Self-stabilizing algorithms for graph parameters

Brahim Neggazi



Numéro d'ordre: 41-2015 Année 2015

Université Claude Bernard Lyon 1

Laboratoire d'InfoRmatique en Image et Systèmes d'information

École Doctorale Informatique et Mathématiques de Lyon

Thèse de l'Université de Lyon

Présentée en vue d'obtenir le grade de Docteur,
spécialité Informatique

par

Brahim NEGGAZI

Self-stabilizing algorithms for graph parameters

Thèse soutenue le 15/04/2015 devant le jury composé de:

Rapporteurs:	Colette Johnen	–	Professeur à l'Université de Bordeaux 1
	Achour Mostefaoui	–	Professeur à l'Université de Nantes
Examinateurs:	Michel Habib	–	Professeur à l'Université Paris Diderot Paris 7
	Jean-luc Baril	–	Professeur à l'Université de Bourgogne
	Volker Turau	–	Professeur à l'Université de Hambourg
Directeur:	Hamamache Kheddouci	–	Professeur à l'Université de Lyon 1
Co-directeur:	Mohammed Haddad	–	Maître de Conférences à l'Université de Lyon 1

Acknowledgments

This dissertation is dedicated to the memory of my father Mâamar.

The completion of this thesis would not have been possible without the support and encouragement of many special people. Hence, I would like to take this opportunity to show my gratitude to those who have assisted me in myriad ways.

I would like to express my deepest gratitude to my advisor, Pr. Hamamache Kheddouci, for his excellent guidance, caring, patience, and for providing me with an excellent atmosphere for doing research. He was very patient with me, especially during the last months of my thesis that were very difficult for me. Thank you again Chief.

I would like to thank my second advisor, Dr. Haddad Mohammed, who let me experience and practical issues beyond the textbooks, patiently corrected my writing papers and dissertation. He was like a friend to me during the last four years. Thank you again my friend.

I would also like to thank Pr. Volker Turau for his collaboration and for guiding my research for the past two years and helping me to develop my background in writing, proving and way of presentation. Also I thank him for his fruitful discussions especially during my stay in Institute of Telematics of Hambourg University.

Special thanks goes to Pr. Colette Johnen, Pr. Achour Mostefaoui, Pr. Michel Habib, Pr. Volker Turau and Pr. Jean-Luc Baril who were willing to participate in my final defense committee.

Out of sight but close to my heart. I would like to thank Dr. Omar Kermia, who as a good friend, was always willing to help and give suggestions.

I am also deeply thankful to my colleague and my friend Fairouz for her kindness and for helping me enormously during my thesis through very fruitful discussions and continuous moral support. Thanks again Fairouz.

Many thanks to my friends Chellal, Sidali, Amer, Adel, Nabil, Kamel, Bacha, Reda, and all other members of the LIRIS laboratory Said, Isabelle, Saida, Hélène, Jean-pierre...

I would also like to thank my mother, two elder sisters, and brothers, especially the youngest one Hamid. They were always supporting me and encouraging me with their best wishes and prayers.

Finally, I would like to thank my best friend Slimane Lemmouchi. He was always there cheering me up and he is the one who stood by me in good and bad moments. Thank you again Boss.

Brahim

Abstract: The concept of self-stabilization was first introduced by Dijkstra in 1973. A distributed system is self-stabilizing if it can start from any possible configuration and converges to a desired configuration in finite time by itself without using any external intervention. Convergence is also guaranteed when the system is affected by transient faults. This makes self-stabilization an effective approach for non-masking fault-tolerance.

The self-stabilization was studied in various fields in distributed systems such as the problems of clock synchronization, communication and routing protocols. Given the importance of graph parameters, especially for organization and communication of networks and distributed systems, several self-stabilizing algorithms for classic graph parameters have been developed in this direction, such as self-stabilizing algorithms for finding minimal dominating sets, coloring, maximal matching, spanning tree and so on.

Thence, we propose in this thesis, distributed and self-stabilizing algorithms to some well-known graphs problems, particularly for graph decompositions and dominating sets problems that have not yet been addressed in a view of self-stabilization.

The four major problems considered in this thesis are: the *partitioning into triangles*, *p-star decomposition*, *edge monitoring set* and *independent strong dominating set* problems. The common point between these four problems is that they are considered as variants of dominating set and matching problems and all propositions deal with the self-stabilization paradigm.

The *partitioning into triangles* describes the graph as the union of disjoint triangles. It has been proved that this problem is NP-complete and even finding the maximum number of triangles in arbitrary graph is NP-hard. Hence we consider the local maximal partitioning called *maximal graph partitioning into triangles*. Then, we present different self-stabilizing algorithms for this problem under two types of schedulers and we give formal proofs for correctness, convergence and complexities.

A p-star is a tree with one center node and p leaves where $p \geq 1$. The *p-star decomposition* subdivides the graph into disjoint components where each one contains a p-star as a subgraph. We propose self-stabilizing algorithms for decomposing a graph into p-stars. Formal proofs for the correctness and the convergence of these algorithms are given within the unfair distributed scheduler.

In 2008, a new parameter of edge domination was introduced by Dong *et al.*, called *Edge monitoring problem*. A node v can monitor (*i.e.* dominate) an edge e if the end nodes of e are neighbors of v, *i.e.* they form a triangle. Moreover, some edges need more than one monitor. Thus, the problem of edge-monitoring consists in identifying a set of nodes that monitor some edges. Furthermore, the minimum set edge-monitoring problem is long known to be NP-complete. In this thesis, we develop a new polynomial distributed and self-stabilizing algorithm for computing a minimal set for edge-monitoring problem within the distributed scheduler.

The last studied parameter, called *Independent Strong Dominating Set* (ISD-set), is an interesting variant of dominating sets. In addition to its domination and independence properties, the ISD-set considers also nodes degrees that make it very useful in practical applications. Thence, we proposed a self-stabilizing algorithm for computing an ISD-set of an arbitrary graph. Formal proofs for the correctness, convergence and complexity of this algorithm are given within the distributed scheduler. Moreover, some simulations with well- known self-stabilizing algorithms are provided.

Keywords: Self-stabilizing algorithms, partitioning into triangles, p-star decomposition, distributed system, edge monitoring problem, strong dominating set, generalized matching, fault-tolerance.

Résumé: Le concept d'auto-stabilisation a été introduit par Dijkstra en 1973. Un système distribué est auto-stabilisant s'il peut démarrer de n'importe qu'elle configuration initiale et retrouver une configuration légitime en un temps fini par lui-même et sans aucune intervention extérieure. La convergence est également garantie lorsque le système est affecté par des fautes transitoires, ce qui en fait une approche élégante, non masquante, pour la tolérance aux pannes.

L'auto-stabilisation a été étudiée dans divers domaines des systèmes distribués tels que les problèmes de synchronisation de l'horloge, de la communication et les protocoles de routage. Vu l'importance des paramètres de graphes notamment pour l'organisation et l'optimisation des communications dans les réseaux et les systèmes distribués, plusieurs algorithmes auto-stabilisa-nts pour des paramètres de graphe ont été proposés dans la littérature, tels que les algorithmes auto-stabilisants permettant de trouver les ensembles dominants minimaux, coloration des graphes, couplage maximal et arbres de recouvrement.

Dans cette perspective, nous proposons, dans cette thèse, des algorithmes distribués et auto-stabilisants pour certains problèmes de graphes bien connus, en particulier pour les décompositions de graphes et les ensembles dominants qui n'ont pas encore été abordés avec le concept de l'auto-stabilisation. Les quatre problèmes majeurs considérés dans cette thèse sont: *partitionnement en triangles*, *décomposition en p-étoiles*, *Monitoring des arêtes*, *fort ensemble dominant et independant*.

Ainsi, le point commun entre ces problèmes, est qu'ils sont tous considérés comme des variantes des problèmes de domination et de couplage dans les graphes et leur traitement se fait d'une manière auto-stabilisante.

Le *partitionnement en triangles* décrit un graphe comme étant l'union de triangles disjoints. Il a été prouvé que ce problème est NP-complet ainsi que trouver le nombre maximum de triangles disjoints dans un graphe arbitraire est NP-difficile. A cet effet, nous considérons une variante locale de ce partitionnement qui est appelé partitionnement maximale en triangles. Ensuite, nous présentons des algorithmes auto-stabilisants à ce problème sous deux types d'ordonnanceurs. Aussi, nous présentons des preuves formelles pour la correction, la convergence et la complexité de ces algorithmes.

Une p-étoile est un arbre avec un noeud central et p feuilles ($p \geqslant 1$). La *décomposition en p-étoiles* divise le graphe en plusieurs composantes disjointes où chacune d'elles contient une p-étoile. Pour cela, nous avons proposé deux algorithmes auto-stabilisants pour la décomposition d'un graphe arbitraire en p-étoiles. Des preuves formelles pour la correction, la convergence et les complexités ont été présentées sous un ordonnanceur distribué.

En 2008, un nouveau paramètre de domination d'arêtes a été introduit par Dong *et al.*, appelé *Monitoring des arêtes*. Un noeud v peut monitorer (dominer) une arête e si les deux noeuds d'extrémité de e sont voisins à v, c'est-à-dire que les trois noeuds forment un triangle. De plus, certaines arêtes ont besoin de plus d'un moniteur. Ainsi, le problème de monitoring consiste à l'identification des noeuds (appelés moniteurs) qui vont constituer l'ensemble de monitoring des arêtes. Il a été démontré que trouver un ensemble minimum pour ce problème est NP-difficile. Dans cette thèse, nous développons un nouvel algorithme polynomial distribué et auto-stabilisant pour calculer l'ensemble minimal à ce problème tout en considérant un ordonnanceur distribué.

Le dernier paramètre étudié, appelé *ensemble dominant independant fort* (ISD-set), est une variante intéressante de l'ensemble dominant dans les graphes. En plus de ses propriétés de domination et d'indépendance, ISD-set considère également les degrés des noeuds rendant cette variante très utile dans des applications pratiques. A cet effet, nous avons proposé un algorithme auto-stabilisant pour le calcul d'un ensemble minimal ISD dans un graphe arbitraire. Des preuves formelles pour la correction, la convergence et la complexité de cet algorithme sont présentées dans cette thèse. De plus, des simulations de comparaison de notre proposition avec d'autres algorithmes bien connus sont fournies.

Contents

Chapter 1

Introduction

Many computer systems are composed of multiple processors connected by communication links. The development of computer networks has facilitated the interconnection of computers, and at the same time, the development of new systems, called *distributed systems*. These systems allow us to perform the computing with very high performance through the collaboration of different machines. Resource sharing and communication are two goals of distributed systems. Every day, these systems grow more and more, and now, we are using systems that run in large scale either social networks, wireless sensors networks or ad-hoc networks.

Larger system implies greater risk of failures, and the management and the monitoring of these systems are more complicated. Furthermore, repair all dysfunctions are difficult. So, fault tolerance and robustness are then absolute necessity for the survival of these systems. For this purpose, there are two techniques to deal with faults in distributed systems: *masking* solutions hide the occurrence of faults to the observer of the system, however, these solutions are often costly in time and resources (computing power, memory) because they have redundant level of critical components or information in order to contain the expected faults. Moreover, these pessimistic solutions (masking techniques) only tolerate faults that are already preset at a machine. However, the *non-masking* solutions are optimistic techniques that accept the unavailability of the system for a given time and deal with all transient faults.

Self-stabilization was introduced by Dijkstra in 1973. It is a non-masking technique for designing fault-tolerant distributed systems. A distributed system is self-stabilizing if it can automatically find the correct behavior after a failure of one or more elements in the system. Since, the return to normal behavior occurs without external intervention, the self-stabilization sands for a very interesting approach for reliable distributed systems technology. However, this concept did not gain any attention in the beginning until 1984, when Lamport referred to Dijkstra's work as an important approach for fault-tolerance. Lamport wrote in regard to this concept:

"I regard this as Dijkstra's most brilliant work — at least, his most brilliant published paper. It's almost completely unknown. I regard it to be a milestone in work on fault tolerance"

In 2000, Shlomi Dolev wrote a very nice book intituled *Self-stabilization*. The author presented the fundamentals of self-stabilization and showed the process of designing self-stabilizing distributed systems. The book proceeds from the basic concept of self-stabilizing algorithms to advanced applications. Right after, Dijkstra received a price for his major contribution in 2002. Since then, the self-stabilization becomes a very interesting field in different researches.

