Polynomial Self-Stabilizing Maximum Matching Algorithm with Approximation Ratio 2/3

Johanne Cohen[1], Khaled Maâmra[2], George Manoussakis[3], and Laurence Pilard[4]

1 **LRI-CNRS, Université Paris-Sud, Université Paris Saclay, Paris, France**
johanne.cohen@lri.fr
2 **LI-PaRAD, Université Versailles-St. Quentin, Université Paris Saclay, Paris, France**
khaled.maamra@uvsq.fr
3 **LRI-CNRS, Université Paris-Sud, Université Paris Saclay, Paris, France**
george.manoussakis@lri.fr
4 **LI-PaRAD, Université Versailles-St. Quentin, Université Paris Saclay, Paris, France**
laurence.pilard@uvsq.fr

Abstract

We present the first polynomial self-stabilizing algorithm for finding a $\frac{2}{3}$-approximation of a maximum matching in a general graph. The previous best known algorithm has been presented by Manne *et al.* [16] and has a sub-exponential time complexity under the distributed adversarial daemon [3]. Our new algorithm is an adaptation of the Manne *et al.* algorithm and works under the same daemon, but with a time complexity in $O(n^3)$ moves. Moreover, our algorithm only needs one more boolean variable than the previous one, thus as in the Manne *et al.* algorithm, it only requires a constant amount of memory space (three identifiers and *two* booleans per node).

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Self-Stabilization, Distributed Algorithm, Fault Tolerance, Matching

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2016.11

1 Introduction

Matching problems have received a lot of attention in different areas. Dynamic load balancing and job scheduling in parallel and distributed networks can be solved by algorithms using a matching set of communication links [2, 7]. Moreover, the matching problem has been recently studied in the algorithmic game theory. Indeed, the seminal problem relative to matching introduced by Knuth is the stable marriage problem [13]. This problem can be modeled as a game used in social networks [10] and in wireless networks [18].

In graph theory, a *matching* M in a graph G is a subset of the edges of G without common nodes. A matching is *maximal* if no proper superset of M is also a matching whereas a *maximum* matching is a maximal matching with the highest cardinality among all possible maximal matchings. Some (almost) linear time approximation algorithm for the maximum weighted matching problem have been well studied [6, 17], nevertheless these algorithms are not distributed. They are based on a simple greedy strategy using *augmenting path*. An *augmenting path* is a path, 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. Let us consider the example in Figure 2d, page 11. In

this figure, u and v are matched nodes and x, y are unmatched nodes. The path (x, u, v, y) is an augmenting path of length 3 (written 3-*augmenting path*). It is well known [11] that given a graph $G = (V, E)$ and a matching $M \subseteq E$, if there is no augmenting path of length $2k - 1$ or less, then M is a $\frac{k}{k+1}-$approximation of the maximum matching. See [6] for the weighted version of this theorem. The greedy strategy in [6, 17] consists in finding all augmenting paths of length ℓ or less and by switching matched and unmatched edges of these paths in order to improve the maximum matching approximation.

In this paper, we present a self-stabilizing algorithm for finding a maximum matching with approximation ratio 2/3 that uses the greedy strategy presented above. Our algorithm stabilizes after $O(n^3)$ moves under the adversarial distributed daemon.

In unweighted or weighted general graphs, self-stabilizing algorithms for computing maximal matching have been designed in various models (anonymous network [1] or not [19], see [8] for a survey). For an unweighted graph, Hsu and Huang [12] gave the first self-stabilizing algorithm and proved a bound of $O(n^3)$ on the number of moves under a sequential adversarial daemon. Hedetniemi *et al.* [9] completed the complexity analysis proving a $O(m)$ move complexity. Manne *et al.* [15] gave a self-stabilizing algorithm that converges in $O(m)$ moves under a distributed adversarial daemon. Note that it is well known that a maximal matching is only an $\frac{1}{2}$-approximation of a maximum matching.

Manne *et al.* [16] and Asada and Inoue [1] presented some self-stabilizing algorithms for finding a $\frac{2}{3}$-approximation of a maximum matching. Manne *et al.* gave an exponential upper bound on the stabilization time ($O(2^n)$ moves under a distributed adversarial daemon) of their algorithm. However, they didn't show that this upper bound is tight. We proved [3] that this lower bound is sub-exponential by exhibiting an execution of $\Omega(2^{\sqrt{n}})$ moves before stabilization. Asada and Inoue [1] gave a polynomial algorithm but under the adversarial sequential daemon.

In a weighted graph, Manne and Mjelde [14] presented the first self-stabilizing algorithm for computing a weighted matching of a graph with an $\frac{1}{2}$-approximation of the optimal solution. They established that their algorithm stabilizes after at most an exponential number of moves under any adversarial daemon (*i.e.*, sequential or distributed). Turau and Hauck [19] gave a modified version of the previous algorithm that stabilizes after $O(nm)$ moves under any adversarial daemon.

The following figure compares features of the aforementioned algorithms and our result. The previous best known algorithm working under the same daemon has a sub-exponential complexity while our algorithm has a cubic complexity.

Problem	$\frac{1}{2}$-approximation (maximal matching)		$\frac{2}{3}$-approximation		
Graph	Identified		Anonymous without cycle with multiple of 3 length	Identified	
Daemon	Adversarial Sequential	Adversarial Distributed	Adversarial Sequential	Adversarial Distributed	
Reference	[12, 9]	[15]	[1]	[16]	**This paper**
Complexity	$O(m)$ moves	$O(m)$ moves	$O(m)$ moves	$\Omega(2^{\sqrt{n}})$ moves	$\mathbf{O(n^3)}$ **moves**

In the rest of the document, we present the model (Section 2), the algorithm (Section 3) and then the correction proof (Section 4) followed by the convergence proof (Section 5).

2 Model

The system consists of a set of processes where two adjacent processes can communicate with each other. The communication relation is represented by an undirected graph $G = (V, E)$ where $|V| = n$ and $|E| = m$. Each process corresponds to a node in V and two processes u and v are adjacent if and only if $(u, v) \in E$. The set of neighbors of a process u is denoted by $N(u)$ and is the set of all processes adjacent to u, and Δ is the maximum degree of G. We assume all nodes in the system have a unique identifier.

For the communication, we consider the *shared memory model*. In this model, each process maintains a set of *local variables* that makes up the *local state* of the process. A process can read its local variables and the local variables of its neighbors, but it can write only in its own local variables. A *configuration* C is the local states of all processes in the system. Each process executes the same algorithm that consists of a set of *rules*. Each rule is of the form of <*name*> :: **if** <*guard*> **then** <*command*>. The *name* is the name of the rule. The *guard* is a predicate over the variables of both the process and its neighbors. The *command* is a sequence of actions assigning new values to the local variables of the process.

A rule is *activable* in a configuration C if its guard in C is true. A process is *eligible* for the rule $\mathcal{R}$ in a configuration C if its rule $\mathcal{R}$ is activable in C and we say the process is *activable* in C. An *execution* is an alternate sequence of configurations and actions $\mathcal{E} = C_0, A_0, \ldots, C_i, A_i, \ldots$, such that $\forall i \in \mathbb{N}^*$, C_{i+1} is obtained by executing the command of at least one rule that is activable in C_i (a process that executes such a rule makes a *move*). More precisely, A_i is the non empty set of activable rules in C_i that has been executed to reach C_{i+1} and such that each process has at most one of its rules in A_i. We use the notation $C_i \mapsto C_{i+1}$ to denote this transition in $\mathcal{E}$. Finally, let $\mathcal{E}' = C'_0, A'_0, \cdots, C'_k$ be a finite execution. We say $\mathcal{E}'$ is a *sub-execution* of $\mathcal{E}$ if and only if $\exists t \geq 0$ such that $\forall j \in [0, \cdots, k]$:$(C'_j = C_{j+t} \wedge A'_j = A_{j+t})$.

