A Self-stabilizing $\frac{2}{3}$-Approximation Algorithm for the Maximum Matching Problem

Fredrik Manne[1], Morten Mjelde[1], Laurence Pilard[2], and Sébastien Tixeuil[3,⋆]

[1] University of Bergen, Norway
{fredrikm,mortenm}@ii.uib.no
[2] University of Franche Comté, France
laurence.pilard@iut-bm.univ-fcomte.fr
[3] LIP6 & INRIA Grand Large, Université Pierre et Marie Curie - Paris 6, France
tixeuil@lri.fr

Abstract. The matching problem asks for a large set of disjoint edges in a graph. It is a problem that has received considerable attention in both the sequential and self-stabilizing literature. Previous work has resulted in self-stabilizing algorithms for computing a maximal ($\frac{1}{2}$-approximation) matching in a general graph, as well as computing a $\frac{2}{3}$-approximation on more specific graph types. In the following we present the first self-stabilizing algorithm for finding a $\frac{2}{3}$-approximation to the maximum matching problem in a general graph. We show that our new algorithm stabilizes in at most exponential time under a distributed adversarial daemon, and $O(n^2)$ rounds under a distributed fair daemon, where n is the number of nodes in the graph.

Keywords: Self-stabilizing algorithm, $\frac{2}{3}$-Approximation, Maximum matching.

1 Introduction

A *matching* in a graph $G = (V, E)$ is a subset M of E such that no pair of edges in M have common endpoints. We say that two nodes v and w are matched if the edge (v, w) is in M. A matching M is *maximal* if no proper superset of M is also a matching. A matching M is *maximum* if there does not exists any matching with cardinality larger than $|M|$. While there exists sequential algorithms for computing a maximum matching in polynomial time, the complexity of such algorithms renders them impractical in many settings when applied to large graphs. Thus, approximation algorithms are often used to rapidly provide matchings that are within an acceptable margin of error. A maximal matching can be computed in linear time over the size of the graph, and it is well known that this results in a $\frac{1}{2}$-approximation to the maximum matching. In order to compute matchings with approximation ratios better than $\frac{1}{2}$, *augmenting paths*

⋆ Support for this work was given by the Aurora program for collaboration between France and Norway.

are often used. An augmenting path is a path in the graph, starting and ending in an unmatched node and where every other edge is either unmatched or matched, i.e. for each consecutive pair of edges exactly one of them must belong to the matching. Once an augmenting path p has been identified one can increase the size of M by performing an augmenting step. This consists of removing each matched edge of p from M and including every unmatched edge of p in M. This way the cardinality of the matching is increased by one. Hopcroft and Karp [12] show that given a graph $G = (V, E)$ and a matching $M \subseteq E$ then if there does not exist an augmenting path of length at most three in G, then M is a $\frac{2}{3}$-approximation to the maximum matching.

The matching problem is often used to model several real world situations. Examples include the problem of assigning tasks to workers or creating pairs of entities. The latter lends itself well to a distributed network, since processes in the network may need to choose exactly one neighbor to communicate with.

In this paper we use augmenting paths and present a self-stabilizing algorithm that computes a $\frac{2}{3}$-approximation to the maximum matching problem in a general, unweighted graph. Our algorithm is based on using an existing maximal matching, and then identifying augmenting paths of length three. These are then used to improve the cardinality of the matching.

1.1 Self-stabilizing Algorithms

Self-stabilizing algorithms [3,4] are distributed algorithms that permit forward failure recovery by means of an attractive property: starting from any arbitrary initial state, the system autonomously resumes correct behavior within finite time. Self-stabilization allows failure detection to be bypassed, yet does not make any assumptions about the nature or the span of those failures. Central to the theory of self-stabilization is the notion of *daemon*, an abstraction for the scheduling of nodes in the system to execute their local code. A daemon is often viewed as an adversary to the algorithm that tries to prevent stabilization by scheduling the worst possible nodes for execution. The weakest possible requirement is that the daemon is *proper*, i.e. only nodes whose scheduling would change the system state are actually scheduled (these nodes are *privileged*). Variants of daemons can be defined along two axis: *(i)* a daemon may be *sequential* (meaning that no two privileged nodes may be selected by the daemon simultaneously) or *distributed* (in which case any number of privileged nodes may be selected at the same time), and *(ii)* a daemon may also be *fair* (which ensures that every privileged node will be allowed to move eventually) or *adversarial* (meaning that a privileged node may have to wait indefinitely, yet always scheduling *some* privileged node for execution). Intuitively, distributed is a more general property than sequential, and adversarial is a more general property than fair. Thus among these daemons, the most general is the distributed adversarial, and the least general is the sequential fair daemon. As a result, an algorithm that tolerates the most general adversary also tolerates the least general one, but the converse is not true.

Time complexity is measured differently depending on the daemon used: for any fair daemon time complexity is measured in *rounds*, where a round is the smallest sub-sequence of an execution in which every node privileged 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. For the adversarial sequential daemon, complexity is measured in single node moves, while for the adversarial distributed daemon it is measured in time steps, where a time step is one step in the execution during which at least one privileged node executes one move.