To exploit distributed systems, software solutions must be developed. These solutions must, of course, manage the communication between different machines, in addition, they may require a particular organization of system components. Since, it is natural to model a distributed system by a graph where nodes and edges represent the processes (resources) and their communication links and given the multiple benefits of self-stabilization and the different needs for the management and control distributed systems, several self-stabilizing algorithms for graph parameters were proposed to allow the structuring, the monitoring and the organization of these systems.

The four major problems considered in this thesis are the *Partitioning into Triangles*, the *p-Star Decomposition*, the *Edge Monitoring Set* and the *Independent Strong Dominating Set* problems. The main common point between these four problems is that they deal with self-stabilization paradigm.

The *Partitioning into Triangles* (PT) is one of the classical NP-complete problems and it is defined as follows. Given q such that $n = 3q$ where q is a positive integer and n is the number of nodes in the graph G, a partitioning into triangles consists of q disjoint sets $T_1, T_2, \ldots, T_q$ where each T_i forms a triangle in G. Since, deciding if a graph can be partitioned into triangles or not is NP-complete, and finding such partitioning in general graphs does not always exist, then we consider the following variant of graph partitioning called *Maximal Graph Partitioning into Triangles* (MPT). The MPT of a graph G is a set of disjoint subsets T_i of nodes such that each subset T_i forms a triangle and no triangle can be added to this set using only nodes not already contained in a set T_i. In addition to its theoretical aspects, this parameter has several practical aspects in distributed system for example, computing estimates of values measured in wireless sensors networks, scheduling and so on. For this reason, we study the problem of maximal partitioning into triangles in distributed systems using the self-stabilization paradigm. Moreover, two self-stabilizing algorithms for such partitioning are developed.

The *Maximal p-Star Decomposition* is one of the well studied graph decomposition problem, also called *star partitions* in graph theory. Given a positive integer p, a p-star is a complete bipartite graph $K_{1,p}$ with one center node and p leaves where $p \geq 1$. The p-star decomposition describes a graph as the union of disjoint stars which all stars have equal size. This variant of decomposition belongs to the class of generalized matchings and subgraph-decomposition problems that were proved

to be NP-complete. Thus, for a given arbitrary graph G, a p-star decomposition SD of G is called *maximal p-star decomposition* (MSD) if the subgraph induced by the nodes of G not contained in any p-star of SD does not contain a p-star as a subgraph. Note that a 1-star decomposition is equivalent to the classical matching in graphs where the aim is to find independent edges in a graph. The star decompositions have several applications in areas such as scientific computing, scheduling, load balancing and parallel computing, studying the robustness of social networks. In addition these applications in distributed systems, the decomposition into p-stars is also used in the field of parallel computing and programming. This decomposition offers similar paradigm as the *Master-Slaves* (a.k.a *Master-Workers*) paradigm used in grid networks, P2P infrastructures and Wireless Sensors Networks. From the foregoing, we study the problem of Maximal p-star Decomposition with the self-stabilizing property. Therefore, two self-stabilizing algorithms for such decomposition are proposed with different complexity measures.

Edge Monitoring Set problem (EMS) is an effective mechanism for security of wireless sensors networks. A node v can monitor an edge e if the end nodes of e are neighbors of v (*i.e.* v and the end nodes of e form a triangle in the graph). Moreover, some edges need more than one monitor. Furthermore, finding the minimum set of monitor nodes for such problem is proved that is NP-complete by *Dong et al.* in 2008. Moreover, they propose two distributed algorithms for such problem with provable approximation ratio in 2011. In this thesis, we develop a self-stabilizing algorithm for such problem that converges in polynomial times, improving the existing self-stabilizing algorithm for such problem.

Independent Strong Dominating Set (abv. ISD-set) is an interesting variant of the dominating sets (DS) and the independent sets (IS) parameters that are largely studied in graph theory due to their several applications especially for designing efficient protocols in wireless sensor and ad-hoc networks. In addition to its domination and independence properties, the ISD-set considers also nodes degrees that makes it very useful in practical applications. Thence, in this part we propose the first linear self-stabilizing algorithm for computing a minimal ISD-set of an arbitrary graph. Furthermore, some simulations and comparisons of our ISD-set algorithm with well-known self-stabilizing algorithms for DS and IS problems are provided.

This thesis is organized as follows: **Chapter 2** gives an introduction to the distributed algorithm and the self-stabilization paradigm as an approach for fault-tolerance. Moreover, this chapter presents the designing of self-stabilizing algorithms and the model's assumptions used. Then the rest of thesis is split into three main parts.

The first part is composed from three chapters. **Chapter 3** presents the maximal partitioning into triangles of an arbitrary graph et its relationship with maximal matching in graphs. Moreover, some applications of this partitioning are also pro-

vided. In the following **Chapters 4** and **5**, we present two self-stabilizing algorithms for such partitioning under the central scheduler and the distributed scheduler respectively.

The second part is also divided into three chapters. **Chapter 6** presents the Maximal p-star Decomposition problem and presents several applications of this parameter in distributed systems. **Chapters 7** and **8** present respectively two complementary self-stabilizing algorithms for such decomposition using different proof techniques.

In the third part, we introduce the last discussed two parameters in this thesis, Edge monitoring and Independent strong dominating sets problems. Then we present their motivation and applications in wireless networks in **Chapter 9**. Then, we present two self-stabilizing algorithms for these parameters in **Chapters 10 and 11** respectively. Finally, **Chapter 12** summarizes all results of this thesis and gives some remarks and directions for further research.

List of publications arising from this thesis

International conferences

1. B. Neggazi, M. Haddad, V. Turau and H. Kheddouci. A Self-stabilizing Algorithm for Edge Monitoring Problem. In the proceedings of Stabilization, Safety, and Security of Distributed Systems, Germany, 2014.
2. B. Neggazi, V. Turau, M. Haddad and H. Kheddouci. A Self-stabilizing Algorithm for Maximal p-Star Decomposition of General Graphs. In the proceedings of Stabilization, Safety, and Security of Distributed Systems, Japan, 2013.
3. B. Neggazi, M. Haddad and H. Kheddouci. Self-stabilizing algorithm for maximal graph decomposition into disjoint paths of fixed length. In the proceedings of Theoretical Aspects of Dynamic Distributed Systems, Italy, 2012.
4. B. Neggazi, M. Haddad and H. Kheddouci. Self-stabilizing Algorithm for Maximal Graph Partitioning into Triangles. In the proceedings of Stabilization, Safety, and Security of Distributed Systems, Canada, 2012.

Accepted paper with minor revisions

5. B. Neggazi, M. Haddad and H. Kheddouci. A new Self-Stabilizing Algorithm for Maximal p-Star Decomposition of General Graphs, submitted to Information Processing Letters.
6. B. Neggazi, N. Guellati, M. Haddad and H. Kheddouci. Efficient self-stabilizing algorithm for independent strong dominating sets in arbitrary graphs, submitted to International Journal of Foundations of Computer Science.

Submitted papers

7. B. Neggazi, M. Haddad, V. Turau and H. Kheddouci. A Self-stabilizing Algorithm for Edge Monitoring Problem, a SSS 2014 paper, submitted to a special issue in Elsevier's Information and Computation journal.
8. B. Neggazi, V. Turau, M. Haddad and H. Kheddouci. A $O(m)$ Self-Stabilizing Algorithm for Maximal Graph Partitioning into Triangles, submitted to Parallel Processing Letters.

Chapter 2

Self-Stabilization & Graph Problems

Contents

This Chapter is devoted to introduce the self-stabilizing algorithms for graph parameters. In the first section, the distributed algorithms and their communication's models are presented. Section 2.2 gives a classification of fault's types in distributed systems and the common approaches for fault-tolerance. Then, Section 2.3 introduces formal definitions of the concepts used for self-stabilizing algorithms and the design of such algorithms. Finally, Section 2.4 presents a brief survey of self-stabilizing algorithms proposed for some graph parameters.

2.1 Distributed algorithms

A *distributed system* is a collection of independent entities that cooperate to solve a problem that cannot be individually solved [KS08]. These entities (a.k.a. processors or resources) communicate between them using message passing or shared memory.

Usually, the concept of distributed system is used to describe communication networks and multi-processor computers. Each processor can only communicate with other adjacent processors, called neighbors. Since, it is more natural to model a distributed system by graph in which processors and communications are represented by nodes and edges respectively. This section uses the terms processors and nodes interchangeably, depending on the context.

There are two models assumed in distributed systems: The synchronous and the asynchronous models. The synchronous model assumes the existence of a *global clock pulse* (or simply a *pulse*) and all processors in the system communicate simultaneously at each pulse. In this model, the processors can detect the lost of messages if a processor does not receive messages within a certain time. Contrary to synchronous model, the asynchronous model have not a global clock for the system and therefore there is no upper bounds on the message delay or local computational for processes. As consequence, the lost messages may never be detected. In this thesis, these types of models (synchronous or asynchronous) are encapsulated under the assumption of the *daemon* and the *communication atomicity* used by the system. These two terms will be defined in details later.

A distributed algorithm is an algorithm that will run on each node in the distributed system. Distributed algorithms are used in several areas such as network communications, distributed information processing, distributed computing. Based on local knowledge only, the nodes can operate and communicate simultaneously with each other to resolve a common problem. Classical problems solved by distributed algorithms include spanning tree construction, leader election end mutual exclusion. This local knowledge on each node constitutes the main difficulty in distributed algorithms.

The communication atomicity between processes are modeled in various ways for distributed algorithms. However, the majority of self-stabilizing algorithms use a high level of atomicity abstraction. According to Dolev in [Dol00], the most common communication atomicities are:

1. *The shared memory model with composite atomicity* (a.k.a state model): In [Dij74], Dijkstra used this model to introduce the concept of self-stabilization. In this model, an atomic step (or a single move) by a node consists of reading states (registers) of all its neighbors, making internal computations and then updating its own state (register).

2. *The read-write atomicity model* (a.k.a shared-register model or Dolev model): In [DIM93], Dolev et al. introduced new type of computational model. This model assumes separate read/write atomicity, *i.e.* each atomic step consists of internal computations and either a single read operation or a single write operation [Dol00].

3. *The message-passing model* [AB93]: In this model, an atomic step consists of either sending or receiving a message but not both simultaneously.

In the state model and Dolev model, each two neighbors share a common memory, contrary to message-passing model where nodes exchange messages. In this thesis, we assume the shared memory model with composite atomicity which is the most common model used in distributed systems.

We have to note that several transformers have been proposed in the literature for converting algorithm under one model into another algorithm that runs under another models. These transformers are mentioned latter in this chapter.

Moreover, the distributed systems have two characterizations the *anonymity of processors* and the *uniformity of the algorithm*. For the first characterization, we distinguish two types:

- *Anonymous system*: all processors are identical and they are unable to distinguish it from other processors with the same degree in deterministic way. In this type of system, each processor identifies its communication links with local numbers, called *port numbers*.
- *Non-anonymous system*: In this type of system, all processors can be distinguished, for example by using of identifiers.

The second characterization consists on the respect of algorithm's uniformity in any processor. Thus, a distributed algorithm is *uniform* if and only if all processors execute the same algorithm. An algorithm is *non-uniform* if at least one processor does not execute the same algorithm. In this thesis, we assume that all nodes have (locally) unique identifiers and only uniform algorithms are developed.

