Recommended Paper

A Self-stabilizing 1-maximal Independent Set Algorithm

Hideyuki Tanaka[1,a) Yuichi Sudo[1,b) Hirotsugu Kakugawa[2,c) Toshimitsu Masuzawa[1,d)
Ajoy K. Datta[3]

Received: May 8, 2020, Accepted: December 1, 2020

Abstract: We consider the 1-maximal independent set (1-MIS) problem: given a graph $G = (V, E)$, our goal is to find a 1-maximal independent set (1-MIS) of a given network G, that is, a maximal independent set (MIS) $S \subset V$ of G such that $S \cup \{v, w\} \setminus \{u\}$ is not an independent set for any nodes $u \in S$, and $v, w \notin S$ $(v \neq w)$. We give a silent, self-stabilizing, and asynchronous distributed algorithm to construct a 1-MIS on a network of any topology. We assume the processes have unique identifiers and the scheduler is weakly-fair and distributed. The time complexity, i.e., the number of rounds to reach a legitimate configuration in the worst case of the proposed algorithm is $O(nD)$, where n is the number of processes in the network and D is the diameter of the network. We use a composition technique called *loop composition* [Datta et al., 2017] to iterate the same procedure consistently, which results in a small space complexity, $O(\log n)$ bits per process.

Keywords: distributed system, self-stabilizing algorithm, loop composition, 1-maximal independent set

1. Introduction

Nowadays, distributed systems generally consist of *numerous* computers (or processes) where the processes collaboratively solve a problem by communicating with each other. Because of the huge scale, distributed systems are prone to have faults in their components. Therefore, it is important to design an algorithm, which works correctly even if some of the processes have failed, the topology of a network changes, and/or the stored data of some processes are corrupted arbitrarily.

Self-stabilization [5] is a promising technique to achieve high fault tolerance. An execution of a self-stabilizing algorithm is guaranteed to reach a *safe* configuration eventually (*Convergence property*), which satisfies the specification of a given problem and keeps the legitimacy thereafter (*Closure property*). These two properties of self-stabilization make distributed systems tolerate any number and any kind of transient fault in the sense that the system can recover and attain the desired behavior from any illegitimate configuration that those faults may cause.

In this paper, we consider the 1-maximal independent set problem which is a variant of the maximal independent set problem. Given a graph (or a network) $G = (V, E)$, a set $S \subseteq V$ of nodes (or processes) is independent if any two nodes in S are not neighbors. Finding a large independent set of a given graph is important for many applications in distributed systems, for example, clustering in wireless networks (See Ref. [1] in detail). However, finding the *maximum* independent set is NP-hard [12]. Therefore, many studies in the literature give solutions to find a *maximal* independent set (MIS), i.e., an independent set such that no proper superset of it is independent. Unfortunately, the maximality of an independent set does not always guarantee a large cardinality of the set. For example, every star graph has an MIS consisting of only one node. Therefore, we consider a stronger maximality called *1-maximality*, which Bollobás et al. [2] introduced [*1]. An MIS $S \subseteq V$ is 1-maximal if $S \cup \{v, w\} \setminus \{u\}$ is not independent for any $u \in S$, and $v, w \notin S$ $(v \neq w)$. The 1-maximality offers a better solution in many cases. For example, the star graph of n nodes has exactly one 1-maximal independent set (1-MIS), whose size is $n - 1$.

1.1 Related Work

The maximal independent set problem is one of the most fundamental problems in graph theory and the field of distributed computing, thus it has been studied in much literature.

Table 1 summarizes recent results on self-stabilizing MIS algorithms, where n and D denote the number of processes and the diameter of the network, respectively.

In 1995, Shukla et al. [10] gave a self-stabilizing MIS algorithm for any anonymous network. Their algorithm assumes the central scheduler, i.e., exactly one process executes an atomic ac-

[1] Graduate School of Information Science and Technology, Osaka University, Suita, Osaka 565–0871, Japan
[2] Faculty of Advanced Science and Technology and Ryukoku Center for Mathematical Sciences and Networks, Ryukoku University, Otsu, Shiga 520–2194, Japan
[3] University of Nevada, Las Vegas, NV 89154, USA (**Ajoy Kumar Datta passed away on May 26, 2019. Rest in Peace, Ajoy**)
a) tanaka.hideyuki@ist.osaka-u.ac.jp
b) y-sudou@ist.osaka-u.ac.jp
c) kakugawa@rins.ryukoku.ac.jp
d) masuzawa@ist.osaka-u.ac.jp

A preliminary extended abstract [11] of this paper appeared in the proceedings of SSS 2019.
The preliminary version of this paper was published at the IPSJ Kansai-Branch Convention in September 2019. The paper was recommended to be submitted to Journal of Information Processing (JIP) by the program chair of the convention.

[*1] They actually introduce more general maximality, k-maximality for any $k \geq 1$

Table 1 Self-stabilizing maximal and 1-maximal independent set algorithms. n denotes the number of processes, D denotes the diameter of the network.

	problem	topology	ID	scheduler	convergence time		space
					steps	rounds	
[10]	MIS	any	unavailable	central	$O(n)$	$O(n)$	$O(1)$ bits
[7]	MIS	any	available	distributed	$O(n^2)$	$O(n)$	$O(1)$ bits
[12]	MIS	any	available	distributed	$O(n)$	$O(n)$	$O(1)$ bits
[9]	1-MIS	tree	unavailable	central	$O(n^2)$	$O(n)$	$O(1)$ bits
[8]	1-MIS	any	available	central	-	$O(n^2)$	$O(n \log n)$ bits
Proposed	1-MIS	any	available	distributed	-	$O(nD)$	$O(\log n)$ bits

tion at each step. Its worst-case convergence from any configuration to a configuration where an MIS is constructed requires $O(n)$ steps and it uses only $O(1)$ bits per process. Ikeda et al. [7] gave a different self-stabilizing MIS algorithm. Their algorithm assumes the existence of the process-identifiers, however, it works correctly under the distributed scheduler, which can activate any number of enabled processes simultaneously at each step. Its space complexity is still $O(1)$, but the convergence time increases to $O(n^2)$ steps. Turau [12] gave a self-stabilizing MIS algorithm with an improved convergence time of $O(n)$ in the same settings.

Shi et al. [9] gave the first self-stabilizing 1-MIS algorithm. It assumes that the network topology is a tree and assumes the central scheduler. Its convergence time is $O(n^2)$ steps and the space complexity is $O(1)$ bits per process. Namba [8] gave a self-stabilizing 1-MIS algorithm for any arbitrary graph. It assumes the central scheduler. No analysis was presented for convergence time in terms of the number of steps, but its convergence time is $O(n^2)$ (asynchronous) rounds. To construct a 1-MIS, his algorithm runs n sub-algorithms in parallel, thus it uses $O(n \log n)$ bits of memory space per process.

To the best of our knowledge, there has been no literature that studies a non-self-stabilizing algorithm for the 1-maximal independent set problem. It is worthwhile to mention that deterministic construction of an MIS (and thus a 1-MIS) is impossible in an anonymous network of an arbitrary topology with the distributed scheduler, due to the impossibility of symmetry breaking.

1.2 Our Contributions

We give a silent self-stabilizing 1-MIS algorithm under the distributed scheduler for any arbitrary network. We assume the existence of process-identifiers. Its convergence time is $O(nD)$ rounds, where D is the diameter of the network, while the space complexity is $O(\log n)$ bits per process. We use a composition technique called *loop composition*, which Datta et al. [3] introduced recently. This technique enables the processes to execute the same subalgorithm repeatedly in a consistent way until a 1-MIS is constructed, which results in a smaller space complexity, $O(\log n)$ bits per process. To the best of our knowledge, the loop composition technique is utilized only for the k-grouping problem [3] although it seems applicable to many problems. Thus, our result shows the applicability by providing the second success case of the loop composition.

2. Preliminaries

An undirected network $G = (V, E)$ consisting of process set V and link set E is given. We denote the number of processes and the diameter of G by n and D, respectively. We assume $n \geq 2$. We assume that the network G is connected without loss of generality; if G is not connected, it suffices to construct a 1-MIS for each component of G. Each process v has a unique identifier $v.id$ chosen from a set ID of non-negative integers where $|ID| = O(poly(n))$. Let N_v denote the neighbors of a process v, i.e., $N_v = \{u \in V \mid \{u, v\} \in E\}$. We call the processes in N_v *v-neighbors*. By an abuse of notation, we will identify each process with its identifier, and vice versa, whenever convenient. We call a member of ID a *false identifier* if it is not the identifier of any process in V.