When no nodes in the graph are privileged, we say that the algorithm is *stable*, or has reached a *stable configuration.*

1.2 Related Work

The first self-stabilizing algorithm for computing a maximal matching was given by Hsu and Huang [13]. The authors showed a stabilization time of $O(n^3)$ moves under a sequential adversarial daemon. This analysis was later improved to $O(n^2)$ by Tel [15] and to $O(m)$ by Hedetniemi et al. [11], where m is the number of edges in the graph. The algorithm assumes an anonymous graph and the sequential daemon is used to break symmetry. By means of randomization Gradinariu and Johnen [9] gave a method for assigning identifiers that are unique within distance two. This was then used to transform the algorithm by Hsu and Huang so that it stabilizes under a distributed adversarial daemon, albeit with an unbounded stabilization time.

Goddard et al. [6] gave a synchronous variant of Hsu and Huangs algorithm and showed that it stabilizes in $O(n)$ rounds. While not explicitly proved in the paper, it can be shown that this algorithm stabilizes in $\theta(n^2)$ time steps under an adversarial distributed daemon. Gradinariu and Tixeuil [10] provide a general scheme to transform an algorithm written for the sequential adversarial daemon into an algorithm that can cope with the distributed adversarial daemon. Using this scheme with the Hsu and Huang algorithm yields a time step complexity of $O(\Delta \cdot m)$, where Δ denotes the maximum degree of the graph. Manne et al. [14] later gave an algorithm for finding a maximal matching that stabilizes in $O(m)$ time step under the distributed adversarial daemon, and $O(n)$ rounds when using the distributed fair daemon. The aforementioned protocols of [6,10,14] assume that the nodes are provided with unique identifiers (either globally, or within a certain distance), as [14] points out that deterministic protocols require symmetry breaking to deal with the adversarial daemon.

When it comes to improving the $\frac{1}{2}$-approximation induced by the maximal matching property, only a few works investigate this issue in a self-stabilizing setting. Ghosh et al. [5] and Blair and Manne [1] presented a framework that can be used for computing a maximum matching in a tree under a distributed adversarial daemon using $O(n^2)$ moves, while Goddard et al. [8] gave a self-stabilizing algorithm for computing a $\frac{2}{3}$-approximation in anonymous rings of length not divisible by three using $O(n^4)$ moves, under a sequential adversarial daemon. The polynomial complexity results mainly from the fact that only strongly constrained topologies are investigated.

The case of general graphs is more intricate and is the topic of this paper. It is possible to compute a $\frac{2}{3}$-approximation (or even an optimal solution) for the maximum matching problem by collecting the entire graph topology on each node using a self-stabilizing topology update protocol, and then run a deterministic sequential algorithm on each node. This would yield a self-stabilizing algorithm for the matching problem, but at the expense of having to duplicate the system graph on each node. This approach is not very practical in most settings, due to its considerable memory usage.

As far as feasibility is concerned, it would be possible to use a generic scheme such as [7,2] that prevents nodes at distance k or less of a particular node u to execute code until further notice from u. Such a scheme would permit to devise a protocol that essentially tries to find and then to integrate augmenting paths starting at a node u. Unfortunately, both schemes suffer from severe drawbacks for this purpose. First, both [7] and [2] make use of a large amount of memory at each node (typically, an exponential number of states with respect to k). Second, the complexity of a $\frac{2}{3}$-approximation scheme using [7] would be unbounded. Third, a scheme based on [2] would require operating under a fair daemon, and may not stabilize under an adversarial one.

1.3 Our Contribution

In this paper we present the first self-stabilizing algorithm for computing a $\frac{2}{3}$-approximation to the maximum matching in a general, non-anonymous graph, that performs under any daemon. Complexity-wise, we show that our algorithm stabilizes in $O(2^{n+2} \cdot \Delta \cdot n)$ time steps under the distributed adversarial daemon, and in $O(n^2)$ rounds under the distributed fair daemon. The memory used at each node by our protocol is low: we use three pointers to neighbors and one boolean variables. The rest of the paper is organized as follows. The algorithm is presented in Section 2. In Section 3 we show the correctness of the algorithm, while the stabilization time for the algorithm is shown in Section 4. Finally, we conclude in Section 5.

2 The Algorithm

In this section we present our new algorithm. The algorithm assumes that there exists an underlying maximal matching algorithm, which has reached a stable configuration. In Section 4.3 we will explain how the algorithm works when this algorithm is not in a stable configuration. The new algorithm functions by identifying augmenting paths of length three, and then rearranging the matching accordingly. This is done in several steps. First every pair of matched nodes v, w will try to find unmatched neighbors to which they can rematch. Then one of v and w will first attempt to match with one of its candidates. Only when the first node succeeds, will the second node also attempt to match with one of its candidates. If this also succeeds the rematching is considered complete. The algorithm will stabilize when there are no such augmenting paths left.

2.1 Predicates and Variables

Given an undirected graph $G = (V, E)$ where each node v has a unique identifier. We assume that these can be ordered, and in the following we do not distinguish between a node and its identifier. By definition $v < null$ for every node $v \in V$.

