HAL
open science

The Mann et al. self-stabilizing 2/3–approximation matching algorithm is sub-exponential.

Johanne Cohen, Khaled Maâmra, Laurence Pilard, George Manoussakis

To cite this version:

Johanne Cohen, Khaled Maâmra, Laurence Pilard, George Manoussakis. The Mann et al. self-stabilizing 2/3–approximation matching algorithm is sub-exponential.. [Research Report] Université de ParisSud. 2016. hal-01307886

HAL Id: hal-01307886

https://hal.science/hal-01307886v1

Submitted on 26 Apr 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

The Mann *et al.* self-stabilizing $\frac{2}{3}$−approximation matching algorithm is sub-exponential.

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

1 **LRI-CNRS, University of Paris-Sud, University ParisSaclay, France.**
johanne.cohen@lri.fr
2 **University of Versailles-St. Quentin, France.**
{ khaled.maamra, laurence.pilard}@prism.uvsq.fr

Abstract

Mann *et al.* [11] designed the first algorithm computing a maximal matching that is a $\frac{2}{3}$-approximation of the maximum matching in $2^{O(n)}$ moves. However, the complexity tightness was not proved. In this paper, we exhibit a sub-exponential execution of this matching algorithm : this algorithm can stabilize after at most $\Omega(2^{\sqrt{n}})$ moves under the central daemon.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Self-Stabilization, Maximal Matching, Approximation algorithm.

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

In graph theory, a *matching* M in a graph is a set of edges without common vertices. A matching is *maximal* if no proper superset of M is also a matching. A *maximum* matching is a maximal matching with the highest cardinality among all possible maximal matchings. In this paper, we present a self-stabilizing algorithm for finding a maximal matching. Self-stabilizing algorithms [3, 4], are distributed algorithms that recover after any transient failure without external intervention *i.e.* starting from any arbitrary initial state, the system eventually converges to a correct behavior. The environment of self-stabilizing algorithms is modeled by the notion of *daemon*. A daemon allows to capture the different behaviors of such algorithms accordingly to the execution environment. Three major types of daemons exist: the *sequential*, the *synchronous* and the *distributed* ones. The sequential daemon means that exactly one eligible process is scheduled for execution at a time. The synchronous daemon means that every eligible process is scheduled for execution at a time. The distributed daemon means that any subset of eligible processes is scheduled for execution at a time. In an orthogonal way, a daemon can be *fair* (meaning that every eligible process is eventually scheduled for execution) or *adversarial* (meaning that the daemon only guarantees global progress, *i.e.* at any time, at least one eligible process is scheduled for execution).

2 Related Works

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, 5]. Moreover, the matching problem has been recently studied in the algorithmic game theory. Indeed, the seminal problem relative to matching introduced by Knuth is the stable marriage problem [10]. This problem can be

© Johanne Cohen, Khaled Maâmra, George Manoussakis, Laurence Pilard;
licensed under Creative Commons License CC-BY
Conference title on which this volume is based on.
Editors: Billy Editor, Bill Editors; pp. 1–11
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

modeled as a game with economic interactions such as two-sided markets [1] or as a game with preference relations in a social network [7].

Several self-stabilizing algorithms have been proposed to compute maximal matching in unweighted or weighted general graphs. For an unweighted graph, Hsu and Huang [9] gave the first algorithm and proved a bound of $O(n^3)$ on the number of moves under a sequential adversarial daemon. The complexity analysis is completed by Hedetniemi et al. [6] to $O(m)$ moves. Manne et al. [11] presented a self-stabilizing algorithm for finding a 2/3-approximation of a maximum matching. The complexity of this algorithm is proved to be $O(2^n)$ moves under a distributed adversarial daemon.

3 Model

A system consists of a set of processes where two adjacent processes can communicate with each other. The communication relation is typically represented by a 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 v is denoted by $\Gamma(v)$ and is the set of all processes adjacent to v.

We consider one communication model : the *state model*. In the *state 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 a set of 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 $<$ *guard* $>\rightarrow<$ *command* $>$. The *guard* is a boolean function 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 *enabled* in a configuration C if the guard is true in C. A process is *activable* in a configuration C if at least one of its rules is enabled. An *execution* is an alternate sequence of configurations and transitions $\mathcal{E} = C_0, A_0, \ldots, C_i, A_i, \ldots$, such that $\forall i \in \mathbb{N}^*$, C_{i+1} is obtained by executing the command of at least one rule that is enabled in C_i (a process that executes such a rule makes a *move*). More precisely, A_i is the non empty set of enabled rules in C_i that has been executed to reach C_{i+1} such that each process has at most one of its rules in A_i. An *atomic operation* is such that no change can takes place during its run, we usually assume an atomic operation is instantaneous. In the case of the state model, such an operation corresponds to a rule. We use the following notation : $C_i \rightarrow C_{i+1}$. 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 in this paper are assumed to be maximal.

