Information Processing Letters 147 (2019) 38–43

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

Self-stabilizing algorithm for two disjoint minimal dominating sets

Pradip K. Srimani *, James Z. Wang

School of Computing, Clemson University, Clemson, SC 29634, USA

ARTICLE INFO

Article history:
Received 19 July 2018
Received in revised form 4 March 2019
Accepted 11 March 2019
Available online 15 March 2019
Communicated by Krishnendu Chatterjee

Keywords:
Self-stabilizing algorithm
Minimal dominating set
Mutually disjoint minimal dominating sets
Graph algorithms

ABSTRACT

We propose a new self stabilizing algorithm to compute two mutually disjoint minimal dominating sets in an arbitrary graph G with no isolates (this is always possible due to famous Ore's theorem in [1] that says "In a graph having no isolated nodes, the complement of any minimal dominating set is a dominating set"). We use an unfair central daemon and unique ids of the nodes. The time complexity of our algorithm is $O(n^3)$, an improvement by a factor of n from that of [2], that uses same assumptions to design an $O(n^4)$ algorithm, where n is the number of nodes in G. The algorithm uses the concept of running two copies of an algorithm in an interleaving manner such that the state spaces of the two copies are always kept mutually disjoint. We expect our approach will prove useful in designing algorithms for other mutually disjoint predicates in a network graph.

1. Introduction

The paradigm of self stabilization, as a tool for designing fault tolerant localized distributed algorithms for networks, was first proposed by Dijkstra in 1974 [3]. It is an *optimistic* way of looking at system fault tolerance and scalable coordination, because it provides a built-in safeguard against transient failures that might corrupt the data in a distributed system. An algorithm is self-stabilizing iff it reaches some legitimate state starting from an arbitrary state. The promise of self-stabilization, as opposed to fault masking, is to recover from failure in a reasonable amount of time and without intervention by any external agency. Since the faults are transient (eventual repair is assumed), it is no longer necessary to assume a bound on the number of failures. The participating nodes communicate only with their immediate neighbors and require minimal storage to keep the local knowledge and yet a desired global objective is achieved. Since no communication is needed beyond a node's immediate neighborhood, communication overhead scales well with increase or decrease in the network size.

Self-stabilization: In a self-stabilizing algorithm, each node maintains a set of local variables; the set of these variables constitutes the local state of the node. The union of the local states of all nodes constitutes the global system state. A self-stabilizing algorithm is usually specified as a set of rules at each node; each rule consists of a *condition* and an *action* and is written as "**if** condition **then** action". A condition is a boolean predicate involving the local states of the node and those of its immediate neighbors. A node, at any step of execution, is called *privileged* iff at least one condition is true. The daemon (runtime scheduler) selects node(s) from among the privileged nodes to take an action (also called *move*) at each step. The *central* daemon selects exactly one privileged node to move at each step; the *distributed* daemon selects a non-empty subset of the privileged nodes to move at each step; the *synchronous* daemon selects all the privileged nodes to move at each step. The daemon is called *fair* if it prevents a node being continuously privileged without performing local actions. Otherwise, the daemon is called *unfair*.

* Corresponding author.
E-mail address: srimani@clemson.edu (P.K. Srimani).

