Self-Stabilizing Domination Algorithms

Stephen T. Hedetniemi

1 Introduction

In this chapter, we introduce the elegantly simple, self-stabilizing algorithm model to researchers having an interest in domination in graphs. In 1974 [12, 13], Dijkstra introduced the algorithm paradigm called *self-stabilizing algorithms* as a special case of distributed algorithms. But algorithms of this type were not studied and developed until the late 1980s, and it was not until the early 2000s that self-stabilizing domination algorithms began to appear.

In this chapter, we present the basic framework and definitions of self-stabilizing algorithms. An in-depth treatment of self-stabilizing algorithms is given in the book by Dolev [16]. We then present self-stabilizing algorithms for finding in an arbitrary connected graph: (i) a maximal independent set, (ii) a maximal matching, (iii) a minimal dominating set, (iv) a minimal total dominating set, and (v) two disjoint minimal dominating sets. It is important to note at the outset that these algorithms are not designed to find either minimum or maximum sets having some domination property, only minimal or maximal sets. We then discuss a variety of other domination-related, self-stabilizing algorithms that have been published. Finally, we present a list of domination-related self-stabilizing algorithms that have yet to be designed.

S. T. Hedetniemi (✉)
School of Computing, Clemson University, Clemson, SC, USA
e-mail: hedet@clemson.edu

T. W. Haynes et al. (eds.), *Structures of Domination in Graphs*, Developments in Mathematics 66, https://doi.org/10.1007/978-3-030-58892-2_16

2 Self-Stabilizing Framework

In the self-stabilizing algorithm paradigm, we assume that a distributed computing system or computer network is modeled by a connected, undirected graph $G = (V, E)$, of *order* $n = |V|$ nodes or processors, and *size* $m = |E|$ edges or bidirectional communication links $\{u, v\}$ between pairs of nodes. If $\{u, v\} \in E$, we say that u and v are *neighbors*, and $N(u) = \{v : \{u, v\} \in E\}$ is the set of neighbors of node u, or the *open neighborhood* of u, while $N[u] = N(u) \cup \{v\}$ is the *closed neighborhood* of u. If $S \subseteq V$ is a set of vertices in a graph G, we let $\overline{S}$ (sometimes denoted $V - S$ or $V \setminus S$) denote the vertices not in set S.

2.1 Program and Computation

Every node, at all times, continues to execute the same program or self-stabilizing algorithm and in doing so maintains a set of variables common to all nodes. A node can only change the value of its own variables. The *state* of a node is defined by the vector of current values of all of its variables. The union of the states of all nodes in the graph/system defines the *global state* and constitutes the *current configuration* of the whole system.

The algorithm, which is being executed independently and simultaneously at every node of the system, consists of the same finite list of *rules*, called *guarded commands*, of the form,

$$Rule : Guard \rightarrow Action$$

or

$$Rule : \textbf{if..}\ Guard\ \textbf{..then..}\ Action,$$

where *Guard* is a Boolean expression involving some or all of the variables of the nodes in the closed neighborhood of a node u; this is called the *shared-variable model*. If this expression (Guard) is evaluated to be true, then node u is said to be *enabled* or *privileged* to execute the corresponding Action. This gives rise to two types of execution. In what is sometimes called *coarse scheduling*, both reading/expression evaluation and writing/making a move are done in one step, while in what is called *read/write atomicity*, two steps are required. A *move* by node u consists of the execution of the designated Action, which consists of changing the values of the variables at node u as specified by the Action.

Normally at most one rule at a node will be enabled at any moment, but if several rules are simultaneously enabled, only the Action in the first enabled rule in the list will be executed.

2.2 Distance-k Knowledge

In 2004 [19], Gairing, Goddard, Hedetniemi, Kristiansen, and McRae introduce the idea that self-stabilizing algorithms can be designed in which the rules have guards whose Boolean expressions involve some or all of the variables of the nodes within distance-k of the given node. It is shown in 2004 [19] and subsequently in 2008 [28] by Goddard, Hedetniemi, Jacobs, and Trevisan how to convert a distance-k algorithm to one in the distance-1 model, but it comes with an increased cost in running time, for example, a self-stabilizing algorithm, which stabilizes in $O(n^2)$ moves in the distance-2 model, will stabilize in $O(n^5)$ moves in the distance-1 model; see also Turau in 2012 [64].

2.3 Anonymous Systems

In an *anonymous* system, or network, nodes do not have unique identifiers, e.g., $ID(u) = ID(v)$, for all $v \in N(u)$, which means that the same rule applies equally to all nodes. By contrast, in *non-anonymous* networks, a rule can compare the identifier of a node u with the identifiers of nodes in its neighborhood $N(u)$, in order to determine if node u is enabled, for example, if $ID(u) > ID(v)$, then the node u may become enabled, otherwise node v may become enabled.

2.4 Schedulers

If, at any time, several nodes are enabled to make a move, a mechanism, called a *scheduler*, or an *adversarial daemon*, is assumed to determine, decide, or choose which node or nodes make the next move(s). In the *central scheduler model*, also called the *serial model*, one node is adversarially selected to make its move. In the *distributed* model, any number of enabled nodes can be adversarially selected to make their moves simultaneously, while in the *synchronous* model, all enabled nodes must make their moves simultaneously.

A further distinction can also be made between *fair* and *unfair* schedulers. With a fair scheduler, every node that is continuously enabled is eventually selected to make a move. With an unfair scheduler, there is no such condition.

2.5 Self-Stabilization

A *computation* c is a finite sequence of global configurations $c = c_0, c_1, \ldots, c_k$, where configuration c_i results from configuration c_{i-1} after all enabled nodes selected by the scheduler have made their next moves.

A configuration is said to be *stable* if no node is enabled. A self-stabilizing algorithm is said to be *stabilizing* if, regardless of any initial configuration c_0, the system always reaches a stable state c_k after a finite number of moves.

The major objective of self-stabilization is for a system to always achieve a *desired* or *legitimate* stable state. An algorithm is called *self-stabilizing* if (i) when started in any initial illegitimate state, it always reaches a legitimate state after a finite number of moves, and (ii) for any legitimate state and for any move enabled by that state, the next state is always a legitimate state.

2.6 Running Times

The (worst-case) running time of a self-stabilizing algorithm under a *central* scheduler is defined to equal the maximum possible number of moves from any initial configuration to a stable configuration.

The running time of an algorithm under the *distributed* scheduler can be measured by the total number of moves, or the number of time steps, or rounds. A *round* as discussed by Dolev in [16] is a minimal sequence of time steps where every enabled node at the start of the round either makes a move or has its move disabled by the move of a neighbor; if the scheduler is *fair*, every round is guaranteed to finish.

For the *synchronous* scheduler, the number of time steps and the number of rounds are identical. In general, the number of moves is an upper bound on the number of time steps.

2.7 Rationale for Self-Stabilizing Algorithms

One of the most important requirements of modern distributed systems is that they should be fault tolerant, which means that a system should be able to function correctly in spite of intermittent or infrequent faults. Ideally, the global state of the system should be legitimate and should remain legitimate. But often enough, system malfunctions can put the system in some arbitrary illegitimate state. It is desirable, therefore, that some mechanism, other than a system-wide reset or external agent, is in place, which can automatically bring the system back to a legitimate global state.

The traditional approach to this type of fault tolerance is to assume worst-case scenarios and make significant efforts to protect the system against such eventualities at the cost of additional hardware and software. Such additional costs may not be an economic option, especially in cases when faults are only transient, subsequent repairs can be made, and short-term unavailability of system service can be tolerated while the system re-establishes a legitimate state.

Since the stabilization time must be small with respect to the frequency of faults, the speed of self-stabilization is important. A self-stabilizing system cannot guarantee that the system is able to operate properly when a node or link continuously injects faults into the system or when communication errors occur so frequently that a new legitimate state cannot be reached. But once the offending fault is removed or corrected, the system can once again provide its necessary services after a reasonable amount of self-stabilizing time.

3 Self-Stabilizing Maximal Independent Set Algorithms

In this section, we present what may well be the simplest and most elegant of all self-stabilizing algorithms, due to Skukla, Rosenkrantz, and Ravi in 1995 [59]. This algorithm only has two rules and in $O(n)$ time finds a maximal independent set of nodes, which of course is also a minimal dominating set. This algorithm assumes that there is a central scheduler, whereby only one, adversarially chosen node can make a move at a time. All nodes are anonymous and make no use of identifier information. Notice, before we get started, that this algorithm does not find either a minimum cardinality maximal independent set or a maximum cardinality independent set, only a maximal independent set, which is all that is required in many distributed system applications.

Recall that a set $S \subset V$ is *independent* if no two nodes in S are neighbors.

In this self-stabilizing algorithm, each node maintains only one Boolean variable x, such that $x(i) = 1$ if node i is in the maximal independent set S and $x(i) = 0$ if node i is not in S. Algorithm MIC in Figure 1 only has the following two, very simple rules.

Rule C1 says that if node i is not in S and has no neighbor in S, then it is enabled to enter S.

Rule C2 says that if node i is in S and has a neighbor in S, then it is enabled to leave the set S.

Given this algorithm, one must prove each of the following:

```
Algorithm MIC: Maximal Independent - Central
C1: if $(x(i) = 0) \wedge (\nexists j \in N(i) : x(j) = 1)$
        then $x(i) := 1$                                    [enter set]
C2: if $(x(i) = 1) \wedge (\exists j \in N(i) : x(j) = 1)$
        then $x(i) := 0$                                    [leave set]
```

Fig. 1 Algorithm MIC: Central Model [59]

(i) Under the central scheduler, regardless of the initial global state, and regardless of the sequence of moves made, a stable state must be reached after a finite number of moves.
(ii) In every stable state, the set of nodes S, for which $x(i) = 1$, must always define a maximal independent set.

It is also important to ascertain the running time, i.e., worst-case performance, of this algorithm. We will show that it stabilizes after at most $O(n)$ moves for any graph of order n.

In order to prove that this algorithm stabilizes, we will use the following lemmas.

Lemma 1 *After a node executes Rule C1, it can never make another move.*

Proof After a node i executes Rule C1, $x(i) = 1$ and all of its neighbors $j \in N(i)$ have $x(j) = 0$. As long as $x(i) = 1$, node i cannot execute Rule C1, and it can only execute Rule C2 if $x(i) = 1$ and a neighbor $j \in N(i)$ has $x(j) = 1$. But as long as $x(i) = 1$, no neighbor $j \in N(i)$ can execute Rule C1, and therefore every neighbor j must remain in state $x(j) = 0$. □

Lemma 2 *After a node executes Rule C2, it can only execute Rule C1.*

Proof After a node executes Rule C2, its value has changed from $x(i) = 1$ to $x(i) = 0$, and therefore it is no longer able to execute Rule C2, which requires $x(i) = 1$. □

Theorem 1 (Shukla et al.) *Algorithm MIC stabilizes in at most* $2n$ *moves.*

Proof A node can only execute four possible move sequences: (i) no move at all, (ii) Rule C1, (iii) Rule C2, and (iv) Rule C2 followed by Rule C1. Thus, if there are n nodes, at most $2n$ moves can ever be executed. □

Lemma 3 *If Algorithm MIC is stable, the set* $S = \{i \mid x(i) = 1\}$ *is an independent set.*

Proof Assume that Algorithm MIC is in a stable set and S is not an independent set. Then, by definition, there must be two adjacent nodes i and j, both of which have $x(i) = 1$ and $x(j) = 1$. But in this case both node i and node j are enabled to execute Rule C2, and hence Algorithm MIC is not stable: a contradiction. □

Lemma 4 *If Algorithm MIC is stable, then the set S is a maximal independent set.*

Proof Assume that Algorithm MIC is in a stable state and S is an independent set but is not a maximal independent set. Then, by definition, there must exist a node i that is not in S and has no neighbors in S, which means that $x(i) = 0$ and for every $j \in N(i)$, $x(j) = 0$. But in this case node i is enabled to execute Rule C1 and, therefore, Algorithm MIC is not stable. □

Thus, as desired, Algorithm MIC stabilizes and finds a maximal independent set in $O(n)$ time, in fact, in at most $2n$ moves. This is arguably the simplest of all self-stabilizing graph algorithms. It is worth pointing out that Algorithm MIC is general, in that it can stabilize with any possible maximal independent set, and can do so starting from the initial All-Zero configuration in which $x(i) = 0$, for all nodes i. For example, let $S = \{v_1, v_2, \ldots, v_k\}$ be any maximal independent set. Starting in the All-Zero configuration, Algorithm MIC, under the central scheduler, could select each of these nodes, in order, to execute Rule 1; at the end of k moves, the maximal independent set will be determined and the algorithm will be stable.

