Maximum Matching for Anonymous Trees with Constant Space per Process*

Ajoy K. Datta[1], **Lawrence L. Larmore**[2], **and Toshimitsu Masuzawa**[3]

1 **University of Nevada, Las Vegas, USA**
`ajoy.datta@unlv.edu`
2 **University of Nevada, Las Vegas, USA**
`lawrence.larmore@unlv.edu`
3 **Graduate School of Information Science and Technology, Osaka University, Osaka, Japan**
`masuzawa@ist.osaka-u.ac.jp`

Abstract

We give a silent self-stabilizing protocol for computing a maximum matching in an anonymous network with a tree topology. The round complexity of our protocol is $O(diam)$, where $diam$ is the diameter of the network, and the step complexity is $O(n\,diam)$, where n is the number of processes in the network. The working space complexity is $O(1)$ per process, although the output necessarily takes $O(\log \delta)$ space per process, where δ is the degree of that process. To implement parent pointers in constant space, regardless of degree, we use the cyclic Abelian group $\mathbb{Z}_7$.

1998 ACM Subject Classification C.2.4 [Computer-Communication Networks] Distributed Systems, F.2.2 [Analysis of Algorithms and Problem Complexity] Nonnumerical Algorithms and Problems, G.2.2 [Discrete Mathematics] Graph Algorithms

Keywords and phrases anonymous tree,maximum matching,self-stabilization, unfair daemon

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.16

1 Introduction

Self-stabilization [5] is a paradigm for enhancing autonomous adaptability of distributed systems to network dynamics, such as transient faults and topology changes. A self-stabilizing system is guaranteed to regain its intended (or legal) behavior when the system is arbitrarily disturbed. Several fundamental problems have been solved by self-stabilizing algorithms, including leader election, spanning tree construction, mutual exclusion, node/edge coloring, and so forth. Maximum or maximal matching is one of the most investigated of these problems *et al.* [8].

1.1 Related Work

The first self-stabilizing algorithm for *maximal* matching was given by Hsu *et al.* [10]. The algorithm works for arbitrary anonymous networks under the *central daemon*, where no two processes can act simultaneously. Efficiency under the central daemon is usually measured by the number of steps required for convergence to a legitimate configuration. Their algorithm takes $O(n^3)$ steps, where n is the number of processes in the system. This result was improved in [9, 12, 15]. Hedetniemi *et al.* give an algorithm which takes $O(m)$ steps where m is the

* This work was supported in part by JSPS KAKENHI, Grant Number (B)26280022.

number of edges in the system [9]. A synchronous version of the algorithm with round complexity $O(n)$ is given by Goddard *et al.* [6].

A self-stabilizing algorithm for *maximal* matching in arbitrary networks under the *read/write daemon*, where only a single read or write is permitted during an atomic step, was given by Chattopadhyay *et al.* [3]. The algorithm converges in $O(n)$ (asynchronous) rounds, provided all processes have identifiers, and no two processes within distance two have the same identifier. The same paper also gives a randomized self-stabilizing algorithm for assigning the locally distinct identifiers in $O(1)$ expected rounds. With the assumption of distinct identifiers within distance two, a self-stabilizing maximal matching algorithm for arbitrary networks under the *distributed daemon*, where any number of processes can act simultaneously, which converges in $O(n)$ rounds and $O(m)$ steps, was given by Manne *et al.* [13].

A number of self-stabilizing *maximum* matching algorithms have been given. Karaata and Saleh give a self-stabilizing algorithm for anonymous tree networks, under the central daemon, which converges in $O(n^4)$ steps [11]. A self-stabilizing algorithm for tree networks under the read/write daemon, which converges in $O(n^2)$ steps, is given by Blair and Manne [2]. For anonymous bipartite networks, a self-stabilizing algorithms converging in $O(n^2)$ rounds under the central daemon is given by Chattopadhyay *et al.* [3].

A maximal matching is defined to be *1-maximal* if the size of matching cannot be increased by replacing one edge in the matching with two other edges. Any 1-maximal matching is a $\frac{2}{3}$-approximation of the maximum matching, while an arbitrary maximal matching is only a $\frac{1}{2}$-approximation to maximum.

A self-stabilizing 1-maximal matching algorithm, under the central daemon, for anonymous trees, and for rings with length not divisible by three, which converges in $O(n^4)$ rounds, is given by Goddard *et al.* [7]. The same paper also shows that there is no self-stabilizing 1-maximal matching algorithm for anonymous rings of length a multiple of three. These results are generalized by Asada and Inoue, who give a self-stabilizing 1-maximal matching algorithm for any anonymous network under the central daemon, provided that network has no cycle of length divisible by three, which converges in $O(m)$ steps [1].

For non-anonymous arbitrary networks, a self-stabilizing 1-maximal matching algorithm under the distributed daemon is given by Manne *et al.* [14]. Their algorithm converges in $O(n^2)$ rounds, but requires exponentially many steps in the worst case.

1.2 Contribution

We use of *virtual pointers*, in which we indicate a parental relation between neighboring processes by assigning each process a label of finite size, that is, written with finitely many bits, where the parental relation is a function of the labels of the two processes. In this paper, each label, which we call *level*, is a member of the cyclic Abelian group $\mathbb{Z}_7$, and x is a parent of y if the difference of the two values of *level* is in a certain range. We are able to maintain constant space per process by eliminating the standard parent pointers, which require $O(\log \delta)$ space per process, where δ is its degree.

MATCH builds a spanning tree, or a spanning structure consisting of two trees, in the network. MATCH then uses a bottom-up dynamic program to choose a maximum matching for the network. The working space complexity of MATCH is $O(1)$ per process, regardless of degree, the round complexity is $O(\mathit{diam})$, where *diam* is the diameter of the network, and the step complexity is $O(n\,\mathit{diam})$. Our use of $\mathbb{Z}_7$ is similar to the use of $\mathbb{Z}_5$ for virtual pointers in [4]. The output must use $O(\log \delta)$ space per process, where δ is the degree of that process.

1.3 Outline

In Section 2, we give some basic definitions, and describe our model of computation. In Section 4, we formally define MATCH. In Section 5, we prove that MATCH is correct. In Section 6 we prove that MATCH takes $O(diam)$ rounds, while in Section 7, we prove that MATCH takes $O(n\, diam)$ steps. Section 8 concludes the paper.

2 Preliminaries

We say that a network is *undirected* if the edges have no specified orientation, *i.e.,* that the edge $\{x, y\}$ is the same as the edge $\{y, x\}$. We say that a network is *anonymous* if the processes have no identifiers. We say that a network is a *tree* if it is connected and has no cycles. The algorithm we give in this paper assumes an undirected anonymous tree.

