Self-Stabilizing *(f,g)*-Alliances with Safe Convergence

Fabienne Carrier[1], Ajoy K. Datta[2], Stéphane Devismes[1], Lawrence L. Larmore[2], and Yvan Rivierre[1]

[1] VERIMAG UMR 5104, Université Joseph Fourier, France,
firstname.lastname@imag.fr
[2] School of Computer Science, University of Nevada Las Vegas, USA,
firstname.lastname@unlv.edu

Abstract. Given two functions f and g mapping nodes to non-negative integers, we give a silent self-stabilizing algorithm that computes a minimal (f, g)-alliance in an asynchronous network with unique node IDs, assuming that every node p has a degree at least $g(p)$ and satisfies $f(p) \geq g(p)$. Our algorithm is *safely converging* in the sense that starting from any configuration, it first converges to a (not necessarily minimal) (f, g)-alliance in at most four rounds, and then continues to converge to a minimal one in at most $5n+4$ additional rounds, where n is the size of the network. Our algorithm is written in the shared memory model. It is proven assuming an unfair (distributed) daemon. Its memory requirement is $O(\log n)$ bits per process, and it takes $O(\Delta^3 n)$ steps to stabilize, where Δ is the degree of the network.

Keywords: Self-Stabilization, Safe Convergence, (f, g)-Alliance.

1 Introduction

Self-stabilization [2] is a versatile technique to withstand *any* transient fault in a distributed system. Informally, a distributed algorithm is self-stabilizing if, after transient faults hit the system and place it in some arbitrary configuration, the system recovers without external (*e.g.*, human) intervention in finite time. Thus, self-stabilization makes no hypothesis on the nature or extent of transient faults that could hit the system, and recovers from the effects of those faults in a unified manner. However, self-stabilization has some drawbacks; perhaps the main one is *temporary loss of safety*, *i.e.*, after the occurrence of transient faults, there is a finite period of time — called the *stabilization phase* — before the system returns to a legitimate configuration. During this phase, there is no guarantee of safety. Several approaches have been introduced to offer more stringent guarantees during the stabilization phase, *e.g.*, *fault-containment* [7], *superstabilization* [4], *time-adaptivity* [12], and *safe convergence* [9].

We consider here the notion of *safe convergence*. The main idea behind this concept is the following: For a large class of problems, it is often hard to design self-stabilizing algorithms that guarantee small stabilization time, even

after few transient faults [6]. Large stabilization time is usually due to strong specifications that a legitimate configuration must satisfy. The goal of a *safely converging self-stabilizing algorithm* is to first quickly converge ($O(1)$ rounds is usually expected) to a *feasible* legitimate configuration, where a minimum quality of service is guaranteed. Once such a feasible legitimate configuration is reached, the system continues to converge to an *optimal* legitimate configuration, where more stringent conditions are required. Safe convergence is especially interesting for self-stabilizing algorithms that compute optimized data structures, *e.g.*, minimal dominating sets [9], approximation of the minimum weakly connected dominating set [10], and approximately minimum connected dominating set [11].

We consider the (f, g)-*alliance problem*. Let $G = (V, E)$ be an undirected graph and f, g two functions mapping nodes to non-negative integers. For every node $p \in V$, $\mathcal{N}_p$ (resp. δ_p) denotes the set of neighbors (resp. the degree) of p in G. A subset of nodes $A \subseteq V$ is an (f, g)-*alliance* of G if and only if

$$(\forall p \in V \setminus A, |\mathcal{N}_p \cap A| \geq f(p)) \wedge (\forall p \in A, |\mathcal{N}_p \cap A| \geq g(p))$$

Moreover, A is *minimal* if and only if no proper subset of A is an (f, g)-alliance of G. The (f, g)-alliance problem is a generalization of several problems that are of interest in distributed computing. Consider any subset S of nodes:

1. S is a (minimal) dominating set if and only if S is a (minimal) $(1, 0)$-alliance;
2. more generally, S is a (minimal) k-domination set if and only if S is a (minimal) $(k, 0)$-alliance;
3. S is a (minimal) k-tuple domination set if and only if S is a (minimal) $(k, k - 1)$-alliance;
4. S is a (minimal) global defensive alliance if and only if S is a (minimal) $(f, 0)$-alliance, such that $\forall p \in V$, $f(p) = \lceil \delta_p/2 \rceil$;
5. S is a (minimal) global offensive alliance if and only if S is a (minimal) $(1, g)$-alliance, such that $\forall p \in V$, $g(p) = \lceil \delta_p/2 \rceil$.

Note that (f, g)-alliances also have applications in the field of population protocols [1], or server allocation in computer networks [8].

Our Contribution. We give a silent self-stabilizing algorithm, $\mathcal{MA}(f, g)$, that computes a minimal (f, g)-alliance in an asynchronous network with unique node IDs, where f and g are integer-valued functions on nodes, such that $f(p) \geq g(p)$ and $\delta_p \geq g(p)$ for all p.[3]

Given two functions f, g mapping nodes to non-negative integers, we say $f \geq g$ if and only if $\forall p \in V, f(p) \geq g(p)$. We remark that the class of minimal (f, g)-alliances with $f \geq g$ generalizes the classes of minimal dominating

[3] We assume that $\delta_p \geq g(p)$ to ensure that an (f, g)-alliance always exists.

sets, k-domination sets, k-tuple domination sets, and global defensive alliance problems. However, minimal global offensive alliances do not belong to this class.

