國 立 交 通 大 學

應 用 數 學 系

博 士 論 文

分散式系統之自我穩定極小控制集演算法與凱氏圖之正負號星控制數

Self-stabilizing Minimal Dominating Set Algorithms of Distributed Systems and the Signed Star Domination Number of Cayley Graphs

研 究 生：邱鈺傑

指導教授：陳秋媛

中 華 民 國 一 百 零 三 年 七 月

分散式系統之自我穩定極小控制集演算法
與凱氏圖之正負號星控制數

Self-stabilizing Minimal Dominating Set Algorithms of Distributed Systems and the Signed Star Domination Number of Cayley Graphs

研 究 生：邱鈺傑 Student: Well Y. Chiu
指導教授：陳秋媛 Advisor: Chiuyuan Chen

國 立 交 通 大 學
應 用 數 學 系
博 士 論 文

A Dissertation
Submitted to Department of Applied Mathematics
College of Science
National Chiao Tung University
in Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy
in
Applied Mathematics
July 2014
Hsinchu, Taiwan, Republic of China

中 華 民 國 一 百 零 三 年 七 月

摘要

圖論的控制集問題的研究始於 1960 年代。一個分散式系統 (例如：一個隨意網路) 可以用一個無向簡單圖 $G = (V, E)$ 來表示，其中 V 表示點集，而 E 表示點跟點之間的聯結關係。圖 G 的點集 V 的子集合 D 被稱爲是控制集，若此子集 D 具有性質：V 中的任一元素 v 屬於 D 或與 D 中的元素相鄰。如果一個控制集不真包含另一控制集，則稱其爲極小控制集 (簡記爲 MDS)。極小控制集在無線網路中的應用之一是使所需的資源中心保持爲較少數目。

自我穩定是一個可用於設計分散式系統容忍暫時性錯誤的一個概念，並且是在 1974 年由 Dijkstra 提出。一個分散式系統是自我穩定的，如果它滿足：不論初始組態爲何，系統總是能保證在有限時間內達到一個合法的 (正確的) 組態。此處所稱的系統組態是由系統中所有節點的狀態所組成。一個自我穩定演算法由一些規則所組成，每個規則都有其觸發條件及其對應動作，該動作能通過更新節點的變數來改變節點的狀態。每執行一個規則稱爲是一步。本論文所提出的演算法的效能皆是以執行總步數來計算。自我穩定演算法有許多不同的執行模式，且執行模式是以排程器的概念來呈現。排程器分爲公平的及不公平的兩種。眾所皆知，不公平的分散式排程器比其他類型的排程器更貼近實際使用狀況。

令 n 表示給定的分散式系統的節點數。在 2007 年，Turau 對於 MDS 問題，提出了不公平的分散式排程器下的第一個線性時間的自我穩定演算法，此演算法在最多 $9n$ 步之後穩定。在 2008 年，Goddard 等人改進了上述演算法並做到最多 $5n$ 步。我們將在本論文中提出一個採用不公平分散式排程器下最多 $4n$ 步的自我穩定 MDS 演算法。

理想中，MDS 演算法是希望能做到 MDS-靜止的，亦即，若分散式系統的初始組態已經是一個 MDS，則演算法將不執行任何動作。值得注意的是，在正常模式中，一個節點只能取得其一步內鄰居的資訊，我們稱這些資訊爲 1 步資訊。不幸的是，在本論文中，我們將證明：1 步資訊不足以建立一個 MDS-靜止的演算法。在本論文中，我們將討論此一問題，並針對自我穩定 MDS 演算法提出一個新的性能評量，稱爲穩定性。我們推廣這部份的結果至其他自我穩定演算法，並將自我穩定演算法分類爲四個層次。特別的是，我們將證明：在 2 步資訊模式下，可建構出自我穩定 MDS-靜止的演算法，且其穩定時間之上界爲 $2n$ 步。

在 2005 年，Xu 提出了圖的帶正負號星控制集的概念，在 2007 年，Wang 給出了完全圖的帶正負號控制數，詳細定義請參見本論文的第五章。在 2010 年，Atapour 等人定義了帶正負號星控制劃分數，並利用完全圖具有可完全 1-因子分解或可完全漢米爾頓圈分解的性質，計算出完全圖的帶正負號星控制劃分數。在 2012 年，我與印度學者 Chelvam 等人合作，定義了有向圖的帶正負號星控制數，給出了全部有向凱式圖的帶正負號星控制數，與無向凱式圖的帶正負號星控制數的部份結果，並推廣這些結果至可完全 $\{2,1\}$-因子分解的正則圖。

Abstract

The study of the domination problem in graph theory began in the nineteen-sixties. A distributed system such as an ad hoc network can be modeled by an undirected simple graph $G = (V, E)$, where V represents the set of nodes (i.e., processes) and E represents the set of interconnections between processes of the distributed system. A subset D of the vertex set V of G is a dominating set if each vertex $v \in V$ is either a member of D or adjacent to a vertex in D. A dominating set of G is a minimal dominating set (MDS) if none of its proper subsets is a dominating set of G. An MDS has an application of clustering in wireless networks and is maintained for minimizing the number of required resource centers.

Self-stabilization is a concept of designing a distributed system for transient fault toleration and was introduced by Dijkstra in 1974. A distributed system is self-stabilizing if, regardless of its initial configuration, the system is guaranteed to reach a legitimate (i.e., correct) configuration in a finite time. Here the system configuration consists of the state of every process. A self-stabilizing algorithm comprises a collection of rules and each rule has a trigger precondition and an action. The action changes the state of the node by updating its variables. An execution of a rule is called a move. The performance of the proposed algorithms of this thesis is measured by the total number of moves executed by an algorithm. Various execution models have been used in self-stabilizing algorithms and these are encapsulated with the notion of daemons. A daemon can be fair or unfair. It is well-known that an unfair distributed daemon is more practical than other types of daemons.

Let n denote the number of nodes (processes) in a given distributed system. In 2007, Turau proposed the first linear-time self-stabilizing algorithm for the MDS problem under an unfair distributed daemon; this algorithm stabilizes in at most $9n$ moves. In 2008, Goddard et al. improved the result to a $5n$-move algorithm. It is interesting to develop an algorithm that takes less moves than the best known result—$5n$ moves using an unfair distributed daemon. In this thesis, we will present a $4n$-move self-stabilizing MDS algorithm using an unfair distributed daemon.

It is desired that an MDS algorithm is MDS-silent, which means that if the original configuration of the distributed system is already an MDS, then the algorithm should not make any move. Note that in the normal model, a node can only access the information of its 1-hop neighbors and we call such information distance-1 information. Unfortunately, in this thesis we will prove that distance-1 information is not sufficient for building up an MDS-silent algorithm for a distributed system. Therefore it is impossible for an algorithm to be MDS-silent under a normal model, in which only distance-1 information is accessible. What will happen if a node can access the information of its k-hop neighbors for $k \geq 2$? In this thesis, we will discuss this problem and propose a new performance measure, called stableness, for self-stabilizing MDS algorithms. We also generalize this result to categorize all self-stabilizing algorithms into four levels. In particular, we will show that a self-stabilizing MDS-silent algorithm can be built up under the distance-2 model and the stabilizing time is upper bounded by $2n$.

Let G be a simple connected graph with vertex set $V(G)$ and edge set $E(G)$. A function $f : E(G) \to \{-1, 1\}$ is called a signed star dominating function (SSDF) on G if $\sum_{e\in E(v)} f(e) \geq 1$ for every $v \in V(G)$, where $E(v)$ is the set of all edges incident to v. The signed star domination number of G is defined as $\gamma_{SS}(G) = \min\{\sum_{e\in E(G)} f(e)|\ f$ is an SSDF on $G\}$. Let D be a finite digraph with vertex set $V(D)$ and arc set $A(D)$. For each vertex $v \in V(D)$, let $A(v)$ be the set of all out-going arcs from v. By replacing $E(v)$ by $A(v)$, one can define SSDF on D and $\gamma_{SS}(D) = \min\{\sum_{a\in A(D)} f(a) \mid f$ is an SSDF on $D\}$. Let Γ be a finite nontrivial group and S be a nonempty subset of Γ. The Cayley digraph $Cay_D(\Gamma, S)$ is the digraph whose vertices are the elements of Γ, and there is an arc from α to $\alpha\sigma$ whenever $\alpha \in \Gamma$ and $\sigma \in S$. Let Ω be a symmetric generating subset of nonidentity elements of Γ. The Cayley graph $Cay(\Gamma, \Omega)$ corresponding to Γ and Ω is the ordinary graph with vertex set Γ and edge set $E = \{\{\alpha, \alpha\sigma\} \mid \alpha \in \Gamma, \sigma \in \Omega\}$. In this thesis, we obtain exact values for the signed star domination number of all Cayley digraphs $Cay_D(\Gamma, S)$ and certain classes of Cayley graphs $Cay(\Gamma, \Omega)$, which is later generalized to $\{2, 1\}$-factorable graphs. Note that these solutions are from a joint work with Chelvam and Kalaimurugan.

Acknowledgments

由衷感謝兩大貴人：我的指導教授陳秋媛老師與我未來的老婆陳怡樺。

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Graph theory has multiple applications in the real world. In this thesis, we consider a distributed system whose topology is represented by an undirected simple graph $G = (V, E)$, where V represents the set of nodes (i.e., processes) and E represents the set of edges (i.e., the interconnections between processes) of the distributed system. We discuss the problem of developing efficient self-stabilizing minimal dominating set algorithms, the existence of self-stabilizing minimal dominating set algorithms with respect to different levels of stableness, and finding the signed-star domination number of graphs. Throughout this thesis, we use the terms "vertex" and "node" interchangeably, and we use the terms "distributed system" and "graph" interchangeably. Also, unless otherwise specified, all graphs are assumed to be simple.

1.1 Basic graph definitions

A graph G is a nonempty finite set V along with a finite set E of 2-element subsets of V. The elements of V are called *vertices* and the elements of E are called *edges*. The number of elements in a set S is called the *cardinality* of S and is denoted by $|S|$. We use n and

m to denote the *order* and the *size* of a graph, respectively; that is, $n = |V|$ and $m = |E|$. An edge with endpoints u and v is denoted as uv. Two vertices u and v in V are said to be *adjacent* in G, denoted by $u \sim v$, if there exists an edge $uv \in E$. Let $v \in V$. A vertex u is a *neighbor* of v if they are adjacent. The *open neighborhood* (or shortly, *neighborhood*) of v is the set of vertices to which v is adjacent; that is, $N(v) := \{u \in V \mid u \sim v\}$. The *degree* of a vertex v is defined as $\deg(v) := |N(v)|$. We use the notation $N[v] = N(v) \cup \{v\}$ to denote the *closed neighborhood* of v. The definition of the maximum degree Δ of G is $\Delta(G) := \max\{\deg(v) \mid v \in V\}$. We define the *diameter* of G to be $\max\{d(u,v) \mid u, v \in V\}$ and the *radius* of G to be $\min\{\max\{d(u,v) \mid v \in V\} \mid u \in V\}$.

The length of a path or a circle is defined to be the number of edges in the path or the circle. The distance between any two vertices u and v in G, denoted as $d(u,v)$, is the length of a shortest path between u and v in G. A vertex u is called a *k-neighbor* of a vertex v if $d(u,v) = k$. The (*open*) *k-neighborhood* $N_k(v)$ of vertex v is the set of vertices u such that $1 \le d(u,v) \le k$. The *closed k-neighborhood* $N_k[v]$ of vertex v is $N_k(v) \cup \{v\}$. For a vertex set S, $N_k(S) = \bigcup_{i \in S} N_k(v)$. Conventionally, the subscript k is omitted if $k = 1$, as a neighbor of v or the neighborhood $N(v)$.

To *color* the vertices in a graph $G = (V,E)$ is to give each vertex a positive integer color value in such a way that no two adjacent vertices get the same color. In many practical considerations, it is desirable to minimize the number of colors used. If at most k colors are used, the result is called a *k-coloring*. The smallest possible positive integer k for which there exists a k-coloring of G is called the *chromatic number* of G and is denoted as $\chi(G)$.

Given an undirected graph $G = (V,E)$, a *matching* is defined to be a subset $M \subseteq E$ such that for all nodes $v \in V$ at most one edge of M is incident with v. A matching M is *maximal* if there does not exist another matching M' such that $M \subset M'$. A subset $M \subseteq E$ is called a *k-matching* of G if $|E(v) \cap M| \le k$ for all nodes $v \in V$, where $E(v)$ is the set of edges incident from v. Note that a 1-matching is a simple matching.

A *vertex cover* for a graph $G = (V, E)$ is a subset $C \subseteq V$ such that, for each edge $uv \in E$, at least one of the two endpoints u and v belongs to C.

1.2 Graph domination and related definitions

Although the study of the graph domination problem began in the nineteen-sixties, the first paper which addresses the queens dominating chessboard problem dates back to 1862. Given an undirected graph $G = (V, E)$, a *dominating set* D is a subset of V such that $N[v] \cap D \neq \emptyset$ for every $v \in V$. We say a node in D *dominates* its neighbors and a node in $V \backslash D$ is *dominated* if it has a neighbor in D. Nodes in the dominating set D are called *dominators*, and nodes not in D are called *dominatees*.

A dominating set D of G is a *minimal dominating set* (MDS) if D does not properly contain another dominating set of G. The *MDS problem* is that of finding a minimal dominating set for any given undirected graph G and this is the problem discussed in this thesis. A dominating set D of G is a *connected dominating set* (CDS) if the subgraph of G induced by D is connected; minimality is defined similarly. The problem of finding a minimum connected dominating set is known to be NP-complete [22].

Dominating sets are closely related to independent sets. In an undirected graph $G = (V, E)$, a subset $I \subseteq V$ is *independent* if no two nodes in I are adjacent. An independent set I of G is a *maximal independent set* (MIS) if it is not a proper subset of any independent set of G. It is well-known that a maximal independent set of G is a dominating set of G. Therefore a maximal independent set of G is also called an *independent dominating set* of G. The *independent set problem* is the optimization problem of finding an independent set of maximum cardinality in a graph. The independent set problem is NP-complete [22].

Given a non-negative integer k, we define a *k-cluster* of G to be a nonempty subgraph of G of radius at most k. If C is a k-cluster of G, we say that $v \in V(C)$ is a *clusterhead* of

C if, for any $u \in V(C)$, there is a path of length at most k in C from v to u. We define a *k-clustering* of G to be a set $\{C_1, \ldots, C_\ell\}$ of k-clusters of G such that every vertex $v \in V$ lies in exactly one of the C_i. A set of vertices $D \subseteq V$ is a *k-dominating set* of G if, for every $v \in V$, there exists $u \in D$ such that $d(u, v) \leq k$. Building a k-dominating set in a graph is useful because it allows to split the graph into k-clusters. We say that a k-dominating set D is *minimum* if no k-dominating set of G has fewer dominators than D. The problem of finding a minimum k-dominating set is known to be NP-hard [22].

Now, we define the problem of computing an MDS in distributed systems. Let $G = (V, E)$ be a graph that represents a distributed system. By $v.id$, we denote the process identifier of v for each process v of the distributed system. In discussing the process identifier, we can use v to denote $v.id$ when it is clear from contexts. For a distributed system, the *minimal dominating set problem* is defined as follows:

- For each process $v \in V$, its identifier and its neighbors are given as an input.
- Each process $v \in V$ must decide its decision d_v as an output.
- The set $\{v \in V : d_v = 1\}$ must be an MDS.

The decision d_v of v can be encoded by the local state q_v of v in an arbitrary way by an algorithm. Formally, $d_v \equiv \mathrm{Dec}(q_v)$ for some decoding function Dec on the local states defined by an algorithm.

In summary, given an undirected graph $G = (V, E)$, a subset S of V is:

- **independent** if for each pair $i, j \in S$, $ij \notin E$;
- **dominating** if for every $i \in V$, either $i \in S$ or $i \in N(S)$;
- **maximal independent** if S is independent and any subset properly containing S is not independent;

- **minimal dominating** if S is dominating and no proper subset of S is dominating.

1.3 Applications of graph domination

A minimal dominating set needs to be maintained to optimize the number and location of resource centers in a network [35]. Some applications of this area include school bus routing, computer communication networks, resource allocation schemes, radio stations with limited broadcasting range, etc. An important concept in the study of a vertex subset problem is the difference between a "minimum" and a "minimal" set with a given property. To find a minimum cardinality of dominating set is hard, but we can easily find a minimal dominating set in polynomial time in a distributed manner.

Specifically, a minimal dominating set can be usefully applied in wireless sensor networks. One way to find a connected dominating set is to firstly find a minimal dominating set and then connect each dominator by including more nodes into the dominating set. A connected dominating set of a communication graph can serve as the virtual backbone for supporting the communication between sensors in a wireless sensor network. This is because its domination property ensures that every node is either in the backbone or adjacent to a node in the virtual backbone, and its connectivity property guarantees that any two nodes can send messages to each other via a series of intermediate nodes in the virtual backbone. In a connected component of a wireless sensor network's communication graph, sensors are linked with one another so that they can interchange messages and deliver messages along a simple path to make the communications between non-adjacent sensors become possible.

1.4 Thesis outline

This thesis is organized as follows. Chapter 2 gives the fundamentals of self-stabilizing algorithms on a distributed system. Self-stabilization is a concept of designing a fault-tolerant distributed system for a finite number of transient faults, such as message loss and memory corruption. The concept of self-stabilization was first proposed by Dijkstra in 1974 [15]. The *configuration*, also called the *global state*, of a distributed system consists of the state of every node and the content of every communication channel. Given an illegal starting configuration, such a system has to be able to reach a legal configuration in a finite time. Once in a legal configuration, the system may only move to another legal configuration if there is no external interference. The concept of self-stabilization has been proven to be an effective and cost-effective paradigm for localized state-based computation to implement distributed algorithms, especially in networks with resource-constrained nodes like sensor networks or ad hoc networks. The objective of self-stabilization is to recover the system from failure in a reasonable time and without intervention by any external agency.

The purpose of Chapter 3 is to discuss the problem of designing efficient self-stabilizing algorithms for the minimal dominating set problem. We consider a distributed system whose topology is represented by an undirected graph G. The MDS problem is to find a minimal dominating set for the given undirected graph G. Let n denote the number of nodes in the system. The main contribution of this chapter is to propose a $(4n-2)$-move self-stabilizing algorithm for the MDS problem under an unfair distributed daemon. We also compare our algorithm with other self-stabilizing algorithms in this chapter.

The purpose of Chapter 4 is to discuss the levels of stableness of self-stabilizing algorithms for the MDS problem. Obviously, a node in a distributed system has limited information about the whole system. Usually, a node can only access the information of nodes in its 1-neighborhood. Let k be a positive integer. We say a distributed system is in the *distance-k model* if a node in the system can access the information of nodes in its k-neighborhood. For convenience, we say a distributed system is in the *normal model* if it is in the distance-1 model. What will happen if a node can access the information of nodes in its k-neighborhood for $k \geq 2$? In this chapter, we discuss the above problem and propose a new performance measure, called *stableness*, for self-stabilizing MDS algorithms. We define *MDS-silent* algorithms and *MDS-stable* algorithms, and we classify four levels of stableness for self-stabilizing algorithms.

It is desired that an MDS algorithm is MDS-silent, meaning that if the original configuration is already an MDS, then the algorithm should not make any moves. In Chapter 5, we discuss the problem of designing a self-stabilizing MDS-silent algorithm. In the distance-2 model, a node can instantaneously access the information of all nodes within distance two from it. The main contribution of this chapter is to propose a $(2n-1)$-move self-stabilizing MDS-silent algorithm for the MDS problem under an unfair distributed daemon in the distance-2 model.