https://doi.org/10.1016/j.ipl.2019.03.007
0020-0190/

Distributed systems and graph algorithms: Many essential services in large scale networked distributed systems (ad hoc, wireless or sensor) involve maintaining a global predicate over the entire network. Each participating node has limited resources (computing, storage, energy) while the networked system needs to achieve a global task. The global task is usually specified by some invariance relation on the global state of the network comprised of the local states of all participating nodes. In addition, individual nodes cannot keep track of the global network state due to limited communication capability and storage. Graph algorithms play important roles in networked and distributed systems. Self stabilizing algorithms for various graph domination problems and their variations like connected dominating sets and weakly connected dominating sets have been widely used for clustering in mobile ad hoc networks [4,5]. Most of the self-stabilizing distributed algorithms for graph predicates related to dominance and independence have considered computation of a single subset of the graph with the desired global property [6]. A big majority of them have assumed an unfair serial (central) daemon (scheduler). The concept of dominating set has been used in clustering networks by choosing the dominating set members of the graph to be the cluster heads for the other nodes it dominates. Such sets have also been used for resource placement centers in network graphs in a similar way [5]. Another important issue in computer network design is effective utilization of available edge bandwidths in the network over all edges of the network graph. Consider placement of resource allocation centers where there are multiple types of resources that cannot be transmitted in parallel along the same network edge. If a single dominating set is used, some of the network edges will be overburdened while many others will be idle; two disjoint dominating sets would more evenly distribute the bandwidth requirement over a larger number of edges; two dominating sets are used for two different resources. Two disjoint dominating sets would be very useful when it is inconvenient to place resources of multiple types at the same node. There are already much theoretical research on existence of multiple disjoint predicates in graphs [7–9] including disjoint dominating sets, disjoint dominating and paired-dominating sets and others. No fault tolerant algorithms exist to compute such sets efficiently in arbitrary network graphs; such algorithms are required especially in mobile networks that allow nodes to enter and exit networks at times. Very recently, authors [2] have provided a self-stabilizing algorithm to compute two mutually disjoint minimal dominating sets of an arbitrary graph, provided the graph does not have any isolate. The correctness of the algorithm in [2] is based on the famous Ore's theorem in [1] that says "In a graph having no isolated nodes, the complement of any minimal dominating set is a dominating set". The algorithm in [2] has a complexity of $O(n^4)$ using state sharing model and the usual assumption that nodes have unique IDs and that each node stores complete information about its adjacent nodes.

Our contribution: In this brief paper, we propose a new self stabilizing algorithm to compute two mutually disjoint minimal dominating sets in an arbitrary graph with no isolates. We employ the concept of running two copies of an algorithm in an interleaving fashion such that mutual disjointness of the two minimal sets is maintained throughout the convergence process. We use the same assumptions as in [2], i.e., we use unfair central daemon and unique ids of the nodes. The time complexity of our algorithm is $O(n^3)$, an improvement by a factor of n from that of [2]. We expect the idea of running two copies of an algorithm concurrently to compute two disjoint subsets of interest in a graph will be useful to design self-stabilizing algorithms for other applications in a network graph.

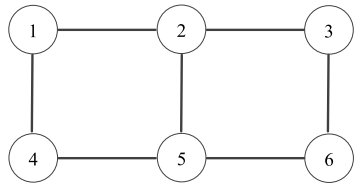

Fig. 1. A graph with 6 nodes.

2. Model and terminology

A network or a distributed system is modeled by an undirected graph $G = (V, E)$, where V is the set of nodes, and E is the set of edges. For a node i, $N(i)$, its *open neighborhood*, denotes the set of nodes adjacent to node i; $N[i] = N(i) \cup i$ denotes its *closed neighborhood*. For a node i, $N^2(i) = \cup_{j \in N[i]} N(j) - \{i\}$, its *2-hop open neighborhood*, denotes the set of nodes that are at most distance of 2 from node i. Each node $j \in N(i)$ is called a *neighbor* of node i and each node $j \in N^2(i)$ is called a *2-neighbor* of node i. We assume G to be connected and $n > 1$.

Consider any connected graph $G = (V, E)$, where $|V| = n$ and $|E| = m$. A set $\mathcal{S} \subseteq V$ is a *dominating* set (DS) if each node $i \in \{V - \mathcal{S}\}$ is adjacent to at least one node in $\mathcal{S}$, i.e., $N(i) \cap \mathcal{S} \neq \emptyset$. A node i is *dominated* if it has neighbor(s) in $\mathcal{S}$, and *undominated* otherwise. A dominating set $\mathcal{S}$ is called a **minimal dominating set (MDS)** iff there does not exist a node $i \in \mathcal{S}$ such that $\mathcal{S} - \{i\}$ is a dominating set.

An example graph is shown in Fig. 1, where $n = 6$, $m = 8$. $\{1, 6\}$ is a minimal dominating set; so are $\{3, 4\}$ and $\{2, 5\}$. Consider any arbitrary MDS, say $\{2, 5\}$; the nodes $V - \{2, 5\} = \{1, 3, 4, 6\}$ contains at least one MDS of the graph G, e.g., $\{1, 4, 3\}$.

The execution of the protocol at each node is managed by a central scheduler (daemon), that selects one privileged node in a system state to move in each step. A node is privileged in a given system state iff it is enabled to move by at least one rule of the protocol. The protocol terminates in a system state when no node is privileged (legitimate global system state). The protocol assumes a *shared-memory model* and each node knows only its own state and the local states of its immediate neighbors as is customary in self-stabilizing algorithms. The proposed algorithm does not need to know the size of the network graph.