2.1 Model of Computation

We use the shared memory model of computation [5], meaning that each process can read its own registers and those of its neighbors, and can change only its own registers. A distributed algorithm consists of a program for each process, and that program consists of a finite set of *actions*. Each action for a process x has a *guard*, which is a predicate (*i.e.*,, Boolean function) on the registers of x and its neighbors, together with a *statement*, which is simply the assignment, or reassignment, of one or more registers of x. If the guard of an action of x is TRUE, then we say that action is *enabled*, and we say x is enabled if at least one of its actions is enabled.

We assume the *unfair distributed daemon*. If at least one process is enabled, the daemon *selects* at least one of these enabled processes. Each selected process executes one of its enabled actions, and that concludes one step. We describe the daemon as *unfair* because an enabled process need never be selected, unless it becomes the only enabled process.

We will assume that there is a set $\mathcal{F}$ of configurations which we call the *legitimate* configurations. (We call the remaining configurations *illegitimate*.) $\mathcal{F}$ satisfies two conditions:

Closure: No action can change a legitimate configuration to an illegitimate configuration.

Correctness: Each legitimate configuration satisfies the output conditions of the problem.

A configuration is *final* if no process is enabled at that configuration. An algorithm is *silent* if every computation ends at a final configuration.

3 Overview of MATCH

We are given an anonymous unoriented network G with a tree topology. The first phase of MATCH is to build a rooted spanning tree for G, or possibly two rooted trees joined at the roots which together span G. During the second phase (which actually begins before the first phase is finished) we use the tree, or trees, to define a maximum matching.

At any given configuration of MATCH, every neighbor of a process x is either a *child* of x, a *parent* of x, or a *peer* of x. From a possibly chaotic initial configuration, MATCH organizes G into one or two trees; if there are two trees, the roots are peers of each other.

The maximum matching itself is then constructed by a bottom-up wave of the tree (or trees). Each process is assigned the Boolean label 0 or 1, which we call *flag*, as follows. Leaves are assigned 0. A process is assigned 1 if it has a child labeled 0, otherwise 0. At the end, every process labeled 1 is matched with one of its children labeled 0, while two roots are matched with each other if both are labeled 0.

Virtual Pointers. It is typical to define rooted trees by using parent pointers. But we need to maintain $O(1)$ space complexity at each process, regardless of degree. For that reason, we express parent/child relations using *virtual pointers.* These virtual pointers are defined by storing a single variable, $x.level$ at each process. The value of $x.level$ will be an element of $\mathbb{Z}_7$, the cyclic Abelian group of order 7, and such requires only three bits to store.

More formally, if $i, j \in \mathbb{Z}_7 = \{0,1,2,3,4,5,6\}$, we say that $i < j$, and $j > i$, if $j - i \in \{1,2,3\}$, where addition and subtraction are in the Abelian group, *e.g.*, $2 < 3$, $2 < 4$, $2 < 5$, and $6 < 2$. This relation are not transitive, since $1 < 3 < 5$, but $5 < 1$. If y is a neighbor of x in the network we say that y is a child of x if $y.level > x.level$. Similarly, y is a parent of x if x is a child of y, *i.e.*, $y.level < x.level$, while y is a peer of x if $y.level = x.level$. Thus, to change the parent/child/peer relationships, we simply change the values of $level$.

How Virtual Pointers are Used. An algorithm which uses local addresses for pointers never has to change its parent pointer unless it changes its parent. That simple rule does not hold for $\mathbb{Z}_7$-virtual pointers. We say that $parent(x) = y$ if $y.level < x.level$. A problem then arises if it is necessary to change the level of either x or y. If $x.level = 2 + y.level$, the "ideal" parent/child relation, either $x.level$ or $y.level$ may be either incremented or decremented without changing the parent/child relationship. However, if $x.level = 1 + y.level$, decrementation of $x.level$ would cause x and y to become peers; similarly, if $x.level = 3 + y.level$, incrementation of $x.level$ would cause the parent/child relation to be reversed: x would become the parent of y. For this reason, whenever $parent(x) = y$ and $x.level \neq 2 + y.level$, x tries to either increment or decrement its level to restore the ideal. Until that ideal is restored, y is not enabled to execute an action which changes its level.

Think of the link between parent and child as a spring which has an ideal length (namely 2) but whose length can be stretched to 3 or compressed to 1. After that distortion, the spring tries to restore its length to the ideal 2.

We use the group $\mathbb{Z}_7$ because there is no smaller group which allows the same flexibility in both directions, namely both compression and stretching of the parent/child link.

3.1 Approach

In order to describe how MATCH creates a tree, or trees, we need to introduce an abstract function $Level(x)$. $Level$ is an integer function, and has the following properties:

1. The standard projection $\mathbb{Z} \to \mathbb{Z}_7$ maps $Level(x)$ to $x.level$. For example, if $Level(x) = 11$ then $x.level = 4$. *i.e.,*
2. If $y \in N(x)$, then $|Level(y) - Level(x)| \leqslant 3$.
3. $Level(x)$ increments (decrements) at each step at which $x.level$ increments (decrements).

We also introduce the abstract function $Min\_Level = \min\{Level(x) : x \in G\}$. $Level$ can be initialized arbitrarily; however in Lemma 6 below, we prove that Min_Level is constant, and thus we can assume, without loss of generality, that $Min\_Level = 0$, which implies that, during the computation, all values of $Level$ are non-negative. Eventually, $Level(x) = 2\,d(x)$ for each process, where $d(x)$ is the distance, through G, from x to the root of the tree, or the nearer root if there are two trees.

4 Formal Definition of MATCH

Variables and functions of MATCH. We use the dot notation, such as $x.var$, for a variable of x, while we use the normal functional notation such as $Func(x)$, for a function of x.

1. $x.level \in \mathbb{Z}_7$. This is the variable that allows us to define virtual parent pointers using constant space per process: x is the parent of y if $x.level < y.level$.
2. $Chldrn\,(x) = \{y \in N(x) : y.level > x.level\}$, the *children* of x, where $N(x)$ is the set of neighbors of x.
3. $Peers\,(x) = \{y \in N(x) : y.level = x.level\}$, the *peers* of x, those neighbors at the same level as x in the spanning tree.
4. $Prnts\,(x) = \{y \in N(x) : y.level < x.level\}$, the *parents* of x, those neighbors above x in the spanning tree.
5. $Children\_Ok\,(x) \equiv \forall y \in Chldrn\,(x)(y.level = 2 + x.level)$, Boolean. This means that all children of x are in the "optimum" position, meaning two levels below x.
6. $\text{Class}\,(x) \in \{\text{I}, \text{II}, \text{III}, \text{IV}, \text{V}\}$, which we define below. At a final configuration, if there is one root, it has Class I, while if there are two roots, they each have Class II; while in both cases, all other processes have Class III.
7. $x.flag$, Boolean. This variable is used in the bottom-up protocol which determines the matched pairs. Except for the possibility of two roots being matched, all matched pairs consist of processes with opposite values of *flag*.
   The values of *flag* are computed using a bottom-up dynamic program, and those values define a maximum matching, once the structure of the rooted spanning tree, or trees, is finalized.
8. $x.partner \in N(x) \cup \{\perp\}$, the partner that x is matched to. If $x.partner = \perp$, then x is unmatched. This variable, which is the output of the protocol, takes $O(\log\delta)$ space, where δ is the degree of x.