2.2 Fault-tolerance approaches

In the previous section, we defined the distributed systems as a set of independent entities that cooperate between them for solving a given problem. However, the context assumed that these systems are not disturbed by one or more external or internal events to the system, a.k.a "faults". Otherwise, the distributed algorithms cannot solve the problem for which they were designed. For this reason, other approaches have been introduced, refining the context of distributed systems, to take into account the occurrence of faults in the system.

The following section presents a classification of different faults that can occur in a distributed system and the main fault-tolerance approaches are presented.

2.2.1 Faults taxonomy in distributed systems

In [Tix09], Tixeuil describes faults in distributed systems using two criteria: *time* and *nature*. Considering the time occurrence of faults, three types are distinguished:

a. *Transient faults*: faults that are arbitrary in nature but there is a time in the execution where these faults not occur again.

b. *Permanent faults*: faults that occur at any time and stay permanently.

c. *Intermittent faults*: faults that are arbitrary in nature and can occur at any time in the execution.

Note that the two first types (transient and permanent) are specific cases of the intermittent faults [Tix09].

The second criterion *nature* depends mainly on the *state* and the *code* of a node. *State-related faults* change the correct state of a node, *i.e.* its register communication or its variables are changed after being affected by one or more faults. Usually, this type of faults models memory corruption of a node due to environmental perturbations, attacks or a dysfunction of the physical memory of a node. *Code-related faults* affect the correct functioning of a node such that crashes, omissions and byzantine faults. More details on this topic can be found in [Tix09].

2.2.2 Classification of fault-tolerance algorithms

The distributed systems are exposed to different kind of faults and they occur at any time as it is mentioned in the previous section. However, it is essential to develop solutions to deal with these faults in order to keep a proper functioning of the system. These solutions are often classified according to the visibility of faults to an observer (user) of the system. A *masking* solution hides the occurrence of faults to the observer, while a *non-masking* solution does not have this property [Tix09] and it accepts the unavailability of the system for a given time. It seems that the masking solutions are more interesting than non-masking solutions, especially for sensitive applications. However, these solutions are often costly in time and resources (computing power, memory). Moreover, these solutions (masking) only tolerate faults that are already preset at a node.

In [Tix09], Tixeuil classifies fault-tolerant algorithms into two categories :

1. *Robust algorithms*: These algorithms are typically masking solutions. They have redundant level of critical components or information in order to contain the expected faults. Usually, these solutions assume that even with a bounded number of faults, the rest of the system still having a proper functioning. However, apart resource required for redundancy, robust algorithms require an exhaustive list of the expected faults.

2. *Self-stabilizing algorithms*: These algorithms are non-masking solutions and assume that all faults are transient (cf. Section 2.2). The self-stabilizing algorithms have no assumption on the nature of faults or extent have to be made. An algorithm is self-stabilizing if it can start from any possible configuration and converges to a desired configuration in finite time by itself without any external intervention. Being able to start from any configuration means that the algorithm does not need any initialization of its variables.

2.3 Self-stabilization

This section introduces in details the self-stabilization and gives more descriptions of the concepts and methods used in this thesis. Furthermore, several techniques for proving convergence of self-stabilizing algorithms are discussed.

2.3.1 Self-stabilization properties

A system is self-stabilizing if it can start from any possible configuration and converges to a desired configuration (legitimate configuration) in finite time by itself without using any external intervention. Convergence is also guaranteed when the system is affected by transient faults (cf. Section 2.2). This makes self-stabilization an elegant approach for transient fault-tolerance [Dol00]. Figure 2.1 illustrates the behavior of self-stabilization system. Note that self-stabilizing system may not reach a legitimate configuration (or desired configuration) if faults occur frequently during the convergence. For this reason, most publications assume that all faults are transient, *i.e.* no further faults occur during the stabilization of the system.

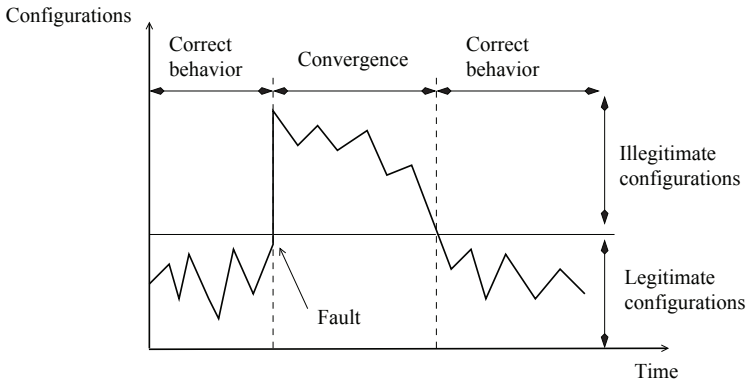


Figure 2.1: Self-stabilizing system's behavior.

The concept of self-stabilization was first introduced by Dijkstra in [Dij74]. This concept did not gain any attention in the beginning until 1984, when Lamport referred to Dijkstra's work as an important approach for fault-tolerance. Lamport said in his invited address [Lam84] in regard to [Dij74]:

"I regard this as Dijkstra's most brilliant work — at least, his most brilliant published paper. It's almost completely unknown. I regard it to be a milestone in work on fault tolerance"

Then:

"I regard self-stabilization to be a very important concept in fault tolerance, and to be a very fertile field for research"

After few years, Lamport's predictions have been realized and the self-stabilization becomes very interesting field in different researches, especially in network communications and graph protocol problems. Some graphs problems within self-

stabilization properties are presented in Section 2.4. Most of these works and other self-stabilizing algorithms for graph problems can be found in the survey of Guellati and Kheddouci [GK10].

Self-stabilizing algorithms can be *silent* or not (a.k.a. *non-silent*). A self-stabilizing algorithm is *silent* if and only if once the system reaches a legitimate configuration, all nodes of the system do not change their states (or register's values of any node remain fixed), until new faults occur. The majority of self-stabilizing algorithms for graph problems are silent, such that dominating set, matching and coloring. Otherwise, the algorithm is *non-silent* such that token circulation algorithm. In this thesis, all self-stabilizing algorithms are silent.

Arora and Gouda define two properties for self-stabilizing algorithms [AG93]: (See Figure 2.2)

- *Closure*: Once the system reaches a legitimate configuration, this property will be preserved, *i.e.* the set of legitimate configurations is closed.
- *Convergence*: The system always reaches a legitimate configuration after a finite time if no further fault occurs during the stabilization.

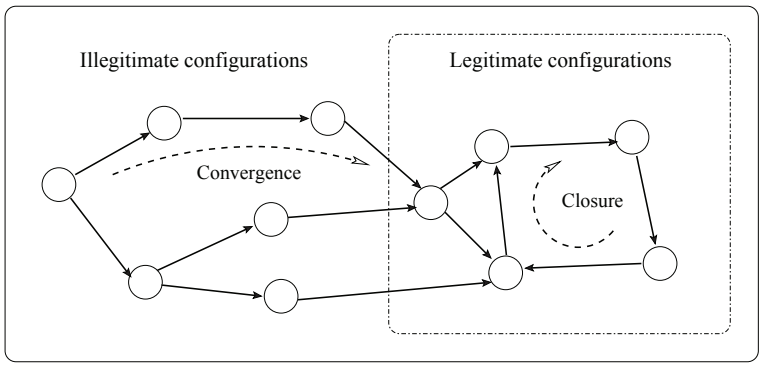


Figure 2.2: Self-stabilization's properties.

A third property for self-stabilizing algorithms, called *Correctness*, can also be found in the literature. Usually, the correctness is used for silent algorithms and it is defined as follows:

- *Correctness*: Every final configuration is legitimate, *i.e.* the algorithm actually computes for which it was originally developed.

The self-stabilizing algorithms presented in this thesis are all silent, as it is the case of most graph protocols. Since, the nodes do not make others moves when a legitimate configuration is reached, then the closure property does not have to be proven explicitly for the proposed self-stabilizing algorithms. More formal definitions of terms *move*, *legitimate* and *illegitimate configurations* are given in the following section.

In addition to fault-tolerance aspect, the self-stabilization presents many advantages:

- *Self-recovering*: The system always returns into a correct behavior without any external intervention or global initialization. Thus, self-stabilization is very useful for scale-free system in which manual intervention is impossible.

- *No initialization*: The system always converges even if it starts from illegitimate configuration. Thus, the self-stabilizing algorithms do not required any correct initialization.

- *Dynamic topology adaptation*: If an algorithm has a correct behavior which depends on the system topology, such as spanning tree construction, network decompositions and the algorithm may have an incorrect behavior when topology changes, then the self-stabilizing algorithm is suitable in this case. Since topology changes can be seen as transient faults that affect the correct behavior of the algorithm, then the self-stabilizing algorithm returns automatically to the correct topology in finite time.

In addition to these advantages, Berns and Ghosh show in [BG09] that self-stabilization is currently a fundamental property for "self-*" system's properties such as self-organization, self-healing, self-configuration.

However, there are of course some disadvantages of self-stabilization concept which cannot be ignored:

- *High complexity*: The performance of self-stabilizing algorithms are often lower than their equivalent non-self-stabilizing algorithms in case where no transient faults.

- *No termination detection*: The nodes of the system have no way of detecting the termination of the algorithm or aware if a legitimate configuration is reached or not.

2.3.2 Self-stabilizing algorithm design

The distributed system is represented by an undirected graph $G = (V, E)$, such that V is a set of nodes corresponding to the processes and E is a set of edges corresponding to the links. Let $n = |V|$ and $m = |E|$. Two nodes v and u are neighbors if and only if $(v, u) \in E$. The set of neighbors of a node v is denoted by $N(v)$, *i.e.* $N(v) = \{u \in V | (v, u) \in E\}$. The closed neighborhood of a node v is denoted by $N[v] = N(v) \cup \{v\}$. We denote by $d(v)$ the degree of a node v (*i.e.* $d(v) = |N(v)|$) and Δ the maximum node degree in the graph. The maximum length of the shortest path between any nodes is called diameter of G and it is denoted by D.

In the system, every node v has a set of variables whose contents specify the *state* "s_v" of the node v. The union of the states of all nodes defines the system's *global state* (or configuration).

Definition 1 (Configuration) *A configuration c of the graph G is defined as the n-tuple of all node's states: $c = (s_{v_1}, \ldots, s_{v_n})$. The set of all configuration is denoted by C_G.*

Each node has only a partial view of the system. Based on its state and that of its neighbors (distance-one model), a node can make *a move* which consists of changing the value of one or more of its variables. Note that in distance-two model (resp. distance-k model), a node can read its state and the state of nodes of distance at most two (resp. at most k). In this thesis, we use only distance-one model because it is more realistic. Therefore, self-stabilizing algorithms are given as a set of rules of the form :

$$[Rule's\ label] :: [\textbf{If}\ p(v)\ \textbf{then}\quad M]$$

The predicate $p(v)$ (a.k.a. guard) is defined over v's partial view. The statement M denotes a move that changes only state of the node v. A rule is called *enabled* if its predicate evaluates to true. A node v is also called *enabled* (or privileged) if at least one of its rules is enabled. Otherwise, the node v is *disabled*, *i.e.* all of its rules are disabled.

The nodes cooperate to solve a specific problem. This problem is defined by a predicate P. This motivates the formal definition of Legitimate configuration:

Definition 2 (Legitimate configuration) *A configuration c is called legitimate (or desired) with respect to P if c satisfies P. Let $L_P \subseteq C_G$ be the set of all legitimate configuration with respect to a predicate P.*

Let us consider the problem of matching in graphs. Predicate P is evaluated to true if any node in the graph G is matched (married) with only one neighbor. Then any configuration that satisfies P is called legitimate configuration. Otherwise, the configuration is illegitimate. Figure 2.3 illustrates a legitimate configuration for matching problem. More details on this problem can be found in Section 2.4.