3. Minimal dominating set

In order to design the algorithm for 2 disjoint MDS for an arbitrary graph G of n nodes $n > 1$, we use an earlier

M1: **if** $(x_i = 0) \wedge (\forall j \in N(i) : x_j = 0)$ **then set** $x_i = 1$ **and** $p_i^x = null$
M2: **if** $(x_i = 1) \wedge (\nexists j \in N(i) : p_j^x = i) \wedge (\exists k \in N(i) : x_k = 1)$
then { $x_i = 0$; **if** $\exists! j \in N(i) : x_j = 1$ **then** $p_i^x = j$ **else** $p_i^x = null$;
P1: **if** $(x_i = 1 \wedge p_i^x \neq null)$ **then** $p_i^x = null$;
P2: **if** $(x_i = 0) \wedge (\exists! j \in N(i) : x_j = 1) \wedge (p_i^x \neq j)$ **then** $p_i^x = j$ **else** $p_i^x = null$

Fig. 2. The Algorithm MDS at Node i, $1 \leq i \leq n$.

self-stabilizing algorithm to design a single MDS of a graph [10]. We briefly describe the algorithm below in a slightly different form to facilitate the development of our new algorithm in the next section.

Data structure: Consider an undirected graph $G = (V, E)$ with n nodes. Each node $i \in V$ has a binary flag x_i; a node i with $x_i = 1$ is called an MDS node while a node with $x_i = 0$ is a non MDS node. In any system configuration (global system state), the current candidate minimal dominating set (MDS) is given by $\mathcal{S} = \{i \in V : x_i = 1\}$. Each node i also has a pointer variable p_i^x that can point to any of its neighbors ($j \in N(i)$) or can be *null*.

We call a node $i \in S$ an *essential* iff $\exists j \in N[i]$ such that $N(j) \cap \mathcal{S} = \{i\}$ (node j is dominated only by node i); note that each node $i \in \mathcal{S}$ is essential when $\mathcal{S}$ is an MDS (legitimate system state); the node j is called *private neighbor* of node i in [10]. The algorithm starts in an arbitrary initial state and we assume a central (serial) daemon. The pseudocode of the protocol is shown in Fig. 2. We reiterate a few observations in Remark 1 and 2 that illustrate the approach underlying the algorithm and state a few lemmas and a theorem; we sketch the proofs (adjusting for our modified notations) from [10]) for ready reference.

Remark 1.

1. At termination of the algorithm, we call the system state **legitimate** (stable) since no node is privileged by any of the rules of the algorithm.
2. Steps M1 and M2 are called **Membership Moves** (a node changes its membership in the set $\mathcal{S}$ in only either of these moves) and steps P1 and P2 are called **Pointer Moves** (a node changes its pointer variable and does not change its membership in the set $\mathcal{S}$ in either of these moves).
3. When a node $i \notin \mathcal{S}$ executes rule M1, it enters into $\mathcal{S}$ and sets its pointer $p^x(i)$ to *null*.
4. When a node $i \in \mathcal{S}$ executes rule M2, it exits from $\mathcal{S}$ and sets its pointer p_i^x to its neighbor $j \in \mathcal{S}$ if j is its only such neighbor or to *null*, otherwise; note that node i has at least one neighbor $j \in S$ by the predicate of the rule.
5. A node $i \in \mathcal{S}$, enabled by rule P1 sets its pointer $p^x(i)$ to *null*; note that node i can be enabled by rule P1 iff it has not been enabled by any other rule before during the execution.
6. A node $i \notin \mathcal{S}$, enabled by rule P2, sets its pointer $p^x(i)$ to j if j is its only $\mathcal{S}$-neighbor and *null*, otherwise.

Lemma 1. *When the algorithm terminates, we have (a) a node $i \in \mathcal{S}$ has $p^x(i) = null$; (b) a node $i \notin \mathcal{S}$, has either $p^x(i) = j$, iff node i has exactly one neighbor $j \in \mathcal{S}$ or null, otherwise.*

Proof. (a) If a node $i \in \mathcal{S}$ (i.e., $x_i = 1$) has $p^x(i) \neq null$, then node i is privileged by rule P1, a contradiction. (b) If a node $i \notin \mathcal{S}$ and $p^x(i)$ is neither *null* or j where j is its only $\mathcal{S}$-neighbor of i, then node i is privileged by rule P2, a contradiction. □

Lemma 2. *(a) During execution, if a node ever makes a* M1 *move, it will not make another membership move; (b) starting from an arbitrary initial state, a node can make at most 2 membership moves until termination.*

