# The first polynomial self-stabilizing 1-maximal matching algorithm for general graphs

Johanne Cohen, Jonas Lefèvre, Khaled Maamra, George Manoussakis, Laurence Pilard

Version of Record: https://www.sciencedirect.com/science/article/pii/S0304397519301379
Manuscript_b85c6e750cf55847b0dc9a24f3f048b4

# The first polynomial self-stabilizing 1-maximal matching algorithm for general graphs.

Johanne Cohen[a], Jonas Lefèvre[b], Khaled Maâmra[c], George Manoussakis[a], Laurence Pilard[c]

[a] *LRI-CNRS, Université Paris-Sud, Université Paris Saclay, France,* {johanne.cohen, george.manoussakis}@lri.fr
[b] *IRIF, Université Paris-Diderot – Paris 7 , France,* {jonas.lefevre}@irif.fr
[c] *LI-PaRAD, Université Versailles-St. Quentin, Université Paris Saclay, France,* {khaled.maamra, laurence.pilard}@uvsq.fr

**Abstract**

We present the first polynomial self-stabilizing algorithm for finding a 1-maximal matching in a general graph. The previous best known algorithm has been presented by Manne *et al.* [20] and we show in this paper it has a sub-exponential time complexity under the distributed adversarial daemon. Our new algorithm is an adaptation of the Manne *et al.* algorithm and works under the same daemon, but with a complexity in $O(m \times n^2)$ moves, with $n$ is the number of nodes and $m$ is the number of edges. This is the first self-stabilizing algorithm that solve this problem with a polynomial complexity. Moreover, our algorithm only needs one more boolean variable than the previous one.

*Keywords:* Self-stabilization, 1-maximal matching, $\frac{2}{3}$-approximation.

## 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, 9]. In the wireless network, the resource management can be modelized as matching problem between resources and users (see [11] for a survey).

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. A matching $M$ is *1-maximal* if it satisfies the following property: $\forall e \in M$, no matching can be constructed by removing $e$ from $M$ and adding two edges to $M \setminus \{e\}$. A 1-maximal matching is a $\frac{2}{3}$-approximation to the maximum matching, and expected to get more matching pairs than a maximal matching, which only guarantees a $\frac{1}{2}$-approximation. In the following, $n$ is the number of nodes and $m$ is the number of edges in $G$.

Some (almost) linear time approximation algorithm for the maximum weighted matching problem have been well studied [8, 21], 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 2.(a). 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 [13] 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 [8] for the weighted version of this theorem. The greedy strategy in [8, 21] consists in finding all augmenting paths of length $\ell$ 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 1-maximal matching that uses the greedy strategy presented above. Our algorithm stabilizes after $O(m \times n^2)$ moves under the adversarial distributed daemon.

For the maximum matching problem, self-stabilizing algorithms have been designed for particular topologies. In anonymous tree networks, a self-stabilizing algorithms converging in $O(n^4)$ moves under the sequential adversarial daemon is given by Karaata and Saleh [17]. Recently, Datta *et al.* [6] improve this result, and give a silent self-stabilizing protocol that converges in $O(n^2)$ moves. For anonymous bipartite networks, a self-stabilizing algorithms converging in $O(n^2)$ rounds under the sequential daemon is given by Chattopadhyay et al. [3].

In unweighted or weighted general graphs, self-stabilizing algorithms for computing maximal matching have been designed in various models (anonymous network [1] or not [22], see [10] for a survey). For an unweighted graph, Hsu and Huang [14] 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.* [12] completed the complexity analysis proving a $O(m)$ move complexity. Manne *et al.* [19] gave a self-stabilizing algorithm that converges in $O(m)$ moves under a distributed adversarial daemon. Cohen *et al.* [4] extend this result and propose a randomized self-stabilizing algorithm for computing a maximal matching in an anonymous network. The complexity is $O(n^2)$ moves with high probability, under the adversarial distributed daemon.

Manne *et al.* [20] and Asada and Inoue [1] presented some self-stabilizing algorithms for finding a 1-maximal matching. Manne *et al.* gave an exponential upper bound on the stabilization time of their algorithm ($O(2^n)$ moves under a distributed adversarial daemon). However, they didn't show that this upper bound is tight. In this paper, we prove this lower bound is sub-exponential by exhibiting an execution of $\Omega(2^{\sqrt{n/2}})$ moves before stabilization. Asada and Inoue [1] gave a polynomial algorithm but under the adversarial sequential daemon. Recently, Inoue *et al.* [15] gave a modified version of [1] that stabilizes after $O(m)$ moves under the distributed adversarial daemon for networks without cycle whose length is a multiple of three.

In a weighted graph, Manne and Mjelde [18] 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 [22] gave a modified version of the previous algorithm that stabilizes after $O(nm)$ moves under any adversarial daemon.

Figure 1 compares features of the aforementioned algorithms and our result.

We are then interested in the following problem: how to efficiently build a 1-maximal matching in an identified graph with a general topology, using an adversarial distributed daemon and in a self-stabilizing way? In this paper, we present two algorithms solving this problem. The first one is the well-known algorithm from Manne *et al.* [20] that was the only one until now that solved this problem. The second algorithm is our contribution. We show that the Manne *et al.* algorithm reaches a sub-exponential complexity while we prove that our algorithm is polynomial (in $O(m \times n^2)$). This paper is an extended version of the conference paper [5], where we present our polynomial algorithm (but with a sketch of the proof only). In [5], we obtained a $O(n^3)$ moves assuming an already built maximal matching. In this paper, under the same assumption, we obtain a $O(n^2)$ moves. Thus, as we will develop this scheme in Sections 3 and 8, using a classical composition [7] of the self-stabilizing maximal matching algorithm given by Manne *et al.* [19] and of our algorithm, we obtain a $O(m \times n^2)$ move complexity. This result has been improved after this article submission in [16].

In the rest of the document, we present the model (Section 2), then we give the strategy based on a 3-augmenting path deletion that is used to build a 1-maximal matching (Section 3). This strategy is used by both algorithms presented next. In Section 4, we precisely describe the Manne *et al.* algorithm [20] and present the proof of the existence of a sub-exponential execution in Section 5. Next, we give our polynomial algorithm in Section 6, its correctness proof in Section 7 followed by its convergence proof in Section 8.

## 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 $\Delta$ is the maximum degree of $G$. We assume all nodes in the system a have distance 3 unique identifier.

| Matching | Topology | Identifiers | Daemon | Complexity (moves) | Work |
|---|---|---|---|---|---|
| Maximum | Tree<br>Bipartite | Global<br>Anonymous | Adver. Sequential | $O(n^2)$<br>$O(n^2)$ rounds | [17, 6]<br>[3] |
| Maximal | Arbitrary | Global | Adver. Sequential<br>Adver. Distributed | $O(m)$<br>$O(m)$ | [14, 12]<br>[19] |
| | | Anonymous | Adver. Sequential<br>Adver. Distributed | $O(n^2)$<br>$O(n^2)$ whp | [14]<br>[4] |
| 1-Maximal | Arbitrary without cycle with multiple of 3 length | Anonymous | Adver. Sequential<br>Adver. Distributed | $O(m)$<br>$O(m)$ | [1]<br>[15] |
| | Arbitrary | Unique at distance 3 | Adver. Distributed | $\mathbf{\Omega(2^{\sqrt{n}})}$<br>$\mathbf{O(m.n^2)}$ | [20]<br>**Here** |

Figure 1: Best results in maximum matching approximation. In bold, our contributions.

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 $C_i \mapsto C_{i+1}$ or $C_i \overset{A_i}{\longmapsto} C_{i+1}$ 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 $\forall j \in [0, \cdots, k]$:$(C'_j = C_{j+t} \wedge A'_j = A_{j+t})$.

If $C$ and $C'$ are two configurations in $\mathcal{E}$, then we note $C \leq C'$ if and only if $C$ appears before $C'$ in $\mathcal{E}$ or if $C = C'$. Moreover, we write $\mathcal{E}\backslash C$ to denote all configurations of $\mathcal{E}$ except configuration $C$.

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. Common strategy to build a 1-maximal matching

In this paper, we present two algorithms. The first one, denoted by ExpoMatch, is the Manne *et al.* algorithm [20]. The second one, called PolyMatch, is the main contribution of this paper. These two

algorithms share different elements and this section is devoted to give these main common points.

Both algorithms operate on an undirected graph, where every node has a distance 3 unique identifier. They also assume an adversarial distributed daemon and that there exists an already built maximal matching, noted $\mathcal{M}$. Based on $\mathcal{M}$, the two algorithms build a 1-maximal matching. To perform that, nodes search and delete any 3-augmenting paths they find in $\mathcal{M}$. An augmenting path is a path in the graph, starting and ending in an unmatched node, and where every other edge is either unmatched or matched.

**Definition 1.** *Let $G = (V, E)$ be a graph and $M$ be a maximal matching of $G$. $(x, u, v, y)$ is a* 3-augmenting path *on $(G, M)$ if: (i) $(x, u, v, y)$ is a path in $G$ (so all nodes are distincts); (ii) $\forall a \in V : (x, a) \notin M \wedge (y, a) \notin M$; and (iii) $(u, v) \in M$.*

Let us consider the example in Figure 2.(a). In this figure, $u$ and $v$ are matched nodes and $x, y$ are unmatched nodes. The path $(x, u, v, y)$ is a 3-augmenting path. Once an 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 (see Figure 2.(b)). This transformation leads to the deletion of the augmenting path and increases by one the cardinality of the matching. Both algorithms will stabilize when there are no augmenting paths of length three left. Thus the hypothesis of Karps's theorem [13] eventually holds, giving a $\frac{2}{3}-$approximation of the maximum matching (and so a 1-maximal matching).


(a) A 3-augmenting path (one matched edge). (b) The path after being exploited (two matched edges).

Figure 2: How to exploit a 3-augmenting path?

*The underlying maximal matching.* In the rest of the paper, $\mathcal{M}$ is the underlying maximal matching. This underlying matching is locally expressed by variables $m_v$ for each node $v$. 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 $m_u = null$. For a set of nodes $A$, we define $single(A)$ and $matched(A)$ as the set of single and matched nodes in $A$, accordingly to the underlying maximal matching $\mathcal{M}$. Since we assume $\mathcal{M}$ to be stable, a node membership in $matched(V)$ or $single(V)$ will not change throughout an execution, and each node $u$ can use the value of $m_u$ to determine which set it belongs to.

Note that $\mathcal{M}$ can be built with any silent self-stabilizing maximal matching algorithm that works for general graph and with an adversarial distributed daemon. We can then use, for instance, the self-stabilizing maximal matching algorithm from [19] that stabilizes in $O(m)$ moves. Observe that this algorithm is silent, meaning that the maximal matching remains constant once the algorithm has stabilized.

*2-phases algorithms.* Both algorithms ExpoMatch and PolyMatch are based on two phases for each edge $(u, v)$ in $\mathcal{M}$: (1) *detecting* augmenting paths and (2) *exploiting* the detected augmenting paths. Node $u$ keeps track of four variables. The pointer $p_u$ is used to define the final matching. The variables $\alpha_u, \beta_u$ are used to detect augmenting paths and contain single neighbors of $u$. Also, $s_u$ is a boolean variable used for the augmenting path exploitation. We will see in section 6 that algorithm PolyMatch uses a fifth variable named $end_u$. In the rest of the paper, we will call $\mathcal{M}^+$ the final 1-maximal matching built by any of the two algorithms. $\mathcal{M}^+$ is defined as follows:

**Definition 2.** *The built set of edges is:*

$$\mathcal{M}^+ = \{(u, v) \in \mathcal{M} : p_u = p_v = null\} \cup \{(a, b) \in E \setminus \mathcal{M} : p_a = b \wedge p_b = a\}$$

The first set in the union is pairs of nodes that do not perform any rematch. These pairs come from $\mathcal{M}$. The second set in the union is pairs of nodes that were not matched together in $\mathcal{M}$, but after a 3-augmenting path detection and exploitation, they matched together.

*Augmenting path detection.* First, every pair of matched nodes $u$, $v$ ($v$=$m_u$ and $u$=$m_v$) tries to find single neighbors they can rematch with. These single neighbors have to be *available*, in particular, they should not be married in a final way with another matched node. We will see in the next sections, that the

meaning of being available is not the same in PolyMatch and ExpoMatch. We say that a single node $x$ is a *candidate* for a matched node $u$ if $x$ is an available single neighbor of $u$. Note that $u$ and $v$ need to have a sufficient number of candidates to detect a 3-augmenting path: each node should have at least one candidate and the sum of the number of candidates for $u$ and $v$ should be at least 2. In both algorithms, the *BestRematch* predicate is used to compute candidates of a matched node $u$, writing in $\alpha_u$ and $\beta_u$. Then, the condition below is used in both algorithms – in the *AskFirst* predicate – to ensure the number of candidates is sufficiently high to detect if $u$ belongs to a 3-augmenting path.

$$\alpha_u \neq null \wedge \alpha_{m_u} \neq null \wedge 2 \leq Unique(\{\alpha_u, \beta_u, \alpha_{m_u}, \beta_{m_u}\}) \leq 4$$

where $Unique(A)$ returns the number of unique elements in the multi-set $A$.

*Augmenting path exploitation.* The exploitation is done in a sequential way. First, two nodes matched together $u$ and $v$ agree on which one starts to build a rematch and which one ends. This local consensus is done using *AskFirst* and *AskSecond* functions. Observe that these predicates are exactly the same in both algorithms. These predicates use the local state of $u$ and $v$ to assign a role to these two nodes. If *AskFirst(u)* is not *null* then $u$ starts to rematch and $v$ ends. Otherwise, *AskSecond(u)* is not *null* and then $v$ starts to rematch and $u$ ends.