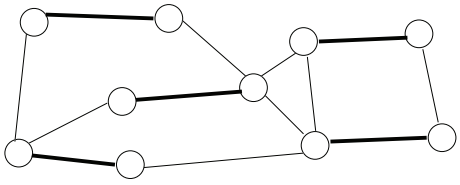


Figure 2.3: A legitimate configuration for matching problem. (The depicted edges form the matching)

Definition 3 (Execution) *An execution x of an algorithm is a maximal sequence of configurations $c_1, c_2, \ldots, c_i, \ldots, c_k$ such that each configuration c_{i+1} is the next configuration of c_i using one unit of time (a.k.a. step).*

The execution of self-stabilizing algorithms are encapsulated under the notion of daemon (a.k.a scheduler). An enabled node v makes a *move* if and only if v is selected by the daemon *i.e.* the node v brought into a new state that is a function of its old state and the states of its neighbors [Dij74]. Thus, several daemons have been proposed for designing self-stabilizing algorithms. The following section describes the most common daemons used in the literature.

2.3.3 Daemons

The execution of self-stabilizing algorithm is captured by an abstraction called daemon (a.k.a. Scheduler) [Dij74]. Intuitively, the daemon is a mechanism for selecting the enabled (privileged) nodes to execute their moves. This mechanism plays the role of both scheduler and adversary against the stabilization of the algorithm. This can be done by scheduling the worst possible cases for algorithm's execution. Thus, the choice of daemon is important in designing of self-stabilizing algorithm in terms of convergence and complexity analysis. Indeed, many types of daemons are assumed in the literature of self-stabilizing algorithms. Dubois presents a good taxonomy of existing daemons in [DT11]. The three most common daemons are the following:

1. *Central daemon* (a.k.a. *serial daemon*): At each step, the central daemon selects exactly one enabled node to make a move.

2. *Distributed daemon*: The distributed daemon selects in each step a non-empty subset of the enabled nodes to make their moves simultaneously.

3. *Synchronous daemon*: This type of daemon can be considered as a special kind of distributed daemon where in each step all enabled nodes make their move simultaneously.

Daemons are also associated with the notion of fairness. A daemon can be fair (weakly), or unfair (adversarial). A daemon is *fair* if every continuously enabled node is eventually selected. The *unfair* daemon on the other hand may delay the move of an enabled node as long as there are other enabled nodes. Self-stabilizing algorithms designed for a specific daemon may not operate under a different daemon. However, an algorithm designed for an unfair distributed daemon works with all other daemons. For this reason, we consider the unfair distributed daemon for all problems discussed in this thesis, except the preliminary algorithm presented in Chapter 4.

2.3.4 Complexity measures

The complexity measures are used to evaluate the performance of a self-stabilizing algorithm. These measures include time, memory or the number of messages sent. The latter is not used in this thesis because we use only shared memory model as communication model (*cf.* Section 2.1).

There are different measures for time complexity of self-stabilizing algorithms. These measures do not consider the local resource demand of the nodes. This is due to the assumption that the time needed for local computation (local resource demand) is negligible (smaller) compared to the time needed for nodes communications. Readers can refer to Tel's book [Tel94a] for more descriptions on this topic.

The standard measure for evaluating self-stabilizing algorithms is moves complexity. This complexity counts the maximum number of actions for all nodes in the system. Formally,

Definition 4 (Move.) *A move of a node v is one transition from state s_v to a new state s'_v after the execution of the statement of an enabled rule in the algorithm.*

In addition to the standard measure using *moves*, two other time measures, called *Step* and *Round*, are also used in the literature.

Intuitively, a step can be seen as a minimum unit of time that permits the system to transit from a configuration c to a new configuration c' such that every node in c can make one move during one step and such nodes make their moves simultaneously, *i.e.* during one step, one or more nodes execute move and a node may take at most one step. Formally,

Definition 5 (Step.) *A step (*a.k.a. *time-step) is a tuple (c, c'), where c and c' are configurations, such that some enabled nodes, in configuration c, make moves during this this step, and c' is the configuration reached after such nodes made their moves simultaneously.*

Informally, a *Round* is the minimal sequence of steps in which every node gets the chance to be selected for making a move. Formally, the definition of round is as follows:

Definition 6 (Round.) *A Round (*a.k.a. *cycle) is a minimal sequence of steps in which every node that was enabled at the beginning of the round, gets the chance to be selected for making a move if it has not become disabled by a move of its neighbors.*

Hence, the complexity of algorithms is defined as follows:

Definition 7 (Time complexity.) *The time complexity of self-stabilizing algorithms is the maximum number of moves, steps or rounds needed for reaching a legitimate configuration, regardless of starting configuration.*

Usually, the complexity is denoted by $O(f(x))$ where $f(x)$ can depend on the number of nodes (n), the number of edges (m) or other graph characterizations as maximum node degree (Δ) and diameter (D). We also use the notation $f(x) \in O(g(x))$ that's mean $|f(x)| \leqslant k.|g(x)|$ for some positive k.

We have to note that under central daemon, the steps complexity is equivalent to moves complexity, since the daemon selects only one enabled node per step. Moreover, for synchronous daemon, the rounds complexity is equivalent to steps complexity, since under this daemon, a round contains only one step.

Since moves complexity is an upper bound of steps and rounds complexities within any daemon, then it would be more interesting to analyze self-stabilizing algorithms using moves instead of steps or rounds. Moreover, this complexity reflects more the effort system, for example using moves, we can evaluate the energy consumed by all nodes in wireless networks [CCT14]. Thus, a reduction of the number of moves enhances the lifetime of a network. Unfortunately, finding such complexity is still a real challenge for several graph problems.

In this thesis, most proposed algorithms are evaluated using moves, except the algorithms presented in Chapters 7 and 11.

The last complexity measure considered in this thesis refers to the space memory used to implement the algorithm. This complexity measures the amount of the memory required for saving all variables used by the algorithm for each node in system. Also, this complexity includes the size of registers for saving exchanged messages.

2.3.5 Transformers

There are several different distributed models assumed in the literature and therefore we need to design different algorithms to solve a problem in each model. A common approach to avoid this is to design general methods that permit to transform an algorithm from one model to other. Thus, many methods, called *Transformers* have been proposed with self-stabilization. A *Transformer* converts a self-stabilizing algorithm A that runs under a given model to a new self-stabilizing algorithm A' such that A' runs under another model. Note that both of algorithms (A, A') share the same set of legitimate configurations. Usually, these transformers cause overhead complexity in terms of time or space memory.

In general, these transformers can be classified into three types:

1. *Communication model transformers*: In previous section, we describe three common communication models used in distributed systems: state model, shared-register model and message-passing model. Thus, the design of self-stabilizing algorithms depends heavily on the communication model used in the system and an algorithm under specific model cannot run under another communication model. For this reason, many transformers have been proposed in literature for converting a distributed algorithm and preserving self-stabilization property such that transformer from shared memory to message passing proposed in [Dol00] and the transformer from message passing to shared memory presented in [Ioa02]. More transformers and details can be found in [Dol00].

2. *Distance-knowledge transformers*: Using model of computation of distributed

algorithms, a node can read only its variables and those of its neighbors in the case of distance-one model. Thus, it is easier to design a self-stabilizing algorithm for certain problems assuming that a node can read the variables of nodes that are in distance two or more. Then, such algorithm must be transformed in order to run in distributed system (*i.e.* distance-one). Gairing proposed a first transformer that allows a node to act only on correct distance-two knowledge [GGH+04]. The idea is as follows: each node maintains its variables and copies of variables of its neighbors. Thus, in order to maintain these variables up-to-date, a node can execute a move if and only if all neighbors have given their permission. The only inconvenience of this transformer is the slowdown factor of $O(n^2m)$ moves. An extension of this work presented in [GHJT08], where the authors give a generalization of this approach for distance-k knowledge in stead of two. This approach has a slowdown factor of $n^{O(logk)}$ in terms of moves and memory requirement. Recently, Turau proposed a new approach, called *expression model* [Tur12]. This technique transforms algorithms for the distance-two knowledge model on the distance-one knowledge model with a slowdown factor of $O(m)$ moves. In this model, a node maintains its variables and a set of named expressions. The value of an expression is based on the state of the node in question and the states of its neighbors. A node reads the variables of another one in two distance away through the evaluation of the expressions of its neighbors. This approach can be considered as a generalization of the distance-two knowledge transformer proposed by Gairing in [GGH+04].

3. *Daemon transformers*: In addition to communication model used by a system, the design of self-stabilizing algorithms also depends on the daemon assumption (cf. Section 2.3.3). Usually, the algorithms designed under central daemon do not stabilize under synchronous or distributed daemon. For example, the self-stabilizing algorithm for maximal matching in graph proposed in [HH92] does not stabilize and never terminates under synchronous daemon.

   Indeed, designing self-stabilizing algorithms under central daemon is often convenient. However, the central daemon does not consider concurrent executions of two neighbors and therefore it is not directly practicable in real distributed systems. For this reason, several transformers have been proposed for converting an algorithm designed for central daemon into an algorithm that stabilizes under the distributed daemon. Since the distributed daemon is more general than others daemons, then transformation from the distributed daemon to the central daemon is not required.

   In [BPV04], Boulinier et al. developed a transformer that converts an algorithm for the central daemon into an algorithm that runs under the distributed daemon. The core of the proposed transformer is a self-stabilizing local mutual exclusion algorithm.

The distance-two knowledge transformer [GGH+04] and the expression model [Tur12] can also be applied as daemon conversion from central daemon to distributed daemon. The slowdown factors for these transformers are $O(n^2m)$ moves and $O(m)$ moves respectively.

In [GT07], Gradinariu and Tixeuil proposed a new transformer, called *Conflict manager*. The basic idea is as follows: Each node that wants to make a move, sets its Boolean flag in order to inform its neighbors, then, a node is allowed to execute a move if and only if this node has the largest (or the smallest) identifier among the nodes that have set theirs flags. The slowdown factor of this transformer is $O(\Delta)$ moves [GT07]. To the best of our knowledge, this conflict manager is the best effective mechanism for daemon transformation. For this reason, the conflict manager is usually used to compare complexities between algorithms designed under the central and distributed daemons.

In addition, another kind of transformers can also be found in the literature. These transformers allow to convert a distributed algorithm (non self-stabilizing) into a self-stabilizing algorithm [AS88, APSVD94, KP93]. However, these transformers usually sacrifice either convergence time complexity or memory requirements.

Usually, it is more efficient to develop specific self-stabilizing algorithms for each model. However, these studies demonstrate the expressive power that have self-stabilizing systems. In this thesis, we use only the distance-one knowledge because it is most suitable in the real systems and we use also some daemon transformers in order to compare the complexities between algorithms that operate under the central daemon and the distributed daemon.

2.3.6 Proof techniques

As mentioned in Section 2.2, most self-stabilizing algorithms for graph problems are silent, then proving their correctness is usually not difficult task; *i.e.* it is sufficient to prove that in configuration where no node is enabled, the configuration of the system is legitimate. However, proving the convergence of these algorithms is a challenging task. For proving the convergence (second property) of a self-stabilizing algorithm, several techniques has been proposed in the literature. This section describes the main proof techniques that we use some of them for proving convergence and complexity analysis of our algorithms presented in the following chapters.

Variant Function: In [Kes88], Kessels proposed an approach for the first time by using a Variant Function (*a.k.a. Potential Function*) to prove the convergence of self-stabilizing algorithms. This technique measures the progress and the evolution of an algorithm during its execution. The basic idea is to use a function over the configuration set whose value is bounded, to prove that this function monotonically increases (or decreases) when nodes execute any rule. This can be done for example by counting nodes with certain properties. There exist very simple examples

for variant functions in [AB93, Dol00, Kes88, Tel94b]. However for a majority of algorithms, only very complex variant function were found. An exercise on Variant Function proof is given in Chapter 4. Thus, finding such a variant function for arbitrary systems is not trivial and requires a lot of intuition. Theel writes in [The00]: "*... deriving a variant function for arbitrary systems is regarded as an art rather than a craft*".