An *atomic operation* is such that no change can take place during its run, we usually assume that an atomic operation is instantaneous. In the shared memory model, a process u can read the local state of all its neighbors and update its whole local state in one atomic step. Then, we assume here that a rule is an atomic operation. An execution is *maximal* if it is infinite, or it is finite and no process is activable in the last configuration. All algorithm executions considered here are assumed to be maximal.

A *daemon* is a predicate on the executions. We consider only the most powerful one: the *adversarial distributed daemon* that allows all executions described in the previous paragraph. Observe that we do not make any fairness assumption on the executions.

An algorithm is *self-stabilizing* for a given specification, if there exists a sub-set $\mathcal{L}$ of the set of all configurations such that: every execution starting from a configuration of $\mathcal{L}$ verifies the specification (*correctness*) and starting from any configuration, every execution eventually reaches a configuration of $\mathcal{L}$ (*convergence*). $\mathcal{L}$ is called the set of *legitimate configurations*. A configuration is *stable* if no process is activable in the configuration. The algorithm presented here, is *silent*, meaning that once the algorithm has stabilized, no process is activable. In other words, all executions of a silent algorithm are finite and end in a stable configuration. Note the difference with a non silent self-stabilizing algorithm that has at least one infinite execution with a suffix only containing legitimate configurations, but not stable ones.

3 Algorithm

The algorithm presented in this paper is called MaxMatch, and is based on the algorithm presented by Manne *et al.* [16]. As in the Manne *et al.* algorithm, MaxMatch assumes there

exists an underlying maximal matching algorithm, which has reached a stable configuration. Based on this stable maximal matching, MaxMatch builds a $\frac{2}{3}$-approximation of the maximum matching by detecting and then deleting all 3-augmenting paths. Once a 3-augmenting path is detected, nodes rearrange the matching accordingly, *i.e.*, transform this path with one matched edge into a path with two matched edges. This transformation leads to the deletion of the augmenting path and increases by one the cardinality of the matching. The algorithm stabilizes when there is no augmenting path of length three left. By the result of Hopcroft *et al.* [11], we obtain a $\frac{2}{3}$-approximation of the maximum matching.

This underlying stabilized maximal matching can be built, for instance, with the self-stabilizing maximal matching algorithm from [15] that stabilizes in $O(m)$ moves under the adversarial distributed daemon (so the same daemon than the one used in this paper). Observe that this algorithm is silent, meaning that the maximal matching remains constant once the algorithm has stabilized. Then, using a classical composition of these two algorithms [5], we obtain a total time complexity in $O(n^2 \times n^3) = O(n^5)$ moves under the adversarial distributed daemon.

In the rest of the paper, $\mathcal{M}$ is the underlying maximal matching, and $\mathcal{M}^+$ is the set of edges built by our algorithm MaxMatch (see Definition 3.1). For a set of nodes A, we define $single(A)$ and $matched(A)$ as the set of unmatched and matched nodes in A, accordingly to the underlying maximal matching $\mathcal{M}$. Moreover, $\mathcal{M}$ is encoded with the variable m_u. If $(u, v) \in \mathcal{M}$ then u and v are *matched nodes* and we have: $m_u = v \wedge m_v = u$. If u is not incident to any edge in $\mathcal{M}$, then u is a *single node* and $m_u = null$. Since we assume the underlying maximal matching is stable, a node membership in $matched(V)$ or $single(V)$ will not change, and each node u can use the value of m_u to determine which set it belongs to.

Variables description: In order to facilitate the rematching, each node $u \in V$ maintains three pointers and two boolean variables. The pointer p_u refers to a neighbor of u that u is trying to (re)match with. If $p_u = null$ then the matching of u has not changed from the maximal matching. Thus, the matching $\mathcal{M}^+$ built by our algorithm is defined as follows:

▶ **Definition 3.1.** The set of edges built by algorithm MaxMatch is $\mathcal{M}^+ = \{(u, v) \in \mathcal{M} : p_u = p_v = null\} \cup \{(a, b) \in E \setminus \mathcal{M} : p_a = b \wedge p_b = a\}$

For a matched node u, pointers α_u and β_u refer to two nodes in $single(N(u))$ that are *candidates* for a possible rematching with u. Also, s_u is a boolean variable that indicates if the node u has performed a successful rematching with its single node candidate. Finally, end_u is a boolean variable that indicates if both u and m_u have performed a successful rematching or not. For a single node x, only one pointer p_x and one boolean variable end_x are needed. p_x has the same purpose as the p-variable of a matched node. The *end*-variable of a single node allows the matched nodes to know whether it is *available* or not. A single node is *available* if it is possible for this node to eventually rematch with one of its neighboring married node, *i.e.*, end_x =*False*.

In our algorithm, $Unique(A)$ returns the number of unique elements in the multi-set A, and $Lowest(A)$ returns the node in A with the lowest identifier. If $A = \emptyset$, then $Lowest(A)$ returns $null$. Moreover, rules have priorities. In the algorithm, we present rules from the highest priority (at the top) to the lowest one (at the bottom).

Algorithm description: Every pair of matched nodes u, v (v=m_u) tries to find single neighbors they can rematch with. Moreover u and v need to have a sufficient number of available single neighbors to detect a 3-augmenting path: each node should have at least

——— **Rules for each node u in single($\mathbf{V}$)**

ResetEnd ::
if $p_u = null \wedge end_u = True$
then $end_u := False$

UpdateP ::
if $(p_u = null \wedge \{w \in matched(N(u)) \mid p_w = u\} \neq \emptyset) \vee (p_u \notin (matched(N(u)) \cup \{null\})) \vee$
$(p_u \neq null \wedge p_{p_u} \neq u)$
then $p_u := Lowest\{w \in N(u) \mid p_w = u\}$
$end_u := False$

UpdateEnd ::
if $(p_u \in matched(N(u)) \wedge (p_{p_u} = u) \wedge (end_u \neq end_{p_u})$
then $end_u := end_{p_u}$

——— **Predicates and functions**

BestRematch$(u) \equiv$
$a = Lowest\{x \in single(N(u)) \wedge (p_x = u \vee end_x = False)\}$
$b = Lowest\{x \in single(N(u)) \setminus \{a\} \wedge (p_x = u \vee end_x = False)\}$
return (a, b)

AskFirst$(u) \equiv$
if $\alpha_u \neq null \wedge \alpha_{m_u} \neq null \wedge 2 \leq Unique(\{\alpha_u, \beta_u, \alpha_{m_u}, \beta_{m_u}\})$
then if $(\alpha_u < \alpha_{m_u}) \vee (\alpha_u = \alpha_{m_u} \wedge \beta_u = null) \vee (\alpha_u = \alpha_{m_u} \wedge \beta_{m_u} \neq null \wedge u < m_u)$
then return α_u
else return $null$

AskSecond$(u) \equiv$
if $AskFirst(m_u) \neq null$
then return $Lowest(\{\alpha_u, \beta_u\} \setminus \{\alpha_{m_u}\})$
else return $null$

——— **Rules for each node u in matched($\mathbf{V}$)**

Update ::
if $(\alpha_u > \beta_u) \vee (\alpha_u, \beta_u \notin (single(N(u)) \cup \{null\})) \vee (\alpha_u = \beta_u \wedge \alpha_u \neq null)$
$\vee p_u \notin (single(N(u)) \cup \{null\}) \vee$
$((\alpha_u, \beta_u) \neq BestRematch(u) \wedge (p_u = null \ \vee \ (p_{p_u} \neq u \ \wedge \ end_{p_u} = True)))$
then $(\alpha_u, \beta_u) := BestRematch(u)$
$(p_u, s_u, end_u) := (null, False, False)$