A *daemon* is a predicate on the executions. We consider only the most powerful one: the *distributed daemon* that allows all executions described in the previous paragraph.

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 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. Both algorithms presented here, are *silent*, meaning that once the algorithm 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.

[bookmarks=false]hyperref We consider the following matching algorithm given by Mann *et*

al. []. This algorithm, denoted $\mathcal{M}^+$, computes a maximal matching that is a $\frac{2}{3}$-approximation of the maximum matching in $2^{O(n)}$ moves. However, the complexity tightness was not proved. In this paper, we exhibit a sub-exponential execution of this matching algorithm.

4 Algorithm $\mathcal{M}^+$ given by Mann *et al.* []

The algorithm $\mathcal{M}^+$ operates on an undirected graph $G = (V, E)$, where every node $v \in V$ has a unique identifier. $\mathcal{M}^+$ assumes that there exists an underlying maximal matching algorithm, which has reached a stable configuration where a stable maximal matching M has been built. Based on M, $\mathcal{M}^+$ builds a $\frac{2}{3}-$approximation of the maximum matching. To perform that, nodes search for augmenting paths of length three.

An augmenting path is a path in the graph, starting and ending in an unmatched node, and where every other edge is either unmatched or matched; *i.e.* for each consecutive pair of edges, exactly one of them must belong to the matching Let us consider the example in Figure **??**.(a). In this figure, v and u are matched nodes and x, y are unmatched nodes. The path (y, u, v, x) 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 **??**.(b)). This transformation leads to the deletion of the augmenting path and increases by one the cardinality of the matching. The algorithm will stabilize when there are no augmenting paths of length three left. Thus the hypothesis of Karps's theorem [8] eventually holds, giving a $\frac{2}{3}-$approximation of the maximum matching.

(a) A 3-augmenting path (one matched edge)

(b) The path after being expoited. (two matched edges)

Figure 1 How to exploit a 3-augmenting path ?

The underlying stable maximal matching M is locally expressed by variables m_v for each node v. These variables are defined as follows:

$\forall v \in V : (m_v = null) \Leftrightarrow (\forall (a, b) \in M, a \neq v \wedge b \neq v)$ – In this case, v is called a *single node* and we note $v \in \sigma(V)$.

$\forall v \in V : (\exists u \in V,\ m_v = u) \Leftrightarrow ((v, u) \in M)$ – In this case, v is called a *matched node* and we note $v \in \mu(V)$.

In Algorithm $\mathcal{M}^+$, node v keeps track of four variables, the pointer p_v is used to define the final matching. The variables α_v, β_v are used to detect augmenting path and contains neighbors of v that are single. Also, s_v is a boolean variable used for the augmenting path transformation.

Thus two neighboring nodes v, u are matched in the final stable solution if and only if either $(p_v = u \wedge p_u = v)$ or if $(p_v = null \wedge p_u = null \wedge m_v = u \wedge m_u = v)$.

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

Algorithm $\mathcal{M}^+$ - Rule for nodes in $\sigma(V)$.

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

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

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

ResetMatch
  if $\textbf{AskFirst}(v, m_v) = \textbf{AskSecond}(v, m_v) = null \wedge (p_v, s_v) \neq (null, false)$
  then $(p_v, s_v) := (null, false)$

Algorithm $\mathcal{M}^+$ - Rules for nodes in $\mu(V)$.

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

AskFirst(v, u)
  if $\alpha_v \neq null \wedge \alpha_u \neq null \wedge 2 \leq Unique(\{\alpha_v, \beta_v, \alpha_u, \beta_u\}) \leq 4$
    if $\alpha_v < \alpha_u \vee (\alpha_v = \alpha_u \wedge \beta_v = null) \vee (\alpha_v = \alpha_u \wedge \beta_u \neq null \wedge v < u)$
      return α_v
  return $null$

AskSecond(v, u)
  if $AskFirst(u, v) \neq null$
    return $Lowest(\{\alpha_v, \beta_v\} \setminus \{\alpha_u\})$
  return $null$

Algorithm $\mathcal{M}^+$ - Functions

For each edge (v, u) in M, matched nodes v and u are going to:

1. **Detect** augmenting path: first, every pair of matched nodes v, u will try to find single neighbors to which they can rematch. These single neighbors have to be *available*, meaning they should not be involved in another augmenting path exploitation, *i.e.* a single node x is available if $p_x = null$. We will say that x is a *candidate* for v if x is an available single neighbor of v. Moreover v and u have 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 v and u should be at least 2. The *BestRematch* predicate is used to compute candidates in variables α and β, and the condition below (in *AskFirst* predicate) is used to ensure the number of candidates is sufficiently high. ($Unique(A)$ returns the number of unique elements in the multi-set A).

$$\alpha_u \neq null \wedge \alpha_v \neq null \wedge 2 \leq Unique(\{\alpha_u, \beta_u, \alpha_v, \beta_v\}) \leq 4$$

2. **Try to exploit** this augmenting path :
   - **a.** The *AskFirst* node starts: exactly one of v and u will attempt to match with one of its candidates.
   - **b.** The *AskSecond* node continues: only when the first node succeeds will the second node also attempt to match with one of its candidates.
     - **i.** If this also succeeds, the rematching is considered *complete*.
     - **ii.** Otherwise the rematch built by the *AskFirst* node is deleted and candidates α and β are computed again, allowing then the detection of new augmenting paths.