The set of neighbors of v in G is denoted by $N(v)$. In the following, we refer to M' as the set of edges in the underlying maximal matching. If v is matched in M', then m_v denotes the node that v is matched with in M', i.e. $(v, m_v) \in M'$. Note that if v is unmatched in M' then $m_v = null$. For a set of nodes A, we define $\mu(A)$ and $\sigma(A)$ as the set of matched and unmatched nodes in A, respectively, in the maximal matching M'. Since we assume that the underlying maximal matching is stable, a nodes membership in $\mu(V)$ or $\sigma(V)$ will not change, and each node v can use the value of m_v to determine which set it belongs to.

In order to facilitate the rematching, each node $v \in V$ maintains three pointers and one boolean variable. The pointer p_v refers to a neighbor of v that v is trying to (re)match with. If $p_v = null$ then the matching of v has not changed from the maximal matching (we define $p_{null} = null$). Thus two neighboring nodes v, w are matched if and only if either $p_v = w$ and $p_w = v$, or if $p_v = null$, $p_w = null$ and $(v, w) \in M'$.

For a node $v \in \mu(V)$, the pointers α_v and β_v refer to two nodes in $\sigma(N(v))$ that are candidates for a possible rematching with v. Also, s_v is a boolean variable that indicates if v has performed a successful rematching or not.

2.2 Rules and Functions

The following section gives the rules and functions of the algorithm. Each rule is executed on a node $v \in V$. We divide the rules into two sets, one for nodes in $\sigma(V)$ and one for nodes in $\mu(V)$. If more than one rule is privileged for a node in $\mu(V)$, the rules are executed in the order presented here. For a set of nodes A, *Unique(A)* returns the number of unique elements in the set[1], and *Lowest(A)* returns the node in A with the lowest identifier, or *null* if $A = \emptyset$.

```
SingleNode
    if   $(p_v = null \wedge Lowest\{w \in N(v) \mid p_w = v\} \neq null) \vee$
         $p_v \notin (\mu(N(v)) \cup \{null\}) \vee (p_v \neq null \wedge p_{p_v} \neq v)$
    then $p_v := Lowest\{w \in N(v) \mid p_w = v\}$
```

Algorithm 1 - Rule for nodes in $\sigma(V)$

Motivation. We now give a brief motivation for each rule in Algorithm 1.

The purpose of the *SingleNode* rule is to ensure that a node $v \in \sigma(V)$ is pointing to a neighbor in $\mu(N(v))$ that points back to v. In doing so, v and p_v will be matched. If there exists more than one candidate, the rule will select the one with the smallest identifier. If no node in $\mu(N(v))$ points to v, the rule ensures that v points to *null*, thereby informing v's neighbors that v is unmatched.

[1] Note that *Unique(A)* $= |A|$. However for the sake of clarity we use *Unique(A)*.

```
Update
    if  $(\alpha_v > \beta_v) \vee (\alpha_v, \beta_v \notin \sigma(N(v)) \cup \{null\}) \vee$
        $(\alpha_v = \beta_v \wedge \alpha_v \neq null) \vee p_v \notin (\sigma(N(v)) \cup \{null\}) \vee$
        $((\alpha_v, \beta_v) \neq BestRematch(v) \wedge (p_v = null \ \vee \ p_{p_v} \notin \{v, null\}))$
    then $(\alpha_v, \beta_v) := BestRematch(v)$
        $(p_v, s_v) := (null, false)$

MatchFirst
    if $(AskFirst(v) \neq null) \wedge (p_v \neq AskFirst(v) \vee s_v \neq (p_{p_v} = v))$
    then $p_v := AskFirst(v)$
        $s_v := (p_{p_v} = v)$

MatchSecond
    if $(AskSecond(v) \neq null) \wedge (s_{m_v} = true) \wedge (p_v \neq AskSecond(v))$
    then $p_v := AskSecond(v)$

ResetMatch
    if $(AskFirst(v) = AskSecond(v) = null) \wedge ((p_v, s_v) \neq (null, false))$
    then $(p_v, s_v) := (null, false)$
```

Algorithm 1 - Rules for nodes in $\mu(V)$.

```
BestRematch($v$)
    $a = Lowest \ \{u \in \sigma(N(v)) \wedge (p_u = null \vee p_u = v)\}$
    $b = Lowest \ \{u \in \sigma(N(v)) \setminus \{a\} \wedge (p_u = null \vee p_u = v)\}$
    return $(a, b)$

AskFirst($v$)
    if $\alpha_v \neq null \wedge \alpha_{m_v} \neq null \wedge 2 \leq Unique(\{\alpha_v, \beta_v, \alpha_{m_v}, \beta_{m_v}\}) \leq 4$
        then if $\alpha_v < \alpha_{m_v} \vee (\alpha_v = \alpha_{m_v} \wedge \beta_v = null) \vee (\alpha_v = \alpha_{m_v} \wedge \beta_{m_v} \neq null \wedge v < m_v)$
            then return $\alpha_v$
    else return $null$

AskSecond($v$)
    if $AskFirst(m_v) \neq null$
        then return $Lowest(\{\alpha_v, \beta_v\} \setminus \{\alpha_{m_v}\})$
    else return $null$
```

Algorithm 1 - Functions

The *Update* rule is used to ensure that a node $v \in \mu(V)$ has α_v and β_v set to two neighbors that v can try to match with. Note that the rule is executed if any one of the current α-, β-, or p-value is not pointing to a node in $\sigma(N(v))$ or to *null*, or if the values of α and β are incorrect, relative to each other. If this is not the case, the rule is executed only if v is not already involved in a rematch attempt. The values of α_v and β_v are returned by the *BestRematch* function, which returns the two unmatched neighbors in $\sigma(N(v))$ with the smallest identifiers.