We use the locally shared memory model [5]. A process is modeled by a finite state machine. The state of a process is defined by the values of its variables. A process can read the variables of its own and its neighbors simultaneously, but can update only its own variables. A distributed algorithm defines the behavior of each process v by a finite set of (guarded) actions of the following form: $<$ *label* $><$ *guard* $>\longrightarrow<$ *statement* $>$. The *label* of each action is a number used for reference. The *guard* is a predicate on the variables and identifiers of v and it's neighbors. The *statement* updates the state (or variables) of v. An action can be executed only if it is *enabled*, i.e., its guard evaluates to true, and a process is *enabled* if at least one of its actions is enabled. The evaluation of the guard and the execution of the corresponding statement are presumed to take place in one atomic step. For simplicity, we use notation "$v.x \longleftarrow \chi(v)$" to represent an action "$v.x \neq \chi(v) \longrightarrow v.x \leftarrow \chi(v)$" for any variable x and any function $\chi(v)$. Thus, the action "$v.x \longleftarrow \chi(v)$" is enabled if and only if $v.x \neq \chi(v)$. We also use a symbol $\perp$ to represent a "null value" and define $\min \emptyset = \perp$ and $\min\{a, \perp\} = a$.

A *configuration* of the network is an n-dimensional vector consisting of the process states, one for each process in the network. We denote by $\gamma(v).x$ the value of variable x of process v in configuration γ. Each transition from a configuration to another, called a *step* of the algorithm, is driven by a *scheduler*. We assume the *distributed scheduler* in this paper. At each step, the distributed scheduler selects one or more enabled processes to execute their action. If a selected process has two or more enabled actions, it executes the action with the smallest label number. We write $\gamma \mapsto_{\mathcal{A}} \gamma'$ if configuration γ can change to γ' by one step of algorithm $\mathcal{A}$. We define an *execution* of algorithm $\mathcal{A}$ to be a sequence of configurations $\gamma_0, \gamma_1, \cdots$ such that $\gamma_i \mapsto_{\mathcal{A}} \gamma_{i+1}$ for all $i \geq 0$. We assume the scheduler to be *weakly-fair*, meaning that a continuously enabled process must be selected eventually.

A self-stabilization algorithm ensures that any execution eventually recovers a correct configuration even if it is started from any configuration, i.e., each process may start from any state. An execution is *maximal* if it is infinite, or it terminates at a *final* configuration, i.e., a configuration where no process is enabled. Let $\mathcal{L}$ be a predicate on configurations. We say that a configuration γ of $\mathcal{A}$ is *safe* for $\mathcal{L}$ if every execution $\gamma_0, \gamma_1, \dots$ of $\mathcal{A}$ starting from γ (i.e., $\gamma_0 = \gamma$) always satisfies $\mathcal{L}$, that is, $\mathcal{L}(\gamma_i)$ holds for all $i \geq 0$. Algorithm $\mathcal{A}$ is said to be *self-stabilizing* for $\mathcal{L}$ if there exists a set C of configurations of $\mathcal{A}$ such that every configuration in C is safe for $\mathcal{L}$ and every maximal execution $\gamma_0, \gamma_1, \dots$ of $\mathcal{A}$ reaches a configuration in C, i.e., $\gamma_i \in C$ holds for some $i \geq 0$. We also say that $\mathcal{A}$ is *silent* if every execution of $\mathcal{A}$ is finite. Thus, a silent algorithm $\mathcal{A}$ is self-stabilizing for predicate $\mathcal{L}$ if and only if every final configuration satisfies $\mathcal{L}$.

We sometimes regard a predicate on configurations as the set of configurations. For example, we write $\gamma \in \mathcal{L}_1 \cap \mathcal{L}_2$ when a configuration γ satisfies both predicates $\mathcal{L}_1$ and $\mathcal{L}_2$.

We measure time complexity of an execution in *rounds* [6]. We say that process v is *neutralized* at step $\gamma_i \mapsto \gamma_{i+1}$ if v is enabled at γ_i and not at γ_{i+1}, but v executes no action at the step. We define the first round of an execution $\varrho = \gamma_0, \gamma_1, \dots$ to be the minimum prefix $\gamma_0 \dots \gamma_s$ during which every process enabled at γ_0 executes an action or is neutralized. The second round of ϱ to be the first round of the execution $\gamma_s, \gamma_{s+1}, \dots$ and so forth. We evaluate the number of rounds of ϱ, denoted by $R(\varrho)$, as the execution *time* of ϱ.

2.1 Problem Specification

We specify the 1-maximal independent set problem. A set $S \subseteq V$ of processes is called an *independent set* (or just IS) of G if no two processes in S are neighbors in G, that is, $\forall u, v \in S : \{u, v\} \notin E$. An independent set of G is called a *maximal independent set* (or just MIS) of G if it is not a proper subset of any other independent set of G. A maximal independent set S of G is called a *1-maximal independent set* (or just 1-MIS) of G if we cannot increase the cardinality of S without violating the *independent* property by removing one process and adding two or more processes, that is, for any process $u \in S$ and any distinct processes $v, w \notin S$, set $S \cup \{v, w\} \setminus \{u\}$ is not an independent set. We assume that each process v has a variable $v.\mathtt{mis} \in \{\mathbf{true}, \mathbf{false}\}$. We define predicate $\mathcal{L}_{\mathrm{1MIS}}$ on configurations as follows: $\mathcal{L}_{\mathrm{1MIS}}(\gamma) = \mathbf{true}$ holds if and only if, in configuration γ, $\{v \in V \mid v.\mathtt{mis} = \mathbf{true}\}$ is a 1-maximal independent set. Our goal is to give a silent and self-stabilizing algorithm for $\mathcal{L}_{\mathrm{1MIS}}$.

3. Loop Composition

We use the loop composition [3] to design a silent self-stabilizing 1-MIS algorithm. The loop composition is a technique to execute a given algorithm repeatedly in a consistent way. To utilize the loop composition, we must design two algorithms $\mathcal{A}$ and $\mathcal{P}$ and a predicate E for a given predicate $\mathcal{L}$. Algorithm $\mathcal{A}$ is a base algorithm that we aim to execute repeatedly. It must satisfy the three requirements, *shiftable convergence*, *loop convergence*, and *correctness*. These requirements are defined in the sequel. Predicate E: $V \mapsto \{\mathbf{false}, \mathbf{true}\}$ is a locally checkable error-detecting predicate. We say that a configuration γ is *erroneous* for E if $E(v)$ holds for some $v \in V$ in γ. Otherwise, we say that γ is *non-erroneous* for E. Algorithm $\mathcal{P}$ is a silent self-stabilizing algorithm that brings the system to a non-erroneous configuration for

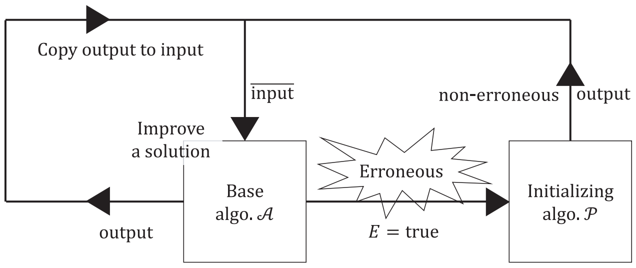


Fig. 1 An execution of $\mathbf{Loop}(\mathcal{A}, E, \mathcal{P})$.

E, starting from any configuration. Then, we obtain a composite algorithm $\mathbf{Loop}(\mathcal{A}, E, \mathcal{P})$ [3], which is a silent and self-stabilizing algorithm for $\mathcal{L}$. Very roughly speaking, $\mathbf{Loop}(\mathcal{A}, E, \mathcal{P})$ shows the following behavior (See **Fig. 1**). Recall that a configuration is final for $\mathcal{A}$ if and only if no action of $\mathcal{A}$ is enabled in any process.

```
1: repeat
2:   if the current configuration is erroneous then
3:     Execute $\mathcal{P}$, which brings the system to a non-erroneous
       configuration.
4:   else
5:     Execute $\mathcal{A}$, which brings the system to a final configu-
       ration with a better solution for $\mathcal{A}$
6:     Copy the outputs of $\mathcal{A}$ to the inputs of $\mathcal{A}$
7:   end if
8: until The current configuration is final and the inputs and the
   outputs of $\mathcal{A}$ are the same.
```