Observe that there are only 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 $\alpha$ and $\beta$ values are. These are: $(null, null, null, null)$ or $(x, null, null, y)$ or $(null, x, y, null)$, with $x$ and $y$ are two distincts single nodes. 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. The second case occurs when $x < y$, 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 $AskSecond(u) \neq null$. So, if a 3-augmenting path is detected though $(u, v)$, the roles of $u$ and $v$ depend on the identifiers of single nodes (candidates) in the augmenting path, *i.e.*, $u$ is *First* iff its single neighbor in the augmenting path has a smaller identifier than the single neighbor of $v$ in the augmenting path.

*Graphical convention.* We will follow the above conventions in all the figures: matched nodes are represented with thick circles and single nodes with thin circles. Node identifiers are indicated inside the circles. Moreover, all edges that belong to the maximal matching $\mathcal{M}$ are represented with a thick line, whereas the other edges are represented with a simple line. We illustrate the use of the $p$-values by an arrow, and the absence of the arrow or symbol '$\mathsf{T}$' mean that the $p$-value of the node equals to null. A prohibited value is first drawn in grey, then scratched out in black. For instance, in Figure 8, node $x_1$ is single, nodes $u_1$ and $v_1$ are matched, the edge $(u_1, v_1) \in \mathcal{M}$ and $p_{x_3} \neq v_2$.

## 4. Description of the algorithm ExpoMatch

We precisely describe here the algorithm ExpoMatch [20]. The algorithm itself is shown in Figure 3.

*Augmenting path detection.* In this algorithm, a single node $x$ is a *candidate* for a matched node $u$ if it is not involved in another augmenting path exploitation, *i.e.*, if $p_x = null \vee p_x = u$.

*Augmenting path exploitation.* A 3-augmenting path is exploited in two phases. These two phases are performed in a sequential way. Recall that node $u$ is said to be *First* if $AskFirst(u) \neq null$ and node $u$ is *Second* if $AskSecond(u) \neq null$. Let us consider two nodes $u$ and $v$ such that $(u, v) \in \mathcal{M}$. Let us assume that $u$ and $v$ detects an augmenting path.

1. The *First* node starts : Exactly one node among $u$ and $v$ attempts to rematch with one of its candidates. This phase is complete when the first node, let say $u$, is such that $s_u = True$ and this indicates to the *Second* node ($v$) that the first phase is over.
2. The *Second* node continues: only when the first node succeeds will the second node attempt to rematch with one of its candidates. (a) If this also succeeds, the exploitation is done and the augmenting path is said to be *fully exploited* ; (b) Otherwise the rematch built by the *First* node is deleted and candidates $\alpha$ and $\beta$ are computed again, allowing then the detection of some new augmenting paths.

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

**SingleNode ::**
**if** $(p_u = null \wedge Lowest(\{v \in N(u) \mid p_v = u\}) \neq null) \vee p_u \notin matched(N(u)) \cup \{null\} \vee$
$(p_u \neq null \wedge p_{p_u} \neq u)$
**then** $p_u := Lowest(\{v \in N(u) \mid p_v = u\})$

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

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

**MatchFirst ::**
**Let** $x = \text{AskFirst}(u)$
**if** $x \neq null \wedge (p_u \neq x \vee s_u \neq (p_{p_u} = u))$
**then** $p_u := x$
$s_u := (p_{p_u} = u)$

**MatchSecond ::**
**Let** $y = \text{AskSecond}(u)$
**if** $y \neq null \wedge s_{m_u} = true \wedge p_u \neq y$
**then** $p_u := y$

**ResetMatch ::**
**if** $\text{AskFirst}(u) = \text{AskSecond}(u) = null$
$\wedge (p_u, s_u) \neq (null, false)$
**then** $(p_u, s_u) := (null, false)$

——— **Predicates and functions**

**BestRematch**$(u) \equiv$
$a := Lowest\ (\{v \in single(N(u)) \wedge (p_v = null \vee p_v = u)\})$
$b := Lowest\ (\{v \in single(N(u)) \setminus \{a\} \wedge (p_v = null \vee p_v = u)\})$
return $(a, b)$

**AskFirst**$(u) \equiv$
**if** $\alpha_u \neq null \wedge \alpha_{m_u} \neq null \wedge 2 \leq Unique(\{\alpha_u, \beta_u, \alpha_{m_u}, \beta_{m_u}\}) \leq 4$
**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 $\alpha_u$
return $null$

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

$Unique(A)$ returns the number of unique elements in the multi-set $A$.
$Lowest(A)$ returns the node in $A$ with the lowest identifier. If $A = \emptyset$, then $Lowest(A)$ returns $null$.

Figure 3: ExpoMatch algorithm

*Rules description.* There are four rules for matched nodes. The *Update* rule is the rule with the highest priority. This rule allows a matched node to update its $\alpha$ and $\beta$ variables, using the *BestRematch* predicate. 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 *ResetMatch* rule is performed to reset bad initialization and also to reset an augmenting path exploitation that did not terminate. For instance, this case happens when the single candidate of the *Second* node rematch with some other node in the middle of the exploitation path process.

Let us consider $(u, v) \in \mathcal{M}$ and assume that $u$ and $v$ detects an augmenting path with $u$ is *First*. The *MatchFirst* rule is used by $u$ to build its rematch. The rule is performed a first time by $u$ to propose a rematch to its candidate $x$ ($u$ sets $p_u$ to $x$). Then, if $x$ accepts ($p_x = u$), $u$ performs this rule a second time to communicate to $v$ that its rematch attempt is a succeed ($u$ sets $s_u$ to $True$). The *MatchSecond* rule is used by the node $v$ to build its rematch. This rule can only be performed if $s_u = True$. Then, the rule is performed once by $v$ to propose a rematch to its candidate $y$ ($v$ sets $p_v$ to $y$). Then, if $y$ accepts ($p_y = v$), the path is fully exploited and will not change during the rest of the execution.

There is only one rule for single nodes, called *SingleNode*. Recall that all neighbors of a single node are

matched, since $\mathcal{M}$ is a maximal matching. A single node should always point to its smallest neighbor that points to it. This rule allows to point to such a neighbor but also to reset a bad $p$-value to *null*. Observe that a single node $x$ cannot perform this rule if $p_{p_x} = x$, which means that if $x$ point to some neighbor that points back to $x$, then $x$ is locked.

## 5. The ExpoMatch algorithm is sub-exponential

In this section, we exhibit an execution of length $2^N$ in a *chosen* graph having $\Theta(N^2)$ nodes. To do that, we define, under some conditions, how to translate a configuration into a binary integer. Then, we give an execution where all configurations corresponding to integers from 0 to $2^N - 1$ appear. This gives us an execution of length in $\Omega(2^N)$.

### *5.1. State of a matched edge*

A bit in the binary integer of a given configuration correspond to a particular state of the nodes in a 3-augmenting path. More precisely, according to the $p$-values of these nodes, the associated bit of the path will be 0, 1 or *undef*. Figure 5 represents an instance of the chosen graph for $N = 4$. Observe that any matched node only has one single neighbor. This property will hold for any $N$. Thus, a 3-augmenting path can be determined by its matched edge.

**Definition 3** (State of a matched edge)**.** *Let $e = (u, v)$ be an edge in the maximal matching $\mathcal{M}$ such that $u$ (resp. $v$) has one single neighbor $x$ (resp. $y$). Assume $y < x$. Edge $e$ is said to be:*

- in state Off *if $p_x = null$, $p_u = null$, $p_v = null$ and $p_y = null$.*
- in state AlmostOff *if $p_x \notin \{null, u\}$, $p_u = null$, $p_v = null$, and $p_y = null$.*
- in state On *if $p_x = null$, $p_u = x$, $p_v = y$ and $p_y = v$.*
- in state AlmostOn *if $p_x \notin \{null, u\}$, $p_u = x$, $p_v = y$ and $p_y = v$.*

Note that a matched edge can be in none of the states presented below. The states of an edge represents the different steps of an augmenting path exploitation. Now, we exhibit an execution to switch an edge $(u, v)$ from state Off to state On in Lemma 1 and then, from state AlmostOn to state AlmostOff in Lemma 2.

**Lemma 1.** *Let $e = (u, v)$ be an edge in the maximal matching $\mathcal{M}$ such that $u$ (resp. $v$) has one single neighbor $x$ (resp. $y$). Assume $y < x$. Let $C$ be a configuration where $e$ is in state* Off *and $v = min(\{w \in N(y) : p_w = y\} \cup \{v\})$. There exists a finite execution starting in $C$ and ending in $D$ such that:*
*(i) only nodes $u$, $v$ and $y$ make moves between $C$ and $D$ and (ii) edge $e$ is in state* On *in $D$.*

*Proof.* We describe a finite execution starting in $C$ and ending in $D$ that allows to switch edge $(u, v)$ from state Off to state On and where only nodes $u$, $v$ and $y$ make moves. Nodes $u$ and $v$ belong to a 3-augmenting path in $C$ since $p_x = p_y = null$ by assumption. If $\alpha_u \neq x$, then node $u$ executes an *Update* move and sets $(\alpha_u, \beta_u) = (x, null)$. If $\alpha_v \neq y$, then node $v$ executes an *Update* move and sets $(\alpha_v, \beta_v) = (y, null)$.

Now, the variables $\alpha_u$ and $\alpha_v$ are well defined. Since $y < x$, we have $AskFirst(v) = y$ and $AskSecond(u) = x$. So node $v$ executes a *MatchFirst* move and sets $p_v = y$. Let $C_1 \mapsto C_2$ be the transition where $v$ makes this $MatchFirst$ move. Observe that only $u$ and $v$ made some moves from $C$ to $C_2$. Moreover, $u \notin N(y)$ since $u$ has only one single neighbor that is $x$. Thus $v = min(\{w \in N(y) : p_w = y\} \cup \{v\})$ still holds in $C_2$ and so, node $y$ chooses node $v$ to match with by executing a *SingleNode* move. Finally, node $u$ is eligible to execute a *MatchSecond* move and it then points to node $x$. The edge $(u, v)$ is now in state On. □

Now, we exhibit an execution to switch edge $(u, v)$ from state AlmostOn to state AlmostOff.

**Lemma 2.** *Let $e = (u, v)$ be an edge in the maximal matching $\mathcal{M}$ such that $u$ (resp. $v$) has one single neighbor $x$ (resp. $y$). Assume $y < x$. Let $C$ be a configuration where: $e$ is in state* AlmostOn *and $\{w \in N(y) : p_w = y\} = \{v\}$. There exists a finite execution starting in $C$ and ending in $D$ such that:*
*(i) only nodes $u$, $v$ and $y$ make moves between $C$ and $D$ and (ii) edge $e$ is in state* AlmostOff *in $D$.*

*Proof.* We describe a finite execution starting in $C$ and ending in $D$ that allows to switch edge $(u, v)$ from state AlmostOn to state AlmostOff and where only nodes $u$, $v$ and $y$ make moves. Since edge $(u, v)$ is in state AlmostOn, then $p_x \notin \{null, u\}$ and so $BestRematch(u) = (null, null)$. If $(\alpha_u, \beta_u) \neq (null, null)$ then node $u$ executes an *Update* move. Otherwise, $AskFirst(u) = AskSecond(u) = null$ and, since $p_u \neq null$, $u$ executes a *ResetMatch* move. In both cases, after the move, $(p_u, s_u) = (null, false)$ and $(\alpha_u, \beta_u) = (null, null)$.

$\alpha_u = null$ implies $AskFirst(v) = null$, and $AskFirst(u) = null$ implies $AskSecond(v) = null$. Moreover, since $p_v \neq null$, $v$ executes a *ResetMatch* move and sets $p_v = null$. Let $C_1 \mapsto C_2$ be the transition where $v$ makes this *ResetMatch* move. Since $\{w \in N(y) : p_w = y\} = \{v\}$ holds in the configuration $C$ and since only $u$ and $v$ made some moves from $C$ to $C_2$ then we have: $\{w \in N(y) : p_w = y\} = \emptyset$ holds in $C_2$. Thus node $y$ performs a *SingleNode* move and sets $p_y = null$. The edge $(u, v)$ is now in state AlmostOff. □

*5.2. The graph $G_N$ and how to interpret a configuration into a binary integer*

In the following, we describe an execution corresponding to count from 0 to $2^N - 1$, where $N$ is an arbitrary integer. This execution occurs in a graph denoted by $G_N$ with $\Theta(N^2)$ nodes. $G_N$ is composed by $N$ sub-graphs, each of them representing a bit. The whole graph then represents an integer, coded from these $N$ bits. $G_N$ has 2 kind of nodes: the nodes represented by circles ($\bullet$-nodes) and those represented by squares ($\blacksquare$-nodes). The $\bullet$-nodes are used to store bit values and hence an integer. The $\blacksquare$-nodes are used to implement the "+1" operation as we count from 0 to $2^N - 1$. We now formally describe the graph $G_N = (V_N, E_N)$:

$$V_N = V_N^{\bullet} \cup V_N^{\blacksquare} \text{ where } \quad \begin{array}{rcl} V_N^{\bullet} & = & \bigcup_{0 \leq i < N} \{b(i,k) | k = 1,2,3,4\} \\ V_N^{\blacksquare} & = & \bigcup_{0 \leq j < i < N} \{r_1(i,j), r_2(i,j)\} \end{array}$$

$$E_N = E_N^{\bullet} \cup E_N^{\blacksquare} \text{ where } \quad \begin{array}{rcl} E_N^{\bullet} & = & \bigcup_{0 \leq i < N} \{(b(i,k), b(i,k+1)) | k = 1,2,3\} \\ E_N^{\blacksquare} & = & \bigcup_{0 \leq j < i < N} \{(b(i,1), r_1(i,j)) \ , \ (r_1(i,j), r_2(i,j)) \ , \ (r_2(i,j), b(j,4))\} \end{array}$$

Figure 4 gives a partial view of the graph $G_N$ corresponding to the $i$th bit-block.