The *MatchFirst* rule is executed by a node $v \in \mu(V)$ in order to initiate a rematch attempt. The *AskFirst* function returns the neighbor of v that v should attempt to rematch with. If this succeeds, then the node m_v may become privileged for a *MatchSecond* move, which employs the *AskSecond* function in the same way that *MatchFirst* uses *AskFirst*. The *AskFirst* function has two consecutive predicates, both which must evaluate to true in order for the calling node v to become privileged for a *MatchFirst* move. The first predicate (the first *if* statement) checks that v and m_v each have at least one possible unique neighbor to rematch with. The second predicate decides whether v or m_v should initiate the rematch attempt.

If a node $v \in \mu(V)$ becomes unable to participate in a rematch attempt, it may be privileged for a *ResetMatch* move, in order to reset its p- and s-value.

Example. We now give a possible execution of Algorithm 1 under a distributed adversarial daemon. Figure 1 presents a graph, consisting of the four nodes x, v, w, and y, where $v < w$ and $x < y$. Nodes v and w are matched in the underlying maximal matching. This is shown by the double line joining them. In the figure we illustrate one node pointing to a neighbor by an arrow (the absence of an arrow means that the node in question is pointing to *null*), and if the s-value is true for a node we show this by a double border. The values of the α- and β-variables are not shown in the figure.

Figure 1a shows the initial state of the graph. We assume that at this point $(\alpha_v, \beta_v) = (x, null)$ and $(\alpha_w, \beta_w) = (x, z)$, where $z \notin N(w)$. Also note that $s_v = false$. Observe that both v and w are pointing to x, which implies that x is privileged for a *SingleNode* move. Since $\beta_w \notin N(w)$, w is privileged for an *Update* move. In Figure 1b x has executed its *SingleNode* move, and v has executed a subsequent *MatchFirst* move and set $s_v = true$. At this point, w may execute an *Update* move, while no other nodes are privileged. This move will set $(p_w, s_w) = (null, false)$ and, since w has no neighbors that are eligible candidates for a rematch attempt, $(\alpha_w, \beta_w) = (null, null)$. However, this gives *AskFirst*$(v) = null$, and v can now execute a *ResetMatch* move, which is followed by a *SingleNode* move by x. The result of these moves is shown in Figure 1c.

At this point, both v and w have, combined, at least two unique candidates for a rematching, namely x and y. Thus both nodes will execute *Update* moves, after which *AskFirst*$(w) = x$ (which implies that *AskFirst*$(v) = null$), and w may execute a *MatchFirst* move, and point to x, as seen in Figure 1d. Following this

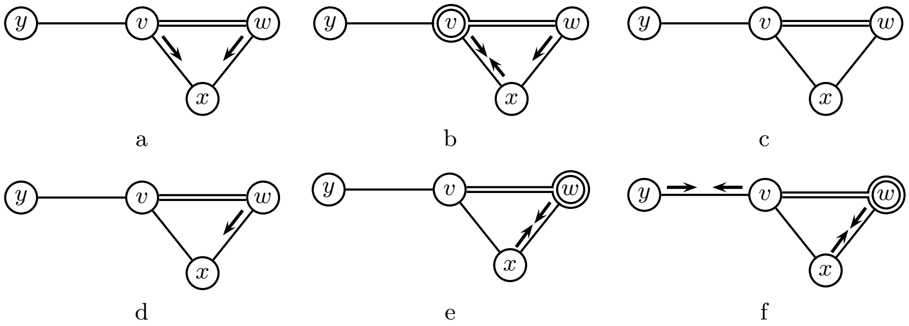

Fig. 1. Execution example of Algorithm 1

move, x executes a *SingleNode* move and points to w after which w will set $s_w = true$ through a *MatchFirst* move (Figure 1e). Since w has successfully established a rematching, v may now attempt the same by executing a *MatchSecond* move and point to y. This will cause y to point back to v (note however that v is not privileged to set $s_v = true$) (Figure 1f). At this point the system has reached a stable configuration, and the augmenting path that existed in Figure 1a has been identified and used to improve the matching.

3 Correct Stabilization

In this section we show that when Algorithm 1 is stable it has computed a $\frac{2}{3}$-approximation to the maximum cardinality matching problem. Due to page constraints we omit some of the proofs.

We first need the following definition.

Definition 1. A node $v \in \mu(V)$ is a *pioneer* if and only if *AskFirst(v)* $\neq$ *null*.

We define the short hand notation

$$R(v) \equiv \alpha_v \neq null \wedge \alpha_{m_v} \neq null \wedge 2 \leq Unique(\{\alpha_v, \beta_v, \alpha_{m_v}, \beta_{m_v}\}) \leq 4$$

Thus $R(v)$ is equal to the outcome of the first if-statement of the *AskFirst* function. $R(v) = true$ states that v and m_w each have at least one candidate for a rematch, and together they have at least two unique candidates. Note that $R(v) = R(m_v)$. We now make the following observation about Algorithm 1.