Proof. (a) If a node i makes a M1 move at time t, then none of its neighbors are in $\mathcal{S}$ at time t. For i to later use M2 there must be a neighbor k for which $x(k) = 1$. But no k will be able to use M1 because $x(i) = 1$. (b) If a node's first membership move is M1, it will not make a membership move again. f its first membership move is M2 then any next membership move must be M1; it cannot make another membership move afterwards. □

Lemma 3. *There can be at most n consecutive pointer moves.*

Proof. Any pointer move by node i leaves i unprivileged. No pointer moves made by other nodes can make i privileged. Therefore, in a sequence of consecutive pointer moves, each node can move at most once. □

Theorem 1. *(a) The algorithm MDS, starting from an arbitrary initial state, can make at most $(2n + 1)n$ or $O(n^2)$ moves before termination; (b) at termination, the set $\mathcal{S}$ is a minimal dominating set of the graph G.*

Proof. By Lemma 2, there are at most $2n$ membership moves. Before and after each membership move there can be, by Lemma 3, at most n consecutive pointer moves. □

Remark 2 *([10]).* The algorithm MDS can stabilize with any arbitrary minimal dominating set of the given graph depending on the arbitrary initial state and the behavior of the daemon during the convergence process.

3.1. Restricted minimal dominating set

Again, consider an arbitrary given graph $G = (V, E)$ and a subset $X \subset G$. X is given to be a dominating (not necessarily a minimal dominating) set of G. Our goal is to design

RM1: **if** $(x \in X) \wedge \Big((x_i = 0) \wedge (\forall j \in N(i) : x_j = 0)\Big)$ **then set** $x_i = 1$ **and** $p_i^x = null$

RM2: **if** $(x \notin X) \vee \Big((i \in X) \wedge (x_i = 1) \wedge (\nexists j \in N(i) : p_j^x = i) \wedge (\exists k \in N(i) \cap X : x_k = 1)\Big)$

then $\begin{cases} x_i = 0; \\ \textbf{if } \exists! j \in N(i) \cap X : x_j = 1 \textbf{ then } p_i^x = j \textbf{ else } p_i^x = null; \end{cases}$

RP1: **if** $(x_i = 1 \wedge p_i^x \neq null)$ **then** $p_i^x = null$;

RP2: **if** $(x_i = 0) \wedge (\exists! j \in N(i) : x_j = 1) \wedge (p_i^x \neq j)$ **then** $p_i^x = j$ **else** $p_i^x = null$

Fig. 3. The Algorithm RMDS at Node i, $1 \leq i \leq n$.

an algorithm that will generate an MDS that is contained in X, i.e., $S \subseteq X$; note that this is always possible since X is known to be a dominating set (Lemma 2). We modify the algorithm MDS to get an MDS S of G such that $S \subseteq X$. We design the Restricted Minimal Dominating Set algorithm RMDS, by adding two simple clauses in the two membership moves in MDS: we force each node $i \notin X$ to unconditionally set $x_i = 0$ and a node i is allowed to conditionally set $x_i = 1$, iff $i \in X$ (identical to step M1 of algorithm MDS). Note that the pointer moves remain the same as in algorithm MDS. Nodes have the same data structure as in algorithm MDS. Fig. 3 shows the complete algorithm RMDS.

Remark 3. The previous Lemmas 1, 2 and 3 equally apply to the algorithm RMDS; the proofs are identical to proofs of those three lemmas.

Theorem 2. *The algorithm RMDS, starting from an arbitrary initial state, can make at most $(2n+1)n$ or $O(n^2)$ moves before termination.*

Proof. Proof of Theorem 1(a) applies here just like for Algorithm MDS). □

Theorem 3. *At termination, the set $\mathcal{S}$ is a minimal dominating set of the graph G.*

Proof. To see this, we simply need to show that at termination $\mathcal{S}$ is a dominating set (the rest is the same as the proof of Theorem 1(b)). Assume there is a node $j \in V$ that is not dominated by any $\mathcal{S}$-node. If $j \in V - X$, j has at least one neighbor in X, since X is a given dominating set; the rest as in the proof of Theorem 1. □

Corollary 1. *Assume algorithm RMDS is run on a graph $G = (V, E)$ and a subset $X \subset G$ when the set X is not a dominating set of graph G. Starting from an arbitrary system state, RMDS will terminate in at most $O(n^2)$ steps (Theorem 2 still applies here as before), but at termination, the set $\mathcal{S}$ is not a minimal dominating set of the graph G (Theorem 3 does not apply).*