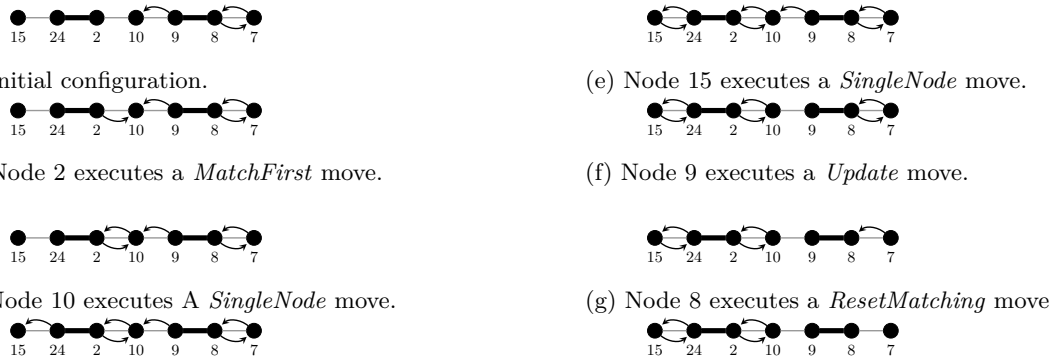
Now, we give a possible execution of Algorithm $\mathcal{M}^+$ under a distributed adversarial daemon. Fig. 1.(a) shows the initial state of the execution. The topology is a path of seven vertices and the identifiers of the nodes are indicated below. The underlying maximal matching represented by bold edges contains two edges $(24, 2)$ and $(9, 8)$. Then nodes 24, 2, 9 and 8 are *matched* nodes (in $\mu(V)$) and nodes 15, 10 and 7 are *single* nodes (in $\sigma(V)$). We illustrate the use of the *p-values* by an arrow and the absence of the arrow means that the *p-value* of the node equals to *null*.

At the beginning, there are two augmenting paths. Nodes 9 and 8 have already started to exploit their augmenting path. We are going to exhibit an execution where this augmenting path will be reset while the other one will be fully exploited.

In the initial configuration, we assume that all *α-values* and *β-values* are defined as follows: $(\alpha_8, \beta_8) = (7, null)$, $(\alpha_9, \beta_9) = (10, null)$ and $(\alpha_{24}, \beta_{24}) = (\alpha_2, \beta_2) = (null, null)$. We also assume all *s-values* are well defined: $s_8 = true$ and $s_9 = s_2 = s_{24} = false$. At this step, node 9 waits for an answer of node 10. Nodes 2 and 24 have two unique candidates for a rematching.

At the beginning of the execution, all *α-values* and *β-values* for all nodes are well defined except for nodes 2 and 24 because $BestRematch(2) = (10, null)$, $BestRematch(24) = (15, null)$. Nodes 2 and 24 execute a *Update* move. After these moves, $(\alpha_{24}, \beta_{24}) = (15, null)$ and $(\alpha_2, \beta_2) = (10, null)$.

Since $2 \leq Unique(\{\alpha_2, \beta_2, \alpha_{24}, \beta_{24}\}) \leq 4$, nodes 2 and 24 detect a 3-augmenting path and start to exploit this augmenting path. Since $AskFirst(2, 24) = 10$ (which implies $AskFirst(24, 2) = null$), node 2 may execute a *MatchFirst* move. Let us assume it does and then it points to node 10, as seen in Figure 1.(b). Since both nodes 9 and 2 are pointing to node 10, node 10 can choose the node to match with from these two nodes. Note that at this point, node 10 is the only enabled node. Figure 1.(c) shows the configuration obtained after node 10 makes this choice executing a *SingleNode* move: since $Lowest\{u \in \Gamma(10) \mid p_u = 10\} = 2$,



(a) Initial configuration.

(b) Node 2 executes a *MatchFirst* move.

(c) Node 10 executes A *SingleNode* move.

(d) Node 24 executes a *MatchSecond* move.

(e) Node 15 executes a *SingleNode* move.

(f) Node 9 executes a *Update* move.

(g) Node 8 executes a *ResetMatching* move

(h) Node 7 executes a *SingleNode* move.