MatchFirst ::
if $(AskFirst(u) \neq null) \wedge$
$[\ p_u \neq AskFirst(u) \vee$
$s_u \neq (p_u = AskFirst(u) \wedge p_{p_u} = u \wedge p_{m_u} \in \{AskSecond(m_u), null\}) \vee$
$end_u \neq (p_u = AskFirst(u) \wedge p_{p_u} = u \wedge s_u \wedge p_{m_u} = AskSecond(m_u) \wedge end_{m_u})\ ]$
then $end_u := (p_u = AskFirst(u) \wedge p_{p_u} = u \wedge s_u \wedge p_{m_u} = AskSecond(m_u) \wedge end_{m_u})$
$s_u := (p_u = AskFirst(u) \wedge p_{p_u} = u \wedge (p_{m_u} \in \{AskSecond(m_u), null\})$
$p_u := AskFirst(u)$

MatchSecond ::
if $(AskSecond(u) \neq null) \wedge (s_{m_u} = True) \wedge$
$[\ p_u \neq AskSecond(u) \vee end_u \neq (p_u = AskSecond(u) \wedge p_{p_u} = u \wedge p_{m_u} = AskFirst(m_u))$
$\vee s_u \neq end_u\ ]$
then $end_u := (p_u = AskSecond(u) \wedge p_{p_u} = u \wedge p_{m_u} = AskFirst(m_u))$
$s_u := end_u$
$p_u := AskSecond(u)$

ResetMatch ::
if $[(AskFirst(u) = AskSecond(u) = null) \wedge ((p_u, s_u, end_u) \neq (null, False, False))] \vee$
$[AskSecond(u) \neq null \wedge p_u \neq null \wedge s_{m_u} = False]$
then $(p_u, s_u, end_u) := (null, False, False)$

one candidate and the sum of the number of candidates for u and v should be at least 2. The *BestRematch* predicate is used to compute candidates in variables α and β, and the condition $(\alpha_u \neq null \wedge \alpha_v \neq null \wedge 2 \leq Unique(\{\alpha_u, \beta_u, \alpha_v, \beta_v\}))$, from predicate *AskFirst*, is used to ensure the number of candidates is sufficiently high.

Observe that we only have three distinct possible values for the quadruplet $(AskFirst(u), AskSecond(u), AskFirst(v), AskSecond(v))$ for any couple $(u, v) \in \mathcal{M}$ and whatever the α and β values are. These are: $(null, null, null, null)$ or $(x, null, null, y)$ or $(null, x, y, null)$, with $x \neq y$. The first case means that there is no 3-augmenting path that contains the couple (u, v). The two other cases mean that (x, u, v, y) is a 3-augmenting path. If $x < y$, we are in the second case, otherwise we are in the third case. Node u is said to be *First* if $AskFirst(u) \neq null$. In the same way, u is *Second* if $AskSecond(u) \neq null$.

In the following, we consider the second case, since the third case is symmetric of the second case with v is *First*. So we assume u is *First*.

A 3-augmenting path is exploited in three phases. The first phase determines the first edge of the 3-augmenting path. This phase is complete when $s_u = True$ and $p_{p_u} = u$ (u is *First*). In the second phase, the third edge of the augmenting path is marked. This phase is complete when node v, that is *Second*, sets its *end* value to *True*. Finally, in the third phase, the *end* value of v is propagated in the whole augmenting path. When this propagation is done, the phase is over, the augmenting path is said to be *fully exploited* and all neighbors of single nodes of this path know that these two single nodes are not available anymore.

The scenario for an augmenting path exploitation when everything goes well is given in the following. Node u starts trying to rematch with x performing a *MatchFirst* move and $p_u := x$. If x accepts the proposition, performing an *UpdateP* move and $p_x := u$, then u will inform v of this first phase success, once again by performing a *MatchFirst* move and $s_u :=$*True*. Observe that at this point, x cannot change its p-value since $p_{p_x} = x$. Finally, node v tries to rematch with y, performing a *MatchSecond* move and $p_v := y$. If y accepts the proposition, performing an *UpdateP* move and $p_y := v$, then v will inform u of this final success, by performing a *MatchSecond* move and $end_v :=$*True*. This complete the second phase. From then, all nodes in this 3-augmenting path will set there *end*-variable to *True*: u by performing a last *MatchFirst* move, and x and y by performing an *UpdateEnd* move. From this point, non of nodes x, u, v, or y will ever be eligible for any move again. Moreover, once single nodes have their *end*-variables set to *True*, they are not available anymore for any other matched nodes.

Rules description: The *Update* rule allows a matched node to update its α and β variables. Then, predicates *AskFirst* and *AskSecond* are used to define the role the node will have in the 3-augmenting path exploitation. If the node is *First* (resp. *Second*), then it will execute *MatchFirst* (resp. *MatchSecond*) several times for this 3-augmenting path exploitation.

The *MatchFirst* rule is used by the node when it is *First*. Let u be this node. The first time this rule is performed, u seduces its candidate setting (end_u, s_u, p_u) to $(False, False, AskFirst(u))$. Then this rule is performed a second time after the u's candidate has accepted the u's proposition, *i.e.*, when $AskFirst(u)$ has set its p-variable to u. So the second *MatchFirst* execution sets (end_u, s_u, p_u) to $(False, True, AskFirst(u))$. Now, variable s_u is equal to $True$, allowing node m_u that is *Second* to seduce its own candidate. Finally, the rule *MatchFirst* is performed a third time when m_u completed is own rematch, *i.e.*, when $end_u = True$. When there is no bad information due to some bad initializations, $end_{m_u} = True$ means that $p_{m_u} = AskSecond(m_u) \wedge p_{p_{m_u}} = m_u$. So this third *MatchFirst* execution sets (end_u, s_u, p_u) to $(True, True, AskFirst(u))$, meaning that the 3-augmenting path has been fully exploited.

In the *MatchFirst* rule, observe that we make the s_u affectation before the p_u affectation, because the s_u value must be computed accordingly to the value of p_u before activating u. Indeed, when u executes *MatchFirst* for the first time, it allows to set p_u from $\perp$ to $AskFirst(u)$ while s_u remains $False$. Then when u executes *MatchFirst* for the second time, s_u is set from $False$ to $True$ while p_u remains equal to $AskFirst(u)$. For the same argument, we make the end_u affectation before the s_u affectation. Thus, the «normal» values sequence for (p_u, s_u, end_u) is: $((\perp, False, False), (AskFirst(u), False, False), (AskFirst(u), True, False), (AskFirst(u), True, True))$.

The *MatchSecond* rule is used by the node when it is *Second*. This rule is performed only twice in one 3-augmenting path exploitation. For the first execution, u has to wait for m_u to set its s_{m_u} to *True*. Then u can perform *MatchSecond* and update its p-variable to $AskSecond(u)$. When the u's candidate has accepted to this proposition, u can perform *MatchSecond* for the second time, setting s_u and end_u to *True*. As in the *MatchFirst* rule, we set *end* and s affectation before the p affectation.

The *ResetMatch* rule is performed to reset bad initialization and its consequences.

Let us now consider rules for single nodes. The *ResetEnd* rule is used to reset bad initializations. In the *UpdateP* rule, the node updates its p-value according to the propositions done by neighboring matched nodes. If there is no proposition, the node sets its p-value to $\perp$. Otherwise, p is set to the minimum identifier among all proposals. Afterward, the p-value can only change when the proposition is canceled. When a single node u has accepted a proposition, its end value should be equal to the end value of p_u. The *UpdateEnd* rule is used for this purpose.