Classes of Processes. At each configuration, each process belongs to one of five Classes; its Class is defined by its number of parents and peers, as given by the table below. The Class of a process is arguably its most important property.

# parents	# peers	Class
0	0	I
0	1	II
1	0	III
0	$\geqslant 2$	IV
1	$\geqslant 1$	V
$\geqslant 2$	arbitrary	V

We write $peer(x)$ for the sole peer of a process in Class II, and we write $parent\,(x)$ for the sole parent of a process in Class III.

For any process x, let T_x be the subtree rooted at x, namely the set consisting of x, its children, its children's children, *etc.*. We define a process x to be *regular* if $\text{Class}\,(x) \in \{\text{I}, \text{II}, \text{III}\}$ and all descendants of x have Class III. At a final configuration, all processes are regular.

We now define additional variables and functions computable by a process.

9. $x.rglr$, Boolean. This variable means that x currently "believes" that it is regular.
   We define a process to be *strongly regular* if it is regular and $y.rglr$ for all $y \in T_x$.
10. $Rglr\,(x) \equiv (\text{Class}\,(x) \in \{\text{I}, \text{II}, \text{III}\}) \wedge (\forall y \in Chldrn\,(x) y.rglr)$, Boolean. This predicate is used to update $x.rglr$.
11. $Flag(x) = \begin{cases} \text{TRUE if } (\text{Class}\,(x) \in \{\text{I}, \text{II}, \text{III}\}) \wedge (\exists y \in Chldrn\,(x)\, y.flag = \text{FALSE}) \\ \text{FALSE otherwise} \end{cases}$
    This predicate is used to update $x.flag$.

12. $\mathit{Partner}(x) \in N(x) \cup \{\perp\}$, the process that x should be matched with. If $x.\mathit{flag} = \textsc{true}$, then $\mathit{Partner}(x)$ is some child of x whose flag is $\textsc{false}$, if any, while if $\mathrm{Class}\,(x) = \mathrm{II}$, $x.\mathit{flag} = \textsc{false}$, and $\mathit{peer}(x).\mathit{flag} = \textsc{false}$, then $\mathit{Partner}(x) = \mathit{peer}(x)$. If $\mathrm{Class}\,(x) = \mathrm{III}$, $x.\mathit{flag} = \textsc{false}$, and $\mathit{parent}(x).\mathit{partner} = x$, then $\mathit{Partner}(x) = \mathit{parent}(x)$. In all other cases, $\mathit{Partner}(x) = \perp$.

The variable $x.\mathit{partner}$, and the corresponding function $\mathit{Partner}(x)$, use $O(\log \delta)$ space per process, but are used only for output. All intermediate computations use $O(1)$ space per process, hence we say that Match uses constant *working space* per process.

Regular, *Rglr*, and *rglr*. We have used the word "regular," or contractions thereof, in three different ways. *Regularity* is an abstract property, not computable by any process during the execution of Match; $x.\mathit{rglr}$ is a working estimate of the regularity of x.

Code of Match

We now present the protocol Match for an arbitrary process x, in program form. We define a process x to be *enabled* if execution of the protocol by x results in the change of at least one variable of x. At each step, the daemon selects an arbitrary non-empty set of enabled processes; if there are no enabled processes, the configuration is final.

```
1:  if $x.\mathit{rglr} \neq \mathit{Rglr}(x)$ then
2:      $x.\mathit{rglr} \leftarrow \mathit{Rglr}(x)$
3:  else if $\mathit{Children\_Ok}(x)$ then
4:      if $(\mathrm{Class}\,(x) = \mathrm{II}) \wedge x.\mathit{rglr} \wedge \neg \mathit{peer}(x).\mathit{rglr}$ then
5:          $x.\mathit{level} \leftarrow x.\mathit{level} + 1$
6:      else if $(\mathrm{Class}\,(x) = \mathrm{III}) \wedge x.\mathit{rglr} \wedge (x.\mathit{level} = \mathit{parent}(x).\mathit{level} + 1)$ then
7:          $x.\mathit{level} \leftarrow x.\mathit{level} + 1$
8:      else if $(\mathrm{Class}\,(x) = \mathrm{III}) \wedge x.\mathit{rglr} \wedge (x.\mathit{level} = \mathit{parent}(x).\mathit{level} + 3)$ then
9:          $x.\mathit{level} \leftarrow x.\mathit{level} - 1$
10:     else if $(\mathrm{Class}\,(x) = \mathrm{V}) \wedge \neg x.\mathit{rglr} \wedge (y \in \mathit{Peers}(x) \cup \mathit{Prnts}(x) \implies \neg y.\mathit{rglr})$ then
11:         $x.\mathit{level} \leftarrow x.\mathit{level} - 1$
12:     end if
13: end if
14: if $x.\mathit{rglr} \wedge (x.\mathit{flag} \neq \mathit{Flag}(x))$ then
15:     $x.\mathit{flag} \leftarrow \mathit{Flag}(x)$
16: else if $x.\mathit{rglr} \wedge x.\mathit{partner} \neq \mathit{Partner}(x)$ then
17:     $x.\mathit{partner} \leftarrow \mathit{Partner}(x)$
18: end if
```

Notation. In our discussion, we will say that a process executes an *action* if it executes one of the lines of the code which changes one of its variables, *i.e.,* Line 2, 5, 7, 9, 11, 15, or 17. Note that a process is enabled to execute at most one of those actions per step.

We say that a process executes a *level action* if it executes a line that changes its level, namely Line 5, 7, 9, or 11.