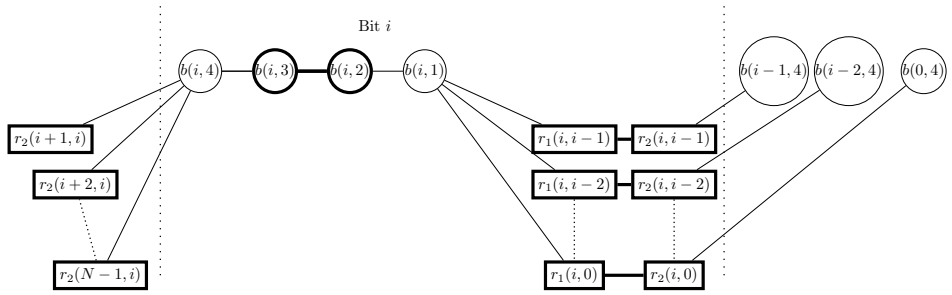



Figure 4: A partial view of graph $G_N$

Our exponential execution used the following underlying maximal matching $\mathcal{M}$:

$$\mathcal{M} = \{(b(i,2), b(i,3)) | 0 \leq i < N\} \cup \{(r_1(i,j), r_2(i,j)) | 0 \leq j < i < N\}$$

This maximal matching is encoded with the $m$-variables then we have:

$$\forall i, j \text{ with } 0 \leq j < i < N : m_{b(i,2)} = b(i,3), m_{b(i,3)} = b(i,2), m_{r_1(i,j)} = r_2(i,j) \text{ and } m_{r_2(i,j)} = r_1(i,j)$$

The matching $\mathcal{M}$ is a $\frac{1}{2}$-approximation of the maximum matching and the algorithm ExpoMatch updates this approximation building $\mathcal{M}^+$, a $\frac{2}{3}$-approximation of the maximum matching. $\mathcal{M}^+$ is encoded with the $p$-variable and we also use this variable to encode the binary integer associated to a configuration.

*Example.* As an illustration, graph $G_4$ is shown in Figure 5. In this example, the bold edges are those belonging to the maximal matching $\mathcal{M}$ and arrows represent the local variable $p$ of the algorithm ExpoMatch. A node having no outgoing arrow has its $p$-variable equal to *null*.

As we said, the $\bullet$-nodes are used to encode the $N$ bits. Each bit $i$ is encoded with the local state of the 4 following nodes: $b(i,1), b(i,2), b(i,3), b(i,4)$. These nodes are then named $b(i,k)$, for "*the $k^{th}$ node of the bit $i$*". For instance, node 10 is the fourth node of the bit 0, thus node 10 is called $b(0,4)$. In the following,

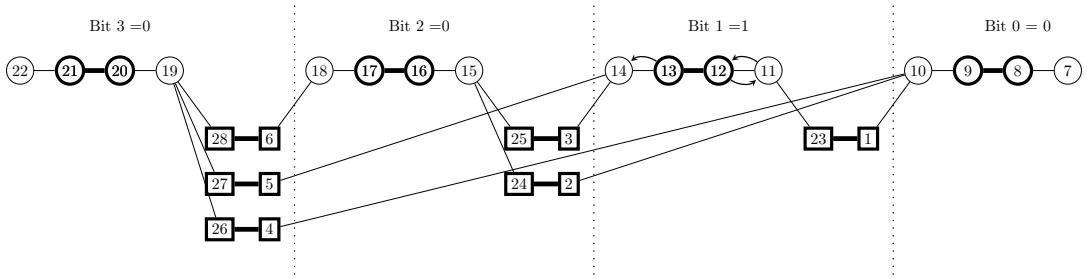



Figure 5: Graph $G_4$ encoding 0010

we will refer to these four nodes as the $i^{th}$ *bit-block*. The binary value associated to a bit-block is computed accordingly to the $p$-value of each node in the bit-block. The following definition gives this association:

**Definition 4** (Bit-block encoding)**.** *In graph $G_N$, nodes $\{b(i,1), b(i,2), b(i,3), b(i,4)\}$ are the $i^{th}$ bit-block, for some $0 \leq i < N$. This bit-block encodes the value 1 (*resp. *0) if the edge $(b(i,2), b(i,3))$ is in state* On *(*resp. Off*) and if $\forall j$ with $0 \leq j < i, p_{r_1(i,j)} = p_{r_2(i,j)} = null$.*

Note that the value encoded by a bit-block is not always defined. But when all bit-blocks encode a bit in a given configuration, then we can associate a positive integer $\omega$ to this configuration.

**Definition 5** ($\omega$-configuration)**.** *Let $\omega$ be an integer* s.t. *$0 \leq \omega < 2^N$, a configuration $C$ is said to be an $\omega$-*configuration *if for any integer $0 \leq i < N$, the $i^{th}$ bit of $\omega$ is the value encoded by the $i^{th}$ bit-block in $C$.*

Observe that all the $p$-values of the ■-nodes have to be *null* in any $\omega$-configuration. In Figure 5, all $p$-values of ■-nodes are *null*. Moreover, the edges $(9,8)$, $(17,16)$ and $(21,20)$ are in state Off while the edge $(13,12)$ is in state On. Thus, $G_4$ encodes the binary integer 0010 and so Figure 5 shows a 2-configuration.

*5.3. Identifiers in $G_N$*

In order to exhibit our execution counting from 0 to $2^N - 1$, we need to be able to switch edges between On and Off. This can be done executing the guarded rules of ExpoMatch. Since this algorithm uses identifiers, we need some properties on identifiers of nodes in $G_N$. The *Ident* function gives the identifier associated to a node in $V_N$. Recall that we assume each node has a unique identifier. These identifiers must satisfy the three following properties:

**Property 1** (Identifiers order in $G_N$)**.** *Let $b(i,k), b(i',k'), b(i,2)$ and $b(i,3)$ be nodes in $V_N^{\bullet}$, and $r_1(i,j)$ and $r_2(i,j)$ be nodes in $V_N^{\blacksquare}$. We have:*

1. $Ident(b(i,k)) > Ident(b(i',k'))$ *if* $(i > i') \vee (i = i' \wedge k > k')$
2. $Ident(b(i,2)) < Ident(r_1(i,j))$
3. $Ident(b(i,3)) > Ident(r_2(j,i))$

We now show that in graph $G_N$, there exists an *Ident* function that satisfies Property 1. Indeed, the property holds for the following naming: Let $c = |V_N^{\bullet}|$ and $s = \frac{|V_N^{\blacksquare}|}{2}$. There are $c$ nodes of kind $b$, $s$ nodes of kind $r_1$ and $s$ nodes of kind $r_2$ as well. Nodes of kind $r_2$ are named from 1 to $s$. Nodes of kind $b$ are named from $s+1$ to $s+c$ such that: $\forall i, 0 \leq i < N, \forall k \in \{1,2,3,4\} : Ident(b(i,k)) = s + 4i + k$. And finally, nodes of kind $r_1$ are named from $s+c+1$ to $s+c+s$. Figure 5 shows graph $G_4$ with such a naming ($c$=16 and $s$=6).

*5.4. Counting from 0 to $2^N - 1$*

We build an execution containing all $\omega$-configurations with $0 \leq \omega < 2^N - 1$. To to this, we build an execution from an $\omega$-configuration to the $(\omega+1)$-configuration using a '+1' operation. Thus we need to be able to switch bit from 0 to 1 and from 1 to 0. The main scheme is the following: let us consider a binary integer $x$. The '+1' operation consists in finding the rightmost 0 in $x$. Then all 1 at the right of this 0 have to switch to 0 and this 0 has to switch to 1 (if $x = x'011\ldots1$ then $x+1 = x'100\ldots0$). Then if 0 is the $i^{th}$ bit of $x$, the $i^{th}$ bit-block has to switch from 0 to 1 during the '+1'operation. And each $j^{th}$ bit-block, with $0 \leq j < i$, has to switch from 1 to 0.

The switch of a bit-block from 0 to 1 only needs the •-nodes to perform moves (see Lemma 1). However, this is not the case when we want to switch a bit-block from 1 to 0. Indeed, we use some other nodes to help to perform the switch: the ■-nodes.

**Theorem 1.** *Let $\omega$ be an integer such that $0 \leq \omega < 2^N - 1$. There exists a finite execution to transform an $\omega$-configuration into an $(\omega+1)$-configuration.*

*Proof.* Let $C$ be an $\omega$-configuration. Let $\rho$ be the integer such that the $\rho-1$ first bits of $\omega$ equal to 1 and the value of its $\rho^{th}$ bit to 0. This implies that the $\rho^{th}$ bit of $\omega+1$ is the first bit equal to 1. We distinguish two cases: $\rho = 0$ and $\rho > 0$. (i) In the case where $\rho = 0$, edge $(b(0,2), b(0,3))$ is in state Off by definition. Since the $0th$ bit of integer $\omega+1$ is equal to 1, $(b(0,2), b(0,3))$ must be in state On in the $(\omega+1)$-configuration. By Property 1, we have $Ident(b(0,1)) < Ident(b(0,4))$. Moreover nodes $b(0,3)$ and $b(0,2)$ only have one single neighbor, so the hypotheses of Lemma 1 are satisfied. Thus, from Lemma 1, there exists an execution to switch edge $(b(0,2), b(0,3))$ from state Off to state On and in this execution, only nodes $b(0,1), b(0,2)$ and $b(0,3)$ make moves. At the end, the $0th$ bit has changed from 0 to 1 and the other did not change. We then have an $(\omega+1)$-configuration. (ii) In the case where $\rho > 0$, for every integer $i$ from 0 to $\rho-1$, edge $(b(i,2), b(i,3))$ is in state On and edge $(b(\rho,2), b(\rho,3))$ is in state Off. We can execute the following sequence of moves to obtain the $(\omega+1)$-configuration:

1. We first consider the 3-augmenting path $(b(\rho,1), r_1(\rho,j), r_2(\rho,j), b(j,4))$ for any integer j, $0 \leq j < \rho$. We prove that the matched edge of this path is in state Off and that it satisfies the assumptions of Lemma 1. Then, we switch this edge from state Off to state On applying Lemma 1 (where the path $(x,u,v,y)$ in this lemma corresponds to the path $(b(\rho,1), r_1(\rho,j), r_2(\rho,j), b(j,4))$).
   Note that $\forall j, 0 \leq j < \rho$, node $r_1(\rho,j)$ (resp. $r_2(\rho,j)$) is adjacent to one single node $b(\rho,1)$ (resp. $b(j,4)$). As for any $j$, $0 \leq j < \rho$, the $j^{th}$ bit-block encodes the value 1 in $C$, then $p_{b(j,4)} = null$ in $C$. In the same way, as the $\rho^{th}$ bit-block encodes the value 0 in $C$, then $p_{b(\rho,1)} = null$ in $C$. As $C$ is an $\omega$-configuration, then $p_{r_1(\rho,j)} = null$ and $p_{r_2(\rho,j)} = null$. Thus the edge $(r_1(\rho,j), r_2(\rho,j))$ is in state Off in $C$.
   Moreover, $Ident(b(j,4)) < Ident(b(\rho,1))$ by Property 1. Finally, in $C$, we have $\{w \in N(b(j,4)) : p_w = b(j,4)\} = \{b(j,3)\}$ since all neighbors of $b(j,4)$ but $b(j,3)$ are ■-nodes, and so they have their $p$-value equal to $null$ in $C$. We have $Ident(r_2(\rho,j)) < Ident(b(j,3))$ by Property 1, then $r_2(\rho,j) = min(\{w \in N(b(j,4)) : p_w = b(j,4)\} \cup \{r_2(\rho,j)\})$ and the hypotheses of Lemma 1 are satisfied. Thus from Lemma 1, we can exhibit an execution to switch edges $(r_1(\rho,j), r_2(\rho,j))$ from state Off to state On and where only nodes $r_1(\rho,j), r_2(\rho,j)$ and $b(j,4)$ make moves.
2. Now, for each integer $j$, $0 \leq j < \rho$, edge $(b(j,2), b(j,3))$ is in state AlmostOn. $Ident(b(j,1)) < Ident(b(j,4))$ and $\{w \in N(b(j,1)) : p_w = b(j,1)\} = \{b(j,2)\}$ so hypothesis of Lemma 2 hold. Thus from Lemma 2, an execution to switch edge $(b(j,2), b(j,3))$ from state AlmostOn to state AlmostOff is performed.
3. Edge $(b(\rho,2), b(\rho,3))$ is still in state Off. Using the same argument of step (1), from Lemma 1, we can exhibit an execution to switch edges $(b(\rho,2), b(\rho,3))$ from state Off to state On.
4. Now, for each integer $j$, $0 \leq j < \rho$, edge $(r_1(\rho,j), r_2(\rho,j))$ is now in state AlmostOn. From Lemma 2, there exists an execution to switch edge $(r_1(\rho,j), r_2(\rho,j))$ from state AlmostOn to state AlmostOff.

At the end of this execution, we obtain a configuration where the $\rho-1$ first bits of $\omega$ are equal to 0 and the $\rho^{th}$ bit is 1. Moreover, observe that all ■-nodes are in state AlmostOff or Off, thus they all have their $p$-value sets to $null$. We are then in an $(\omega+1)$-configuration. □

**Corollary 1.** *Let $n$ be the number of nodes. In the worst case, Algorithm* ExpoMatch *stabilizes after $\Omega(2^{\sqrt{n/2}})$moves under the central daemon.*

*Proof.* We can build an execution that contains all the $\omega$-configurations for every value $\omega$, $0 \leq \omega < 2^N$. By applying Theorem 1, this execution can be split into $2^N$ parts corresponding to the execution from $\omega$-configuration to $(\omega+1)$-configuration, for $0 \leq \omega < 2^N$. Thus, this execution contains $2^N$ configurations. Since graph $G_N$ has $4N + N(N+1)$ vertices, then $n \leq 2N^2$ for $n \geq 5$, and then $\sqrt{n} \leq N\sqrt{2}$. Thus $2^{\sqrt{n/2}} \leq 2^N$ and the corollary holds. □

## 6. Our algorithm PolyMatch

The algorithm presented in this paper is called PolyMatch, and is based on the algorithm presented by Manne *et al.* [20], called ExpoMatch. As in ExpoMatch, PolyMatch assumes there exists an underlying maximal matching, called $\mathcal{M}$. PolyMatch algorithm is presented in Figure 6. Predicates $AskFirst$ and $AskSecond$ are not given since they are the same as in ExpoMatch algorithm (see Fig. 3).