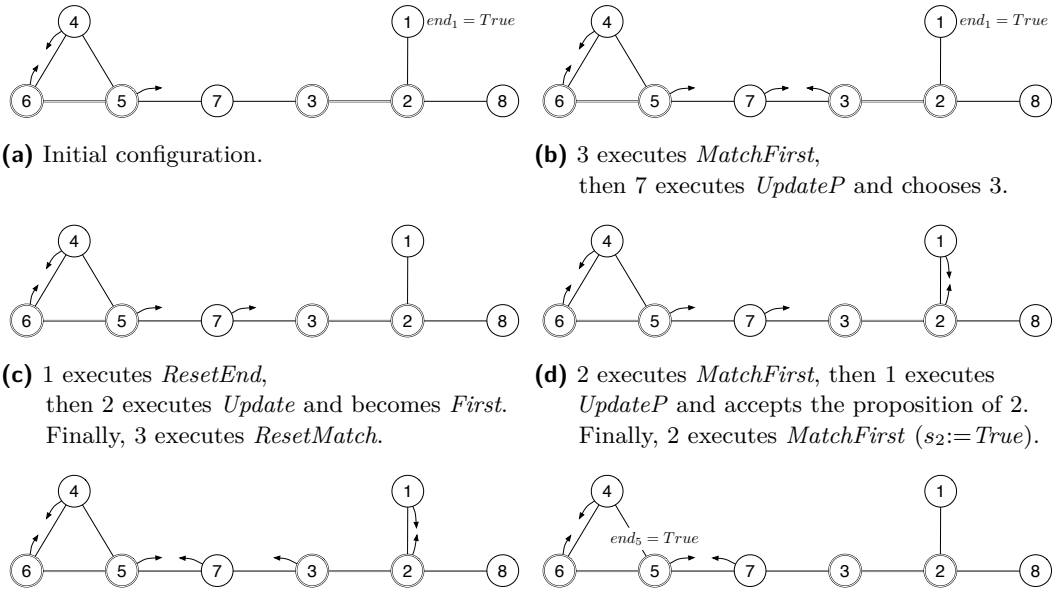
Graphical convention: We will follow the above conventions in all the figures: matched nodes are represented with double circles and single nodes with simple circles. Moreover, all edges that belong to the maximal matching $\mathcal{M}$ are represented with a double line, whereas the other edges are represented with a simple line. Black arrows show the content of the local variable p. If the p-value is *null*, we draw a 'T'. A prohibited value is first drawn in grey, then scratched out in black. If there is no knowledge on the p-value, nothing is drawn. For instance, in Figure 2e, page 11, x is a single node, u and v are matched nodes and $(u, v) \in \mathcal{M}$, $p_u = x$, and $p_x \neq u$. In Figure 2d page 11, $p_u = \perp$.

3.1 Execution example

Now, we give a possible execution of Algorithm MaxMatch under the adversarial distributed daemon. Figure 1a shows the initial state of the execution. Node identifiers are indicated inside the circles. The underlying maximal matching contains two edges $(5, 6)$ and $(2, 3)$. Then nodes 2, 3, 5 and 6 are *matched* nodes and nodes 1, 4, 7 and 8 are *single* nodes. At the beginning, there are three 3-augmenting paths: $(4, 6, 5, 7), (7, 3, 2, 1)$ and $(7, 3, 2, 8)$.

The initial configuration (Figure 1a): In the initial configuration, we assume that all α-*values* and β-*values* are defined as follows: $(\alpha_2, \beta_2) = (8, null)$, $(\alpha_3, \beta_3) = (7, null)$, $(\alpha_5, \beta_5) = (4, 7)$, and $(\alpha_6, \beta_6) = (4, null)$. We also assume all s-*values* are well defined ($s_6 =$*True* while all other s-values are *False*) whereas all *end*-values are *False* but end_1 that is *True*. At this moment, node 2 considers that since $end_1 =$*True*, node 1 already belongs to a fully exploited 3-augmenting path: $BestRematch(2) = (8, null)$.

Nodes 6 and 5 have already started to exploit their augmenting path: $p_6 = 4$, $p_4 = 6$ and $p_5 = 7$. Node 6 is *First* because $\beta_6 = null$ and since $s_6 =$*True*, node 5 knows that it can start its exploitation too, performing a *MatchSecond*: $p_5 = 7$. At this step, node 5 waits for



(a) Initial configuration.

(b) 3 executes *MatchFirst*, then 7 executes *UpdateP* and chooses 3.

(c) 1 executes *ResetEnd*, then 2 executes *Update* and becomes *First*. Finally, 3 executes *ResetMatch*.

(d) 2 executes *MatchFirst*, then 1 executes *UpdateP* and accepts the proposition of 2. Finally, 2 executes *MatchFirst* (s_2:=*True*).

(e) 3 and 7 execute in parallel *MatchSecond* and *UpdateP* respectively.

(f) 5 executes *MatchSecond*, then the *True* value of end_5 is propagated in the path $(4, 6, 5, 7)$. Finally the path $(7, 3, 2, 1)$ is destroyed.

Figure 1 An execution of Algorithm MaxMatch (Only the *True* value of the *end*-variables are given).

an answer of node 7. There are only two kinds of answer: node 7 accepts to take part of this path exploitation setting $p_7 = 5$ with an *UpdateP* rule, or it refuses setting $end_7 =$*True* while $p_7 \neq 5$ with an *UpdateEnd* rule. But this last choice can only be done if 7 belongs to another fully exploited augmenting path. So at this point, node 7 cannot refuse.

The other 3-augmenting path is $(7, 3, 2, 8)$. Node 2 considers that node 1 is not available because $end_1 =$*True*. Since $2 \leq Unique(\{\alpha_2, \beta_2, \alpha_3, \beta_3\}) \leq 4$, nodes 2 and 3 detect a 3-augmenting path and start to exploit it. Since node 3 is *First* ($AskFirst(3) = 7$ and $AskFirst(2) = null$), node 3 may execute a *MatchFirst* move. Let us assume it does.

The 3-augmenting path exploitation starts (Figure 1b): Node 3 executes here a *MatchFirst* move and points to node 7. Since both nodes 3 and 5 are pointing to node 7, node 7 can choose the node to match with from these two nodes. Note that at this point, node 7 is the only activable node among all nodes except node 1. Node 7 makes this choice by executing an *UpdateP* move: since $Lowest\{u \in N(7) \mid p_u = 7\} = 3$, node 7 points to node 3.

Difference with Manne *et al.* algorithm: In our algorithm, even after 7 has chosen 3, node 5 still waits for an acceptation of node 7, and will do so while end_7 remains *False*. However, at this point, in Manne *et al.* algorithm, node 5 can destroy the augmenting-path construction. This is the main difference that allows our algorithm to prevent from exponential executions.

So, at this point there is a binary choice for node 5: destroy or not its augmenting-path construction. In Manne *et al.* algorithm, the choice is to destroy, thus the destruction of a partially exploited augmenting-path can be done while no fully exploited augmenting path has been built. Moreover, for one fully exploited augmented path, we can exhibit some executions where we destroy a sub-exponential number of exploited augmented-path [3]. In

our algorithm, we do the other choice which is: do not destroy while there is still hope to exploit the augmenting path. So, if node 5 breaks a partially exploited augmenting path, then node 7 belongs to a fully exploited augmenting-path. Thus the destruction of 5 implies one 3-augmenting path has been fully exploited and thus the matching size has been increased by 1.

This difference is implemented in the algorithm through the *BestRematch* predicate. The condition $p_x = null$ in Manne *et al.* algorithm has been replaced by the condition $end_x = False$ in our algorithm. Then, in our algorithm, $BestRematch(5)$ remains constant when 7 chooses node 3, while it does not in Manne *et al.* algorithm, making 5 eligible for *Update*.

Go back to the execution, node 1 wakes up (Figure 1c): Let us focus on node 1. Its *end*-value is not well defined since $end_1 = True$ while node 1 does not belong to a fully exploited augmenting path. Thus, node 1 is eligible for *ResetEnd* rule. After this activation, $end_1 = False$. This implies that $BestRematch(2) = (1, 8)$ and thus $(\alpha_2, \beta_2) = (8, null) \neq BestRematch(2)$. So, node 2 is eligible for *Update* rule. Let us assume it makes this move. Thus, after this activation, node 2 is *First*. This implies that node 3 is *Second*, and it is eligible for *ResetMatch* because $AskSecond(3) \neq null \wedge p_3 \neq null \wedge s_2 = False$. So, it does it and sets $p_3 = null$.