Our algorithm $\mathcal{MA}(f, g)$ is *safely converging* in the sense that starting from any configuration, it first converges to a (not necessarily minimal) (f, g)-alliance in at most four rounds, and then continues to converge to a minimal one in at most $5n + 4$ additional rounds, where n is the size of the network. Our algorithm is written in the shared memory model, and is proven assuming an unfair (distributed) daemon, the weakest daemon of this model. $\mathcal{MA}(f, g)$ uses $O(\log n)$ bits per process, and stabilizes to a terminal (legitimate) configuration in $O(\Delta^3 n)$ steps, where Δ is the degree of the network. Finally, $\mathcal{MA}(f, g)$ does not need any knowledge of any bound on global parameters of the network (such as its size or its diameter).

Related Work. The (f, g)-alliance problem is introduced in [5]. In the same paper, the authors give several distributed algorithms for that problem and its variants, but none of them is self-stabilizing. To the best of our knowledge, this has been the only publication on (f, g)-alliances up to now. However, there have been results on particular instances of (minimal) (f, g)-alliances, *e.g.*, [9, 13–15]. All of these consider arbitrary identified networks; however a safely converging solution is given only in [9]. Srimani and Xu [13] give a self-stabilizing algorithm to compute a minimal global defensive alliance in $O(n^3)$ steps; however, they assume a central daemon. Turau [14] gives a self-stabilizing algorithm to compute a minimal dominating set in $9n$ steps, assuming an unfair (distributed) daemon. Wang *et al* [15] give a self-stabilizing algorithm to compute a minimal k-domination set in $O(n^2)$ steps, assuming a central daemon. A safely converging self-stabilizing algorithm is given in [9] for computing a minimal dominating set. The algorithm first computes a (not necessarily minimal) dominating set in $O(1)$ rounds and then safely stabilizes to a *minimal* dominating set in $O(\mathcal{D})$ rounds, where $\mathcal{D}$ is the diameter of the network. However, they assume a synchronous daemon.

Roadmap. In the next section we describe our model of computation and give some basic definitions. We define our algorithm $\mathcal{MA}(f, g)$ in Section 3. In Section 4, we sketch the correctness of $\mathcal{MA}(f, g)$ and highlight its complexity analysis.[4] We write concluding remarks and perspectives in Section 5.

[4] All detailed proofs are given in a technical report available online at `http://www-verimag.imag.fr/TR/TR-2012-19.pdf`.

2 Preliminaries

Distributed Systems. We consider distributed systems of n processes equipped of *unique* IDs. By an abuse of notation, we identify a process with its ID whenever convenient. Each process p can directly communicate with a subset $\mathcal{N}_p$ of other processes, called its *neighbors*. We assume that if $q \in \mathcal{N}_p$, then $p \in \mathcal{N}_q$. For every process p, $\delta_p = |\mathcal{N}_p|$ is the *degree of* p. We assume that $\delta_p \geq g(p)$ for every process p. Let $\Delta = \max_{p \in V} \delta_p$ be the degree of the network. The topology of the system is a simple undirected graph $G = (V, E)$, where V is the set of processes and E is a set of edges representing (direct) communication relations.

Computational Model. We assume the *shared memory model* of computation introduced by Dijkstra [2], where each process communicates with its neighbors using a finite set of *locally shared variables*, henceforth called simply *variables*. Each process can read its own variables and those of its neighbors, but can write only to its own variables. Each process operates according to its (local) *program*. We define a *(distributed) algorithm* to be a collection of n *programs*, each operating on a single process. The program of each process is a finite ordered set of actions, where the ordering defines *priority*. This priority is the order of appearance of actions in the text of the program. A process p is not enabled to execute any (lower priority) action if it is enabled to execute an action of higher priority. Let $\mathcal{A}$ be a distributed algorithm, consisting of a local program $\mathcal{A}(p)$ for each process p. Each action in $\mathcal{A}(p)$ is of the following form: $\langle$*label*$\rangle$ :: $\langle guard \rangle \rightarrow \langle$*statement*$\rangle$. *Labels* are only used to identify actions. The *guard* of an action in $\mathcal{A}(p)$ is a Boolean expression involving the variables of p and its neighbors. The *statement* of an action in $\mathcal{A}(p)$ updates some variables of p. The *state* of a process in $\mathcal{A}$ is defined by the values of its variables in $\mathcal{A}$. A *configuration* of $\mathcal{A}$ is an instance of the states of processes in $\mathcal{A}$. $\mathcal{C}_{\mathcal{A}}$ is the set of all possible configurations of $\mathcal{A}$. (When there is no ambiguity, we omit the subscript $\mathcal{A}$.) An action can be executed only if its guard evaluates to *true*; in this case, the action is said to be *enabled*. A process is said to be enabled if at least one of its actions is enabled. We denote by *Enabled*(γ) the subset of processes that are enabled in configuration γ. When the configuration is γ and *Enabled*$(\gamma) \neq \emptyset$, a *daemon*[5] (scheduler) selects a non-empty set $\mathcal{X} \subseteq$ *Enabled*(γ); then every process of $\mathcal{X}$ *atomically* executes its highest priority enabled action, leading to a new configuration γ', and so on. The transition from γ to γ' is called a *step* (of $\mathcal{A}$). The possible steps induce a binary relation over configurations of $\mathcal{A}$, denoted by $\mapsto$. An *execution* of $\mathcal{A}$ is a maximal

[5] The daemon realizes the asynchrony of the system.

sequence of its configurations $e = \gamma_0\gamma_1 \ldots \gamma_i \ldots$ such that $\gamma_{i-1} \mapsto \gamma_i$ for all $i > 0$. The term "maximal" means that the execution is either infinite, or ends at a *terminal* configuration in which no action of $\mathcal{A}$ is enabled at any process. As we saw previously, each step from a configuration to another is driven by a daemon. In this paper we assume the daemon is *unfair*; *i.e.*, the daemon might never permit an enabled process to execute unless it is the only enabled process.