The purpose of Chapter 6 is to find out the signed star domination number (a variant of the domination number) of Cayley digraphs and Cayley graphs. A function $f : E(G) \to \{-1, 1\}$ is called a *signed star dominating function* (SSDF) on G if $\sum_{e\in E(v)} f(e) \geq 1$ for every $v \in V(G)$, where $E(v)$ is the set of all edges incident with v. The *signed star domination number* of G is defined as $\gamma_{SS}(G) = \min\{\sum_{e\in E(G)} f(e) |\ f \text{ is an SSDF on } G\}$. In Chapter 6, we obtain exact values for the signed star domination number for certain classes of Cayley digraphs and Cayley graphs.

Concluding remarks are given in Chapter 7.

Chapter 2

Self-stabilization

The concept of self-stabilization was first proposed by Dijkstra [15] in 1974. This concept has been proven to be an effective and cost-effective paradigm for localized state-based computation to implement distributed algorithms, especially in networks with resource-constrained nodes like sensor or ad hoc networks. The objective of self-stabilization is to recover the system from failure in a reasonable time and without intervention by any external agency. Self-stabilization is based on two basic ideas: (i) the code executed by a node is incorruptible (as if written in a fault-resilient memory) and transient faults affect only data values; (ii) the goal system behavior can be checked by evaluating some predicates of the system state variables.

This chapter is organized as follows. Section 2.1 gives the early draft of self-stabilizing systems. Section 2.2 states the fundamental concepts of self-stabilizing systems. Section 2.3 introduces various execution models which are encapsulated with in the notion of daemons and fairness. Section 2.4 gives a brief review of self-stabilizing algorithms on graphs (spanning tree construction, independent sets, dominating sets, and so on). An application of self-stabilizing algorithms to wireless sensor networks is given in Section 2.5.

2.1 The early draft of self-stabilizing systems

Self-stabilization is a concept of designing distributed systems for transient fault toleration. The original idea of self-stabilization was introduced by Dijkstra in a seminal paper in 1974 [15]. The earliest definition of a self-stabilizing system requires that:

(i) in each legitimate (i.e., legal) state, one or more privileges will be present;

(ii) in each legitimate state, each possible move will bring the system again in a legitimate state;

(iii) each privilege must be present in at least one legitimate state; and

(iv) for any pair of legitimate states, there exists a sequence of moves transferring the system from the one into the other.

Dijkstra considered machines placed in a ring, and had chosen the legitimate configuration in which exactly one privilege is present. Imagine there is a group of people sitting in a circle. If they want to speak, they have to raise their hands and wait for calls. A central chairman chooses a person to speak once at a time. In a while, the situation becomes stable: exactly one person raises his or her hand and is chosen to speak.

Three solutions were given. The first one with K-state machines is easy since K is larger than the number n of machines. The second one is a solution with four-state machine. The most complicated one with three-state machines is shown in Algorithm 1. These algorithms are correct: in a finite time, exactly one privilege will be present.

2.2 Fundamental concepts of self-stabilizing systems

In spite of the fact that the concept of self-stabilization was introduced by Dijkstra in 1974 [15], serious work only began in the late nineteen-eighties. See Dolev's book for an overview

Algorithm 1 `Dijkstra`

```
variables
    $M_i.S \in \{0, 1, 2\}$ with addition taken modulo 3;

for the machine $M_0$
    if $M_0.S + 1 = M_1.S$    then $M_0.S := M_0.S + 2$;

for the machine $M_{n-1}$
    if $M_{n-2}.S = M_0.S$ and $M_{n-2}.S + 1 \neq M_{n-1}.S$    then $M_{n-1}.S := M_{n-2}.S + 1$;

for the other machines $M_i$
    if $M_i.S + 1 = M_{i-1}.S$    then $M_i.S := M_{i-1}.S$;
    if $M_i.S + 1 = M_{i+1}.S$    then $M_i.S := M_{i+1}.S$;
```

[16].

Self-stabilizing algorithms are specified as a collection of rules executed on each node. Each rule has a trigger *precondition* and an *action*. The precondition of a rule is a Boolean predicate involving the states of the node and its neighbors, and the action changes the state of the node by updating its variables. A rule is *enabled* if its precondition evaluates to be true. A node is *privileged* if at least one of its rules is enabled. A node *moves* by changing state if it is selected by the control scheduler. Therefore the execution of an action is called a *move*, no matter how many variables it changes in the action.

A fundamental idea of self-stabilizing algorithms is that the distributed system may be started from an arbitrary configuration. After a finite time, the system reaches a correct configuration, called a *legitimate* configuration. We say that the system has *stabilized* if no nodes are privileged. An algorithm on a distributed system is called *self-stabilizing* [15, 16, 18, 65] if the following two properties hold:

Convergence The convergence property ensures that, starting from any illegitimate state, the distributed system reaches a legitimate state in a finite time without any external intervention.

Closure The closure property ensures that, after convergence, the system remains in the set of legitimate states.

The concept of self-stabilization has been developed for different communication styles, and algorithms on a distributed system may assume the inter-node communication models: message-passing model and shared memory model. In communication computer networks, computers communicate by exchanging messages, which is called the *message-passing model*. In the message-passing model, asynchronous computers sending and receiving messages through first-in first-out (FIFO) queues which may cause message losing. Other distributed systems usually use the *shared memory model* of communication, where neighboring communicating entities may communicate via common variables or registers. We distinguish two variants of the shared memory models. The *link-register model* is associated with a multiprocessor computer or a multitasking single computer, in which each processor/process can communicate with each other only by using separate registers. By using a global clock pulse, writing in and reading from the shared memory of two or more neighboring entities can be synchronized without losing information. The other is the *state reading model*, which is wildly used in the communicating graph algorithms. In the state reading model, each node can directly read the internal states of its neighboring nodes. Our algorithms assume the state reading model of communication.

Nodes in a distributed system can be modeled as state machines performing a sequence of steps. More precisely, we use *composite atomicity*, meaning that a node may read all its input variables, perform a state transition, and write all its output variables in a single atomic step. We assume that rules are atomically executed, that is, the evaluation of a precondition and the move are performed in one atomic step. All of the self-stabilizing algorithms presented in this thesis are concerned with composite atomicity and the state reading model unless otherwise specified. Note that the weakest communication model with register is known as the *read/write atomicity* model where a process can only atomically read the state of its neighbors or update its own state [18].

When estimating the time complexity of a self-stabilizing algorithm, we consider the stabilization time not the running time, since a self-stabilizing algorithm is usually a do forever loop and it does not terminate. The *stabilization time* of a self-stabilizing algorithm is the maximum amount of time it takes for the system to reach a legitimate configuration. The stabilization time is estimated in terms of time-steps or in terms of rounds. A *time-step* of a system is the amount of time in which a process can make an atomic step. A *round* is a minimal sequence of time-steps where every process privileged at the start is either tapped or sees its move disabled by the move of a neighbor. In general, the number of time-steps is bounded above by the number of moves.

We assume in a self-stabilizing algorithm, each node maintains its local variables and can make up decisions based on its local variables and its neighbor's local variables. A node can change its state by making a move, that is, by changing the value of at least one of its local variables. In this thesis, we will follow the conventions used in [30, 34, 75]. In particular, every node executes the same set of rules, and maintains and changes its own state based on its current state and the states of its neighbors [66]. These algorithms can be implemented on wireless sensor networks by using the beacon messages [30]. During the execution, every sensor node broadcasts its state to its neighbors after every change of the state.

2.3 Daemons and fairness

Various execution models have been used in self-stabilizing algorithms and these are encapsulated within the notion of daemons (or schedulers). Under different daemons, the algorithms differ greatly. Three commonly used daemons are: distributed daemon, synchronous daemon, and central daemon. In order to consider the worst-case scenario, a self-stabilizing algorithm is assumed to face an adversarial daemon. The *central daemon* picks up only one privileged node to execute an atomic step at each time-step. If a *synchronous daemon* is supposed, then all the privileged nodes execute an atomic step at each time-step. The *distributed daemon* chooses an arbitrary subset of the privileged nodes to execute an atomic step at each time-step. For the synchronous daemon, the measures time-steps and rounds coincide.

A scheduler can be fair or unfair. A *fair* daemon guarantees that every node is eventually selected to make a move. Otherwise it is *unfair*. If a daemon is unfair, then a round is not guaranteed to finish. It is well-known that an unfair distributed daemon is more practical than the other types of daemons and it is the daemon used by our algorithms and by [30, 75].

A distributed algorithm is said to be *uniform* if all of the individual processes run the same set of rules. If all processes run the same set of rules except a single process, called the root or the leader, then the algorithm is *semi-uniform*. When algorithms are designed for *anonymous* systems, the processes do not have any identification [67]. Some algorithms assume that processes have globally unique identifiers. Also, when a process can make moves with random outputs, the algorithm is said to be *probabilistic* (or randomized); otherwise it is *deterministic*. A distributed algorithm is *dynamic* if it can tolerate changes in the topology of the system during its execution. In [18] it is pointed out that self-stabilizing uniform algorithms designed for systems of arbitrary topology are dynamic.

Symmetry breaking is essential for algorithms in distributed systems assuming synchronous or distributed daemons [46]. For some problems it has been shown that deterministic uniform self-stabilizing algorithms are impossible for an anonymous system because the difficulty encountered in *symmetry breaking* [68]. We assume that each node is designated a unique numeric identifier (ID) so that two neighboring nodes can compete against each other by comparing their IDs. It is to be noted that most self-stabilizing algorithms actually only require locally unique identifiers (i.e., no closed neighborhood contains two identical identifiers). We denote by $v.id$ the identifier of a node v, or by v itself if no ambiguity is caused.

2.4 Self-stabilizing algorithms on graphs

Graph algorithms have natural applications to networks and distributed systems since graphs can be used to model networks or distributed systems. For example, dominating sets are suitable for cluster formation. Colorings are used in mutual exclusion and resource allocation problems. Spanning trees are used in tasks like routing and broadcasting, while matchings can be used in situations of communication where a node must be coupled with exactly one of its neighbors. Thus, many distributed graph algorithms have been developed to be used in network protocols or distributed systems.

Since the publication of Dijkstra's pioneering paper, lots of self-stabilizing algorithms for a variety of problems have been proposed in the literature. Here, we are interested in the category of graph theoretic problems, which has been deeply investigated since nineteen-nineties. Self-stabilizing algorithms for a variety of graph theoretic problems have been published in recent years. Some problems such as spanning tree construction [2, 6, 10, 23, 39, 63, 69, 71, 73], independence [21, 30, 32, 36, 45, 66, 67, 75], domination [4, 21, 28, 29, 30, 36, 43, 44, 50, 51, 75, 76, 83], coloring [24, 26, 38, 41, 42, 48, 53, 56, 59, 70, 72, 77], and matching [8, 27, 37, 40, 57, 58] have received more attention than other problems such as finding centers [5], vertex covering [54], clustering [13], etc.

In a graph algorithm for finding a vertex subset such as the MDS or MIS, the states of processes are categorized into two types. The first type is `IN`, with which the process is considered in the set S with the desired property (MDS or MIS). On the other hand, if the process is considered not in S, then we say it is `OUT`. The nodes are referred to as `IN` nodes and `OUT` nodes according to their states. A neighbor is an `IN` (resp., `OUT`) neighbor if it is an `IN` (resp., `OUT`) node.

In such a setting when a node makes an execution and therefore changes the values of its variables, we will say it *makes a move*. A *membership move* is a move such that the state of the node changes from `IN` to `OUT` or from `OUT` to `IN` after the execution. An MDS of a distributed system consists of the set of `IN` nodes after the algorithm stabilizes. If the system is self-stabilizing with respect to MDS, one must show that the convergence and closure properties both hold. While self-stabilizing algorithms exist for computing an MDS, no self-stabilizing algorithm is known to compute a minimal CDS; the predicate seems to be locally non-computable, due to its nature of global predicates [33].

2.5 Applications to wireless sensor networks

Self-stabilization has applications to wireless sensor networks (WSNs). In a wireless system, sensor nodes broadcast their local states to neighbors after each move. The asynchronous execution of an algorithm in each sensor node can be described as of an unfair distributed daemon independently selects a nonempty subset of privileged nodes. The sensor nodes in the network are influenced by the natural forces which make them mobile. A sensor node would be dead when its limited battery power depletes and if this happens then it leaves the WSN. Hence the topology of a WSN may change frequently. A WSN can model by a distribution system and each node maintains its own neighbor list and can communicate to neighbors via broadcasting messages. When the topology of the system changes, we say an external interference occurs and the system is not stable any more, thus we need a self-stabilizing algorithm to recover the system from an illegitimate configuration.

Chapter 3

Self-stabilizing MDS algorithms

In this chapter, we consider a distributed system whose topology is represented by an undirected, simple graph $G = (V, E)$. Let n denote the number of nodes in the system. Assume each node can read the local state of its neighboring nodes in G. The MDS problem is that of finding a minimal dominating set in any given graph G and this is the problem concerned in this chapter. The main contribution of this chapter is to propose a $(4n - 2)$-move self-stabilizing algorithm for the MDS problem under an unfair distributed daemon. A preliminary version of this chapter appeared as [12].

This chapter is organized as follows. In Section 3.1, we give previous results in self-stabilizing MDS algorithms. Section 3.2 presents a $(4n - 2)$-move self-stabilizing algorithm called `Well4n` for the MDS problem. Section 3.3 proves the correctness and convergence properties of `Well4n` and gives an upper bound of $4n - 2$ moves. Simulations in average analysis of our algorithm and comparisons with other linear-move MDS algorithms are given in Section 3.4.

3.1 Previous results

In [36], Hedetniemi et al. proposed the first self-stabilizing algorithm for the MDS problem; their algorithm assumes the central daemon. In [83], Xu et al. presented an algorithm under the synchronous daemon. In [75], Turau proposed a $9n$-move algorithm under an unfair distributed daemon; this algorithm is the first linear-time self-stabilizing algorithm for the MDS problem. In [30], Goddard et al. presented a $5n$-move algorithm. A good survey for the self-stabilizing algorithms for the MDS problem can be found in [34].

For simplicity of notation in the context, we say that a configuration *conforms* to MDS if the set of `IN` nodes is an MDS of G; otherwise the configuration *violates* MDS. The time complexity of a self-stabilizing algorithm is estimated in terms of moves or in terms of rounds. As was mentioned in the literature, for a wireless system with bounded resources, the number of moves is at least as important as the number of rounds. The reason is that a node has to broadcast the state to its neighbors after taking a move and therefore a reduction of the number of moves prolongs the lifetime of a network. In the paper [83], Xu uses the number of rounds to estimate the time complexity; but [30, 36, 75] and this thesis use the number of moves. All the known results are summarized in Table 3.1.

Table 3.1: Self-stabilizing algorithms for the minimal dominating set problem.

	stabilization time	daemon type
Hedetniemi et al. [36]	$(2n+1)n$ moves	central
Xu et al. [83]	$4n$ rounds	synchronous
Turau [75]	$9n$ moves	distributed
Goddard et al. [30]	$5n$ moves	distributed
this thesis (in Sec. 3.2)	$4n-2$ moves	distributed

In the following, we review algorithms in the previous results.

3.1.1 The first self-stabilizing MDS algorithm

In [36], Hedetniemi et al. proposed the first MDS self-stabilizing algorithms. Under a central daemon, the configuration of a graph stabilizes in $O(n^2)$ moves. In their MDS algorithm, every node has two variables: a Boolean variable indicating its state is `IN` or `OUT`, and a pointer pointing to one of its neighbors. For convenience, the neighbors of state `IN` are called `IN` neighbors. A node will point to the unique `IN` neighbor, otherwise it will point to null. A node is allowed to enter the MDS if it has no `IN` neighbor. In contrast, a node will leave the MDS if it has at least one `IN` neighbor and there is no neighbor pointing to it.

3.1.2 The second self-stabilizing MDS algorithm

In [83], Xu et al. presented a self-stabilizing algorithm for the MDS problem using unique identifiers under the synchronous daemon. The stabilization time is $O(n)$. Like Hedetniemi's algorithm, every node has two variables: a Boolean variable and a pointer. A node will point (i) to the unique `IN` neighbor, (ii) to itself if it has no `IN` neighbor, or (iii) to null if it has more than one `IN` neighbor. A node will enter the MDS if it has no `IN` neighbor and it has the smallest identifier within its closed neighborhood. A node will leave the MDS under the same condition as in Hedetniemi's algorithm.

3.1.3 The first linear-move self-stabilizing MDS algorithm assuming the distributed daemon

In [75], Turau proposed a linear-time self-stabilizing algorithm (we call it `Turau3n`) for the MIS problem with the unique identifier assumption. Every node has a variable that may have one of three different values: `IN` (in the set), `OUT` (out of the set), or `WAIT` (an `OUT` node with no `IN` neighbor, waiting to join the set). So, `Turau3n` runs as follows: an `OUT` node that has no neighbor in the MIS will first change its variable to `WAIT`. After doing so, the node may change its variable to IN if it has no WAIT neighbor with a lower identifier. Also, an IN node may leave the MIS and change its variable to OUT if it has an IN neighbor. Algorithm Turau3n is self-stabilizing under an unfair distributed daemon and stabilizes after at most $3n$ moves with an MIS. Turau3n is shown in Algorithm 2.

Algorithm 2 Turau3n

```
variables
  $v.state \in \{\texttt{IN}, \texttt{OUT}, \texttt{WAIT}\}$;            // $I = \{v \mid v.state = \texttt{IN}\}$ is an MIS

macros
  $inNeighbor(v) \equiv \exists w \in N(v) : w.state = \texttt{IN}$;
  $waitNeighborWithLowerId(v) \equiv \exists w \in N(v) : w.state = \texttt{WAIT} \wedge w < v$;
  $inNeighborWithLowerId(v) \equiv \exists w \in N(v) : w.state = \texttt{IN} \wedge w < v$;

rules
  1: if $v.state = \texttt{OUT} \wedge \neg inNeighbor(v)$
       then $v.state := \texttt{WAIT}$;                              // wait for change
  2: if $v.state = \texttt{WAIT} \wedge inNeighbor(v)$
       then $v.state := \texttt{OUT}$;                               // stop waiting
  3: if $v.state = \texttt{WAIT} \wedge \neg inNeighbor(v) \wedge \neg waitNeighborWithLowerId(v)$
       then $v.state := \texttt{IN}$;                                // enter $I$
  4: if $v.state = \texttt{IN} \wedge inNeighbor(v)$
       then $v.state := \texttt{OUT}$;                               // leave $I$
```

Based on Turau3n, Turau extended the rules to design the first self-stabilizing MDS algorithm and we call it Turau9n. Each node has two variables. The first three-valued variable *state* is defined as the one in Turau3n. The second variable is a pointer variable called *dependent*. An IN node changes *dependent* to *null*. An OUT node changes *dependent* to *null* if there is more than one IN neighbor, or to the only one IN neighbor. The entering rule is the same as in Turau3n. Besides, the leaving rule is modified by adding the precondition "there is no neighbor pointing to it". Turau proved that Algorithm Turau9n is self-stabilizing under an unfair distributed daemon and stabilizes after at most $9n$ moves with an MDS. The detail of Turau9n is shown in Algorithm 3.