A second 3-augmenting path exploitation starts (Figure 1d): Let us consider node 2. It is *First* and it can execute a *MatchFirst* rule. After this activation, it sets $p_2 = 1$ and $s_2 = end_2 = False$. Now, node 1 accepts the node 2 proposition by applying *UpdateP*. After this activation, node 1 points to node 2 ($p_1 = 2$). Now, node 2 is eligible for executing a *MatchFirst* rule. It sets $p_2 = 1$ and $s_2 = True$. This implies that node 3 becomes eligible for *MatchSecond*.

A matched node proposition in parallel with a single node abandonment (Figure 1e): In the configuration shown in Figure 1d, only nodes 3 and 7 are activable, node 3 can propose to node 7 with a *MatchSecond* and node 7 can accept the node 5's proposition with an *UpdateP*. Assume 3 and 7 are activated at the same time. Figure 1e shows the configuration obtained after theses moves: $p_3 = 7$, $p_7 = 5$. Note that after these activations, we have $s_3 = False$ since, *before* these activations, the *p*-values of nodes 3 and 7 are not as follow: $p_3 = 7$ and $p_7 = 3$. This kind of transitions, where a matched node proposition is performed in parallel with a single node abandonment, is the reason why we make the *s*-affectation, then the *p*-affectation in the *MatchFirst* rule. This trick allows to obtain after a *MatchFirst* rule: $s_u = True$ implies $p_{p_u} = u$. Finally, observe at this step that node 3 still waits for an answer of node 7.

The path (4,6,5,7) becomes fully exploited (Figure 1f): Since $end_5 \neq (p_5 = AskSecond(5) \wedge p_7 = 5 \wedge p_6 = AskFirst(6))$, node 5 is eligible for a *MatchSecond* rule to set end_5 to *True* and then to make the other nodes aware that the path is fully exploited. Assume node 5 executes a *MatchSecond* move. This will cause node 7 (resp. 6) to execute an *UpdateEnd* move (resp. a *MatchFirst* move) and sets $end_7 = True$ (resp. $end_6 = True$). Now, it is the turn to node 4 to execute an *UpdateEnd* move. As the *end*-value of nodes 4, 5, 6, and 7 are equal to *True*, the 3-augmenting path is fully exploited.

Recall that node 3 was waited for an answer from node 7. Now, $end_7 = True \wedge p_7 \neq 3$. Thus node 7 is not available for node 3 anymore and so node 3 executes the *Update* rule:

$(\alpha_3, \beta_3) = (null, null)$. This will cause node 2 to execute a *ResetMatch* move ($p_2 = null$) and then node 1 to execute an *UpdateP* move ($p_1 = null$). The system has reached a stable configuration (see Fig. 1e). Thus, the size of the matching is increasing by one and there is no 3-augmenting path left.

Now, we present the proof of our algorithm.

4 Correctness Proof

In all the proofs, if a lemma, a theorem, or a corollary is labeled with a capital letter, then the associated proof is in [4].

A natural way to prove the correction of MaxMatch algorithm could have been to follow the approach below. We consider a stable configuration C in MaxMatch and we prove C is also stable in the Manne *et al.* algorithm. As we use the exact same variables but the *end*-variable and because the matching is only defined on the common variables, the correctness follows from Manne *et al.* paper. Moreover, we can easily show that if C is stable in MaxMatch, then no rule from the Manne *et al.* algorithm but the *Update* rule can be performed in C. Unfortunately, it is not straightforward to prove that the *Update* rule from Manne *et al.* algorithm cannot be executed in C. Indeed, our *Update* rule is more difficult to execute than the one of Manne *et al.* in the sens that some possible *Update* in Manne *et al.* are not possible in our algorithm. By the way, this is why our algorithm has a better time complexity since the number of partially exploited augmented path destruction in our algorithm is smaller than in the Manne *et al.* algorithm. In particular, we have to prove that in a stable configuration, for any matched node, if $p_u \neq null$, then $end_{p_u} = True$. To prove that, we need Lemmas A, B, C, D, E and a part of the proof from Theorem 4.1. Observe that from these results, the correctness is straightforward without using the Manne *et al.* proof.

Let $Ask : V \to V \cup \{null\}$ be a function where $Ask(u) = AskFirst(u)$ if $AskFirst(u) \neq null$, otherwise $Ask(u) = AskSecond(u)$. We will say a node makes a *match* rule if it performs a *MatchFirst* or *MatchSecond* rule.

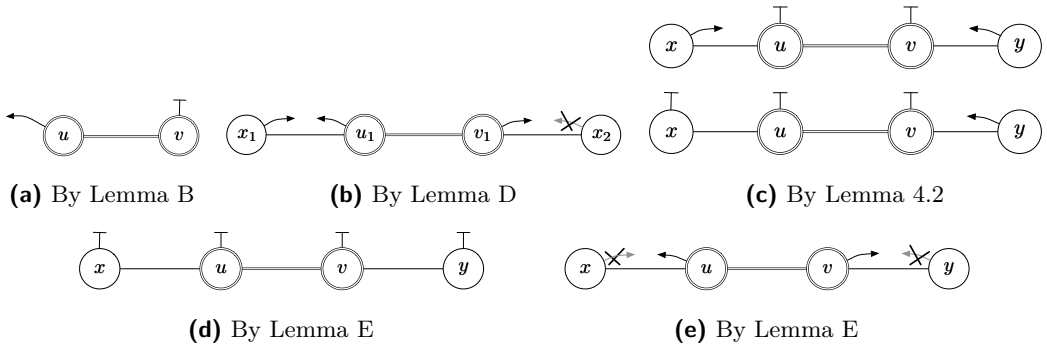
For the correctness part, we prove that in a stable configuration, $\mathcal{M}^+$ is a 2/3-approximation of a maximum matching on graph G. To do that we demonstrate there is no 3-augmenting path on $(G, \mathcal{M}^+)$. In particular we prove that for any edge $(u, v) \in \mathcal{M}$, we have either $p_u = p_v = null$, or u and v have two distinct single neighbors they are rematched with, *i.e.*, $\exists x \in single(N(u)), \exists y \in single(N(v))$ with $x \neq y$ such that $(p_x = u) \wedge (p_u = x) \wedge (p_y = v) \wedge (p_v = y)$. In order to prove that, we show every other case for (u, v) is impossible. Main studied cases are shown in Figure 2. Finally, we prove that if $p_u = p_v = null$ then (u, v) does not belong to a 3-augmenting-path on $(G, \mathcal{M}^+)$.

▶ **Lemma A.** *In any stable configuration, we have the following properties:*

- $\forall u \in matched(V) : p_u = Ask(u)$;
- $\forall x \in single(V)$ *: if* $p_x = u$ *with* $u \neq null$*, then* $u \in matched(N(x)) \wedge p_u = x \wedge end_u = end_x$.

▶ **Lemma B.** *Let (u, v) be an edge in $\mathcal{M}$. Let C be a configuration. If $p_u \neq null \wedge p_v = null$ holds in C (see Fig. 2a), then C is not stable.*

▶ **Lemma C.** *Let (x, u, v, y) be a 3-augmenting path on $(G, \mathcal{M})$. Let C be a stable configuration. In C, if $p_x = u$, $p_u = x$, $p_v = y$ and $p_y = u$, then $end_x = end_u = end_v = end_y = True$.*



(a) By Lemma B **(b)** By Lemma D **(c)** By Lemma 4.2

(d) By Lemma E **(e)** By Lemma E

Figure 2 3-augmenting paths that are not possible in a stable configuration.