In what follows, we describe the requirements for $\mathcal{A}$ and $\mathcal{P}$ and explain the meaning of *copying* from the outputs of $\mathcal{A}$ to the inputs of $\mathcal{A}$. We define $O_{\mathcal{A}}$ (resp. $O_{\mathcal{P}}$) as the set of variables of $\mathcal{A}$ (resp.$\mathcal{P}$) whose values can be updated by actions of $\mathcal{A}$ (resp.$\mathcal{P}$), and $I_{\mathcal{A}}$ (resp. $I_{\mathcal{P}}$) as the set of variables of $\mathcal{A}$ (resp.$\mathcal{P}$) whose values are never updated and only read by actions of $\mathcal{A}$ (resp. $\mathcal{P}$). We assume $O_{\mathcal{A}} \cap O_{\mathcal{P}} = \emptyset$ and $I_{\mathcal{P}} = \emptyset$. The error detecting predicate $E(v)$ is evaluated by process $v \in V$, and its evaluation depends on variables in $I_{\mathcal{A}} \cup O_{\mathcal{P}}$ of the v-neighbors and v itself. Let $\mathcal{E}$ be a predicate on configurations such that $\mathcal{E}(\gamma)$ holds if and only if $\bigvee_{v \in V} E(v)$ holds in configuration γ. We assume that algorithm $\mathcal{A}$ has a *copying variable* $\overline{x} \in I_{\mathcal{A}}$ for every variable $x \in O_{\mathcal{A}}$. We define γ^{copy} as the configuration obtained by replacing the value of $v.\overline{x}$ with the value of $v.x$ for every process v and every variable $x \in O_{\mathcal{A}}$ in configuration γ. We define predicate $C_{\mathrm{goal}}(\mathcal{A}, E)$ as follows: configuration γ satisfies $C_{\mathrm{goal}}(\mathcal{A}, E)$ if and only if $\gamma \in \neg\mathcal{E}$, $\gamma^{copy} = \gamma$, and no action of $\mathcal{A}$ is enabled in any process. We must design $\mathcal{A}$ to satisfy the following three requirements:

Shiftable Convergence Every maximal execution of $\mathcal{A}$ that starts from a configuration in $\neg\mathcal{E}$ terminates at a configuration γ such that $\gamma^{\mathrm{copy}} \in \neg\mathcal{E}$.

Loop Convergence There exist two integers $L_{\mathcal{A}}$ and $R_{\mathcal{A}}$ that satisfy the following proposition: if $\varrho_0, \varrho_1 \dots$ is an infinite sequence of maximal executions of $\mathcal{A}$ where $\varrho_i = \gamma_{i,0}, \gamma_{i,1}, \dots, \gamma_{i,s_i}, \gamma_{0,0} \in \neg\mathcal{E}$, and $\gamma_{i+1,0} = \gamma_{i,s_i}^{\mathrm{copy}}$ for each $i \geq 0$, then $\gamma_{j,s_j} \in C_{\mathrm{goal}}(\mathcal{A}, E)$ and $R(\varrho_0) + R(\varrho_1) + \dots R(\varrho_j) \leq R_{\mathcal{A}}$ hold for some $j < L_{\mathcal{A}}$, and

Correctness $\gamma \in C_{\mathrm{goal}}(\mathcal{A}, E) \Rightarrow \gamma \in \mathcal{L}$ holds for every configuration γ.

Two intergers $L_{\mathcal{A}}$ and $R_{\mathcal{A}}$ are an upper bound on the number of

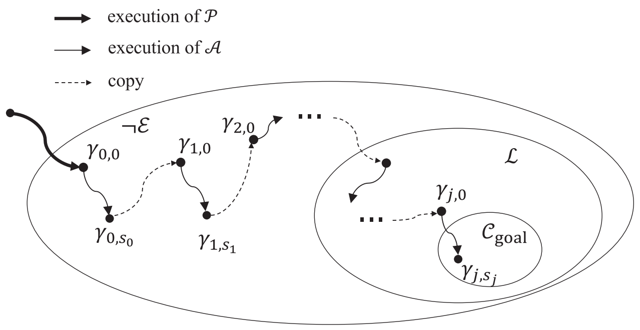


Fig. 2 An execution of **Loop**$(\mathcal{A}, E, \mathcal{P})$.

iterations of $\mathcal{A}$'s executions and an upper bound on the total number of rounds of those (iterated) executions in $\mathcal{A}$. We also must design $\mathcal{P}$ such that every maximal execution of $\mathcal{P}$ terminates at a configuration in $\neg\mathcal{E}$ within $T_{\mathcal{P}}$ rounds.

If we design $\mathcal{A}$, E, and $\mathcal{P}$ that satisfy the above requirements, the composited algorithm **Loop**$(\mathcal{A}, E, \mathcal{P})$ presented in Ref. [3] has the property described in the following theorem.

Theorem 1 (Refs. [3], [4] [*2]). *Algorithm* **Loop**$(\mathcal{A}, E, \mathcal{P})$ *is a silent and self-stabilizing algorithm for predicate $\mathcal{L}$. Every execution of* **Loop**$(\mathcal{A}, E, \mathcal{P})$ *terminates within $O(n + T_{\mathcal{P}} + R_{\mathcal{A}} + L_{\mathcal{A}}D)$ rounds. Its space complexity is $O(S_{\mathcal{A}} + S_{\mathcal{P}} + \log n)$ bits per process, where $S_{\mathcal{A}}$ (resp. $S_{\mathcal{P}}$) is space complexity of $\mathcal{A}$ (resp. $\mathcal{P}$) in bits per process.*

We briefly describe the behavior of **Loop**$(\mathcal{A}, E, \mathcal{P})$ in the rest of this section. In an execution of **Loop**$(\mathcal{A}, E, \mathcal{P})$, the processes simulate an execution of $\mathcal{A}$ or an execution of $\mathcal{P}$. Starting from any configuration, the processes eventually agree with which algorithm they should simulate. When they detect that the current configuration does not satisfy $\neg\mathcal{E}$, they simulate an execution of $\mathcal{P}$, by which a configuration in $\neg\mathcal{E}$ is reached (See the (leftmost) thick arrow in **Fig. 2**). If the current configuration satisfies $\neg\mathcal{E}$, then the processes simulate $\mathcal{A}$. (See the solid arrows in Fig. 2). Whenever a simulated execution of $\mathcal{A}$ terminates, the processes check whether or not they already reach a configuration in $C_{\text{goal}}(\mathcal{A}, E)$. If it does, they will do nothing thereafter. Otherwise, they copy the values of all output variables to the corresponding copy variables, which results in transition corresponding to the dashed arrows in Fig. 2. After that they simulate a new execution of $\mathcal{A}$ again. From the property of the shiftable convergence and the loop convergence of $\mathcal{A}$, they eventually reach a configuration in $C_{\text{goal}}(\mathcal{A}, E)$ and they terminate (although they cannot detect termination). Thereafter, legitimate predicate $\mathcal{L}$ is always satisfied thanks to the correctness property of $\mathcal{A}$.

4. Self-stabilizing 1-MIS Algorithm

In this section, we design a silent self-stabilizing algorithm for constructing a 1-MIS using the loop composition method described in the previous section. Specifically, we give a base algorithm *Inc*, an initialization algorithm *Init* and an error detecting predicate E_{MIS} such that **Loop**$(Inc, E_{\text{MIS}}, Init)$ is a silent and self-stabilizing algorithm for $\mathcal{L}_{\text{1MIS}}$. The space complexity of **Loop**$(Inc, E_{\text{MIS}}, Init)$ is $O(\log n)$ bits per process and the worst case time complexity is $O(nD)$ rounds.

Every process v maintains a Boolean variable $v.\mathtt{mis} \in \{\textbf{false}, \textbf{true}\}$ and the corresponding copying variable $v.\overline{\mathtt{mis}} \in \{\textbf{false}, \textbf{true}\}$. A variable $v.\overline{\mathtt{mis}}$ is an output variable of *Init* and an input variable of *Inc*. A variable $v.\mathtt{mis}$ is not accessed by *Init*. It is updated only by *Inc*. We define two sets $S_I = \{v \in V \mid v.\overline{\mathtt{mis}}\}$ and $S_O = \{v \in V \mid v.\mathtt{mis}\}$. Define $\mathcal{L}_{\text{input}}$ as the predicate on configurations such that $\mathcal{L}_{\text{input}}(\gamma) = \textbf{true}$ if and only if S_I is an MIS in a configuration γ.