Intuitive Explanation of the Actions

Line 5 increases the level of a regular process, and changes its Class from II to III, provided its peer is not regular; its peer then becomes its parent. In Figure 1, **b** executes the action at

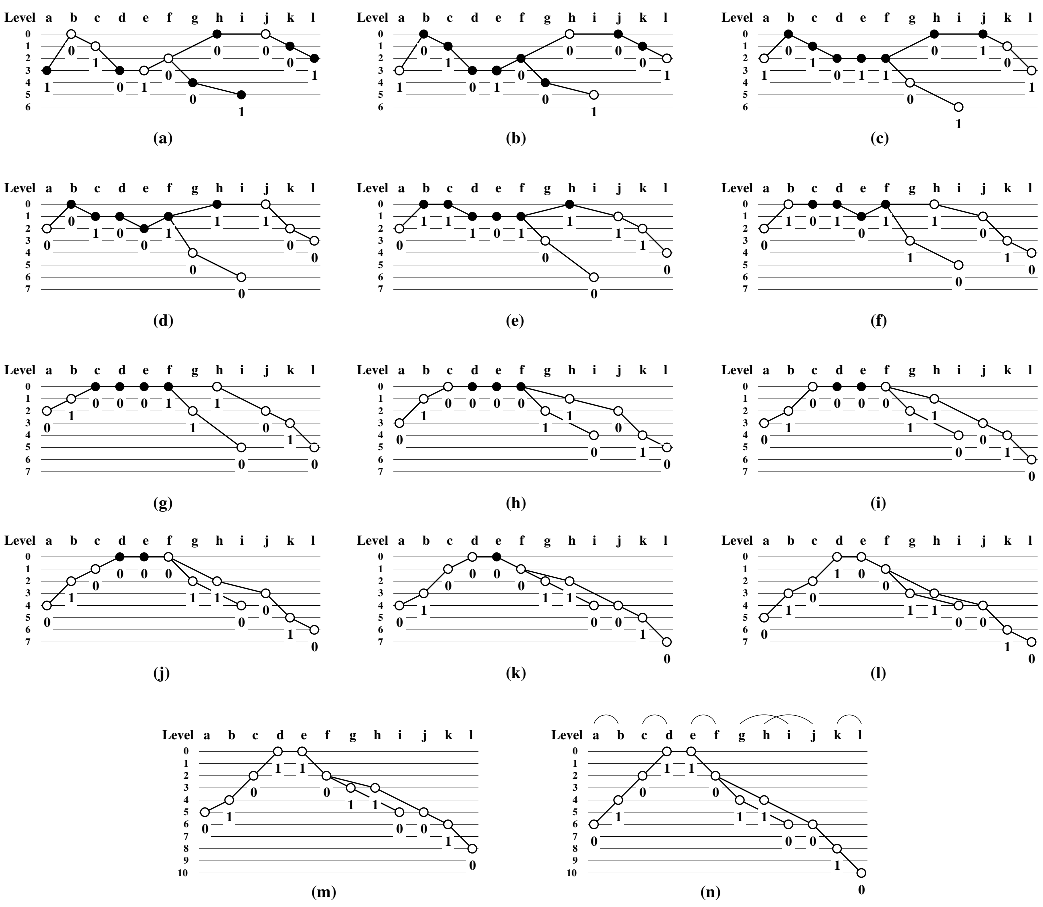

Figure 1 Example computation of Match, where G consists of twelve processes, named $\mathbf{a}, \mathbf{b}, \ldots \mathbf{k}, \mathbf{l}$. Values of *flag* are indicated under the processes. Process x is shown by an open dot if $x.rglr$, otherwise a solid dot. Configuration (a) is "arbitrary." Processes **d** and **e** are computed to be roots, though neither has level zero in the initial configuration. We assume the synchronous daemon, *i.e.,* at each step every enabled process executes. Several steps are skipped between (m) and the final configuration (n). To avoid clutter, matching actions, *i.e.,* executions of Line 17, are not shown except in the last figure. The matched pairs are $\{\mathbf{a}, \mathbf{b}\}$, $\{\mathbf{c}, \mathbf{d}\}$, $\{\mathbf{e}, \mathbf{f}\}$, $\{\mathbf{g}, \mathbf{i}\}$, $\{\mathbf{h}, \mathbf{j}\}$, and $\{\mathbf{k}, \mathbf{l}\}$.

(f), **h** executes the action at (g), **c** executes the action at (i), and **f** executes the action at (j).

Lines 7 and 9 adjust the levels of regular processes of Class III without changing their virtual pointers; in those cases, a process x adjusts the difference between its level and that of its parent from 1 or 3 to 2; execution of those lines has no effect on the parent/child/peer structure of the network. There are many examples of those actions illustrated in Figure 1, such as **k** at (c). In fact, at each step from (b) to (m), at least two processes of Class III execute a level action.

Of the actions shown in Figure 1, Line 15 has the lowest priority, meaning that the value of $x.flag$ cannot change if x executes Line 2 or any level action. All $x.flag$ have reached their final values, as shown in (m), before the level actions are finished. The level actions are the last to be completed in the example, and the final configuration is shown in (n).

Execution of Line 11 of the code decreases the level of process of Class V. We do not allow a process x to execute that action unless $y.rglr$ = FALSE for all $y \in Peers(x) \cup Prnts(x)$.

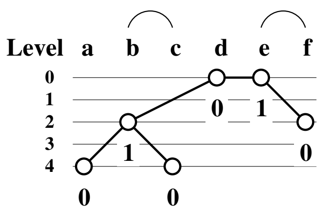

Figure 2 An example final configuration of MATCH, where G is a tree consisting of six processes, named $\mathbf{a}, \mathbf{b}, \ldots \mathbf{f}$. Values of *flag* are indicated under the processes. The matched pairs are $\{\mathbf{b}, \mathbf{c}\}$ and $\{\mathbf{e}, \mathbf{f}\}$, while **a** and **d** remain unmatched.

There are a number of examples of this action in Figure 1, such as processes **d** and **e** at (b), **d** and **f** at (c), **c** and **e** at (d).

The Kernel. Let K be the set of all processes of Classes I, II, IV, and V, together with all irregular processes of Class III. We call K the *kernel.* The purpose of Line 11 is to squeeze K into the 0th level. Once that has happened, as shown in Figure 1(g), all irregular processes have Class IV, and K is a tree subnetwork (actually merely a chain in Figure 1) of G whose leaves all have Class II. In the remaining steps, that subnetwork will be decremented as its leaves change Class from II to III.

4.1 Top Level Summary of MATCH

At the top level, MATCH executes the following level actions.

1. All processes in the K move to the 0th level, executing Line 11. K will be a tree.
2. The leaves of that tree will all have Class II. Each Class II process will increase its level by executing Line 5, deleting itself from the kernel, and leaving the 0th level.
3. Eventually, the kernel will consist of either one or two processes at the 0th level. If one process, it will be the root. If two, and $rglr = \text{TRUE}$ for both, they will be co-roots.
4. As all these level actions are proceeding, regular processes will continually execute Lines 7 and 9, continually adjusting differences between the levels of parents and children to make the links ideal, so that the other level actions will be enabled.
5. The Flag and Matching actions, Lines 15 and 17, implement a simple dynamic program that assigns a flag value for each process, and then chooses a maximum matching in a rooted tree. A process of flag 1 always has at least one child of flag 0, and matches with it. A process of flag 0 cannot match with any of its children. In the example shown in Figure 1, there are no unmatched processes. However, if a process has more than one child of flag 0, one of those children will remain unmatched, as indicated in Figure 2 below; in that figure **b** has flag 1 and its children both have flag 0; thus, one of those children must remain unmatched.
6. At the very end of the dynamic program, if there are two roots, they match with each other if they both have flag 0. A root with flag 1 matches with one of its children, as *per* the above rule. Thus, if one root has flag 1 and the other 0, the root with flag 0 remains unmatched, as shown in Figure 2 below.