3.1 Distributed Model Maximal Independent Set Algorithm

While Algorithm MIC is designed to run under a central scheduler, Algorithm MID, shown in Figure 2, due to Ikeda, Kamei and Kakugama in 2002 [41], is designed to find a maximal independent set under a distributed scheduler. This means that at any time, any adversarially chosen subset of enabled nodes can simultaneously make a move. Such a set of moves constitutes one *round*.

We show here that Algorithm MID stabilizes in $O(n)$ rounds.

Again, each node maintains only one Boolean variable x such that $x(i) = 1$ if node i is in S and $x(i) = 0$ if node i is not in S. But notice the key difference between Algorithms MIC and MID; Algorithm MID, under the distributed scheduler, uses the ID, which is the integer index i of a node, to determine its eligibility to make a move, where we assume that all nodes have a unique ID. In fact, we only need to assume that no two nodes in any closed neighborhood have the same ID.

Note that while Rule D1 is the same as Rule C1 in Algorithm MIC, Rule D2 is slightly different than Rule C2 and asserts that a node in the set S can only be forced to leave S if it has a neighbor in S whose ID is larger.

Theorem 2 (Ikeda, Kamei, Kakugawa) *Starting from an arbitrary state, Algorithm MID stabilizes in at most $O(n^2)$ moves, and when stable, the set $S = \{i : x(i) = 1\}$ is a maximal independent set.*

In [41], Ikeda et al. construct an example where Algorithm MID takes $\Theta(n^2)$ time steps. In 2008 [26], Goddard, Hedetniemi, Jacobs, Srimani, and Xu determine the running time of Algorithm MID in terms of rounds, as follows.

Theorem 3 (Goddard et al.) *Starting from an arbitrary state, Algorithm MID stabilizes in at most n rounds.*

Proof We prove this by showing that in every round R there is a node v_R, which moves and, having moved, never moves again.

```
Algorithm MID: Maximal Independent - Distributed
D1: if $(x(i) = 0) \wedge (\nexists j \in N(i) : x(j) = 1)$
        then $x(i) := 1$                                    [enter set]
D2: if $(x(i) = 1) \wedge (\exists j \in N(i) : j > i \wedge x(j) = 1)$
        then $x(i) := 0$                                    [leave set]
```

Fig. 2 Algorithm MID: Distributed Model [41]

Case 1. Assume that some node executes Rule D1 in round R. Since no two nodes have the same ID value, let v_R be a node with maximum ID that executes Rule D1 in round R and sets $x(v_R) = 1$. Since v_R executed Rule D1, before this time step none of its neighbors were in S. By the choice of v_R, any neighbor of v_R that also executes Rule D1 in round R has a smaller ID value than node v_R. After this round, no other neighbor of v_R can execute Rule D1, since $x(v_R) = 1$. Furthermore, v_R will never leave the set S by executing Rule D2, since it has a larger ID than any of its neighbors also in S. Hence, v_R will never move again.

Case 2. If no node executes Rule D1 in round R, let v_R be a node in S, with $x(v_R) = 1$, that executes Rule D2 during this round, because it has a neighbor, say w_R with $x(w_R) = 1$ and $w_R > v_R$. Assume furthermore that over all such pairs of neighbors, v_R and w_R, where v_R executes Rule D2, w_R has the maximum ID.

By the choice of v_R and w_R, w_R is not enabled to execute Rule D2 in round R. Hence, w_R stays in S for the rest of the round. It follows that all neighbors v_R of w_R in S that execute Rule D2 in round R will ever move again.

Because of Cases 1 and 2, it follows that the number of rounds is at most the number of nodes. □

As with Algorithm MIC, it is possible for Algorithm MID to stabilize with any possible maximal independent set and can do so starting from the initial All-Zero configuration.

3.2 Synchronous Model Maximal Independent Set Algorithm

In this section, we present a synchronous model, self-stabilizing Algorithm MIS, in Figure 3, for finding a maximal independent set, due to Goddard, Hedetniemi, Jacobs, and Srimani in 2003 [22].

We again assume that no two neighbors have the same ID and that every node can compare its ID with the IDs of all of its neighbors.

Rule S1 says that a node not in S may enter S provided it does not have a neighbor with larger ID already in S. If it enters S with a neighbor with smaller ID already in S, then subsequently that neighbor with a smaller ID will be forced to leave S.

Similarly, Rule S2 says that a node must leave set S if it has a neighbor in S, which has a larger ID.

```
Algorithm MIS: Maximal Independent - Synchronous
S1: if $(x(i) = 0)(\nexists j \in N(i) : j > i \wedge x(j) = 1)$
        then $x(i) := 1$                                    [enter set]
S2: if $(x(i) = 1) \wedge (\exists j \in N(i) : j > i \wedge x(j) = 1)$
        then $x(i) := 0$                                    [leave set]
```

Fig. 3 Algorithm MIS: Synchronous Model [22]

The proofs of correctness and the running time of this algorithm are given by Goddard et al. [22] as follows.

Lemma 5 *If at any time t, the set S of nodes with* $x(i) = 1$ *does not form an independent set, then at least one node will execute Rule S2 during the next round.*

Proof Assume that at some time t there exists at least one pair of adjacent nodes, both of which are in S, that is, the set S is not independent. Among all nodes in S, which have neighbors also in S, let node v_R have the smallest ID. It follows that this node is enabled to execute Rule S2 and must do so during the next round. □

Lemma 6 *If at any time t, the set S of nodes with* $x(i) = 1$ *forms an independent set but does not form a maximal independent set, then at least one node will execute Rule S1 during the next round.*

Proof Assume that at some time t the set S of nodes with $x(i) = 1$ forms an independent set but does not form a maximal independent set. Then there must exist a node v_R for which $x(v_R) = 0$ and all neighbors $w_R \in N(v_R)$ have $x(w_R) = 0$. Clearly, node v_R is enabled to execute Rule S1 during the next round. □

Theorem 4 *If Algorithm MIS stabilizes, then the set S of nodes with* $x(i) = 1$ *forms a maximal independent set.*

Proof From Lemma 5, we know that if Algorithm MIS stabilizes then S must be an independent set, and from Lemma 6, we know that if S stabilizes then S must be a maximal independent set. □

Theorem 5 *Algorithm MIS stabilizes in* $O(n)$ *rounds.*

Proof At time $t = 1$, after the first round, we know that all nodes v whose ID is larger than the IDs of all of their neighbors will have value $x(v) = 1$. If they have $x(v) = 1$ at time $t = 0$, then they are not enabled to execute Rule S2 and will remain after the first round with $x(v) = 1$. If they have $x(v) = 0$ at time $t = 0$ and will be enabled to execute Rule S1, then they have $x(v) = 1$ after the first round. Furthermore, none of these largest ID nodes will ever be enabled to execute rule S2. Since there is one largest ID node, call it v_1, it will be permanently set to $x(v_1) = 1$ after round one, and every neighbor $w \in N(v_1)$ will be permanently set to $x(w) = 0$ after round two.

By time $t = 3$, after the third round, the node, say v_3, with the largest ID among the nodes in $V - N[v_1]$ will be permanently set to $x(v_3) = 1$, and after time $t = 4$, all neighbors of v_3 will have their x-values set permanently to zero.

This process will continue until all nodes are stable after at most n rounds. □

3.3 Other Self-Stabilizing Independent Set Algorithms

Several other self-stabilizing algorithms have appeared for finding maximal independent sets. For example, in 2007 [63] Turau presents such an algorithm using an unfair distributed scheduler, which stabilizes in at most $max\{3n-5, 2n\}$ moves.

In 2013 [35], Hedetniemi, Jacobs, and Kennedy present several self-stabilizing algorithms for finding disjoint independent sets S_1 and S_2, where S_1 is a maximal independent set, and S_2 is a maximal independent set in the graph $G[\overline{S}_1]$ induced by $\overline{S}_1$.

Maximal *k*-Packings
An equivalent definition of an independent set is a set S of nodes having the property that for any $i, j \in S$, $d(i, j) > 1$, that is, no two nodes are adjacent. This immediately generalizes to a *k-packing*, which is a set S of nodes having the property that for any $i, j \in S$, $d(i, j) > k$. It should be noted that a maximal k-packing is also a *minimal distance-k dominating set*, which means that every node $i \in \overline{S}$ is within distance-k of at least one node in S.

Note, in this regard, that every maximal independent set is a minimal distance-1 dominating set. Self-stabilizing maximal k-packing algorithms have been developed by Kristiansen in his 2002 PhD thesis [49], by Gairing, Geist, Hedetniemi, and Kristiansen in 2004 [17], Goddard, Hedetniemi, Jacobs, and Srimani in 2005 [25], by Shi in 2012 [58], and by Trejo-Sánchez, Fernández-Zepeda, and Ramírez-Pacheco in 2017 [62].

Maximal *k*-Dependent Sets
Still another equivalent definition of an independent set is that it is a set S of nodes having the property that the maximum degree of a node in the subgraph $G[S]$ of G induced by S is zero. A *k-dependent set* is a set S of nodes having the property that the maximum degree of a node in the induced subgraph $G[S]$ is at most k, or equivalently if for every $i \in S$, $|N(i) \cap S| \leq k$.

In 2004 [18], Gairing, Goddard, Hedetniemi, and Jacobs present the following simple, two-rule, self-stabilizing Algorithm MKD, in Figure 4, for finding a maximal k-dependent set, using a central scheduler; it stabilizes in at most $2kn + 3n$ moves. This algorithm uses only one, non-negative integer variable $f(i) \in \{0, 1\}$, where $f(i) = 1$ if node $i \in S$, $f(i) = 0$ if node $i \notin S$, and $f(N(v_i)) = \Sigma_{j \in N(i)} f(j)$.

```
Algorithm MKD: Maximal $k$-Dependent
KD1: if $(f(i) = 0) \wedge (f(N(i)) \leq k)$
       then $f(i) := 1$
KD2: if $(f(i) = 1) \wedge (f(N(i)) > k)$
       then $f(i) := 0$
```

Fig. 4 Algorithm MKD: Central Model [18]

Minimal Vertex Covers

A *vertex cover* is a set S of nodes having the property that every edge $e = uv$ contains a vertex in S, that is, either $u \in S$ or $v \in S$ or both. It is well known and easily proved that the complement $\overline{S}$ of every (maximal) independent set S is a (minimal) vertex cover, and conversely, the complement of every (minimal) vertex cover is a (maximal) independent set. Given this, every self-stabilizing algorithm for finding a maximal independent set S also finds a minimal vertex cover $\overline{S}$, that is the nodes i with $x(i) = 0$.

Several papers have focused on finding minimal vertex covers within a constant factor of optimality, such as Turau in 2010 [65], Turau and Hauck in 2011 [69], and Delbot, Laforest, and Rovedakis in 2014 [11].

4 Self-Stabilizing Maximal Matching Algorithms

Given an undirected graph $G = (V, E)$, a *matching* is defined to be a set $M \subseteq E$ of pairwise disjoint edges. That is, no two edges in M are incident with the same node. A matching M is *maximal* if there does not exist another matching M' such that $M' \supset M$.

4.1 Central Model Maximal Matching Algorithm

In 1992 [38], Hsu and Huang present the first self-stabilizing algorithm for finding a *maximal matching* in a distributed network $G = (V, E)$ under a central scheduler. They show that their algorithm stabilizes in $O(n^3)$ moves. A further running time analysis of their algorithm is given in by Tel in 1994 [61], who shows that Algorithm Hsu–Huang stabilizes in $O(n^2)$ moves. A subsequent paper by Hedetniemi, Jacobs, and Srimani in 2001 [37] shows that, in fact, Algorithm Hsu–Huang, in Figure 5, stabilizes in $O(m)$ moves, where $m = |E|$ is the number of edges. We present next the Hsu–Huang algorithm.

```
Algorithm Hsu-Huang
M1: if $(i \to \text{null}) \wedge (\exists j \in N(i) : j \to i)$
        then $i \to j$                                              [accept proposal]
M2: if $(i \to \text{null}) \wedge (\forall k \in N(i) : \neg(k \to i)) \wedge (\exists j \in N(i) : j \to \text{null})$
        then $i \to j$                                              [make proposal]
M3: if $(i \to j) \wedge (j \to k) \wedge (k \neq i)$
        then $i \to \text{null}$                                    [withdraw proposal]
```

Fig. 5 Algorithm Hsu–Huang, Central scheduler [38]

Each node maintains just one variable, a pointer, which is either null, denoted $i \rightarrow null$, or points to a neighbor $j \in N(i)$, denoted $i \rightarrow j$. The algorithm has just three rules.

