Springer, Heidelberg (2003)
4. Goddard, W., Hedetniemi, S.T., Jacobs, D.P., Srimani, P.K.: Self-stabilizing protocols for maximal matching and maximal independent sets for ad hoc networks. In: IPDPS, p. 162 (2003)
5. Gradinariu, M., Johnen, C.: Self-stabilizing neighborhood unique naming under unfair scheduler. In: Sakellariou, R., Keane, J.A., Gurd, J.R., Freeman, L. (eds.) Euro-Par 2001. LNCS, vol. 2150, Springer, Heidelberg (2001)
6. Hedetniemi, S.T., Jacobs, D.P., Srimani, P.K.: Maximal matching stabilizes in time o(m). Inf. Process. Lett. 80(5), 221–223 (2001)
7. Hsu, S.-C., Huang, S.-T.: A self-stabilizing algorithm for maximal matching. Inf. Process. Lett. 43(2), 77–81 (1992)
8. Manne, F., Mjelde, M., Pilard, L., Tixeuil, S.: A new self-stabilizing maximal matching algorithm. In: Sirocco 2007. Proceedings of the 14^{th} International Colloquium on Structural Information and Communication Complexity, pp. 96–108. Springer, Heidelberg (2007)
9. Tel, G.: Maximal matching stabilizes in quadratic time. Inf. Process. Lett. 49(6), 271–272 (1994)
10. Vinkemeier, D.E.D., Hougardy, S.: A linear-time approximation algorithm for weighted matchings in graphs. ACM Trans. Algorithms 1(1), 107–122 (2005)