4.2 Legitimate Configurations of MATCH

A legitimate configuration consists of one or two trees, whose root, or roots, are at level zero. All parental links are ideal, meaning that the difference in levels between parent and child is two in every case. Furthermore, in a legitimate configuration, $x.rglr = \textsc{true}$ for all x. It follows that there is either one process of Class I or two neighboring processes of Class II, and all other processes are of Class III. In addition, and $x.flag = Flag(x)$ for all x, and all processes have matched with their final partners, or remain unmatched.

Every final configuration is legitimate, as we prove in Lemma 1. However, a legitimate configuration may not be final. For example, Figure 1(m) shows a legitimate but non-final configuration. The legitimate configuration is not unique. There is no general polynomial bound on the number of distinct maximum matchings of a tree graph.

5 Correctness

Correctness of Match follows from the following two statements:

1. Every final configuration of Match is legitimate, as we prove in Lemma 1 below,
2. There is no infinite computation of Match, as we prove in Lemma 12 below.

▶ **Lemma 1.** *Any final configuration of* Match *is legitimate.*

Proof. Suppose that the current configuration of Match is final, *i.e.*, no process is enabled to execute any action of the code.

▶ Claim 2. $x.rglr = \textsc{true}$ for any process x.

Proof of Claim 2. Suppose not. Let $L = \max\{Level(x) : \neg x.rglr\}$, and let x be a process of level L such that $x.rglr = \textsc{false}$. Thus $y.rglr = \textsc{true}$ for all $y \in Chldrn(x)$. Hence, if the Class of x is I, II, or III, $Rglr(x) = \textsc{true}$, contradiction.

▶ Claim 3. If Class $(x) = $ V then x is enabled.

Proof of Claim 3. If $y \in Prnts(x)$ and $y.rglr$, then y is enabled to execute Line 2, contradiction. If $y \in Peers(x)$, then Class $(y) = $ IV, hence $y.rglr = \textsc{false}$, because otherwise y would be enabled. Thus, x is enabled. ◀

The only remaining possibility is that Class $(x) = $ IV. Since each process at level L has Class IV, and each has at least two peers also at level L, the subgraph of processes at level L must contain a cycle, contradiction. This completes the proof of Claim 2. ◀

By Claim 2 and since $x.rglr = Rglr(x)$ for all x, there are no processes of Class IV or V. Since any process of Class III has a parent, any process at Level zero must have Class I or II.

▶ Claim 4. If $Level(x) = 0$ and Class $(x) = $ I, then all processes other than x have Class III and are descendants of x.

Proof of Claim 4. Suppose $y \neq x$ is a process, and y is not a descendant of x. Let σ be the unique path through G from x to y, and let z be the process of maximum level in σ which is closest to x. If $z \neq y$, then z has Class V, contradiction. Thus, $z = y$, and y is a descendant of x. Since all processes are descendants of x, all other processes must have Class III. ◀

▶ Claim 5. If there is no process of Class I at level zero, then there are two processes of Class II at level zero which are peers of each other, and every other process has Class III.

Proof of Claim 5. There must be a process x of Class II at level zero. Let $y = peer(x)$. Let z be any other process, and let σ be the unique path through G from z to x. By an argument similar to that in the proof of Claim 4, z is a descendant of y if σ passes through y, and otherwise is a descendant of x. In either case Class $(z) = $ III. ◀

The lemma follows from Claims 4 and 5. ◀

▶ **Lemma 6.** *Min_Level does not change during any computation of* MATCH*.*

Proof. By contradiction. Suppose Min_Level decreases during a step. Then there is some process x such that $Level(x) = Min\_Level$ and $Level(x)$ decreases at the next step. Since no process has a level less than $Level(x)$, we know that Class (x) is neither III nor V. However, the guards of the actions do not permit $x.level$ to decrease if x belongs to any other Class. Thus, Min_Level does not decrease.

Now, suppose Min_Level increases during the step. For any process x such that $Level(x) = M = Min\_Level$ before the step, $Level(x)$ must increase, hence Class (x) must be either II or III. But x has no parent, hence Class (x) = II, which implies that x has a peer. Since $peer(x).rglr$ = FALSE, $Level(peer(x))$ cannot decrease, and thus there is still a process whose level is M after the step, contradiction. ◀

▶ Remark 7. If a process x is regular at some step, then x is regular at all subsequent steps.

No Computational Cycle. In this paragraph, we assume that Ξ is a non-trivial computational cycle of MATCH. Our goal is to prove that Ξ cannot exist.

▶ **Lemma 8.** *No process changes its regularity during Ξ.*

Proof. By Remark 7, a process does not change from regular to irregular. Since every step of Ξ must be reversible, no process can change from irregular to regular during Ξ. ◀

▶ **Lemma 9.** *If $x.rglr$ =* TRUE *at any step of Ξ, then x is regular.*

Proof. By contradiction. Suppose the lemma is false. Let S be the set of ordered triples (x, t, L) such that the process x is irregular at the t^{th} step of Ξ, and that $x.rglr$ = TRUE and $Level(x) = L$ at that step. Pick such a triple (x, t, L) such that L is maximum over all members of S. Without loss of generality, x executes an action at step t.

By Lemma 8, x is irregular at step $t-1$. If Class (x) = V at step $t-1$, then $x.rglr$ = FALSE at step t, contradiction. Otherwise, there is some $y \in Chldrn^{t-1}(x)$ such that y is irregular at step $t-1$. Let $L' = Level(y)$ at step $t-1$. If $y.rglr$ = FALSE at step $t-1$, then $x.rglr$ = FALSE at step t, contradiction. Thus $y.rglr$ = TRUE at step $t-1$, hence $(y, t-1, L') \in S$, and $L' > L$, which contradicts the maximality of L. ◀

▶ **Lemma 10.** *During Ξ, no irregular process changes level.*

Proof. By contradiction. Suppose x is irregular and changes its level during Ξ. Since Ξ is a cycle, $x.level$ must both increase and decrease during Ξ. By Lemma 9, $x.rglr$ = FALSE during Ξ, and thus by the definitions of the actions $x.level$ can only decrease, contradiction. ◀

▶ **Lemma 11.** *During Ξ, no regular process changes level.*

Proof. We first observe that no process of Class III can change to any other Class. Suppose x is a regular process. If Class (x) = I, it cannot change its level. If Class (x) = II, then x cannot change its level, because its Class would change to III, and that step would be irreversible. Thus Class (x) = III.