We say that a process p is *neutralized* in the step $\gamma_i \mapsto \gamma_{i+1}$ if p is enabled in γ_i and not enabled in γ_{i+1}, but does not execute any action between these two configurations. Neutralization of a process can be caused by the following situation: at least one neighbor of p changes its state between γ_i and γ_{i+1}, and this change makes the guards of all actions of p false. To evaluate time complexity, we use the notion of *round*. The first round of an execution e, noted e', is the minimal prefix of e in which every process that is enabled in the initial configuration either executes an action or becomes neutralized. Let e'' be the suffix of e starting from the last configuration of e'. The second round of e is the first round of e'', and so forth.

Self-Stabilization, Silence, and Safe Convergence. Let $\mathcal{A}$ be a distributed algorithm. Let P be a predicate over $\mathcal{C}$. $\mathcal{A}$ is *self-stabilizing w.r.t.* P if and only if there exists a non-empty subset $\mathcal{S}_P$ of $\mathcal{C}$ such that:

1. $\forall \gamma \in \mathcal{S}_P$, $P(\gamma)$ (*Correction*);
2. for each possible step $\gamma \mapsto \gamma'$ of $\mathcal{A}$, $\gamma \in \mathcal{S}_P \Rightarrow \gamma' \in \mathcal{S}_P$ (*Closure*);
3. each execution of $\mathcal{A}$ (starting from an arbitrary configuration) contains a configuration of $\mathcal{S}_P$ (*Convergence*).

The configurations of $\mathcal{S}_P$ are said to be *legitimate*, and other configurations are called *illegitimate*.

$\mathcal{A}$ is *silent* if all its executions are finite [3]. To show that $\mathcal{A}$ is silent and self-stabilizing *w.r.t.* P, it is sufficient to show that (1) all executions of $\mathcal{A}$ are finite and (2) all terminal configurations of $\mathcal{A}$ satisfy P.

Let P_1 and P_2 be two predicates over $\mathcal{C}$ such that $\forall \gamma \in \mathcal{C}$, $P_2(\gamma) \Rightarrow P_1(\gamma)$. $\mathcal{A}$ is *safely converging self-stabilizing w.r.t.* (P_1, P_2) if and only if the following three properties hold:

1. $\mathcal{A}$ is *self-stabilizing w.r.t.* P_1;
2. $\mathcal{A}$ is *self-stabilizing w.r.t.* P_2; and
3. every execution of $\mathcal{A}$ starting from a configuration of $\mathcal{S}_{P_1}$ eventually reaches a configuration of $\mathcal{S}_{P_2}$, where $\mathcal{S}_{P_1}$ and $\mathcal{S}_{P_2}$ are respectively the sets of legitimate configurations for P_1 and P_2 (*Safe Convergence*).

The configurations of $\mathcal{S}_{P_1}$ are said to be *feasible legitimate*. The configurations of $\mathcal{S}_{P_2}$ are said to be *optimal legitimate*.

Assume that $\mathcal{A}$ is *safely converging self-stabilizing w.r.t.* (P_1, P_2). The *first convergence time* is the maximum time to reach a feasible legitimate config-

uration, starting from any configuration. The *second convergence time* is the maximum time to reach an optimal legitimate configuration, starting from any feasible legitimate configuration. The *stabilization time* is the sum of the first and second convergence times.

Minimality and 1-Minimality of (f, g)-alliances. We recall that an (f, g)-alliance A of a graph G is *minimal* if and only if no proper subset of A is an (f, g)-alliance. Then, A is 1-*minimal* if and only if $\forall p \in A$, $A \setminus \{p\}$ is not an (f, g)-alliance. Surprisingly, a 1-*minimal* (f, g)-alliance is not necessarily a *minimal* (f, g)-alliance, [5]. However, we have the following property:

Property 1. [5] Given two functions f and g mapping nodes to non-negative integers, we have:

1. Every minimal (f, g)-alliance is a 1-minimal (f, g)-alliance, and
2. if $f \geq g$, every 1-minimal (f, g)-alliance is a minimal (f, g)-alliance.

3 The Algorithm

The formal code of $\mathcal{MA}(f, g)$ is given as Algorithm 1. Given input functions f and g, $\mathcal{MA}(f, g)$ computes a single Boolean variable $p.inA$ for each process p. For any configuration γ, let $A_\gamma = \{p \in V : p.inA\}$. (We omit the subscript γ when it is clear from the context.) If γ is terminal, then A_γ is a 1-*minimal* (f, g)-*alliance*, and consequently, if $f \geq g$, A_γ is a *minimal* (f, g)-*alliance*.

During an execution, a process may need to leave or join A. The basic idea of safe convergence is that it should be more difficult for a process to leave A than to join it. This permits quick recovery to a configuration in which A is an (f, g)-alliance, but not necessarily a minimal one.

3.1 Leaving A

Action `Leave` allows a process to leave A. To obtain 1-minimality, we allow a process p to leave A if

Requirement 1: p will have enough neighbors in A (*i.e.*, at least $f(p)$) once it has left, and

Requirement 2: each $q \in \mathcal{N}_p$ will still have enough neighbors in A (*i.e.*, at least $g(q)$ or $f(q)$, depending on whether q is in A) once p has been deleted from A.