### *6.1. Variables description*

Our algorithm has the same set of local variables as in ExpoMatch plus one additional boolean variable, called *end*. As in ExpoMatch, for a matched node $u$, the pointer $p_u$ refers to a neighbor of $u$ that $u$ is trying to (re)match with, and pointers $\alpha_u$ and $\beta_u$ refer to two candidates for a possible rematching with $u$. And again, $s_u$ is a boolean variable that indicates if $u$ has performed a successful rematching with its candidate. Finally, the new variable $end_u$ 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 $x$ is *available* for a matched node $u$ if it is not involved in another augmenting path that is fully exploited, *i.e.*, if it is possible for $x$ to eventually rematch with $u$, and thus if $p_x = u \vee end_x = False$ (see *BestRematch* predicate).

### *6.2. Augmenting paths exploitation*

A 3-augmenting path is exploited in three phases. These phases are performed in a sequential way. Let us consider two nodes $u$ and $v$ such that $(u, v) \in \mathcal{M}$. Let us assume that $u$ and $v$ detects a 3-augmenting path.

1. The *First* node starts (same as in ExpoMatch): The *First* node, let say $u$, tries to rematch with its candidate. This phase is complete when $s_u = True$ and this indicate to the *Second* node, let say $v$, that the first phase is over.
2. The *Second* node continues: only when the first node succeeds will the second node attempt to rematch with one of its candidates. This phase is complete when $end_v = True$ and this indicate to the $v$'s neighbors that the second phase is over.
3. All nodes in the path set their *end* variable to *True*: the *end* value of $v$ is propagated in the path. The goal of this phase is to write *True* in the *end* variables of the two single nodes in the path in order to make them unavailable for other married nodes. Indeed, the *end* variable is used to compute the candidates of a matched node.

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 again 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 these 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.

### *6.3. Rules description*

There are four rules for matched nodes. As in ExpoMatch, the *Update* rule allows a matched node to update its $\alpha$ and $\beta$ variables, using the *BestRematch* predicate. 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*) for this 3-augmenting path exploitation. The *ResetMatch* rule is performed to reset bad initialization and also to reset an augmenting path exploitation that did not terminate.

The *MatchFirst* rule is used by the node when it is *First*. Let $u$ be this node. The rule is performed three times in a usual path exploitation:

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

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

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

**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** $p_u := Lowest\{w \in matched(N(u)) \mid p_w = u\}$
$end_u := False$

——— **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 $(a, b)$ )

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

**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** $end_u := (p_u = AskSecond(u) \wedge p_{p_u} = u \wedge p_{m_u} = AskFirst(m_u))$
$s_u \quad := end_u$
$p_u \quad := AskSecond(u)$

**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** $(p_u, s_u, end_u) := (null, False, False)$

Figure 6: PolyMatch algorithm

1. The first time, $u$ seduces its candidate setting $(end_u, s_u, p_u)$ to $(False, False, AskFirst(u))$.
2. 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.
3. Finally, the *MatchFirst* rule is performed a third time when $m_u$ completed his own rematch, *i.e.*, when $end_{m_u} = True$. Observe that when there is no bad information due to some bad initializations, then $end_{m_u} = True$ means $p_{m_u} = AskSecond(m_u) \wedge p_{p_{m_u}} = m_u$ (see the third line of the *MatchSecond* rule). 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 assignment operation of $s_u$ before the one of $p_u$, 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 $end_u$ assignment before the $s_u$ assignment. 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 a usual 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 his proposition, $u$ can perform *MatchSecond* for the second time, setting $s_u$ and $end_u$ to *True*. As in the *MatchFirst* rule, we set the *end* and $s$ assignments before the $p$ assignment.

There are three 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 *null*. 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.

### *6.4. Execution examples*

We give two different executions of algorithm PolyMatch under the adversarial distributed daemon. The first execution points out the main differences between our algorithm PolyMatch and algorithm ExpoMatch. In the second execution, we focus on the *end* variable role for the exploited path process.

*Main difference between* PolyMatch *and* ExpoMatch *algorithms.* When two neighboring augmenting-paths are exploited in parallel, then at most one among the two will eventually become a fully exploited augmented-path. In Manne *et al.* algorithm, a destruction of a partially exploited augmenting-path can be done while no fully exploited augmenting-path has been built instead. Moreover, for one fully exploited augmented-path, we can exhibit some executions where we destroy a sub-exponential number of exploited augmented-paths (see Section 5). In our algorithm, this is not possible since we do not destroy any partially exploited augmented-path while there is still hope to exploit it. This difference is implemented in the algorithm through the *BestRematch(u)* predicate. The condition $p_x = null$ in Manne *et al.* algorithm has been replaced by the condition $end_x =$*False* in our algorithm, meaning that node $x$ must belong to a fully exploited augmented-path in order to disappear from the candidates of $u$.

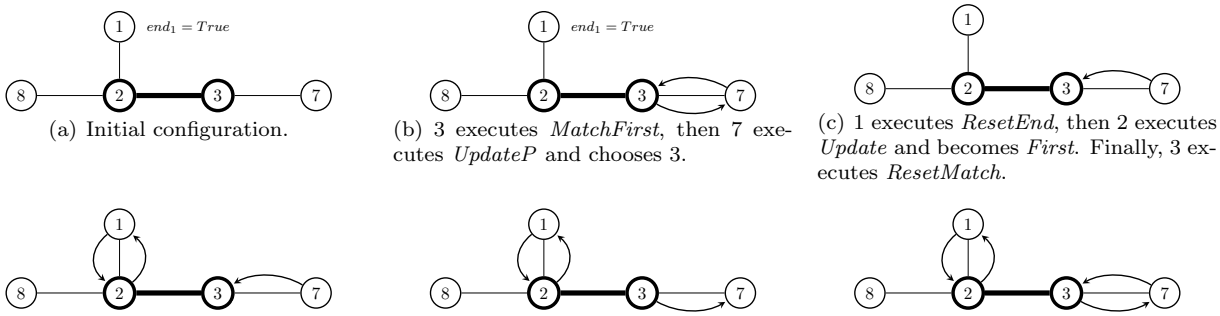
*How to handle the end-variable?*

Second, we consider the following execution in order to illustrate the role of local *end*-variable. Figure 7(a) shows the initial state of the execution. The underlying maximal matching contains one edge $(2, 3)$. Then nodes 2, 3 are *matched* nodes, and nodes 1, 7, and 8 are *single* nodes. At the beginning, there are two 3-augmenting paths: $(1, 2, 3, 7)$ and $(8, 2, 3, 7)$.

*The initial configuration (Figure 7(a)).* In the initial configuration, we assume that all $\alpha$*-values* and $\beta$*-values* are defined as follows: $(\alpha_2, \beta_2) = (8, null)$, and $(\alpha_3, \beta_3) = (7, null)$. We also assume all *s-values* are well defined (*i.e.*, equal to *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: $BestRematch(2) = (8, null)$.

The 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 7(b)).* Node 3 executes here a *MatchFirst* move and points to node 7. Since node 3 is pointing to node 7, node 7 is the only activable node among all nodes except node 1. Node 7 points to node 3 by executing a *UpdateP* move.




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

(f) 7 executes *UpdateP*, then 3 executes *MatchSecond*, then the *True* value of $end_3$ is propagated in the path $(1, 2, 3, 7)$.

Figure 7: An execution of Algorithm PolyMatch (Only the *True* value of the *end*-variables are given)

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. Let us assume it makes this move. After this move, we have $end_1 =$*False*. This implies that $BestRematch(2) = (1, 8)$ and thus $(\alpha_2, \beta_2) = (8, null) \neq BestRematch(2)$. So, only node 2 is activable, and is eligible for *Update* rule. Thus, after this mode, 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*. Let us assume it does it.

*A second* 3*-augmenting path exploitation starts (Figure 7(d)).* 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 executing a *UpdateP* move. 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*.

In the configuration shown in Figure 7(e), node 3 can propose to node 7 with a *MatchSecond*. Note that node 7 is also eligible for *UpdateP* since $p_3 \neq 7$. Let us assume these two nodes do the move in parallel. Figure 7(e) shows the configuration obtained after theses moves: $p_3 = 7$, $p_7 = null$. 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 $s$-assignment, then the $p$-assignment 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* $(1, 2, 3, 7)$ *becomes fully exploited (Figure 7(f)).* Now, node 7 can choose 3 by executing *UpdateP*. Assume that it does. Since $end_3 \neq (p_3 = 7 \wedge p_3 = AskSecond(3) \wedge p_2 = AskFirst(2))$, node 3 is eligible for a *MatchSecond* rule to set $end_3$ to *True* and then to make the other nodes aware that the path is fully exploited. Assume node 3 executes a *MatchSecond* move. This will cause node 7 (resp. 2) to execute an *UpdateEnd* move (resp. a *MatchFirst* move) and sets $end_7 =$*True* (resp. $end_2 =$*True*). Now, it is the turn to node 1 to execute an *UpdateEnd* move. As the *end*-value of nodes 1, 2, 3, and 7 are equal to *True*, the 3-augmenting path is fully exploited. The system has reached a stable configuration (see Figure 7(f)). Thus, the size of the matching is increasing by one and there is no 3-augmenting path left.

## 7. Correctness Proof

A natural way to prove the correction of PolyMatch algorithm could have been to follow the approach below. We consider a stable configuration $C$ in PolyMatch 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 PolyMatch, 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 sense 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 3, 4, 5, 6 and a part of the proof from Theorem 2. Observe that from these results, the correctness is straightforward without using the Manne *et al.* proof.

We first introduce some notations. A matched node $u$ is said to be *First* if $AskFirt(u) \neq null$. In the same way, $u$ is *Second* if $AskSecond(u) \neq null$. 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.

Recall that the set of edges built by our algorithm PolyMatch 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 the correctness part of the proof, 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 distincts 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. 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 3.** *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* $u \in matched(N(x)) \wedge p_u = x \wedge end_u = end_x$*.*

*Proof.* First, we will prove the first property. We consider the case where $AskFirst(u) \neq null$. We have $p_u = AskFirst(u)$, otherwise node $u$ can execute rule $AskFirst$. We can apply the same result for the case where $AskSecond(u) \neq null$. Finally, we consider the case where $AskFirst(u) = AskSecond(u) = null$. If $p_u \neq null$, then node $u$ can execute rule $ResetMatch$ which yields the contradiction. Thus, $p_u = null$.

Second, we consider a stable configuration $C$ where $p_x = u$, with $u \neq null$. $u \in matched(N(x))$, otherwise $x$ is eligible for an *UpdateP* rule. Now there are two cases: $p_u = x$ and $p_u \neq x$. If $p_u \neq x$, this means that $p_{p_x} \neq x$. Thus, $x$ is eligible for rule $UpdateP$, and this yields to a contradiction with the fact that $C$ is stable. Finally, we have $end_u = end_x$, otherwise $x$ is eligible for rule $UpdateEnd$. □

**Lemma 4.** *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* $C$*, then* $C$ *is not stable.*

*Proof.* By contraction. We assume $C$ is stable. From Lemma 3, we have $p_u = Ask(u) \neq null$ and $p_v = Ask(v)$. So, by definition of predicates $AskFirst$ and $AskSecond$, $Ask(u) = x \neq null$ implies that $Ask(v) \neq null$. This contradicts that fact that $p_v = Ask(v) = null$. □

**Lemma 5.** *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* $end_x = end_u = end_v = end_y = True$*.*

*Proof.* From Lemma 3, $p_u = Ask(u)$ (resp. $p_v = Ask(v)$) thus $Ask(u) \neq null$ and $Ask(v) \neq null$. W.l.o.g, we can assume that $AskFirst(u) \neq null$. We have $s_u = True$, otherwise $u$ can execute $MatchFirst$ rule. Now, as $s_u = True$, we must have $end_v = True$, otherwise $v$ can execute $MatchSecond$ rule. As $s_u = end_v = True$, we must have $end_u = True$, otherwise $u$ can execute $MatchFirst$ rule. From Lemma 3, we can deduce that $end_x = end_u = end_v = end_y = True$ and this concludes the proof. □

**Lemma 6.** *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* $C$*, then* $C$ *is not stable.*

*Proof.* By contraction. We assume $C$ is stable. From Lemma 3, $Ask(u_1) = x_1$ and $Ask(v_1) = x_2$.

First we assume that $AskSecond(u_1) = x_1$ and $AskFirst(v_1) = x_2$. The local variable $s_{v_1}$ is $False$, otherwise $v_1$ would be eligible for executing the $MatchFirst$ rule. Since $AskSecond(u_1) \neq null \wedge p_{u_1} \neq null \wedge s_{v_1} = False$, this implies that $u_1$ is eligible for the $ResetMatch$ rule which is a contradiction.

Second, we assume that $AskFirst(u_1) = x_1$ and $AskSecond(v_1) = x_2$. We have $s_{u_1} = True$, otherwise $u_1$ can execute the $MatchFirst$ rule. This implies that $end_{v_1} = False$, otherwise $v_1$ can execute the $MatchSecond$ rule. As $end_{v_1} = False$, then $end_{u_1} = False$, otherwise $u_1$ can execute the $MatchFirst$ rule. From Lemma 3, $end_{x_1} = end_{u_1} = end_{v_1} = False$. Since $Ask(v_1) = x_2$, we have $x_2 \in \{\alpha_{v_1}, \beta_{v_1}\}$. Let us assume $end_{x_2} = True$. Then $x_2 \notin BestRematch(v_1)$ and then $v_1$ is elligible for an $Update$. Thus $end_{x_2} = False$.