The statement of the lemma is proven by induction on level. If $Level(x) = 0$, then Class $(x) \neq$ III. If $Level(x) > 0$, let $y = parent(x)$. By either Lemma 10 or the inductive hypothesis, depending on whether y is irregular or regular, y does not change level during Ξ. Thus, x can change level at most once during Ξ, and that change is irreversible. Since Ξ is a cycle, x does not change level at all. ◀

▶ **Lemma 12.** *The computation of* MATCH *is acyclic.*

Proof. Suppose Ξ is a cycle of computation of MATCH. By Lemmas 8, 10, and 11, no process executes Line 2 nor any level action during Ξ. Given that regularity and levels are fixed, a process can execute neither Line 15 nor Line 17 infinitely often, and thus Ξ cannot exist. Hence MATCH is acyclic. ◀

▶ **Theorem 13.** MATCH *is correct.*

Proof. Correctness follows immediately from Lemmas 1 and 12. ◀

6 Round Complexity

Monus Notation. The operator *monus,* written "$\dot{-}$" on non-negative integers is a variant of subtraction, but never yields a negative. Formally, $x \dot{-} y = \max\{x - y, 0\}$. For example, $5 \dot{-} 3 = 2$, while $3 \dot{-} 5 = 0$. Monus is left-associative and has the same precedence as addition and subtraction. Note that $i \dot{-} j \dot{-} k = i \dot{-} (j + k)$.

We define a sequence of potentials. Let *SR* be the set of strongly regular processes.

1. $\pi(x) = \begin{cases} -1 \text{ if } (\text{Class}(x) = \text{III}) \wedge (x \in SR) \wedge (x.level = 2 + parent(x).level) \\ 1 \text{ otherwise} \end{cases}$
2. $\theta(x) = \max\{0, \pi(x) + \max\{\theta(y) : y \in Chldrn(x)\}\}$
3. $\mu(x) = \begin{cases} 0 \text{ if } Chldrn(x) = \varnothing \\ \max\{\theta(y) : y \in Chldrn(x)\} \text{ otherwise} \end{cases}$
4. $d(x) =$ the distance through G from x to r where r is the root that will eventually be computed by MATCH. If two roots are computed, take r to be the one closer to x.
5. $\psi(x) = d(x) + 2\,Level(x)$
6. $\Psi = \max\{\psi(x) : x \in K \cup I\}$ where $I = \{x : \neg x.rglr\}$.
7. $\alpha(x) = \begin{cases} 1 \text{ if } (\text{Class}(x) = \text{I}) \wedge (x \in I) \\ 1 \text{ if } (\text{Class}(x) = \text{II}) \wedge (x \in I \vee peer(x) \notin I) \\ 2 \text{ if } (\text{Class}(x) = \text{V}) \wedge (x \notin I) \\ 1 \text{ if } (\text{Class}(x) = \text{V}) \wedge (x \in I) \wedge (Peers(x) \cup Prnts(x) \nsubseteq I) \\ 0 \text{ otherwise} \end{cases}$
8. $\phi(x) = 4\,\psi(x) + \mu(x) + \alpha(x)$.
9. $\Phi = \max\{\phi(x) : x \in K \cup I\}$.

The Potential Φ Decreases. In a sequence of lemmas, we prove that Φ decreases. Henceforth, let $X = \{x \in K \cup I : \phi(x) = \Phi\}$.

▶ Remark 14. (a) For any $x \in K \cup I$, $\phi(x)$ does not increase. (b) Φ does not increase.

▶ **Lemma 15.** *If $x \in X$ and $\mu(x) = 0$, then either $\phi(x)$ decreases or $x \notin K \cup I$ during the next round.*

Proof. All descendants of x are strongly regular, since otherwise $\phi(x)$ would not be maximal.
Case 1. Class $(\boldsymbol{x})$ = **I.** If $x \in I$, then x is enabled to execute Line 2, and will do so within one round, decreasing $\alpha(x)$, hence decreasing $\phi(x)$. If x is the sole process at level 0, then $x = r$, hence $\Phi = 0$, contradiction. Otherwise, there is a path σ from x to some process at level 0. This path must contain a process y of Class V whose level is greater than that of x, implying that $\phi(y) > \phi(x)$, contradiction.
Case 2. Class $(\boldsymbol{x})$ = **II.** If $\alpha(x) = 0$, then x will execute Line 5 within one round, becoming completely regular, and hence leaving X. If $\alpha(x) = 1$, either x will execute Line 2 or $peer(x)$ will execute Line 2, or both. After that execution $\alpha(x) = 0$.

Case 3. Class $(\boldsymbol{x})$ = **III.** Then $x \in I$, and x will execute Line 2 within one round, after which $x \notin K \cup I$.
Case 4. Class $(\boldsymbol{x})$ = **IV.** There must be some $y \in \mathit{Peers}(x)$ such that $d(y) > d(x)$. Thus $\phi(y) > \phi(x)$, contradiction.
Case 5. Class $(\boldsymbol{x})$ = **V.** If $\alpha(x) = 2$, then x will execute Line line: regular within one round, after which $\alpha(x) \leqslant 1$. If $\alpha(x) = 1$, then every member of $\mathit{Peers}(x) \cup \mathit{Prnts}(x)$ will execute Line 2, after which $\alpha(x) = 0$. If $\alpha(x) = 0$, x will execute Line 11 within one round, decreasing the value of $\phi(x)$. ◀

Level Actions of Class III Processes. If x is a Class III process, we say that x has *type* 0 if $x.\mathit{level} = 2 + \mathit{parent}(x).\mathit{level}$. Otherwise, we say that x has type 1.

▶ Remark 16. If x is regular, then x is enabled to execute a level action if and only if x has type 1 and all children of x have type 0.

▶ **Lemma 17.** *No two neighboring Class III processes are simultaneously enabled to execute a level action.*

Proof. If two Class III processes are neighbors, one must be the parent of the other. The result then follows immediately from Remark 16. ◀

▶ **Lemma 18.** *If x is a strongly regular process which is enabled to execute a level action, then x will execute that action within the next round.*

Proof. All we need to show is that x cannot be neutralized before it acts. Since x has type 1, $\mathit{parent}(x)$ cannot change its level, and since all children of x have type 0, they also cannot change level. ◀

Chains not Trees. We analyze the evolution of the function θ only for the case that every subtree of regular processes is a chain. Our logic is that the evolution of $\theta(x)$ is determined by the worst case behavior of any chain of T_x. (Henceforth, when we say "chain" of processes we shall always mean a chain that ends at a leaf.)