Algorithm 3 Turau9n

```
variables
    $v.state \in \{\texttt{IN}, \texttt{OUT}, \texttt{WAIT}\}$;                // $D = \{v \mid v.state = \texttt{IN}\}$ is an MDS
    $v.dependent \in N(v) \cup \{null\}$;            // point to the dominator if $|N(v) \cap D| = 1$

macros
    $inNeighbor(v) \equiv \exists w \in N(v) : w.state = \texttt{IN}$;
    $waitNeighborWithLowerId(v) \equiv \exists w \in N(v) : w.state = \texttt{WAIT} \wedge w < v$;
    $inNeighborWithLowerId(v) \equiv \exists w \in N(v) : w.state = \texttt{IN} \wedge w < v$;
    $uniqueInNeighbor(w, v) \equiv \exists! w \in N(v) : w.state = \texttt{IN}$;
    $dependentNeighbors(v) \equiv \exists w \in N(v) : w.dependent = v$;

rules
    1: if $v.state = \texttt{OUT} \wedge \neg inNeighbor(v)$
           then $v.state := \texttt{WAIT}$;                               // wait for change
    2: if $v.state = \texttt{WAIT} \wedge inNeighbor(v)$
           then $v.state := \texttt{OUT}$;                                // stop waiting
    3: if $v.state = \texttt{WAIT} \wedge \neg inNeighbor(v) \wedge \neg waitNeighborWithLowerId(v)$
           then $v.state := \texttt{IN}$; $v.dependent := null$;          // enter $D$
    4: if $v.state = \texttt{IN} \wedge inNeighbor(v) \wedge \neg dependentNeighbors(v)$
           then $v.state := \texttt{OUT}$;                                // leave $D$
    5: if $v.state = \texttt{IN} \wedge v.dependent \neq null$
           then $v.dependent := null$;                                    // modify the pointer
    6: if $v.state = \texttt{OUT} \wedge uniqueInNeighbor(w, v) \wedge v.dependent \neq w$
           then $v.dependent := w$;                                       // modify the pointer
    7: if $v.state = \texttt{OUT} \wedge moreThanOneInNeighbor(v) \wedge v.dependent \neq null$
           then $v.dependent := null$;                                    // modify the pointer
```

3.1.4 The second linear-move self-stabilizing MDS algorithm

In [30], Goddard et al. proposed a $5n$-move algorithm (we call it Goddard5n) for the MDS problem with nodes having locally distinct identifiers under an unfair distributed daemon. When Algorithm Goddard5n stabilizes, the set $S = \{i : x(i) = 1\}$ is an MDS of the given distributed system. In detail, each node maintains a Boolean variable x and a three-valued variable c. The value $x(i) = 1$ indicates that $i \in S$, while $x(i) = 0$ indicates that $i \notin S$. The counter $c(i)$ counts the number of IN neighbors: $c(i) = 0$ indicates that i has no IN neighbor, $c(i) = 1$ means that i has exactly one IN neighbor, and $c(i) = 2$ means that i has at least two IN neighbors. The value of $c(i)$ is rectified only when $x(i) = 0$ and is ignored if $x(i) = 1$. A node is allowed to join S if it has no IN neighbor and its counter evaluates 0 and it has no lower identifier neighbor having $c = 0$. On the other hand, a node is allowed to leave the MDS if it has an IN neighbor and the counters of all OUT neighbors evaluate to 2, which means that they all have more than one IN neighbor. Algorithm Goddard5n is shown in Algorithm 4.

Algorithm 4 Goddard5n

```
variables
    flag $x(i) \in \{0, 1\}$;                                   // $S = \{i : x(i) = 1\}$
    integer $c(i) \in \{0, 1, 2\}$;                             // counter of $N(i) \cap S$

rules
    D1: if $c(i)$ incorrect $\wedge (x(i) = 0)$
            then correct $c(i)$;                                // rectify counter
    D2: if $(|N(i) \cap S| = 0) \wedge (x(i) = 0) \wedge (c(i) = 0) \wedge (\nexists j \in N(i) : j < i, c(j) = 0)$
            then $x(i) := 1$;                                   // enter $S$
    D3: if $(|N(i) \cap S| > 0) \wedge (x(i) = 1) \wedge (\forall j \in N(i) \backslash S : c(j) = 2)$
            then $x(i) := 0$ and make sure $c(i)$ is correct;   // leave $S$
```

In [30], Goddard et al. showed that Algorithm Goddard5n stabilizes in at most $5n$ moves under the distributed daemon and it stabilizes in at most $4n + 1$ time-steps under the synchronous daemon. Although the publication time of [30] is in 2008 which is later then [75] in 2007, I personally suggest that this paper gives birth to the first linear-time self-stabilizing MDS algorithm since it is received in May 2006, and there is a two month gap before Turau submitted for publication of [75] in July 2006.

3.2 A $(4n - 2)$-move MDS algorithm

The purpose of this section is to present our main result: Well4n, a $(4n - 2)$-move self-stabilizing algorithm for the MDS problem under an unfair distributed daemon. Algorithm

`Well4n` uses four states, which are defined by the four-valued variable *state*. The range of values of *state* is: `IN`, `OUT1`, `OUT2`, and `WAIT`. A node with $state = $ `IN` will be referred to as an `IN` node. Let $S = \{v : v.state = \texttt{IN}\}$; i.e., S is the set of `IN` nodes. Nodes with $state = $ `OUT1` or `OUT2` or `WAIT` will be referred to as an `OUT` node.

The values of *state* have the following meanings. The value `IN` indicates that the node is in the MDS. The value `OUT1` means that the node is not in the MDS and it has a unique `IN` neighbor. The value `OUT2` indicates that the node is not in the MDS and it has at least one `IN` neighbor. The value `WAIT` means that the node is not in the MDS and it does not have any `IN` neighbors. For the MDS problem, a legitimate state means: the distributed system reaches the desired global property that the configuration conforms to MDS. To make it precise, in our self-stabilizing MDS algorithm, a *legitimate configuration* is: the set of `IN` nodes form an MDS, and every `OUT` node with $state = $ `OUT1` has a unique `IN` neighbor, $state = $ `OUT2` has at least one `IN` neighbor, and $state = $ `WAIT` has no `IN` neighbor. Notice that a legitimate configuration will not contain any `WAIT` node.

To formally define the rules of `Well4n`, the following predicates defined for each node v are needed:

- $noBtNbr(v) \equiv \nexists\, w \in N(v) : w.state = \texttt{WAIT} \wedge w.id < v.id$.
- $noDpNbr(v) \equiv \nexists\, w \in N(v) : w.state = \texttt{OUT1}$.

We now explain the meaning of *noBtNbr* and *noDpNbr* and the idea of using local distinct identifier to break symmetry assuming the distributed daemon. When two or more neighboring nodes want to enter the MDS simultaneously, our algorithm chooses the one with the smaller (smallest) *id*. According to this, if v is an `OUT` node and has a neighbor w such that $w.state = $ `WAIT` and $w.id < v.id$, then w is called a *better neighbor* of v. The predicate *noBtNbr* indicates that v has no better neighbor. Also, if v is an `IN` node and has a neighbor w with $w.state = $ `OUT1`, then w is called a *dependent neighbor* of v since w

depends on its unique neighbor in the MDS (the neighbor is v). The predicate *noDpNbr* indicates that v has no dependent neighbor.

For convenience, we introduce

- $InNbr(v) = |\{u \in N(v) \mid u.state = \texttt{IN}\}|$.

In the algorithm `Well4n`, each node executes the six rules shown in Algorithm 5. The state diagram of `Well4n` is given in Figure 3.1.

Algorithm 5 `Well4n`

```
variables
    $v.state \in \{\texttt{IN}, \texttt{OUT1}, \texttt{OUT2}, \texttt{WAIT}\}$;                // $S = \{v : v.state = \texttt{IN}\}$

macros
    $InNbr(v) = |\{u \in N(v) \mid u.state = \texttt{IN}\}|$;
    $noBtNbr(v) \equiv \nexists w \in N(v) : w.state = \texttt{WAIT} \wedge w.id < v.id$;
    $noDpNbr(v) \equiv \nexists w \in N(v) : w.state = \texttt{OUT1}$;

rules
    R1: if $v.state = \texttt{WAIT} \wedge InNbr(v) = 0 \wedge noBtNbr(v)$
            then $v.state := \texttt{IN}$;                                  // enter $S$
    R2: if $v.state = \texttt{IN} \wedge InNbr(v) = 1 \wedge noDpNbr(v)$
            then $v.state := \texttt{OUT1}$;                                // leave $S$
    R3: if $v.state = \texttt{IN} \wedge InNbr(v) > 1 \wedge noDpNbr(v)$
            then $v.state := \texttt{OUT2}$;                                // leave $S$
    R4: if $v.state = \texttt{WAIT} \wedge InNbr(v) = 1$
            then $v.state := \texttt{OUT1}$;                                // rectify the counter
    R5: if $(v.state = \texttt{OUT1} \vee v.state = \texttt{WAIT}) \wedge InNbr(v) > 1$
            then $v.state := \texttt{OUT2}$;                                // rectify the counter
    R6: if $(v.state = \texttt{OUT1} \vee v.state = \texttt{OUT2}) \wedge InNbr(v) = 0$
            then $v.state := \texttt{WAIT}$;                                // rectify the counter
```

3.3 Correctness and convergence

We now prove the correctness of `Well4n`.

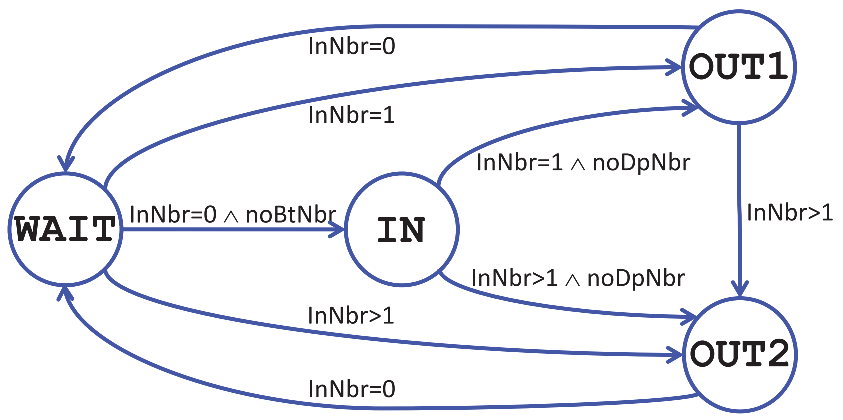

Figure 3.1: The state diagram of `Well4n`.

Lemma 3.3.1. *In any configuration in which no node is privileged, (a) the number of* `IN` *neighbors of every* `OUT` *node consistent with its state, and (b) the set S is a minimal dominating set for G.*

Proof. For (a), it is easy to check that if the state of an `OUT` node is inconsistent with the description on the number of its `IN` neighbors, then a rule can be enabled for this node.

For (b), suppose to the contrary that S is not a minimal dominating set for G. Then either (i) S is not a dominating set or (ii) S is a dominating set but not minimal. First consider (i). Since S is not a dominating set, there exists at least one node $u \notin S$ which has no `IN` neighbor; let S' be the set of all such nodes. Since rule R6 is not enabled, every node in S' has $state$ = `WAIT`. Let u_0 be the node in S' with minimum id. Then u_0 satisfies all the constraints of rule R1. Hence rule R1 is enabled and this contradicts to the assumption that no node is privileged.

Next consider (ii). Since S is a dominating set but not minimal, there must exist at least one node $u \in S$ such that $S \backslash \{u\}$ is also a dominating set for G. Then $|N(u) \cap S| \geq 1$ and for all u' in $N(u) \backslash S$, we have $|N(u') \cap S| \geq 2$. Thus, every node u' in $N(u) \backslash S$ has $InNbr(u') > 1$. Hence, every node u' in $N(u) \backslash S$ must have $u'.state$ = `OUT2`; otherwise rule R5 is enabled on u'. Consequently, node u has $noDpNbr(u) = true$ and either $InNbr(u) = 1$ (if $|N(u) \cap S| = 1$) or $InNbr(u) > 1$ (if $|N(u) \cap S| > 1$). Hence, either rule R2 or rule R3 is enabled on node u, which is a contradiction. $\square$

Note that when the algorithm terminates, there is no node with $state$ = `WAIT`. Since R6 is not enabled, any `OUT` node without an `IN` neighbor must have $state$ = `WAIT`. Hence to prove that there is no node with $state$ = `WAIT`, it suffices to prove that S is a minimal dominating set of G.

We now show that `Well4n` converges in a finite time. In particular, we show that the number of moves of `Well4n` is at most $4n-2$. Let k be a nonnegative integer and $\langle r_1, r_2, \ldots, r_k\rangle$ be a sequence of rules (r_i's are not necessarily distinct). The sequence $\langle r_1, r_2, \ldots, r_k\rangle$ is called a *move sequence* if a node can execute rule r_1, then rule r_2, …, then rule r_k. The following two lemmas show that in any possible move sequence of a specific node, rule R1 and rule R6 appear at most once when a distributed system run `Well4n` for a time period without any external intervention.

Lemma 3.3.2. *If a node executes rule R1, then it will not execute any other rule. Consequently, if a node enters the set S, then it will never leave S.*

Proof. Let v be a node which executes rule R1. Then $v.state$ has been set to `IN` and thereafter v enters S. By the precondition of rule R1, v has no `IN` neighbor and no better neighbor; therefore no neighbor of v enters S at the same time when v enters S. Thus, no node in $N(v)$ that enters S and therefore $InNbr = 0$. After executing rule R1, $v.state$ is `IN` and the possible rule that v can execute is either rule R2 or rule R3. Rule R2 is impossible since it requires $InNbr(v) = 1$; similarly rule R3 is also impossible since it requires $InNbr(v) > 1$. Therefore, v will not execute any other rule. The second statement of this lemma now follows. $\square$

Lemma 3.3.3. *A node can execute rule R6 at most once, or equivalently, a node can set its state to* `WAIT` *at most once.*

Proof. Let v be a node which executes rule R6 once. By the precondition of rule R6, v has no `IN` neighbor when executing rule R6. After executing rule R6, $v.state$ is set to `WAIT` and the possible rules that v can execute is rule R1 or rule R4 or rule R5. If v executes rule R1, then by Lemma 3.3.2, v will not execute any other rule and we have this lemma. If v executes rule R4 or R5, then $InNbr(v) = 1$ or $InNbr(v) > 1$ must be *true* before rule R4 or R5 is enabled, meaning that v has a neighbor (say, u) which has executed rule R1; by Lemma 3.3.2 again, u will never leave S. Therefore it is impossible to have $InNbr(v) = 0$, which means that v cannot execute rule R6 again. □

In the next, we firstly claim that the length of a move sequence of any node in G is at most 4. If this is true, then the total number of moves among all nodes in G is at most $4n$. Furthermore, we improve the upper bound to $4n - 2$ and show that this bound is tight.

Theorem 3.3.4. *The proposed algorithm* `Well4n` *is self-stabilizing under an unfair distributed daemon and it stabilizes after at most $4n-2$ moves with a minimal dominating set, where n is the number of nodes. Moreover, the bound $4n - 2$ is tight.*

Proof. By Lemma 3.3.1, the algorithm `Well4n` is correct. To prove that `Well4n` stabilizes after at most $4n - 2$ moves, we first prove that it stabilizes after at most $4n$ moves, from which we conclude that `Well4n` has the convergence property. To do this, it suffices to show that any move sequence of a node is of length at most 4 under an unfair distributed daemon. Let v be an arbitrary node in G. By Lemma 3.3.3, v can execute rule R6 at most once. Thus, there are two cases: v never executes R6 and v executes R6 once.

First consider the case that v never executes rule R6. Then $v.state$ never changes to `WAIT`. Thus, the move sequence of v is either $\langle$R1$\rangle$ or $\langle$R2,R5$\rangle$ or $\langle$R4,R5$\rangle$. It follows that any move sequence of v is of length at most 2.

Now consider the case that v executes rule R6 once. In this case, regard a move sequence of v as the concatenation of a prefix and a suffix. By Lemma 3.3.2, the prefix of any move sequence of v cannot contain rule R1 since if v executes rule R1 then v will not execute any other rule, including rule R6. Hence, the possible prefix of any move sequence of v is either $\langle$R2,R6$\rangle$ or $\langle$R3,R6$\rangle$ or $\langle$R4,R6$\rangle$ or $\langle$R5,R6$\rangle$. After v executes rule R6, $v.state$ changes to `WAIT`. Thus, the possible suffix of any move sequence of v is either $\langle$R6,R1$\rangle$ or $\langle$R6,R4,R5$\rangle$ or $\langle$R6,R5$\rangle$. Concatenating the prefix and suffix, we conclude that any move sequence of v is of length at most 4.

From the above, `Well4n` stabilizes after at most $4n$ moves with a minimal dominating set. We now prove that the bound can be strengthen to $4n-2$. The cases of $n = 1$ and $n = 2$ are trivial. Suppose $n \geq 3$ and one of the nodes makes 4 moves; by the above argument, this node has two neighbors executing rule R1. Thus, at least two nodes in G make less than 4 moves and the upper bound can be strengthen to $4n - 2$.

We now give an example to show that the upper bound $4n - 2$ is tight. Consider the complete bipartite graph $K_{2,n-2}$, where $n \geq 3$. Let the two nodes in the partite set of cardinality two have the maximum and the minimum identifiers among the n nodes. If initially all nodes are in state `IN`, then there is a way that all the rest of the nodes executes $\langle$R3,R6,R4,R5$\rangle$ but nodes with the maximum and minimum identifiers execute $\langle$R3,R6,R1$\rangle$. All together $4n - 2$ moves are made. □

Note that our algorithm `Well4n` will be equivalent to `Goddard5n` if rule R4 is modified as follows:

R4': **if** $(v.state = \texttt{OUT2} \ \vee v.state = \texttt{WAIT}) \wedge InNbr(v) = 1$

then $v.state := \texttt{OUT1}$;

The idea of state `OUT1` is to lock the unique `IN` neighbor. If node v change its state from `OUT1` to `OUT2`, there exists a neighbor u of v entering the set S. By Lemma 3.3.2, u will never leave S. Hence, u will dominate v thereafter. Suppose then another `IN` neighbor w of v wants to leave S, and somehow it really does. If u becomes the only one `IN` neighbor of

v, then there is no need for v to change its state back to `OUT1` since neighbor u will never leave S. Hence the modification we made is reasonable and it can shorten the longest move sequence from ⟨R5,R4,R5,R4,R5⟩ to ⟨R3,R6,R4,R5⟩.

3.4 Simulations and comparisons

Theoretically, we have proven that: in the worst case, `Well4n` makes the smallest number of moves to stabilize among `Well4n`, `Goddard5n`, and `Turau9n`. However, the average behaviors of these algorithms are not known. This section presents simulations and comparisons of these three algorithms. Notice that we implement the original version of `Turau9n` since the modified version is incorrect (we explain the reason at the end of this section).

Simulation environments are conducted as follows and we calculate the mean and the standard deviation. We regard the distributed system as a wireless sensor network, where the transmission range of each node is the same. Thus the communication graph of the distributed system is a unit disk graph (UDG). Notice that our simulations only consider UDGs and the underlying graph may not be connected. We consider UDGs since they are the most commonly used models for wireless sensor networks (of course, other classes of graphs can be considered). In our simulations, the transmission range R of all of the nodes is set to from 15m to 45m with a step of 5m (m means meter). For each R, each algorithm is run 1000 times and each time 100 nodes are randomly placed in a square of size 200m × 200m (again, m means meter). The resultant average node degrees are 1.64 (when $R = 15$), 2.84, 4.35, 6.12, 8.17, 10.40, and 12.86 (when $R = 45$). In each simulation, we randomly assign each node an initial state and each state of the node is equally likely to happen. In particular, in `Goddard5n`, each node has four possible states: $x(i) = 0 \wedge c(i) = 0$, $x(i) = 0 \wedge c(i) = 1$, $x(i) = 0 \wedge c(i) = 2$, and $x(i) = 1$.

Recall that a move is called a membership move if its execution makes an `IN` node changes to an `OUT` node and vise versa, i.e., if the node executes rule R1, R2, or R3. In each simulation, every node independently computes the next state then summons the distributed daemon. The daemon then chooses a nonempty subset of nodes to make moves. The simulation continues until no node is privileged.