▶ **Lemma D.** *Let (x_1, u_1, v_1, x_2) be a 3-augmenting path on $(G, \mathcal{M})$. Let C be a configuration. If $p_{x_1} = u_1 \wedge p_{u_1} = x_1 \wedge p_{v_1} = x_2 \wedge p_{x_2} \neq v_1$ holds in C (see Fig. 2b), then C is not stable.*

Proof. This is a sketch of the proof. We can prove that there exists a 3-augmenting path (x_2, u_2, v_2, x_3) such that $p_{x_2} = u_2 \wedge p_{u_2} = x_2 \wedge p_{v_2} = x_3 \wedge p_{x_3} \neq v_2$. This augmenting path has the exact same properties than the first considered augmenting path (x_1, u_1, v_1, x_2) and in particular u_1 is *First*. Now we can continue the construction in the same way. Therefore, for C to be stable, there must to exist a chain of 3-augmenting paths $(x_1, u_1, v_1, x_2, u_2, v_2, x_3, \ldots, x_i, u_i, v_i, x_{i+1}, \ldots)$ where $\forall i \geq 1 : (x_i, u_i, v_i, x_{i+1})$ is a 3-augmenting path with $p_{x_i} = u_i \wedge p_{u_i} = x_i \wedge p_{v_i} = x_{i+1} \wedge p_{x_{i+1}} = v_{i+1}$ and u_i is *First*. Thus, $x_1 < x_2 < \ldots < x_i < \ldots$ since the u_i will always be *First*. Since graph G is finite some x_k must be equal to some x_ℓ with $\ell \neq k$ which contradicts the fact that the identifier's sequence is strictly increasing. ◀

▶ **Lemma E.** *Let (x, u, v, y) be a 3-augmenting path on $(G, \mathcal{M})$. Let C be a configuration. If $(p_u = x \wedge p_x \neq u \wedge p_v = y \wedge p_y \neq v)$ or if $(p_y = p_u = p_v = p_y = null)$ holds in C (see Fig. 2d and Fig. 2e respectively), then C is not stable.*

▶ **Theorem 4.1.** *In a stable configuration we have, $\forall (u, v) \in \mathcal{M}$:*

- $p_u = p_v = null$ *or*
- $\exists x \in single(N(u)), \exists y \in single(N(v))$ *with* $x \neq y$ *such that* $(p_x = u) \wedge (p_u = x) \wedge (p_y = v) \wedge (p_v = y)$.

Proof. We will prove that all cases but these two are not possible in a stable configuration. First, Lemma B says the configuration cannot be stable if one of p_u or p_v is not *null*.

Second, assume that $p_u \neq null \wedge p_v \neq null$. Let $p_u = x$ and $p_v = y$. Observe that $x \in single(N(u))$ (resp. $y \in single(N(v))$), otherwise u (resp. v) is eligible for *Update*.

Case $x \neq y$: If $p_x \neq u$ and $p_y \neq v$ then Lemma E says the configuration cannot be stable. If $p_x = u$ and $p_y \neq v$ then Lemma D says the configuration cannot be stable. Thus, the only remaining possibility when $p_u \neq null$ and $p_v \neq null$ is: $p_x = u$ and $p_y = v$.

Case $x = y$: $Ask(u)$ (resp. $Ask(v) \neq null$), otherwise u (resp. v) is eligible for a *ResetMatch*. W.l.o.g. let us assume that u is *First*. $x = AskFirst(u)$ (resp. $x = AskSecond(v)$), otherwise u (resp. v) is eligible for *MatchFirst* (resp. *MatchSecond*). Thus $AskFirst(u) = AskSecond(v)$ which is impossible according to these two predicates. ◀

▶ **Lemma 4.2.** *Let x be a single node. In a stable configuration, if $p_x = u, u \neq null$ (see Fig. 2c) then it exists a 3-augmenting path (x, u, v, y) on $(G, \mathcal{M})$ such that $p_x = u \wedge p_u = x \wedge p_v = y \wedge p_y = v$.*

Proof. By Lemma A, if $p_x = u$ with $u \neq null$ then $u \in matched(N(x))$ and $p_u = x$. Since $p_u \neq null$, by Theorem 4.1 the result holds. ◀

Thus, in a stable configuration, for all edges $(u, v) \in \mathcal{M}$, if $p_u = p_v = null$ then (u, v) does not belong to a 3-augmenting-path on $(G, \mathcal{M}^+)$. In other words, we obtain:

▶ **Corollary 4.3.** *In a stable configuration, there is no 3-augmenting path on* $(G, \mathcal{M}^+)$ *left.*

5 Convergence Proof

This section is devoted to a sketch of the convergence proof. In the following, μ will denote the number of matched nodes and σ the number of single nodes.

The first step consists in proving that the values of s and *end* represent the different phases of the path exploitation. Recall that $s_u = True$ means $p_{p_u} = u$. Moreover $end_u = True$ means that the path is fully exploited. We can easily prove that after one activation of a matched node u, $s_u = True$ implies $p_{p_u} = u$ (Lemma F). However, a bad initialization of end_{m_u} to *True* can induce u to wrongly write *True* in end_u. But this can appear only once and thus, the second times u writes *True* in end_u means that a 3-augmenting path involving u has been fully exploited (Theorem 5.1).

▶ **Lemma F.** *Let* u *be a matched node. Consider an execution* $\mathcal{E}$ *starting after* u *executed some rule. Let* C *be any configuration in* $\mathcal{E}$*. In* C*, if* $s_u = True$ *then* $\exists x \in single(N(u)) : p_u = x \wedge p_x = u$*.*

▶ **Theorem 5.1.** *In any execution, a matched node* u *can write* $end_u := True$ *at most twice.*

We now count the number of destruction of partially exploited augmenting paths. Recall that in Manne *et al.* algorithm, for one fully exploited augmenting path, it is possible to destroy a sub-exponential number of partially exploited ones.

In our algorithm, observe that for a path destruction, the set of single neighbors that are candidates for a matched edge has to change and this change can only occur when a single node changes its *end*-value. Such a change induces a path destruction if a matched node takes into account this modification by applying an *Update* rule. So, we first count the number of time a single node can change its *end*-value (Lemma G) and then we deduce the number of time a matched node can execute *Update* (Corollary H). Finally, we conclude we destroy at most $O(n^2)(= O(\Delta(\sigma + \mu)))$ partially exploited augmenting path.

The rest of the proof consists in counting the number of moves that can be performed between two *Update*, allowing us to conclude the proof (Theorem I).

In the following, we detailed point by point the idea behind each result cited above.

Since single nodes just follow orders from their neighboring matched nodes, we can count the number of times single nodes can change the value of their *end* variable. There are σ possible modifications due to bad initializations. A matched node u can write *True* twice in end_u, so end_u can be *True* during 3 distinct sub-executions. As a single node x copies the *end*-value of the matched node it points to ($p_x = u$), then a single node can change its *end*-value at most 3 times as well. And we obtain 6μ modifications.

▶ **Lemma G.** *In any execution, the number of transitions where a single node changes the value of its end variables (from True to False or from False to True) is at most* $\sigma + 6\mu$ *times.*

We count the maximal number of *Update* rule that can be performed in any execution. To do that, we observe that the first line of the *Update* guard can be *True* at most once in an execution (Lemma L). Then we prove for the second line of the guard to be *True*, a single node has to change its *end* value. Thus, for each single node modification of the *end*−value, at most all matched neighbors of this single node can perform an *Update* rule.

▶ **Corollary H.** *Matched nodes can execute at most* $\Delta(\sigma + 6\mu) + \mu$ *times the* Update *rule.*

Third, we count the maximal number of moves performed by matched nodes between two *Update*. The idea is that in an execution without *Update*, α and β values of all matched nodes remain constant. Thus, in these small executions, at most one augmenting path is detected per matched edge and at most one rematch attempt is performed per matched edge. We obtain that the maximal number of moves of a matched node in these small executions is 12. By the previous remark and Corollary H, we obtain:

▶ **Theorem I.** *In any execution, matched nodes can execute at most* $12\mu(\Delta(\sigma + 6\mu) + \mu)$ *rules.*

Finally, we count the maximal number of moves that single nodes can perform, counting rule by rule. The *ResetEnd* is done at most once. The number of *UpdateEnd* is bounded by the number of times single nodes can change their *end*-value, so it is at most $\sigma + 6\mu$. Finally, *UpdateP* is counted as follows: between two consecutive *UpdateP* executed by a single node x, a matched node has to make a move. The total number of executed *UpdateP* is then at most $12\mu(\Delta(\sigma + 6\mu) + \mu) + 1$.