Ensuring Requirement 1. To maintain Requirement 1, we implement our algorithm in such a way that deletion from A is *locally sequential*, *i.e.*, during a step, at most one process can leave A in the neighborhood of each process p (including p itself). Using this locally sequential mechanism, if a process p wants

to leave A, it must first verify that $\texttt{NbA}(p) = |\{q \in \mathcal{N}_p \ : \ q.inA\}|$ is greater or equal to $f(p)$ before leaving A. Hence, if p actually leaves A, it is the only one in its neighborhood allowed to do so; consequently, Requirement 1 still holds once p has left A.

The locally sequential mechanism is implemented using a neighbor pointer $p.choice$ at each process p, which takes value in $\mathcal{N}_p \cup \{\perp\}$; $p.choice = q \in \mathcal{N}_p$ means that p authorizes q to leave A, while $p.choice = \perp$ means that p does not authorize any neighbor to leave A. The value of $p.choice$ is maintained using Action `Vote`, which will be defined later.

To leave A, a process p should not authorize any neighbor to leave A ($p.choice = \perp$) and should be authorized to leave by all of its neighbors ($\forall q \in \mathcal{N}_p,\ q.choice = p$). For example, consider the $(1,0)$-alliance in Figure 1. Only Process 2 is able to leave A. Process 2 can leave A because it has enough neighbors in A (*i.e.*, 2 neighbors, while $f(2) = 1$); if Process 2 leaves A, it will still have two neighbors in A, and Requirement 1 will not be violated.

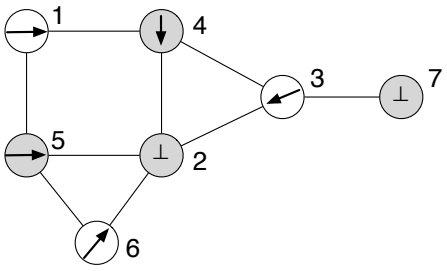

Fig. 1: Neighbor pointers when computing a minimal $(1,0)$-alliance. Numbers indicate IDs. A is the set of gray nodes. The value of *choice* is represented by an arrow or a tag "$\perp$" inside the node.

Ensuring Requirement 2. Requirement is also maintained by the fact that a process p must have authorization from each of its neighbors to leave A. A neighbor q can give such an authorization to p only if q still has enough neighbors in A without p. For a process q to authorize a neighbor p to leave A, p must currently be in A, *i.e.*, $p.inA = true$, and q must have more neighbors than necessary in A, *i.e.*, the predicate $\texttt{HasExtra}(q)$ should be true, meaning that $\mathcal{N}_q \cap A$ has more than $g(q)$, respectively $f(q)$, members if q is in A, respectively not in A. For example, consider the $(1,0)$-alliance in Figure 1. Processes 4 and 5 can designate Process 2 because they belong to A and $g(4) = g(5) = 0$. Moreover, Processes 3 and 6 can designate Process 2 because they do not belong to A and $f(3) = f(6) = 1$: if Process 2 leaves A, Process 3 (resp. Process 6) still has one neighbor in A, which is Process 7 (resp. Process 5).

Busy Processes. It is possible that a neighbor p of q cannot leave A — in this case p is said to be *busy* — because one of these two conditions is *true*:

(i) $\texttt{NbA}(p) < f(p)$**:** in this case, p does not have enough neighbors in A to be allowed to leave A.

(ii) $\neg\texttt{IsExtra}(p)$**:** in this case, at least one neighbor of p needs p to stay in A.

If q chooses such a neighbor p, this may lead to a deadlock. We use the Boolean variable $p.busy$ to inform q that one of the two aforementioned conditions holds for p. Action `Flag` maintains $p.busy$. So, to prevent deadlock, q must not choose any neighbor p for which $p.busy = true$.

A process p evaluates Condition (i) by reading the variables *inA* of all its neighbors. On the other hand, evaluation of Condition (ii) requires that p knows, for each of its neighbors, both its status (*inA*) and the number of its own neighbors that are in A. This latter information is obtained using an additional variable, *nbA*, where each process maintains, using Action `Count`, the number of its neighbors that are in A.

In Figure 2, consider the $(2, 0)$-alliance. Process 5 is busy because of Condition (i): it has only one neighbor in A, while $f(5) = 2$. Process 2 is busy because of Condition (ii): its neighbor 1 is not in A, $f(1) = 2$, and has only two neighbors in A, so it cannot authorize any of its neighbors to leave. Consequently, Process 1 cannot designate any neighbor (all its neighbors in A are busy); while Process 3 should not designate Process 2.

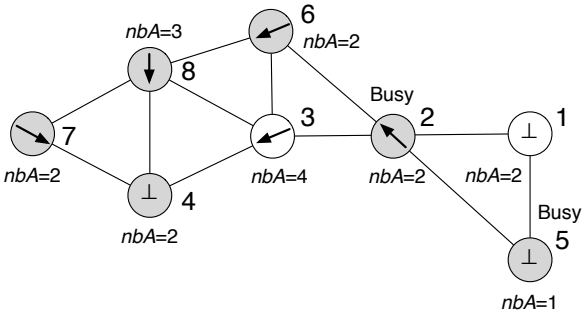

Fig. 2: Busy processes when computing a minimal $(2, 0)$-alliance. Values of *nbA* are also given.

Action `Vote`**.** Hence, the value of $p.choice$ is chosen, using Action `Vote`, as follows:

1. $p.choice$ is set to $\perp$ if the condition $\texttt{Cand}(p) \neq \emptyset \wedge \texttt{HasExtra}(p) \wedge (\texttt{IamCand}(p) \Rightarrow \texttt{MinCand}(p) < p)$ in Macro $\texttt{ChosenCand}(p)$ is *false*, *i.e.*, if one of the following conditions holds:
   - $\texttt{Cand}(p) = \emptyset$, which means that no neighbor of p can leave A.
   - $\texttt{HasExtra}(p) =$ *false*, which means that p cannot authorize any neighbor to leave A.
   - $\texttt{IamCand}(p) \wedge p < \texttt{MinCand}(p)$, which means that p is also candidate to leave A and has higher priority to leave A than any other candidate in its neighborhood. (Remember that to be allowed to leave A, p should, in particular, satisfy $p.choice = \perp$.)