Attractor: The technique of attractor (*a.k.a Convergence Stairs*) is used to prove the convergence of a self-stabilizing algorithm when it is difficult to find variant function. The idea is to define a sequence of predicates $p_1, \ldots, p_k$ over the configuration set, where all legitimate configurations satisfy the predicate p_k. Moreover, each predicate p_{i+1} is a refinement of p_i where $1 \leqslant i \leqslant k$. The predicate p_{i+1} refines the predicate p_i if p_i holds whenever p_{i+1} holds. The term *attractor* is often used for each p_i predicate [Dol00]. Then, the goal is to prove that a system in which p_i holds reaches a configuration satisfies p_{i+1}. This technique is used in different works such as [JM11, KM08, JM14].

Global State Analysis: A single node has not knowledge about the configuration of the whole system. However, this global view can be used for proving the termination of an algorithm. For instance, it may be possible to prove that there is no configuration that can occur twice. This proves that the number of possible configurations is finite due to the fixed number of nodes and their local states. Usually, most algorithms define several local states for each node, this causes an exponential number of possible configurations C_G. Hence, this technique may not be a good decision when the goal is to prove the performance of an algorithm. For example, in [SX07], Srimani et al. used this technique for proving the convergence of a self-stabilizing algorithm for computing a minimal weakly connected dominating set. They prove that no configuration can occur twice for showing the termination of their algorithm, but the authors did not give an upper bound for the complexity. Later, Hauck presents an example in [Hau12] where the algorithm proposed by Srimani needs an exponential moves to stabilize under a central daemon.

Analysis of Local States and Sequences: Contrary to the global state analysis, this technique considers only the analysis of the state of a single node and its neighbors. Some systems have the property that nodes become disabled after executing certain moves. The basic idea is to show that any node in the system has a bounded number of moves or bounded number of state sequence. This technique is used in [Tur07] and [GHJ+08] for proving the convergence of self-stabilizing algorithms for dominating set problems. A detailed description of this method can be found in chapters 5 and 8.

Graph Reduction and Induction: Recently in [TH11], Turau and Hauck developed a new technique to prove the stabilization under central and distributed daemon. The basic idea of this technique is to create a mapping from the algo-

rithm's execution sequence of a graph to that of a reduced graph. This allows to use complete induction proofs [TH11]. The authors used this technique for finding the worst-case complexity of self-stabilizing algorithms for finding the maximum weight matching with approximation ratio 2. Inspired by this technique, we analyzed the complexity of our algorithm for computing p-star decomposition, presented in Chapter 7.

Neighborhood Resemblance: This technique is used to prove lower bounds of memory to solve a given problem within self-stabilizing paradigm. In fact, using this technique, we obtain some impossibility results, *i.e.* it is impossible to find a self-stabilizing algorithm for a given problem with less than a certain amount of memory. Using this method, we prove that all self-stabilizing algorithms for the decomposition into triangles (*cf.* Chapter 5) and p-stars (*cf.* Chapter 7) under distributed daemon require a certain amount of memory for breaking symmetry between nodes.

Finally, we have to note that there is no general technique that is suitable to all self-stabilizing algorithms for verifying their convergence. Then, it is very difficult and an important step to choose which method maybe the best adapted for proving the convergence and the complexity of a given algorithm.

2.4 Self-stabilizing algorithms for some graph problems

Given the importance of graph theory for studying different problems that arise in many areas (communication networks, scheduling, distributed computing), several self-stabilizing algorithms for classic graph parameters have been developed in this direction, such as self-stabilizing algorithms for finding minimal dominating sets, coloring, maximal matching, maximal packing, spanning tree [BM12, MMPT09, DWS15, Joh97]. Several surveys of such algorithms can be found in the literature [Her02, GK10]. Herman [Her02] presents a list of self-stabilizing algorithms according to several categories such as topology or proof techniques. Gartner [Gär03] surveys self-stabilizing algorithms for spanning trees. Later, Guellati and Kheddouci [GK10] present a survey of self-stabilizing algorithms for independence, domination, coloring, and matching problems. In this part, some references on matching, domination and independence problems are summarized and more recent works are presented in the following sections.

2.4.1 Matching

A matching is a classical problem in graph theory. Matching in a undirected graph $G(V, E)$ is a set M of independent edges (*i.e.* node-disjoint). A matching M is maximal if no proper superset of M is also a matching (*i.e.* there is no another matching M' such that $M \subset M'$). Figure 2.4 presents a maximal matching of a graph. A matching M is *maximum* if it has the largest cardinality ($|M|$) among

all possible matchings in G. Matching problem has many applications in fields as diverse as transversal theory, assignment problems [BNBJ+08], network flows [REJ+07], scheduling in switches [WS05] and so on. since it is associated with marriage-like problems where the goal is to form maximum couples while optimizing specific criteria. For example, in networks, each client communicates with only one server. More details on applications of matching can be found in [Gib85].

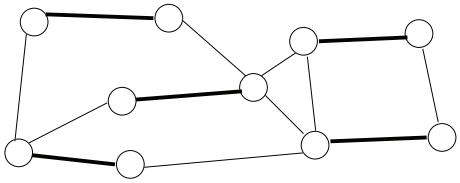


Figure 2.4: A maximal matching M in a graph G. (The depicted edges form the matching)

The first self-stabilizing algorithm for computing a maximal matching was proposed by Hsu and Haung in [HH92]. The algorithm is uniform and works in anonymous system. It assumes a central daemon. The proposed algorithm maintains a variable for each node v in the system that contains a pointer. This pointer may be *null* or may point at a v's neighbor. Two nodes v and u are matched (*i.e.* married) if and only if they point at each other. Then, in final configuration, each pairs of matched nodes form a maximal matching. The basic idea of the algorithm is as follows: Each node v that points *null* will point at an arbitrary neighbor u such that u points at v (which means v accepts to be matched with u). If a node v that points to *null* and any of its neighbors points at v then v points an arbitrary neighbor u such that u points to *null* (which means that v invites/proposes a node u to be matched). Then, a node v that points at neighbor u and the latter points at another node w, so v will change its pointer to *null* (which means that v withdraws its proposition/invitation). Hsu and Huang proved that the time complexity is $O(n^3)$ steps. The complexity of the same algorithm was improved to $O(n^2)$ steps by Tel in [Tel94b] and later it was improved to $O(m)$ steps by Hedetniemi et al. in [HJS01]. Chattopadhyay et al. proposed in [CHS02], two algorithms for the same problem with read/write atomicity (cf. Section 2.1). The first algorithm that stabilizes in $O(n)$ rounds assumes that each node has a distinct local identifier. The idea that each node tries to be matched with its neighbor that has the minimum identifier. The authors extend this version by proposing the second algorithm for anonymous system with (n^2) rounds under central daemon. However, the second algorithm assumes that each node knows Δ (maximum node degree in the system) and G is a bipartite graph. In [GHJS03c], Goddard et al. proposed a synchronous version of the algorithm Hsu and Haung that stabilizes in $O(n)$ rounds in mobile ad-hoc networks. However, the authors assumed distinct local identifiers for nodes and communication is ensured through message exchanges between nodes. Goddad et al. in [GHS06] proposed a uniform version for finding 1-maximal matching in trees assuming central daemon. A 1-maximal matching is maximal matching and its

cardinality cannot be increased by removing an edge and adding two edges. Their algorithm is based on the algorithm of Hsu and Haung and by adding a mechanism for exchanging an edge of the matching by two when it was possible. The proposed algorithm needs $O(n^4)$ steps. In [MMPT07, MMPT09], Manne et al. proposed an algorithm for maximal matching that stabilizes in $O(m)$ moves under distributed daemon. The authors assumed distinct local identifiers within distance two. In the algorithm, each node maintains two variables, one variable for pointer (the same as used by Hsu and Haung in [HH92]) and one boolean variable for informing neighbors whether the node is matched or not. The basic idea of the algorithm is as follows: (1) a node can make a move if and only if its boolean variable is updated. (2) a non-matched node v invites its neighbor u if u has a greater identifier and u is non-matched node. (3) a node v accepts an invitation of a neighbor u if v is pointed by u. (4) a node v withdraws its invitation from u (v sets it pointer to $null$) if u is either already matched with another node (i.e. u points to another node and has its boolean variable to true) or u has a lower identifier than v.

Another variant of matching, called generalized matching (b-matching), was proposed by Goddard et al. in [GHJS03b]. The b-matching is considered as a generalization of classical matching where each node in the graph is matched with at most b neighbors. The algorithm converges in $O(m)$ moves under central daemon.

We have to note that there is no self-stabilizing algorithm for finding a maximum matching of general graphs in the literature. However, there are some algorithms for certain classes of graph such tree [KS00] or bipartite graphs [CHS02].

Considering weighted graphs, Manne et al. proposed a self-stabilizing for maximum weighted matching in general graph [MM07]. The authors give upper bounds of $O(2^n)$ moves under the central daemon and $O(3^n)$ for the distributed daemon. Recently, Turau and Hauck improve this complexity in [TH11]. The authors present a new analysis of the algorithm proposed by Manne and they proved that the same algorithm stabilizes in $O(mn)$ moves under the central daemon. Moreover, the authors give a modified version that stabilizes within $O(mn)$ moves within the distributed daemon.

In this thesis, different self-stabilizing algorithms are developed that can be also considered as generalization of maximal matching in graphs. The maximal partitioning into triangles can be used for finding a maximal tripartite matching in a graph where a node is matched with two neighbors instead of one, for example: given three sets B, G and H that represent the sets of boys, girls and homes respectively and their elements have ternary relation $T \subseteq B \times G \times H$. The question is to find a maximal set of triples in T such that no two of which have a component in common. In other words, each boy is married to a different girl and each couple has a home of its own. For this, two self-stabilizing algorithms for maximal partitioning into triangles were proposed in this thesis, called SMPT^c and SMPT^D. Moreover, other self-stabilizing algorithms (called SMSD^1 and SMSD^2) developed in this thesis for finding a p-star decomposition in arbitrary graph, also provide a maximal matching if p is fixed to 1. More details on these generalizations can be found in Parts I and II. The algorithms presented for the maximal matching problem and its variants

are summarized in Table 2.1.

Reference	Result	Topology	Anon.	Daemon	Complexity
[HH92]	Maximal	Arbitrary	Yes	Central	$O(m)$ moves
[CHS02]-1	Maximal	Arbitrary	No	Distributed	$O(n)$ rounds
[GHJS03c]	Maximal	Arbitrary	No	Synchronous	$O(n)$ rounds
[GHJS03b]	Generalized	Arbitrary	Yes	Central	$O(m)$ moves
[GHS06]	1-Maximal	Tree	Yes	Central	$O(n^4)$ moves
[MMPT09]	Maximal	Arbitrary	No	Distributed	$O(m)$ moves
SMSD[1] (p=1)	Maximal	Arbitrary	No	Distributed	$O(n)$ rounds
[KS00]	Maximum	Tree	Yes	Central	$O(n^4)$ moves
[CHS02]-2	Maximum	Bipartite	Yes	Central	$O(n^2)$ rounds
SMPT^D	Max. Tripartite	Arbitrary	No	Distributed	$O(m)$ moves
[MM07]	1/2 ap. max. wei.	Arbitrary	No	Distributed	$O(3^n)$ moves
[TH11]	1/2 ap. max. wei.	Arbitrary	No	Distributed	$O(mn)$ moves

Table 2.1: Self-stabilizing algorithms for maximal matchings and its variants.

2.4.2 Dominating setss

Domination in graphs has been extensively studied in graph theory. In a graph $G = (V, E)$, a set of nodes $D \subseteq V$ is called a *dominating set* (DS) if every node of V is either in D or has a neighbor in D, *i.e.* $\forall v \in V - D : N(v) \cap D \neq \emptyset$. A dominating set is *minimal* (MDS) if no proper subset of D is a dominating set (see Figure 2.5).