Figure 2 An execution of Algorithm $\mathcal{M}^+$

node 10 points to node 2. Now, node 24 is eligible to execute a *MatchSecond* move and, since $BestRematch(9)$ has changed, node 9 is eligible to execute an *Update* move.

Let us assume node 24 is activated (see Figure 1.(d) for configuration after this move). It then points to node 15 thus, node 15 can accept the proposition executing a *SingleNode* move. So, it does it and it sets $p_{15} = 24$. Figure 1.(e) shows after this moves.

Since $p_{10} \neq 9$ and $(\alpha_9, \beta_9) \neq BestRematch(9)$, node 9 can execute an *Update* move. Figure 1.(f) shows the configuration obtained after this move: $(\alpha_9, \beta_9) = (null, null)$ and $(p_9, s_9) = (null, false)$. This will cause $AskFirst(8, 9) = AskSecond(8, 9) = null$. Then node 8 executes a *ResetMatch* move (see configuration after this move Figure 1.(g)). This will cause node 7 to execute a *SingleNode* move and sets $p_7 = null$ as seen in Figure 1.(h). The system then has reached a stable configuration. Thus, the size of the matching is increasing by one and only one augmenting path has been fully exploited.

4.1 Description of Algorithm $\mathcal{M}^+$

In order to build an instance, we will focus on edges in the maximal matching M. We will define a state for these edges according to the p-values.

▶ **Definition 1** (Edges in state *On* or in state *Off*). Let $e = (u, v)$ be an edge in the maximal matching M. Let x (resp. y) be the single node adjacent to u (resp. v). Edge e is said to be *in state Off* if $p_u = null$, $p_v = null$, $p_x = null$ and $p_y = null$. Moreover edge e is said to be *in state On* if $p_u = y, p_x = v$, $p_v = x$ and $p_y = null$. Edge e is said to be *in state Almost On* if $p_u = y, p_x = v$, $p_v = x$ and $p_y \notin \{null, u\}$.

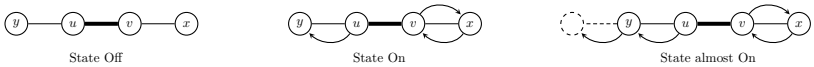


Figure 3 Edges in state Off and On : the arrows drawn represent the local variables $p.$ of nodes.

An example of Definition 1 can be seen in Figure **??**. Moreover, in Figure 1.(a), edge $(24, 2)$ is in state *Off* while edge $(9, 8)$ is in state *On*.

The states of edges represent the detection process step of the 3-augmenting path. Now, we will exhibit an execution to switch edge (u, v) from state *Off* to state *On*.

▶ **Lemma 2.** *Let $e = (u, v)$ be an edge in the maximal matching M and in state Off. Let y (resp. x) be the single node adjacent to u (resp. v) with $Ident(x) < Ident(y)$.*

If $y = Lowest\{BestRematch(u)\}$, $x = Lowest\{BestRematch(v)\}$ and $v \leq Lowest\{w \in \Gamma(x) | p_x = w\}$, then there exists a finite execution to switch edge (u, v) from state Off *to state* On*. Moreover the only nodes executing a move in this execution are $\{x, u, v\}$.*

Proof: We describe a finite execution to switch edge (u, v) from state Off to state On. Nodes u and v belong to a 3-augmenting path since $p_x = p_y = null$. If $\alpha_u \neq y$, then node u executes a *Update* move : $(\alpha_u, \beta_u) = (y, null)$ because $p_y = null$. If $\alpha_v \neq x$, then node v executes a *Update* move : $(\alpha_v, \beta_v) = (x, null)$ because $p_x = null$ and $v = Lowest\{w \in \Gamma(x) | p_x = w\}$.

Thus, the variables α_u and α_v are well defined : $\alpha_u = y$ and $\alpha_v = x$. $Ident(x) < Ident(y)$ implies $AskFirst(v, u) = x$ and $AskFirst(u, v) = null$ because $2 \leq Unique(\{\alpha_u, \beta_u, \alpha_v, \beta_v\}) \leq 4$. Thus node v executes a *MatchFirst* move: $p_v = x$. Since $v = Lowest\{w \in \Gamma(x) | p_w = x\}$ by the hypothesis of this lemma, node x 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 y (because $y = Lowest\{BestRematch(u)\}$). □

Note that Figures 1.(a)- 1.(d) represent an execution to switch edge $(24, 2)$ from state *Off* to state *On*: nodes 15 and 10 are respectively nodes y and x for the execution of Lemma 3. Now, Now, we will exhibit an execution to switch edge (u, v) from state *Almost On* to state *Off*.