Bit String Representation of Chains. We replace a chain σ of regular processes by a bit string $w(\sigma)$, obtained by replacing each process by its type. We index the symbols of a string starting from the right; for example, if $w = 01$, then $w_1 = 1$ and $w_2 = 0$. The i^{th} *suffix* of w, $S_i(w)$, is defined to be the suffix of w starting at w_i, *i.e.,* $S_i(w) = w_i w_{i-1} \cdots w_1$. We define θ recursively for both a string, and a symbol within a string, as follows.
1. $\theta(\varepsilon) = 0$, where ε is the empty string.
2. $\theta(0w) = \theta(w) \dot{-} 1$
3. $\theta(1w) = \theta(w) + 1$.
4. $\theta(w, i) = \theta(S_i(w))$.

▶ **Lemma 19** (Monotonicity of θ)**.** *Let w be a bit string.*
(a) *If any 1 in w is replaced by 0, $\theta(w)$ does not increase.*
(b) *If any collection of substrings 10 in w are each replaced by 01, $\theta(w)$ does not increase.*

Proof. Let w and w' be strings such that $\theta(w) \geqslant \theta(w')$. Then

▶ Claim 20. $\theta(0w) \geqslant \theta(0w')$.

▶ Claim 21. $\theta(1w) \geqslant \theta(1w')$.

▶ Claim 22. $\theta(1w) > \theta(0w)$.

▶ Claim 23. $\theta(10w) \geqslant \theta(01w')$.

Claims 20, 21, and 22 follow trivially from the definition of θ. By Claims 20 and 21, we have $\theta(10w) = \theta(w) \dot{-} 1 + 1 \geqslant \theta(w) = \theta(01w) \geqslant \theta(01w')$, proving IV. We have (a) by recursion on $|w|$, using Claims 20, 21, and 22, and (b) by recursion on $|w|$, using 20, 21, and 23. ◀

▶ **Lemma 24.** *If w is a bit string, $\theta(w) > 0$, and w' is obtained from w by replacing each substring 10 by 01, and $w'_1 \leftarrow 0$, then $\theta(w') < \theta(w)$.*

Proof. The proof is by induction. The inductive step consists of a number of cases, each characterized by the values of w_i, w_{i-1}, and w'_{i-1}. There are special cases for $i = 1$. We leave the details of this proof for the full paper. ◀

▶ **Lemma 25.** *If $\Phi > 0$, then Φ decreases during the next round.*

If it were not for the requirement that a process x can only change its level if *Children_Ok*(x) is true, we would be able to prove that Φ decreases every round. What is true is that, at every configuration where $I \neq \varnothing$, every process whose value of ϕ is maximum is enabled, provided that the (annoying) condition *Children_Ok* holds.

The term $\mu(x)$ provides the needed correction. It measures the number of rounds needed before *Children_Ok*(x) becomes true. The coefficient of 4 is needed because of the time needed to ensure μ is again zero on some process of maximum ϕ.

Proof. Let $\Phi > 0$. Let $X = \{x \in K \cup I : \phi(x) = \Phi\}$.

Let $x \in X$. We need to show that $\phi(x)$ decreases within one round. If $\mu(x) = 0$, then $4\psi(x) + \alpha(x)$ decreases, by Lemma 15.

▶ Claim 26. If $\mu(x) = 0$ and x executes a level action, then within one round, $\mu(x)$ does not increase by more than 2.

Sketch of Proof of Claim 26. If x is enabled to execute a level action, and σ is a maximal chain of regular processes ending with a child of x, then $\theta(\sigma) = 0$, which implies that the top member of σ, a child of x, is of type 0, meaning that the bit string $w = w(\sigma)$ starts with 0. When x executes the level action, that 0 is replaced by either 1 or 11, depending on the action. In the worst case, $\theta(w)$ is increased by 3, by Lemmas 19 and 24. ◀

We omit the remaining details of the proof of the lemma. ◀

The Strongly Regular Case. We now consider the case that all processes are strongly regular.

▶ **Lemma 27.** *If all processes are strongly regular, then within $O(diam)$ rounds, all processes are of type 0.*

Proof. We use Lemma 24 and monotonicity of θ, namely Lemma 19. Again simplifying to the case of a chain of length h, the worst case of a strongly regular chain is where each process is of type 1, and a chain is represented by the bit string $111\ldots111$ of length h, where $h \leqslant diam$. Within h rounds the bit string will consist of alternating zeros and ones, and within another h rounds, it will be all zeros. By monotonicity, any other string will become all zeros in the same number of rounds, or fewer. ◀

▶ **Theorem 28.** *The round complexity of* Match *is $O(diam)$.*

Proof Sketch: Each of the terms that makes up the potential Φ is $O(diam)$, and thus by Lemma 25, Φ decreases until $I = \varnothing$ and K consists only of the one process of Class I or the two processes of Class II, within $O(diam)$ rounds.

The configuration now satisfies the conditions of the strongly regular case, and within $O(diam)$ additional rounds, every process is at its final level.

Within $O(diam)$ additional rounds, the flag values converge and processes are matched. The total round complexity of Match is thus $O(diam)$. ◀

7 Step Complexity

In this section, we sketch the proof that Match has step complexity $O(n\, diam)$. We can assume, without loss of generality, that only one process executes at any given step. Henceforth in this section, we assume that each step consists of the execution of one process.

"Star" Notation. We use a star superscript on a variable or function to indicate the value of that variable or function at the end of the current computation. For example, we write $parent^*(x)$ for the value of $parent(x)$ at the final configuration of the computation, and T_x^* for the subtree rooted at x in the final configuration.

We say that a configuration of Match is *aligned* if $parent(x) = y$ implies $parent^*(x) = y$; otherwise we say the configuration is *chaotic*.

We will give the complete proof of the step complexity of Match in the full paper. In this extended abstract, we outline the proof in the aligned case. By Lemma 29, alignment is a closed property.

▶ **Lemma 29.** . *Alignment is a closed property.*

Proof. We only sketch the proof. The configuration is chaotic if and only if there is an *inverted pair*, which we define to be neighboring processes x, y such that $parent(x) = y$ and $parent^*(y) = x$. The only action that can create an inverted pair is Line 11, but careful inspection of this action shows that it can only create an inverted pair if there is already an inverted pair. Thus, an aligned configuration can never become chaotic. ◀

7.1 Regularity Actions

In this subsection, we prove that the total number of executions of Line 2 during a computation of Match which starts at an aligned configuration is $O(n\, diam)$.

Let $Chldrn^*(x) = \{y \in N(x) : parent^*(y) = x\}$, the *eventual children* of x.

▶ **Lemma 30.** *During a computation of* Match *starting from an aligned configuration, the total number of executions of Line 2 is $O(n\, diam)$.*

Proof. We first introduce some potentials.