     The aforementioned priorities are based on process IDs, *i.e.*, for every two process u and v, u has higher priority than v if and only if the ID of u is smaller than the ID of v.
2. Otherwise, p uses $p.choice$ to designate a neighbor that is in A, and not busy, in order to authorize it to leave A. If p has several possible candidates among

its neighbors, it selects the one of highest priority (*i.e.*, of smallest ID). For example, if we consider the $(2, 0)$-alliance in Figure 2, then we can see that Process 3 designates Process 4 because it is its smallest neighbor that is both in A and not busy.

There is one last problem: A process q may change its pointer while simultaneously one of its neighbors p leaves A, and consequently Requirement 2 may be violated. Indeed, q chooses new candidate assuming that p remains in A. This may happen only if the previous value of $q.choice$ was p. To avoid this situation, we do not allow q to directly change $q.choice$ from one neighbor to another. Each time q wants to change its pointer, if $q.choice \in \mathcal{N}_q$, q first resets $q.choice$ to $\perp$; see $\texttt{Choice}(q)$.

Figures 3 and 4 illustrate this last issue in the case of a $(1, 0)$-alliance. In the step from Configuration (a) to Configuration (b) of Figure 3, Process 2 directly changes its pointer from 3 to 1. Simultaneously, 3 leaves A. So, Process 2 authorizes Process 1 to leave A, while it should not do so. After that, Process 1 is authorized to leave A and does so at the step from Configuration (b) to Configuration (c), and thus Requirement 2 is violated. Figure 4 illustrates how we solve the problem. In Configuration (b), Process 3 has left, but the pointer of Process 2 is equal to $\perp$. So, Process 1 cannot leave yet, and Process 2 will not authorize it to leave.

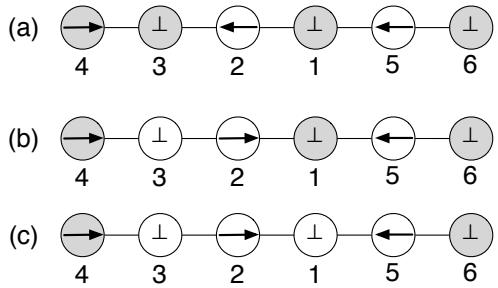

Fig. 3: Requirement 2 violation when computing a minimal $(1, 0)$-alliance. (We only show the values that are needed in the discussion.)

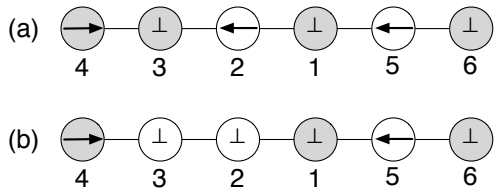

Fig. 4: The reset of the neighbor pointer is applied to the example of Figure 3.

Algorithm 1 $\mathcal{MA}(f, g)$, code for each process p

Variables: $p.inA$, $p.busy$: Booleans; $p.choice \in \mathcal{N}_p \cup \{\perp\}$; $p.nbA \in [0..\delta_p]$

Macros:

$\texttt{NbA}(p) = |\{q \in \mathcal{N}_p, q.inA\}|$

$\texttt{Cand}(p) = \{q \in \mathcal{N}_p, q.inA \wedge \neg q.busy\}$

$\texttt{MinCand}(p) = \min(\texttt{Cand}(p) \cup \{\infty\})$

$\texttt{ChosenCand}(p) =$ **if** $\texttt{Cand}(p) \neq \emptyset \wedge \texttt{HasExtra}(p) \wedge (\texttt{IamCand}(p) \Rightarrow \texttt{MinCand}(p) < p)$
then $\texttt{MinCand}(p)$
else $\perp$

$\texttt{Choice}(p) =$ **if** $p.choice = \perp$ **then** $\texttt{ChosenCand}(p)$ **else** $\perp$

Predicates:

$\texttt{IsMissing}(p) \equiv \exists q \in \mathcal{N}_p, (\neg q.inA \wedge q.nbA < f(q)) \vee (q.inA \wedge q.nbA < g(q))$

$\texttt{IsExtra}(p) \equiv \forall q \in \mathcal{N}_p, (\neg q.inA \Rightarrow q.nbA > f(q)) \wedge (q.inA \Rightarrow q.nbA > g(q))$

$\texttt{HasExtra}(p) \equiv (\neg p.inA \Rightarrow \texttt{NbA}(p) > f(p)) \wedge (p.inA \Rightarrow \texttt{NbA}(p) > g(p))$

$\texttt{IsBusy}(p) \equiv \texttt{NbA}(p) < f(p) \vee \neg \texttt{IsExtra}(p)$

$\texttt{IamCand}(p) \equiv p.inA \wedge \neg \texttt{IsBusy}(p)$

$\texttt{MustJoin}(p) \equiv \neg p.inA \wedge (\texttt{NbA}(p) < f(p) \vee \texttt{IsMissing}(p)) \wedge (\forall q \in \mathcal{N}_p, q.choice \neq p)$

$\texttt{CanLeave}(p) \equiv p.inA \wedge \texttt{NbA}(p) \geq f(p) \wedge (\forall q \in \mathcal{N}_p, q.choice = p) \wedge p.choice = \perp$