4. Two disjoint minimal dominating sets

In this section, we develop a self stabilizing algorithm to compute two disjoint minimal dominating sets of a graph of n nodes, $n > 1$. While most of the self-stabilizing distributed algorithms for graph predicates related to dominance and independence have considered computation of a single subset of the graph with the desired property [6], authors in a recent paper [2] have provided an interesting (and the only one as far as we know) self-stabilizing algorithm that can compute two mutually disjoint minimal dominating sets in a graph, provided the graph does not have any isolate; this algorithm has been designed based on the famous Ore's theorem in [1] that says "In a graph having no isolated nodes, the complement of any minimal dominating set is a dominating set". The algorithm in [2] has a complexity of $O(n^4)$ using state sharing model and assuming nodes with unique IDs, as is usual in almost all such algorithms [6]. We show that the simple MDS algorithm (Fig. 2) to compute a single minimal dominating set and the RMDS algorithm (Fig. 3) can run concurrently in an interleaving manner to compute two mutually disjoint minimal dominating sets of an arbitrary graph with no isolates ($n > 1$) in $O(n^3)$ time using unique IDs of nodes and an arbitrary central daemon as in [2].

Data structures: Each node $i \in V$ has two binary variables x_i and y_i. A node with $x_i = 1$, (y_i is either 0 or 1) is called an $\mathcal{S}_1$-node and a node with $x_i = 0, y_i = 1$ is called a $\mathcal{S}_2$-node. In any configuration (global system state), $\{i \in V : x_i = 1\}$ denotes the current candidate set $\mathcal{S}_1$ and $\{i \in V : x_i = 0, y_i = 1\}$ denotes the current candidate set $\mathcal{S}_2$. In addition, each node i has two pointers p_i^x and p_i^y each of which can point to any of its neighbors $j \in N(i)$ or can be *null*; for a given node i, we call p_i^x its $\mathcal{S}_1$-pointer and p_i^y its $\mathcal{S}_2$-pointer respectively.

Remark 4. In any arbitrary global system state (configuration), $\mathcal{S}_1 \cap \mathcal{S}_2 = \emptyset$ and a node $i \in V - (\mathcal{S}_1 \cup \mathcal{S}_2)$ ($x_i = 0, y_i = 0$), does not belong to either $\mathcal{S}_1$ or $\mathcal{S}_2$; similarly, $V - \mathcal{S}_1 = \{i \in V : x_i = 0\}$.

Approach: Starting from an arbitrary initial state, we run algorithm MDS on graph G to compute $\mathcal{S}_1$ and algorithm RMDS (where the set $X = V - \mathcal{S}_1$) on graph G to compute $\mathcal{S}_2$, we will call them MDS1 and MDS2, concurrently in an interleaved fashion. Initially, the sets $\mathcal{S}_1$ and $\mathcal{S}_2$ are arbitrary depending on the arbitrary initial system state. Algorithm MDS1 reads the variables x_i and p_i^x of a node i and writes only on x_i and p_i^x of a node i. Algorithm MDS2 reads variables x_i, y_i and p_i^y of a node i and writes only on y_i and p_i^y. The goal is, when the combined protocol terminates, $\mathcal{S}_1$ is an MDS of the graph $G = (V, E)$), computed by MDS1, and $\mathcal{S}_2 \subseteq V - \mathcal{S}_1$ is an MDS of the graph $G = (V, E)$, computed by MDS2. Note that the execution of MDS2 does not affect the state variables that are read by MDS1, but the reverse is not true, i.e., execution of MDS1 affects the state variables that are read by MDS2. We begin by making some observations and defin-