Rule M1 allows a node i to *accept* a proposed match with another node j, which is pointing to node i, provided $i \rightarrow null$.

Rule M2 allows a node i to *propose* matching with a neighbor j, which currently is not matched ($j \rightarrow null$), provided no other node k is currently proposing a match with node i by pointing to i.

Rule M3 allows a node i to *withdraw* a proposal if the node j to which it is pointing is currently pointing to some other node k.

An edge between two adjacent nodes i and j becomes a permanent edge of a maximal matching when each is pointing to the other, $i \rightarrow j$ and $j \rightarrow i$, in which case we say that nodes i and j are *matched*. The maximal matching M produced by the Algorithm Hsu–Huang is the set of edges $e = \{i, j\}$ such that $i \leftrightarrow j$.

We present the proof given in [37] that this algorithm stabilizes in at most $2m + n$ moves.

For every move made by a node i, there is a corresponding node j that enables the move; we will denote such a move by (i, j, Mk), for $1 \leq k \leq 3$, and say that it is an (i, j)-move. Let $c(i, j)$ denote the number of (i, j)-moves that has been executed, and let $c(e)$ denote the number $c(e) = c(i, j) + c(j, i)$.

After a move $(i, j, M1)$ or $(j, i, M1)$ has been executed, we will say that i and j are *matched*.

Lemma 7 *After nodes i and j have been matched, neither node can make another move.*

Proof After an $(i, j, M1)$ move, neither node i nor node j will have a null pointer and are therefore not enabled to make move M1 nor M2. Furthermore, since $i \leftrightarrow j$, neither node is enabled to execute Rule M3. □

Lemma 8 *After an* $(i, j, M2)$*-move, at most one more* (i, j)*-move is possible, namely* $(i, j, M3)$.

Proof Let $m = (i, j, M2)$ be a move on the edge (i, j), and let $m' = (i, j, Mk)$ be the next move on the same edge. Clearly, it can only be $(i, j, M3)$. It then suffices to show that no further (i, j)-move can occur.

After move m, we must have $i \rightarrow j$ and $j \rightarrow null$, and prior to move m', we must have $i \rightarrow j$ and $j \rightarrow k$ for some $k \neq i$. Thus, sometime after move m and before move m', there must have been a move m" of the form $m" = (j, k, M1)$, which implies that node j is permanently matched with node k. Being permanently matched, there can be no more (i, j)-moves. □

Lemma 9 *Following a move* $(i, j, M2)$*, there can be only one more move on the edge* (i, j)*, either* $(j, i, M1)$ *or* $(i, j, M3)$.

Proof Once a proposal has been made with an $(i, j, M2)$ move, node j is enabled to make move $M1$. It must either accept a proposal from node i or another node k. If it chooses node i and makes the move $(j, i, M1)$, then it will become permanently matched with node i and no further move can be made on this edge. But if node j chooses another node k, it will become permanently matched with node k, forcing node i to execute $M3$, after which no further move can be made on edge (i, j). □

Lemma 10 *Following a move* $(i, j, M3)$*, there are at most two more moves on the edge* $e = (i, j)$*.*

Proof If there is to be another move on edge (i, j) following a move $(i, j, M3)$, then node j will have to reset its pointer to null by executing $M3$. With both pointers set to null, the next move on the edge can only be a proposal, $(i, j, M2)$ or $(j, i, M2)$. But by Lemma 9, there can only be one more move on this edge. □

Consider an arbitrary initial state of the system, and if initially $i \rightarrow j$, let edge (i, j) be called an *initial edge*. Let I denote the set of all initial edges. Note that initially there can only at most n initial edges, one for each node i, thus, $|I| \leq n$. Recall that we have defined $c(e) = c(i, j) + c(j, i)$ to count the number of moves made on the undirected edge $e = (i, j)$.

Lemma 11 *For each edge* $e \in E$*,* $c(e) \leq 3$ *and for at most* n *edges,* $c(e) = 3$*.*

Proof If $c(e) > 0$, then there is a first move m on this edge $e = (i, j)$, either $m = (i, j, M1)$, or $m = (i, j, M2)$ or $m = (i, j, M3)$. Lemmas 7, 9 and 10 prove that $c(e) \leq 3$.

In order to prove that for at most n edges, $c(e) = 3$, let $C_3 = \{e | c(e) = 3\}$. If some edge $e \in C_3$, then the first move on this edge must be of the form $(i, j, M3)$. But this implies that the initial state of node i is $i \rightarrow j$, and this means that $e \in I$ and so $C_3 \subseteq I$, and therefore $|C_3| \leq n$. □

Theorem 6 *For any graph* $G = (V, E)$ *having order* $n = |V|$ *and size* $m = |E|$*, Algorithm Hsu–Huang stabilizes in at most* $2m + n$ *moves under the central scheduler.*

Proof This follows from Lemma 11. □

4.2 Synchronous and Distributed Model Maximal Matching Algorithm

In 2003 [22], Goddard, Hedetniemi, Jacobs, and Srimani show that the following Algorithm MMDS, in Figure 6, finds a maximal matching and stabilizes for any graph of order n in at most $n + 1$ rounds, under the synchronous scheduler. Notice that in Rule DM2, a node i having a null pointer and no node k pointing to it may point to a neighbor j whose pointer is null, and thereby make a proposal of a match, provided that j has the minimum ID among the neighbors of node i whose pointer in null. The proof of correctness of this algorithm and its running time is considerably longer than that of Algorithm Hsu–Huang and is omitted.

In 2008 [26], Goddard, Hedetniemi, Jacobs, Srimani, and Xu proved that Algorithm MMDS also finds a maximal matching and stabilizes in at most $O(n)$

```
Algorithm MMDS: Maximal Matching - Distributed or Synchronous
DM1: if $(i \to null) \wedge (\exists j \in N(i) : j \to i)$
         then $i \to j$                                        [accept proposal]
DM2: if $(i \to null) \wedge (\forall k \in N(i) : k \not\to i) \wedge (\exists j \in N(i) : j \to null)$
         then $i \to \min\{j \in N(i) : j \to null\}$          [make proposal]
DM3: if $(i \to j) \wedge (j \to k$ where $k \neq i)$
         then $i \to null$                                     [withdraw proposal]
```

Fig. 6 Algorithm MMDS: Maximal matching, distributed, and synchronous scheduler

rounds and at most $O(n^3)$ time steps under a distributed scheduler. We provide their proof here, as it is instructive. At any point in the execution of Algorithm MMDS, under the distributed scheduler, let $M = \{\{i, j\} : i \leftrightarrow j\}$ denote the set of matched edges.

Recall that a *round*, as discussed by Dolev in [16], is a minimal sequence of time steps where every enabled node at the start of the round either makes a move or has its move disabled by the move of a neighbor; if the scheduler is *fair*, every round is guaranteed to finish.

Theorem 7 *If Algorithm MMDS stabilizes, then the set M is a maximal matching in the graph G.*

Proof It is clear that the set M is a matching since a node can only be matched with one other node; thus, no two edges can have a node in common. Assume that Algorithm MMDS has stabilized but M is not a maximal matching. Since Algorithm MMDS is stable, no node is enabled to execute Rule DM3. Therefore, every node either has a null pointer or is matched. Since M is not maximal, there must be two adjacent nodes, both of which have null pointers. But in this case, both nodes are enabled to execute Rule DM2, and the algorithm is not stable, which is a contradiction. □

Lemma 12 *After nodes i and j have been matched, neither node can make another move.*

Proof After nodes i and j have been matched, neither node i nor node j will have a null pointer and are therefore not enabled to execute Rule DM1 or DM2. Furthermore, since $i \leftrightarrow j$, neither node is enabled to execute Rule DM3. □

Lemma 13 *Consider a time step where at least one node executes Rule DM2 and makes a proposal, but no new match occurs. Then there exists some node that is proposed to but does not make a move.*

Proof Suppose that during a time step no new match occurs, and some node i executes DM2 and proposes to node j. If during this time step, node j does not make a move, then the lemma is true. Suppose, therefore, that node j makes a move. Since no match occurs, it must execute DM2 and propose to some node k. If node

k does not make a move, the lemma is proved. One can then follow the sequence of proposals: node i proposes to node j, which proposes to node k, which proposes to still some other node, etc. Since the graph is finite, either a node is reached which does not make move or there must exist a cycle of proposals.

But consider the node in the cycle having the largest ID, say u. Some node, say v, must propose to u in this cycle. But, in turn, some node w must have proposed to node v and node w must have a smaller ID than node u. Therefore, node v should have proposed to w, a contradiction.

Therefore, if during some time step, no match occurs but some node proposes to some other node, then there must be a node that receives a proposal but does not make a move. □

Lemma 14 *In the execution of Algorithm MMDS, there cannot be two consecutive rounds without a new match.*

Proof Let R be a round in which no new match occurs. If no more rounds are executed, then the algorithm is stable and the lemma is true, so assume that there is another round R'. We will show that a new match must occur.

Case 1. Assume that during round R no new match occurs, that is, no node executes DM1, but some node executes DM2. Then, by Lemma 13, at that time step, some node x is proposed, which does not make a move. It follows that x is enabled at the end of round R to execute DM1 by the end of the following round, creating a new match after round R', since every node pointed to must execute DM1 for some node pointing to it in any given round.

Case 2. Assume that during round R no node executes DM1 or DM2. That is, all moves in R are DM3. It follows that by the end of round R, every node is either matched, has a null pointer, or points to a neighbor that has a null pointer.

So, the first time step of the next round R' is an execution of DM1 or DM2. If a DM1 move is executed, then the lemma is proved. If no new match occurs, a DM2 move must be executed during R'. Then, by Lemma 13, some node x must be proposed to. But x was privileged at the start of the round and so must accept by the end of the round, creating a new match. □

Theorem 8 *Starting from an arbitrary state, Algorithm MMDS stabilizes in at most n rounds.*

Proof By Lemma 12, all matched nodes remain matched. By Lemma 14, there cannot be two consecutive rounds without a new match. Since every new match matches two nodes, the theorem follows. □

The following result shows that the number of time steps is a bit larger than for previous algorithms:

Lemma 15 *Algorithm MMDS stabilizes in at most $O(n^3)$ time steps under a distributed scheduler.*

Proof We know there are at most $O(n)$ time steps where a new match occurs.

We claim that there are at most n^2 time steps in which some node executes DM2 but no new match occurs. By Lemma 13, in each such time step there is some node that is pointed to but does not make a move. Each node can be pointed to only $n - 1$ times. Thus, the claim follows.

In between the above time steps, each node can execute DM3 at most once. Thus the total number of time steps between two matches is at most $O(n^2)$. Since there can be at most $n/2$ matches, it follows that the total number of time steps is at most $O(n^3)$. □

4.3 Other Self-Stabilizing Matching Algorithms

In 2001 [4] Blair, Hedetniemi, Hedetniemi and Jacobs present a self-stabilizing algorithm for finding a maximum, rather than the typical maximal, matching in an arbitrary tree.

In 2006 [29] Goddard, Hedetniemi and Shi present an anonymous self-stabilizing algorithm for finding a 1-maximal matching in a tree, and ring of length not divisible by 3. Their algorithm converges in $O(n^4)$ moves under a central daemon.

In 2007 [52], Manne, Mjelde, Pilard, and Tixeuil present a self-stabilizing algorithm for finding a maximal matching, using a distributed scheduler, which stabilizes in $O(|E|)$ rounds, improving on previous bounds of $O(n^2)$ and $O(\Delta|E|)$. Their algorithm also has the same running time as previous self-stabilizing, maximal matching algorithms, using central, distributed, and synchronous schedulers.