▶ **Corollary J.** *The algorithm* MaxMatch *converges in* $O(n^3)$ *steps under the adversarial distributed daemon.*

Due to the lack of space, we cannot give the whole convergence proof. We choose to present the proof of Theorem 5.1 since it is the key point of the convergence proof. Indeed, with this result, we have a first strong step leading to the proof of the silent property of our algorithm. The remaining of this section is devoted to prove Theorem 5.1. The rest of the proof is placed in [4].

5.1 A matched node can write *True* in its *end*-variable at most twice

The first two lemmas are technical lemmas.

▶ **Lemma K.** *Let* u *be a matched node. Consider an execution* $\mathcal{E}$ *starting after* u *executed some rule. Let* C *be any configuration in* $\mathcal{E}$*. If* $end_u =$*True in* C *then* $s_u =$*True as well.*

▶ **Lemma L.** *Let* u *be a matched node and* $\mathcal{E}$ *be an execution containing a transition* $C_0 \mapsto C_1$ *where* u *makes a move. From* C_1*, the predicate in the first line of the guard of the* Update *rule will never hold from* C_1*.*

Now, we will focus on particular configurations for a matched edge (u, v) corresponding to the fact they have completely exploited a 3-augmenting path.

▶ **Lemma 5.1.** *Let* (u, v) *be a matched edge,* $\mathcal{E}$ *be an execution and* C *be a configuration of* $\mathcal{E}$*. If in* C*, we have:*
1. $p_u \in single(N(u)) \wedge p_u = AskFirst(u) \wedge p_{p_u} = u$*;*
2. $p_v \in single(N(v)) \wedge p_v = AskSecond(v) \wedge p_{p_v} = v$*;*
3. $s_u = end_u = s_v = end_v =$*True;*

then neither u *nor* v *will ever be eligible for any rule from* C*.*

Proof. Observe first that neither u nor v are eligible for any rule in C. Moreover, p_u (resp. p_v) is not eligible for an *UpdateP* move since u (resp. v) does not make any move. Thus p_{p_u} and p_{p_v} will remain constant since u and v do not make any move and so neither u nor v will ever be eligible for any rule from C. ◀

The configuration C described in Lemma 5.1 is called a configuration $stop_{uv}$. From such a configuration neither u nor v will ever be eligible for any rule.

In Lemmas 5.4 and M, we consider executions where a matched node u writes *True* in end_u twice, and we focus on the transition $C_0 \mapsto C_1$ where u performs its second writing. Lemma 5.4 shows that, if u is *First* in C_0, then C_1 is a $stop_{um_u}$ configuration. Lemma M shows that, if u is *Second* in C_0, then either C_1 is a $stop_{um_u}$ configuration or it exists a configuration C_3 such that $C_3 > C_1$ and u does not make any move from C_1 to C_3 and C_3 is a $stop_{um_u}$ configuration.

Lemma 5.2 and Corollary 5.3 are required in order to prove Lemmas 5.4 and M.

▶ **Lemma 5.2.** *Let (u, v) be a matched edge. Let $\mathcal{E}$ be some execution in which v does not execute any rule. If it exists a transition $C_0 \mapsto C_1$ in $\mathcal{E}$ where u writes True in end_u, then u is not eligible for any rule from C_1.*

Proof. To write *True* in end_u in transition $C_0 \mapsto C_1$, u must have executed a *match* rule. According to this rule, $(p_u = Ask(u) \wedge p_{p_u} = u)$ holds C_0 with $p_u \in single(N(u))$, otherwise u would have executed an *Update* instead of a *match* rule. Now, in $C_0 \mapsto C_1$, p_u cannot execute *UpdateP* then it cannot change its p-value and v does not execute any move then it cannot change $Ask(u)$. Thus, $(p_u = Ask(u) \wedge p_{p_u} = u)$ holds in both C_0 and C_1.

Assume now by contradiction that u executes a rule after configuration C_1. Let $C_2 \mapsto C_3$ be the next transition in which it executes a rule. Recall that between configurations C_1 and C_2 both u and v do not execute rules. Observe also that p_u is not eligible for *UpdateP* between these configurations. Thus $(p_u = Ask(u) \wedge p_{p_u} = u)$ holds from C_0 to C_2. Moreover the following points hold as well between C_0 and C_2 since in $C_0 \mapsto C_1$ u executed a *match* rule and v does not apply rules in $\mathcal{E}$:

- α_u, α_v, β_u and β_v do not change.
- The values of the variables of v do not change.
- $Ask(u)$ and $Ask(v)$ do not change.
- If u was *First* in C_0 it is *First* in C_2 and the same holds if it was *Second*.

Using these remarks, we start by proving that u is not eligible for *ResetMatch* in C_2. If it is *First* in C_2, this holds since $AskFirst(u) \neq null$ and $AskSecond(u) = null$. If it is *Second* then to be eligible for *ResetMatch*, $s_v =$*False* must hold in C_2 since $AskSecond(u) \neq null$. Since u executed $end_u =$*True* in $C_0 \mapsto C_1$ and since u was *Second* in C_0, then necessarily $s_v =$*True* in C_0 and thus in C_2 (using remark 2 above). So u is not eligible for *ResetMatch* in C_2.

We show now that u is not eligible for an *Update* in C_2. The α and β variables of u and v remain constant between C_0 and C_2. Thus if any of the three first disjunctions in the *Update* rule holds in C_2 then it also holds in C_0 and in $C_0 \mapsto C_1$ u should have executed an *Update* since it has higher priority than the *match* rules. Moreover since in C_2 $(p_u = Ask(u) \wedge p_{p_u} = u)$ holds, the last two disjunctions of *Update* are *False* and we can state u is not eligible for this rule.

We conclude the proof by showing that u is not eligible for a *match* rule in C_2. If u was *First* in C_0 then it is *First* in C_2. To write *True* in end_u then $(p_u = AskFirst(u) \wedge p_{p_u} = u \wedge s_u \wedge p_{m_u} = AskSecond(m_u) \wedge end_{m_u})$ must hold in C_0. Since in $C_0 \mapsto C_1$ v does not

execute rules, it also holds in C_1. The same remark between configurations C_1 and C_2 implies that this predicate holds in C_2. Thus in C_2, all the three conditions of the *MatchFirst* guard are *False* and u not eligible for *MatchFirst*. A similar remark if u is *Second* implies that u will not be eligible for *MatchSecond* in C_2 if it was *Second* in C_0. ◀

▶ **Corollary 5.3.** *Let (u, v) be a matched edge. In any execution, if u writes True in end_u twice, then v executes a rule between these two writing.*

▶ **Lemma 5.4.** *Let (u, v) be a matched edge and $\mathcal{E}$ be an execution where u writes True in its variable end_u at least twice. Let $C_0 \mapsto C_1$ be the transition where u writes True in end_u for the second time in $\mathcal{E}$. If u is* First *in C_0 then the following holds:*

1. *in configuration C_0,*
   - **(a)** $s_v = end_v = True$;
   - **(b)** $p_u = AskFirst(u) \wedge p_{p_u} = u \wedge s_u = True \wedge p_v = AskSecond(v)$;
   - **(c)** $p_u \in single(N(u))$;
   - **(d)** $p_v \in single(N(v)) \wedge p_{p_v} = v$;
2. *v does not execute any move in $C_0 \mapsto C_1$;*
3. *in configuration C_1,*
   - **(a)** $s_u = end_u = True$;
   - **(b)** $p_u \in single(N(u)) \wedge p_v \in single(N(v))$;
   - **(c)** $s_v = end_v = True$;
   - **(d)** $p_u = AskFirst(u) \wedge p_v = AskSecond(v)$;
   - **(e)** $p_{p_u} = u \wedge p_{p_v} = v$.