Therefore, $C$ is a configuration such that $u_1$ is $First$ and $v_1$ is $Second$ with $end_{x_1} = end_{u_1} = end_{v_1} = end_{x_2} = False$. Now we are going to show there exists another augmenting path $(x_2, u_2, v_2, x_3)$ with $end_{x_2} = end_{u_2} = end_{v_2} = end_{x_3} = False$ and $p_{u_2} = x_2$, $p_{x_2} = u_2$, $p_{v_2} = x_3$ and $p_{x_3} \neq v_2$ such that $u_2$ is $First$ and $v_2$ is $Second$ (see Figure 8).

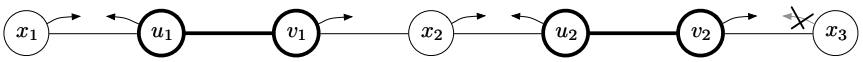



Figure 8: A chain of 3-augmenting paths.

$p_{x_2} \neq null$ otherwise $x_2$ is elligible for an $UpdateP$ rule. Thus there exists a vertex $u_2 \neq v_1$ such that $p_{x_2} = u_2$. From Lemma 3, $u_2 \in matched(N(x_2))$ and $p_{u_2} = x_2$. Therefore, there exists a node $v_2 = m_{u_2}$. From Lemma 4, we can deduce that $p_{v_2} \neq null$ and there exists a node $x_3$ such that $p_{v_2} = x_3$. $x_3 \in single(N(v_2))$ otherwise $v_2$ is eligible for an $Update$ rule. Finally, if $p_{x_3} = v_2$, then Lemma 5 implies that $end_{x_2} = end_{a_2} = end_{b_2} = end_{x_3} = True$. This yields the contradiction with the fact $end_{x_2} = False$. So, we have $p_{x_3} \neq v_2$.

We can then conclude that $(x_2, u_2, v_2, x_3)$ is a 3-augmenting path 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, it has 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 the graph is finite some $x_k$ must be equal to some $x_\ell$ with $\ell \neq k$ which contradicts the fact that the identifier' sequence is strictly increasing. □

**Lemma 7.** *Let $(x, u, v, y)$ be a 3-augmenting path on $(G, \mathcal{M})$. Let $C$ be a configuration. If $p_u = x \wedge p_x \neq u \wedge p_v = y \wedge p_y \neq v$ holds in $C$, then $C$ is not stable.*

*Proof.* By contradiction, assume $C$ is stable. From Lemma 3, $Ask(u) = x$. Assume to begin that $AskFirst(u) \neq null$. Because $p_{p_u} \neq u$ we have $s_u = False$, otherwise $u$ is eligible for $MatchFirst$. Since $AskSecond(v) \neq null$ and $s_{m_v} = s_u = False$ then $v$ can apply the $ResetMatch$ rule which yields a contradiction. Therefore assume that $AskSecond(u) \neq null$. The situation is symmetric (because now $AskFirst(v) \neq null$) and therefore we get the same contradiction as before. □

**Lemma 8.** *Let $(x, u, v, y)$ be a 3-augmenting path on $(G, \mathcal{M})$. Let $C$ be a configuration. If $p_y = p_u = p_v = p_y = null$ holds in $C$, then $C$ is not stable.*

*Proof.* By contradiction, assume $C$ is stable. $end_x = False$ (resp. $end_y = False$), otherwise $x$ (resp. $y$) is eligible for a $ResetMatch$. $(\alpha_u, \beta_u) = BestRematch(u)$ (resp. $(\alpha_v, \beta_v) = BestRematch(v)$), otherwise $u$ (resp. $v$) is eligible for an $Update$. Thus, there is at least an available single node for $u$ and $v$ and so $Ask(u) \neq null$ and $Ask(v) \neq null$. Then, this contradicts the fact that $Ask(u) = null$ (see Lemma 3). □

**Theorem 2.** *In a stable configuration we have, $\forall (u, v) \in \mathcal{M}$: (i) $p_u = p_v = null$ or (ii) $\exists x \in single(N(u))$, $\exists y \in single(N(v))$ 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 4 says the configuration cannot be stable if exactly 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 7 says the configuration cannot be stable. If $p_x = u$ and $p_y \neq v$ then Lemma 6 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 $x = y$]: $Ask(u) \neq null$ (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 9.** *Let $x$ be a single node. In a stable configuration, if $p_x = u, u \neq null$ then there 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 3, 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 2 the result holds. □

**Corollary 2.** *In a stable configuration, there is no 3-augmenting path on $(G, \mathcal{M}^+)$ left.*

*Proof.* Let us assume in a stable configuration, there is a 3-augmenting path on $(G, \mathcal{M}^+)$. By Theorem 2, any remaining augmenting-path $(x, u, v, y)$ contains an edge $(u, v) \in \mathcal{M}^+$ such that $p_u = p_v = null$. From Lemma 8, $p_x \neq null \vee p_y \neq null$. W.l.o.g let us assume $p_x \neq null$. By Lemma 9, $(x, p_x) \in \mathcal{M}^+$ and so $(x, u, v, y)$ is not an augmenting path. Contradiction. □

## 8. Convergence Proof

This section is devoted to a sketch of the convergence proof. In the following, $\mu$ will denote the number of matched nodes and $\sigma$ 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 11.** *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 $\exists x \in single(N(u)) : p_u = x \wedge p_x = u$.*

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 time $u$ writes *True* in $end_u$ means that a 3-augmenting path involving $u$ has been fully exploited.

**Theorem 3.** *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 times a single node can change its *end*-value (Lemma 21) and then we deduce the number of times a matched node can execute *Update* (Corollary 5). 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 4).

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 $\sigma$ 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 $6\mu$ modifications.

**Lemma 21.** *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 12). 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 5.** *Matched nodes can execute at most $\Delta(\sigma + 6\mu) + \mu$ times the* Update *rule.*

Third, we consider two particular matched nodes $u$ and $v$ and an execution with no *Update* rule performed by these two nodes. Then we count the maximal number of moves performed by these two nodes in this execution. The idea is that in such an execution, the $\alpha$ and $\beta$ values of $u$ and $v$ remain constant. Thus, in these small executions, $u$ and $v$ detect at most one augmenting path and perform at most one rematch attempt. We obtain that the maximal number of moves of $u$ and $v$ in these small executions is 12. By the previous remark and Corollary 5, we obtain:

**Theorem 4.** *In any execution, matched nodes can execute at most $12\Delta(\sigma + 6\mu) + 18\mu$ 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 $12\Delta(\sigma + 6\mu) + 18\mu + 1$.

**Corollary 3.** *The algorithm* PolyMatch *converges in $O(n^2)$ moves under the adversarial distributed daemon and in a general graph, provided that an underlying maximal matching has been initially built.*

The Manne *et al.* algorithm [19] builds a self-stabilizing maximal matching under the adversarial distributed daemon in a general graph, in $O(m)$ moves. This leads to a $O(m.n^2)$ moves complexity to build a 1-maximal matching with our algorithm without any assumption of an underlying maximal matching.

Now, the next section is devoted to the description of the technical proof.

### *8.1. A matched node can write True in its end-variable at most twice*

The first three lemmas are technical lemmas.

**Lemma 10.** *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.*

*Proof.* Let $C_0 \mapsto C_1$ be the transition in $\mathcal{E}$ or in its prefix in which $u$ executed a rule for the last time before $C$. Observe first that this transition necessarily exists by definition of $\mathcal{E}$ and second that $C$ may be equal to $C_1$. The executed rule is necessarily a *match* rule, otherwise $end_u$ could not be $True$ in $C_1$. If it is a $MatchSecond$ the lemma holds since in that case $s_u$ is a copy of $end_u$. Assume now it is a $MatchFirst$. For $end_u$ to be $True$ in $C_1$, $p_u = AskFirst(u) \wedge p_{p_u} = u \wedge p_{m_u} = AskSecond(m_u)$ must hold in $C_0$, according to the action of *MatchFirst*. This implies that $u$ writes $True$ in $s_u$ in transition $C_0 \mapsto C_1$. □

**Lemma 11.** *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 $\exists x \in single(N(u)) : p_u = x \wedge p_x = u$.*

*Proof.* Consider transition $C_0 \mapsto C_1$ in which $u$ executed a rule for the last time before $C$. Observe that this transition necessarily exists by definition of $\mathcal{E}$. The executed rule is necessarily a *match* rule, otherwise $s_u$ could not be $True$ in $C_1$. Observe now that whichever *match* rule is applied, $Ask(u) \neq null$ – let us assume $Ask(u) = x$ – and $p_u = x$ and $p_x = u$ must hold in $C_0$ for $s_u$ to be $True$ in $C_1$. $p_u = x$ still holds in $C_1$ and until $C$. Moreover, $x$ must be in $single(N(u))$, otherwise $u$ would have executed an *Update* instead of a match rule in $C_0 \mapsto C_1$, since *Update* has the higest priority among all rules. Finally, in transition $C_0 \mapsto C_1$, $x$ cannot execute $UpdateP$ nor $ResetEnd$ since $p_x \in matched(N(x)) \wedge p_{p_x} = x$ holds in $C_0$. Thus in $C_1$, $p_u = x$ and $p_x = u$ holds. Using the same argument, $x$ cannot execute $UpdateP$ nor $ResetEnd$ between configurations $C_1$ and $C$. Thus $p_u = x \wedge p_x = u$ in $C$. □

**Lemma 12.** *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.*

*Proof.* Let $C_2$ be any configuration in $\mathcal{E}$ such that $C_2 \geq C_1$. Let $C_{10} \mapsto C_{11}$ be the last transition before $C_2$ in which $u$ executes a move. Notice that by definition of $\mathcal{E}$, this transition exists. Assume by contradiction that one of the following predicates holds in $C_2$.

1. $(\alpha_u > \beta_u) \vee (\alpha_u, \beta_u \notin (single(N(u)) \cup \{null\})) \vee (\alpha_u = \beta_u \wedge \alpha_u \neq null)$
2. $p_u \notin (single(N(u)) \cup \{null\})$

By definition between $C_{11}$ and $C_2$, $u$ does not execute rules. To modify the variables $\alpha_u, \beta_u$ and $p_u$, $u$ must execute a rule. Thus one of the two predicates also holds in $C_{11}$.

We first show that if predicate (1) holds in $C_{11}$ then we get a contradiction. If $u$ executes an *Update* rule in transition $C_{10} \mapsto C_{11}$, then by definition of the $BestRematch$ function, predicate (1) cannot hold in $C_{11}$ (observe that the only way for $\alpha_u = \beta_u$ is when $\alpha_u = \beta_u = null$). Thus assume that $u$ executes a *match* or $ResetMatch$ rule. Notice that these rules do not modify the value of the $\alpha_u$ and $\beta_u$ variables. This implies that if $u$ executes one of these rules in $C_{10} \mapsto C_{11}$, predicate (1) not only hold in $C_{11}$ but also in $C_{10}$. Observe that this implies, in that case that $u$ is eligible for *Update* in $C_{10} \mapsto C_{11}$, which gives the contradiction since *Update* is the rule with the highest priority among all rules.

Now assume predicate (2) holds in $C_{11}$. In transition $C_{10} \mapsto C_{11}$, $u$ cannot execute *Update* nor $ResetMatch$ as this would imply that $p_u = null$ in $C_{11}$. Assume that in $C_{10} \mapsto C_{11}$ $u$ executes a *match* rule. Since in $C_{11}$, $p_u \notin (single(N(u)) \cup \{null\})$ this implies that in $C_{10}$, $Ask(u) \notin (single(N(u)) \cup \{null\})$. This implies that $\alpha_u, \beta_u \notin (single(N(u)) \cup \{null\})$ in $C_{10}$. Thus $u$ is eligible for *Update* in transition $C_{10} \mapsto C_{11}$ and this yields the contradiction since *Update* is the rule with the highest priority among all rules.

Since these two predicates cannot hold in $C_2$, this concludes the proof. □

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

**Lemma 13.** *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 13 is called a $stop_{uv}$ configuration. From such a configuration neither $u$ nor $v$ will ever be eligible for any rule. In Lemmas 15 and 16, 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 15 shows that, if $u$ is First in $C_0$, then $C_1$ is a $stop_{um_u}$ configuration. Lemma 16 shows that, if $u$ is Second in $C_0$, then either $C_1$ is a $stop_{um_u}$ configuration or there exists a configuration $C_3$ such that $C_3 > C_1$, $u$ does not make any move from $C_1$ to $C_3$ and $C_3$ is a $stop_{um_u}$ configuration. Lemma 14 and Corollary 4 are required to prove Lemmas 15 and 16.

**Lemma 14.** *Let $(u, v)$ be a matched edge. Let $\mathcal{E}$ be some execution in which $v$ does not execute any rule. If there 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 $\mathcal{E}$:

- $\alpha_u$, $\alpha_v$, $\beta_u$ and $\beta_v$ 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 $\alpha$ and $\beta$ 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 4.** *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 15.** *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 10 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 12, Point 1c holds and will always hold. By Corollary 4, $u$ 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 11, $\exists x \in single(N(v)) : p_v = x \wedge p_x = v$. 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 12, 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 16.** *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) $s_v = True \wedge p_v = AskFirst(v)$; (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) $s_u = end_u = True$; (b) $p_v \in single(N(v)) \wedge p_v = AskFirst(v) \wedge p_{p_v} = v$;*
   *(c) $p_u \in single(N(u)) \wedge p_u = AskSecond(u) \wedge p_{p_u} = u$; (d) $s_v = True$;*
4. *$u$ is not eligible for any move in $C_1$;*
5. *If $end_v = False$ 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) $s_u = end_u = True$; (ii) $p_v \in single(N(v)) \wedge p_v = AskFirst(v) \wedge p_{p_v} = v$;*
   *(iii) $p_u \in single(N(u)) \wedge p_u = AskSecond(u) \wedge p_{p_u} = u$; (iv) $s_v = True$;*
   *(v) $u$ does not execute moves between $C_1$ and $C_3$; (vi) $end_v = True$;*

*Proof.* We show Point 1a. For $u$ to write $True$ in transition $C_0 \mapsto C_1$, $u$ executes a $MatchSecond$ in this transition. Thus $s_v = True$ must hold in $C_0$ and $p_v = AskFirst(v)$ as well. By Corollary 4, $u$ cannot write $True$ consecutively if $v$ does not execute any move. Thus at some point before $C_0$, $v$ applied some rule. Thus, and by Lemma 11, $\exists x \in single(N(v)) : p_v = x \wedge p_x = v$ in configuration $C_0$, so Point 1b holds.