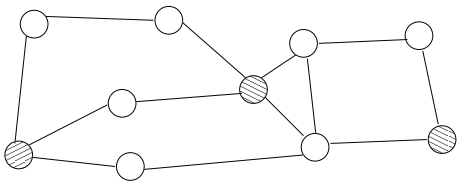


Figure 2.5: Minimal Dominating Set D of a graph G. (The members of D are hatched)

In the literature, there are several self-stabilizing algorithms for finding different variants of dominating sets such as total Dominating set, k-dominating set, connected dominating set and weakly connected dominating set. A set D is called *total dominating set* (TDS) if every node of the graph has a neighbor in D, *i.e.* $\forall v \in V : N(v) \cap D \neq \emptyset$. The set D is called *k-dominating set* (KDS) if every node outside of D has at least k neighbors inside D. A dominating set D is said *connected* dominating set (CDS) if D is connected and it is called *weakly connected* (WCDS)if the subgraph weakly induced by D, *i.e.* $(N[D], E \cap (D \times N[D]))$ is connected, where $N[S] = \bigcup_{v \in S} N[v]$.

The structure of dominating sets can be used as virtual overlays in a distributed system. These structures are often used for designing efficient protocols in wireless

and ad-hoc networks [GHJ+08, UT11, GHJS03a, BDTC05]. Minimal dominating set can be used for locating some nodes to be servers; thus clients must be closely with the servers [GHJ+08]. Connected dominating sets and weakly connected dominating set are often used to represent virtual backbone in wireless networks [BDTC05].

Simple domination variant.

Hedetniemi et al. [HHJS03] developed two self-stabilizing algorithms for dominating set (DS) and the minimal dominating set (MDS). The proposed algorithms work for any connected graph and assume a central daemon. The basic idea of their first algorithm is to partition the graph into two disjoint sets of nodes such that each set is dominating set. For this, each node has a boolean variable that indicates whether it is in the first set or in the second set. Then, a node changes its states if all neighbors have the same state. The authors called this method a dominating bipartition. Inspired from the algorithm MDS in [HHJS03], Xu et al. proposed an algorithm for finding an MDS under the synchronous daemon [XHGS03]. The basic idea is as follows: each node maintains a boolean variable (to indicate if the node belongs to the dominating set or not) and a pointer. This pointer is *null* if a node v is dominated by at least two neighbors; otherwise the node v points at a unique neighbor that dominates it. Thus, if any node v is not dominated by any neighbors, then this node will enter to dominating set by changing its variable to true. In [GHJS03a], Goddard et al. proposed self-stabilizing algorithm for computing a minimal total dominating set (MTDS), inspired form the algorithm for MDS in [XHGS03] and assumed a central daemon. However, the authors proved that the algorithm is finite and no complexity analysis is given. Later in [GHJS05], the same algorithm is proved that it stabilizes in exponential moves under central daemon. Recenlty in [BYK14], Belhoul et al. present an efficient self-stabilizing algorithm for finding MTDS using the expression model (*cf.* Section 2.3.5) proposed by Turau in [Tur12]. Their algorithm converges in polynomial moves under the distributed daemon. In [GHJ+08] Goddard et al. proposed another algorithm for finding an MDS in an arbitrary graph under the distributed daemon. In addition to a boolean variable, the algorithm uses an integer to count the number of neighbors that are members of MDS. Using these counters, a node can make the decision to enter or to leave the MDS. The authors assume the distributed daemon and nodes have distinct local identifiers. In [Tur07], Turau extends his MIS algorithm to design a self-stabilizing algorithm for the MDS problem. In addition to the pointers as used in [XHGS03], the author in [Tur07] used three states (*In, Out, Wait*) for each node in the system. Then each node in state *Out* (*i.e.* not belonging to MDS) must transit by state *Wait* before to enter the MDS (*i.e.* will be in state *In*). A node in state *Wait* can enter to MDS if it is not dominated by any neighbor and this node has the smallest *id* among the waiting nodes. Using this idea, the author proved that the proposed algorithm converges after at most $9n$ moves within the distributed daemon. Recenlty in [CCT14], Chiu et al. inspired from the technique

proposed by Turau in [Tur07] and they proposed a new algorithm for MDS problem within complexity at most $4n$. The authors used four states (*In, Out1, Out2, Wait*) in order to improve the moves complexity.

Multiple domination.

Concerning the multiple domination, several self-stabilizing algorithms have been proposed in the literature. Kamei and Kakugawa [KK03] presented two algorithms for the minimal k-dominating set (MKDS) problem in a tree. The first algorithm works in anonymous system and assumes a central daemon, while the second one assumes that nodes have global unique identifiers and works under distributed daemon. In preference to propose a new algorithm, Huang et al. [HCW08] relaxed some assumptions used with the first MKDS algorithm proposed in [KK03]. The authors showed that the same algorithm converges in polynomial times to find 2-dominating set (M2DS) in an arbitrary graph, under a central daemon. Another algorithm for MKDS was proposed by Kamei and Kakugawa in [KK05]. Their algorithm works for arbiratry graph and assumes that nodes have distinct local identifiers. The authors proved that the algorithm converges in linear time under synchronous daemon. Huang et al. [HLCW07] proposed the first general algorithm for finding M2DS in arbitrary graph under distributed daemon. Using expression model (*cf.* Section 2.3.5), Turau presents in [Tur12] an efficient self-stabilizing algorithm for MKDS in arbitrary graphs. The algorithm stabilizes in polynomial moves under the distributed daemon.

Connected Dominating Set.

Considering connected dominating set (*i.e.* the set D is connected), Jain and Gupta [JG05] proposed an algorithm to construct a CDS in arbitrary graphs. The proposed algorithm works under a synchronous daemon. A special communication model used by the authors where nodes are assumed to have instant read access in their 3-hop neighborhood and write access in their 2-hop neighborhood. The algorithm proposed by Drabkin et al. in [DFG06] also assumes 2-hop read access for the nodes. the algorithm constructs a CDS in arbitrary graphs assuming the distributed daemon. Goddard and Srimani [GS10] proposed the first self-stabilizing algorithm for CDS in arbitrary graphs that handles both anonymous nodes and a distributed daemon at the same time. Kamei and Kakugawa [KK10] proposed an algorithm that constructs a connected minimum dominating set (CMDS) assuming that a rooted BFS (Breadth-First Spanning) tree of the graph is given. The algorithm operates under a central daemon. The same authors proposed in [KK08] an algorithm that constructs a CMDS in arbitrary graphs under the synchronous daemon. In [RTAS09], Raei et al. proposed a self-stabilizing algorithm for finding a CMDS under the central daemon. The algorithm assumes a disk graph with bidirectional links (DGB). This model allows the nodes to have different ranges. The authors prove a constant approximation ratio for the proposed algorithm.

Weakly Connected Dominating Set.

A Weakly Connected Dominating Set is a Dominating Set where the induced subgraph of the closed neighborhood of the set is connected. In [SX07], Srimani and Xu developed the first self-stabilizing algorithm that finds a weakly connected minimal dominating set (WCMDS) under the distributed daemon. This algorithm assumes that a breadth first spanning (BFS) tree of the graph is given and converges in exponential moves. Kamei and Kakugawa [KK07] proposed an algorithm that constructs a WCMDS in arbitrary graphs under the synchronous daemon with polynomial complexity. Turau and Hauck [TH09] proposed an algorithm that constructs a WCMDS under the distributed daemon. The known algorithm uses a polynomial algorithm that constructs a BFS tree of a given graph. Recently in [DWS14], Ding et al. proposed an algorithm that constructs a WCMDS in linear rounds under the synchronous daemon.

Independent Strong Dominating Set.

All previous algorithms cited above do not consider node's degrees. However, the nodes having higher degrees in graphs usually play important roles in distributed systems. For example, clustering in wireless networks [YKR06], providing stable cluster structures [KMW04] and studying of communities structures in p2p [LHK13]. In [SL96], Sampathkumar introduced to graph theory an interesting variant of dominating sets problem, called *Independent Strong Dominating Set* (ISD-set). In addition to its domination and independence properties, the ISD-set considers also nodes degrees. Given a graph $G = (V, E)$, a set $D \subseteq V$ is an independent set if no two nodes of D are adjacent. A node v strongly dominates a node u and u weakly dominates v if $(u, v) \in E$ and $deg(v) \geqslant deg(u)$. A set $D \subseteq V$ is an ISD-set of G if D is an independent set and every node in $V - D$ is strongly dominated by at least one node in D. In part III, we present a self-stabilizing algorithm for computing a minimal ISD-set of an arbitrary graph, called *ISDS*. The algorithm provides MDS and MIS at the same time and converges in linear rounds within unfair distributed daemon.

The algorithms presented for the dominating set problems and its variants are summarized in Table 2.2.

2.4.3 Independent sets

As defined above, a set of nodes is an *Independent Set* if no two nodes are adjacent. A maximal independent set (MIS) is an independent set that is not properly contained in any other independent set with bigger cardinality. The definition of MIS implies that for any graph $G = (V, E)$, if a node is not in the MIS, then it must be adjacent to at least one node in the MIS. Therefore, an MIS of a graph G is also a minimal dominating set, however an MDS is not necessary an MIS. The dominance property of the MIS and the sparseness of its nodes make it an important structure

Reference	Result	Topology	Anon.	Daemon	Complexity
[HHJS03]-1	DS	Arbitrary	Yes	Central	$O(n)$ moves
[HHJS03]-2	MDS	Arbitrary	Yes	Central	$O(n^2)$ moves
[XHGS03]	MDS	Arbitrary	No	Synchronous	$O(n)$ rounds
[GHJ+08]	MDS	Arbitrary	No	Distributed	$O(n)$ moves
[Tur07]	MDS	Arbitrary	No	Distributed	$O(n)$ moves
[CCT14]	MDS	Arbitrary	No	Distributed	$O(n)$ moves
ISDS	MDS/MIS	Arbitrary	No	Distributed	$O(n)$ rounds
[GHJS03a]	MTDS	Arbitrary	No	Central	Exponential moves
[BYK14]	MTDS	Arbitrary	No	Distributed	$O(mn)$ moves
[KK03]-1	MKDS	Tree	Yes	Central	$O(n^2)$ moves
[KK03]-2	MKDS	Tree	No	Distributed	$O(n^2)$ moves
[HCW08]	M2DS	Arbitrary	Yes	Central	$O(n)$ moves
[KK05]	MKDS	Arbitrary	No	Synchronous	$O(n^2)$ rounds
[HLCW07]	M2DS	Arbitrary	No	Distributed	–
[DLV10]	MKDS	Arbitrary	No	Distributed	$O(k)$ rounds
[DHR+11]	MKDS	Arbitrary	No	Distributed	$O(Dn^2)$ rounds
[Tur12]	MKDS	Arbitrary	No	Distributed	$O(nm)$ moves
[JG05]	CDS	Arbitrary	No	Synchronous	$O(n^2)$ moves
[DFG06]	CDS	Arbitrary	No	Distributed	$O(n)$ moves
[GS10]	CDS	Arbitrary	Yes	Distributed	–
[KK10]	CMDS	BFS tree	No	Central	$O(k)$ rounds
[KK08]	CMDS	Arbitrary	No	Synchronous	$O(n)$ rounds
[RTAS09]	CMDS	DGB	No	Central	$O(n^2)$ moves
[SX07]	WCMDS	BFS tree	No	Distributed	$O(2^n)$ moves
[KK07]	WCMDS	Arbitrary	No	Synchronous	$O(n^2)$ rounds
[TH09]	WCMDS	BFS tree	No	Distributed	$O(mn)$ moves
[DWS14]	WCMDS	Arbitrary	No	Synchronous	$O(n)$ rounds

Table 2.2: Self-stabilizing algorithms for dominating sets and its variants.

for many applications, such as clustering in wireless ad hoc networks [AWF03].