Lemma 1. *For a node $v \in \mu(V)$ in a stable configuration where $R(v) = true$ then either AskFirst$(v) \neq null$ or AskFirst$(m_v) \neq null$.*

Next we show the following connection between *AskFirst*(v) and *AskSecond*(m_v).

Lemma 2. *If $v \in \mu(V)$ then AskFirst$(v) \neq null$ if and only if AskSecond(m_v) $\neq null$.*

We now proceed to show that in a stable configuration whenever a node v has $p_v \neq null$ then we must have $p_v \in N(v)$ and $p_{p_v} = v$. To do so we look at three different cases. The first case is $v \in \sigma(V)$. For $v \in \mu(V)$ we distinguish if v is a pioneer or not.

Lemma 3. *Let $v \in \sigma(V)$ in a stable configuration. Then $p_v \neq null$ implies that $p_v \in \mu(N(v))$ and that $p_{p_v} = v$.*

To show that the equivalent of Lemma 3 also holds for $v \in \mu(V)$ we first need to show the following two intermediate results.

Lemma 4. *Let $v \in \mu(V)$. Then we cannot have $p_v \neq null$, $p_{p_v} \neq v$, and $(\alpha_v, \beta_v) \neq$ BestRematch(v) in a stable configuration.*

Corollary 1. *Let $v \in \mu(V)$. Then we cannot have $p_v \neq null$, $p_{p_v} \neq v$, and $p_v \in \{\alpha_v, \beta_v\}$ in a stable configuration.*

We can now show that the equivalent of Lemma 3 also holds for $v \in \mu(V)$.

Lemma 5. *Let $v \in \mu(V)$ in a stable configuration. If AskFirst$(v) \neq null$ then (i) $p_v \neq null$, (ii) $p_v \in \sigma(N(v))$, (iii) $p_{p_v} = v$, and (iv) $s_v = true$.*

Lemma 6. *Let $v \in \mu(V)$ in a stable configuration. If AskSecond$(v) \neq null$ then (i) $p_v \neq null$, (ii) $p_v \in \sigma(N(v))$, and (iii) $p_{p_v} = v$.*

We have now established for any node $v \in V$ that if $p_v \neq null$ then $p_{p_v} = v$. We next show that if $v \in \mu(V)$ is matched to a node other than m_v in a stable configuration, then m_v is also matched to a node other than v.

Lemma 7. *If $v \in \mu(V)$ in stable configuration then $p_v \neq null \Leftrightarrow p_{m_v} \neq null$.*

Next we show that when Algorithm 1 is stable the original matching M' and the p values define an unambiguous matching. Recall that two neighboring nodes v and w are matched if either $(v, w) \in M'$, $p_v = null$, and $p_w = null$ or if $p_v = w$ and $p_w = v$. Similarly, a node v is unmatched if $v \in \sigma(V)$ and if $p_v = null$.

Lemma 8. *In a stable configuration every node is either matched or unmatched.*

We can now finally show that a stable configuration of Algorithm 1 is a $\frac{2}{3}$-approximation to the maximum cardinality matching problem.

Theorem 1. *A stable configuration of Algorithm 1 is a $\frac{2}{3}$-approximation to the maximum matching problem.*

Proof. We first note from Lemma 8 that a stable matching is well defined, meaning that every node is either matched or unmatched. Next, from Hopcroft and Karp [12], we have that for a graph G with a matching M, if there does not exists an augmenting path of length three or less then M is a $\frac{2}{3}$-approximation to the maximum matching in G.

From the definition of an augmenting path it follows that any node in $\mu(V)$ will also be a member of the final matching. Consequently, an augmenting path in a stable configuration must both start and end with nodes from $\sigma(V)$. Due to the underlying maximal matching we know that there does not exist an augmenting path in M' of length one, i.e. two unmatched nodes cannot be neighbors. It is therefore sufficient to show that there does not exist an augmenting path x, v, w, y in a stable configuration where x and y are distinct unmatched nodes and v and w are matched.

Assume that such a path exists in a stable configuration, then $v, w \in \mu(V)$, otherwise two adjacent nodes would be in $\sigma(V)$. Since v and w are matched in the final matching then either *(i)* $p_v = w$ and $p_w = v$ or *(ii)* $p_v = p_w = null$ and $(v, w) \in M'$.

Note that in *Case (i)* $p_v \in \mu(V)$ (and similarly for p_w), which would trigger an *Update* move, contradicting that the configuration is stable.

For *Case (ii)* first note that since x and y are unmatched, $p_x = p_y = null$. Thus, if *Unique*$(\{\alpha_v, \beta_v\}) = 0$ then v is privileged for an *Update* move (and similarly for w). However, if both $\{\alpha_v, \beta_v\} \neq \emptyset$ and $\{\alpha_w, \beta_w\} \neq \emptyset$ we see from Lemma 1 that either *AskFirst*$(v) \neq null$ or *AskSecond*$(v) \neq null$. From lemmas 5 and 6 this implies that the configuration is not stable. $\square$

4 Stabilization Time

We now progress to bound the time needed for the algorithm to stabilize, both for the distributed adversarial and for the distributed fair daemon. For these analysis we assume that the underlying maximal matching is stable. We address the interaction between the maximal matching and Algorithm 1 in Section 4.3. Note that due to page constraints we omit some of the proofs.