▶ **Lemma 3.** *Let $e = (u, v)$ be an edge in the maximal matching M and in state* Almost On*. Let y (resp. x) be the single node adjacent to u (resp. v) with $Ident(x) < Ident(y)$. There exists a finite execution to switch edge (u, v) from state* Almost On *to state* Off*. Moreover the only nodes executing a move in this execution are $\{x, y, u, v\}$.*

Proof: A finite execution to switch edge (u, v) from state *Almost On* to state *Off* is described. Since edge (u, v) is in state *Almost On*, $p_y \notin \{u, null\}$ and so $(\alpha_u, \beta_u) \neq BestRematch(u)$. Node u executes a *Update* move. After this move, $(p_u, s_u) = (null, false)$. The fact that $\alpha_u = null$ will cause $AskFirst(v, u) = AskSecond(v, u) = null$. Then node v executes a *ResetMatch* move: $p_v = null$. Then node x is actived by executing a *SingleNode* move and it sets $p_x = null$. Finally, node v can execute a *Update* move, and thus $(\alpha_v, \beta_v) = (null, null)$. □

Note that in Figure 1.(e), edge $(9, 8)$ is in state *Almost On*. Figures 1.(f)-1.(h) represent the execution of Lemma 3 in order that edge $(9, 8)$ will be in state *Off*.

4.2 Complexity of Algorithm $\mathcal{M}^+$

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 in N sub-graphs, each of them representing a bit. The whole graph then represents an integer, coding from theses N bits. G_N has 2 kind of nodes: the nodes represented by circles (•-nodes) and those represented by squares (■-nodes). The •-nodes are used to store bits value and hence an integer. The ■-nodes are used to implement the "+1" operation as we count from 0 to $2^N - 1$.

4.2.1 Example:

As an illustration, graph G_4 is shown in Figure **??**. In this example, the bold edges are those that belong to the maximal matching M computed by algorithm $\mathcal{M}$ and arrows represent

the local variable p of the 2/3-approximation algorithm. A node having no outgoing arrow has its p variable equals to *null*.

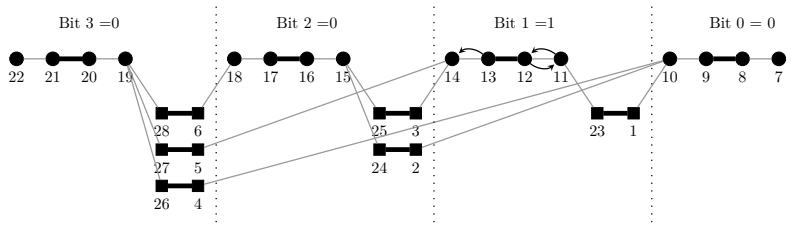


Figure 4 Graph G_4 encoding 0010

As we said, the •-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 kth node of the bit i*". For instance, node 10 is the fourth node of the bit 1, thus 10 is called $b(1,4)$. In the following, we will refer to these four nodes as the ith *bit-block*.

A binary value can be associated to each bit-block according to the p-values of each nodes in the bit-block. We will formally define this association later, but we can already say that in this example, according to the p-value of all the nodes in the 4 bit-blocks, G_4 encodes the binary integer 0010.

4.2.2 Definition of the graph $\mathbf{G_N}$:

In the following, we formally describe the graph $G_N = (V_N, E_N)$.

1. $V_N = V_N^{\bullet} \cup V_N^{\blacksquare}$ where

$$V_N^{\bullet} = \bigcup_{0 \le i < N} \{b(i,k) | k = 1,2,3,4\}$$

$$V_N^{\blacksquare} = \bigcup_{0 \le j < i < N} \{r_1(i,j), r_2(i,j)\}$$

2. $E_N = E_N^{\bullet} \cup E_N^{\blacksquare}$ where

$$E_N^{\bullet} = \bigcup_{0 \le i < N} \{(b(i,k), b(i,k+1)) | k = 1,2,3\}$$

$$E_N^{\blacksquare} = \bigcup_{0 \le j < i < N} \{(b(i,1), r_1(i,j)) \ , \ (r_1(i,j), r_2(i,j)) \ , \ (r_2(i,j), b(j,4))\}$$

Figure 2 gives a partial view of the graph G_N corresponding to the ith bit-block.

Our execution is based on the maximal matching M computed by the algorithm $\mathcal{M}$:

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

This maximal matching M is encoded with the m-variable. Then we have:

$$m_{b(i,2)} = b(i,3), m_{b(i,3)} = b(i,2), m_{r_1(i,j)} = r_2(i,j) \text{ and } m_{r_2(i,j)} = r_1(i,j)$$

This matching is an $\frac{1}{2}$-approximation of the maximum matching and the algorithm $\mathcal{M}^+$ updates this approximation building a $\frac{2}{3}$-approximation of the maximum matching based on M. This $\frac{2}{3}$-approximation is encoded with the p-variable in $\mathcal{M}^+$. We also use the variable p to encode a bit associated to a bit-block. The two following definitions give this association:

▶ **Definition 4** (Bit-block encoding). In graph G_N, let $\{b(i,1), b(i,2), b(i,3), b(i,4)\}$ be the ith bit-block, for some $0 \le i < N$. This bit-block encodes the value 1 (*resp.* 0) if the edge $(b(i,2), b(i,3))$ is in state *On* (*resp.* Off).