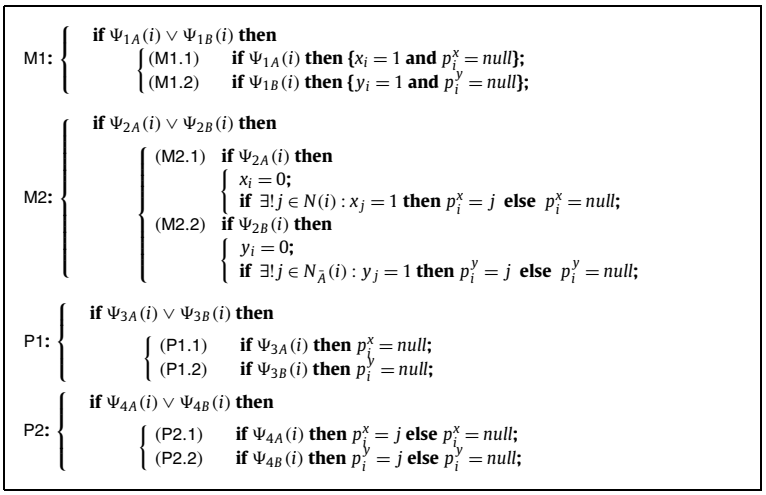

Fig. 4. The Algorithm 2-MDS at Node i, $1 \le i \le n$.

ing a few predicates to facilitate the stepwise development of the proposed protocol.

Definition 1. The local state of a node $i \in G$ is defined by the state variables $\{x_i, y_i, p_i^x, p_i^y\}$. Algorithm MDS1 can read only node variables $\{p_i^x, x_i, \forall i \in V\} = RS_1$ (Read set of MDS1), and can write on only node variables $\{p_i^x, x_i, \forall i \in V\} = WS_1$ (Write Set of MDS1) Similarly, algorithm MDS2 can read node variables $\{p_i^y, x_i, y_i, \forall i \in V\} = RS_2$ (Read Set of MDS_2) and can write on node variables $\{p_i^y, y_i, \forall i \in V\} = WS_2$ (Write Set of MDS2).

Lemma 4. *$WS_2 \cap RS_1 = \emptyset$, i.e., execution of* MDS2 *does not change the state variables of nodes, read by* MDS1*, as long as* MDS1 *does not execute.* ***Note:*** *membership moves of* MDS1 *may change the state variables of nodes, read by* MDS2*.*

Proof. The proof directly follows from the Definition 1. □

Definition 2. In a system state, a node i can locally compute each of the following Boolean predicates:

(a) For a node i, a Boolean predicate $\Psi_{1A}(i) = 1$ iff node $i \notin \mathcal{S}_1$ and none of its neighbors are in $\mathcal{S}_1$ (i.e., node i is privileged to enter $\mathcal{S}_1$), and $\Psi_{1B}(i) = 1$ iff node $i \notin \mathcal{S}_1 \cup \mathcal{S}_2$ and none of its neighbors are in $\mathcal{S}_1 \cup \mathcal{S}_2$.

$$\Psi_{1A}(i) \stackrel{\text{def}}{=} (x_i = 0) \wedge (\forall j \in N(i) : x_j = 0),$$

$$\Psi_{1B}(i) \stackrel{\text{def}}{=} ((x_i = 0) \wedge (y_i = 0)(\forall j : x_j = 0 \wedge y_j = 0)$$

Note: In any system state, if $\Psi_{1A}(i) = 1$, node i is privileged to enter $\mathcal{S}_1$; similarly, if $\Psi_{1B}(i) = 1$, node i is privileged to enter $\mathcal{S}_2$.

(b) For a node i, a Boolean predicate $\Psi_{2A}(i) = 1$ iff node $i \in \mathcal{S}_1$, there exists a neighbor of i in $\mathcal{S}_1$, and there does not exist a private neighbor of i, and $\Psi_{2B}(i) = 1$ iff node $i \in \mathcal{S}_2$, there exists a neighbor k of i in $\mathcal{S}_2$ and there does not exist a private neighbor of i in $(V - \mathcal{S}_1)$.

$$\Psi_{2A}(i) \stackrel{\text{def}}{=} (x_i = 1) \wedge (\nexists j \in N(i) : p_i^x = i) \wedge (\exists k \in N(i) : x_k = 1),$$

$$\Psi_{2B}(i) \stackrel{\text{def}}{=} (x_i = 0) \wedge (y_i = 1) \wedge (\nexists j \in N(i) : x_i = 0 \wedge p_j^y = i) \wedge (\exists k \in N(i) : x_i = 0 \wedge y_k = 1)$$

Note: If $\Psi_{2A}(i)$, node i is privileged to exit $\mathcal{S}_1$; similarly, if $\Psi_{2B}(i)$, node i is privileged to exit $\mathcal{S}_2$.