4.1 Distributed Adversarial Daemon

In this section we bound the number of time steps needed for Algorithm 1 to stabilize with the distributed adversarial daemon. Recall that one time step is one step in the execution during which at least one node privileged at the start of the time step has executed exactly one move.

We say that a node $v \in \mu(V)$ has executed a *forced Update* move if an *Update* move was executed due to one of the following conditions: *(i)* $\alpha_v > \beta_v$, *(ii)* $\alpha_v, \beta_v \notin \sigma(N(v))$, or *(iii)* $\alpha_v = \beta_v$ while $\alpha_v \neq$ *null*. Since neither of these states can occur as a result of an executed move they must occur as a result of incorrect initial values. Thus, each node $v \in \mu(V)$ can execute at most one forced *Update* move, and this will be the first move that v executes, if it was initially privileged to do so. We now make the following observation about Algorithm 1.

Lemma 9. *Let $v \in \mu(V)$. Then AskFirst$(v) \neq$ null if and only AskSecond(v) = null.*

Lemma 10. *For every nodes $v \in \mu(V)$, if neither v nor m_v is privileged for a forced Update move and AskFirst$(v) \neq$ null then AskFirst$(v) <$ AskSecond(m_v).*

The following result shows that once a successful rematching has been established, then if the involved nodes in $\mu(V)$ are not privileged for a forced *Update* move, the involved nodes in $\sigma(V)$ will not move again.

Lemma 11. *Given nodes v, w, x, and y where $(v, w) \in M'$, $x \in \sigma(N(v))$ and $y \in \sigma(N(w))$. If $p_v = x$, $p_w = y$, $p_x = v$, $p_y = w$, AskFirst$(v) = x$, and AskSecond$(w) = y$, then if neither v nor w is privileged for a forced Update move, neither x nor y will move again.*

Next we show that the nodes in $\mu(V)$ will stabilize rapidly if no node in $\sigma(V)$ executes a move.

Lemma 12. *A node $v \in \mu(V)$ can make $O(1)$ moves between each time step that includes a move by a node in $\sigma(N(v))$*

Proof. Let $v \in \mu(V)$ and consider a maximal sequence S of time steps where no node in $\sigma(N(v))$ makes a move. Let a, b be the initial values of α_v, β_v and a', b' their values after the first (if any) *Update* move by v in S. Then from the *BestRematch* function we have that $a', b' \in \sigma(N(v)) \cup \{null\}$. Since the values of α_v and β_v are only changed by the *Update* rule they will remain in

$\sigma(N(v)) \cup \{null\}$ for the duration of S while $(\alpha_v, \beta_v) = BestRematch(v)$ also remains true.

The *Update* rule sets $p_v = null$ and any value subsequently assigned to p_v must be taken from the set $\{\alpha_v, \beta_v, null\}$. It follows that $p_v \in \sigma(N(v)) \cup \{null\}$ will remain true throughout S after the first *Update* move. From these observations it follows that there can at most be one *Update* move in S.

The remaining rules can only be triggered by changes in the values of α_v, β_v, $\alpha_{m_v}, \beta_{m_v}, p_v$, and p_{p_v}. From the above observation we know that there can only be four configurations of $\alpha_v, \beta_v, \alpha_{m_v}, \beta_{m_v}$ in S since each α, β pair can only change value once in S. It follows from Lemma 9 that for fixed $\alpha_v, \beta_v, \alpha_{m_v}, \beta_{m_v}$ values we must have one of the following configurations: *(i)* $AskFirst(v) \neq null$ and $AskSecond(v) = null$, *(ii)* $AskFirst(v) = null$ and $AskSecond(v) \neq null$, or *(iii)* $AskFirst(v) = null$ and $AskSecond(v) = null$. Thus only one of the rules *MatchFirst, MatchSecond*, and *ResetMatch* can be privileged before at least one of $\alpha_v, \beta_v, \alpha_{m_v}, \beta_{m_v}, p_v$ changes value. For each of these rules it is straightforward to see that the assignment to p_v or s_v cannot make the same rule become privileged again. The only assignment that can cause a new move is if p_{p_v} changes value which could result in *MatchFirst* to be executed consecutively more than once. But if $p_v \in \sigma(N(v))$ then p_{p_v} will not change in S. Also, if $p_v = null$ then p_{p_v} cannot change and if $p_v \notin \sigma(N(v)) \cup \{null\}$ then the next move executed by v will be an *Update* move. It follows that v can at most execute one move between each time that at least one of $\alpha_v, \beta_v, \alpha_{m_v}, \beta_{m_v}, p_v$ changes value in S and the result follows. □

In order to reason about *SingleNode* moves and the cause of these, we use the following definitions: Given a node $x \in \sigma(V)$ and a node $v \in \mu(V)$, we refer to x as being *asked first* in a rematch attempt if $AskFirst(v) = x$ and p_v is set to x. Similarly, we refer to x as being *asked second* if $AskSecond(v) = x$ and p_v is set to x. We say that x *accepts* the matching attempt if following either of the above cases it sets $p_x = v$. If x sets $p_x \neq v$ then x *rejects* the matching attempt by v.