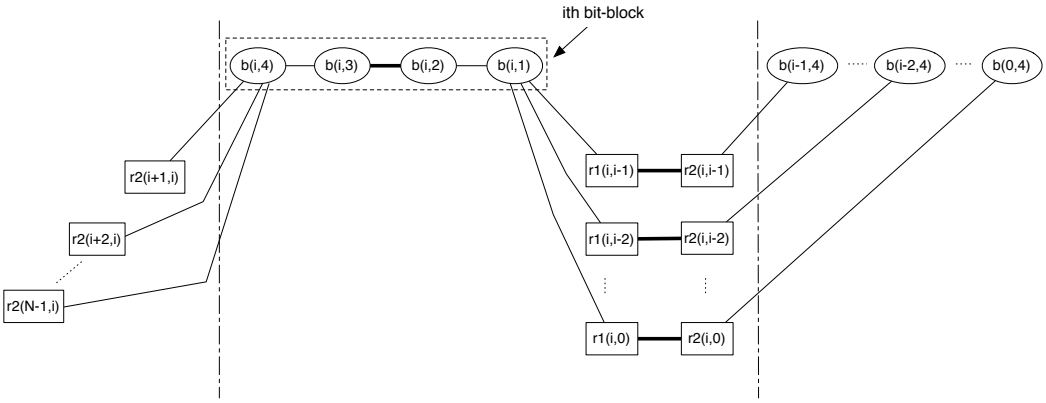


Figure 5 A partial view of graph G_N

Note that the value is not always defined. We can associate an integer ω to such a configuration of the graph.

▶ **Definition 5** (ω-configuration)**.** Let ω be represented the integer such that $\omega < 2^N$, a configuration is said to be an *ω-configuration* if for any integer $i \leq N$, the ith bit of ω is the value encoded by the ith block of nodes.

Figure **??** shows a 3-configuration.

4.2.2.1 Identifiers:

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 $\mathcal{M}^+$. 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. We assume each node has a unique identifier. These identifiers must satisfy the three following properties:

▶ Property 1 (Identifiers order). 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. $ident(b(i,3)) > ident(r_2(j,i))$

Note that in graph G_N, it exists an *ident* function that satisfies Property 1. For instance, 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:

$$\forall i, 0 \leq i < N, \forall k \in \{1,2,3,4\} : ident(b(i,k)) = s + i + k$$

- Nodes of kind r_1 are named from $s+c+1$ to $s+c+s$

Figure **??** shows graph G_4 with such a naming.

4.2.2.2 Counting from 0 to $2^N - 1$:

We will build an execution containing all ω-configurations with $1 \leq \omega < 2^N$ To to this, we will build an execution from ω-configuration to $(\omega + 1)$-configuration using "+1" operation. This allows for the counting from $\mathbf{0}$ to $\mathbf{2^N - 1}$. As we said before, the nodes in $V_N^{\blacksquare}$ are used to implement the "+1" operation. To do that, we need to be able to switch bit from 0 to 1 and from 1 to 0, in a clever way. To switch from 0 to 1 is easier than to switch from 1 to 0. The nodes in $V_N^{\blacksquare}$ are used to implement the switch 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 $x + 1 = x'100\ldots0$). Let us assume that 0 is the ith bit of x. The ith bit-block has to switch from 0 to 1 during the '+1' operation. Afterwards, each jth bit-block, with $0 \leq j < i$, has to switch from 1 to 0. To perform this switch, we use vertices in $V_N^{\blacksquare}$.

We will now describe a piece of the execution, starting on the configuration represented on Figure **??**. The graph drawn in this figure encodes integer (0010). We illustrate the use of the *p-values* by an arrow and the absence of the arrow means that the *p-value* of the node equals to *null*. First, we will focus on vertices in the 0th bit-block. Edge $(b(0,2), b(0,3))$ belongs to the underlying maximal matching represented by bold edges and is in state *Off*. Lemma 2, describes an execution from the 0010-configuration represented on Figure **??** to the 0011-configuration represented on Figure **??**. Moreover, Figures **??**, **??**, **??**, **??** and **??** illustrate the transformation from 0011-configuration to 0111-configuration in graph G_4.

▶ **Theorem 6.** *Let ω be an integer such that $\omega < 2^N - 1$. There exists a finite execution to transform an ω-configuration into an $(\omega + 1)$-configuration.*

Proof: Let i be the integer such that the $i - 1$ first bits of ω equal to 1 and the value of its ith bit to 0. This implies that the ith bit of $\omega + 1$ bits is the first bit equal to 1.