(c) For a node i, a Boolean predicate $\Psi_{3A}(i) = 1$ iff $i \in \mathcal{S}_1$ and its $\mathcal{S}_1$-pointer is not null; and, $\Psi_{3B}(i) = 1$ iff $i \in \mathcal{S}_2$ and its $\mathcal{S}_2$-pointer is not null.

$$\Psi_{3A}(i) \stackrel{\text{def}}{=} (x_i = 1) \wedge (p_i^x \neq null),$$

$$\Psi_{3B}(i) \stackrel{\text{def}}{=} (x_i = 0) \wedge (y_i = 1) \wedge (p_i^y \neq null)$$

Note: If $\Psi_{3A}(i)$, node $i \in \mathcal{S}_1$ is privileged to correct its p_i^x pointer; similarly, if $\Psi_{3B}(i)$, node $i \in \mathcal{S}_2$ is privileged to correct its p_i^y pointer.

(d) For a node i, a Boolean predicate $\Psi_{4A}(i) = 1$ iff $i \notin \mathcal{S}_1$ and there exists a single neighbor $j \in \mathcal{S}_1$, but i's $\mathcal{S}_1$-pointer is not j; $\Psi_{4B}(i) = 1$ iff $i \notin \mathcal{S}_1 \cup \mathcal{S}_2$ and there exists a single neighbor $j \in \mathcal{S}_2$, but i's $\mathcal{S}_2$-pointer is not j.

$$\Psi_{4A}(i) \stackrel{\text{def}}{=} (x_i = 0) \wedge (\exists! j \in N(i) : (x_j = 1 \wedge p_i^x \neq j)),$$

$$\Psi_{4B}(i) \stackrel{\text{def}}{=} (x_i = 0) \wedge (y_i = 0) \wedge (\exists! j \in N(i) : (y_j = 1) \wedge (p_i^y \neq j)$$

Note: If $\Psi_{4A}(i)$, node $i \notin \mathcal{S}_1$ is privileged to correct its p_i^x pointer; similarly, if $\Psi_{4B}(i)$, node $i \notin \mathcal{S}_2$ is privileged to correct its p_i^y pointer.

The complete pseudocode of the complete algorithm $2 - \text{MDS}$ is shown in Fig. 4; we make a few observations in Remark 5 that further illustrate the approach underlying the algorithm.

Definition 3. In any system state when no node is privileged by any one of the moves in the algorithm $2-\text{MDS}$, we say the system has stabilized, i.e., has reached in a legitimate state.

Remark 5.

1. Steps M1.1, M2.1, P1.1 and P2.1 constitute algorithm MDS1 and steps M1.2, M2.2, P1.2 and P2.2 constitute algorithm MDS2. Note that M1.1, M2.1 are memberships moves and P1.1 and P2.1 are pointer moves for algorithm MDS1; similarly, M1.2, M2.2 are memberships moves and P1.2 and P2.2 are pointer moves for algorithm MDS2 (similar to Remark 1, 2).
2. In any illegitimate state the central daemon arbitrarily selects any privileged node; rules are executed atomically. When a node i is privileged, it can be privileged by both MDS1 and MDS2 or by either one of them. Thus, when a privileged node is selected by the daemon, it executes a step for either or both of the algorithms.

Lemma 5. *The pointer adjustments in the algorithm* $2-\text{MDS}$ *are such that, in a legitimate state, (1) each node* $i \in \mathcal{S}_1$ *has* $p_i^x = null$*; (2) each node* $i \notin \mathcal{S}_1$ *has* $p_i^x = j$ *iff node* i *has exactly one neighbor in* $\mathcal{S}_1$ *and* $p_i^x = null$*, otherwise; (3) each node* $i \in \mathcal{S}_2$ *has* $p_i^y = null$*; (4) each node* $i \notin \mathcal{S}_2$ *has* $p_i^y = j$ *iff node* i *has exactly one neighbor in* $\mathcal{S}_2$ *and* $p_i^y = null$*, otherwise.*

Proof. By Straightforward generalization of Lemma 1. □

Lemma 6.

1. *Pointer moves of algorithm* MDS1 *do not affect the Read Set* RS_2 *of algorithm* MDS2.
2. *Membership moves of* MDS1 *affect the read set* RS_2 *of algorithm* MDS2.
3. *No moves (either membership or pointer) of* MDS_2 *affect the* RS_1 *of algorithm* MDS1.