In 2009 [53], Manne, Mjelde, Pilard, and Tixeuil present a self-stabilizing algorithm for the maximal matching problem that improves the running time of the previous best algorithm for a distributed scheduler and at the same time meets the bounds of the previous best algorithms for the sequential and distributed fair schedulers. Their algorithm requires unique IDs at distance two and uses a Boolean variable at each node, which enables neighbors to communicate whether this node is already matched.

In 2015 [2], Asada and Inoue present a self-stabilizing algorithm for finding a 1-maximal matching, which is guaranteed to stabilize, under the anonymous model, with a fair central scheduler, but only when restricted to graphs having no cycles of lengths a multiple of 3; this includes all bipartite graphs, including grid graphs and trees. Since it stabilizes in $O(|E|)$ moves, it stabilizes in $O(n)$ moves for trees and cycles C_n, for n not a multiple of 3.

In 2016 [8], Cohen, Lefévre, Maâmra, Pilard, and Sohier present a self-stabilizing algorithm for finding a maximal matching in an anonymous network. The running time is $O(n^2)$ moves with high probability, under the adversarial distributed scheduler. Among all self-stabilizing algorithms using a distributed scheduler and the anonymous model, their algorithm provides the best known running time. Moreover, the previous best known algorithm working under the same scheduler and using IDs has an $O(m)$ running time, leading to the same order of growth than their anonymous algorithm. Although their algorithm does not make the assumption that a node can determine whether one of its neighbors points to it or to another node, it still has the same asymptotic behavior.

In 2016 [42], Inoue, Ooshita, and Tixeil present a self-stabilizing 1-maximal matching algorithm, using the unfair distributed scheduler. Their algorithm is restricted to graphs having no cycles of length a multiple of 3 and stabilizes in $O(|E|)$ moves. It also provides a 2/3-approximation of a maximum matching in these graphs, which improves on the 1/2-approximation guaranteed by any maximal matching.

Generalized *b*-Matchings

Given a graph $G=(V, E)$, let $E_i=\{(i, j)\in E\}$ denote the set of edges incident to a node i and let $d(i)=|E_i|$ denote the *degree* of node i.

Let $b: V\to\{0, 1, \ldots, n-1\}$ define a bound $b(i)$ on the number of edges that can be incident to node i. A subset $M\subseteq E$ is called a *b-matching* if for all $1\le i\le n$, $b(i)\le d(i)$. A b-matching M is called *maximal* if there does not exist a b-matching M' such that $M\subset M'$.

In 2003 [23], Goddard, Hedetniemi, Jacobs, and Srimani present a self-stabilizing maximal b-matching algorithm that stabilizes in $O(m)$ moves under an unfair central scheduler, independently of the particular b-values $b(i)$.

Self-Stabilizing Matching Approximation Algorithms

In 2011 [54], Manne, Mjelde, Pilard, and Tixeuil present the first self-stabilizing algorithm for finding a 2/3-approximation of a maximum matching in an arbitrary graph. Their algorithm stabilizes in at most $O(n^2)$ rounds, under a distributed scheduler. However, it might make an exponential number of moves.

In 2011 [68], Turau and Hauck present a more refined analysis of the running time of the first self-stabilizing algorithm for computing a 2-approximation of a maximum matching by Manne and Mjelde [51], who showed that their algorithm stabilizes in $O(2n)$ moves under a central scheduler, and in $O(3n)$ moves under a distributed scheduler. Turau and Hauck show that the Manne–Mjelde algorithm, in fact, stabilizes in $O(mn)$ moves under a central scheduler and, when modified, can stabilize in $O(mn)$ moves under a distributed scheduler.

In 2016 [10], Datta, Larmore, and Masuzawa present an anonymous-model, silent self-stabilizing algorithm for computing the maximum matching number of any tree. Their algorithm stabilizes in $O(n\cdot diam)$ moves, where *diam* is the diameter of the tree.

In 2017 [9], Cohen, Maâmra, Manoussakis, and Pilard present the first polynomial, self-stabilizing algorithm for finding a 2/3-approximation of a maximum matching in an arbitrary graph. The previous best known algorithm, by Manne et al. in 2011 [54], has a sub-exponential time running time under the distributed scheduler. The algorithm by Cohen et al. is an adaptation of the Manne et al. algorithm, works under the same scheduler, but stabilizes in $O(n^3)$ moves.

In 2017 [43], Inoue, Ooshita, and Tixeuil present an ID-based, self-stabilizing, 1-maximal matching algorithm that works under the distributed unfair scheduler for arbitrary graphs. It finds a 2/3-approximation of a maximum matching and stabilizes in $O(|E|)$ moves. The algorithm assumes that node IDs are distinct up to distance three.

The proposed algorithm closes the running time gap between two recent results: in 2016 [42], Inoue et al. present a 1-maximal matching algorithm that stabilizes in $O(|E|)$ moves but requires that the graph not contain a cycle of length a multiple of three; the algorithm of Cohen et al. in 2017 [9] stabilizes on arbitrary graphs but makes $O(n^3)$ moves. The Inoue–Ooshita–Tixeuil algorithm makes the same $O(|E|)$ moves but stabilizes on arbitrary graphs.

Strong Matchings

The definition of a maximal matching can be generalized to distance-k matchings. In particular, a *strong matching* is a matching $M \subseteq E$ having the property that no two edges in M are connected by an edge. This is equivalent to saying that for any two edges $e_1, e_2 \in M$, $d(e_1, e_2) > 1$. In 2005 [25], Goddard, Hedetniemi, Jacobs, and Srimani present an exponential running time, self-stabilizing algorithm for finding a maximal strong matching; this algorithm has only one rule; see also [24] in 2003 by the same authors.

5 Self-Stabilizing Dominating Set Algorithms

In this section, we present self-stabilizing algorithms for finding minimal dominating sets in arbitrary connected graphs $G = (V, E)$, first under a central scheduler, then under a synchronous scheduler, and finally under an unfair distributed scheduler. We conclude this section by presenting the first self-stabilizing algorithm for finding a minimal total dominating set.

A *dominating set* is a subset S of nodes such that $\forall i \in V : N[i] \cap S \neq \emptyset$, that is, every node i is either a member of S or is adjacent to a node j in S. A dominating set S is *minimal* if it does not contain a proper subset that is also a dominating set. It is important to know that a dominating set S is *minimal* if and only if every node $i \in S$ is either (i) not adjacent to any other vertex in S, in which case we say that node i is *its own private neighbor* or (ii) node i is the only vertex in S, which dominates some vertex j not in S, $j \in \overline{S}$, in which case we say that node j is an *external private neighbor* of node i.

5.1 Central Model Minimal Dominating Set Algorithm

The following Algorithm MDC, in Figure 7, is the first self-stabilizing algorithm for finding a minimal dominating set in an arbitrary graph, due to Hedetniemi, Hedetniemi, Jacobs, and Srimani in 2003 [32]; it assumes a central scheduler.

```
Algorithm MDC: Minimal Dominating - Central
D1: if $(x(i) = 0) \wedge (\forall j \in N(i))(x(j) = 0)$
        then $x(i) := 1$                                   [enter set]
D2: if $(x(i) = 1) \wedge (\nexists j \in N(i))(j \to i) \wedge (\exists k \in N(i))(x(k) = 1)$
        then $x(i) := 0$                                   [leave set]
P1: if $(x(i) = 1) \wedge (i \not\to null)$
        then $i \to null$                                  [no private neighbor]
P2: if $(x(i) = 0) \wedge (\exists$ unique $j \in N(i))((x(j) = 1) \wedge (i \not\to j))$
        then $i \to j$                                     [point to private neighbor]
P3: if $(x(i) = 0) \wedge (\exists$ more than one $j \in N(i))((x(j) = 1) \wedge (i \not\to null))$
        then $i \to null$                                  [no private neighbor]
```

Fig. 7 Algorithm MDC: Central Model [32]

The first rule D1 says that if a node i is currently not a member of the dominating set S ($x(i) = 0$) and no neighbor is in S, then it is enabled to enter S (by setting $x(i) = 1$).

Rule D2 says that if a node i is currently in S ($x(i) = 1$) but is not a private neighbor of any vertex (no node is pointing to i) and node i has a neighbor in S, then node i can leave the set S (by setting $x(i) = 0$).

Algorithm MDC has three kinds of pointer moves.

Rule P1 says that if node i is in S, its pointer should be null.

Rule P2 says that if node i is not in S and has a private neighbor j in S, then it should point to j.

Rule P3 says that if node i is not in S and has two or more neighbors in S, then its pointer should be null.

The proof of correctness of Algorithm MDC proceeds as follows. We will omit some of the details. Let S_t denote the set of nodes i having $x(i) = 1$ at time t.

Lemma 16 *If at any time t, S_t is not a minimal dominating set, then Algorithm MDC is not stable.*

Proof Suppose that Algorithm MDC is stable but the set S is not a dominating set. If S is not a dominating set, then there exists a node i not in S ($x(i) = 0$) and no neighbor of i is in S. This means that node i is enabled to execute D1, and thus, Algorithm MDC is not stable.

Assume therefore that S is a dominating set but is not a minimal dominating set. Thus, there exists a node i in S such that $S - \{i\}$ is a dominating set. This implies that node i must have a neighbor, say k in S, since it is not its own private neighbor, and node i does not have an external private neighbor.

There must also be a neighbor of i, say j, with $j \to i$, for if not, then node i is enabled to execute D2. Furthermore, $j \notin S$, else node j is enabled to execute P1. In addition, node j must not have another neighbor than i in S, else it is enabled to execute P3. Therefore, j is not in S, has exactly one neighbor in S, namely i, and therefore, node i has a private neighbor, contradicting our assumption that it has no private neighbor. □

Lemma 17 *If a node i executes D1, then it will never again make move D1 or D2.*

Proof If a node i executes D1 at some time t, then none of its neighbors are in S_t, meaning that for all neighbors $j \in N(i)$, $x(j) = 0$. As long as $x(i) = 1$, it could only execute D2, but it can only execute D2 if it has a neighbor j with $x(j) = 1$. □

Lemma 18 *A node i can execute at most two D1 or D2 moves.*

Proof If a node i makes its first move D1, then by Lemma 17, it will never make another D1 or D2 move. If node i makes its first move D2, then it can only make move D1, after which it can make no further D1 or D2 moves. □

Lemma 19 *There can be at most n consecutive pointer moves, P1, P2, or P3.*

Proof If a node i executes a pointer move P1, P2, or P3, and subsequently, there are no moves D1 or D2 made by any node, then node i is not enabled to execute P1, P2, or P3. Therefore, in any sequence of consecutive pointer moves, each node can only execute one pointer move. □

Lemma 20 *Algorithm MDC can make at most $n^2 + n$ moves.*

Proof By Lemma 18, there can be at most $2n$ moves D1 and D2. By Lemma 19, there can be at most n consecutive pointer moves between successive D1 or D2 moves. □

Theorem 9 *Algorithm MDC finds a minimal dominating set and stabilizes in $O(n^2)$ moves.*

Proof This follows from Lemmas 16 and 20. □

```
Algorithm MDS: Minimal Dominating - Synchronous
$S \equiv \{ i : x(i) = 1 \}$
SD1: if $(x(i) = 0) \wedge c(i) \neq |\{j \in N(i) : x(j) = 1\}|$
        then $c(i) = |\{j \in N(i) : x(j) = 1\}|$    [correct S-neighbor count]
SD2: if $(x(i) = 0) \wedge (|N(i) \cap S| = 0) \wedge (c(i) = 0) \wedge (\nexists j \in N(i)((j < i) \wedge (c(j) = 0))$
        then $x(i) := 1$    [enter set S]
SD3: if$(x(i) = 1) \wedge (|N(i) \cap S| > 0) \wedge (\forall j \in N(i)($if $x(j) = 0$ then $c(j) := 2)$
        then $x(i) := 0$ and
            $c(i) := \begin{cases} 1 & \text{if } |N(i) \cap S| = 1 \\ 2 & \text{otherwise} \end{cases}$    [leave set S]
```

Fig. 8 Algorithm MDS: Synchronous Model [26]

5.2 *Synchronous Model Minimal Dominating Set Algorithm*

We next present Algorithm MDS, in Figure 8, which is the first, synchronous model, self-stabilizing algorithm for finding a minimal dominating set, due to Goddard, Hedetniemi, Jacobs, Srimani, and Xu in 2008 [26]; an earlier 2003 version of this algorithm, by Xu, Hedetniemi, Goddard, and Srimani [71]. As with previous synchronous model algorithms, this algorithm assumes that all nodes have unique ID values. Again, $x(i) = 0$ means that node i is not in the dominating set S, and $x(i) = 1$ means that node i is in S. The variable $c(i) \in \{0, 1, 2\}$ keeps count of the number of neighbors of node i in the set S, where $c(i) = 2$ means that node i has 2 or more neighbors in S. Thus, if $|\{j \in N(i) : x(j) = 1\}| \geq 2$, then we set $c(i) = 2$. The value of $c(i)$ is not used if node i is a member of S.