Figures 3.2 and 3.3 show the comparisons of the three algorithms. Figure 3.2 depicts the average number of "moves" with regard to the transmission ranges. In our simulations, `Well4n` always made fewer moves than `Goddard5n` and `Turau9n`. Figure 3.3 illustrates the average number of "membership moves" in terms of the transmission ranges. In our simulations, `Goddard5n` and `Well4n` make fewer membership moves than `Turau9n`. The simulation results show that if the average number of "moves" is considered, then `Well4n` outperforms `Goddard5n`; if the average number of "membership moves" is considered, then `Goddard5n` outperforms `Well4n`.

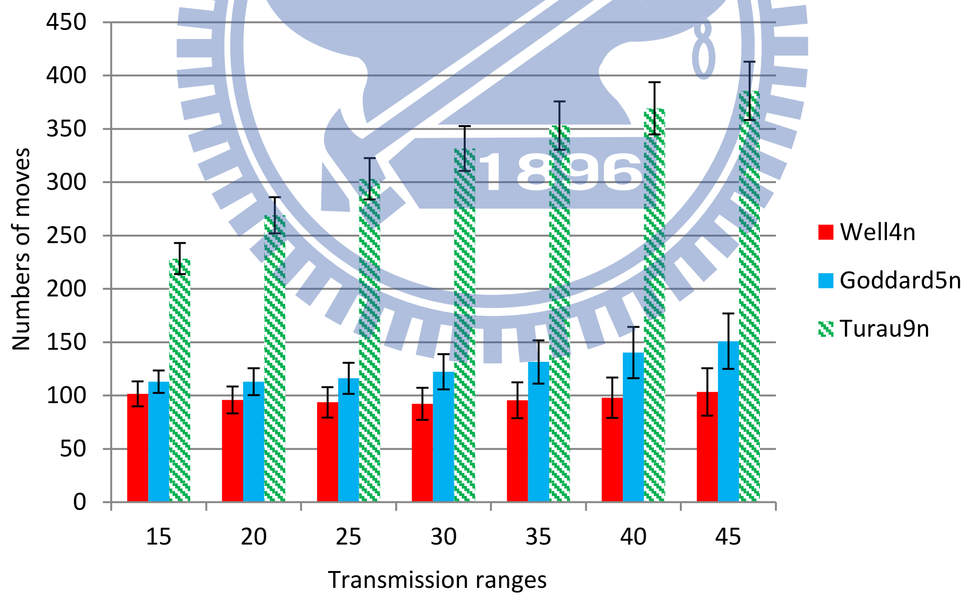

Figure 3.2: The average number of moves made by `Well4n`, `Goddard5n`, and `Turau9n` with regard to the transmission ranges, where the vertical line segments denote the standard deviations.

Before ending this section, we would like to point out an error in the modified `Turau9n`, which is called modified $\mathcal{A}_{\text{MDS}}$ in [75]. Turau claimed (in page 93 in [75]) that rule 4 can

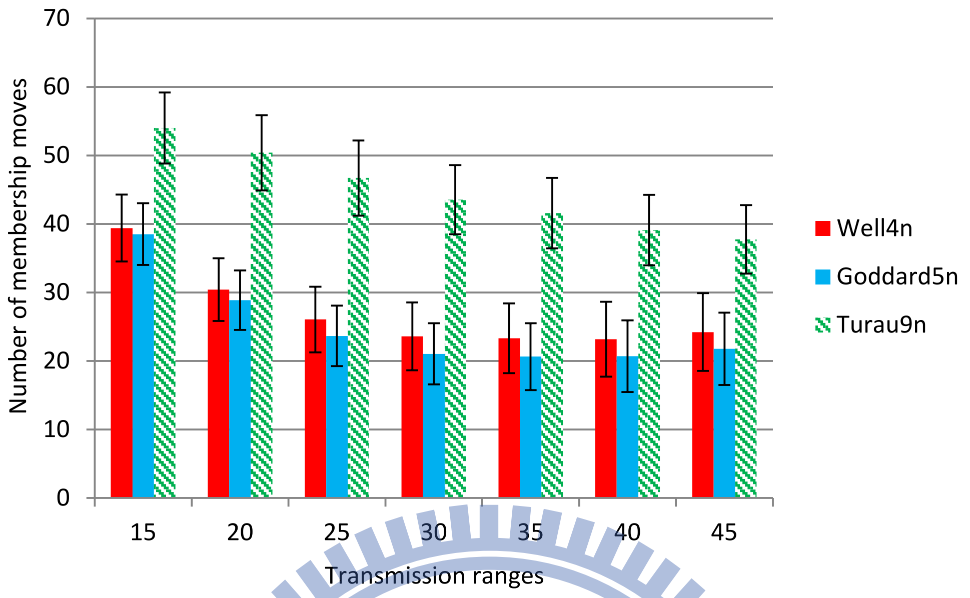

Figure 3.3: The average number of membership moves made by `Well4n`, `Goddard5n`, and `Turau9n` in terms of the transmission ranges, where the vertical line segments denote the standard deviations.

be changed by replacing the predicate $inNeighbor$ with $inNeighborWithLowerId$ so that the total number of moves can be further reduced. We now show that Turau's claim is incorrect and should be eliminated. Suppose the replacement is done and the resultant rule is called rule $4'$. Let G be a path of three nodes v_1, v_2, v_3 and edges v_1v_2, v_2v_3; suppose the initial configuration is: v_1, v_2 are `IN` nodes with $dependent = \Lambda$ and v_3 is an `OUT` node with $dependent = v_2$. Suppose $v_i.id$ is i. Since $\{v_1, v_2\}$ is not an MDS, modified $\mathcal{A}_{\text{MDS}}$ must make a move. It is not difficult to verify that no rule can be enabled by modified $\mathcal{A}_{\text{MDS}}$.

Chapter 4

The stableness of MDS algorithms

A node in a distributed system has limited information about the whole system. Usually, a node can only access the information of nodes in its 1-neighborhood. Let k be a positive integer. We say a distributed system is in the *distance-k model* if a node in the system can access the information of nodes in its k-neighborhood. For convenience, we say a distributed system is in the *normal model* if it is in the distance-1 model. What will happen if a node can access the information of nodes in its k-neighborhood for $k \geq 2$? In this chapter, we will discuss the above problem and propose a new performance measure, called *stableness*, for self-stabilizing MDS algorithms. We will define *MDS-silent* and *MDS-stable* algorithms, and we will categorize self-stabilizing algorithms into four levels: S_1*-stable*, S_2*-stable*, S_3*-stable*, and S_4*-stable*. In particular, Algorithms `Turau9n`, `Goddard5n`, and `Well4n` assume the distance-1 model and none of them is MDS-silent.

This chapter is organized as follows. Section 4.1 gives the motivation of classifying MDS algorithms into MDS-silent, MDS-covered, and MDS-stable algorithms. Section 4.2 introduces the transition diagram of a self-stabilizing system. Section 4.3 categorizes the self-stabilizing algorithms into four levels of stableness. A variation of self-stabilization, multi-self-stabilization, is formulated in Section 4.4.

4.1 Motivation

It is well-known that a maximal independent set is a minimal dominating set but the converse is not true. Moreover, the stabilizing time of a self-stabilizing MIS algorithm is usually less than that of a self-stabilizing MDS algorithm. Thus, why bother to develop a self-stabilizing MDS algorithm? Why not use a self-stabilizing MIS algorithm instead of a self-stabilizing MDS algorithm? In [75], Turau mentioned:

> Since it is desirable that a self-stabilizing algorithm initialized with a minimal dominating set does not make any moves, MIS-algorithms are not suitable solutions for the MDS problem.

This characterizes an important feature of a self-stabilizing MDS algorithm:

> The algorithm will not make any move if the given distributed system is initialized with a minimal dominating set.

We call this important feature *MDS-silent*. The notion of *MIS-silent* can be defined similarly. We first prove a lemma.

Lemma 4.1.1. *Any self-stabilizing MIS algorithm is a self-stabilizing MDS algorithm, but not necessarily MDS-silent.*

Proof. The former part of this lemma is straightforward and we only prove the latter part: a self-stabilizing MIS algorithm is not necessarily MDS-silent. Let G be a path of four nodes, labeled by $v_1, v_2, \ldots, v_4$ in order. More specifically, G has the vertex set $V = \{v_1, v_2, v_3, v_4\}$ and the edge set $E = \{v_1v_2,\ v_2v_3,\ v_3v_4\}$. Suppose initially v_2 and v_3 are `IN` nodes and v_1 and v_4 are `OUT` nodes. Since $\{v_2, v_3\}$ is not an MIS, any self-stabilizing MIS algorithm will make a move. However, $\{v_2, v_3\}$ is an MDS. If the algorithm is MDS-silent, then it should not make any move. Therefore a self-stabilizing MIS algorithm is not necessarily MDS-silent. □

Unfortunately, unlike self-stabilizing MIS algorithms which are usually MIS-silent (for example, Algorithm MIS in [30] is MIS-silent), none of the self-stabilizing MDS algorithms `Turau9n` [75], `Goddard5n` [30], and `Well4n` in Section 3.2 is MDS-silent. Notice that `Turau9n`, `Goddard5n`, and `Well4n` are executed in the normal model. We now prove by contradiction that a distributed system in the normal model cannot have an MDS-silent algorithm.

Theorem 4.1.2. *There exists no self-stabilizing MDS-silent algorithm in the normal model.*

Proof. Suppose this theorem is not true and there exists a self-stabilizing MDS-silent algorithm $\mathcal{A}$ in the normal model. For convenience, let I and O denote the state `IN` and `OUT`, respectively. Let G_1 be a path of four nodes with initial configuration $OIIO$, G_2 be a 4-path with $OIOI$, and G_3 a 5-path with $OIIOI$; see below.

G_1: $O - I - I - O$

G_2: $O - I - O - I$

G_3: $O - I - I - O - I$

Run algorithm $\mathcal{A}$ on these three graphs. Since the initial configuration of G_1 conforms to MDS, which is legitimate, $\mathcal{A}$ will not make any move on G_1. Similarly, since the initial configuration of G_2 is also legitimate, $\mathcal{A}$ will not make any move on G_2.

Now consider G_3. The first three nodes of G_3 cannot distinguish themselves with the first three nodes of G_1, hence no rule is enabled for the first three nodes of G_3. Also, the last two nodes of G_3 cannot distinguish themselves with the last two nodes of G_2, hence no rule is enabled for the last two nodes of G_3. Therefore for G_3, Algorithm $\mathcal{A}$ stabilizes with initial configuration $OIIOI$, which violates MDS, a contradiction. □

Another reason for developing self-stabilizing MDS algorithms is that while MIS algorithms can only stabilize with an independent set, MDS algorithms are capable of stabilizing with any minimal dominating set. In [36], Hedetniemi et al. mentioned that the significance of MDS algorithms is that if the system is initialized to any MDS with the correct variable settings (including minimal dominating sets that are not independent sets), then the system will remain stable. They also pointed out that an MDS algorithm is usually more complex than an MIS algorithm but can potentially produce *any* MDS. We call an algorithm *MDS-covered* if it can potentially produce any MDS. All of the MDS algorithms `Turau9n` [75], `Goddard5n` [30], and `Well4n` in Section 3.2 are MDS-covered.

Note that the proof of Theorem 4.1.2 also indicates that a self-stabilizing MDS-covered algorithm cannot have only one Boolean variable as its state variable. Otherwise, the information of non-minimality will not pass through the intermediate nodes. See the following corollary.

Corollary 4.1.3. *A self-stabilizing MDS-covered algorithm in the normal model cannot have only one Boolean variable as its state variable.*

By Corollary 4.1.3, as long as MDS-covered is a desired property of a self-stabilizing MDS algorithm, we may assume that an MDS algorithm requires two (or more) variables. The first variable is a Boolean variable with range $\{1, 0\}$ indicating the state `IN` or `OUT`, respectively; without loss of generality, call this variable *membership*. The second variable can be regarded as a integer variable and it has no effect on the `IN`/`OUT` state; again, without loss of generality, call this variable *information*.

Again, recall that a move of an MDS algorithm is a membership move if its execution makes an `IN` node change to an `OUT` node or makes an `OUT` node change to an `IN` node. This definition was originally given in [36]. Clearly, a non-membership move has no effect on the variable *membership*. In Theorem 4.1.2, we have proved that there exists no MDS-silent algorithm in the normal model. Now we relax the MDS-silent property (not to make any move if the distributed system is initialized with an MDS) to allow the execution of non-membership moves. Formally, an MDS algorithm is *MDS-stable* if it will not make any membership move when the system is initialized with an MDS. The containment relation of properties MDS-silent, MDS-covered, and MDS-stable will be explored in Section 4.3.

Before ending this section, we propose a conjecture. None of `Turau9n`, `Goddard5n`, and `Well4n` is MDS-stable.

Conjecture 4.1.4. *There exists no self-stabilizing MDS-stable algorithm in the normal model.*

4.2 Transition diagrams

In this section we explain how to draw the transition diagram of a self-stabilizing system. We approach self-stabilization from a different angle and give a method of visualization to different levels of stableness.

Let Ω_v be the set of all possible combinations of local variables of a process $v \in V$. Each element Q_v in Ω_v denotes a *local state* of v. For simplicity, we assume $V = \{1, 2, \ldots, n\}$. A tuple $(Q_1, Q_2, \ldots, Q_n)$ of local states of all the processes forms a *global state* (or *configuration*) of a distributed system G. For any configuration γ_t at time t, let γ_{t+1} denote the configuration that follows γ_t. Denote the transition relation by

$$\gamma_t \rightarrow \gamma_{t+1}.$$

A *computation sequence* starting from γ_0 is an infinite sequence of configurations $\gamma_0, \gamma_1, \ldots$ such that $\gamma_t \rightarrow \gamma_{t+1}$ for each $t \geq 0$.

Let Γ be the set of all configurations of G. The *transition diagram* of a distributed system is a digraph $D = (\Gamma, A)$, where $(\gamma, \gamma') \in A$ if $\gamma \rightarrow \gamma'$. The arc set of a transition diagram may change under different execution daemon or due to external intervention, and here we assume that the daemon is pre-designated and no external intervention during the execution and hence the diagram will be fixed. In this way, computation sequences can be regarded as directed paths in the transition diagram. The transition diagram may have *loops* or *cycles* and thus a computation sequence can be of infinite length. In the transition diagram, if a node has out-degree 0, we assume it has a loop implicitly. As shown in Figure 4.1, the out-degree of a configuration γ can be larger than one; hence the algorithm may not be deterministic.

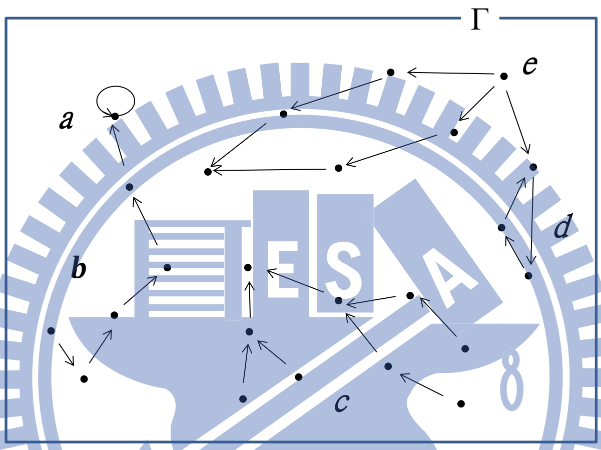

Figure 4.1: The transition diagram of a distributed system. The structures of a loop, a path, a tree, and a 3-cycle are shown in the areas a to d, respectively. Area e depicts a node with out-degree 3; this node denotes a configuration with three possible following configurations.

Recall that Γ is the set of all configurations of a distributed system. We say that a distributed system is *self-stabilizing with respect to* $\Lambda \subseteq \Gamma$ if the following conditions hold:

Convergence Starting from an arbitrary configuration $\gamma_0 \in \Gamma$, there exists an integer t such that $\gamma_t \in \Lambda$ in any computation sequence;

Closure For every configuration $\lambda \in \Lambda$, any configuration that follows λ is also in Λ.

Each $\lambda \in \Lambda$ is called a *legitimate configuration*.

4.3 Four levels of stableness

Stableness is the quality or state of being stable, or firmly established. In distributed computing, stableness informally requires that "good configurations eventually stop" in a distributed system or algorithm (i.e., the system or the algorithm "resists change").

We have defined self-stabilizing MDS-covered, MDS-silent, and MDS-stable algorithms. We now generalize these definitions. A configuration is *silent* if no rules are enabled at any node. In the literature, self-stabilizing graph algorithms are usually *silent*, that is, the algorithms eventually reach a silent configuration in a finite time. Let P denote a desired property. Let Λ' be the set of all decisions $(d_1, d_2, \ldots, d_n)$ satisfy the desired property P, where d_v is the decision of node v. Self-stabilizing algorithms can be described as having different degrees of stableness with respect to property P.

Definition 4.3.1. An algorithm is P if after stabilizing, the configuration conforms the desired property P. A P algorithm is:

P**-covered** if after stabilizing, the configuration potentially can be decoded to any possible element in Λ'.

P**-stable** if the distributed system starts with an element in Λ', the algorithm makes no membership moves.

P**-silent** if the distributed system starts with an element in Λ', the algorithm makes no moves.

Let Σ_A be the set of all silent configurations in a silent self-stabilizing algorithm A. If Σ_A is a proper subset of Λ, i.e., all silent configurations are legitimate, then it is easy to design a self-stabilizing P algorithm by shrinking the definition of legitimate configurations Λ_A of an algorithm A from Λ to Σ_A. One advantage of doing this is that we do not need to prove the closure property. Thus, by setting $\Lambda_A := \Sigma_A$, a silent self-stabilizing graph algorithm only needs to prove the convergence property and the correctness, where correctness is that all silent configurations are legitimate, that is, $\Sigma_A \subseteq \Lambda$

Now consider a silent self-stabilizing P algorithm and let $\Lambda \subseteq \Gamma$ be the set of all legitimate configurations in a input graph G, which conform to P. By the arguments in previous section, there exists a variable called *membership* and a variable called *information* which keeps the neighborhood information. Let Γ' be the vector space expanded by decisions of all nodes, and let $Proj : \Gamma \to \Gamma'$ be the projection function defined by

$$Proj((q_1, q_2, \ldots, q_n)) = (d_1, d_2, \ldots, d_n),$$

where $d_v = \mathrm{Dec}(q_v)$ for all $v \in V$. An equivalent definition of stableness can be given by setting the legitimate configurations of algorithm Λ_A equal to the silent configurations Σ_A, which conform to the desired property P as follows.

Definition 4.3.2. A silent self-stabilizing P algorithm A is

- S_1-stable : if $Proj(\Lambda_A) \subseteq \Lambda'$.
- S_2-stable : if $Proj(\Lambda_A) = \Lambda'$.
- S_3-stable : if $Proj(\Lambda_A) = \Lambda'$ and
  for all $\gamma \in Proj^{-1}(\Lambda')$ and $\gamma \to \gamma'$, $Proj(\gamma') = Proj(\gamma)$.
- S_4-stable : if $Proj^{-1}(\Lambda') = \Lambda_A$.

The level of stableness is shown by the index i of S_i. The larger index will be seen as being more stable. Note that these are increasingly stringent requirements; see the following theorem.

Theorem 4.3.3. *S_{i+1}-stableness implies S_i-stableness, for $i \in \{1, 2, 3\}$.*

Proof. We give one line proof for each index i:

$(S_4 \Rightarrow S_3)$: $Proj^{-1}(\Lambda') = \Lambda_A$ implies $\forall\gamma \in Proj^{-1}(\Lambda')$, $\gamma' = \gamma$; hence $Proj(\gamma') = Proj(\gamma)$.