Actions:

`Join` :: $\texttt{MustJoin}(p)$ $\rightarrow$ $p.inA \leftarrow true$; $p.choice \leftarrow \perp$; $p.nbA \leftarrow \texttt{NbA}(p)$

`Vote` :: $p.choice \neq \texttt{ChosenCand}(p)$ $\rightarrow$ $p.choice \leftarrow \texttt{Choice}(p)$; $p.nbA \leftarrow \texttt{NbA}(p)$; $p.busy \leftarrow \texttt{IsBusy}(p)$

`Count` :: $p.nbA \neq \texttt{NbA}(p)$ $\rightarrow$ $p.nbA \leftarrow \texttt{NbA}(p)$

`Flag` :: $p.busy \neq \texttt{IsBusy}(p)$ $\rightarrow$ $p.busy \leftarrow \texttt{IsBusy}(p)$

`Leave` :: $\texttt{CanLeave}(p)$ $\rightarrow$ $p.inA \leftarrow false$

3.2 Joining A

Action `Join` allows a process to join A. A process p not in A must join A if:
(1) p does not have enough neighbors in A ($\texttt{NbA}(p) < f(p)$), or
(2) a neighbor of p needs p to join A ($\texttt{IsMissing}(p)$).

Moreover, to prevent p from cycling in and out of A, we require that every neighbor of p stop designating it (with their *choice* pointer) before p can join A (again). Note that all neighbors of p stop designating p immediately after it leaves A; see Action `Vote`. (Actually, this introduces a delay of only one round.)

A process evaluates condition (1) by reading the variables *inA* of all its neighbors. To evaluate condition (2), it needs to know, for each neighbor q, both its status *w.r.t.* A ($q.inA$) and the number of its neighbors that are in A ($q.nbA$).

4 Correctness

Recall that, for any configuration γ, we define the set $A = A_\gamma = \{p \in V : p.inA\}$. Moreover, throughout this section, we assume that $f \geq g$.

Termination. We first show that the unfair daemon cannot prevent $\mathcal{MA}(f, g)$ from reaching a terminal configuration, regardless the initial configuration. The proof consists of proving that the number of steps to reach a terminal configuration, starting from an arbitrary configuration, is bounded, regardless of the choices of the daemon.

The core of the proof consists of showing that each process can execute `Join` at most once. To join A, a process p should be not be pointed to by any neighbor. If later p leaves A, (1) it should satisfy $p.choice = \perp$ and (2) all its neighbors should have executed `Vote` so as to point to p with their variable *choice*, meaning that none of them needs p to stay in A. Hence, after leaving A, p will always have enough neighbors in A since by (1), Requirement 1 cannot be violated. Moreover, all its neighbors will always have enough neighbors in A since by (2), Requirement 2 is never violated. Hence, from that time forward, no process in the neighborhood of p (including p itself) will ever need p to rejoin A.

We can directly deduce that each process can execute `Leave` at most twice. Using similar reasoning, the number of times a process executes `Count` is bounded by one plus the number of times each of its neighbors joins or leaves A. We use the same approach for all other actions. Overall, we find that the maximum number of actions executed by all processes — and consequently the maximum number of steps to reach a terminal configuration — is bounded as follows:

Lemma 1. *Starting from any configuration, $\mathcal{MA}(f, g)$ reaches a terminal configuration in $O(n \times \Delta^3)$ steps.*

Partial Correctness. Let γ be any terminal configuration. To complete the proof, we have to show that the specification $\mathcal{MA}(f, g)$ is achieved at γ, *i.e.*, γ satisfies the following predicate:

$$SP_{Minimal} \stackrel{\texttt{def}}{=} A \text{ is a minimal } (f, g)\text{-alliance}$$

To prove this, we first show that A is an (f, g)-alliance in γ (Lemma 3); then we show that A is also minimal in γ (Lemma 4). To show these two results, we use two intermediate claims: Lemma 2 and Corollary 1. The former states that every process of A is busy in γ, meaning that either p does not have enough neighbors in A to leave A, or that at least one neighbor of p requires that p stay

in A, *i.e.,* A is 1-minimal. The latter is a simple corollary of Lemma 2 and states that no process authorizes a neighbor to leave A at γ.

At γ, Action `Count` is disabled at every process, thus:

Remark 1. At γ, for every process p, $p.nbA = \texttt{NbA}(p) = |\{q \in \mathcal{N}_p \,:\, q.inA\}|$.

Lemma 2. *At γ, for every process p,* $p.\text{inA} \Rightarrow p.\text{busy}$.

Proof Outline. By contradiction. Assume that there is at least one process p such that $p.inA = true$ and $p.busy = false$ at γ. Then, for each such process p, we have $\texttt{IsBusy}(p) = false$ at γ, because Action `Flag` is disabled at every process. The remainder of the proof consists of showing that the process $p_{\min}$ having the smallest ID among all p satisfies $\texttt{CanLeave}(p_{\min})$ at γ, and consequently is enabled to leave A, contradiction. Informally, if we made such an assumption, then $p_{\min}$ would be the smallest candidate to leave A, and consequently it would be designated by the pointer *choice* of all its neighbors and $p_{\min}.choice = \perp$. □

By Lemma 2, for every process p, $\texttt{Cand}(p) = \emptyset$ at γ. Thus $\texttt{ChosenCand}(p) = \perp$ at γ, and from the negation of the guard of Action `Vote`, we have:

Corollary 1. *At γ, for every process p,* $p.\text{choice} = \perp$.