As $AskFirst(v) \neq null$ in $C_0$, $v$ is not eligible for *ResetMatch* in $C_0$. We prove now that $v$ is not eligible for *Update*. By Corollary 4 and Lemma 12, line 1 of the *Update* guard does not hold in $C_0$. Finally, according to Point 1b, the second line of the *Update* guard does not hold, which concludes Point 2.

We consider now Point 3a. In $C_1$, $s_u = end_u = True$ holds because, executing a $MatchSecond$, $u$ writes $True$ in $end_u$ and writes $end_u$ in $s_u$ during transition $C_0 \mapsto C_1$.

We now show Point 3b. $AskFirst(v)$ and $AskSecond(u)$ remain constant in $C_0 \mapsto C_1$ since neither $u$ nor $v$ execute an *Update* in this transition. Moreover, the only rule $v$ can execute in $C_0 \mapsto C_1$ is a $MatchFirst$, according to Point 2. Thus $v$ does not change its $p$-value in $C_0 \mapsto C_1$ and so $p_v = AskFirst(v)$ in $C_1$. Now, in $C_0$, $v \in matched(N(p_v)) \wedge p_{p_v} = v$ thus $p_v$ cannot execute $UpdateP$ in $C_0 \mapsto C_1$ and thus it cannot change its $p$-value. So, $p_{p_v} = v$ in $C_1$.

Point 3c holds since after $u$ executed a $MatchSecond$ in $C_0 \mapsto C_1$, observe that necessarily $p_u = AskSecond(u)$ in $C_1$. Moreover, $s_u = True$ in $C_1$ so, according to Lemma 11, $\exists y \in single(N(u)) : p_u = y \wedge p_y = u$ in $C_1$.

$p_v = AskFirst(v)$ and $p_{p_v} = v$ hold in $C_0$, according to Points 1a and 1b. Moreover, $p_u = AskSecond(u)$ holds in $C_0$ since $u$ writes $True$ in $end_u$ while executing a $MatchSecond$ in $C_0 \mapsto C_1$. Finally, by Point 2, $v$ can only execute $MatchFirst$ in $C_0 \mapsto C_1$, thus variable $s_v$ remains $True$ in transition $C_0 \mapsto C_1$ and Point 3d holds.

We now prove Point 4. If $end_v = True$ in $C_1$, then according to Lemma 13, $u$ is not eligible for any rule in $C_1$. Now, let us consider the case $end_v = False$ in $C_1$. By Points 3c and 3d, $u$ is not eligible for $ResetMatch$. By Point 3c and Lemma 12, $u$ is not eligible for $Update$. By Points 3a, 3b and 3c, $u$ is not eligible for $MatchSecond$. Finally, since $u$ is Second in $C_1$, $u$ is not eligible for $MatchFirst$ neither and Point 4 holds.

Now since between $C_1$ and $C_2$, $v$ does not execute any rule (by Point 5b), and since $p_u$ (resp. $p_v$) is not eligible for $UpdateP$ while $u$ (resp. $v$) does not move (because $p_{p_u} = u$ (resp. $p_{p_v} = v$)), then $Ask(u), Ask(v)$, $p_{p_u}$ and $p_{p_v}$ remain constant while $u$ does not make any move. And so, properties 3a, 3b, 3c and 3d hold for any configuration between $C_1$ and $C_2$, thus $u$ is not eligible for any rule between $C_1$ and $C_2$ and $u$ will not execute any move from $C_1$ to $C_3$. Moreover, the $end_v$-value is the same from $C_1$ to $C_2$.

If $end_v = False$ in $C_2$, then $v$ is eligible for a $MatchFirst$ and it will write $True$ in its $end_v$-variable while all properties of Point 3 will still hold in $C_3$. Thus Point 5 holds. □

**Theorem 3.** *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 13 and 15, (resp. 16), from $C_1$, neither $u$ nor $v$ will ever be eligible for any rule. □

*8.2. The number of times single nodes can change their end-variable*

In the following, $\mu$ denote the number of matched nodes and $\sigma$ the number of single nodes.

**Lemma 17.** *Let $x$ be a single node. If $x$ writes $True$ in $end_x$ in some transition $C_0 \mapsto C_1$ then, in $C_0$, $\exists u \in matched(N(x)) : p_x = u \wedge p_u = x \wedge end_x = False \wedge end_u = True$.*

*Proof.* To write True in its *end* variable, a single node must apply $UpdateEnd$. Observe now that to apply this rule, the conditions described in the Lemma must hold. □

**Lemma 18.** *Let $u$ be a matched node. Consider an execution $\mathcal{E}$ starting after $u$ executed some rule and in which $end_u$ is always $True$, except for the last configuration $D$ of $\mathcal{E}$ in which it may be $False$. Let $\mathcal{E}\backslash D$ be all configurations of $\mathcal{E}$ but configuration $D$. In $\mathcal{E}\backslash D$, the following holds:*
*(a) $p_u \in single(N(u))$;  (b) $p_u$ remains constant.*

*Proof.* Since $end_u = True$ in $\mathcal{E}\backslash D$, the last rule executed before $\mathcal{E}$ is necessarily a $Match$ rule. So, at the beginning of $\mathcal{E}$, $p_u \in single(N(u))$, otherwise, $u$ would not have executed a $Match$ rule, but an $Update$ instead. We prove now that in $\mathcal{E}\backslash D$, $p_u$ remains constant. Assume by contradiction that there exists a transition in which $p_u$ is modified. Let $C_0 \mapsto C_1$ be the first such transition. First, observe that in $\mathcal{E}\backslash D$, $u$ cannot execute $ResetMatch$ nor $Update$ since that would set $end_u$ to $False$. Thus $u$ must execute a $Match$ rule in $C_0 \mapsto C_1$. Since the value of $p_u$ changes in this transition, this implies that $Ask(u) \neq p_u$ in $C_0$. Thus, whatever the $Match$ rule, observe now that in $C_1$, $end_u$ must be $False$, which gives a contradiction and concludes the proof. □

**Definition 6.** *Let $u$ be a matched node. We say that a transition $C_0 \mapsto C_1$ is of type* "a single copies True from $u$" *if there exists a single node $x$ such that $(p_x = u \wedge p_u = x \wedge end_x = False)$ in $C_0$ and $end_x = True$ in $C_1$. Notice that by Lemma 17, $end_u = True$ in $C_0$ and $x \in single(N(u))$.*

*If a transition $C_0 \mapsto C_1$ is of type* "a single node copies True from $u$" *and if $x$ is the single node with $(p_x = u \wedge p_u = x \wedge end_x = False)$ in $C_0$ and $end_x = True$ in $C_1$, then we will say $x$* copies $True$ from $u$.

**Lemma 19.** *Let $u$ be a matched node and $\mathcal{E}$ be an execution. In $\mathcal{E}$, there are at most three transitions of type* "a single copies $True$ from $u$".

*Proof.* Let $\mathcal{E}$ be an execution. We consider some sub-executions of $\mathcal{E}$.

Let $\mathcal{E}_{init}$ be a sub-execution of $\mathcal{E}$ that starts in the initial configuration of $\mathcal{E}$ and that ends just after the first move of $u$. Let $C_0 \mapsto C_1$ be the last transition of $\mathcal{E}_{init}$. Observe that $u$ does not execute any move until configuration $C_0$ and executes its first move in transition $C_0 \mapsto C_1$. We will write $\mathcal{E}_{init} \setminus C_1$ to denote all configurations of $\mathcal{E}_{init}$ but the configuration $C_1$. We prove that there is at most one transition of type *"a single copies $True$ from $u$"* in $\mathcal{E}_{init}$.

There are two possible cases regarding $end_u$ in all configuration of $\mathcal{E}_{init} \setminus C_1$: either $end_u$ is always $True$ or $end_u$ is always $False$. If $end_u = False$ then by Definition 6, no single node can copy $True$ from $u$ in $\mathcal{E}_{init}$, not even in transition $C_0 \mapsto C_1$, since no single node is eligible for such a copy in $C_0$. If $end_u = True$, once again, there are two cases: either (i) $(p_u = null \vee p_u \notin single(N(u)))$ in all configuration of $\mathcal{E}_{init} \setminus C_1$, or (ii) $(p_u \in single(N(u)))$ in $\mathcal{E}_{init} \setminus C_1$. In case (i) then by Definition 6 no single node can copy $True$ from $u$ in $\mathcal{E}_{init}$, not even in $C_0 \mapsto C_1$. In case (ii), observe that $p_u$ remains constant in all configurations of $\mathcal{E}_{init} \setminus C_1$, thus at most one single node can copy $True$ from $u$ in $\mathcal{E}_{init}$.

Let $\mathcal{E}_{true}$ be a sub-execution of $\mathcal{E}$ starting after $u$ executed some rule and such that: for all configurations in $\mathcal{E}_{true}$ but the last one, $end_u = True$. There is no constraint on the value of $end_u$ in the last configuration of $\mathcal{E}_{true}$. According to Lemma 18, $p_u \in single(N(u))$ and $p_u$ remains constant in all configurations of $\mathcal{E}_{true}$ but the last one. This implies that at most one single can copy $True$ from $u$ in $\mathcal{E}_{true}$.

Let $\mathcal{E}_{false}$ be an execution starting after $u$ executed some rule and such that: for all configurations in $\mathcal{E}_{false}$ but the last one, $end_u = False$. There is no constraint on the value of $end_u$ in the last configuration of $\mathcal{E}_{false}$. By Definition 6, no single node will be able to copy $True$ from $u$ in $\mathcal{E}_{false}$.

To conclude, by Corollary 3, $u$ can write $True$ in its *end* variable at most twice. Thus, for all executions $\mathcal{E}$, $\mathcal{E}$ contains exactly one sub-execution of type $\mathcal{E}_{init}$, and at most two sub-executions of type $\mathcal{E}_{true}$ and the remaining sub-executions are of type $\mathcal{E}_{false}$. This implies that in total, we have at most three transitions of type "*a single copies $True$ from $u$*" in $\mathcal{E}$. □

**Lemma 20.** *In any execution, the number of transitions where a single node writes $True$ in its end variable is at most $3\mu$.*

*Proof.* Let $\mathcal{E}$ be an execution and $x$ be a single node. If $x$ writes $True$ in $end_x$ in some transition of $\mathcal{E}$, then $x$ necessarily executes an *UpdateEnd* rule and by Definition 6, this means *$x$ copies $True$ from some matched node* in this transition. Now the lemma holds by Lemma 19. □

**Lemma 21.** *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.*

*Proof.* A single node can write $True$ in its *end* variable at most $3\mu$ times, by Corollary 20. Each of this writing allows one writing from $True$ to $False$, which leads to $6\mu$ possible modifications of the *end* variables. Now, let us consider a single node $x$. If $end_x = False$ initially, then no more change is possible, however if $end_x = True$ initially, then one more modification from $True$ to $False$ is possible. Each single node can do at most one modification due to this initialization and thus the Lemma holds. □

*8.3. How many Update in an execution?*

**Definition 7.** *Let $u$ be a matched node and $C$ be a configuration. We define $Cand(u, C) = \{x \in single(N(u)) : (p_x = u \ \vee \ end_x = False)\}$ which is the set of vertices considered by the function $BestRematch(u)$ in configuration $C$.*

**Lemma 22.** *Let $u$ be a matched node. If there exists a transition $C_0 \mapsto C_1$ such that the value of $BestRematch(u)$ is not the same in $C_0$ and in $C_1$, then there exists a single node $x$ such that $x \in Cand(u, C_0) \setminus Cand(u, C_1)$ or $x \in Cand(u, C_1) \setminus Cand(u, C_0)$. Moreover, in transition $C_0 \mapsto C_1$, $x$ flips the value of its end variable.*

*Proof.* We prove the first point by contradiction. Since $BestRematch(u)$ is a deterministic function over $Cand(u, C)$ for some configuration $C$, so if $Cand(u, C_0) = Cand(u, C_1)$ then the value of $BestRematch(u)$ is the same in $C_0$ and $C_1$ which yields the contradiction.

For the second point, we first consider the case $x \in Cand(u, C_1)$ and $x \notin Cand(u, C_0)$. Necessarily $end_x = True \wedge p_x \neq u$ in $C_0$ and $end_x = False \vee p_x = u$ in $C_1$. If $p_x = u$ in $C_1$ then in transition $C_0 \mapsto C_1$, $x$ has executed an $UpdateP$ and the second point holds. Assume now that $p_x \neq u$ in $C_1$. Necessarily $end_x = False$ in $C_1$ and the Lemma holds.