We distinguish two cases : $i = 0$ and $i > 0$.

In the case where $i = 0$, 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, $(b(0,2), b(0,3))$ is in state *On* in $(\omega + 1)$-configuration. By Property 1, we have $Ident(b(0,1)) < Ident(b(0,4))$ and by definition of edge in state *Off*, $p_{b(0,1)} = p_{b(0,4)} = null$. Note that $b(0,2) \leq Lowest\{w \in \Gamma(b(0,1)) | p_{b(0,1)} = w\}$. Since nodes $b(0,3)$ and $b(0,2)$ only have one Single node as neighbour, the hypotheses of Lemma 2 are satisfied From Lemma 2 , there exists an execution to switch edge $(b(0,2), b(0,3))$ from state *Off* to state *On*. At the end, the least significant bit of the integer correspond to this current configuration is set to 1. So we obtain a $(\omega + 1)$-configuration.

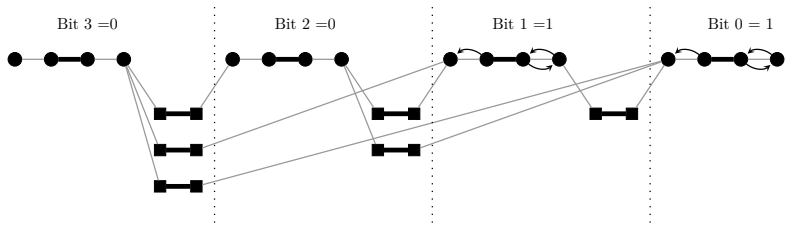


Figure 6 After turning on the 0th bit-block, G_4 encodes 0011.

In the case where $i > 0$, for every integer j from 0 to $i - 1$, edge $(b(j,2), b(j,3))$ is in state *On* and edge $(b(i,2), b(i,3))$ is in state *Off*.

More precisely, we can execute the following sequence of moves :

1. For each integer j, $1 \leq j \leq i-1$, edge $(r_1(i,j), r_2(i,j))$ is in state *Off*. Note that node $r_1(i,j)$ (resp. $r_2(i,j)$) is adjacent to one Single node $b(i,1)$ (resp. $b(j,4)$). Since $b(j,4) \leq Lowest\{w \in \Gamma(r_2(i,j)) | p_{r_2(i,j)} = w\}$, the hypotheses of Lemma 2 are satisfied. Thus from Lemma 2, we can exhibit an execution to switch edges $(r_1(i,j), r_2(i,j))$ from state *Off* to state *On*. The configuration shown in Figure **??** that corresponds to this step.

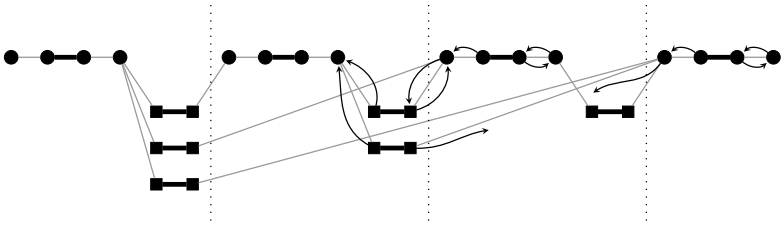


Figure 7 After activating the ■-nodes of the 3rd bit-block, G_4 does not encode any integer.

2. Now, for each integer j, $1 \leq j \leq i-1$, edge $(b(j,2), b(j,3))$ is in state *Almost on*. From Lemma 3, (since $Ident(b(j,1)) < Ident(b(j,4))$) an execution to switch edge $(b(j,2), b(j,3))$ from state *Almost on* to state *Off* is performed. The configuration shown in Figure **??** that corresponds to this step.

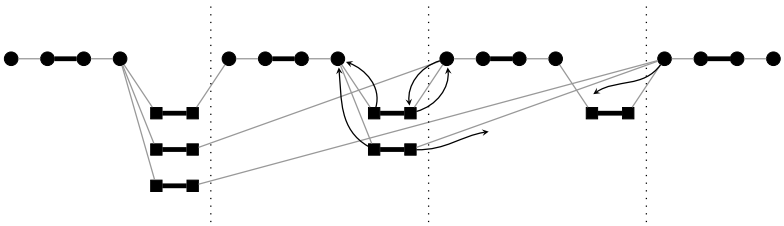


Figure 8 Starting to turn off the 0th and 1st bit-blocks.

3. Edge $(b(i,2), b(i,3))$ is still in state *off*. Using the same argument of step (1), from Lemma 2, we can exhibit an execution to switch edges $(b(i,2), b(i,3))$ from state *Off* to state *On*.