Our goal is to design *Inc*, E_{MIS}, and *Init* such that;

- If S_I is not an MIS, i.e., the current configuration deviates from $\mathcal{L}_{\text{input}}$, then at least one process v must detect the deviation with an error detecting predicate $E_{\text{MIS}}(v)$, that is, $\mathcal{L}_{\text{input}}(\gamma)$ holds if and only if $\neg \bigvee_{v \in V} E_{\text{MIS}}(v)$ holds in a configuration γ,
- Every maximal execution of *Init* starting from any configuration terminates within $O(n)$ rounds at a configuration in $\mathcal{L}_{\text{input}}$,
- Every maximal execution ϱ of *Inc* starting from a configuration in $\mathcal{L}_{\text{input}}$ where S_I is not a 1-MIS terminates at a configuration where S_O is an MIS such that $|S_O| \ge |S_I| + 1$, within $O(\epsilon + 1)$ rounds where ϵ is $|S_O| - |S_I|$ in the final configuration of ϱ, and
- Every maximal execution ϱ of *Inc* starting from a configuration in $\mathcal{L}_{\text{input}}$ where S_I is a 1-MIS terminates within $O(1)$ rounds at a configuration where $S_O = S_I$.

Note that the predicate $\mathcal{L}_{\text{input}}$ corresponds to $\neg\mathcal{E}$ in the previous section. If the above conditions hold, then *Inc*, E_{MIS}, and *Init* satisfy all the requirements of the loop composition for $\mathcal{L} = \mathcal{L}_{\text{1MIS}}$, $T_{Init} = O(n)$, $R_{Inc} = O(n)$, $L_{Inc} = n$, thus **Loop**$(Inc, E_{\text{MIS}}, Init)$ is a silent and self-stabilizing algorithm for $\mathcal{L}_{\text{1MIS}}$ and its time complexity is $O(n + T_{Init} + R_{inc} + L_{inc} \cdot D) = O(nD)$ rounds.

We give *Init* and E_{MIS} in Section 4.1 and give *Inc* in Section 4.2.

4.1 Error Detecting Predicate E_{MIS} and Algorithm *Init*

First, we give the error detecting predicate E_{MIS} as follows:

$$E_{\text{MIS}}(v) \equiv (v.\overline{\mathtt{mis}} \land \exists u \in N_v : u.\overline{\mathtt{mis}}) \lor (\neg v.\overline{\mathtt{mis}} \land \forall w \in N_v : \neg w.\overline{\mathtt{mis}}).$$

Lemma 1. *For any configuration γ, $\mathcal{L}_{\text{input}}(\gamma)$ holds if and only if $\neg \bigvee_{v \in V} E_{\text{MIS}}(v)$ holds in γ.*

Proof. Note that $\neg E_{\text{MIS}}(v) \equiv (v.\overline{\mathtt{mis}} \Rightarrow \forall u \in N_v : \neg u.\overline{\mathtt{mis}}) \land (\neg v.\overline{\mathtt{mis}} \Rightarrow \exists w \in N_v : w.\overline{\mathtt{mis}})$. Suppose that $\neg \bigvee_{v \in V} E_{\text{MIS}}(v)$ holds in a configuration γ, that is, we have $\neg E_{\text{MIS}}(v)$ for all $v \in V$ in γ. Then, every process in S_I has no neighbor in S_I and every process in $V \setminus S_I$ has at least one process in S_I. Hence, S_I is an MIS in γ and $\mathcal{L}_{\text{input}}(\gamma)$ holds. Suppose the other case, that is, $\bigvee_{v \in V} E_{\text{MIS}}(v)$ holds in γ. In this case, some process in S_I has a neighbor in S_I or some process in $V \setminus S_I$ has no neighbor in S_I. Hence, S_I is not an MIS in γ and $\mathcal{L}_{\text{input}}(\gamma)$ does not hold. □

Next, we give an algorithm *Init*. The goal of this algorithm is to bring the network to a configuration where S_I is an MIS within $O(n)$ rounds starting from any configuration. The algorithm *Init* consists of only one action I_1 as given in **Table 2**.

In this algorithm, a process with a smaller identifier has a higher priority for becoming a member of S_I. If a process v finds

[*2] Loop composition **Loop**$(\mathcal{A}, E, \mathcal{P})$ was originally given in Ref. [3],and its time complexity was slightly improved by Ref. [4].

Table 2 *Init*.

[Actions of process v]			
I_1:	$v.\overline{\mathtt{mis}}$	$\longleftarrow$	$(\forall w \in N_v : \neg w.\overline{\mathtt{mis}} \vee (v.id < w.id))$

that there is no neighbor with a smaller identifier in S_I, then v becomes a member of S_I, that is, v executes $v.\overline{\mathtt{mis}} \leftarrow$ **true**. Otherwise, if there exists a process with a smaller identifier in $N_v \cap S_I$, v executes $v.\overline{\mathtt{mis}} \leftarrow$ **false**.

We give an example of the initialization in **Fig. 3** (1) and (2). Suppose that an execution of *Inc* begins from a configuration γ in Fig. 3 (1) that $S_I = \{19, 21, 53, 62, 81, 87\}$, which is not an MIS. Then, $\mathcal{L}_{\mathrm{input}}(\gamma)$ does not hold because process $31 \notin S_I$ has no neighbor in S_I and process $19 \in S_I$ has a neighbor in S_I (i.e., 21). Thus, initialization algorithm *Init* is executed. Processes 5 and 71 become members of S_I within one round because they have no neighbor in S_I with a smaller identifier and processes 13, 19, 21, 28, 31, 62, 79, and 87 converge to non-members of S_I by the end of the next round because they have neighbors 5 or 71. After that, process 23 becomes a member of S_I within one round because they have no neighbor in S_I with a smaller identifier and processes 35, 53, and 67 converge to non-members of S_I by the end of the next round because process 23 is their neighbor. Thus, any execution of *Init* terminates at a configuration whose output is an MIS of G, in this example, a configuration such that $S_I = \{5, 23, 71\}$.

Lemma 2. *Every maximal execution of Init starting from any configuration terminates within $O(n)$ rounds at a configuration in $\mathcal{L}_{\mathrm{input}}$.*

Proof. First, we claim that any final configuration of *Init* satisfies $\mathcal{L}_{\mathrm{input}}$. By definition of notation "$v.x \longleftarrow \chi(v)$" (See Section 2), a process v is enabled if and only if $v.\overline{\mathtt{mis}} \not\equiv (\forall w \in N_v : \neg w.\overline{\mathtt{mis}} \vee (v.id < w.id)))$. Therefore, in any final configuration γ, where no process is enabled, every process v in S_I has no neighbor in S_I and every process v in $V \setminus S_I$ has at least one neighbor in S_I. Thus, any final configuration satisfies $\mathcal{L}_{\mathrm{input}}$.

Next, we prove that every maximal execution terminates within $O(n)$ rounds. Let $v_1, v_2, \ldots, v_n$ be the processes in V such that $v_1.id < v_2.id < \cdots < v_n.id$. The guard of I_1 in process v, i.e., $v.\overline{\mathtt{mis}} \not\equiv (\forall w \in N_v : \neg w.\overline{\mathtt{mis}} \vee (v.id < w.id))$, depends only on $v.\overline{\mathtt{mis}}$ and $w.\overline{\mathtt{mis}}$ such that $w.id < v.id$. Therefore, v_1 becomes disabled in the first round of any maximal execution of *Init* and never becomes enabled thereafter. Similarly, v_i becomes disabled within one round after all neighboring processes with smaller identifiers than v_i are disabled. Thus, any maximal execution terminates within $O(n)$ rounds. □

4.2 Algorithm *Inc*

We give an algorithm *Inc* in this section. This algorithm assumes that S_I is an MIS of G. The goal of this algorithm is to bring the network to a configuration where S_O is an MIS such that $|S_O| \geq |S_I| + 1$ if S_I is not a 1-MIS. Otherwise the goal is to reach a configuration where $S_O = S_I$ holds. Note that if S_I is not a 1-MIS, S_O such that $|S_O| \geq |S_I| + 1$ holds necessarily exists.

4.2.1 Key Idea

In this subsection, we give a key idea to find an MIS S_O such that $S_O = S_I$ if S_I is a 1-MIS of G, otherwise $|S_O| \geq |S_I| + 1$.