Since MDS and MIS are strongly related, many self-stabilizing algorithms were also proposed for finding MIS. The first self-stabilizing algorithm that finds a maximal independent set was proposed by Shukla et al. in [SRR+95]. The algorithm assumes a central daemon and converges in $O(n)$ moves. Another algorithm for the MIS problem was presented by Ikeda et al. in [IKK02]. Their algorithm operates under the distributed daemon and stabilizes in $O(n^2)$ steps. In 2003, Goddard et al. [GHJS03c] proposed a self-stabilizing algorithm that maintains an MIS in a mobile ad-hoc network. The authors assume a synchronous model and proved that the algorithm converges in $O(n)$ rounds. Shi et al. [SGH04] give a particular interest for MIS problem in anonymous systems. They presented a self-stabilizing algorithm for the *1-maximal independent set* (1-MIS) problem in tree graphs. A 1-maximal independent set means that the set is a MIS, with the additional property that is the cardinality of MIS cannot be increased by removing one node and adding more other nodes. Their algorithm operates only under the central daemon and stabilizes

in $O(n^2)$ moves. In [Tur07], Turau proposed an efficient self-stabilizing algorithm for the MIS problem. The algorithm stabilizes in $O(n)$ moves under the unfair distributed daemon. The algorithms presented for the independent set problems are summarized in Table 2.3.

Reference	Result	Topology	Anon.	Daemon	Complexity
[SRR+95]	MIS	Arbitrary	Yes	Central	$O(n)$ moves
[IKK02]	MIS	Arbitrary	No	Distributed	$O(n^2)$ steps
[GHJS03c]	MIS	Arbitrary	No	Synchronous	$O(n)$ rounds
[SGH04]	1-MIS	Tree	Yes	Central	$O(n^2)$ moves
[Tur07]	MIS	Arbitrary	No	Distributed	$O(n)$ moves
ISDS	MDS/MIS	Arbitrary	No	Distributed	$O(n)$ rounds

Table 2.3: Self-stabilizing algorithms for independent sets problem.

2.5 Conclusion

Self-stabilization is an elegant approach for fault-tolerance in distributed system. In this chapter, we introduced the self-stabilization paradigm, then, we presented the basic concepts required for better understanding of the communication's models. We also provided in this chapter a classification of several transformers proposed in this domain.

Finally, we presented a brief survey of self-stabilizing algorithms proposed for independent sets, dominating sets and matching problems. These algorithms constitute the basis of our reasoning for the resolution of four parameters addressed in this thesis: *Partitioning into Triangles*, *p-Star Decomposition* and *Edge Monitoring & Independent Strong Dominating sets* problems. These four problems can be considered as the generalization of the two parameters matching and domination in graphs. More descriptions of the link between theses parameters can be found in the following chapters.

Part I

Partitioning into Triangles (MPT)

Chapter 3

Introduction and motivation of part I

Contents

3.1 Introduction

In the two past decades, distributed systems began to expand and became larger, making their control and management much harder. A new line of research on system partitioning (*a.k.a.* decomposition) is launched and motivated by the simplification and improvement of system management.

Thus, graph partitioning finds applications in various fields including scientific computing, scheduling, load balancing and network communications. Graph partitioning problem is defined on a graph $G = (V, E)$, where V is the set of nodes and E is the set of edges, such that G is decomposed into small disjoint components having specific properties. These properties are often defined on the size of the partitions (clusters), on their shape (subgraphs) or both (patterns). There is a rich literature on graph partitioning [FPSV09, OR09, Shy10]. However, only a small fraction consider the self-stabilization paradigm.

Graph partitioning into clusters was considered in some works as [BDJV05, CDDL09, JM14]. Usually, a cluster contains one clusterhead (*a.k.a* leader) and some ordinary nodes (*a.k.a* members). Often, the criteria considered in the clustering takes into account the distance between nodes and their clusterhead [BDJV05, CDDL09] or the bound number of nodes in each cluster [JM14].

Another decomposition was proposed by Belkouch et al. in [BBCD02]. The authors considered a particular graph partitioning problem that consists in decomposing the graph with k^2 nodes into k partitions of order k. Their algorithm relies on a self-stabilizing spanning tree construction. Considering the shape (topology) of each partition, *Ishii et al.* proposed a self-stabilizing algorithm for partitioning an arbitrary graph into maximal cliques [IK02].

In this part of thesis, we focus on the partitioning into triangles of general graphs. More specifically, we will study its local maximization variant, called *Maximal Partitioning into Triangles* (MPT). Partitioning into triangles (PT) describes a graph as the union of disjoint partitions where each partition is a triangle. PT is *Maximal* if no triangle can be added to this partitioning using only nodes not already contained in partitions. Observe that a maximal partitioning into triangles is not perfect (*i.e.* without a rest). In this part, we will study the problem of maximal partitioning into triangles in distributed systems using self-stabilization paradigm. Moreover, two self-stabilizing algorithms for such partitioning are developed considering the central and the distributed daemons. More details on these algorithms can be found in chapters 4 and 5.

The maximal partitioning into triangles (MPT) can be considered as a generalization of the maximal matching problem where nodes are matched with two of their neighbors instead of one. This generalization is called *Tripartite Matching* in graph theory. More formal definitions of this partitioning are given in Section 3.2. Furthermore, some applications of MPT are provided in Section 3.3 in order to motivate the study of this variant.

3.2 Overview and definitions

The *Partitioning into Triangles* (PT) is one of the classical NP-complete problems and it is defined as follows. Given q such that $n = 3q$ where q is a positive integer and n is the number of nodes in the graph G, a partitioning into triangles consists of q disjoint sets $T_1, T_2, ..., T_q$ where each T_i forms a triangle in G. The NP-completeness proof of this partitioning problem was presented in [GJ79].

However, another problem linked to graph partitioning into triangles, called *node disjoint triangle packing* consists in finding the maximum number of node disjoint triangles in a graph. It is well known that finding this number in arbitrary graph is NP-Hard [CR02].

The partitioning into triangles can be also viewed as *Tripartite Matching* problem in graphs. Tripartite matching is defined as follows: Given three disjoint sets of nodes B (Boys), G (Girls) and H (Homes) where $|B| = |G| = |H| = n$ and a ternary relation (*i.e.* affinities) $T \subseteq B \times G \times H$. The question is to find n triples in T such that no two of them have a component in common (See Figure 3.1). This decision problem is known to be NP-complete [Pap94].

Since finding the maximum number of disjoint triangles is NP-hard, and deciding if a graph can be partitioned into triangles is NP-complete, we consider the following local variant of graph partitioning called *Maximal Graph Partitioning into Triangles* (MPT). The MPT of graph G is a set of disjoint subsets T_i of nodes such that each subset T_i forms a triangle and no triangle can be added to this set using only nodes not already contained in a set T_i. Formally, a given partitioning PT is maximal if there are no $v, u, w \in V \backslash PT$ such that $(v, u), (u, w), (v, w) \in E$.

However, this local maximization provides at least a third of the maximum

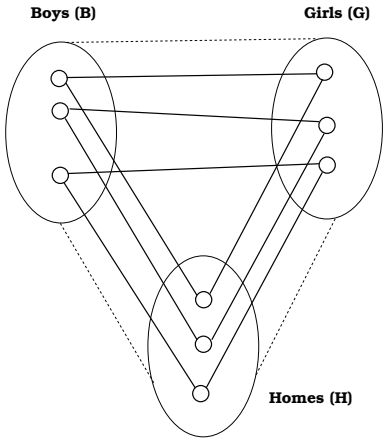


Figure 3.1: Tripartite matching in a graph

triangles partitioning of an arbitrary graph G. The Figure 3.2 depicts a gadget graph to present this ratio. The structure of G is illustrated in this figure. Then $MPT = \{(1,2,3),(4,5,6),(7,8,9)\}$ is the maximum partitioning into triangles of G and $MPT = \{(3,4,7)\}$ is a maximal partitioning into triangles of G. We note that each triangle that belongs to the partitioning can *desactivate* at most three other disjoint triangles in G. Hence, the maximal partitioning contains at least a third of the maximum partitioning into triangles of an arbitrary graph G.

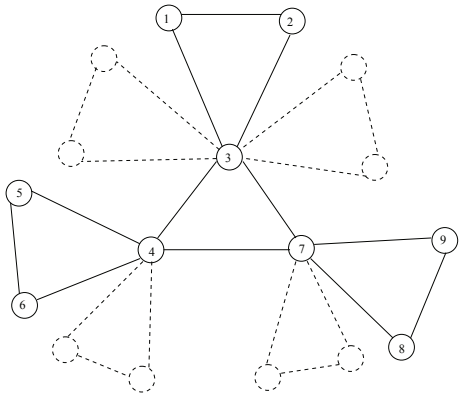


Figure 3.2: A gadget graph

3.3 Motivation

The classical matching problem consists in finding the maximum number of independent edges in a given graph. This problem has received large interest due to the abundant number of applications in fields as diverse as transversal theory, tasks assignment [BNBJ+08], network flows [REJ+07], and scheduling [WS05]. Exam-

ples include the problem of assigning tasks to workers or connecting client to server where each machine in the network may need to choose exactly one neighbor to communicate with. Many studies have addressed this problem even in the field of self-stabilization (*cf.* Section 2.4). As graph partitioning into disjoint triangles is a generalization of maximal matching then many applications of maximal matching also apply. As an application of the assignment problem, each client can communicate with only two specific servers and the latter communicate between them in order to satisfy the client.

In addition to its theoretical aspects, the maximal partitioning into triangles is motivated by several practical aspects in distributed system. MPT was shown to be effective in terms of energy consumption and scalability support. Delouille et al. proposed an efficient approach based on this partitioning for coming up with accurate estimates of values measured in wireless sensors networks [DNCB03, DNB06].

More recently, triangle patterns were used in community detection problems [SAG11] and for studying their robustness in peer-to-peer social networks [LHK13]. On the theoretical side, the partitioning problem is closely related to the tripartite matching problem in graphs.

The contribution of the first part is the study of the *Maximal Partitioning into Triangles* of general graphs and the presentation of different self-stabilizing algorithms for finding such partitioning using different daemons. Chapter 4 presents the generalization of maximal matching algorithm of Hsu and Huang, in order to develop the first self-stabilizing algorithm for graph partitioning into triangles under the central daemon. Formal proofs are given for showing the correctness of the first proposed algorithm followed by its convergence proof using the variant function technique. Chapter 5 presents an improved version of the algorithm presented in Chapter 4. The second algorithm converges in linear time using the distributed daemon. Formal proof of its correctness and its convergence proof using local states technique are also presented. Moreover, impossibility result for finding a deterministic self-stabilizing algorithm for such partitioning in anonymous system is provided. Section 5.7 concludes this first part.

Chapter 4

Algorithm for MPT under the Central Daemon

Contents

4.1 Introduction

In the previous chapter, we introduced the problem of *Maximal Partitioning into Triangles* (MPT) and we presented some of its applications in distributed systems. In this chapter, we present a first self-stabilizing algorithm for finding an MPT of an arbitrary graph, called SMPT^c. The algorithm works under the unfair central daemon and assumes that nodes have distinct local identifiers. A preliminary version of this work appeared in [NHK12].

Note that this chapter is devoted to present our reasoning to solve MPT problem based on Hsu and Huang's algorithm for maximal matching in arbitrary graphs. Although the algorithm will be improved in the next chapter, it is useful to develop the first version in order to improve the readability of the second proposed algorithm.

This chapter is organized as follows: In Section 4.2, we present the proposed algorithm SMPT^c for this problem. Then, we give formal proofs of its correctness and convergence in Section 4.3 and Section 4.4 respectively. Furthermore, the complexity analysis is developed in Section 4.5.

4.2 Algorithm description

Before presenting our algorithm for maximal partitioning into triangles, we briefly revisit the essential design of the original maximal matching algorithm of Hsu and Huang [HH92], already tackled in Section 2.4; the description of its rules is given in Algorithm 1. Let's recall that $N(v)$ denotes the set of neighbors of a node v.

Each node v maintains one variable, called $v.p$, which is either $null$, or contains an identifier id of a neighbor, we say v points at a neighbor u such that $u \in N(v)$. The algorithm has three rules: acceptation rule `[A]`, invitation `[I]` and withdrawal `[W]`: the edge between two neighbors nodes becomes part of a matching when each node points at the other. The Rule `[A]` allows a node to accept a proposed matching (*i.e.* accept an invitation) by a neighbor. The Rule `[I]` permits a node to invite another node to be matched. The Rule `[W]` allows a node to withdraw an invitation.