We consider the second case in which $x \notin Cand(u, C_1)$ and $x \in Cand(u, C_0)$. Necessarily in $C_1$, $p_x \neq u$ and $end_x = True$. Thus if $end_x = False$ in $C_0$ the lemma holds. Assume by contradiction that $end_x = True$ in $C_0$. This implies $p_x = u$ in $C_0$. But since in $C_1$ $p_x \neq u$ then $x$ executed $UpdateP$ in $C_0 \mapsto C_1$ which implies $end_x = False$ in $C_1$, a contradiction. This completes the proof. □

**Corollary 5.** *Matched nodes can execute at most $\Delta(\sigma + 6\mu) + \mu$ times the Update rule.*

*Proof.* Let $u$ be a matched node. Initially each matched node can be eligible for an *Update*. Thus, let us consider a sub-execution $\mathcal{E}$ staring after $u$ has executed a move and in which $BestRematch(u)$ remains constant. By Lemma 12, the first line of the *Update* rule is always false for $u$ in $\mathcal{E}$. So $u$ can execute the *Update* rule at most once in $\mathcal{E}$. So, for $u$ to become eligible again for an *Update* after $\mathcal{E}$, $BestRematch(u)$ must change and so, by Lemma 22, at least one single node must change the value of its *end* variable. Each change of the *end* value of a single node can generate at most $\Delta$ matched nodes to be eligible for an *Update*. By Lemma 21, the number of transitions where a single node changes the value of its *end* variables is at most $\sigma + 6\mu$ times. Thus at most $\Delta(\sigma + 6\mu)$ *Update* generated by a change of the *end* value of a single node and the Lemma holds. □

*8.4. A bound on the total number of moves in any execution*

**Definition 8.** *Let $(u, v)$ be a matched edge. In the following, we call $\mathcal{F}$, a finite execution where neither $u$ nor $v$ execute the Update rule. Let $D_{\mathcal{E}}$ be the first configuration of $\mathcal{F}$ and $D'_{\mathcal{E}}$ be the last one.*

Observe that in the execution $\mathcal{F}$, all variables $\alpha$ and $\beta$ of nodes $u$ and $v$ remain constant and thus, predicates $AskFirst$ and $AskSecond$ for these two nodes remain constant too.

**Lemma 23.** *If $Ask(u) = Ask(v) = null$ in $\mathcal{F}$, then $u$ and $v$ can both execute at most one $ResetMatch$.*

*Proof.* In the execution $\mathcal{F}$, by definition, $u$ and $v$ do not execute the *Update* rule. Moreover, these two nodes are not eligible for *Match* rules since $Ask(u) = Ask(v) = null$. Thus they are only eligible for $ResetMatch$. Observe now it is not possible to execute tis rule twice in a row, which completes the proof. □

**Lemma 24.** *Assume that in $\mathcal{F}$, $u$ is $First$ and $v$ is Second. If $s_u$ is False in all configurations of $\mathcal{F}$ but the last one, then $v$ can execute at most one rule in $\mathcal{F}$.*

*Proof.* Since $s_u$=$False$ in all configurations of $\mathcal{F}$ but the last one, node $v$ which is *Second* can only be eligible for $ResetMatch$. If $v$ executes $ResetMatch$, it is not eligible for a rule anymore and the Lemma holds. □

**Lemma 25.** *Assume that in $\mathcal{F}$, $u$ is $First$ and $v$ is Second. If $s_u$ is False throughout $\mathcal{F}$, then $u$ can execute at most one rule in $\mathcal{F}$.*

*Proof.* Node $u$ can only be eligible for $MatchFirst$. Assume $u$ executes $MatchFirst$ for the first time in some transition $C_0 \mapsto C_1$, then in $C_1$, necessarily, $p_u = AskFirst(u)$, $s_u = False$ (by hypothesis) and $end_u = False$ by Lemma 10. Let $\mathcal{F}_1$ be the execution starting in $C_1$ and finishing in $D'_{\mathcal{E}}$. Since in $\mathcal{F}_1$, there is no *Update* of nodes $u$ and $v$, observe that $p_u = AskFirst(u)$ remains $True$ in this execution. Assume by contradiction that $u$ executes another $MatchFirst$ in $\mathcal{F}_1$. Consider the first transition $C_2 \mapsto C_3$ after $C_1$ when it executes this rule. Notice that between $C_1$ and $C_2$ it does not execute rules. Thus in $C_2$, $p_u = AskFirst(u)$, $s_u = False$ and $end_u = False$ hold. Now if $u$ executes $MatchFirst$ in $C_2$ it is necessarily to modify the value of $s_u$ or $end_u$. By definition, it cannot change the value of $s_u$. Moreover it cannot modify the value of $end_u$ as this would imply by Lemma 10 that $s_u = True$ in $C_3$. This completes the proof. □

**Lemma 26.** *Let $(u, v)$ be a matched edge. Assume that in $\mathcal{F}$, $u$ is First, $v$ is Second and that $u$ writes True in $s_u$ in some transition of $\mathcal{F}$. Let $C_0 \mapsto C_1$ be the transition in $\mathcal{F}$ in which $u$ writes True in $s_u$ for the first time. Let $\mathcal{F}_1$ be the execution starting in $C_1$ and finishing in $D'_{\mathcal{E}}$. In $\mathcal{F}_1$, $u$ can apply at most 3 rules and $v$ at most 2.*

*Proof.* We first prove that in $\mathcal{F}_1$, $s_u$ remains $True$. Observe that $u$ cannot execute $Update$ neither $ResetMatch$ since it is $First$. So $u$ can only execute $MatchFirst$ in $\mathcal{F}_1$. For $u$ to write $False$ in $s_u$, there must exist a configuration in $\mathcal{F}_1$ such that $p_u \neq AskFirst(u) \vee p_{p_u} \neq u \vee p_v \notin \{AskSecond(v), null\}$. Let us prove that none of these cases are possible.

Since $u$ executed $MatchFirst$ in transition $C_0 \mapsto C_1$ writting $True$ in $s_u$ then, by definition of this rule, $p_u = AskFirst(u) \wedge p_{p_u} = u \wedge p_v \in \{AskSecond(v), null\}$ holds in $C_0$. As there is no $Update$ of $u$ and $v$ in $\mathcal{F}$, then $AskFirst(u)$ and $AskSecond(v)$ remain constant throughout $\mathcal{F}$ (and $\mathcal{F}_1$). So each time $u$ executes a $MatchFirst$, it writes the same value $AskFirst(u)$ in its $p$-variable. Thus $p_u = AskFirst(u)$ holds throughout $\mathcal{F}_1$. Moreover, each time $v$ executes a rule, it writes either $null$ or the same value $AskSecond(v)$ in its $p$-variable. Thus $p_v \in \{AskSecond(v), null\}$ holds throughout $\mathcal{F}_1$. Now by Lemma 11, in $C_1$ we have, $\exists x \in single(N(u)) : p_u = x \wedge p_x = u$, since $s_u = True$ . This stays $True$ in $\mathcal{F}_1$ as $p_u$ remains constant and $x$ will then not be eligible for $UpdateP$ in $\mathcal{F}_1$. Thus $p_{p_u} = u$ holds throughout $\mathcal{F}_1$. Thus, $p_u = AskFirst(u) \wedge p_{p_u} = u \wedge p_v \in \{AskSecond(v), null\}$ holds throughout $\mathcal{F}_1$ and so $s_u = True$ throughout $\mathcal{F}_1$.

This implies that in $\mathcal{F}_1$, $v$ is only eligible for $MatchSecond$. The first time it executes this rule in some transition $B_0 \mapsto B_1$, with $B_1 \geq C_1$, then in $B_1$, $p_v = AskSecond(v)$, $s_v = end_v$ and this will hold between $B_1$ and $D'_{\mathcal{E}}$. If $end_v = True$ in $B_1$ then this will stay $True$ between $B_1$ and $D'_{\mathcal{E}}$. Indeed, $p_v$ is not eligible for $UpdateP$ and we already showed that $p_u = AskFirst(u)$ holds in $\mathcal{F}_1$. In that case, between $B_1$ and $D'_{\mathcal{E}}$, $v$ will not be eligible for any rule and so $v$ will have executed at most one rule in $\mathcal{F}_1$. In the other case, that is $end_v (= s_v) = False$ in $B_1$, since $p_v = AskSecond(v)$ holds between $B_1$ and $D'_{\mathcal{E}}$, necessarily, the next time $v$ executes a $MatchSecond$ rule, it is to write $True$ in $end_v$. After that observe that $v$ is not eligible for any rule. Thus, $v$ can execute at most 2 rules in $\mathcal{F}_1$.

To conclude the proof it remains to count the number of moves of $u$ in $\mathcal{F}_1$. Recall that we proved $s_u$ is always $True$ in $\mathcal{F}_1$. Thus whenever $u$ executes a $MatchFirst$, it is to modify the value of its $end$ variable. Observe that this value depends in fact of the value of $end_v$ and of $p_v$ since we proved $p_u = AskFirst(u) \wedge p_{p_u} = u \wedge s_u \wedge p_v \in \{AskSecond(v), null\}$ holds throughout $\mathcal{F}_1$. Since we proved that in $\mathcal{F}_1$, $v$ can execute at most two rules, this implies that these variables can have at most three different values in $\mathcal{F}_1$. Thus $u$ can execute at most 3 rules in $\mathcal{F}_1$. □

**Lemma 27.** *Assume that in $\mathcal{F}$, $u$ is First and $v$ is Second. If $s_u$ is True throughout $\mathcal{F}$ and if $u$ does not execute any move in $\mathcal{F}$, then $v$ can execute at most two rules in $\mathcal{F}$.*

*Proof.* By Definition 8, $v$ cannot execute $Update$ in $\mathcal{F}$. Since we suppose that in $\mathcal{F}$, $s_u = True$ then $v$ is not eligible for $ResetMatch$. Thus in $\mathcal{F}$, $v$ can only execute $MatchSecond$. After it executed this rule for the first time, $p_v = AskSecond(v)$ and $s_v = end_v$ will always hold, since $v$ is only eligible for $MatchSecond$. Thus the second time it executes this rule, it is necessarily to modify its $end_v$ and $s_v$ variables. Observe that after that, since $u$ does not execute rules, $v$ is not eligible for any rule. □

**Lemma 28.** *In $\mathcal{F}$, $u$ and $v$ can globally execute at most 12 rules.*

*Proof.* If $Ask(u) = Ask(v) = null$, the Lemma holds by Lemma 23. Assume now that $u$ is $First$ and $v$ $Second$. We consider two executions in $\mathcal{F}$.

Let $C_0 \mapsto C_1$ be the first transition in $\mathcal{F}$ in which $u$ executes a rule. Let $\mathcal{F}_0$ be the execution starting in $D_{\mathcal{E}}$ and finishing in $C_0$. There are two cases. If $s_u = False$ in $\mathcal{F}_0$ then $v$ is only eligible for $ResetMatch$ in this execution. Observe that after it executes this rule for the first time in $\mathcal{F}_0$, it is not eligible for any rule after that in $\mathcal{F}_0$. If $s_u = True$ in $\mathcal{F}_0$ then by Lemma 27, $v$ can execute at most two rules in this execution. In transition $C_0 \mapsto C_1$, $u$ and $v$ can execute one rule each.

Let $\mathcal{F}_1$ be the execution starting in $C_1$ and finishing in $D'_{\mathcal{E}}$. Whatever rule $u$ executes in transition $C_0 \mapsto C_1$ observe that $u$ either writes $True$ or $False$ in $s_u$. If $u$ writes $True$ in $s_u$ in transition $C_0 \mapsto C_1$, then by Lemma 26, $u$ and $v$ can execute at most five rules in total in $\mathcal{F}_1$.

Consider the other case in which $u$ writes $False$ in $C_1$. Let $C_2 \mapsto C_3$ be the first transition in $\mathcal{F}_1$ in which $u$ writes $True$ in $s_u$. Call $\mathcal{F}_{10}$ the execution between $C_1$ and $C_3$ and $\mathcal{F}_{11}$ the execution between $C_3$ and $D'_{\mathcal{E}}$. By definition, $s_u$ stays $False$ in $\mathcal{F}_{10} \backslash C_3$. Thus in $\mathcal{F}_{10} \backslash C_3$, $u$ can execute at most one rule, by Lemma 25. Now in $\mathcal{F}_{10}$, $u$ can execute at most two rules. By Lemma 24, $v$ can execute at most one rule in $\mathcal{F}_{10}$. In total, $u$ and $v$ can execute at most three rules in $\mathcal{F}_{10}$. In $\mathcal{F}_{11}$, $u$ and $v$ can execute at most five rules by Lemma 26. Thus in $\mathcal{F}_1$, $u$ and $v$ can apply at most eight rules. □

**Theorem 4.** *In any execution, matched nodes can execute at most* $12\Delta(\sigma + 6\mu) + 18\mu$ *rules.*

*Proof.* Let $k$ be the number of edges in the underlying maximal matching, $k = \frac{\mu}{2}$. For $i \in [1, .., k]$, let $\{(u_i, v_i) = a_i\}$ be the set of matched edges. By $Update(a_i)$ we denote an $Update$ rule executed by node $u_i$ or $v_i$. By Lemma 28, between two $Update(a_i)$ rules, nodes $u_i$ and $v_i$ can execute at most 12 rules. By Corollary 5, there are at most $\Delta(\sigma + 6\mu) + \mu$ executed $Update$ rules. Thus in total, nodes can execute at most $\sum_{i=1}^{k} 12 \times (\#Update(a_i) + 1) = 12 \sum_{i=1}^{k} \#Update(a_i) + 12 \sum_{i=1}^{k} 1 \leq 12(\Delta(\sigma + 6\mu) + \mu) + 12k$ rules □

**Lemma 29.** *In any execution, single nodes can execute at most* $\sigma$ *times the ResetEnd rule.*

*Proof.* We prove that a single node $x$ can execute the $ResetEnd$ rule at most once. Assume by contradiction that it executes this rule twice. Let $C_0 \mapsto C_1$ be the transition when it executes it the second time. In $C_0$, $end_x = True$, by definition of the rule. Since $x$ already executed a $ResetEnd$ rule, it must have some point wrote $True$ in $end_x$. This is only possible through an execution of $UpdateEnd$. Thus consider the last transition $D_0 \mapsto D_1$ in which it executed this rule. Observe that $D_1 \leq C_0$. Since between $D_1$ and $C_0$, $end_x$ remains $True$, observe that $x$ does not execute any rule between these two configurations. Now since in $D_1$, $p_x \neq null$ and this holds in $C_0$ then $x$ is not eligible for $ResetEnd$ in $C_0$, which gives the contradiction. This implies that single nodes can execute at most $\sigma$ times the $ResetEnd$ rule. □

**Lemma 30.** *In any execution, single nodes can execute at most* $\sigma + 6\mu$ *times the UpdateEnd rule.*

*Proof.* By Lemma 21, single nodes can change the value of their $end$ variable at most $\sigma + 6\mu$ times. Thus they can apply $UpdateEnd$ at most $\sigma + 6\mu$ times, since in every application of this rule, the value of the $end$ variable must change. □

**Lemma 31.** *In any execution, single nodes can execute* $O(\Delta(\sigma + \mu))$ *times the UpdateP rule.*

*Proof.* Let $x$ be a single node. Let $C_0 \mapsto C_1$ be a transition in which $x$ executes an $UpdateP$ rule and let $C_2 \mapsto C_3$ be the next transition after $C_1$ in which $x$ executes an $UpdateP$ rule. We prove that for $x$ to execute the $UpdateP$ rule in $C_2 \mapsto C_3$, a matched node had to execute a move between $C_0$ and $C_2$.