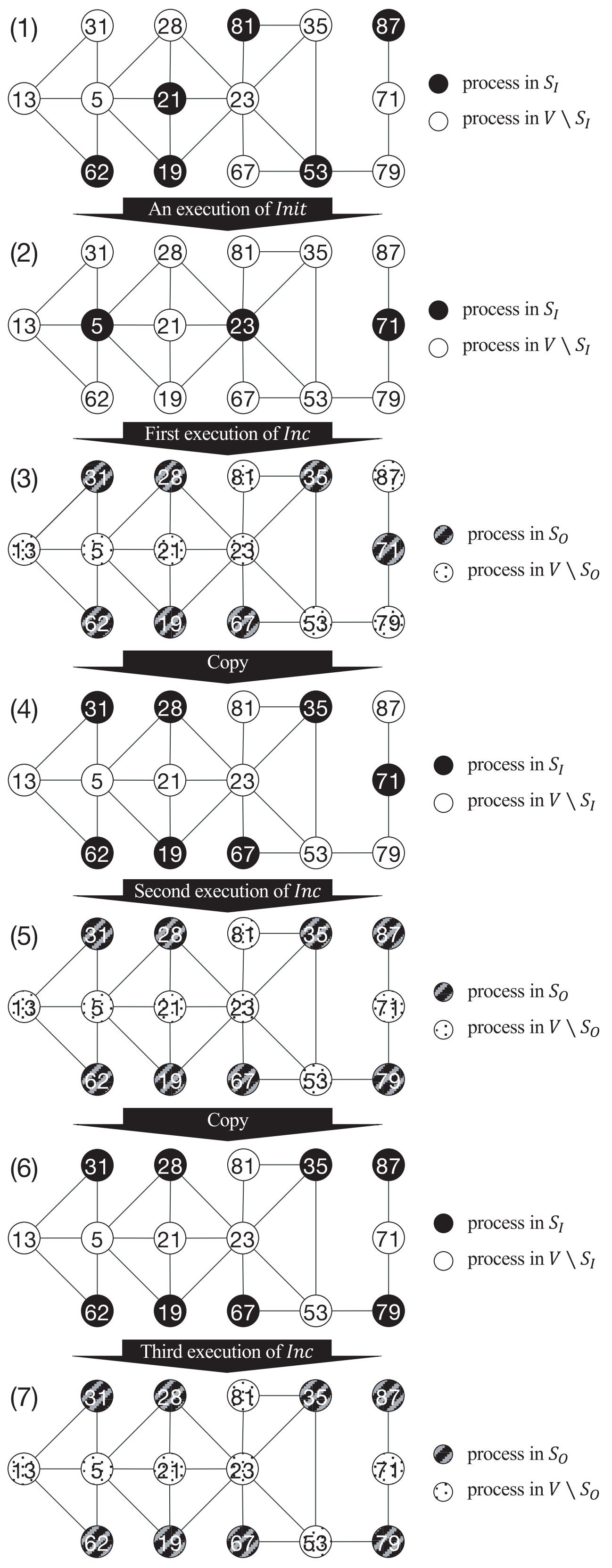

(1)
process in S_I
process in $V \setminus S_I$
An execution of *Init*
(2)
process in S_I
process in $V \setminus S_I$
First execution of *Inc*
(3)
process in S_O
process in $V \setminus S_O$
Copy
(4)
process in S_I
process in $V \setminus S_I$
Second execution of *Inc*
(5)
process in S_O
process in $V \setminus S_O$
Copy
(6)
process in S_I
process in $V \setminus S_I$
Third execution of *Inc*
(7)
process in S_O
process in $V \setminus S_O$

Fig. 3 Example of *Init* and *Inc*.

Implementation of this idea as a distributed algorithm will be described in Section 4.2.2.

First, we define the parent-child relationship on processes: If a process $v \in V \setminus S_I$ has exactly one neighbor u in S_I, i.e., $N_v \cap S_I = \{u\}$, we say that u is a *parent* of v and v is a *child*

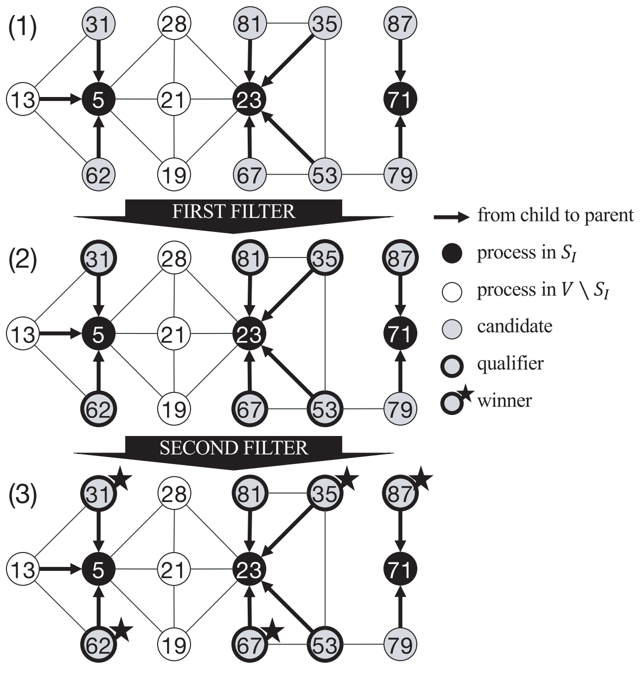


Fig. 4 The first and the second filters.

of u. The relationship only constructs trees of height 1. As we will see later, each process v memorizes the identifier of its parent in v.parent if v has a parent. Otherwise, we assign v.parent $= \perp$. Define C_u as the set of children of process u, that is, $C_u = \{v \in N_u \mid v.\mathtt{parent} = u\}$. Let u and v be any two processes such that u is a parent of v, i.e., v.parent $= u$. If $|C_u| - |C_u \cap N_v| \geq 2$, we call v a *candidate*. In other words, v is a candidate if and only if it has a parent and this parent has another child in $V \setminus N_v$. Let us see an example (See **Fig. 4** (1)). Process 31 is a candidate because its parent, process 5, has another child, process 62, which is not a neighbor of process 31. On the other hand, process 13 is not a candidate because all other children of process 5 are neighbors of Process 13. We use the word "candidate" because we can increase the size of the MIS by adding some candidates and removing their parent. For example, we obtain a larger MIS than S_I if we remove process 5 and add the two non-neighboring candidates among its children, processes 31 and 62. However, we cannot add all candidates to obtain a larger MIS. This is because some of the candidates may be neighbors and thus have conflicts to join the independent set. For example, we cannot add both processes 53 and 79, and we cannot add both processes 35 and 81.

Next, we perform two distinct filters to avoid the conflicts. We call a process that survives the first filter (resp. the second filter) a *qualifier* (resp. a *winner*). If a candidate v does not have any neighboring candidate w such that w.parent $<$ v.parent, v is a qualifier. Otherwise, v is not a qualifier. For example, in the example of Fig. 4 (2), process 79 is not a qualifier because the identifier of its parent is 71 and one of its neighbors, process 53, has a parent with identifier $23 < 71$. All the other candidates are qualifiers in this example. Any two qualifiers that have distinct parents are not neighbors, thus do not have an inter-tree conflict while some two qualifiers with the same parent may have an intra-tree conflict. The second filter chooses winners among the qualifiers by comparing their identifiers in the same way as algorithm *Init*. The winners are decided by each parent such that it has two or more qualifiers. Let $q_1, q_2, \dots, q_s$ be the qualifiers of G in ascending order, i.e., $q_1.id < q_2.id < \dots q_s.id$. Define the set W of qualifiers recursively as follows; $q_1 \in W$, and for each $i \geq 2$, $q_i \in W$ if and only if there is no q_i-neighbor $q_j \in W$ such that $j < i$. The qualifiers in the resulting W are winners and the other qualifiers are non-winners. In the example of Fig. 4 (3), processes 31, 35, 62, 67, and 87 are winners. We have the following two lemmas about winners.

Lemma 3. *There is at least one process that has two or more winners in its children if S_I is not a 1-MIS of G.*

Proof. Let u be the process with the minimum identifier such that it has a candidate among its children. Such u clearly exists if S_I is not a 1-MIS. Let c_1 be the candidate in C_u with the smallest identifier. By definition of the candidates, there exists one or more other candidates in C_u that are not c_1-neighbors. Let c_2 be the candidate with the smallest identifier among them. Both c_1 and c_2 survive the first and the second filters thanks to the minimality of u's identifier and the absence of a candidate in C_u that makes c_1 or c_2 drop in the second filter. Hence, there exist at least two winners in u's children. □

Lemma 4. *There exists no winner if S_I is a 1-MIS of G.*

Proof. Assume for contradiction that S_I is a 1-MIS of G and there is a candidate $v \in V$. Let u be the parent of v. By definition of a candidate, u has at least one candidate other than v, say w, in its children, which is not a neighbor of v. Since v and w are candidates, $S_I \cap (N_v \cup N_w) = \{u\}$ holds. Therefore, $S_O = S_I \cup \{v, w\} \setminus \{u\}$ is an independent set and $|S_O| = |S_I| + 1$, which contradicts the assumption that S_I is a 1-MIS of G. □