Rule SD1 makes sure that a node i not in S has the correct value of $c(i)$.

Rule SD2 says that a node i can enter S if it has no neighbor in S, its current value $c(i) = 0$ is correct, and its ID is smaller than any neighbor j with $c(j) = 0$.

Rule SD3 says that a node i is enabled to leave S, by setting $x(i) = 0$ and setting a correct value of $c(i)$, if it has at least one neighbor in S, and according to the $c(j)$-values of its neighbors, it has no private neighbors in $\overline{S}$.

We first show the correctness.

Theorem 10 *If Algorithm MDS stabilizes, then the set $S = \{i : x(i) = 1\}$ is a minimal dominating set.*

Proof Suppose that Algorithm MDS is stable but S is not a dominating set. Thus, there is a node i such that $S \cap N[i] = \varnothing$. Among all such undominated nodes, let i have the minimum ID. Then, $x(i) = 0$. Further, since Algorithm MDS is stable, node i is not enabled to execute Rule SD1, and therefore, $c(i) = 0$ is correct.

Consider any neighbor $j \in N(i)$ whose ID j is smaller than i, $j < i$. Then, $x(j) = 0$, but by the choice of i, j must be dominated by a node in S. So, since node j is not enabled to execute Rule SD1, $c(j) > 0$ must be true. It follows then that node i is enabled to execute Rule SD2, a contradiction. Therefore, S is dominating.

Suppose that S is a dominating set but is not minimal. Then there is a node $i \in S$ such that $S - \{i\}$ is a dominating set. It follows that, for each $j \in N[i]$, we have $|N[j] \cap S| > 1$. If $j \in N(i) - S$, then by Rule SD1, since $x(j) = 0$, $c(j) = 2$. Hence, node i must be enabled to execute Rule SD3, a contradiction. Thus, S is a minimal dominating set. □

We next show that Algorithm MDS stabilizes.

Lemma 21 *If $x(i)$ changes from 0 to 1, then $x(i)$ will never again change.*

Proof If $x(i)$ changes from 0 to 1, then by Rule SD2, all nodes j in the neighborhood $N(i)$ must have $x(j) = 0$. By Rule SD2, only the node of i and j with smaller ID is enabled to execute Rule SD2 in the same time step according to the synchronous model, so $x(j)$ does not change in the same time step. Therefore, after this time step no neighbor of i is in S. After that, no neighbor j of i can enter S since there is at least one node (namely i) in $S \cap N(j)$, and i will not leave S since none of its neighbors are in S. □

Theorem 11 *Starting from any arbitrary state, Algorithm MDS stabilizes in at most* $4n+1$ *time steps under the synchronous scheduler.*

Proof By Lemma 21, each node can change its x-value at most twice. Therefore, there can be at most $2n$ changes of x-values on all n nodes. If there is no change in the x-value of any node during a time step, then the time step only involves corrections of c-values. The change in a c-value is determined only by x-values. Since we are working with the synchronous scheduler, there cannot be two consecutive time steps without a change in x-value. Therefore, the upper bound of execution time is $4n+1$ time steps. □

One can also show that Algorithm MDS converges under the distributed scheduler.

Theorem 12 *Algorithm MDS stabilizes with a minimal dominating set in at most* $5n$ *moves under the distributed scheduler.*

Proof We claim that every node can make at most 5 moves under a distributed scheduler.

Case 1. Assume that for a node i, $c(i)$ never changes to 0. By Lemma 21, if i executes Rule SD2 and changes from $x(i)=0$ to $x(i)=1$, then $x(i)$ will never change again. Thus, we may assume that after its first move, $x(i)=0$. So, apart from possibly its first move being Rule SD3, node i makes only Rule SD1 moves. Each such move changes the value of $c(i)$, which must oscillate between 1 and 2. Each 1-to-2 move is due to a neighbor entering S; once two neighbors have entered, i has two neighbors in S until the end of the algorithm, and so cannot move again. It follows that the longest possible sequence of changes for $c(i)$ is ?–2–1–2–1–2.

Case 2. Assume that $c(i)$ changes to 0 at some point. No neighbor enters before $c(i)$ goes to 0. So before the move $c(i)=0$, node i may make at most two moves (a leave move or a $c(i)=2$ move, perhaps followed by a $c(i)=1$ move). After $c(i)$ becomes 0, i may make either an enter move, or a $c(i)=1$ move, perhaps followed by a $c(i)=2$ move.

□

5.3 Distributed Model Minimal Dominating Set Algorithm

We next present a $4n$-move, self-stabilizing Algorithm MDD, in Figure 9, for finding a minimal dominating set using an unfair distributed scheduler, by Chiu, Chen, and Tsai in 2014 [7]. An earlier 2013 version of this algorithm by Chiu and Chen appears in [6]. For reasons of consistency with the notation used in our previous

```
Algorithm MDD: Minimal Dominating - Distributed
R1: if $(x(i) = 00) \wedge |N(i) \cap S| = 0 \wedge (\nexists j \in N(i))(x(j) = 00) \wedge (j < i))$
        then $x(i) := 1$                                    [enter $S$]
R2: if $(x(i) = 1) \wedge |N(i) \cap S| = 1 \wedge (\nexists j \in N(i))(x(j) = 01)$
        then $x(i) := 01$              [leave with unique private neighbor]
R3: if $(x(i) = 1) \wedge |N(i) \cap S| > 1 \wedge (\nexists j \in N(i))(x(j) = 01)$
        then $x(i) := 02$                  [leave with private neighbors]
R4: if $(x(i) = 00) \wedge |N(i) \cap S| = 1$
        then $x(i) := 01$           [ stay out with unique private neighbor]
R5: if $(x(i) = 01 \vee 00) \wedge |N(i) \cap S| > 1$
        then $x(i) := 02$               [stay out with private neighbors]
R6: if $(x(i) = 01 \vee 02) \wedge |N(i) \cap S| = 0$
        then $x(i) := 00$                 [stay out with no S-neighbors]
```

Fig. 9 Algorithm MDD: Distributed Model [7]

self-stabilizing algorithms, we will change the notation used by Chiu, Chen, and Tsai to be similar to that used in this chapter.

Algorithm MDD assigns to each node i a four-valued variable $x(i)$, which defines the local state of node i, such that $x(i) \in \{1, 00, 01, 02\}$. As before, at any time, $S = \{i : x(i) = 1\}$ and all such nodes are called *S-nodes*. All other nodes, those in states 00, 01, or 02, are called *Out nodes*, nodes in $\overline{S}$.

A node in state 1 is a member of S.

A node in state 00 is not in S and has no neighbor in S.

A node in state 01 is not in S but has a unique neighbor in S.

A node in state 02 is not is S but has at least two neighbors in S

The correctness of Algorithm MDD can be proved as follows: we omit the details.

Lemma 22 *If Algorithm MDD is stable, then S is a minimal dominating set.*

Lemma 23 *If a node executes R1, it will never make another move.*

Lemma 24 *A node can execute R6 at most once.*

Theorem 13 *Algorithm MDD stabilizes under an unfair distributed scheduler in at most* $4n - 2$ *moves.*

5.4 Minimal Total Dominating Set Algorithm

In this section, we present Algorithm MTDC, in Figure 10, which is the first self-stabilizing, minimal total dominating set algorithm, due to Goddard, Hedetniemi, Jacobs, and Srimani in 2005 [25]; it assumes the central scheduler model (see also [21] in 2003 by the same authors). Recall that a total dominating set of a graph

```
Algorithm MTDC: Minimal Total Dominating - Central
R1: if $(x(i) \neq pointedto(i)) \vee (p(i) \neq q(i))$

        then $x(i) := pointedto(i)$ and $p(i) = q(i)$
```

Fig. 10 Algorithm MTDC: Central Model [25]

$G = (V, E)$ is a set $S \subseteq V$ having the property that $N(S) = V$, which means that every node in $\overline{S}$ is adjacent to at least one node in S, and every node $v \in S$ is adjacent to another node $w \in S$, where $v \neq w$. This means that a graph G does not have a total dominating set if it has an isolated node. Therefore, we assume that G is a nontrivial connected graph. This algorithm is based on the fact that in any minimal total dominating set S, every node $v \in S$ has an external private neighbor.

In this algorithm, each node i has two variables:

(i) a Boolean x, where $x(i) = true$ if node i is in the minimal total dominating set S, and $x(i) = false$ if node i is not in S;
(ii) a pointer variable $p(i)$ such that if $p(i) = j$ then $i \to j$.

We need the following three definitions:

Definition 1 $minbr(i) = min\{j : j \in N(i)\}$, the neighbor of i having the smallest ID.

Definition 2 Boolean: $pointedto(i) = (\exists j \in N(i))(j \to i)$

Definition 3 $q(i)$ is the following pointer expression:

$$q(i) := \begin{cases} minbr(i) & \text{if} N(i) \cap S = \emptyset \\ j & \text{if} N(i) \cap S = \{j\} \\ null & \text{if} |N(i) \cap S| \geq 2 \end{cases}$$

The minimal total dominating set algorithm has but one rule.

This one rule says that if there is a node i pointing to a node j, then node j should become a member of the minimal total dominating set by setting $x(j) = true$. It also says that if a node i is in S ($x(i) = true$) and has no neighbors in S, then it should point to that node j in its neighborhood having the smallest ID, in which case node j must become a member of S by setting $node(j) = true$. A node having two or more neighbors in S sets its pointer to null, and if it has exactly one neighbor i in S, then it must point to that node, informing it that node i must remain in S.

The correctness of this algorithm can be shown as follows.

Lemma 25 *If Algorithm MTDC stabilizes, then the set $S = \{i : x(i) = true\}$ is a minimal total dominating set.*

Proof We first show that if Algorithm MTDC is stable, then S is a total dominating set. If S is not a total dominating set, then there must exist a node i such that $N(i) \cap S = \varnothing$. Since the algorithm is stable, it must be true that $p(i) = q(i) = minbr(i)$ and $minbr(i) \notin S$. But this implies that $pointedto(minbr(i))$ is true but $x(minbr(i)) = false$ so $minbr(i)$ is enabled to execute Rule R1, a contradiction.

Next, we must show that S is a minimal total dominating set. Assume that there is some node j such that $S - \{j\}$ is a total dominating set. Since $j \in S$, $x(j) = true$ and there must be some node $i \in N(j)$ for which $p(i) = j$. But since the algorithm is stable, it must be the case that since $p(i) = q(i)$, node j must be the unique neighbor of i in S. Thus, the removal of j from S will leave node i undominated, a contradiction. □

We say that *node i invites node j* if at some time t, node i has no neighbor in S and then executes Rule R1, causing $p(i) = q(i) = j$. In order for a node j to become a member of S, it must either be pointed to from an initial erroneous state or be invited to be a member by being pointed to by a node i in S.

In order to show that Algorithm MTDC stabilizes, we note that if the set S does not change its membership, then every node can only execute at most once, to correct its pointer value. We say that an *in-move* is a move that causes $x(i)$ to become true.

Lemma 26 *If during some time interval, there is no in-move by a node having a larger ID than some node i, then during this time interval node i can make at most two moves.*

Proof The first in-move made by a node i maybe have been because a neighbor $j \in N(i)$ happened to be pointing to i. A second in-move made by node i must be by invitation. So suppose that node i is invited by a neighbor, node j. Then j must be the smallest node in $N(i)$ since $minbr(j) = i$ and at the time of the invitation, all other nodes in $N(i)$ are not in S.