In $C_1$ there are two cases: either $p_x = null$ or $p_x \neq null$. Assume to begin that $p_x = null$. This implies that in $C_0$ the set $\{w \in N(x) | p_w = x\}$ is empty. In $C_2$, $p_x = null$, since between $C_1$ and $C_2$, $x$ can only apply $UpdateEnd$ or $ResetEnd$. Thus if it applies $UpdateP$ in $C_2$, necessarily $\{w \in N(x) | p_w = x\} \neq \emptyset$. This implies that a matched node must have executed a $Match$ rule between $C_1$ and $C_2$ and the lemma holds in that case. Consider now the case in which $p_x = u$ with $u \neq null$ in $C_1$. By definition of the $UpdateP$ rule, we also have $u \in matched(N(x)) \wedge p_u = x$ holds in $C_0$. In $C_2$ we still have that $p_x = u$ since between $C_1$ and $C_2$, $x$ can only execute $UpdateEnd$ or $ResetEnd$. Thus if $x$ executes $UpdateP$ in $C_2$, necessarily $p_{p_x} \neq x$. This implies that $p_u \neq x$ and so $u$ executed a rule between $C_0$ and $C_2$. Now, the lemma holds by Theorem 4. □

**Corollary 6.** *In any execution, nodes can execute at most* $\mathcal{O}(n^2)$ *moves.*

*Proof.* According to Lemmas 29, 30 and 31, single nodes can execute at most $\mathcal{O}(n^2)$ moves. Moreover, according to Theorem 4, matched nodes can execute at most $\mathcal{O}(n^2)$ moves. □

## 9. Composition

Recall that the algorithm PolyMatch assumes an underlying maximal matching. Let MaxMatch be a silent self-stabilizing maximal matching algorithm having a complexity of $z = O(f(n,m))$ moves under the distributed daemon. In this section, we prove the fair composition of MaxMatch and PolyMatch has a complexity in $O(zn^2)$ moves under the distributed daemon. The MaxMatch algorithm could be for instance the Manne *et al.* algorithm [19] that has a complexity in $O(m)$ moves that would lead to a final complexity of the composition in $O(n^2.m)$ moves.

An execution of such a composition is an alternated concatenation of two kinds of finite sub executions. The first kind only contains actions from PolyMatch and the second one contains any action. Les us call the first kind of executions, the PolyMatch-*sub-executions*. We do not know any upper bound on the number of moves of a PolyMatch-sub-executions in the case where MaxMatch has not stabilized yet. Indeed, Corollary 6 only says that such a sub-execution contains $O(n^2)$ moves when MaxMatch is already stabilized.

Let $G$ be the input graph and $e$ be an execution on $G$ of the composition between MaxMatch and PolyMatch. We consider two transitions of $e$ containing a move from MaxMatch and such that there is no more move from it between them. Let us call $\mathcal{E}$ the execution between these two transitions and let $C_0$ be the first configuration of $\mathcal{E}$ ($\mathcal{E}$ is a PolyMatch-sub-execution). In $C_0$, if MaxMatch is in a stable configuration, then the underlying matching is maximal in $\mathcal{E}$, and every node $u$ is either *single* (*i.e.*, $m_u = \perp$) or *matched* (*i.e.*, $\exists v \in N(u)$ such that $m_u = v \wedge m_v = u$). However, MaxMatch is not necessarily in a stable configuration in $C_0$. Therefore, in that case, a node can also be *falsy-matched* in $\mathcal{E}$, *i.e.*, $\exists v \in N(u)$ such that $m_u = v \wedge m_v \neq u$. Observe that a node $x$ cannot decide if a neighbor $u$ is *matched* or *falsy-matched* since it can only check if $m_u \neq \perp$ but cannot read the variable $m_{m_u}$. Thus, all *falsy-matched* nodes are considered as *matched* by their neighbors. Finally, observe that only *single* and *matched* nodes are activable for PolyMatch moves in $\mathcal{E}$ (by definition of the guarded rules of the algorithms). Therefore, by definition of $\mathcal{E}$, no *falsy-matched* nodes perform move in $\mathcal{E}$.

In the following, informally we build a new graph $G'$ from $G$ and a new configuration $C'_0$ from $C_0$ such that the values of the MaxMatch variables in $C'_0$ makes this configuration stable for MaxMatch. Then, we define $\mathcal{H}$ as the set of all possible executions of the composition of algorithms PolyMatch and MaxMatch starting from $C'_0$ in graph $G'$. Finally, we show that $\mathcal{E}$ is an execution that can be "projected into" an execution in $\mathcal{H}$. Since $C'_0$ is stable for MaxMatch, all executions in $\mathcal{H}$ contain $O(n^2)$ moves by Corollary 6, and so this projection result would prove that $\mathcal{E}$ contains $O(n^2)$ moves.

We now define $G'$ and $C'_0$. First assume that $F \subseteq V$ is the set of all *falsy-matched* nodes of $G$ in $C_0$. $G'$ is a copy of $G$ in which, for every node $x \in F$, we create a new node $x'$ and an edge $(x, x')$ in $G'$. Finally, we delete from the edge set of $G$ all the edges between two *single* nodes. We define $C'_0$ as follow:

- the local states of *single* and *matched* nodes are the same as in $C_0$;
- for each node $x$ in $F$:
  - the local states of $x$ is the same as in $C_0$ for the PolyMatch algorithm but is such that $m_x = x'$ where $x'$ is the new node associated to $x$;
  - the local state of $x'$ is initialized with all boolean variables at false and all other variables at $\perp$ but $m_{x'}$ that is set to $x$ for MaxMatch.

Observe that there is no *falsy-matched* node anymore in $G'$ and $C'_0$ is a stable configuration for MaxMatch. Figure 9 shows an example of a construction from $G/C_0$ to $G'/C'_0$. Also observe that any execution of $\mathcal{H}$ has two properties: (i) it does not contain any action of MaxMatch since $C'_0$ is a stable configuration for MaxMatch ; and (ii) it contains at most $O(n^2)$ moves from PolyMatch, according to Corollary 6.

Let us assume $\mathcal{E} = C_0\, a_0\, C_1\, a_1 \ldots C_k\, a_k\, C_{k+1} \ldots$ We now prove there exists an execution $\mathcal{E}' \in \mathcal{H}$ with $\mathcal{E}' = C'_0\, a'_0\, C'_1\, a'_1 \ldots C'_k\, a'_k\, C'_{k+1} \ldots$ and such that $\forall i \geq 0, a'_i = a_i$. To do that, we start by proving any move $\pi \in a_0$, with $C_0 \overset{a_0}{\longmapsto} C_1$ can be performed from $C'_0$.

First note that $\pi$ is performed by a *single* or a *matched* node, let say $u$. Observe that $u$ belongs to both graphs $G$ and $G'$. By definition of $C'_0$, $u$ has the exact same local state in $C_0$ and in $C'_0$. Between $G/C_0$ and $G'/C'_0$, the neighborhood of $u$ can be different on two points only. The first point appears when

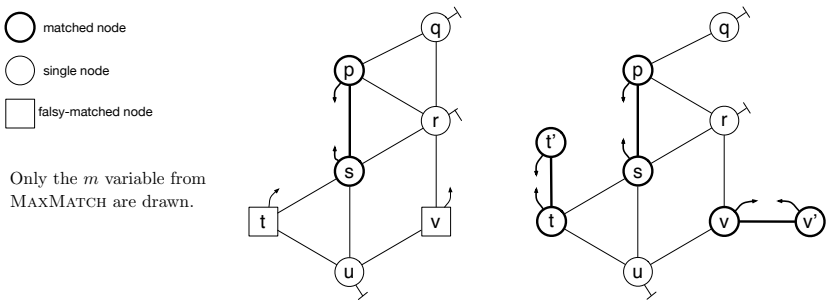



Figure 9: The transformation from $G/C_0$ (on the left) to $G'/C'_0$ (on the right)

$u$ has a *falsy-matched* neighbor on $G$, let say $x$. Indeed, $x$ becomes a *matched* node in $G'$. However, this modification does not change the local view of $u$ at distance 1, since as we previously stated, $u$ does not make any distinction between a *falsy-matched* neighbor and a *matched* one. Thus as long as $x$ do not perform any move, the local change in node $x$ do not have any impact on the actions performed by $u$. Recall that $x$ does not make any move in $\mathcal{E}$ since it is *falsy-matched*.

The second point appears when $u$ is *single* and it has a *single* neighbor $x$ in $G$. In this case, the edge $(u, x)$ disappears in $G'$ leading to a modification of the local view of $u$ at distance 1. However, this suppression has no impact on the actions performed by $u$ since $u$ consider all its *single* neighbors as nodes that are not in its neighborhood. Indeed, all references to neighbors of a single node in PolyMatch rules are combined with the *matched* predicate.

Finally, we can conclude that if $u$ can perform $\pi \in a_0$ from $C_0$, it can also perform it from $C'_0$. Thus from the transition $C_0 \overset{a_0}{\longmapsto} C_1$ in $G$, we can exhibit a configuration $C'_1$ such that the transition $C'_0 \overset{a_0}{\longmapsto} C'_1$ exists in $G'$. Observe that the relation between $C_1$ and $C'_1$ is the same as the one between $C_0$ and $C'_0$. Indeed, *falsy-matched* nodes in $C_0$ as well as nodes in $F'$ do not change their local state neither in $C_0 \mapsto C_1$ nor in $C'_0 \mapsto C'_1$. Moreover, *single* and *matched* nodes changed their local state in the same way in $C_0 \mapsto C_1$ and in $C'_0 \mapsto C'_1$, since they performed the exact same action(s). Thus, this leads to the following result : $\forall i \geq 0$, from the transition $C_i \overset{a_i}{\longmapsto} C_{i+1}$ in $G$, we can exhibit a configuration $C'_{i+1}$ such that the transition $C'_i \overset{a_0}{\longmapsto} C'_{i+1}$ exists in $G'$. Finally, from $\mathcal{E}$, we can build $\mathcal{E}'$ as defined above and so $\mathcal{E}$ contains at most $O(N^2)$ moves, with $N$ is the number of nodes in $G'$. Note that we added at most $n$ nodes in $G'$, so $\mathcal{E}$ contains at most $O(n^2)$ moves.

A last observation is about the possibility for $G'$ to be an unconnected graph, according to the suppression of all *single*-to-*single* edges in $G$. However, the result still hold with this possibility, since PolyMatch will stabilize in each connected component in $\mathrm{O}(\sum_i N_i^2)$, where $N_i$ is the size of the $i^{th}$ connected component in $G'$. And obviously, $\sum_i N_i^2 \leq N^2$ when $\sum_i N_i \leq N$, for any large value of $N$.

## 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*, pages 187–198. Springer, 2015.

[2] P. Berenbrink, T. Friedetzky, and R. A. Martin. On the stability of dynamic diffusion load balancing. *Algorithmica*, 50(3):329–350, 2008.

[3] Subhendu Chattopadhyay, Lisa Higham, and Karen Seyffarth. Dynamic and self-stabilizing distributed matching. In *Symposium on Principles of distributed computing*, pages 290–297. ACM, 2002.

[4] J. Cohen, J. Lefevre, K. Maâmra, L. Pilard, and D. Sohier. A self-stabilizing algorithm for maximal matching in anonymous networks. *Parallel Processing Letters*, 26(04):1650016, 2016.

[5] J. Cohen, K. Maâmra, G. Manoussakis, and L. Pilard. Polynomial self-stabilizing maximum matching algorithm with approximation ratio 2/3. In *OPODIS*, 2016.

[6] Ajoy K Datta, Lawrence L Larmore, and Toshimitsu Masuzawa. Maximum matching for anonymous trees with constant space per process. In *International Proceedings in Informatics*, volume 46, 2016.

[7] S. Dolev. *Self-Stabilization*. MIT Press, 2000.

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

[9] B. Ghosh and S. Muthukrishnan. Dynamic load balancing by random matchings. *J. Comput. Syst. Sci.*, 53(3):357–370, 1996.

[10] 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.

[11] Zhu Han, Yunan Gu, and Walid Saad. *Matching theory for wireless networks*. Springer, 2017.

[12] 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.

[13] 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.

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

[15] Michiko Inoue, Fukuhito Ooshita, and Sébastien Tixeuil. An efficient silent self-stabilizing 1-maximal matching algorithm under distributed daemon without global identifiers. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 195–212. Springer, 2016.

[16] Michiko Inoue, Fukuhito Ooshita, and Sébastien Tixeuil. An efficient silent self-stabilizing 1-maximal matching algorithm under distributed daemon for arbitrary networks. In *19th Int. SymposiumStabilization, Safety, and Security of Distributed Systems (SSS)*, LNCS, pages 93–108. Springer, 2017.

[17] Mehmet Hakan Karaata and Kassem Afif Saleh. Distributed self-stabilizing algorithm for finding maximum matching. *Comput Syst Sci Eng*, 15(3):175–180, 2000.

[18] F. Manne and M. Mjelde. A self-stabilizing weighted matching algorithm. In *9th Int. Symposium Stabilization, Safety, and Security of Distributed Systems (SSS)*, LNCS, pages 383–393. Springer, 2007.

[19] 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.

[20] 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.

[21] R. Preis. Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In *STACS*, LNCS, pages 259–269. Springer, 1999.

[22] 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.