Lemma 13. *The node y with the highest identifier in $\sigma(V)$ can execute moves during at most $O(\delta_y)$ time steps where δ_y is the degree of y.*

We now bound the total number of moves executed by nodes in $\sigma(V)$.

Lemma 14. *Each node in $\sigma(V)$ can execute moves during at most $O(2^{n+2} \cdot \Delta)$ time steps, where Δ is the maximum degree in the graph.*

Proof. Order the nodes in $\sigma(V)$ as $x_0, x_1, ..., x_{t-1}$ where $t = |\sigma(V)|$ such that $x_0 > x_1 > ... > x_t$. We denote the number of moves that a node x_i can execute as $L(i)$, and show by induction that $L(i) \leq \sum_{e=0}^{i-1} L(e) + O(\Delta)$.

The base case is $i = 0$. It was shown in Lemma 13 that the single node with the highest identifier in $\sigma(V)$ can execute at most $O(\Delta)$ moves. Thus $L(0) = O(\Delta)$.

For the induction step we assume that the bound holds for every node x_0, x_1, $..., x_{i-1}$ and prove that this implies that it also holds for x_i. We show this by considering the instances where x_i is asked second separately from where x_i is asked first.

The case where x_i is asked second is similar to the base case, and will thus result in $O(\Delta)$ moves.

For the case where x_i is asked first by some node v we first observe that if v is initially privileged for a forced *Update* move, then following this move x_i may become privileged to set $p_{x_i} \neq v$. However, if x_i is again asked first by v, we know that there exists a node $w = m_v$ where $k = AskSecond(w)$ and $k \neq null$. We now consider two cases: *(i)* $k \in \sigma(N(w))$ or *(ii)* $k \notin \sigma(N(w))$.

In Case *(i)* it follows that there exists a node $x_j \in \sigma(V)$ such that $x_j = k$. If $x_j < x_i$ then $\alpha_w \geq \beta_w$, which must be due to an incorrect initialization. Thus, w is privileged to execute a forced *Update* move, after which x_i may again become privileged. Subsequently, if x_i is again asked first by v, then Case *(i)* is again true, but now with $x_j > x_i$.

We will now show that x_i may only become privileged again due to moves made by x_j. At this point, both v and w must have executed any forced *Update* move, if they were privileged to do so. Obviously x_i will not become privileged while $p_v = x_i$, and from the predicate of the *Update* move we see that v will not become privileged for an *Update* move while $p_{x_i} = v$. From the *ResetMatch* predicate it follows that v may only become privileged if $AskSecond(w) = null$, which implies that x_j has made a move. Furthermore, from Lemma 11 we know that if x_j accepts the rematch attempt from w, x_i will not move again. Hence, when $k \in \sigma(N(w))$, the number of moves by x_i is bounded by $\sum_{k=0}^{i-1} L(k)$.

For Case *(ii)* note first that $k \notin \sigma(N(w))$ can only occur once initially due to incorrect initialization. In this case w is privileged for an *Update* move, and x_i may only become privileged again following this move. Since x_i has at most Δ neighbors, it follows that Case *(ii)* may at most cause $O(\Delta)$ additional moves for x_i. Combining the case where x_i is asked second with *(i)* and *(ii)* we get $L(i) \leq L(i-1) + L(i-2) + ... + L(0) + O(\Delta) \leq 2^{i+2} \cdot O(\Delta)$ and the result follows. □

Based on lemmas 12 and 14 we get the following bound on the step complexity of Algorithm 1 when using a distributed adversarial daemon.

Theorem 2. *Algorithm 1 will stabilize in* $O(2^{n+2} \cdot \Delta \cdot n)$ *time steps.*

4.2 Distributed Fair Daemon

In this section we consider the complexity of Algorithm 1 when run with a distributed fair daemon. Due to page constraints we only give an outline of the analysis.

We first note that following the first round, for any node $z \in V$ $p_z \in N(z) \cup \{null\}$, and additionally, for any node $v \in \mu(V)$, if $AskFirst(v) \neq null$ then $\alpha_v, \beta_v \in \sigma(N(v))$ and $AskFirst(v) < AskSecond(m_v)$. Consequently, if there exists an augmenting path of length three in the graph, then within $O(1)$ rounds, at least one node $v \in \mu(V)$ must have $p_v = x \neq null$ (possibly as a result of a *MatchFirst* move), where $x \in \sigma(V)$. Thus, within the end of the subsequent round, $p_x = w \neq null$ (note that w may be equal to v). If x was asked second by

w we know that a rematch attempt has succeeded. If x was asked first, we know that there exists a node y where $x < y$ that is asked second by m_w. Thus we can repeat the above argument, creating a chain of nodes in $\sigma(V)$ with increasing identifiers that must eventually lead to two edges joining the matching. Observe that the length of this chain is at most $O(n)$.

Thus we see that after at most $O(n)$ rounds at least two edges must join the matching, and since the cardinality of the matching is at most $O(n)$, we get the following result.