Finally, we choose $S_O = S_A(S_I) \cup S_B(S_I) \cup S_C(S_I)$ where $S_A(S_I)$, $S_B(S_I)$, and $S_C(S_I)$ are the sets of processes that we will define in the following. These sets depend on S_I, but we always omit S_I from their notations, i.e., just write S_A, S_B, and S_C, whenever it is clear from the context. Define S_A as the set of all processes u in S_I such that u has only one or no winners in its children. Define S_B as the set of all winners v such that v.parent $\notin S_A$. By definition, $S_A \cup S_B$ is an independent set of G. Moreover, $|S_A \cup S_B| \geq |S_I| + |S_B|/2 \geq |S_I| + 1$ holds if S_I is not a 1-MIS. This is because (i) $S_B \neq \emptyset$ holds by Lemma 3, and (ii) each process in $S_I \setminus S_A$ has at least two winners in its children, thus $|S_B| \geq 2|S_I \setminus S_A|$ holds. On the other hand, $S_A = S_I$ and $S_B = \emptyset$ holds if S_I is a 1-MIS, by Lemma 4. Note that $S_A \cup S_B$ is an independent set but may not be an MIS of G (if S_I is not a 1-MIS). In the example of Fig. 4 (3), $S_A \cup S_B = \{31, 35, 62, 67, 71\}$ is not an MIS because $S_A \cup S_B \cup \{21\}$ and $S_A \cup S_B \cup \{19, 28\}$ are also independent sets of G. Let $u_1, u_2, \dots, u_s$ be the processes in $V \setminus (S_A \cup S_B)$ such that there is no neighbor in $S_A \cup S_B$ in ascending order, i.e., $u_1.id < u_2.id < \dots u_s.id$. Define the set S_C recursively as follows; $u_1 \in S_C$, and for each $i \geq 2$, $u_i \in S_C$ if and only if there is no u_i-neighbor u_j such that $j < i$. Thus, $S_C = \{19, 28\}$ in the example of Fig. 4 (3). The set $S_A \cup S_B \cup S_C$ is an independent set since each process in S_C does not have a neighbor in $S_A \cup S_B \cup S_C$. Furthermore, $S_A \cup S_B \cup S_C$ is an MIS of G because otherwise there must be a non-winner $v \in V \setminus S_I$ such that there exists no v-neighbor in $S_A \cup S_B \cup S_C$, but it implies $v \in S_C$ by definition of S_C, a contradiction. Therefore, we have the following three lemmas.

Lemma 5. *The set* $S_A \cup S_B \cup S_C$ *is an MIS of* G *if* S_I *is an MIS of* G.

Lemma 6. $|S_A \cup S_B \cup S_C| \geq |S_I| + |S_B|/2 + |S_C| \geq |S_I| + 1$ *if* S_I *is an MIS but not a 1-MIS of* G.

Lemma 7. $S_A \cup S_B \cup S_C = S_I$ *and* $S_B = S_C = \emptyset$ *if* S_I *is a 1-MIS of* G.

Thus we achieve our goal by letting $S_O = S_A \cup S_B \cup S_C$. If S_I is not a 1-MIS, $|S_O| \geq |S_I| + 1$. Otherwise, $S_O = S_I$. In the example of Fig. 4 (3), $S_A = \{71\}$, $S_B = \{31, 35, 62, 67\}$, $S_C = \{19, 28\}$, and hence $|S_O| = 7 > 3 = |S_I|$ (See Fig. 3 (2) and (3)).

We show how to reach a configuration in $\mathcal{C}_{\mathrm{goal}}(\mathit{Inc}, E_{\mathrm{MIS}})$ for S_I in Fig. 3. When the first execution of *Inc* terminates, processes copy the values of all output variables to the corresponding copy variables, that is, S_O computed by the first execution of *Inc* (See Fig. 3 (3)) is copied to S_I of the second execution of *Inc* (See Fig. 3 (4)). When the second execution of *Inc* terminates, $S_O = \{19, 28, 31, 35, 62, 67, 79, 87\}$ holds because processes 79 and 87 are winners and only process 71 in S_I has two or more winners in its neighbors (See Fig. 3 (5)). In the third execution of *Inc*, $S_I = S_O$ holds because there exists no winner, that is, S_I is a 1-MIS of G (See Fig. 3 (6)). Hence, all the processes do nothing thereafter (See Fig. 3 (7)).

4.2.2 Distributed Implementation

Each process v has six variables $v.\mathtt{parent} \in ID$, $v.\mathtt{numchild} \in \{0, 1, 2, \ldots, |N_v|\}$, $v.\mathtt{cand} \in \{\mathbf{false}, \mathbf{true}\}$, $v.\mathtt{qualifier} \in \{\mathbf{false}, \mathbf{true}\}$, $v.\mathtt{winner} \in \{\mathbf{false}, \mathbf{true}\}$, and $v.\mathtt{mis} \in \{\mathbf{false}, \mathbf{true}\}$, and the corresponding copying variables for the six variables. Each child v, i.e., a process in $V \setminus S_I$ that has exactly one neighbor in S_I, stores the identifier of its parent on $v.\mathtt{parent}$. Each process u in S_I stores the number of its children, i.e., $|C_u|$, on $u.\mathtt{numchild}$. Since $|ID| = O(\mathit{poly}(n))$, the two variables `parent` and `numchild` require $O(\log n)$ bits per each process. The Boolean variable $v.\mathtt{cand}$, $v.\mathtt{qualifier}$, and $v.\mathtt{winner}$ represent whether or not a process v is a candidate, a qualifier, and a winner, respectively.

Table 3 *Inc*.

[Actions of process v]			
M$_1$:	$v.\mathtt{parent}$	$\longleftarrow$	$\mathit{Parent}(v)$
M$_2$:	$v.\mathtt{numchild}$	$\longleftarrow$	$\lvert C_v \rvert$
M$_3$:	$v.\mathtt{cand}$	$\longleftarrow$	$\mathit{Cand}(v)$
M$_4$:	$v.\mathtt{qualifier}$	$\longleftarrow$	$\mathit{Qualifier}(v)$
M$_5$:	$v.\mathtt{winner}$	$\longleftarrow$	$\mathit{Winner}(v)$
M$_6$:	$v.\mathtt{mis}$	$\longleftarrow$	$\mathit{InS}_A(v) \vee \mathit{InS}_B(v) \vee \mathit{InS}_C(v)$

Actions of *Inc* are given in **Table 3** and the functions used in Table 3 are given in **Table 4**. We use a hierarchical composition to design *Inc*. Actions $M_1, M_2, \ldots, M_6$ maintain variables `parent`, `numchild`, `cand`, `qualifier`, `winner`, and `mis`, respectively. We say that an action M_i converges if all of $M_1, M_2, \ldots, M_i$ are disabled in all the processes. Generally, action M_i $(i \geq 2)$ refers the variables maintained by $M_1, M_2, \ldots, M_{i-1}$. Therefore, before M_{i-1} converges, some of those variables in some process may not be correct, thus action M_i does not compute the correct value of the variable it maintains. However, after M_{i-1} converges, those variables maintained by $M_1, M_2, \ldots, M_{i-1}$ are correct in all the processes, thus action M_i can use the correct values of those variables.

Actions M_1, M_2, M_3, and M_4 are simple and straightforward. By these actions, each process v computes its parent (if it has), the number of its children, whether or not it is a candidate, and whether or not it is a qualifier, respectively. Action M_5 simulates the second filter. By M_5, a qualifier v sets $v.\mathtt{winner} = \mathbf{true}$ (becomes a winner) if and only if every $w \in N_v$ such that $w.\mathtt{winner} = \mathbf{true}$ has a larger identifier than v. The second filter implemented by Action M_5 obviously computes the correct value of `winner`, i.e., $v.\mathtt{winner}$ holds if and only if $v \in W$, the set of winners. However, it sometimes requires more than a constant number of rounds. In the example shown in Fig. 4 (2), process 67 may become the first winner because there is no winner among its neighbors in the configuration shown in the figure. However, process 53 may become a winner later, and then, process 67 will get back to a non-winner since process 53 has a smaller identifier. Process 53 also must get back to a non-winner because a neighboring process 35 with a smaller identifier eventually becomes a winner. After that, process 67 will become a winner again because now it has no winner among its neighbors. Eventually, an execution of *Inc* reaches the configuration shown in Fig. 4 (3). As we will see later, the flipping behavior like this example may require $\Theta(k)$ rounds in the worst case when some process in S_I has k winners among its children after this filter converges.