Proof. (1) A pointer move of algorithm MDS1 changes only the pointer p_i^x of a node i which is never read by any move of algorithm MDS2 (Lemma 4). (2) A membership move of algorithm MDS1 changes the x_i variable of a node i which is read by the moves of algorithm MDS2 (Lemma 4). (3) Any move of algorithm MDS2 changes the variables p_i^y and y_i of a node i and moves of algorithm MDS1 reads y_i of a node i (Lemma 4). □

Lemma 7. *When algorithm* $2-\text{MDS}$ *executes (concurrent interleaving execution of two component algorithms* MDS1 *and* MDS2*), there can be at most* $O(n^2)$ *moves of algorithm* MDS2 *and n Pointer moves of algorithm* MDS1 *between two consecutive Membership moves of algorithm* MDS1.

Proof. Consider a time interval t between two consecutive Membership moves of algorithm MDS1. During the interval t, the set $\mathcal{S}_1 = \{i \in V : x_i = 1 \wedge y_i = 0\}$ does not change; while algorithm MDS1 can make at most n pointer moves during interval t, these pointer moves does not change $\mathcal{S}_1$ (Remark 1, 2 hence, the set $\mathcal{S}_1$ does not change. Thus, only algorithm MDS_2 (algorithm RMDS with $X = V - \mathcal{S}_1$) will execute and terminate in $O(n^2)$ moves in the worst case (Theorem 2). □

Theorem 4. *Starting from an arbitrary illegitimate state, the algorithm* $2-\text{MDS}$ *terminates in* $O(n^3)$ *steps (moves).*

Proof. By Lemma 2 algorithm MDS1 can make $2n$ Membership moves in the worst case before termination and in between two such moves there can be at most $(n^2 + n)$ moves by algorithm MDS2 (Lemma 7). Thus the algorithm $2-\text{MDS}$ terminates in $O(2n \times n^2)$ or $O(n^3)$ steps (moves). □

Theorem 5. *When the algorithm* `2-MDS` *terminates, the two sets* $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *denote two mutually disjoint minimal dominating sets of the graph* G*, i.e.,* $\mathcal{S}_1 \cap \mathcal{S}_2 = \emptyset$.

Proof. By definition, the two sets $\mathcal{S}_1$ and $\mathcal{S}_2$ are always mutually disjoint (Remark 4). When the component algorithm MDS1 has stabilized during the execution of `2-MDS`, $\mathcal{S}_1$ is a minimal dominating set of G (Theorem 1); the set $\mathcal{S}_1$ does not change until `2-MDS` terminates. When the component algorithm MDS2 terminates, the resulting $\mathcal{S}_2$ is a minimal dominating set of G (Theorem 3). □

Acknowledgements

We are grateful to the reviewers for their comments to improve the presentation. We acknowledge partial support from NSF Grant DBI 1759856.

References

[1] O. Ore, Theory of Graphs, Amer. Math. Soc. Colloq. Publ., 1962.
[2] S.T. Hedetniemi, D.P. Jacobs, K.E. Kennedy, A theorem of ore and self-stabilizing algorithms for disjoint minimal dominating sets, Theor. Comput. Sci. 593 (2015) 132–138.
[3] E.W. Dijkstra, Self-stabilizing systems in spite of distributed control, Commun. ACM 17 (11) (Nov. 1974) 643–644.
[4] B. Han, W. Jia, Clustering wireless ad hoc networks with weakly connected dominating set, J. Parallel Distrib. Comput. 67 (6) (2007) 727–737.
[5] J. Yu, N. Wang, G. Wang, Constructing minimum extended weakly-connected dominating sets for clustering in ad hoc networks, J. Parallel Distrib. Comput. 72 (1) (2012) 35–47.
[6] N. Guellati, H. Kheddouci, A survey on self-stabilizing algorithms for independence, domination, coloring, and matching in graphs, J. Parallel Distrib. Comput. 70 (4) (2010) 406–415.
[7] Justin Southey, Michael A. Henning, A characterization of graphs with disjoint dominating and paired-dominating sets, J. Comb. Optim. 22 (2011) 217–234.
[8] W. Klostermeyer, M.E. Messinger, A. Ayello, Disjoint dominating sets with a perfect matching, Discrete Math. Algorithms Appl. 9 (2017).
[9] Michael A. Henning, Iztok Peterin, A characterization of graphs with disjoint total dominating sets, Ars Math. Contemp. 16 (2) (2019) 359–375.
[10] S.M. Hedetniemi, S.T. Hedetniemi, D.P. Jacobs, P.K. Srimani, Self-stabilizing algorithms for minimal dominating sets and maximal independent sets, Comput. Math. Appl. 46 (2003) 805–811.