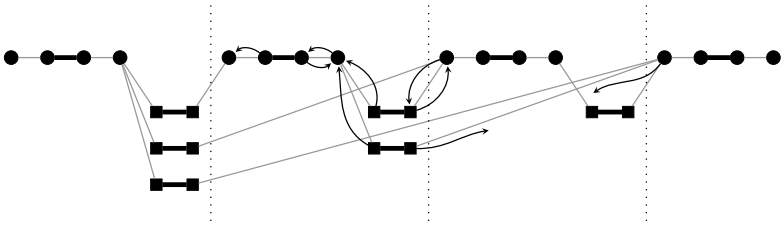


Figure 9 Starting to turn on the 3rd bit-block.

4. Now, for each integer j, $1 \leq j \leq i-1$, edge $(r_1(i,j), r_2(i,j))$ is now in state *Almost on*. From Lemma 3, an execution to switch edge $(r_1(i,j), r_2(i,j))$ from state *Almost on* to state *Off*.

At the end of this execution, the configuration still verifies the two conditions, and the $i-1$ first bits of ω are set to 0 and the ith to 1. So we obtain a $(\omega+1)$-configuration. □

From now, we can construct an instance from which an execution having $\Omega(2^{\sqrt{n}})$ moves can be built.

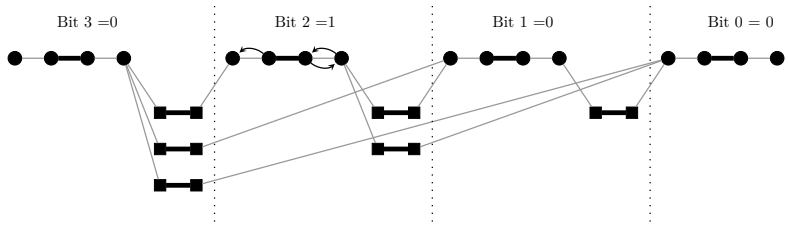


Figure 10 Ending to turn off the 0th and 1st bit-blocks and to turn on the 3rd bit-block. G_4 encodes 0100.

▶ **Corollary 7.** *Algorithm $\mathcal{M}^+$ can stabilize after at most $\Omega(2^{\sqrt{n}})$ moves under the central daemon.*

Proof: To prove the corollary, we can exhibit an execution of $\Omega(2^{\sqrt{n}})$ moves. Let N be an integer. The initial configuration is a 0-configuration in graph G_N.

We can build an execution that contains all the ω-configurations for every value ω, $1 \leq \omega \leq 2^N$. By applying Theorem 6, this execution can be split into 2^N parts corresponding to the execution from ω-configuration to $(\omega + 1)$-configuration, for $1 \leq \omega \leq 2^N$. Thus, this execution has $O(2^N)$ configurations. Since graph G_N has $O(N^2)$ vertices, this execution has $O(2^{\sqrt{n}})$ configurations and the corollary holds.

□

5 Conclusion

The algorithm designed by Mann *et al.* [11] computes a maximal matching that is a $\frac{2}{3}$-approximation of the maximum matching in $2^{O(n)}$ moves. In this work, we exhibit a sub-exponential execution of this matching algorithm. An open question is to adapt this algorithm in order to stabilize in polynomial moves.

References

1 Heiner Ackermann, Paul W. Goldberg, Vahab S. Mirrokni, Heiko Röglin, and Berthold Vöcking. Uncoordinated two-sided matching markets. *SIAM J. Comput.*, 40(1):92–106, 2011.
2 Petra Berenbrink, Tom Friedetzky, and Russell A. Martin. On the stability of dynamic diffusion load balancing. *Algorithmica*, 50(3):329–350, 2008.
3 Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.
4 Shlomi Dolev. *Self-Stabilization.* MIT Press, 2000.
5 Bhaskar Ghosh and S. Muthukrishnan. Dynamic load balancing by random matchings. *J. Comput. Syst. Sci.*, 53(3):357–370, 1996.
6 Stephen T. Hedetniemi, David Pokrass Jacobs, and Pradip K. Srimani. Maximal matching stabilizes in time o(m). *Inf. Process. Lett.*, 80(5):221–223, 2001.
7 Martin Hoefer. Local matching dynamics in social networks. pages 20–35, 2013.
8 John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.
9 Su-Chu Hsu and Shing-Tsaan Huang. A self-stabilizing algorithm for maximal matching. *Inf. Process. Lett.*, 43(2):77–81, 1992.
10 Donald Knuth. *Marriages stables et leurs relations avec d'autres problèmes combinatoires.* Les Presses de l'Université de Montréal, 1976.

11 Fredrik Manne, Morten Mjelde, Laurence Pilard, and Sébastien Tixeuil. A self-stabilizing 2/3-approximation algorithm for the maximum matching problem. *Theor. Comput. Sci.*, 412(40):5515–5526, 2011.