A variable $v.\mathtt{mis}$ is maintained by Action M_6. Our goal is to set $v.\mathtt{mis}$ such that $S_O = S_A \cup S_B \cup S_C$ holds. After M_5 converges, every process $v \in S_I$ computes $v.\mathtt{mis}$ correctly within one round by M_6; every $v \in S_I$ executes $v.\mathtt{mis} \leftarrow \mathbf{true}$ if and only if $v \in S_A$, i.e., there is only one or no winner among v's children. Every $v \in W$ computes $v.\mathtt{mis}$ correctly in the follow-

Table 4 Functions of *Inc*.

$$\mathit{Parent}(v) = \begin{cases} w.\mathit{id} \text{ for } w \in N_v \text{ with } w.\overline{\mathtt{mis}} = \mathbf{true} & \mathbf{if}\ |\{w \in N_v \mid w.\overline{\mathtt{mis}}\}| = 1 \\ \perp & \mathbf{otherwise} \end{cases}$$

$$C_v = \{w \in N_v \mid w.\mathtt{parent} = v\}$$

$$\mathit{Cand}(v) \equiv v.\mathtt{parent} \neq \perp$$
$$\wedge\ (v.\mathtt{parent}).\mathtt{numchild} - |\{w \in N_v \mid v.\mathtt{parent} = w.\mathtt{parent}\}| \geq 2$$

$$\mathit{Qualifier}(v) \equiv v.\mathtt{cand} \wedge (\forall w \in N_v : w.\mathtt{cand} \Rightarrow v.\mathtt{parent} \leq w.\mathtt{parent})$$

$$\mathit{Winner}(v) \equiv v.\mathtt{qualifier} \wedge (\forall w \in N_v : v.\mathit{id} < w.\mathit{id} \vee \neg w.\mathtt{winner})$$

$$\mathit{InS}_A(v) \equiv v.\overline{\mathtt{mis}} \wedge |\{w \in C_v \mid w.\mathtt{winner}\}| \leq 1$$

$$\mathit{InS}_B(v) \equiv \neg v.\overline{\mathtt{mis}} \wedge v.\mathtt{winner} \wedge \neg(v.\mathtt{parent}).\mathtt{mis}$$

$$\mathit{InS}_C(v) \equiv \neg v.\overline{\mathtt{mis}} \wedge \neg v.\mathtt{winner} \wedge \begin{pmatrix} \forall w \in N_v \ s.t.\ w.\mathtt{mis} : \\ \neg \left(w.\overline{\mathtt{mis}} \vee w.\mathtt{winner} \vee w.\mathit{id} < v.\mathit{id}\right) \end{pmatrix}$$

ing rounds; it executes $v.\mathtt{mis} \leftarrow \textbf{true}$ if and only if $v \in S_B$, i.e., $(v.\mathtt{parent}).\mathtt{mis} = \textbf{false}$. Thereafter, every process $v \notin S_I \cup W$ computes $v.\mathtt{mis}$ correctly; it executes $v.\mathtt{mis} \leftarrow \textbf{true}$ if and only if $v \in S_C$, i.e., there is no v's neighbor w such that $w \in S_A \cup S_B$ or $w \in S_C \wedge w.id < v.id$. Since the last computation (for $v \notin S_I \cup W$) is recursive, it requires $O(|S_C|)$ rounds for the same reason as the computation of M_5.

Lemma 8. *Every maximal execution ϱ of Inc starting from a configuration in $\mathcal{L}_{\mathrm{input}}$ terminates at a configuration where $S_O = S_A \cup S_B \cup S_C$, within $O(1 + |S_B| + |S_C|)$ rounds.*

Proof. By definition of the six actions of *Inc*, $S_O = S_A \cup S_B \cup S_C$ holds when ϱ terminates. Therefore, it suffices to show that ϱ terminates within $O(1 + |S_B| + |S_C|)$ rounds. Actions M_1, M_2, M_3, and M_4 converge within $O(1)$ rounds because Action M_i $(1 \le i \le 4)$ refers only variables that are maintained by the actions $M_1 \dots, M_{i-1}$. The same does not hold for Action M_5 because it is recursive in the sense that a qualifier v may refer a variable `winner` of some neighbors to compute its own `winner`. In the following, for any process v and any variable x, we say that $v.x$ converges at a point of a maximal execution if v does not change the value of $v.x$ after the point. Let u be any process in S_I and W_u be the set of winners among u's children, i.e., $W_u = W \cap C_u$. Let $w_{u,1}, w_{u,2}, \dots, w_{u,|W_u|}$ be the processes in W_u in ascending order of their identifiers. After M_4 converged, $w_{u,1}.\mathtt{winner}$ converges to **true** within one round and $v.\mathtt{winner}$ of all its neighbors v converges to **false** within the next round. After that, $w_{u,2}.\mathtt{winner}$ converges within one round since $w_{u,2}$ has a smaller identifier than any qualifier in $N_{w_{u,2}} \setminus N_{w_{u,1}}$. Generally, $w_{u,i}.\mathtt{winner}$ converges to **true** within $O(i)$ rounds after M_4 converged. Thus, Action M_5 converges within $O(\max_{u \in S_I} |W_u|) = O(|S_B|)$ rounds. After M_5 converges, M_6 converges within $O(|S_C|)$ rounds for the same reason. □

In what follows, we show that algorithm *Inc* satisfies three requirements, shiftable convergence, loop convergence, and correctness.

Lemma 9 (Shiftable Convergence)**.** *Every maximal execution ϱ of Inc starting from a configuration in $\mathcal{L}_{\mathrm{input}}$ terminates at a configuration γ such that $\gamma^{\mathrm{copy}} \in \mathcal{L}_{\mathrm{input}}$.*

Proof. In the final configuration γ, $S_O = S_A \cup S_B \cup S_C$ holds by Lemma 8. This set S_O is an MIS of G by Lemma 5. Clearly, S_O in γ is equal to S_I in γ^{copy}. Then, S_I is an MIS of G in γ^{copy}, which yields $\gamma^{\mathrm{copy}} \in \mathcal{L}_{\mathrm{input}}$. □

Recall that, for any configuration of *Inc*, $\gamma \in \mathcal{C}_{\mathrm{goal}}(Inc, E_{\mathrm{MIS}})$ means that $\gamma \in \mathcal{L}_{\mathrm{input}}$, $\gamma^{copy} = \gamma$, and no action of *Inc* is enabled in any process.

Lemma 10 (Loop Convergence)**.** *If $\varrho_0, \varrho_1, \dots$ is an infinite sequence of maximal executions of $\mathcal{A}$ where $\varrho_i = \gamma_{i,0}, \gamma_{i,1}, \dots, \gamma_{i,s_i}$, $\gamma_{0,0} \in \mathcal{L}_{\mathrm{input}}$, and $\gamma_{i+1,0} = \gamma_{i,s_i}^{\mathrm{copy}}$ for each $i \ge 0$, then $\gamma_{j,s_j} \in \mathcal{C}_{\mathrm{goal}}(Inc, E_{\mathrm{MIS}})$ and $R(\varrho_0) + R(\varrho_1) + \dots R(\varrho_j) = O(n)$ hold for some $j \le n$.*

Proof. By Lemma 9, the initial configuration of every execution ϱ_i satisfies $\gamma_{i,0} \in \mathcal{L}_{\mathrm{input}}$ by induction on $i \ge 0$. Let $S_{I,0}, S_{I,1}, \dots$ be S_I in the initial configurations $\gamma_{0,0}, \gamma_{1,0}, \dots$ of $\varrho_0, \varrho_1, \dots$, respectively. Let $S_{O,0}, S_{O,1}, \dots$ be S_O in the final configurations $\gamma_{0,s_0}, \gamma_{1,s_1}, \dots$ of $\varrho_0, \varrho_1, \dots$, respectively. By definition, $S_{I,i+1} = S_{O,i}$ holds for all $i \ge 0$. By Lemmas 6 and 8, $|S_{I,i}| < |S_{I,i+1}|$ holds unless $S_{I,i}$ is a 1-MIS of G. Since $|S_{I,i}| < n$ holds for all i and $|S_{I,0}| \ge 1$, there exists $j' < n$ such that $S_{I,j'} = S_{I,j'+1} = S_{I,j'+2} = \dots$, i.e., $S_{I,k}$ is the same for all $k \ge j'$ by Lemmas 7 and 8. We consider the minimum such j' in what follows. Algorithm *Inc* refers only variable $\overline{\mathtt{mis}}$ among the six copying variables. Therefore, letting $v \in V$ be any process, $v.\overline{\mathtt{mis}}$ in $\gamma_{j',0}$ equals to $v.\mathtt{mis}$ in $\gamma_{j',s_{j'}}$ because $\gamma_{j',s_{j'}}^{\mathrm{copy}} = \gamma_{j'+1,0}$. and $v.\overline{\mathtt{mis}}$ never changes in execution $\varrho_{j'+1}$. Therefore, $\gamma_{j',s_{j'}}^{\mathrm{copy}} = \gamma_{j',s_{j'}}$, which yields $\gamma_{j',s_{j'}} \in \mathcal{C}_{\mathrm{goal}}(Inc, E_{\mathrm{MIS}})$. Lemmas 6, 7, and 8 give $R(\varrho_0) + R(\varrho_1) + \dots R(\varrho_{j'}) = O(n)$. □