By our assumption that during some time interval there is no in-move by a node having a larger ID than node i, their membership in S does not change, so node j remains pointing to i throughout the time interval, and node i remains in S for the remainder of the time interval. □

Theorem 14 *Algorithm MTDC always stabilizes and finds a minimal total dominating set.*

Proof It suffices to show that every node makes only a finite number of in-moves. By Lemma 26, node n, which has the largest ID, makes at most two in-moves. During each of the three time intervals between such in-moves, using Lemma 26 again, node $n - 1$ can make at most two in-moves. By repeating this argument, it is easy to show that each node can make only finitely many in-moves during the intervals in which larger nodes are inactive. □

It can be shown, although we will not do so here, that in the worst case, Algorithm MTDC can make an exponential number of moves. This is our first example of a worst-case exponential time self-stabilizing algorithm. In the field of self-stabilizing algorithms, this is often acceptable, since on average, these algorithms can stabilize fairly quickly.

In 2014 [3], Belhoul, Yahiaoui, and Kheddouci present the first polynomial, self-stabilizing algorithm for finding a minimal total dominating set in an arbitrary graph.

They also generalize their algorithm to find a minimal total k-dominating set. Both of their algorithms stabilize in $O(mn)$ moves.

6 Other Self-Stabilizing Domination Algorithms

The reader is referred to an excellent 2010 survey by Guellati and Kheddouci [31] on self-stabilizing algorithms for finding maximal independent sets and minimal dominating sets. Several other papers have been published, which present self-stabilizing, minimal dominating set algorithms.

In 2003 [71], Xu, Hedetniemi, Goddard, and Srimani present a synchronous, self-stabilizing algorithm for finding a minimal dominating set, which stabilizes in $4n$ rounds, starting from any arbitrary global state. A round is defined as the period of time during which every node receives messages from all of its neighbors. The algorithm is general in the sense that it can stabilize with every possible minimal dominating set, as distinct from other self-stabilizing minimal dominating set algorithms, which stabilize only with independent dominating sets.

In 2015 [15], Ding, Wang, and Srimani present a synchronous model, self-stabilizing algorithm for finding a minimal dominating set, which finds a dominating set in just two rounds, but then takes additional n rounds to obtain a minimal dominating set.

Distance-*k* Dominating Sets
A dominating set $S \subseteq V$ is called a *distance-k dominating set* if for every node $j \in \overline{S}$ there exists a node $i \in S$ such that $d(i, j) \leq k$.

In 2008 [50], Lin, Huang, Wang, and Chen present a self-stabilizing algorithm for finding a minimal distance-2 dominating set in an arbitrary graph.

Distance-*k* Independent Dominating Sets
Given a graph $G = (V, E)$, a *distance-k independent dominating set*, also called a *maximal distance-k independent set*, is both a distance-k independent set and a distance-k dominating set. That is, given any node $v \in S$, no other node $u \in S$ is at distance k or less from v, and any node $w \in \overline{S}$ is at distance k or less from some node in S.

In 2014 [44], Johnen presents a self-stabilizing algorithm for finding a distance-k independent dominating set, under the unfair distributed scheduler, which stabilizes in at most $4n + k$ rounds. This is further discussed in a subsequent paper by the author in 2015 [45].

Disjoint Dominating Sets

A well-known theorem of Ore [56] states that in any graph having no isolated nodes, the complement $\overline{S}$ of every minimal dominating set is a dominating set. This means that any self-stabilizing algorithm for finding a minimal dominating set in effect finds two disjoint dominating sets, although the complement $\overline{S}$ need not be a minimal dominating set.

```
Algorithm 2DSC: 2 Dominating Sets - Central
S1: if $(x(i) = 0)(\forall j \in N(i))(x(j) = 0)$
        then $x(i) := 1$                                   [enter set $V_1$]
S2: if $(x(i) = 1) \wedge (\forall j \in N(i))(x(j) = 1)$
        then $x(i) := 0$                                   [enter set $V_0$]
```

Fig. 11 Algorithm 2DS: Central Model [32]

```
Algorithm Unfriendly - Central
S1: if $(C(i) = Blue) \wedge (B(i) > R(i))$
        then $C(i) = Red$
S2: if $(C(i) = Red) \wedge (R(i) > B(i))$
        then $C(i) = Blue$
```

Fig. 12 Algorithm Unfriendly: Central Model [34]

A *dominating bipartition* is a bipartition $V = V_0 \cup V_1$ into two disjoint dominating sets, neither of which needs to be a minimal dominating set.

In 2003 [32], Hedetniemi, Hedetniemi, Jacobs, and Srimani present the following very simple, self-stabilizing Algorithm 2DSC, in Figure 11, under the central scheduler, for creating a dominating bipartition.

This algorithm stabilizes in at most $n-1$ moves.

An *unfriendly partition* is a two-coloring of the nodes of a graph, say with colors Red and Blue, having the property that every node colored Red has at least as many Blue neighbors as it has Red neighbors, and every node colored Blue has at least as many Red neighbors as Blue neighbors. These partitions were originally defined and studied by Borodin and Koshtochka in 1977 [5], Aharoni, Milner, and Prikry in 1990 [1] and Shelah and Milner in 1990 [57]. They observed the following simple result.

Theorem 15 *Every finite connected graph G of order $n \geq 2$ has an unfriendly partition.*

It is immediate from the definition that every unfriendly partition $V = R \cup B$ is a bipartition into two dominating sets. In 2013 [34], Hedetniemi, Hedetniemi, Kennedy, and McRae present three self-stabilizing algorithms for finding an unfriendly partition, all using the central scheduler model. The first and simplest of these is Algorithm Unfriendly—Central, in Figure 12, where $C(i) \in \{Blue, Red\}$, $B(i) = |\{j : j \in N(i) \wedge C(j) = Blue\}|$ equals the number of Blue neighbors of node i and $R(i) = |\{j : j \in N(i) \wedge C(j) = Red\}|$ equals the number of Red neighbors of node i.

This algorithm stabilizes with an unfriendly partition in at most $m = |E|$ moves.

In 2015 [36], Hedetniemi, Jacobs, and Kennedy, using the distance-2 model, in which nodes can utilize state information of all nodes within distance-2 in making a move, present a self-stabilizing algorithm for finding one maximal independent set, and a second disjoint minimal dominating set. This algorithm stabilizes in $O(n^2)$ moves, which can be converted to a distance-1 model algorithm that makes $O(n^5)$

```
Algorithm Optimally Efficient - Central
S1: if $(x(i) = 0) \wedge (|N_0(i)| > |N_1(i)| + 1SNbr(i)$
        then $x(i) = 1$
S2: if $(x(i) = 1) \wedge (|N_2(i)| \geq |N_1(i)| - 1SNbr(i))$
        then $x(i) = 0$
```

Fig. 13 Algorithm Optimally Efficient: Central Model [66]

moves. They also present a distance-2, self-stabilizing algorithm for finding two disjoint minimal dominating sets, which also stabilizes in $O(n^2)$ moves. Two other self-stabilizing, unfriendly partition algorithms are also given, using the distance-2 model, where the objective is to increase the number of bicolored edges in the resulting unfriendly partition.

Optimally Efficient Sets
The *efficiency* of a set $S \subseteq V$ is defined as $\varepsilon(S) = |\{v \in \overline{S} : |N(v) \cap S| = 1\}|$, which equals the number of nodes not in S that are adjacent to exactly one node in S, or are dominated exactly once by the nodes in S. The *efficiency of a graph G* is defined to be $\varepsilon(G) = max\{\varepsilon(S) : S \subseteq V\}$. A set S is called *optimally efficient* if adding nodes cannot increase its efficiency, but deleting a node decreases its efficiency.

In 2012 [33], Hedetniemi, Hedetniemi, Jiang, Kennedy, and McRae present a self-stabilizing algorithm, under the central scheduler and the distance-2 model, to find a maximal optimally efficient set S in $O(n^2)$ distance-2 moves, or $O(n^5)$ distance-1 moves.

In 2013 [66], Turau presents two self-stabilizing algorithms, the first of which considerably improves on the algorithm mentioned above, by Hedetniemi et al. [33], for finding an optimally efficient set, and which stabilizes in $O(n^5)$ moves. Algorithm Optimally Efficient—Central, in Figure 13, by Turau, operating under the unfair distributed scheduler, stabilizes in just $O(nm)$ moves. Since this algorithm has just two rules, we present this algorithm. Once again we change the notation to be similar to that used throughout this chapter. This algorithm is designed for a central scheduler.

Let $N_k(i) = \{j \in N(i) : x(j) = 0 \wedge N_S(j) = k\}$ denote the neighbors of node i having $x(j) = 0$ and exactly k neighbors in S. Let $1SNbr(i) = 1$ if node i has exactly one neighbor in S, and $1SNbr(i) = 0$ otherwise.

The second algorithm in [66] is the first self-stabilizing algorithm, sequential or otherwise, which computes the exact value of the efficiency $\varepsilon(T)$ of a tree T.

7 Avenues for Further Study

As indicated in the introduction to this chapter, research on self-stabilizing domination algorithms has only been going on for about 20 years. Furthermore, researchers who design self-stabilizing, domination-related algorithms are relatively few in number. But given the relative ease of designing self-stabilizing domination algorithms, and their usefulness, it may be a fruitful area for both graph theorists and algorithms researchers.

In this closing section, we will list a number of areas of domination in graphs in which self-stabilizing algorithms have either not yet been designed or in which relatively little has been done. Keep in mind that for any one type of domination listed below, self-stabilizing algorithms can be designed with three types of schedulers: central, synchronous, and distributed; they can be ID-based or anonymous; and they can use distance-k knowledge for varying values of k. Thus, one always has a lot of design options. The reader is challenged to see if you can design a self-stabilizing algorithm for finding a minimal dominating set of any of the following types.

1. **Paired domination**. A dominating set $S \subseteq V$ is called a *paired dominating set* if the induced subgraph $G[S]$ has a perfect matching.
2. **Restrained domination**. A dominating set $S \subseteq V$ is called a *restrained dominating set* if the subgraph $G[\overline{S}]$ induced by $\overline{S}$ contains no isolated vertices, that is, every vertex in $\overline{S}$ has at least one neighbor in $\overline{S}$.
3. **Signed domination**. A function $f : V \rightarrow \{-1, 1\}$ is called a *signed dominating function* if for every vertex $v \in V$, $f(N[v]) = \Sigma_{w \in N[v]} f(w) \geq 1$.
4. **Minus domination**. A function $f : V \rightarrow \{-1, 0, 1\}$ is called a *minus dominating function* if for every vertex $v \in V$, $f(N[v]) = \Sigma_{w \in N[v]} f(w) \geq 1$.
5. **Odd domination**. A dominating set S is called an *odd dominating set* if for every vertex $v \in V$, $|N[v] \cap S|$ is an odd number.
6. **Secure domination**. A dominating set S is called a *secure dominating set* if for every vertex $v \in \overline{S}$ there exists an adjacent vertex $u \in S$ such that $S - \{u\} \cup \{v\}$ is a dominating set.
7. **Roman domination**. A function $f : V \rightarrow \{0, 1, 2\}$ is called a *Roman dominating function* if every vertex $v \in V$ with $f(v) = 0$ is adjacent to at least one vertex $w \in N(v)$ with $f(w) = 2$.
8. **Cost-effective domination**. A dominating set $S \subseteq V$ is called a *cost-effective dominating set* if every vertex $v \in S$ has at least as many neighbors in $\overline{S}$ as it has in S.
9. **Capacity-k domination**. A dominating set $S = \{v_1, v_2, \ldots, v_r\} \subseteq V$ is called a *capacity-k dominating set* if there exists a partition $V = \{V_1, V_2, \ldots, V_r\}$ such that for every $1 \leq i \leq r$, (i) $v_i \in V_i$, (ii) $V_i \subseteq N[v_i]$, and (iii) $|V_i| \leq k$.
10. **Connected domination**. This deserves some discussion. Given a graph $G = (V, E)$, a dominating set S is a *connected dominating set* if the subgraph $G[S]$ induced by S is connected. The problem of finding a minimal connected dominating set in a graph has been quite a challenge using the self-stabilizing paradigm. Indeed, is such an algorithm even possible, given that each node only has local knowledge of the graph?

    In 2010 [46], Kamei and Kakugawa present a self-stabilizing algorithm that approximates the connected domination number $\gamma_c(G)$ within a factor of at most $7.6\gamma_c(G) + 1.4$. Their algorithm stabilizes in $O(k)$ rounds, where k is the depth of an input breadth-first-search spanning tree of G.

In 2010 [30], Goddard and Srimani present two self-stabilizing algorithms for finding reasonably minimal connected dominating sets, but not guaranteed to be minimal, the second of which constructs a breadth-first spanning tree and then discards the leaves. Their algorithms run with anonymous nodes and with a distributed scheduler.