$(S_3 \Rightarrow S_2)$: by definition, S_3 satisfies $Proj(\Lambda_A) = \Lambda'$.

$(S_2 \Rightarrow S_1)$: $\Lambda' \subseteq \Lambda'$. □

Another way to classify the degrees of stableness is to describe the convergence of an algorithm A.

Definition 4.3.4. If the desired property is P, we say an algorithm A is

- single-point-stable: if A converges to one specific point in Λ'.
- subset-stable: if A converges to a subset of Λ'.
- whole-set-stable: if A potentially can converge to every point in Λ'.
- no-membership-move: if A makes no membership move when the initial configuration is in Λ'.
- no-move: if A makes no move when the initial configuration is in Λ'.

For example, MDS algorithms `Turau9n`, `Goddard5n`, and `Well4n` are whole-set-stable, and MIS algorithm `Turau3n` is subset-stable with respect to MDS. The containment relations and the corresponding proofs are listed below.

- whole-set-stable $\Rightarrow$ subset-stable: $\Lambda' \subseteq \Lambda'$.
- single-point-stable $\Rightarrow$ subset-stable: For all $\lambda' \in \Lambda'$, $\{\lambda'\} \subseteq \Lambda'$.
- no-membership-move $\Rightarrow$ whole-set-stable: For all $\lambda' \in \Lambda'$, if A is initialized with λ', then the projection of the configuration after stabilizing is λ'.
- no-move $\Rightarrow$ no-membership-move: membership moves are moves.

Table 4.1: The levels of stableness and convergence of self-stabilizing algorithms.

	Stableness defined by		
Level	Degree	Convergence	Algorithms w.r.t. MDS
S_1-stable	P	subset-stable	MIS algorithms (e.g. `Turau3n`)
S_2-stable	P-covered	whole-set-stable	`Turau9n`, `Goddard5n`, `Well4n`
S_3-stable	P-stable	no-membership-move	?
S_4-stable	P-silent	no-move	none

We now summarize the levels of stableness and convergence in Table 4.1.

In the remaining tables, Dist-2 means Distance-2. Table 4.2 summarizes the levels of stableness of self-stabilizing MDS algorithms. As can be observed from Table 4.2, Algorithm `Well4n` has the best performance among known S_2-stable MDS algorithms.

Table 4.2: Self-stabilizing MDS algorithms on general graphs with different levels of stableness.

Reference	Degree	Model	Daemon	Complexity
Hedetniemi et al. [36]	MDS-covered	Normal	Central	$(2n+1)n$ moves
Xu et al. [83]	MDS-covered	Normal	Synchronous	$4n$ rounds
Turau [75]	MDS-covered	Normal	Distributed	$9n$ moves
Goddard et al. [30]	MDS-covered	Normal	Distributed	$5n$ moves
Sec. 3.2 in this thesis	MDS-covered	Normal	Distributed	$4n-2$ moves
Sec. 5.3 in this thesis	MDS-silent	Dist-2	Distributed	$2n-1$ moves

Similar to the tables in [34], we list the stableness of each self-stabilizing independent set and dominating set algorithms in Table 4.3 and Table 4.4.

Recall that for a graph $G = (V, E)$, a vertex subset S is independent if no two nodes in S are adjacent. A maximal independent set (MIS) is an independent set that is not properly contained in any independent set. Note that an MIS is also an MDS. A 1-maximal independent set (1-MIS) is an MIS, with the additional property that one cannot increase the cardinality of the independent set by removing one node and adding more nodes. A vertex subset of S is a k-packing if $d(u, v) > k$ for all pairs of distinct vertices u and v of S [61]. A maximal 2-packing (M2P) is a 2-packing that is not properly contained in any

2-packing.

In a graph $G = (V, E)$, a vertex subset S is called a dominating set (DS) if every vertex is either a member of S or is adjacent to a member of S. A minimal dominating set (MDS) is a dominating set such that no proper subset of it is a dominating set. A vertex subset S is a total dominating set (TDS) if each vertex of the graph is adjacent to a member in S. A vertex subset S is a k-dominating set (kDS) if each nonmember of S is adjacent to at least k members in S. When the positive integer $k = 1$, it is a question of (single) domination. The minimal total dominating sets (MTDS) and minimal k-dominating sets (MkDS) can be similarly defined. A function $f : V \to \mathbb{N}$ is $\{k\}$-dominating if for every $v \in V$ we have $\sum_{u\in N[v]} f(u) \geq k$. The case $k = 1$ is a normal dominating set [21].

In summery, a subset S of V is:

independent if $\forall i, j \in S$, $ij \notin E$.

maximal independent if S is independent and any subset of V properly containing S is not independent.

k-packing if $\forall i \in V$, either $i \in S$ or $i \in N_k(S)$.

dominating if $\forall i \in V$, either $i \in S$ or $i \in N(S)$.

minimal dominating if S is dominating and no proper subset of S is dominating.

total dominating if $\forall i \in V$, $i \in N(S)$.

k-dominating if $\forall i \in V$, either $i \in S$ or $|N(i) \cap S| \geq k$.

The stableness of each algorithm in Table 4.3 is S_1 if the output is replaced by MDS.

4.4 Multi-self-stabilization

In [17], Dolev and Herman introduced superstabilizing algorithms for dynamic distributed system. An algorithm is *superstabilizing* if it is (i) self-stabilizing, meaning that it is guaranteed to respond to any transient fault and eventually reach a legitimate configuration, and (ii) it is guaranteed to satisfy a *passage* predicate all the times when the system undergoes topology changes starting from a legitimate configuration. The passage configuration is a safety property that should hold while the algorithm makes progress towards the re-establishing legitimacy following a topology change. The illegitimate configurations are partitioned into two layers, depending on whether or not they satisfy a passage predicate.

The concept of fault containment of self-stabilizing algorithms was introduced by Ghosh et al. [25]. They gave a transformer which uses newly added local repair actions to modify the original self-stabilizing algorithm in order to improve the stabilization time when the system is subject to only one transient fault. It has been shown that the stabilization time of a fault-containing MIS self-stabilizing can be reduced to $O(\Delta)$ moves [55]. For other variations of self-stabilization and transformers, see [7, 19, 20, 52, 62] for references.

Similarly to superstabilizing, Kakugawa and Masuzawa formulated a concept of safe convergence in the frame work of self-stabilization in [49]. A safely converging self-stabilizing system defines two layers of legitimate configurations. One is a set of configurations in which the property or service is *feasible*, i.e., minimum quality of service is guaranteed. The other is a set of configurations in which the property or service is *optimal*. The *safe convergence* property guarantees that, from any initial configuration, a system quickly converges to a feasible configuration, and then, it continuingly moves to an optimal configuration without leaving the feasible configurations.

Kakugawa et al. took the MDS problem as an example. They considered the minimal dominating set problem and gave an algorithm called `Kakugawa` with safe convergence in which any dominating set is defined to be feasible; see Algorithm 6. In addition, Algorithm `Kakugawa` computes the lexicographically first minimal independent domination set which is defined to be optimal. Since there exists only one lexicographically first minimal independent domination set (MIDS) for a given graph, Algorithm `Kakugawa` is single-point-stable. The transition diagram of the algorithm `Kakugawa` is shown below; see Figure 4.2.

Algorithm 6 `Kakugawa`

```
variables
    Boolean $d_i \in \{0,1\}$;                                  // $i$ is a dominator iff $d_i = 1$
    pointer $m_i \in N[i]$;                                     // a dominator that node $i$ depends on

macros
    $LocalMax(i) \equiv \forall j \in N(i) : j < i$;            // true iff node $i$ is the maximum among its neighbors
    $ExIndDomNbr(i) \equiv \exists j \in N(i) : (d_j = 1) \wedge (m_j = j)$;
                                                                // true iff there exists an independent dominator neighbor
    $MaxIndDomNbr(i) \equiv \max\{j \in N(i) : (d_j = 1) \wedge (m_j = j)\}$;
                                                                // the max independent dominator in neighbors; NIL if none exists
    $NoDepend(i) \equiv \forall j \in N(i) : m_j \neq i$;       // true iff no neighbor depends on $i$

rules
    1: if $LocalMax(i) \wedge ((d_i \neq 1) \vee (m_i \neq i))$
           then $d_i := 1; m_i := i$;                           // If $i$ is the locally max, it becomes an independent dominator.
    2: if $\neg LocalMax(i) \wedge \neg ExIndDomNbr(i) \wedge ((d_i \neq 1) \vee (m_i \neq i))$
           then $d_i := 1; m_i := i$;                           // If there is no independent dominator neighbor, $i$ becomes an independent dominator.
    3: if $\neg LocalMax(i) \wedge ExIndDomNbr(i) \wedge (MaxIndDomNbr(i) < i) \wedge ((d_i \neq 1) \vee (m_i \neq i))$
           then $d_i := 1; m_i := i$;  // If $i$ is the max among independent dominator neighbors, it becomes an independent dominator.
    4: if $\neg LocalMax(i) \wedge ExIndDomNbr(i) \wedge (MaxIndDomNbr(i) > i)$
           $\wedge ((MaxIndDomNbr \neq m_i) \vee (d_i \neq 1))$
           then $d_i := 1; m_i := MaxIndDomNbr(i)$;             // If $i$ is not the max among independent dominator neighbors,
                                    // it becomes a dominator (for safety) and depends on the max independent dominator neighbor.
    5: if $\neg LocalMax(i) \wedge ExIndDomNbr(i) \wedge (MaxIndDomNbr(i) > i)$
           $\wedge (MaxIndDomNbr = m_i) \wedge (d_i \neq 0) \wedge NoDepend(i)$
           then $d_i := 0$;      // $i$ turns to be a dominatee only when (1) $i$ is not the max independent dominator among neighbors,
                                 // (2) $i$ depends on the max independent dominator neighbor, and (3) (for safety) no neighbor depends on $i$.
```

Under the synchronous daemon, Algorithm 6 converges to a domination set in one step, and further converges to the lexicographically first MIDS in at most $4D = O(D)$ steps, where D denotes the diameter of the input graph. The authors conjectured their algorithm still works under the distributed daemon. It can be proved that their conjecture appears correct, since in each step the configuration moves toward the optimal one (which is the lexicographically first MIDS). Unfortunately, the number of moves can be exponential in n. If the input graph is a path of n nodes in the increasing order, and initially all d_i in Algorithm 6 are set to 0. A bad example of execution is given by letting the distributed daemon always choose the privileged node with the lowest ID first, one at each time. Then it takes $\Omega(2^n)$ moves to reach the optimal configuration.

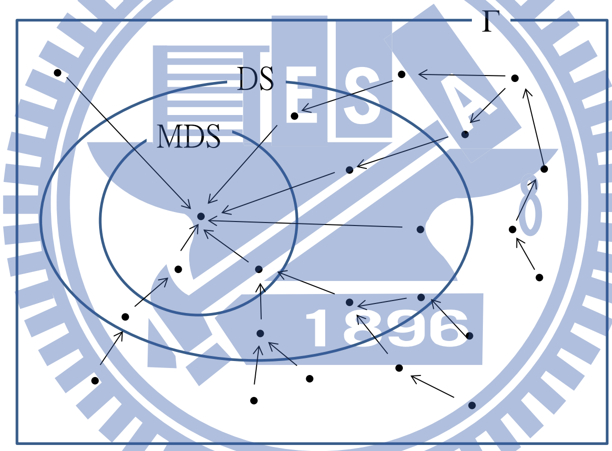

Figure 4.2: The transition diagram of the algorithm `Kakugawa`. A dominating set (DS) is feasible and a minimal dominating set (MDS) is optimal. The algorithm converges to a single point, which is the lexicographically first minimal independent domination set.

The concept of safe stabilization can be generalized as follows. Suppose there are a finite number of layers of legitimate configurations which we want the higher layer the better. Let $\Lambda_0 = \Gamma$ and Λ_k be the k-th layer of legitimate configuration, and moreover, $\Lambda_i \subset \Lambda_{i-1}$ for all $i = 1$ to k. We say a distributed algorithm is *multi-self-stabilizing* with respect to $\{\Lambda_1, \Lambda_2, \ldots, \Lambda_k\}$ if the following two properties hold for all $i = 1$ to k:

Convergence Starting from an arbitrary configuration in Λ_{i-1}, the distributed system reaches a configuration in Λ_i in a finite time.

Closure For any configuration in Λ_i, any configuration that follows is also in Λ_i.

Algorithm `Kakugawa` is an example of a multi-self-stabilizing algorithm with respect to three layers of legitimate configurations; see Figure 4.3. The first layer of legitimacy is domination, which makes the service feasible. The second layer of legitimacy is minimality, which provides a better quality of service. And the third layer of legitimacy is independence, which further reduces the number of dominators. Note that the Λ_0 is defined to be the set of all configurations. This makes the definition of multi-self-stabilizing and the original definition of self-stabilizing coincide and we can omit the prefix multi without confusion or ambiguity. Hence we merely say that Algorithm `Kakugawa` is self-stabilizing with respect to {DS, MDS, MIDS}.

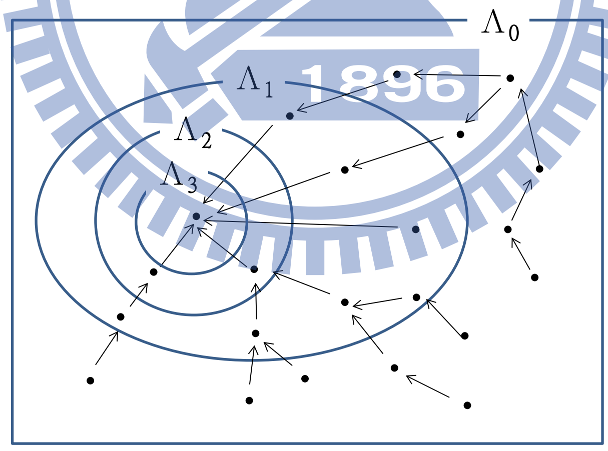

Figure 4.3: The transition diagram of a multi-self-stabilizing system with respect to three layers of legitimacy.

Table 4.3: Self-stabilizing independent set algorithms.

Reference	Output	Stableness	Required topology	Anonymous	Uniform	Daemon	Model	Complexity
Shukla et al. [67]	MIS	S_4	Arbitrary	Yes	Yes	Central	Normal	$\leq 2n$ moves
Ikeda et al. [45]	MIS	S_4	Arbitrary	No	Yes	Distributed	Normal	$\leq \frac{(n+2)(n+1)}{4}$ moves
Hedetniemi et al. [36]	MIS	S_4	Arbitrary	Yes	Yes	Central	Normal	$\leq 2n$ moves
Goddard et al. [32, 30]	MIS	S_4	Arbitrary	No	Yes	Synchronous	Normal	$\leq n$ rounds
Gairing et al. [21]	M2P	S_4	Arbitrary	Yes	Yes	Central	Dist-2	$O(n)$ moves
Shi et al. [66]	1-MIS	S_2	Tree	Yes	Yes	Central	Normal	$\leq n^2$ moves
Turau [75]	MIS	S_3	Arbitrary	No	Yes	Distributed	Normal	$\leq 3n$ moves
Goddard et al. [30]	MIS	S_4	Arbitrary	No	Yes	Synchronous	Normal	$\leq n$ rounds

Table 4.4: Self-stabilizing dominating set algorithms.

Reference	Output	Stableness	Required topology	Anonymous	Uniform	Daemon	Model	Complexity
Hedetniemi et al. [36]-1	DS	S_1	Arbitrary	Yes	Yes	Central	Normal	$\leq n-1$ moves
Hedetniemi et al. [36]-2	MDS	S_2	Arbitrary	Yes	Yes	Central	Normal	$\leq (2n+1)n$ moves
Xu et al. [83]	MDS	S_2	Arbitrary	No	Yes	Synchronous	Normal	$\leq 4n$ rounds
Goddard et al. [28, 29]	MTDS	S_2	Arbitrary	No	Yes	Central	Normal	$\leq 2^n$ moves
Turau [75]	MDS	S_2	Arbitrary	No	Yes	Distributed	Normal	$\leq 9n$ moves
Goddard et al. [30]	MDS	S_2	Arbitrary	No	Yes	Distributed	Normal	$\leq 5n$ moves
Kamei et al. [50]-1	MkDS	S_1	Tree	Yes	Yes	Central	Normal	$\leq n^2$ moves
Kamei et al. [50]-2	MkDS	S_1	Tree	No	Yes	Distributed	Normal	$\leq 3(n-1)^2$ moves
Huang et al. [43]	M2DS	S_1	Arbitrary	Yes	Yes	Central	Normal	$\leq 10n+1$ moves
Kamei et al. [51]	MkDS	S_1	Arbitrary	No	Yes	Synchronous	Normal	$\leq n$ rounds
Huang et al. [44]	M2DS	S_1	Arbitrary	No	Yes	Distributed	Normal	
Gairing et al. [21]	MDS	S_4	Arbitrary	Yes	Yes	Central	Dist-2	$\leq 2n$ moves
Turau et al. [76]	MkDS	S_4	Arbitrary	Yes	Yes	Central	Dist-2	$\leq 2n$ moves
Belhoul et al. [4]	MTDS	S_4	Arbitrary	Yes	Yes	Central	Dist-2	$\leq 2n$ moves
Sec. 3.2 in this thesis	MDS	S_2	Arbitrary	No	Yes	Distributed	Normal	$\leq 4n-2$ moves
Sec. 5.3 in this thesis	MDS	S_4	Arbitrary	No	Yes	Distributed	Dist-2	$\leq 2n-1$ moves

Chapter 5

The distance-2 model and self-stabilizing MDS-silent algorithms

The purpose of this chapter is to develop a self-stabilizing MDS-silent algorithm. Let k be a positive integer. In the previous chapter, we have defined the *distance-k model*. Recall that the *normal model* is the distance-1 model. For convenience, we use the *extended model* to refer to the distance-2 model. In the extended model, a node can instantaneously access the information of all nodes within distance two from it. Clearly, if a distributed system is in the distance-$(k+1)$ model, then it is in the distance-k model. See also [21].

A distributed system in the normal model can be implemented on an ad hoc network by using the beacon messages and the neighbor list messages of a node to inform neighbors of its continued presence and the change of the local state [30]. In this chapter we give a generalization to implement the distance-2 model by enlarging the beacon messages to contain the local states of nodes in one's neighborhood.

This chapter is organized as follows. Section 5.1 briefly reviews previous results in the distance-2 model. Section 5.2 generalizes the neighbor list messages in an ad hoc network to implements a distance-2 model by enlarging the beacon messages to contain the local states of nodes in one's neighborhood. Section 5.3 discuss the development of a self-stabilizing MDS-silent algorithm.

5.1 The extended model and the expression model

Before going further, we briefly review previous results in the extended model. In [21], the authors considered algorithms run in the extended model, where a node can instantaneously see the states of all nodes within distance two from it. The extended model is a distance-2 model.

In [76], the author considered a generalization of the extended model and called it the *expression model*. In particular, [21] proposed an $n(k+1)$-move self-stabilizing minimal $\{k\}$-dominating set algorithm; when $k = 1$, the algorithm finds an MDS using at most $2n$ moves. The paper [21] also showed how to implement the algorithm in the normal model by using a conversion mechanism ([76] called it a *transformer*). The paper [76] presented a $2n$-move self-stabilizing minimal k-dominating set algorithm; when $k = 1$, this algorithm finds an MDS. In [76], Turau introduced two techniques (the transformers $\mathcal{C}$ and $\mathcal{D}$) that can emulate algorithms for the expression model in the normal model and this technique improves [21].

5.2 Implementing the distance-2 model by message passing

A distributed system in the normal model can be implemented on an ad hoc network by using the beacon messages and the neighbor list messages of a node to inform neighbors of its continued presence and the change of the local state [30]. In this section we give a generalization to implement a distance-2 model by enlarging the beacon messages to contain the local states of nodes in one's neighborhood.