Lemma 11 (Correctness)**.** *Every configuration $\gamma \in \mathcal{C}_{\mathrm{goal}}(Inc, E_{\mathrm{MIS}})$ satisfies $\mathcal{L}_{1\mathrm{MIS}}$.*

Proof. We have $S_I = S_O$ because $\gamma \in \mathcal{C}_{\mathrm{goal}}(Inc, E_{\mathrm{MIS}})$. Assume for contradiction that $S_O (= S_I)$ is not a 1-MIS of G in γ. Since γ is a final configuration (i.e., no process is enabled), $S_O = S_A(S_I) \cup S_B(S_I) \cup S_C(S_I)$ holds by Lemma 8. Therefore, $|S_O| \ge |S_I| + 1$ holds by the assumption and Lemma 6, which yields $S_O \ne S_I$, a contradiction. □

Theorem 2. *Algorithm* $\mathbf{Loop}(Inc, E_{\mathrm{MIS}}, Init)$ *is silent and self-stabilizing for $\mathcal{L}_{1\mathrm{MIS}}$. Every maximal execution of* $\mathbf{Loop}(Inc, E_{\mathrm{MIS}}, Init)$ *starting from any configuration terminates within $O(nD)$ rounds. Algorithm* $\mathbf{Loop}(Inc, E_{\mathrm{MIS}}, Init)$ *uses $O(\log n)$ bits per process.*

Proof. Immediately follows from Theorem 1 and Lemmas 1, 2, 9, 10, and 11 because $O(n + T_{Init} + R_{Inc} + L_{Inc} \cdot D) = O(nD)$. □

5. Conclusion

We have presented a silent self-stabilizing algorithm for the 1-maximal independent set problem by using the loop-composition [3]. The time complexity is $O(nD)$ rounds and the space complexity is $O(\log n)$ bits per process, where n is the number of processes and D is the diameter of a given network.

We have seen that the loop-composition technique fits well to the 1-maximal independent set (1-MIS) problem so that we successfully obtained a self-stabilizing 1-MIS algorithm. The loop composition fits this problem well because (i) we can detect whether $\{v \in V \mid v.\mathtt{mis} = \textbf{true}\}$ is an MIS or not, (ii) we have a self-stabilizing MIS algorithm, and (iii) we can design an algorithm *Inc* such that given a feasible solution (i.e., MIS) X as an input, *Inc* outputs a better MIS (i.e., an MIS with larger size) if X is not a 1-MIS, and *Inc* outputs X without any change if X is already a 1-MIS. From these facts (i), (ii), and (iii), all we have to do is to construct an MIS when we detect that the current input is not an MIS, and execute the algorithm $\mathcal{A}$ repeatedly. Namba [8] implements this strategy in a naive way: each process executes n instances of *Inc* in parallel such that the k-th instance uses the output of $(k-1)$-th instance as its input for any k. Thus, his algorithm requires $O(n \log n)$ bits and has to know (at least an upper bound on) n. The loop composition is a framework that enables executing a given algorithm (i.e., *Inc*) repeatedly in a sequential way, not in parallel. Thus, by using the loop composition, we succeeded in implementing the above strategy in a much more sophisticated way. As a result, we obtained a self-stabilizing 1-MIS algorithm with $O(\log n)$ bits per process, which requires no global knowledge such as n.

The loop composition is a general framework, not specific to the 1-MIS problem. Once we design time and/or space efficient modules $\mathcal{A}$, E, and $\mathcal{P}$ for any problem, we can immediately obtain an efficient self-stabilizing algorithm for the problem by using the loop composition. Our future work is utilizing the loop composition to design self-stabilizing algorithms for various problems.

Acknowledgments This work was supported by JSPS KAKENHI Grant Numbers 19H04085, 19K11826, and 20H04140 and JST SICORP Grant Number JPMJSC1606.

References

[1] Awerbuch, B., Luby, M., Goldberg, A.V. and Plotkin, S.A.: Network decomposition and locality in distributed computation, *30th Annual Symposium on Foundations of Computer Science*, pp.364–369, IEEE (1989).

[2] Bollobás, B., Cockayne, E.J. and Mynhardt, C.M.: On generalised minimal domination parameters for paths, *Annals of Discrete Mathematics*, Vol.48, pp.89–97, Elsevier (1991).

[3] Datta, A.K., Larmore, L.L., Masuzawa, T. and Sudo, Y.: A self-stabilizing minimal k-grouping algorithm, *Proc. 18th International Conference on Distributed Computing and Networking*, p.3, ACM (2017).

[4] Datta, A.K., Larmore, L.L., Masuzawa, T. and Sudo, Y.: A self-stabilizing minimal k-grouping algorithm, arXiv preprint arXiv: 1907.10803 (2019).

[5] Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control, *Comm. ACM*, Vol.17, No.11, pp.643–644 (1974).

[6] Dolev, S.: *Self-stabilization*, MIT press (2000).

[7] Ikeda, M., Kamei, S. and Kakugawa, H.: A space-optimal self-stabilizing algorithm for the maximal independent set problem, *3rd International Conference on Parallel and Distributed Computing, Applications and Technologies* (*PDCAT*), pp.70–74 (2002).

[8] Namba, E.: A hierachical self-stabilizing 1-MIS algorihtm (in Japanese), Master's thesis, Osaka University (2017).

[9] Shi, Z., Goddard, W. and Hedetniemi, S.T.: An anonymous self-stabilizing algorithm for 1-maximal independent set in trees, *Information Processing Letters*, Vol.91, No.2, pp.77–83 (2004).

[10] Shukla, S.K., Rosenkrantz, D.J., Ravi, S.S., et al.: Observations on self-stabilizing graph algorithms for anonymous networks, *Proc. 2nd Workshop on Self-stabilizing Systems*, Vol.7, p.15 (1995).

[11] Tanaka, H., Sudo, Y., Kakugawa, H., Masuzawa, T. and Datta, A.K.: A self-stabilizing 1-maximal independent set algorithm, *International Symposium on Stabilizing, Safety, and Security of Distributed Systems*, pp.338–353, Springer (2019).

[12] Turau, V.: Linear self-stabilizing algorithms for the independent and dominating set problems using an unfair distributed scheduler, *Information Processing Letters*, Vol.103, No.3, pp.88–93 (2007).

Editor's Recommendation

This paper reports a self-stabilizing distributed iterative algorithm for the 1-maximal independent set problem. The paper is described concisely and simply, including the problem setup, definition, and application of the algorithm, and is considered to be highly useful. Therefore, the paper has been selected as a recommended paper.

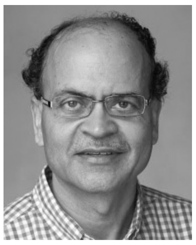
(Program chair of 2019 IPSJ Kansai-Branch Convention, Kohei Ichikawa)

Hideyuki Tanaka received his B.E. degree in computer science from Osaka University in 2019. He is now a master's student at the Graduate School of Information Science and Technology, Osaka University. His research interests include distributed algorithms.

Yuichi Sudo received his B.E., M.E., and Ph.D. degrees in computer science from Osaka University in 2009, 2011, and 2015. He worked at NTT Corporation and was engaged in research on network security during 2011–2017. He is now an assistant professor at the Graduate School of Information Science and Technology, Osaka University. His research interests include distributed algorithms, graph theory, and network security.

Hirotsugu Kakugawa received his B.E. degree in engineering in 1990 from Yamaguchi University, and his M.E. and D.E. degrees in information engineering in 1992, 1995 respectively from Hiroshima University. He is currently a professor of Ryukoku University. He is a member of the IEEE Computer Society and the Information Processing Society of Japan.

Toshimitsu Masuzawa received his B.E., M.E. and D.E. degrees in computer science from Osaka University in 1982, 1984 and 1987. He had worked at Osaka University during 1987–1994, and was an associate professor of Graduate School of Information Science, Nara Institute of Science and Technology (NAIST) during 1994–2000. He is now a professor of Graduate School of Information Science and Technology, Osaka University. He was also a visiting associate professor of Department of Computer Science, Cornell University between 1993–1994. His research interests include distributed algorithms, parallel algorithms and graph theory. He is a member of ACM, IEEE, IEICE and IPSJ.



Ajoy K. Datta is a professor of Computer Science at the University of Nevada Las Vegas. His research interests are in the areas of Distributed Computing and Self-Stabilization.