For every process p, let

$$\texttt{Fga}(p) \stackrel{\texttt{def}}{=} (\neg p.inA \Rightarrow \texttt{NbA}(p) \geq f(p)) \wedge (p.inA \Rightarrow \texttt{NbA}(p) \geq g(p))$$

When a process p satisfies $\texttt{Fga}(p)$, it is locally correct, *i.e.*, it has enough neighbors in A, according to its status. So, by definition we have:

Remark 2. A is an (f, g)-alliance if and only if $\texttt{Fga}(p)$ holds for all $p \in V$.

Lemma 3. *At γ, A is an (f, g)-alliance.*

Proof Outline. By Remark 2, we merely need show that every process p satisfies $\texttt{Fga}(p)$ in γ. Consider the following two cases:

$p \notin A$ in γ: First, $p.inA = false$ at γ. Then, γ being terminal, $\neg\texttt{MustJoin}(p)$ holds in γ. Now, by Corollary 1, since $p.inA = false$, $\texttt{NbA}(p) \geq f(p)$ holds at γ. Consequently $\texttt{Fga}(p)$ holds at γ.

$p \in A$ at γ: First, $p.inA = true$ at γ. Assume that $\texttt{Fga}(p) = false$ at γ. Since $\delta_p \geq g(p)$, we can show that $\exists q \in \mathcal{N}_p$ such that $\neg q.inA \wedge \texttt{IsMissing}(q) = true$ at γ. Then, by Corollary 1, we can conclude that q is enabled to join A at γ, contradiction.

□

Lemma 4. *At γ, A is a minimal (f, g)-alliance.*

Proof Outline. We already know that at γ, A defines an (f,g)-alliance. Moreover, by Property 1, if A is 1-minimal and $f \geq g$, then A is a minimal (f,g)-alliance. Thus, we only need to show the 1-minimality of A. Assume that A is not 1-minimal. Then, $\exists p \in A$ such that $A - \{p\}$ is an (f,g)-alliance. So:

1. $|A \cap \mathcal{N}_p| \geq f(p)$,
2. $\forall q \in \mathcal{N}_p, q \in A \Rightarrow |A \cap \mathcal{N}_q - \{p\}| \geq g(q)$, and
3. $\forall q \in \mathcal{N}_p, q \notin A \Rightarrow |A \cap \mathcal{N}_q - \{p\}| \geq f(q)$.

From the negation of the guard in the local program of p, we can show that 1, 2, and 3 imply that $p.busy = false$ at γ. As we assume that $p.inA = true$ at γ ($p \in A$), this contradicts Lemma 2. □

By Lemmas 1-4, we can conclude:

Theorem 1. *$\mathcal{MA}(f,g)$ is silent and self-stabilizing* w.r.t. *$SP_{Minimal}$, and its stabilization time is $O(\Delta^3 n)$ steps.*

Complexity Analysis and Safe Convergence in Rounds. First, we recall the notion of a *closed predicate*: let P be a predicate over configuration of $\mathcal{MA}(f,g)$; P is *closed* if and only if $\forall \gamma, \gamma' \in \mathcal{C}$, $P(\gamma) \wedge \gamma \mapsto \gamma' \Rightarrow P(\gamma')$.

To establish safe convergence of $\mathcal{MA}(f,g)$, we have showed that it gradually converges to more and more specific closed predicates until reaching a terminal configuration. The gradual convergence to those specific closed predicates is given in Figure 5.

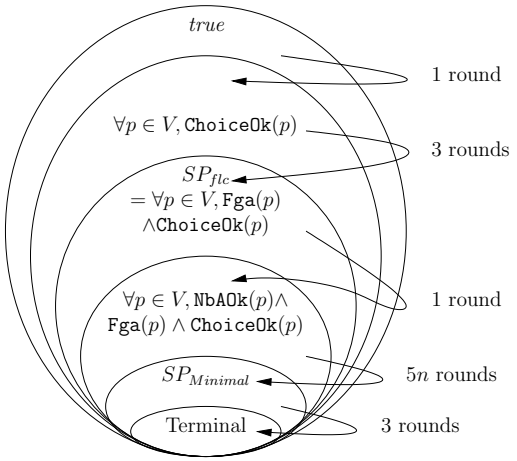

Fig. 5: Safe Convergence of $\mathcal{MA}(f,g)$

We have first showed that, regardless the initial configuration, in at most one round the system reaches a configuration from which the predicate $\forall p \in V$, ChoiceOk(p) is forever *true*, where for every process p:

$$\texttt{ChoiceOk}(p) \stackrel{\texttt{def}}{=} (p.choice \neq \perp \wedge p.choice.inA) \Rightarrow \texttt{HasExtra}(p)$$

Once ChoiceOk(p) holds at p, no neighbor of p can make p locally incorrect by leaving A.

Then, we have showed that, at most three rounds later, the system is forever in a *feasible legitimate configuration*, which is defined as any configuration that

satisfies

$$SP_{flc} \stackrel{\texttt{def}}{=} \forall p \in V,\ \texttt{ChoiceOk}(p) \wedge \texttt{Fga}(p)$$

Note that, as expected, A defines an (f,g)-alliance in any feasible legitimate configuration (Remark 2).

Then, we have showed that after one more round, the system is in a configuration from which the predicate $\forall p \in V, \texttt{ChoiceOk}(p) \wedge \texttt{Fga}(p) \wedge \texttt{NbAOk}(p)$ is forever *true*, where for every process p,

$$\texttt{NbAOk}(p) \stackrel{\texttt{def}}{=} (\neg p.\mathit{inA} \Rightarrow p.\mathit{nbA} \geq f(p)) \wedge (p.\mathit{inA} \Rightarrow p.\mathit{nbA} \geq g(p))$$

When $\texttt{Fga}(p) \wedge \texttt{NbAOk}(p)$ is *true* at p, this means that p is locally correct and the variable $p.\mathit{nbA}$ gives this information to the neighbors of p.