In 2012 [47], Kamei and Kakugama improve on their previous result by presenting a self-stabilizing algorithm, which approximates the connected domination number within a factor of 6, when restricted to unit-disk graphs.

In 2013 [48], Kamei, Kakugawa, Devismes, and Tixeuil present a self-stabilizing algorithm that approximates the maximum number of leaves in any spanning tree of a graph G within a factor of 1/3, meaning that it is guaranteed to have at least 1/3 of the maximum possible number of leaves. Their algorithm stabilizes in at most $O(n^2)$ rounds.

11. **Weakly connected domination**. Given a graph $G = (V, E)$, a dominating set S is called *weakly connected* if the subgraph induced by the edges having at least one node in S is connected. While the problem of designing a self-stabilizing algorithm for finding a minimal connected dominating set has proven to be difficult, in 2009 [67] Turau and Hauck present a self-stabilizing algorithm for finding a weakly connected dominating set. In 2015 [14], Ding, Wang, and Srimani present another self-stabilizing algorithm for finding a weakly connected dominating set.
12. $\{k\}$**-domination**. Given a graph $G = (V, E)$, let $f: V \rightarrow \{0, 1, \ldots, k\}$ be a function from the node set V to the set of integers $\{0, 1, \ldots, k\}$. For any subset $S \subset V$, define $f(S) = \Sigma_{v \in S} f(v)$. Such a function f is called a $\{k\}$*-dominating function* if for every node $i \in V$, $f(N[i]) \geq k$.

    In 2003 [20], Gairing, Hedetniemi, Kristiansen, and McRae present a five-rule, self-stabilizing algorithm for finding a minimal $\{k\}$-dominating function, which stabilizes in at most $(2n+1)(n+1)2^{n+2}$ moves. They also present a self-stabilizing algorithm for finding a $\{2\}$-dominating function, which stabilizes in at most $3n + 2m$ moves. This in turn provides a self-stabilizing algorithm, which stabilizes in $O(n)$ moves when restricted to planar graphs. A version of this self-stabilizing algorithm for $k = 2$ can be found in the 2002 PhD thesis of Kristiansen [49].
13. **Strong and weak domination**. Given a graph $G = (V, E)$, a dominating set S is called *strong* if for every node $j \in \overline{S}$ there exists a node $i \in N(j) \cap S$ whose degree $d(i)$ satisfies $d(i) \geq d(j)$. Similarly, a dominating set S is called *weak* if for every node $j \in \overline{S}$ there exists a node $i \in N(j) \cap S$ with $d(i) \leq d(j)$.

    In 2015, Neggazi, Guellati, Haddad, and Kheddouci [55] present a self-stabilizing algorithm for finding an independent strong dominating set, which operates under the unfair distributed scheduler and stabilizes in at most $n+1$ rounds. The authors show that using rules that choose nodes having larger degrees than their neighbors (strong domination) results empirically in smaller dominating sets than the maximal independent sets and minimal dominating sets found by previous self-stabilizing algorithms.

```
Algorithm Maximal Irredundant - Central
ENTER: if  $(x(i) = 0) \wedge (i$ is safe$)$
        then $x(i) = 1$
LEAVE: if  $(x(i) = 1) \wedge (i$ has no private neighbor$)$
        then $x(i) = 0$
```

Fig. 14 Algorithm Maximal Irredundant: Central Model [28]

14. ***k*-domination**. Given a graph $G = (V, E)$, a *k-dominating set* is a set $S \subseteq V$ having the property that for every node $j \in \overline{S}$, $|N(j) \cap S| \geq k$, that is, every node in $\overline{S}$ is dominated at least k-times. Self-stabilizing algorithms for finding a minimal 2-dominating set have been designed in 2007 [40] by Huang, Lin, Chen, and Wang, using a distributed scheduler, and in 2008 [39] by Huang, Chen, and Wang, using a central scheduler.
15. **Maximal irredundant sets**. Given a graph $G = (V, E)$, a set $S \subseteq V$ is called *irredundant* if for every node $i \in S$, either (i) there exists a node $j \in \overline{S}$ such that $N(j) \cap S = \{i\}$, in which case we say that node j is an *external private neighbor* of node i, with respect to the set S, or (ii) node i is not adjacent to any node in S, in which case we say that node i is *its own private neighbor*. An irredundant set S is *maximal* if for every node $j \in \overline{S}$, the set $S \cup \{j\}$ is not irredundant. This means that either node j does not have a private neighbor with respect to the set $S \cup \{j\}$ or there exists a node $i \in S$ such that i has a private neighbor with respect to S but does not have a private neighbor with respect to $S \cup \{j\}$. In this latter case, we say that adding node j to S *destroys* node i. We say that a node $j \in \overline{S}$ is *safe* with respect to a set S if adding it to S does not destroy any node in S and j has a private neighbor with respect to the set $S \cup \{j\}$.

    It is well known that every minimal dominating set is maximal irredundant. Thus, every self-stabilizing algorithm for finding a minimal dominating set also finds a maximal irredundant set. But since there are maximal irredundant sets that are not dominating sets, a true self-stabilizing, maximal irredundant algorithm had not been designed until 2008, when Goddard, Hedetniemi, Jacobs, and Trevisan [28] found a way to design such an algorithm using distance-4 knowledge. Their Algorithm Maximal Irredundant—Central, in Figure 14, has only two rules.

    The authors show that this algorithm finds a maximal irredundant set in $O(n^7)$ moves.
16. **Alliances in graphs**. Given a graph $G = (V, E)$, a set $S \subset V$ is a *global offensive alliance* if each node $i \in \overline{S}$ has $|N[i] \cap S| \geq |N[i] \cap \overline{S}|$, that is, every node in $\overline{S}$ has at least as many neighbors in S as it has in $\overline{S}$ plus itself. Similarly, a set $S \subset V$ is a *global defensive alliance* if each node $i \in S$ has $|N[i] \cap S| \geq |N[i] \cap \overline{S}|$, that is, every node in S has at least as many neighbors in S plus itself as it has in $\overline{S}$. A set S is a *global powerful alliance* if it is both a global defensive and global offensive alliance. Self-stabilizing global algorithms have

been designed in 2006 [70] by Xu, in 2013 [72] by Yahiaoui, Belhoul, Haddad, and Kheddouci, and in 2014 [60] by Srimani.

17. **Dominating sets with external private neighbors**. According to a well-known theorem of Bollobás and Cockayne, every graph G without isolated vertices has a minimum dominating set in which every vertex has an external private neighbor. So far, no self-stabilizing algorithm has appeared for a minimal dominating set having this property.
18. **Self-stabilizing algorithms on special classes of graphs**. In general, self-stabilizing algorithms are designed for arbitrary graphs. But it may well be possible to design algorithms that stabilize even faster on special classes of graphs, such as grid graphs, n-cubes, planar graphs, trees, and chordal graphs.
19. **Relative performance of self-stabilizing algorithms**. The algorithms we have presented in this chapter always have rules, whereby a node i enters the set S in question. But these in-moves can be modified in ways other than by comparing the IDs of nodes in the neighborhood of a node i, for example, you could give preference to allowing a node to make an in-move if its degree is either greater than or less than the degrees of nodes in its neighborhood. This would allow preference to be given to nodes of larger degree, or nodes of smaller degree. And this then means that you can get empirical data on the relative speeds and relative performance of self-stabilizing algorithms, namely, on average which algorithms stabilize more quickly, and when they stabilize, on average how large or how small are the maximal independent sets or minimal dominating sets that are found.
20. **Distance-*k* self-stabilizing algorithms**. Recent research has expanded the assumption that a node can only "see" the values of the variables of the nodes in its neighborhood $N(i)$; this is called the *shared-variable model*. But what if a node could see not only the values of the variables of its neighbors, but the neighbors of its neighbors? Several papers have been published on *distance-k knowledge* and how it is possible to convert a distance-k self-stabilizing algorithm to a standard distance-1 model algorithm, albeit at an increased cost in running time. These algorithms quickly become more sophisticated but enable other types minimal and maximal sets to be found, like maximal irredundant sets or k-packings (cf. Gairing et al. in 2004 [19], Goddard et al. in 2006 [27] and Goddard et al. in 2008 [28]).

Acknowledgment The author would like to thank the reviewers for the time it took them to carefully read this chapter and for their many suggestions to improve its presentation.

References

1. R. Aharoni, E.C. Milner, K. Prikry, Unfriendly partitions of a graph. J. Combin. Theory Ser. B **50**(1), 1–10 (1990)
2. Y. Asada, M. Inoue, An efficient silent self-stabilizing algorithm for 1-maximal matching in anonymous networks, in *WALCOM: Algorithms and Computation*. Lecture Notes in Computer Science, vol. 8973 (2015), pp. 187–198

3. Y. Belhoul, S. Yahiaoui, H. Kheddouci, Efficient self-stabilizing algorithms for minimal total k-dominating sets in graphs. Inform. Process. Lett. **114**(7), 339–343 (2014)
4. J.R.S. Blair, S.M. Hedetniemi, S.T. Hedetniemi, D.P. Jacobs, Self-stabilizing maximum matchings. Congr. Numer. **153**, 151–159 (2001)
5. A.V. Borodin, A.V. Kostochka, On an upper bound of a graph's chromatic number, depending on the graph's degree and density. J. Combin. Theory Ser. B **23**, 247–250 (1977)
6. W.Y. Chiu, C. Chen, Linear-time self-stabilizing algorithms for minimal domination in graph, in *Proceedings of the International Workshop on Combinatorial Algorithms, IWOCA 2013*. Lecture Notes in Computer Science, vol. 8288 (2013), pp. 115–126
7. W.Y. Chiu, C. Chen, S.-Y. Tsai, A $4n$-move self-stabilizing algorithm for the minimal dominating set problem using an unfair distributed demon. Inform. Process. Lett. **114**(10), 515–518 (2014)
8. J. Cohen, J. Lefévre, K. Maâmra, L. Pilard, D. Sohier, A self-stabilizing algorithm for maximal matching in anonymous networks. Parallel Process. Lett. **26**(4), 1650016 (2016)
9. J. Cohen, K. Maâmra, G. Manoussakis, L. Pilard, Polynomial self-stabilizing maximum matching algorithm with approximation ratio 2/3. *20th International Conference on Principles of Distributed Systems*, Art. No. 11, 17 pp., LIPIcs. Leibniz Int. Proc. Inform., 70, Schloss Dagstuhl. Leibniz-Zent. Inform., Wadern, 2017
10. A.K. Datta, L. Larmore, T. Masuzawa, Maximum matching for anonymous trees with constant space per process. *19th International Conference on Principles of Distributed Systems*, Art. No. 16, 16 pp., LIPIcs. Leibniz Int. Proc. Inform., 46, Schloss Dagstuhl. Leibniz-Zent. Inform., Wadern, 2016
11. F. Delbot, C. Laforest, S. Rovedakis, Self-stabilizing algorithms for connected vertex cover and clique decomposition problems, in *Principles of Distributed Systems*. Lecture Notes in Computer Science, vol. 8878 (2014), pp. 307–322
12. E.W. Dijkstra, Self-stabilizing systems in spite of distributed control. Commun. ACM **17**(11), 643–644 (1974)
13. E.W. Dijkstra, A belated proof of self-stabilization. J. Distributed Comput. **1**, 5–6 (1986)
14. Y. Ding, J.Z. Wang, P.K. Srimani, New self-stabilizing algorithms for minimal weakly connected dominating sets. Internat. J. Found. Comput. Sci. **26**(2), 229–240 (2015)
15. Y. Ding, J.Z. Wang, P.K. Srimani, Self-stabilizing algorithm for minimal dominating set with safe convergence in an arbitrary graph. Parallel Process. Lett. **25**(4), 1550011 (2015)
16. S. Dolev, *Self-Stabilization* (MIT Press, Cambridge, 2000)
17. M. Gairing, R.M. Geist, S.T. Hedetniemi, P. Kristiansen, A self-stabilizing algorithm for maximal 2-packing. Nordic J. Comput. **11**(1), 1–11 (2004)
18. M. Gairing, W. Goddard, S.T. Hedetniemi, D.P. Jacobs, Self-stabilizing maximal k-dependent sets in linear time. Parallel Process. Lett. **14**(1), 75–82 (2004)
21. M. Gairing, W. Goddard, S.T. Hedetniemi, P. Kristiansen, A.A. McRae, Distance-two information in self-stabilizing algorithms. Parallel Process. Lett. **14**, 387–398 (2004)
20. M. Gairing, S.T. Hedetniemi, P. Kristiansen, A.A. McRae, Self-stabilizing algorithms for $\{k\}$-domination, in *Proceedings of the Sixth Symposium on Self-Stabilization (SSS 2003)*, San Francisco. Lecture Notes in Computer Science, vol. 2074 (Springer, Berlin, 2003), pp. 49–60
21. W. Goddard, S.T. Hedetniemi, D.P. Jacobs, P.K. Srimani, A self-stabilizing distributed algorithm for minimal total domination in an arbitrary system graph, in *Proceedings of the Eighth IPDPS Workshop on Formal Methods for Parallel Programming: Theory and Applications*, Nice (2003)
22. W. Goddard, S.T. Hedetniemi, D.P. Jacobs, P.K. Srimani, Self-stabilizing protocols for maximal matching and maximal independent sets for ad hoc networks, in *Proceedings of the Fifth IPDPS Workshop on Advances in Parallel and Distributed Computational Models*, Nice (2003)
23. W. Goddard, S.T. Hedetniemi, D.P. Jacobs, P.K. Srimani, A robust distributed generalized matching protocol that stabilizes in linear time, in *Proceedings of the ICDCS International Workshop on Mobile Distributed Computing (MDC03)*, Rhode Island (2003)
24. W. Goddard, S.T. Hedetniemi, D.P. Jacobs, P.K. Srimani, Self-stabilizing distributed algorithm for strong matching in a system graph, in *HiPC 2003*, ed. by T.M. Pinkston, V.K. Prasanna, Lecture Notes in Computer Science, vol. 2913 (Springer, Berlin, 2003), pp. 66–73