Proof. We prove Point 1a. Observe that for u to write *True* in end_u, end_v must be *True* in C_0. By Lemma K this implies that s_v is *True* as well. Now Point 1b holds by definition of the *MatchFirst* rule. As in C_0, u already executed an action, then according to Lemma L, Point 1c holds and will always hold. By Corollary 5.3, u cannot write *True* consecutively if v does not execute moves. Thus at some point before C_0, v applied some rule. This implies that in configuration C_0, since $s_v = True$, by Lemma F, $\exists x \in single(N(v)) : p_v = x \wedge p_x = v$. Thus Point 1d holds.

We now show that v does not execute any move in $C_0 \mapsto C_1$ (Point 2). Recall that v already executed an action before C_0, so by Lemma L, line 1 of the *Update* guard does not hold in C_0. Moreover, by Point 1d, line 2 does not hold either. Thus, v is not eligible for *Update* in C_0. We also have that $s_u = True$ and $AskSecond(v) \neq null$ in C_0, thus v is not eligible for *ResetMatch*. Observe now that by Points 1a, 1b and 1d, v is not eligible for *MatchSecond* in C_0. Finally v cannot execute *MatchFirst* since $AskFirst(v) = null$. Thus v does not execute any move in $C_0 \mapsto C_1$ and so Point 2 holds.

In C_1, end_u is *True* by hypothesis and according to Point 1b, u writes *True* in s_u in transition $C_0 \mapsto C_1$. Thus Point 3a holds. Points 3b holds by Points 1c and 1d. Points 3c holds by Points 1a and 2. $AskFirst(u)$ and $AskSecond(v)$ remain constant in $C_0 \mapsto C_1$ since neither u nor v executes an *Update* in this transition. Moreover p_v remains constant in $C_0 \mapsto C_1$ by Point 2 and p_u remains constant also since it writes $AskFirst(u)$ in p_u in this transition while $p_u = AskFirst(u)$ in C_0. Thus Points 3d holds. Observe that nor p_u neither p_v is eligible for an *UpdateP* in C_0, thus Point 3e holds. ◀

Now, we consider the case where u is *Second*.

▶ **Lemma M.** *Let (u, v) be a matched edge and $\mathcal{E}$ be an execution where u writes True in its variable end_u at least twice. Let $C_0 \mapsto C_1$ be the transition where u writes True in end_u for the second time in $\mathcal{E}$. If u is* Second *in C_0 then the following holds:*

1. *in configuration* C_0*:*
   (a) $s_v = True \wedge p_v = AskFirst(v)$ *and*
   (b) $p_v \in single(N(v)) \wedge p_{p_v} = v$
2. *in transition* $C_0 \mapsto C_1$*,* v *is not eligible for* Update *nor* ResetMatch*;*
3. *in configuration* C_1*,*
   (a) $s_u = end_u = True$ *and*
   (b) $p_v \in single(N(v)) \wedge p_v = AskFirst(v) \wedge p_{p_v} = v$ *and*
   (c) $p_u \in single(N(u)) \wedge p_u = AskSecond(u) \wedge p_{p_u} = u$ *and*
   (d) $s_v = True$*;*
4. u *is not eligible for any move in* C_1*;*
5. *If* $end_u = False$ *in* C_1 *then the following holds:*
   (a) *From* C_1*,* v *executes a next move and this move is a* MatchFirst*;*
   (b) *Let us assume this move (the first move of* v *from* C_1*) is done in transition* $C_2 \mapsto C_3$*. In configuration* C_3*, we have:*
      (i) $s_u = end_u = True$ *and*
      (ii) $p_v \in single(N(v)) \wedge p_v = AskFirst(v) \wedge p_{p_v} = v$ *and*
      (iii) $p_u \in single(N(u)) \wedge p_u = AskSecond(u) \wedge p_{p_u} = u$ *and*
      (iv) $s_v = True$ *and*
      (v) u *does not execute moves between* C_1 *and* C_3 *and*
      (vi) $end_v = True$*.*

▶ **Theorem 5.1.** *In any execution, a matched node* u *can write* $end_u := True$ *at most twice.*

Proof. Let (u, v) be a matched edge and $\mathcal{E}$ be an execution where u writes *True* in its variable end_u at least twice. Let $C_0 \mapsto C_1$ be the transition where u writes *True* in end_u for the second time in $\mathcal{E}$. If u is *First* (resp. *Second*) in C_0 then from Lemmas 5.1 and 5.4, (resp. Lemma M), from C1, neither u nor v will ever be eligible for any rule. ◀

References

1 Y. Asada and M. Inoue. An efficient silent self-stabilizing algorithm for 1-maximal matching in anonymous networks. In *WALCOM: Algorithms and Computation – 9th International Workshop*, pages 187–198. Springer International Publishing, 2015.

2 P. Berenbrink, T. Friedetzky, and R. A. Martin. On the stability of dynamic diffusion load balancing. *Algorithmica*, 50(3):329–350, 2008. doi:10.1007/s00453-007-9081-y.

3 J. Cohen, K. Maamra, G. Manoussakis, and L. Pilard. The Manne *et al.* self-stabilizing 2/3-approximation matching algorithm is sub-exponential. *CoRR*, abs/1604.08066, 2016. URL: http://arxiv.org/abs/1604.08066.

4 J. Cohen, K. Maamra, G. Manoussakis, and L. Pilard. Polynomial self-stabilizing algorithm and proof for a 2/3-approximation of a maximum matching. *CoRR*, abs/1611.06038, 2016. URL: http://arxiv.org/abs/1611.06038.

5 S. Dolev. *Self-Stabilization.* MIT Press, 2000.

6 D. E. Drake and S. Hougardy. A simple approximation algorithm for the weighted matching problem. *Inf. Process. Lett.*, 85(4):211–213, 2003.

7 B. Ghosh and S. Muthukrishnan. Dynamic load balancing by random matchings. *J. Comput. Syst. Sci.*, 53(3):357–370, 1996. doi:10.1006/jcss.1996.0075.

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

9 S. T. Hedetniemi, D. Pokrass Jacobs, and P. K. Srimani. Maximal matching stabilizes in time o(m). *Inf. Process. Lett.*, 80(5):221–223, 2001.

10 M. Hoefer. Local matching dynamics in social networks. *Inf. Comput.*, 222:20–35, 2013.

11 J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

12 S.-C. Hsu and S.-T. Huang. A self-stabilizing algorithm for maximal matching. *Inf. Process. Lett.*, 43(2):77–81, 1992.

13 D. Knuth. *Marriages stables et leurs relations avec d'autres problèmes combinatoires*. Les Presses de l'Université de Montréal, 1976.

14 F. Manne and M. Mjelde. A self-stabilizing weighted matching algorithm. In *9th Int. Symposium Stabilization, Safety, and Security of Distributed Systems (SSS)*, Lecture Notes in Computer Science, pages 383–393. Springer, 2007.

15 F. Manne, M. Mjelde, L. Pilard, and S. Tixeuil. A new self-stabilizing maximal matching algorithm. *Theoretical Computer Science (TCS)*, 410(14):1336–1345, 2009.

16 F. Manne, M. Mjelde, L. Pilard, and S. Tixeuil. A self-stabilizing 2/3-approximation algorithm for the maximum matching problem. *Theoretical Computer Science (TCS)*, 412(40):5515–5526, 2011.

17 R. Preis. Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In *16th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, Lecture Notes in Computer Science, pages 259–269. Springer, 1999.

18 M. Touati, R. El-Azouzi, M. Coupechoux, E. Altman, and J. M. Kelif. Controlled matching game for user association and resource allocation in multi-rate wlans? In *2015 53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 372–380, Sept 2015. doi:10.1109/ALLERTON.2015.7447028.

19 V. Turau and B. Hauck. A new analysis of a self-stabilizing maximum weight matching algorithm with approximation ratio 2. *Theoretical Computer Science (TCS)*, 412(40):5527–5540, 2011.