The distance-2 model assumes the following about the distributed system. A link-layer protocol at each node v maintains the identifiers and states of its neighbors in the neighbor list $nbl(v)$. Furthermore, after exchange the neighbor lists, each node v constructs the 2-neighbor list 2-$nbl(v)$ which contains the identifiers and states of 2-neighbors of v. In detail, each node periodically broadcasts a *beacon message*. When a neighboring node u broadcasts a beacon message to node v, it includes the local state of the node u in the algorithm. A beacon message provides information about its neighbor nodes synchronously, and a node takes action after receiving beacon messages from all neighboring nodes. When node v receives the beacon signal from a neighbor u which is not in $nbl(v)$, it adds u to its neighbor list to establishing link (u, v). After nodes v updating its neighbor list, node v broadcasts $nbl(v)$. All nodes in the neighborhood $N(v)$ know the existence of link (u, v) according the $nbl(v)$ message and update their 2-neighbor lists. In the meantime, node v updates 2-$nbl(v)$ according to the $nbl(u)$ message.

If node v does not receive the beacon message from u within a fixed period according the timer t_{vu}, it assumes the link (u, v) is no longer available and removes u from both $nbl(v)$ and 2-$nbl(v)$. A node takes action only after receiving beacon messages or neighbor list messages from its neighboring nodes. The self-stabilizing algorithm can tolerate transient faults such as allowing nodes to join or leave the system and/or link creations or failures. In this implementation, one move of nodes induces $O(\Delta)$ broadcasting messages. As the convergence time to a legitimate configuration have to be small with respect to the periodicity of change, the quickness of self-stabilization is crucial.

5.3 Developing an MDS-silent algorithm

The purpose of this section is to discuss the development of a self-stabilizing MDS-silent algorithm.

The algorithms in both [21] and [76] operate with a central daemon; we now propose an algorithm that operates with a distributed daemon. In particular, we propose a self-stabilizing MDS-silent algorithm `Well2n` under an unfair distributed daemon in a distance-2 model. Our algorithm uses a two-valued variable *state*, which has values `IN` and `OUT`. We use the same terminology as in Section 3.2 unless otherwise indicated.

To formally define the rules of `Well2n`, three predicates defined for each node v are needed: $noInNbr$, $noBtNbr_2$, and $noDpNbr_2$; see Algorithm 7. Notice that $noBtNbr_2$ and $noDpNbr_2$ require distance-2 knowledge since v has to check the states of its neighbor's neighbors. Algorithm `Well2n` uses only two simple rules: rule S1 is regarded as the entering rule and rule S2, the leaving rule.

Algorithm 7 `Well2n`

```
variables
    $v.state \in \{\texttt{IN}, \texttt{OUT}\}$;                                  // $S = \{v : v.state = \texttt{IN}\}$

macros
    $noInNbr(v) \equiv \nexists w \in N(v) : w.state = \texttt{IN}$;
    $noBtNbr_2(v) \equiv \nexists w \in N(v) : w.state = \texttt{OUT} \wedge w.id < v.id \wedge w$ has no IN neighbor;
    $noDpNbr_2(v) \equiv \nexists w \in N(v) : w.state = \texttt{OUT} \wedge w$ has exactly one IN neighbor;

rules
    S1: if $v.state = \texttt{OUT} \wedge noInNbr(v) \wedge noBtNbr_2(v)$
            then $v.state := \texttt{IN}$;                                        // enter $S$
    S2: if $v.state = \texttt{IN} \wedge \neg noInNbr(v) \wedge noDpNbr_2(v)$
            then $v.state := \texttt{OUT}$;                                       // leave $S$
```

Recall that the output of Algorithm 7 is $S = \{v : v.state = \texttt{IN}\}$. An MIS algorithm usually applies the following entering rule and leaving rule: "A node having no neighbor in

S joins S and a node having a neighbor in S leaves S." Algorithm 7 modifies the entering rule and leaving rule to be: "A node enters S if (i) it has no neighbor in S and (ii) its identifier is smaller than any `OUT` neighbor having no `IN` neighbor, and a node leaves S if (a) it has a neighbor in S and (b) every neighbor is either in S or has at least two neighbors in S."

We now prove the correctness of Algorithm 7. First, by using the method for the correctness of `Well4n` in Lemma 3.3.1, one can prove that in any configuration in which no node is privileged, the set S is a minimal dominating set of G.

Lemma 5.3.1. *In any configuration in which no node is privileged, the set S is a minimal dominating set for G.*

Proof. Suppose to the contrary that S is not a minimal dominating set for G. Then either (i) S is not a dominating set or (ii) S is a dominating set but not minimal. Consider case (i). Since S is not a dominating set, there exists an `OUT` node having no `IN` neighbor. Let u be such a node with the minimum identifier. Since u has no `IN` neighbor and no better neighbor, rule S1 is enabled, which is a contradiction. Now consider case (ii). Since S is a dominating set but not minimal, there must exist at least one node $u \in S$ such that $S \backslash \{u\}$ is also a dominating set for G. Then $|N(u) \cap S| \geq 1$ and $|N(w) \cap S| \geq 2$ for all w in $N(u) \backslash S$. Since both $\neg noInNbr(u)$ and $noDpNbr_2(u)$ are *true*, rule S2 is enabled on node u, which is a contradiction. □

We now show that Algorithm `Well2n` is MDS-silent. Note that in each node, rule S1 is not enabled if S is dominating, and rule S2 is not enabled if S is minimal.

Lemma 5.3.2. *Algorithm 7 will not make any move if the initial configuration conforms to MDS.*

Proof. Recall that `Well2n` modifies the entering rule and leaving rule of an MIS algorithm to be: "A node enters S if (i) it has no neighbor in S and (ii) its identifier is smaller than any `OUT` neighbor having no `IN` neighbor, and a node leaves S if (a) it has a neighbor in S and (b) every neighbor is either in S or has at least two neighbors in S." Suppose the initial configuration conforms to MDS. Let S be the set of nodes with $state$ = `IN`. First consider an arbitrary node u in S. The only rule can be enabled on u is the leaving rule. Since S is an MDS, it is impossible that u has an `IN` neighbor but has no dependent neighbor; otherwise $S \backslash \{u\}$ is also a dominating set for G. Thus, at least one of (a) and (b) is $false$ and the leaving rule cannot be enabled on node u. Next consider an arbitrary node w not in S. The only rule that can be enabled on w is the entering rule. Since S is an MDS, it is impossible that w has no `IN` neighbor. Thus, (i) is $false$ and the entering rule cannot be enabled on node w. We have this lemma. □

It is not difficult to see that if a node executes the entering rule, then it enters S and will never leave S afterward; furthermore, neighbors of this node will not enter S. See the following lemma.

Lemma 5.3.3. *If a node executes rule S1, then it will never leave. Furthermore, neighbors of this node will not enter the set S.*

Proof. Let v be a node that executes rule S1. At this moment all neighbors of v have state `OUT` and with identifier larger then v. Thus, none of these neighbors can execute rule S1 or rule S2. After v enters S, its neighbors have at least one `IN` neighbor v so they cannot execute rule S1. The only rule that v can execute next is rule S2, but in order to do so, one neighbor of v has to change into state `IN` first. As long as v is in state `IN` this is impossible. Therefore, v will never execute any rule, and its neighbors will not move, either. □

We now give the main theorem of this section.

Theorem 5.3.4. *Algorithm 7 is self-stabilizing and* MDS-silent *under an unfair distributed daemon in the distance-*2 *model. It stabilizes after at most* $2n - 1$ *moves with a minimal dominating set, where* n *is the number of nodes.*

Proof. The theorem follows from Lemmas 5.3.1, 5.3.2, and 5.3.3. The "minus one" part comes from the fact that in a connected graph of order larger than 1, the size of any MDS is at most $n - 1$. □

To see $2n - 1$ in Theorem 5.3.4 is tight, consider the star graph of order n with initial state of each node to be `IN`. Before ending this section, we would like to point out that a preliminary version of this thesis has discussed how to execute our MDS-silent algorithm; see [11]. In particular, a transformation from the distance-2 model to the distance-1 model is needed. Although the transformers described in [21] and [76] provide ways to transform from an algorithm assuming distance-2 model and central daemon to the distance-1 model, the output algorithm is still not MDS-silent even when we relax the definition of stableness to consider only "membership moves".

Chapter 6

The signed star domination number of Cayley graphs

The purpose of Chapter 6 is to find out the signed star domination number (a variant of the domination number) of Cayley digraphs and Cayley graphs. A function $f : E(G) \to \{-1, 1\}$ is called a *signed star dominating function* (SSDF) on G if $\sum_{e \in E(v)} f(e) \geq 1$ for every $v \in V(G)$, where $E(v)$ is the set of all edges incident with v. The *signed star domination number* of G is defined as $\gamma_{SS}(G) = \min\{\sum_{e \in E(G)} f(e) | \ f$ is an SSDF on $G\}$. In this chapter, we obtain exact values for the signed star domination number of all Cayley digraphs $Cay_D(\Gamma, S)$ and certain classes of Cayley graphs $Cay(\Gamma, \Omega)$. Throughout this chapter, we write $\{u, v\}$ for an edge with endpoints u and v if undirected graphs are considered. For directed graphs, we write (u, v) for "there is an arc from u to v", that is, the first vertex of the ordered pair is the *tail* of the arc, and the second is the *head*.

This chapter is organized as follows. In Section 6.1, we give basic definitions and review previous results. In Sections 6.2 and 6.3, we study the signed star domination of the Cayley digraphs and Cayley graphs, respectively. In Section 6.4, we study the signed star domination of $\{2, 1\}$-factorable graphs (defined later). A preliminary version of this chapter appeared as [9]. Note that these solutions are from a joint work with Chelvam and Kalaimurugan.

6.1 Definitions and previous results

Our graph terminologies are standard unless otherwise indicated; see [35, 79]. Let G be a simple connected graph with vertex set $V(G)$ and edge set $E(G)$. For a vertex $v \in V(G)$, let $E(v) = \{\{u, v\} \in E(G) \mid u \in V(G)\}$. A function $f : E(G) \to \{-1, 1\}$ is called a *signed star dominating function* (SSDF) on G if $\sum_{e\in E(v)} f(e) \geq 1$ for every $v \in V(G)$. The *signed star domination number* of G is defined as $\gamma_{SS}(G) = \min \{\sum_{e\in E(G)} f(e) \mid f$ is an SSDF on $G\}$ and such an f attaining the minimum value is called a *minimum* SSDF on G. Note that $\gamma_{SS}(G)$ is well-defined only if G contains no isolated vertex. A set $\{f_1, f_2, \dots, f_d\}$ of SSDFs on G with the property that $\sum_{i=1}^{d} f_i(e) \leq 1$ for each $e \in E(G)$ is called a *signed star dominating family* (of functions) on G. The maximum number of functions in a signed star dominating family on G is called the *signed star domatic number* of G and is denoted by $d_{SS}(G)$. Let D be a digraph with vertex set $V(D)$ and arc set $A(D)$. For each vertex $v \in V(D)$, let $A(v)$ be the set of all out-going arcs from v. By replacing $E(v)$ by $A(v)$, one can define the SSDF on D and $\gamma_{SS}(D) = \min\{\sum_{a\in A(D)} f(a) \mid f$ is an SSDF on $D\}$.

Let Γ be a finite nontrivial group with the identity element ι and S be a nonempty subset of Γ. The *Cayley digraph* $Cay_D(\Gamma, S)$ is the digraph whose vertices are the elements of Γ, and there is an arc from α to $\alpha\sigma$ whenever $\alpha \in \Gamma$ and $\sigma \in S$. A subset S of Γ is *symmetric* if $\sigma^{-1} \in S$ whenever $\sigma \in S$. A *generating set* of Γ is a subset that is not contained in any proper subgroup of Γ. Let Ω be a symmetric generating subset of nonidentity elements of Γ. The *Cayley graph* $Cay(\Gamma, \Omega)$ corresponding to Γ and Ω is the ordinary graph with vertex set Γ and edge set $E = \{\{\alpha, \alpha\sigma\} \mid \alpha \in \Gamma, \sigma \in \Omega\}$. Note that in the Cayley graph, the edges α to $\alpha\sigma$ and $\alpha\sigma$ to α are considered one and the same.

In the past few years, several types of domination problems in graphs have been studied by various authors, most of these belonging to the vertex domination. Xu [81] initiated the study of signed star domination numbers of graphs and for more study on signed star domination, one can refer [3, 47, 64, 78, 80, 82]. Following are some known results on signed star domination numbers of graphs without isolated vertices.

Theorem 6.1.1. *[80] For all graphs G of order $n \geq 2$, $\gamma_{SS}(G) \geq \lceil \frac{n}{2} \rceil$.*

Theorem 6.1.2. *[80] For all graphs G of order $n \geq 4$, $\gamma_{SS}(G) \leq 2n - 4$, and this bound is sharp.*

Theorem 6.1.3. *[3] Let G be a graph of size m with signed star domination number $\gamma_{SS}(G)$ and signed star domatic number $d_{SS}(G)$. Then $\gamma_{SS}(G) \cdot d_{SS}(G) \leq m$.*

Theorem 6.1.4. *[3] Let G be an r-regular and 1-factorable graph. Then*

$$d_{SS}(G) = \begin{cases} r & \textit{when } r \textit{ is odd,} \\ \frac{r}{2} & \textit{when } r \equiv 2 \pmod 4, \\ \frac{r}{2} - 1 & \textit{when } r \equiv 0 \pmod 4. \end{cases}$$

Theorem 6.1.5. *[3] Let G be factorable into r Hamiltonian cycles. Then*

$$d_{SS}(G) = \begin{cases} r & \textit{when } r \textit{ is odd,} \\ r - 1 & \textit{when } r \textit{ is even.} \end{cases}$$

6.2 Signed star domination of the Cayley digraphs

In this section, we obtain the signed star domination number of the Cayley digraph $D = Cay_D(\Gamma, S)$ constructed out of a finite group Γ and a nonempty subset $S = \{\sigma_1, \sigma_2, \ldots, \sigma_r\}$

of Γ. Note that each vertex of D has the same out-degree r. Given $\sigma_i \in S$, let $\mathcal{C}_i$ denotes the set of all the arcs generated by σ_i, i.e., $\mathcal{C}_i = \{(\alpha, \alpha\sigma_i)\} \mid \alpha \in \Gamma\}$. Then $A(D) = \bigcup_{i=1}^{r} \mathcal{C}_i$.

Fact 1. Let f be an SSDF on D and let $v \in V(D)$. If $\sum_{a \in A(v)} f(a) = 1$ for every vertex v of odd out-degree and $\sum_{a \in A(u)} f(a) = 2$ for every vertex u of even out-degree, then f is a minimum SSDF on D.

Theorem 6.2.1. *Let Γ be a finite group of order n and $S = \{\sigma_1, \sigma_2, \ldots, \sigma_r\}$ be a nonempty subset of Γ. Then*

$$\gamma_{SS}(Cay_D(\Gamma, S)) = \begin{cases} n & \text{if } r \text{ is odd,} \\ 2n & \text{if } r \text{ is even.} \end{cases}$$

Proof. By the construction of Cayley digraph $D = Cay_D(\Gamma, S)$, $A(D)$ is the disjoint union of $\mathcal{C}_i$'s, $1 \le i \le r$. Define a function $f : A(D) \to \{-1, 1\}$ by

$$f(a) = \begin{cases} (-1)^{i-1} & \text{if } a \in \mathcal{C}_i,\ 1 \le i \le r-1, \\ 1 & \text{if } a \in \mathcal{C}_r. \end{cases}$$

Now, for each vertex $v \in V(D)$

$$\sum_{a \in A(v)} f(a) = \begin{cases} 1 & \text{if } r \text{ is odd,} \\ 2 & \text{if } r \text{ is even.} \end{cases}$$

By Fact 1, f is a minimum SSDF on D; thus we have this theorem. □

Example 6.2.2. Consider the group $\mathbb{Z}_6 = \{0, 1, \ldots, 5\}$ under addition and $S = \{1, 2, 5\}$. The signed star domination number of $Cay_D(\mathbb{Z}_6, S)$ is 6 and the corresponding SSDF is given in Figure 6.1.

Now we generalize the result of Cayley digraphs to general digraphs. Let D be an arbitrary digraph. For a vertex $v \in V(D)$, we say v is an *odd vertex* if its out-degree is odd.

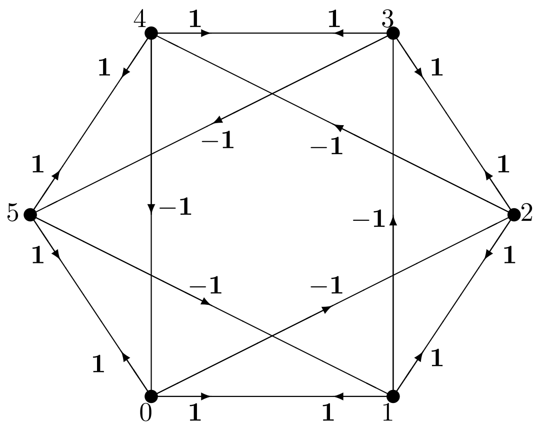

Figure 6.1: An illustration of the signed star domination number of $Cay_D(\mathbb{Z}_6, \{1, 2, 5\})$ and the corresponding SSDF.

See the following theorem.

Theorem 6.2.3. *Let D be a digraph of order n in which every vertex has at least one out-going arc. Then*

$$\gamma_{SS}(D) = 2n - k,$$

where k is the number of odd vertices.

Proof. Let $\deg^+(v)$ denote the out-degree of a vertex v. Since an SSDF on D requires every vertex $v \in V(D)$ to sum up the values of the arcs in $A(v)$, each arc only contributes its value to its tail vertex. Thus we can use the following procedure to assign the arcs in D either -1 or 1: For each vertex v, ignore all of its incoming arcs and assign 1 and -1, alternatively, to its out-going arcs (the first arc receives 1, the second arc receives -1, and so on); reassign 1 to the last out-going arc if this arc receives -1.

The above procedure constructs a function $f : A(D) \to \{-1, 1\}$ on D. For each v, we have

$$\sum_{a \in A(v)} f(a) = \begin{cases} 1 & \text{if } \deg^+(v) \text{ is odd,} \\ 2 & \text{if } \deg^+(v) \text{ is even.} \end{cases}$$

By Fact 1, f is a minimum SSDF on D. Therefore $\gamma_{SS}(D) = \sum_{a \in A(D)} f(a) = k + 2(n -$

$k) = 2n - k$. □

6.3 Signed star domination of the Cayley graphs

In this section, we give the signed star domination number of the Cayley graph $G = Cay(\Gamma, \Omega)$ constructed out of a finite nontrivial group Γ and a symmetric generating subset Ω of nonidentity elements of Γ. In the Cayley graph G, we arrange the elements of the symmetric generating set Ω of Γ in the form

$$\Omega = \{\sigma_1, \sigma_2, \ldots, \sigma_{2s}, \tau_1, \tau_2, \ldots, \tau_t\}$$

satisfying

$$\sigma_i^{-1} = \sigma_{i+s},\ 1 \le i \le s \ \text{ and }\ \tau_j^{-1} = \tau_j,\ 1 \le j \le t.$$

That is, σ_i's are used to represent elements in Ω of order larger than 2 and τ_j's, elements in Ω of order exactly 2.