25. W. Goddard, S.T. Hedetniemi, D.P. Jacobs, P.K. Srimani, Self-stabilizing global optimization algorithms for large network graphs. Internat. J. Dist. Sensor Netw. **1**, 329–344 (2005)
26. W. Goddard, S.T. Hedetniemi, D.P. Jacobs, P.K. Srimani, Z. Xu, Self-stabilizing graph protocols. Parallel Process. Lett. **18**(1), 189–199 (2008)
27. W. Goddard, S. Hedetniemi, D. Jacobs, V. Trevisan, Distance-k information in self-stabilizing algorithms, in *13th Colloquium on Structural Information and Communication Running Time (SIROCCO)*, (Chester, UK, 2006). Lecture Notes in Computer Science, vol. 4056 (2006), pp. 349–356
28. W. Goddard, S.T. Hedetniemi, D.P. Jacobs, V. Trevisan, Distance-k knowledge in self-stabilizing algorithms. Theoret. Comput. Sci. **399**, 118–127 (2008)
29. W. Goddard, S.T. Hedetniemi, Z. Shi, An anonymous self-stabilizing algorithm for 1-maximal matching in trees, in *Proceedings of the 2006 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA"6)*, Las Vegas, vol. II (2006), pp. 797–803
30. W. Goddard, P.K. Srimani, Anonymous self-stabilizing distributed algorithms for connected dominating set in a network graph, in *Proceedings of the International Multi-Conference Complexity, Informatics and Cybernetics: IMCIC 2010*
31. N. Guellati, H. Kheddouci, A survey on self-stabilizing algorithms for independence, domination, coloring, and matching in graphs. J. Parallel Distrib. Comput. **70**(4), 406–415 (2009)
32. 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**, 805–811 (2003)
33. S.M. Hedetniemi, S.T. Hedetniemi, H. Jiang, K.E. Kennedy, A. McRae, A self-stabilizing algorithm for optimally efficient sets in graphs. Inform. Process. Lett. **112**(16), 621–623 (2012)
34. S.M. Hedetniemi, S.T. Hedetniemi, K.E. Kennedy, A. A. McRae, Self-stabilizing algorithms for unfriendly partitions into two disjoint dominating sets. Parallel Process. Lett. **23**(1), 1350001 (2013)
35. S.T. Hedetniemi, D.P. Jacobs, K.E. Kennedy, Linear-time self-stabilizing algorithms for disjoint independent sets. Comput. J. **56**(11), 1381–1387 (2013)
36. S.T. Hedetniemi, D.P. Jacobs, K.E. Kennedy, A theorem of Ore and self-stabilizing algorithms for disjoint minimal dominating sets. Theoret. Comput. Sci. **593**, 132–138 (2015)
37. S.T. Hedetniemi, D.P. Jacobs, P.K. Srimani, Maximal matching stabilizes in time $O(m)$. Inform. Process. Lett. **80**, 221–223 (2001)
38. S.-C. Hsu, S.-T. Huang, A self-stabilizing algorithm for maximal matching. Inform. Process. Lett. **43**, 77–81 (1992)
39. T.-C. Huang, C.-Y. Chen, C.-P. Wang, A linear-time self-stabilizing algorithm for the minimal 2-dominating set problem in general networks. JISE J. Inf. Sci. Eng. **24**(1), 175–187 (2008)
40. T.-C. Huang, J.-C. Lin, C.-Y. Chen, C.-P. Wang, A self-stabilizing algorithm for finding a minimal 2-dominating set assuming the distributed demon model. Comput. Math. Appl. **54**(3), 350–356 (2007)
41. M. Ikeda, S. Kamei, H. Kakugawa, A space-time optimal self-stabilizing algorithm for the maximal independent set problem, in *PDCAT Proceedings of the Third International Conference on Parallel and Distributed Computing, Applications and Technologies* (2002), pp. 70–74
42. M. Inoue, F. Ooshita, S. Tixeuil, An efficient silent self-stabilizing 1-maximal matching algorithm under distributed scheduler without global identifiers, in *Stabilization, Safety and Security of Distributed Systems*. Lecture Notes in Computer Science, vol. 10083 (2016), pp. 195–212
43. M. Inoue, F. Ooshita, S. Tixeuil, An efficient silent self-stabilizing 1-maximal matching algorithm under distributed scheduler for arbitrary networks, in *Stabilization, Safety and Security of Distributed Systems*. Lecture Notes in Computer Science, vol. 10616 (2017), pp. 93–108
44. C. Johnen, Fast, silent self-stabilizing distance-k independent dominating set construction. Inform. Process. Lett. **114**(10), 551–555 (2014)

45. C. Johnen, Memory efficient self-stabilizing distance-k independent dominating set construction, in *Networked Systems*. Lecture Notes in Computer Science, vol. 9466 (2015), pp. 354–366
46. S. Kamei, H. Kakugawa, A self-stabilizing distributed approximation algorithm for the minimum connected dominating set. Int. J. Found. Comput. Sci. **21**(3), 459–476 (2010)
47. S. Kamei, H. Kakugawa, A self-stabilizing 6-approximation for the minimum connected dominating set with safe convergence in unit disk graphs. Theoret. Comput. Sci. **428**, 80–90 (2012)
48. S. Kamei, H. Kakugawa, S. Devismes, S. Tixeuil, A self-stabilizing 3-approximation for the maximum leaf spanning tree problem in arbitrary networks. J. Comb. Optim. **25**(3), 430–459 (2013)
49. P. Kristiansen, *New Results on the Domination Chain, Graph Homomorphisms, Alliancesand Self-stabilizing Algorithms*. Ph.D. Thesis, Department of Informatics, University of Bergen, 2002
50. J.-C. Lin, T.-C. Huang, C.-P. Wang, C.-Y. Chen, A self-stabilizing algorithm for finding a minimal distance-2 dominating set in distributed systems. JISE J. Inf. Sci. Eng. **24**(6), 1709–1718 (2008)
51. F. Manne, M. Mjelde, A memory efficient self-stabilizing algorithm for maximal k-packing, in *Eighth International Symposium Stabilization, Safety and Security of Distributed Systems, SSS* (November 2006, Dallas), ed. by A.K. Datta, M. Gradinariu. Lecture Notes in Computer Science, vol. 4280 (2006)
52. F. Manne, M. Mjelde, L. Pilard, S. Tixeuil, A new self-stabilizing maximal matching algorithm, in *Structural Information and Communication Complexity*. Lecture Notes in Computer Science, vol. 4474 (2007), pp. 96–108
53. F. Manne, M. Mjelde, L. Pilard, S. Tixeuil, A new self-stabilizing maximal matching algorithm. Theoret. Comput. Sci. **410**(14), 1336–1345 (2009)
54. F. Manne, M. Mjelde, L. Pilard, S. Tixeuil, A self-stabilizing 2/3-approximation algorithm for the maximum matching problem. Theoret. Comput. Sci. **412**(40), 5515–5526 (2011)
55. B. Neggazi, N. Guellati, M. Haddad, H. Kheddouci, Efficient self-stabilizing algorithm for independent strong dominating sets in arbitrary graphs. Int. J. Found. Comput. Sci. **26**(6), 751–768 (2015)
56. O. Ore, *Theory of Graphs*. American Mathematical Society Colloquium Publications (1962)
57. S. Shelah, E.C. Milner, Graphs with no unfriendly partitions, in *A Tribute to Paul Erdös*, ed. by A. Baker, B. Bollobás, A. Hajnal (Cambridge University Press, Cambridge, 1990), pp. 373–384
58. Z. Shi, A self-stabilizing algorithm to maximal 2-packing with improved complexity. Inform. Process. Lett. **112**(13), 525–531 (2012)
59. S. Shukla, D.J. Rosenkrantz, S.S. Ravi, Observations on self-stabilizing graph algorithms for anonymous networks, in *Proceedings of the Second Workshop on Self-Stabilizing Systems* (1995), pp. 7.1–7.15
60. P.K. Srimani, Self-stabilizing minimal global offensive alliance algorithm with safe convergence in an arbitrary graph, in *Theory and Applications of Models of Computation*, Lecture Notes in Computer Science, vol. 8402 (2014), pp. 366–377
61. G. Tel, Maximal matching stabilizes in quadratic time. Inform. Process. Lett. **49**, 271–272 (1994)
62. J.A.Trejo-Sánchez, J.A. Fernández-Zepeda, J.C. Ramírez-Pacheco, A self-stabilizing algorithm for a maximal 2-packing in a cactus graph under any scheduler. Int. J. Found. Comput. Sci. **28**(8), 1021–1045 (2017)
63. V. Turau, Linear self-stabilizing algorithms for the independent and dominating set problems using an unfair distributed scheduler. Inform. Process. Lett. **103**(3), 88–93 (2007)
64. V. Turau, Efficient transformation of distance-2 self-stabilizing algorithms. J. Parallel Distrib. Comput. **72**, 603–612 (2012)
65. V. Turau, Self-stabilizing vertex cover in anonymous networks with optimal approximation ratio. Parallel Process. Lett. **20**(2), 173–186 (2010)

66. V. Turau, Self-stabilizing algorithms for efficient sets of graphs and trees. Inform. Process. Lett. **113**(19–21), 771–776 (2013)
67. V. Turau, B. Hauck, A self-stabilizing algorithm for constructing weakly connected minimal dominating sets. Inform. Process. Lett. **109**(14), 763–767 (2009)
68. V. Turau, B. Hauck, A new analysis of a self-stabilizing maximum weight matching algorithm with approximation ratio 2. Theoret. Comput. Sci. **412**(40), 5527–5540 (2011)
69. V. Turau, B. Hauck, A fault-containing self-stabilizing $(3 - \frac{2}{\Delta+1})$-approximation algorithm for vertex cover in anonymous networks. Theoret. Comput. Sci. **412**(33), 4361–4371 (2011)
70. Z. Xu, *Self-stabilizing Protocols for Distributed Systems*. Ph.D. thesis, Clemson University, 2002
71. Z. Xu, S.T. Hedetniemi, W. Goddard, P.K. Srimani, A synchronous self-stabilizing minimal dominating protocol in an arbitrary network graph, in *Proceedings of the Fifth International Workshop on Distributed Computing (IWDC2003)*. Lecture Notes in Computer Science, vol. 2918 (2003), pp. 26–32
72. S. Yahiaoui, Y. Belhoul, M. Haddad, H. Kheddouci, Self-stabilizing algorithms for minimal global powerful alliance sets in graphs. Inform. Process. Lett. **113**(10–11), 365–370 (2013)