Theorem 3. *Algorithm 1 will stabilize in $O(n^2)$ rounds under a distributed fair daemon.*

4.3 Interaction with the Maximal Matching

While the previous two sections show that Algorithm 1 stabilizes when the underlying maximal matching is stable, we need to consider how Algorithm 1 functions on a non-stable maximal matching. We assume a maximal matching algorithm such as the one given by Manne et al. [14] and denote this as Algorithm 0. This algorithm has the property that if an edge becomes part of the matching then it will remain so for the remainder of the execution. We enforce that no rule in Algorithm 1 will become privileged on a node z if a rule in Algorithm 0 is privileged for the same node. Furthermore, if a node z in Algorithm 0 has made a bid to establish a new matching, then no rule in Algorithm 1 will become privileged for z until the attempt has either succeeded or failed (note that z is not necessarily privileged). This may for example occur if z is attempting to match with a neighbor, but has not yet received a response (for details of Algorithm 0, see [14]). Finally, we assume that Algorithm 0 does not use any variables from Algorithm 1.

Given the above, then at any point during the execution of the combined algorithm, there exists a (possibly empty or disconnected) subgraph of G where Algorithm 0 is stable. Since the non-stable nodes that border on this subgraph will not become privileged for Algorithm 1, it follows that any execution of Algorithm 1 will stabilize on G. Due to page restraints we omit further details.

The algorithm given in [14] has a complexity of $O(m)$ and $O(n)$ for the distributed adversarial and distributed fair daemon respectively, and thus the combined complexity of algorithms 0 and 1 is $O(2^{n+2} \cdot \Delta \cdot n \cdot m)$ for the distributed adversarial daemon and $O(n^2)$ for the distributed fair daemon.

5 Conclusion

We have presented the first self-stabilizing algorithm for computing a $\frac{2}{3}$- approximation to the maximum cardinality matching problem in a general graph. The algorithm uses only constant number of variables for each node, and stabilizes in $O(2^{n+2} \cdot \Delta \cdot n)$ time steps and $O(n^2)$ rounds for the distributed adversarial and distributed fair daemon, respectively, when assuming a stable underlying maximal matching.

It is worth noting that it would have been possible to design an algorithm such that through the use of identifiers, the eventual solution is deterministic, i.e. unaffected by the initial state of the graph and the order in which rules are executed. This algorithm would conceivably be both shorter and have a better complexity than the one presented here, but at the cost of robustness. That is, in the presented algorithm, adding or removing a node in a stable solution would have little or no effect on the majority of the graph, while the hypothetical strict algorithm would possibly have to redo the entire stabilization process.

A possible area for future research is to investigate how better approximation ratios than $\frac{2}{3}$ could be achieved with complexity efficient self-stabilizing algorithms. Furthermore, it would be of interest to see if the algorithm given here could be generalized for weighted instances of the matching problem, or if the stabilization time can be improved.

References

1. Blair, J.R.S., Manne, F.: Efficient self-stabilizing algorithms for tree networks. In: ICDCS 2003: Proceedings of the 23rd International Conference on Distributed Computing Systems, Washington, DC, USA, pp. 20–26. IEEE Computer Society Press, Los Alamitos (2003)
2. Danturi, P., Nesterenko, M., Tixeuil, S.: Self-stabilizing philosophers with generic conflicts. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, pp. 214–230. Springer, Heidelberg (2006)
3. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM 17(11), 643–644 (1974)
4. Dolev, S.: Self-Stabilization. MIT Press, Cambridge (2000)
5. Ghosh, S., Gupta, A., Karaata, M.H., Pemmaraju, S.V.: Self-stabilizing dynamic programming algorithms on trees. In: Proceedings of the Second Workshop on Self-Stabilizing Systems (WSSS 1995), Las Vegas, pp. 11.1–11.15 (1995)
6. 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 2003: Proceedings of the 17th International Symposium on Parallel and Distributed Processing, Washington, DC, USA, p. 162.2. IEEE Computer Society Press, Los Alamitos (2003)
7. Goddard, W., Hedetniemi, S.T., Jacobs, D.P., Trevisan, V.: Distance-k knowledge in self-stabilizing algorithms. Theor. Comput. Sci. 399(1-2), 118–127 (2008)
8. Goddard, W., Hedetniemi, S.T., Shi, Z.: 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, pp. 797–803. CSREA Press (2006)
9. 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, vol. 2150, pp. 458–465. Springer, Heidelberg (2001)
10. Gradinariu, M., Tixeuil, S.: Conflict managers for self-stabilization without fairness assumption. In: ICDCS 2007: Proceedings of the International Conference on Distributed Computing Systems. IEEE Computer Society Press, Los Alamitos (2007)

11. Hedetniemi, S.T., Jacobs, D.P., Srimani, P.K.: Maximal matching stabilizes in time O*(m)*. Inf. Process. Lett. 80(5), 221–223 (2001)
12. Hopcroft, J.E., Karp, R.M.: An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. SIAM J. Comput. 2(4), 225–231 (1973)
13. Hsu, S.-C., Huang, S.-T.: A self-stabilizing algorithm for maximal matching. Inf. Process. Lett. 43(2), 77–81 (1992)
14. Manne, F., Mjelde, M., Pilard, L., Tixeuil, S.: A new self-stabilizing maximal matching algorithm. In: Prencipe, G., Zaks, S. (eds.) SIROCCO 2007. LNCS, vol. 4474, pp. 96–108. Springer, Heidelberg (2007)
15. Tel, G.: Maximal matching stabilizes in quadratic time. Inf. Process. Lett. 49(6), 271–272 (1994)