Do notice that in a Cayley graph, σ_i and σ_{i+s} generate the same cycles. Thus we define two notations $\mathcal{C}_i$ and $\mathcal{M}_j$ as follows (these notations will also be used in the next section). For $1 \le i \le s$, let $\mathcal{C}_i$ denote the set of edges of all the (undirected) cycles in G generated by σ_i (or σ_{i+s}). Similarly, for $1 \le j \le t$, let $\mathcal{M}_j$ denote the matching in G generated by τ_j. By the above setting, the edge set of G has a partition $\{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_s, \mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_t\}$, i.e.,

$$E(G) = \left(\bigcup_{i=1}^{s} \mathcal{C}_i\right) \cup \left(\bigcup_{j=1}^{t} \mathcal{M}_j\right).$$

Example 6.3.1. Let $\Gamma = \langle \sigma \rangle$ be a finite cyclic group and $\Omega = \{\sigma, \sigma^{-1}\}$. Then $\Omega = \{\sigma_1, \sigma_2\}$, $s = 1$ and $t = 0$. When Γ is the dihedral group $D_5 = \{\iota = \sigma^5 = \tau^2, \sigma, \sigma^2, \sigma^3, \sigma^4, \tau, \sigma\tau, \sigma^2\tau, \sigma^3\tau, \sigma^4\tau = \tau\sigma\}$ and $\Omega = \{\tau, \sigma\tau\}$, as per our notation $\Omega = \{\tau_1, \tau_2\}$, $s = 0$ and $t = 2$.

Fact 2. Let f be an SSDF on G. If $\sum_{e\in E(v)} f(e) = 1$ for every vertex v of odd degree and $\sum_{e\in E(u)} f(e) = 2$ for every vertex u of even degree, then f is a minimum SSDF on G. Thus $\gamma_{SS}(G) \geq n - \frac{k}{2}$, where k is the number of odd degree vertices in G.

Lemma 6.3.2. *Let G be an r-regular graph. Then we have*

$$\gamma_{SS}(G) \geq \begin{cases} \frac{n}{2} & \text{if } r \text{ is odd,} \\ n & \text{if } r \text{ is even,} \\ n+1 & \text{if } r \text{ is a multiple of } 4 \text{ and } n \text{ is odd.} \end{cases}$$

Proof. The first two cases are deduced from Fact 2. For the last case, suppose to the contrary that there exists an SSDF f on G with $\sum_{e\in E(G)} f(e) = n$. Then $f^{-1}(1) = \{e \in E(G) \mid f(e) = 1\}$ forms a subgraph of G containing odd number of odd degree vertices, which is impossible. □

Lemma 6.3.3. *Let Γ be a finite group of odd order. Then one cannot find a symmetric generating set Ω of Γ with odd $|\Omega|$.*

Proof. If $|\Omega|$ is odd, then Ω contains an element of order 2. By Lagrange's Theorem, the order of any element of Γ divides the order of Γ. Therefore $|\Gamma|$ is even. □

For an r-regular graph G, if we can find a $\lfloor \frac{r-1}{2} \rfloor$-matching M of G, then by setting $f(e) = -1$ for all edges $e \in M$, we have $\gamma_{SS} \leq |E(G)| - 2|M|$.

Theorem 6.3.4. *Let Γ be a finite group of even order n, and let Ω be a symmetric generating set of Γ. If $|\Omega|$ is odd, then $\gamma_{SS}(Cay(\Gamma,\Omega)) = \frac{n}{2}$.*

Proof. Let $\Omega = \{\sigma_1, \sigma_2, \ldots, \sigma_{2s}, \tau_1, \tau_2, \ldots, \tau_t\}$. Since $|\Omega|$ is odd, t must be odd. Let $G = Cay(\Gamma, \Omega)$. Define a function $f : E(G) \to \{-1, 1\}$ by

$$f(e) = \begin{cases} (-1)^{i-1} & \text{if } e \in \mathcal{C}_i,\ 1 \leq i \leq s, \\ (-1)^{s+j-1} & \text{if } e \in \mathcal{M}_j,\ 1 \leq j \leq t. \end{cases}$$

Then, for every vertex $v \in V(G)$, $\sum_{e\in E(v)} f(e) = 1$. Thus $\gamma_{SS}(G) \le \sum_{e\in E(G)} f(e) = \frac{n}{2}$. By Fact 2 and Lemma 6.3.2, f is a minimum SSDF on G and $\gamma_{SS}(G) = \frac{n}{2}$. $\square$

Theorem 6.3.5. *Let Γ be a finite group of order n, and let Ω be a symmetric generating set of Γ with even $|\Omega|$. If Ω has an odd number of distinct $\{\sigma, \sigma^{-1}\}$ pairs, or contains an element of even order, then $\gamma_{SS}(Cay(\Gamma, \Omega)) = n$.*

Proof. Let $G = Cay(\Gamma, \Omega)$. Arrange $\Omega = \{\sigma_1, \sigma_2, \ldots, \sigma_{2s}, \tau_1, \tau_2, \ldots, \tau_t\}$ with the condition $\sigma_i^{-1} = \sigma_{i+s}, \tau_j^{-1} = \tau_j$, $1 \le i \le s$, $1 \le j \le t$. Since $|\Omega|$ is even, t must be even. Define a function $f : E(G) \to \{-1, 1\}$ as follows:

Case 1: s is odd. Define f by

$$f(e) = \begin{cases} (-1)^{i-1} & \text{if } e \in \mathcal{C}_i,\ 1 \le i \le s, \\ (-1)^{j-1} & \text{if } e \in \mathcal{M}_j,\ 1 \le j \le t. \end{cases}$$

Case 2: s is even and $t \ne 0$. Define f by

$$f(e) = \begin{cases} (-1)^{i-1} & \text{if } e \in \mathcal{C}_i,\ 1 \le i \le s, \\ (-1)^{j-1} & \text{if } e \in \mathcal{M}_j,\ 1 \le j \le t-1, \\ 1 & \text{if } e \in \mathcal{M}_t. \end{cases}$$

Case 3: s is even, $t = 0$ and there exists an element $\sigma \in \Omega$ of even order at least 4. In this case, without loss of generality, we assume that Ω is arranged so that σ_s is an element of even order $\ell \ge 4$. Let $I = n/\ell$ be the number of cycles in $\mathcal{C}_s$, i.e., $\mathcal{C}_s = \mathcal{C}_s(\alpha_1) \cup \mathcal{C}_s(\alpha_2) \cup \cdots \cup \mathcal{C}_s(\alpha_I)$ with each cycle of length ℓ. Define f by

$$f(e) = \begin{cases} (-1)^{i-1} & \text{if } e \in \mathcal{C}_i,\ 1 \le i \le s-1, \\ (-1)^{j} & \text{if } e = \{\alpha_k\sigma_s^j, \alpha_k\sigma_s^{j+1}\} \in \mathcal{C}_s,\ 1 \le k \le I,\ 1 \le j \le \ell. \end{cases}$$

For all cases, since $|\Omega|$ is even, we have $\sum_{e\in E(v)} f(e) = 2$ for every vertex $v \in V(G)$. By Fact 2 and Lemma 6.3.2, f is a minimum SSDF on G and $\gamma_{SS}(G) = n$. $\square$

Theorem 6.3.6. *Let Γ be a finite cyclic group of odd order n. Assume that Ω is a symmetric subset containing a generator of Γ. If $|\Omega|$ is a multiple of 4, then $\gamma_{SS}(Cay(\Gamma,\Omega)) = n+1$.*

Proof. By Lemma 6.3.3, Ω contains no element of order 2. Let $\Omega = \{\sigma_1, \sigma_2, \ldots, \sigma_{2s}\}$. Since $|\Omega|$ is a multiple of 4, s must be even. Let $G = Cay(\Gamma, \Omega)$. Without loss of generality, assume that $\sigma_s \in \Omega$ is a generator, i.e., $\Gamma = \langle \sigma_s \rangle$ and $\mathcal{C}_s$ is a cycle of length n. Define a function $f : E(G) \to \{-1, 1\}$ by

$$f(e) = \begin{cases} (-1)^{i-1} & \text{if } e \in \mathcal{C}_i,\ 1 \le i \le s-1, \\ (-1)^{j-1} & \text{if } e = \{\sigma_s^{j-1}, \sigma_s^{j}\} \in \mathcal{C}_s,\ 1 \le j \le n. \end{cases}$$

Since $|\Omega|$ is even, for each vertex v except ι, $\sum_{e \in E(v)} f(e) = 2$. Note that $\sum_{e\in E(\iota)} f(e) = 4$. Hence $\gamma_{SS}(G) \le \sum_{e \in E(G)} f(e) = n+1$. By Lemma 6.3.2, $\gamma_{SS}(G) \ge n+1$ and so $\gamma_{SS}(G) = n+1$. □

Example 6.3.7. Consider the group $\mathbb{Z}_7 = \{0, 1, \ldots, 6\}$ under addition and $\Omega = \{1, 2, 5, 6\}$. Then $\gamma_{SS}(Cay(\mathbb{Z}_7, \Omega)) = 8$ and the corresponding SSDF is given in Figure 6.2.

By Theorems 6.3.4, 6.3.5 and 6.3.6, we have the following three corollaries.

Corollary 6.3.8. *[64, Lemma 5] Let $\mathbb{Z}_n$ be the cyclic group of order $n \ge 2$ with identity 0. Then $Cay(\mathbb{Z}_n, \mathbb{Z}_n \backslash \{0\})$ is isomorphic to the complete graph K_n and*

$$\gamma_{SS}(K_n) = \begin{cases} \frac{n}{2} & \text{if } n \equiv 0, 2 \pmod 4, \\ n & \text{if } n \equiv 3 \pmod 4, \\ n+1 & \text{if } n \equiv 1 \pmod 4. \end{cases}$$

Corollary 6.3.9. *[64, Lemma 6] Let $\mathbb{Z}_n$ be the cyclic group of order $n \ge 3$ with generators $1, -1$. Then $Cay(\mathbb{Z}_n, \{1, -1\})$ is isomorphic to the cycle C_n and $\gamma_{SS}(C_n) = n$.*

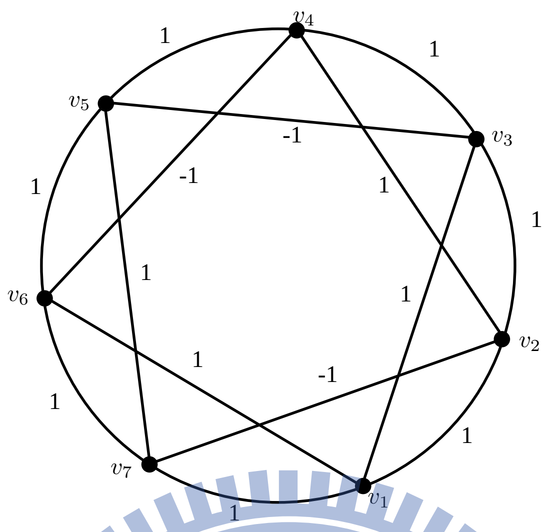

Figure 6.2: An illustration of the signed star domination number of $Cay(\mathbb{Z}_7, \{1, 2, 5, 6\})$ and the corresponding SSDF.

Corollary 6.3.10. *[64, Lemma 6] Let $\mathbb{Z}_{2n}$ be the cyclic group of order $2n$. Then $Cay(\mathbb{Z}_{2n}, \{1, 3, \ldots, 2n-1\})$ is isomorphic to the complete bipartite graph $K_{n,n}$ and*

$$\gamma_{SS}(K_{n,n}) = \begin{cases} n & \text{if } n \text{ is odd,} \\ 2n & \text{if } n \text{ is even.} \end{cases}$$

Example 6.3.11. Consider the group $\mathbb{Z}_{15} = \{0, 1, \ldots, 14\}$ under addition and $\Omega = \{3, 5, 10, 12\}$. Then $\gamma_{SS}(Cay(\mathbb{Z}_{15}, \Omega)) = 16$ and the corresponding SSDF is given in Figure 6.3.

From Example 6.3.11, we have the following conjecture.

Conjecture 6.3.12. Let Γ be a finite group of order n, and let $\Omega = \{\sigma_1, \sigma_2, \ldots, \sigma_{2s}\}$ be a symmetric generating set of Γ. If s is even and no element of Ω is of even order, then

$$\gamma_{SS}(Cay(\Gamma, \Omega)) = \begin{cases} n & \text{if } n \text{ is even,} \\ n+1 & \text{if } n \text{ is odd.} \end{cases}$$

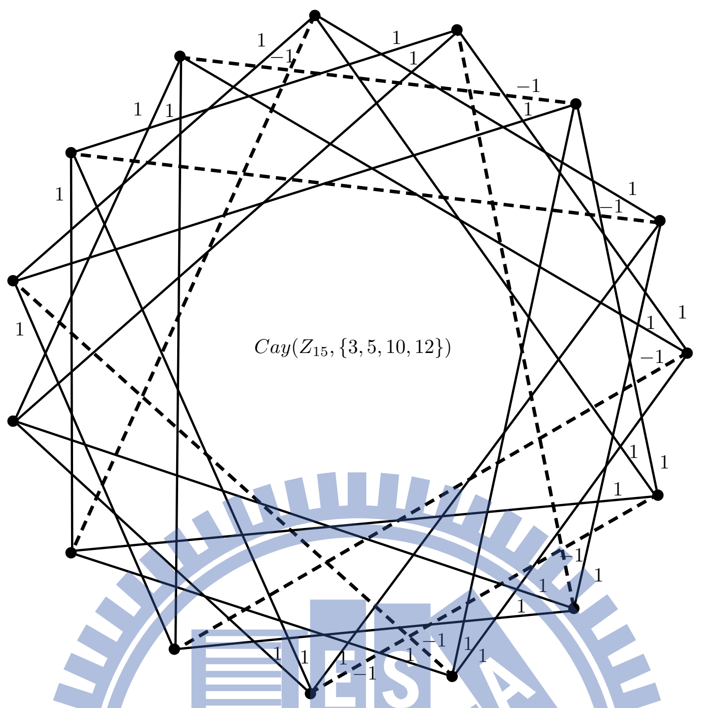

Figure 6.3: An illustration of the signed star domination number of $Cay(\mathbb{Z}_{15}, \{3, 5, 10, 12\})$ and the corresponding SSDF.

6.4 Signed star domination of $\{2, 1\}$-factorable graphs

Before going further, we introduce a new definition: A graph is $\{2, 1\}$-*factorable* if it can be decomposed into 2-factors and/or 1-factors. For the definitions of "k- factor"and "k-factorable"refer to [79].

Theorem 6.4.1. *All Cayley graphs are* $\{2, 1\}$-*factorable graphs.*

Proof. For Cayley graphs $G = Cay(\Gamma, \Omega), \mathcal{C}_i$'s and $\mathcal{M}_j$'s (defined in Section 6.3) are 2-factors and 1-factors, respectively, and hence G is $\{2, 1\}$-factorable. □

Notice that although every Cayley graph is $\{2, 1\}$-factorable, a $\{2, 1\}$-factorable graph may not be a Cayley graph. For example, the Petersen graph and the dodecahedral graph are $\{2, 1\}$-factorable but not Cayley graphs [60]. Notice also that a $\{2, 1\}$-factorable graph is regular but it is not necessarily 1-factorable; the Petersen graph is an example.

The property of Cayley graphs used in the proofs of Theorems 6.3.4, 6.3.5 and 6.3.6 is that: A Cayley graph can be decomposed into a bunch of $\mathcal{C}_i$'s and $\mathcal{M}_j$'s. Since $\mathcal{C}_i$'s are 2-factors and $\mathcal{M}_j$'s are 1-factors, we now extend Theorems 6.3.4, 6.3.5 and 6.3.6 to $\{2,1\}$-factorable graphs in the following three theorems. We will omit the proofs of these theorems since we can replace $\mathcal{C}_i$'s with 2-factors and $\mathcal{M}_j$'s with 1-factors in the proofs of Theorems 6.3.4, 6.3.5 and 6.3.6.

Theorem 6.4.2. *Let G be an r-regular and $\{2,1\}$-factorable graph of order n. If r is odd, then $\gamma_{SS}(G) = \frac{n}{2}$.*

Theorem 6.4.3. *Let G be an r-regular and $\{2,1\}$-factorable graph of order n with even r. If there exists a $\{2,1\}$-decomposition of G containing odd number of 2-factors, or a 1-factor, or a 2-factor consisting of even cycles, then $\gamma_{SS}(G) = n$.*

Theorem 6.4.4. *Let G be an r-regular and 2-factorable graph of odd order n. If there exists a 2-factorization of G containing a Hamiltonian cycle and r is a multiple of 4, then $\gamma_{SS}(G) = n + 1$.*

We now have a conjecture.

Conjecture 6.4.5. Let G be an r-regular and 2-factorable graph of order n. Then

$$\gamma_{SS}(G) = \begin{cases} n & \text{if } r \equiv 2 \pmod 4 \text{ or } n \text{ is even,} \\ n+1 & \text{if } r \equiv 0 \pmod 4 \text{ and } n \text{ is odd.} \end{cases}$$

Chapter 7

Conclusions

In this chapter, we present a summary of this thesis, and we discuss some directions for further research.

A distributed system is self-stabilizing if, regardless of the initial state, the system guarantees to reach a legitimate state in a finite time. In 2007, Turau proposed the first linear-time self-stabilizing algorithm for the minimal dominating set (MDS) problem under an unfair distributed daemon [75]; this algorithm stabilizes in at most $9n$ moves, where n is the number of nodes. In 2008, Goddard et al. [30] proposed a $5n$-move algorithm. The main result of this thesis is a $(4n-2)$-move self-stabilizing algorithm for the MDS problem using an unfair distributed daemon; the bound $4n-2$ is tight. The model that we use is the normal model, also called the distance-1 model. It is challenging to design a self-stabilizing MDS using a distributed daemon that makes fewer than $4n-2$ moves.

While MIS algorithms can only stabilize with an independent set, MDS algorithms are capable of being stable with any minimal dominating set. In [36], the authors mentioned that the significance of MDS algorithms is that if the system is initialized to any minimal dominating set with the correct variable settings (including minimal dominating sets that are not independent), then it will remain stable. They also pointed out that an MDS algorithm is usually more complex than an MIS algorithm but can potentially produce *any* minimal dominating set. Following this idea, we give four different levels of stableness of self-stabilizing algorithms. We also prove that if an MDS-silent algorithm is preferred, then distance-1 knowledge is insufficient, where a self-stabilizing MDS algorithm is MDS-silent if it will not make any move when the starting configuration of the system is already an MDS. We conjecture that if we relax the MDS-silent property to MDS-stable (the execution of non-membership moves is allowed), then there will not exist an MDS-stable algorithm in the normal model.

We also consider developing self-stabilizing MDS-silent algorithms. In [21] and [76], the authors considered the distance-2 model, in which every node can read the states of nodes up to distance 2; see also [31] for the distance-k model. In particular, [21] proposed an $n(k+1)$-move self-stabilizing minimal $\{k\}$-dominating set algorithm; when $k = 1$, the algorithm finds an MDS using at most $2n$ moves. The paper [76] presented a $2n$-move self-stabilizing minimal k-dominating set algorithm; when $k = 1$, this algorithm also finds an MDS. However, the algorithms in both [21] and [76] operate correctly only with a central daemon. In this thesis, we present an algorithm, which is also $2n$-move but under an unfair distributed daemon and hence is more practical. It is easy to generalize our MDS-silent algorithm to self-stabilizing minimal $\{k\}$-dominating or k-dominating set algorithms under an unfair distributed daemon for $k \geq 2$.