1. $\varrho(x) \equiv (x.rglr \neq Rglr(x))$, Boolean, meaning that x is enabled to execute Line 2. We write 0 or 1 for the values of ϱ.
2. $\tau(x) = \begin{cases} 1 \text{ if } x.rglr \wedge (\text{Class}(x) = \text{II}) \\ 0 \end{cases}$
3. $\omega(x) = \varrho(x) + \sum \omega(y) : y \in Chldrn^*(x)$
4. $\Omega = \sum \{\omega(x) + \tau(x) : x \in G\}$

▶ Claim 31. $\omega(x) \leqslant |T_x^*|$.

Proof of Claim 31. By induction on $height(T_x^*)$. If $height(T_x^*) = 0$, meaning x is a leaf of the final tree, then $\omega(x) = \varrho(x) \leqslant 1$. Suppose $height(T_x^*) > 0$. By the inductive hypothesis, $\omega(x) = \varrho(x) + \sum\{\omega(y) : y \in Chldrn^*(x)\} \leqslant 1 + \sum\{|T_y^*| : y \in Chldrn^*(x)\} = |T_x^*|$. ◀

Recall that $d(x)$ is the distance, through G, from x to the nearest root r. Clearly, $d(x) \leqslant diam$. For any d, Let $\Omega_d = \sum\{\omega(x) : d(x) = d\}$.

▶ Claim 32. $\Omega_d \leqslant n$ for all d.

Proof of Claim 32. T_x^* and T_y^* are disjoint if $d(x) = d(y)$. Thus $\Omega_d = \sum\{|T_x| : d(x) = d\} \leqslant n$. ◀

▶ Claim 33. $\Omega = O(n\,diam)$.

Proof of Claim 33. By Claim 32 $\Omega = \sum_{d=0}^{diam} \Omega_d + \sum\{\tau(x) : x \in G\} \leqslant n\,(diam + 2)$ ◀

▶ Claim 34. Ω does not increase, and Ω decreases at each step where some process executes Line 2.

Proof of Claim 34. Consider the execution of one action by a process x. The only actions that have an effect on the potentials ϱ, τ, ω, and Ω are those listed in the table below. In each case that x executes Line 2, Ω decreases, while in the one other case listed, an action of Line 5, the value of Ω does not increase. The table below summarizes the effect of each of those actions on the potentials, where Δ indicates the increase of a quantity at the step.

	$\Delta\varrho(x)$	$\Delta\omega(x)$	$\Delta\varrho(y)$	$\Delta\omega(y)$	$\Delta\tau(x)$	$\Delta\Omega$
Class (x) = I; x executes Line 2.	-1	-1			0	-1
Class (x) = II; x executes Line 2; $y = peer(x)$.	-1	-1	0	-1	1	-1
Class (x) = II; x executes Line 5; $y = peer(x)$.	0	0	$\leqslant 1$	$\leqslant 1$	-1	$\leqslant 0$
Class (x) = III; x executes Line 2; $y = parent(x)$.	-1	-1	$\leqslant 1$	$\leqslant 0$	0	$\leqslant -1$

The other actions have no effect on any of the potentials listed above. ◀

The lemma follows from Claims 33 and 34. ◀

▶ **Lemma 35.** *During a computation of* Match *starting from an aligned configuration, the number of steps at which some process executes Line 15 is* $O(n\,diam)$.

We skip the proof of Lemma 35, which is almost identical to the proof of Lemma30. We can show that the step complexity of a computation starting from an aligned computation consists of $O(n\,diam)$ steps, using Lemmas 30 and 35, as well as additional lemmas which we postpone to the full paper. In the full paper, we will prove the step complexity of Match.

8 Conclusion

We have given a self-stabilizing algorithm, under the unfair distributed daemon, for finding a maximum matching of the processes of an anonymous network with a tree topology. Our algorithm runs in $O(diam)$ rounds and $O(n\,diam)$ steps, and needs only $O(1)$ working space per process, that is, space required for intermediate computations.

References

1 Y. Asada and M. Inoue. A silent self-stabilizing algorithm for 1-maximal matching in anonymous networks. In *Proc. WALCOM 2015*, pages 187–198, 2015.

2 J.R.S. Blair and F. Manne. Efficient self-stabilizing algorithms for tree networks. In *Proc. ICDCS 2003*, pages 20–26, 2003.

3 S. Chattopadhyay, L. Higham, and K. Seyffarth. Dynamic and self-stabilizing. In *Proc. PODC 2002*, pages 290–297, 2002.

4 A. K. Datta and L. L. Larmore. Leader election and centers and medians in tree networks. In *Proc. SSS 2013*, pages 113–132, 2013.

5 E.W. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the ACM*, 17:643–644, 1974.

6 W. Goddard, S.T. Hedetniemi, D.P. Jacobs, and P.K. Srimani. Self-stabilizing protocols for maximal matching and maximal independent sets for ad hoc networks. In *Proc. IPDPS 2003*, page 162, 2003.

7 W. Goddard, S.T. Hedetniemi, and Z. Shi. An anonymous self-stabilizing algorithm for 1-maximal matching in trees. In *Proc. PDPTA 2006*, pages 797–803, 2006.

8 N. Guellati and H. Kheddouci. A survey on self-stabilizing algorithms for independence, domination, coloring, and matching in graphs. *Journal of Parallel and Distributed Computing*, 70(4):406–415, 2010.

9 S.T. Hedetniemi, D.P. Jacobs, and P.K. Srimani. Maximal matching stabilizes in time $O(m)$. *Information Processing Letters*, 80(5):221–223, 2001.

10 S.C. Hsu and S.T. Huang. A self-stabilizing algorithm for maximal matching. *Information Processing Letters*, 43(2):77–81, 1992.

11 M.H. Karaata and K.A. Saleh. A distributed self-stabilizing algorithm for finding maximum matching. *Computer Systems Science and Engineering*, 15(3):175–180, 2000.

12 M. Kimoto, T. Tsuchiya, and T. Kikuno. The time complexity of Hsu and Huang's self-stabilizing maximal matching algorithm. *IEICE Trans. Infrmation and Systems*, E93-D(10):2850–2853, 2010.

13 F. Manne, M. Mjelde, L. Pilard, and S. Tixeuil. A new self-stabilizing maximal matching algorithm. In *Proc. SIROCCO 2007*, pages 96–108, 2007.

14 F. Manne, M. Mjelde, L. Pilard, and S. Tixeuil. A self-stabilizing 2/3-approximation algorithm for the maximum matching problem. *Theoretical Computer Science*, 412(4):5515–5526, 2011.

15 G. Tel. Maximal matching stabilizes in quadratic time. *Information Processing Letters*, 49(6):271–272, 1994.