From that point, we have showed that while A is not a minimal (f,g)-Alliance, at least one process definitely leaves A within the next five rounds. Hence, in most $5n$ additional rounds, the predicate $SP_{Minimal}$ is forever true, meaning that A is a minimal (f,g)-Alliance.

Finally, we have showed that three additional rounds are necessary to reach a terminal configuration. Hence, we can conclude:

Theorem 2. *$\mathcal{MA}(f,g)$ is silent and safely converging self-stabilizing* w.r.t. *$(SP_{flc}\ ,\ SP_{Minimal})$, its first convergence time is at most four rounds, its second convergence time is at most $5n+4$ rounds, and its stabilization time is at most $5n+8$ rounds.*

5 Conclusion and Perspectives

We have given a silent self-stabilizing algorithm, $\mathcal{MA}(f,g)$, that computes a minimal (f,g)-alliance in an asynchronous network with unique node IDs, assuming that $f \geq g$ and every process p has a degree at least $g(p)$. $\mathcal{MA}(f,g)$ is also *safely converging*: It first converges to a (not necessarily minimal) (f,g)-alliance in at most four rounds and then continues to converge to a minimal one in at most $5n+4$ additional rounds. We have verified correctness and time complexity of $\mathcal{MA}(f,g)$, assuming the weakest scheduling assumption: the distributed unfair daemon. Its memory requirement is $O(\log n)$ bits per process and its stabilization time in steps is $O(\Delta^3 n)$.

One possible extension of our work is to explore the possibility of reducing the stabilization time to $O(\mathcal{D})$ rounds. It would be interesting to study the (f,g)-alliance problem without the constraint that $f \geq g$. We conjecture that $\mathcal{MA}(f,g)$ is still self-stabilizing in that case. However, we already know that it does not guarantee a good safe convergence property in the case $f < g$: Indeed, in that case, any process can join A several times, giving us a round complexity

of $\Omega(n)$ for convergence to a feasible legitimate configuration. We believe that when $f < g$, it is impossible to guarantee $O(1)$ round convergence to a feasible legitimate configuration, where a (not necessarily minimal) (f, g)-alliance is defined.

References

1. Angluin, D., Aspnes, J., Eisenstat, D., Ruppert, E.: The computational power of population protocols. Distributed Computing 20(4), 279–304 (2007)
2. Dijkstra, E.W.: Self-Stabilizing Systems in Spite of Distributed Control. Commun. ACM 17, 643–644 (1974)
3. Dolev, S., Gouda, M.G., Schneider, M.: Memory Requirements for Silent Stabilization. In: PODC. pp. 27–34 (1996)
4. Dolev, S., Herman, T.: Superstabilizing protocols for dynamic distributed systems. Chicago J. Theor. Comput. Sci. 1997 (1997)
5. Dourado, M.C., Penso, L.D., Rautenbach, D., Szwarcfiter, J.L.: The south zone: Distributed algorithms for alliances. In: SSS. pp. 178–192 (2011)
6. Genolini, C., Tixeuil, S.: A lower bound on dynamic k-stabilization in asynchronous systems. In: 21st Symposium on Reliable Distributed Systems (SRDS 2002), 13-16 October 2002, Osaka, Japan. pp. 212–. IEEE Computer Society (2002)
7. Ghosh, S., Gupta, A., Herman, T., Pemmaraju, S.V.: Fault-containing self-stabilizing algorithms. In: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996. pp. 45–54. ACM (1996)
8. Gupta, A., Maggs, B.M., Oprea, F., Reiter, M.K.: Quorum placement in networks to minimize access delays. In: Aguilera, M.K., Aspnes, J. (eds.) Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing, PODC 2005, Las Vegas, NV, USA, July 17-20, 2005. pp. 87–96. ACM (2005)
9. Kakugawa, H., Masuzawa, T.: A self-stabilizing minimal dominating set algorithm with safe convergence. In: IPDPS (2006)
10. Kamei, S., Kakugawa, H.: A self-stabilizing approximation algorithm for the minimum weakly connected dominating set with safe convergence. In: Proceedings of the First International Workshop on Reliability, Availability, and Security (WRAS). pp. 57–67. Paris, France (September 2007)
11. Kamei, S., Kakugawa, H.: A self-stabilizing 6-approximation for the minimum connected dominating set with safe convergence in unit disk graphs. Theoretical Computer Science 428, 80–90 (2012)
12. Kutten, S., Patt-Shamir, B.: Time-adaptive self stabilization. In: Burns, J.E., Attiya, H. (eds.) Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing, Santa Barbara, California, USA, August 21-24, 1997. pp. 149–158. ACM (1997)
13. Srimani, P.K., Xu, Z.: Distributed protocols for defensive and offensive alliances in network graphs using self-stabilization. In: ICCTA. pp. 27–31 (2007)
14. Turau, V.: Linear self-stabilizing algorithms for the independent and dominating set problems using an unfair distributed scheduler. Inf. Process. Lett. 103(3), 88–93 (2007)
15. Wang, G., Wang, H., Tao, X., Zhang, J.: A self-stabilizing algorithm for finding a minimal k-dominating set in general networks. In: Xiang, Y., Pathan, M., Tao, X., Wang, H. (eds.) Data and Knowledge Engineering, pp. 74–85. Lecture Notes in Computer Science, Springer Berlin Heidelberg (2012)