Notice that the transformed version of algorithm $\mathcal{A}_1$ of [76] is not MDS-stable, even the algorithm is transformed by the transformer $\mathcal{C}$ (here we set $k = 1$). For an counterexample, consider a graph with five nodes v_1, v_2, v_3, v_4, and v_5 lying on a path in order. Let the states of $v_1, v_2, \ldots, v_5$ be `OUT-IN-IN-OUT-IN`. If $v_i.INcount$ are correct for all i, then only rule R2 of v_3 is enabled. By the transformer $\mathcal{C}$, all nodes are consistent and v_3 makes a request. After that v_2 and v_4 approve it. Suppose then v_5 fails (no longer alive) just before v_3 executes the leaving rule R2. At the mean time, v_4 wants to make an update urgently, but unfortunately the central daemon of the distance-1 model chooses v_3 to make a move. Since the transformed algorithm $\mathcal{A}_1^{\mathcal{C}}$ makes a membership move, $\mathcal{A}_1^{\mathcal{C}}$ does not satisfy the MDS-stable property. However, the expression model assumes distance-2 knowledge and therefore v_3 can see the failure of v_5 and v_3 will not leave the MDS; thus the transformer may not derive the same result as the one in the distance-2 model.

Chapter 6 has provided the first study of the problem of finding the values of signed star domination number of Cayley digraphs and Cayley graphs. We define the directed version of an SSDF on a digraph D and give the value of signed star domination number of D. We also obtain exact values for the signed star domination number for certain classes of Cayley graphs. Using these results on Cayley graph, we deduce the signed star domination number of C_n, K_n, and $K_{n,n}$. We also generalize the notion of k-factorable graphs to $\{2,1\}$-factorable graphs, in which we address the signed star domination number. However, there remains some Cayley graphs whose signed star domination number are unknown. Another future work is to address the signed star domatic number for Cayley graphs.

References

[1] Khaled M. Alzoubi, Peng-Jun Wan, and Ophir Frieder. Maximal independent set, weakly-connected dominating set, and induced spanners in wireless ad hoc networks. *International Journal of Foundations of Computer Science*, 14(2):287–303, 2003.

[2] Gheorghe Antonoiu and Pradip K. Srimani. Distributed self-stabilizing algorithm for minimum spanning tree construction. In *European Conference on Parallel Processing*, pages 480–487. Springer-Verlag, 1997.

[3] M. Atapour, S. M. Sheikholeslami, A.N. Ghameshlou, and L. Volkmann. Signed star domatic number of a graph. *Discrete Applied Mathematics*, 158(3):213–218, 2010.

[4] Yacine Belhoul, Said Yahiaoui, and Hamamache Kheddouci. Efficient self-stabilizing algorithms for minimal total k-dominating sets in graphs. *Information Processing Letters*, 114(7):339–343, June 2014.

[5] Steven C. Bruell, Sukumar Ghosh, Mehmet Hakan Karaata, and Sriram V. Pemmaraju. Self-stabilizing algorithms for finding centers and medians of trees. *SIAM Journal on Computing*, 29(2):600–614, October 1999.

[6] Franck Butelle, Christian Lavault, and Marc Bui. A uniform self-stabilizing minimum diameter spanning tree algorithm. In Jean-Michel Hélary and Michel Raynal, editors,

Distributed Algorithms, volume 972 of *Lecture Notes in Computer Science*, pages 257–272. Springer Berlin Heidelberg, 1995.

[7] Sébastien Cantarell, AjoyK. Datta, and Franck Petit. Self-stabilizing atomicity refinement allowing neighborhood concurrency. In *Self-Stabilizing Systems*, volume 2704 of *Lecture Notes in Computer Science*, pages 102–112. Springer Berlin Heidelberg, 2003.

[8] Subhendu Chattopadhyay, Lisa Higham, and Karen Seyffarth. Dynamic and self-stabilizing distributed matching. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, PODC'02, pages 290–297, New York, NY, USA, 2002. ACM.

[9] Thirugnanam Tamizh Chelvam, G. Kalaimurugan, and Well Y. Chou. The sighed star domination number of Cayley graphs. *Discrete Mathematics, Algorithms and Applications*, 04(02):1250017, 2012.

[10] Nian-Shing Chen, Hwey-Pyng Yu, and Shing-Tsaan Huang. A self-stabilizing algorithm for constructing spanning trees. *Information Processing Letters*, 39(3):147–151, 1991.

[11] Well Y. Chiu and Chiuyuan Chen. Linear-time self-stabilizing algorithms for minimal domination in graphs. In *Combinatorial Algorithms*, volume 8288 of *Lecture Notes in Computer Science*, pages 115–126. Springer Berlin Heidelberg, 2013.

[12] Well Y. Chiu, Chiuyuan Chen, and Shih-Yu Tsai. A $4n$-move self-stabilizing algorithm for the minimal dominating set problem using an unfair distributed daemon. *Information Processing Letters*, 114(10):515–518, 2014.

[13] Ajoy K. Datta, Lawrence L. Larmore, and Priyanka Vemula. A self-stabilizing $O(k)$-time k-clustering algorithm. *The Computer Journal*, 53(3):342–350, March 2010.

[14] Lyes Dekar and Hamamache Kheddouci. Distance-2 self-stabilizing algorithm for a b-coloring of graphs. In Sandeep Kulkarni and André Schiper, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 5340 of *Lecture Notes in Computer Science*, pages 19–31. Springer Berlin / Heidelberg, 2008.

[15] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

[16] Shlomi Dolev. *Self-stabilization.* MIT Press, 2000.

[17] Shlomi Dolev and Ted Herman. Superstabilizing protocols for dynamic distributed systems. In *Proceedings of the 2nd Workshop on Self-Stabilizing Systems*, WSS'95, pages 3.1–3.15, 1995.

[18] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7(1):3–16, November 1993.

[19] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Resource bounds for self-stabilizing message-driven protocols. *SIAM Journal on Computing*, 26(1):273–290, February 1997.

[20] Shlomi Dolev and Nir Tzachar. Empire of colonies: Self-stabilizing and self-organizing distributed algorithm. *Theoretical Computer Science*, 410(6–7):514–532, February 2009.

[21] Martin Gairing, Wayne Goddard, Stephen T. Hedetniemi, Petter Kristiansen, and Alice A. McRae. Distance-two information in self-stabilizing algorithms. *Parallel Processing Letters*, 14(3–4):387–398, 2004.

[22] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1979.

[23] Felix C. Gartner. A survey of self-stabilizing spanning-tree construction algorithms. Technical report, Swiss Federal Institute of Technology Tech. Rep. IC/2003/38, School of Computer and Communication Sciences, Lausanne, Switzerland, 2003.

[24] Cyril Gavoille, Ralf Klasing, Adrian Kosowski, Lukasz Kuszner, and Alfredo Navarra. On the complexity of distributed graph coloring with local minimality constraints. *Networks*, 54(1):12–19, August 2009.

[25] Sukumar Ghosh, Arobinda Gupta, Ted Herman, and Sriram V. Pemmaraju. Fault-containing self-stabilizing algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, PODC'96, pages 45–54, New York, NY, USA, 1996. ACM.

[26] Sukumar Ghosh and Mehmet Hakan Karaata. A self-stabilizing algorithm for coloring planar graphs. *Distributed Computing*, 7(1):55–59, November 1993.

[27] Wayne Goddard, Stephen T. Hedetniemi, David P. Jacobs, and Pradip K. Srimani. A robust distributed generalized matching protocol that stabilizes in linear time. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCSW'03, pages 461–465, Washington, DC, USA, 2003. IEEE Computer Society.

[28] Wayne Goddard, Stephen T. Hedetniemi, David P. Jacobs, and Pradip K. Srimani. A self-stabilizing distributed algorithm for minimal total domination in an arbitrary system graph. In *Proceedings of the 8th International Symposium on Parallel and Distributed Processing*, IPDPS'03, pages 240–243, Washington, DC, USA, 2003. IEEE Computer Society.

[29] Wayne Goddard, Stephen T. Hedetniemi, David P. Jacobs, and Pradip K. Srimani. Self-stabilizing global optimization algorithms for large network graphs. *International Journal of Distributed Sensor Networks*, 1(3–4):329–344, 2010.

[30] Wayne Goddard, Stephen T. Hedetniemi, David P. Jacobs, Pradip K. Srimani, and Zhenyu Xu. Self-stabilizing graph protocols. *Parallel Processing Letters*, 18(1):189–199, 2008.

[31] Wayne Goddard, Stephen T. Hedetniemi, David P. Jacobs, and Vilmar Trevisan. Distance-k knowledge in self-stabilizing algorithms. *Theoretical Computer Science*, 399(1–2):118–127, 2008.

[32] Wayne Goddard, Stephen T. Hedetniemi David P. Jacobs, and Pradip K Srimani. Self-stabilizing protocols for maximal matching and maximal independent sets for ad hoc networks. In *Proceedings of the 5th IPDPS Workshop on Advances in Parallel and Distributed Computational Models*, WAPDCM'03, pages 162–167. IEEE Computer Society, 2003.

[33] Wayne Goddard and Pradip K. Srimani. Anonymous self-stabilizing distributed algorithms for connected dominating set in a network graph. In *Proceedings of the 1st International Multi-Conference on Complexity, Informatics, and Cybernetics*, IMCIC'10, 2010.

[34] Nabil Guellati and Hamamache Kheddouci. A survey on self-stabilizing algorithms for independent, domination, coloring, and matching in graphs. *Journal of Parallel and Distributed Computing*, 70(4):406–415, 2010.

[35] Teresa W. Haynes, Stephen T. Hedetniemi, and Peter J. Slater. *Fundamentals of Domination in Graphs*. Monographs and Textbooks in Pure and Applied Mathematics. CRC Press, 1998.

[36] Sandra M. Hedetniemi, Stephen T. Hedetniemi, David P. Jacobs, and Pradip K. Srimani. Self-stabilizing algorithms for minimal dominating sets and maximal independent sets. *Computer Mathematics and Applications*, 46(5–6):805–811, 2003.

[37] Stephen T. Hedetniemi, David P. Jacobs, and Pradip K. Srimani. Maximal matching stabilizes in time $O(m)$. *Information Processing Letters*, 80(5):221–223, December 2001.

[38] Stephen T. Hedetniemi, David P. Jacobs, and Pradip K. Srimani. Linear time self-stabilizing colorings. *Information Processing Letters*, 87(5):251–255, September 2003.

[39] Lisa Higham and Zhiying Liang. Self-stabilizing minimum spanning tree construction on message-passing networks. In *Distributed Computing*, volume 2180 of *Lecture Notes in Computer Science*. Springer-Verlag, October 2001.

[40] Su-Chu Hsu and Shing-Tsaan Huang. A self-stabilizing algorithm for maximal matching. *Information Processing Letters*, 43(2):77–81, August 1992.

[41] Shing-Tsaan Huang, Su-Shen Hung, and Chi-Hung Tzeng. Self-stabilizing coloration in anonymous planar networks. *Information Processing Letters*, 95(1):307–312, July 2005.

[42] Shing-Tsaan Huang and Chi-Hung Tzeng. Distributed edge coloration for bipartite networks. In *Proceedings of the 8th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'06, pages 363–377, Berlin, Heidelberg, 2006. Springer-Verlag.

[43] Tetz C. Huang, Chih-Yuan Chen, and Cheng-Pin Wang. A linear-time self-stabilizing algorithm for the minimal 2-dominating set problem in general networks. *Journal of Information Science and Engineering*, 24(1):175–187, 2008.

[44] Tetz C. Huang, Ji-Cherng Lin, Chih-Yuan Chen, and Cheng-Pin Wang. A self-stabilizing algorithm for finding a minimal 2-dominating set assuming the distributed demon model. *Computers and Mathematics with Applications*, 54(3):350–356, August 2007.

[45] Michiyo Ikeda, Sayaka Kamei, and Hirotsugu Kakugawa. A space-optimal self-stabilizing algorithm for the maximal independent set problem. In *Proceedings of the 3rd International Conference on Parallel and Distributed Computing, Applications and Technologies*, PDCAT'02, pages 70–74, 2002.

[46] Alon Itai and Michael Rodeh. Symmetry breaking in distributed networks. *Information and Computation*, 88:150–158, 1981.

[47] S. Jahanbekam. A comment to: Two classes of edge domination in graphs. *Discrete Applied Mathematics*, 157(2):400–401, 2009.

[48] Ankur Jain and Arobinda Gupta. A distributed self-stabilizing algorithm for finding a connected dominating set in a graph. In *Proceedings of the 6th International Conference on Parallel and Distributed Computing Applications and Technologies*, PDCAT'05, pages 615–619, Washington, DC, USA, 2005. IEEE Computer Society.

[49] Hirotsugu Kakugawa and Toshimitsu Masuzawa. A self-stabilizing minimal dominating set algorithm with safe convergence. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 263–263, Washington, DC, USA, 2006. IEEE Computer Society.

[50] Sayaka Kamei and Hirotsugu Kakugawa. A self-stabilizing algorithm for the distributed minimal k-redundant dominating set problem in tree networks. In *Proceedings of the 4th International Conference on Parallel and Distributed Computing, Applications and Technologies*, PDCAT'03, pages 720–724, Washington, DC, USA, 2003. IEEE.

[51] Sayaka Kamei and Hirotsugu Kakugawa. A self-stabilizing approximation algorithm for the distributed minimum k-domination. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E88-A(5):1109–1116, May 2005.

[52] Mehmet Hakan Karaata. Self-stabilizing strong fairness under weak fairness. *IEEE Transactions on Parallel and Distributed Systems*, 12(4):337–345, April 2001.

[53] Adrian Kosowski and Lukasz Kuszner. Self-stabilizing algorithms for graph coloring with improved performance guarantees. In *Proceedings of the 8th International Conference on Artificial Intelligence and Soft Computing*, ICAISC'06, pages 1150–1159, Berlin, Heidelberg, 2006. Springer-Verlag.

[54] Fabian Kuhn, Thomas Moscibroda, and Rogert Wattenhofer. What cannot be computed locally! In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing*, PODC'04, pages 300–309, New York, NY, USA, 2004. ACM.

[55] Ji-Cherng Lin and Tetz C. Huang. An efficient fault-containing self-stabilizing algorithm for finding a maximal independent set. *IEEE Transactions on Parallel and Distributed Systems*, 14(8):742–754, August 2003.

[56] Nathan Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, February 1992.

[57] Fredrik Manne and Morten Mjelde. A self-stabilizing weighted matching algorithm. In *Proceedings of the 9th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'07, pages 383–393, Berlin, Heidelberg, 2007. Springer-Verlag.

[58] Fredrik Manne, Morten Mjelde, Laurence Pilard, and Sébastien Tixeuil. A new self-stabilizing maximal matching algorithm. In *Proceedings of the 14th International Conference on Structural Information and Communication Complexity*, SIROCCO'07, pages 96–108, Berlin, Heidelberg, 2007. Springer-Verlag.

[59] Toshimitsu Masuzawa and Sébastien Tixeuil. A self-stabilizing link-coloring protocol resilient to unbounded Byzantine faults in arbitrary networks. In JamesH. Anderson,

Giuseppe Prencipe, and Roger Wattenhofer, editors, *Principles of Distributed Systems*, volume 3974 of *Lecture Notes in Computer Science*, pages 118–129. Springer Berlin Heidelberg, 2006.

[60] Brendan D. McKay and Cheryl E. Praeger. Vertex-transitive graphs which are not cayley graphs, I. *Journal of the Australian Mathematical Society (Series A)*, 56:53–63, February 1994.

[61] A. Meir and J. W. Moon. Relations between packing and covering numbers of a tree. *Pacific Journal of Mathematics*, 61(1):225–233, 1975.

[62] Mikhail Nesterenko and Anish Arora. Stabilization-preserving atomicity refinement. *Journal of Parallel and Distributed Computing*, 62(5):766–791, May 2002.

[63] Reuay-Ching Pan, Jone-Zen Wang, and Louis R. Chow. A self-stabilizing distributed spanning tree construction algorithm with a distributed demon. *Tamsui Oxford Journal of Mathematical Sciences*, 15:23–32, May 1999.

[64] R. Saei and S.M. Sheikholeslami. Signed star k-subdomination numbers in graphs. *Discrete Applied Mathematics*, 156(15):3066–3070, 2008.

[65] Marco Schneider. Self-stabilization. *ACM Computing Surveys*, 25(1):45–67, 1993.

[66] Zhengnan Shi, Wayne Goddard, and Stephen T. Hedetniemi. An anonymous self-stabilizing algorithm for 1-maximal independent set in trees. *Information Processing Letters*, 91(2):77–83, 2004.

[67] Sandeep K. Shukla, Daniel J. Rosenkrantz, and S. S. Ravi. Observations on self-stabilizing graph algorithms for anonymous networks. In *Proceedings of the 2nd Workshop on Self-Stabilizing Systems*, WSS'95, pages 7.1–7.15, 1995.

[68] Sandeep K. Shukla, Daniel J. Rosenkrantz, and S. S. Ravi. Simulation and validation tool for self-stabilizing protocols. In *Proceedings of 2nd SPIN Workshop, Volume 32 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 1997.

[69] Pradip K. Srimani and Zhenyu Xu. Self-stabilizing algorithms of constructing spanning tree and weakly connected minimal dominating set. In *Proceedings of the 27th International Conference on Distributed Computing Systems Workshops*, ICDCSW'07, pages 3–11, Washington, DC, USA, 2007. IEEE Computer Society.

[70] Huang Sun, Brice Effantin, and Hamamache Kheddouci. A self-stabilizing algorithm for the minimum color sum of a graph. In *Proceedings of the 9th International Conference on Distributed Computing and Networking*, ICDCN'08, pages 209–214, Berlin, Heidelberg, 2008. Springer-Verlag.

[71] Sumit Sur and Pradip K Srimani. A self-stabilizing distributed algorithm to construct bfs spanning trees of a symmetric graph. *Computers and Mathematics with Applications*, 30, 1992.

[72] Sumit Sur and Pradip K. Srimani. A self-stabilizing algorithm for coloring bipartite graphs. *Information Sciences*, 69(3):219–227, April 1993.

[73] Ming-Shin Tsai and Shing-Tsaan Huang. A self-stabilizing algorithm for the shortest paths problem with a fully distributed demon. *Parallel Processing Letters*, 4(1–2):65–72, June 1994.

[74] Shihyu Tsai. An efficient self-stabilizing algorithm for the minimal dominating set problem under a distributed scheduler. Master's thesis, Department of Applied Mathematics, National Chiao Tung University, Hsinchu, Taiwan, 2011.

[75] Volker Turau. Linear self-stabilizing algorithms for the independent and dominating set problems using an unfair distributed scheduler. *Information Processing Letters*, 103(3):88–93, 2007.

[76] Volker Turau. Efficient transformation of distance-2 self-stabilizing algorithms. *Journal of Parallel Distribted Computing*, 72(4):603–612, 2012.

[77] Chi-Hung Tzeng, Jehn-Ruey Jiang, and Shing-Tsaan Huang. A self-stabilizing $(\delta + 4)$-edge-coloring algorithm for planar graphs in anonymous uniform systems. *Information Processing Letters*, 101(4):168–173, February 2007.

[78] Changping Wang. The signed star domination numbers of the cartesian product graphs. *Discrete Applied Mathematics*, 155(11):1497–1505, 2007.

[79] Douglas B. West. *Introduction to Graph Theory.* Prentice-Hall, Inc., 2nd edition, 2000.

[80] Baogen Xu. On signed edge domination numbers of graphs. *Discrete Mathematics*, 239(1–3):179–189, 2001.

[81] Baogen Xu. On edge domination numbers of graphs. *Discrete Mathematics*, 294(3):311–316, 2005.

[82] Baogen Xu. Two classes of edge domination in graphs. *Discrete Applied Mathematics*, 154(10):1541–1546, 2006.

[83] Zhenyu Xu, Stephen T. Hedetniemi, Wayne Goddard, and Pradip K. Srimani. A synchronous self-stabilizing minimal domination protocol in an arbitrary network graph. In *Proceedings of the 5th International Workshop on Distributed Computing, Springer LNCS 2918*, IWDC'03, pages 26–32, 2003.