Algorithm 1: Maximal Matching Algorithm of Hsu and Huang

Nodes: v is the current node

$(v.p = null) \wedge (\exists u \in N(v) : u.p = v) \quad \longrightarrow v.p = u;$ `[A]`

$(v.p = null) \wedge (\forall w \in N(v) : \neg(w.p = v)) \wedge (\exists u \in N(V) : u.p = null) \longrightarrow v.p = u;$ `[I]`

$(v.p = u) \wedge (u.p = w) \wedge (w \neq v) \quad \longrightarrow v.p = null;$ `[W]`

However, the maximal matching algorithm of Hsu and Huang is not suitable for MPT problem since each node is matched with only one neighbor. Indeed, the MPT problem is a generalization of the maximal matching, where each node is matched with two neighbors. Thereby, we extend the Hsu algorithm to face the MPT problem by allowing each node to maintain a list of pointers to its neighbors.

Thus, the main idea of the proposed algorithm for maximal partitioning into triangles can be summarized as follows: each node v, in the graph G, maintains a list of pointers $v.L$ that defines to which triangle v may belong. We say $v.L$ is valid, if $|v.L| = 0$ or $|v.L| = 2$; $v.L$ contains only pointers (id) to neighbors of v ($v.L \subseteq N(v)$) and does not contain duplicate id. So, it is possible that at the starting of the system, the list to be not valid. However, it is easy to add a rule that forces it to become valid. For this reason and to simplify the description of the algorithm, we assume that these lists are valid.

Furthermore, each node v maintains a variable S which contains its closed neighborhoods ($N[v]$). Through the variable S of each u's neighbor, a node u knows its neighbors at distance two.

The proposed algorithm SMPT^c permits to the nodes to coordinate between them in order to belong to disjoint triangles. To do this, we have four rules:

- The updating Rule ($[U]$): when the variable $v.S$ of the node v contains an incorrect list of the closed neighborhood, the node v updates its variable $i.e$ to set $v.S = N[v]$.
- The invitation Rule ($[I]$): when the pointer list of the node v is empty ($v.L = \emptyset$) and there are two neighbors (say, u and w) that may form a triangle with v and their lists are empty, then node v invites/points the two neighbors u, w by executing the Rule $[I]$.
- The withdrawal rule ($[W]$): when v points at two nodes to form a triangle and at least one of these two nodes points another triangle. In this situation,

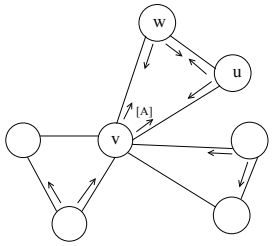


Figure 4.1: When v executes $[A]$

we say that v is chaining. Hence, the node v withdraws its invitation by executing the Rule $[W]$. $[W]$ is also executed when the pointer list does not induce a triangle in the graph G.

- The acceptation Rule ($[A]$): if the pointer list of the node v is empty and there is at least a node belonging to the same triangle $\{v, u, w\}$ which points it, then the node v accepts the invitation. We added another predicate $Max\_P_v(u, w)$ in the Rule $[A]$ imposing to a node for belonging to a triangle as quickly as possible in order to achieve it. For example, in Figure 4.1, the node v accepts to belong to the triangle $\{v, u, w\}$ instead of other triangles because the triangle $\{v, u, w\}$ contains already two confirmed nodes (u and w).

In addition to the two variables $v.S$ and $v.L$, the algorithm SMPT^c needs four predicates.

The first predicate, called $triangle_v(u, w)$, means that in perspective of v, the set $\{v, u, w\}$ induces a triangle in the graph G. The second predicate, $Pointed_v(u, w)$ means that it exists at least one node u or w which points at v and the second remained node and $triangle_v(u, w)$ is true. The third predicate, $Chain_v(u, w)$, means that the node v points at two nodes u, w that do not form a triangle or one of these nodes points at another triangle. The last predicate, $Max\_P_v(u, w)$, means that the node v is pointed by the nodes u, w that form a triangle and there is no adjacent triangle that contains more pointers than $\{v, u, w\}$. Formally, the predicates are defined as follows:

- $triangle_v(u, w) \equiv \{v, u, w\} \subseteq u.S \cap w.S$ and $|\{v, u, w\}| = 3$.
- $Pointed_v(u, w) \equiv (u.L = \{v, w\} \vee w.L = \{v, u\}) \wedge triangle_v(u, w)$.
- $Chain_v(u, w) \equiv (v.L = \{u, w\} \ \wedge (\neg triangle_v(u, w) \vee$
  $|\{v\} \cup v.L \cup u.L \cup w.L| > 3))$.
- $Max\_P_v(u, w) \equiv (Pointed_v(u, w) \wedge |u.L \cup w.L| \leq 3) \wedge$
  $(\nexists u_1, u_2 \in N(v) : Pointed_v(u_1, u_2) \wedge |u.L \cup w.L| < |u_1.L \cup u_2.L| \leq 3 \wedge (u, w) \neq (u_1, u_2))$.

The proposed algorithm is composed of four rules that are mutually exclusive. Observe also that the Rule $[U]$ is the priority rule. The details of SMPT^c is presented in Algorithm 2.

Consider G as a chain of three adjacent triangles. In starting configuration, $v.L = \emptyset$ for any $v \in V$, as illustrated in Figure 4.2(a). An example of the execution

Algorithm 2: Self-stabilizing algorithm for MPT (SMPT^c)

Nodes: v is the current node

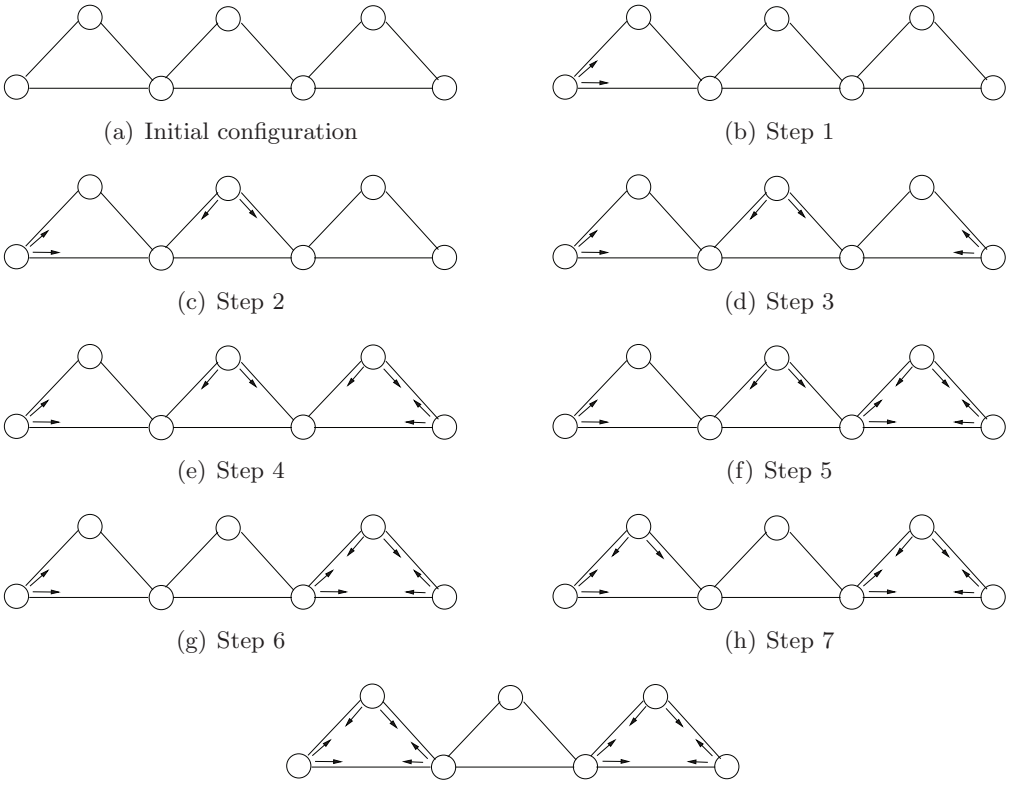
$v.S \neq N[v] \quad \longrightarrow v.S = N[v];$ [U]

$v.S = N[v] \wedge v.L = \emptyset \wedge (\forall u, w \in N(v) : \neg Pointed_v(u, w)) \wedge$
$(\exists u, w \in N(v) : u.L = w.L = \emptyset \wedge triangle_v(u, w)) \quad \longrightarrow v.L = \{u, w\};$ [I]

$v.S = N[v] \wedge v.L = \{u, w\} \wedge Chain_v(u, w) \quad \longrightarrow v.L = \emptyset;$ [W]

$v.S = N[v] \wedge v.L = \emptyset \wedge (\exists u, w \in N(v) : Max\_P_v(u, w)) \quad \longrightarrow v.L = \{u, w\};$ [A]

of the algorithm SMPT^c for such a graph with seven nodes in presented in Figure 4.2. Using the central daemon, the proposed algorithm finds two triangles and one single node after eight steps. The arrows show the list of pointers $v.L$ for each node v in G. When there is no arrows at a node v, means that its list is empty ($v.L = \emptyset$).



(a) Initial configuration (b) Step 1

(c) Step 2 (d) Step 3

(e) Step 4 (f) Step 5

(g) Step 6 (h) Step 7

(i) Step 8 (Final configuration)

Figure 4.2: Example of executing SMPT^c.

4.3 Correctness proof

In this section, we prove the correctness of SMPT^c, *i.e.* in final configuration, each node v having $|v.L| = 2$ forms a triangle with nodes $v.L$. Moreover, the set of all such triangles gives a maximal partitioning into triangles of the graph G. Therefore, Lemmas 4.3.1 to 4.3.4 prove that in final configurations any node v has a correct $v.L$ and the set $\{v\} \cup v.L$ form an independent triangle. Moreover, we also prove in Lemma 4.3.5 that the final configuration always provides a maximal partitioning into independent triangles.

Lemma 4.3.1 *In a configuration with no enabled node, each node v has a correct value of $v.S$.*

Proof. Assuming that in a configuration with no enabled node, there exists a node v has an incorrect value of $v.S$, means that $v.S \neq N[v]$, then the node v is enabled by the Rule $[U]$. □

Lemma 4.3.2 *In a configuration with no enabled node, if a node v has $v.L = \{u, w\}$ then $\{v, u, w\}$ forms a triangle in the graph G.*

Proof. Suppose that in a configuration with no enabled node, $\exists v \in V$ such that $v.L = \{u, w\}$ and $\{v, u, w\}$ is not a triangle in the graph G. In this case, we have $\neg triangle_v(u, w)$ and by Lemma 4.3.1, $v.S$ and $u.S$ and $w.S$ are correct, so the Rule $[W]$ is enabled for the node v. □

Lemma 4.3.3 *In a configuration with no enabled node, if a node v has $v.L = \{u, w\}$ then $u.L = \{v, w\} \wedge w.L = \{v, u\}$.*

Proof. Suppose that in a configuration with no enabled node, there is a node v having $v.L = \{u, w\}$ such that $u.L \neq \{v, w\}$ or $w.L \neq \{v, u\}$.

Since the reasoning is symmetric for u and w, without loss of generality, we assume that $v.L = \{u, w\} \wedge u.L \neq \{v, w\}$. Since $u.L$ is valid, $u.L \neq \{v, w\}$ implies four possible cases (1) $u.L = \emptyset$, (2) $u.L = \{v, x\}$, (3) $u.L = \{x, y\}$, (4) $u.L = \{w, x\}$.

- **Case 1:** $v.L = \{u, w\} \wedge u.L = \emptyset$.
  By Lemmas 4.3.1 and 4.3.2, $v.L = \{u, w\}$ and $\{v, u, w\}$ forms a triangle and $v.S$, $u.S$, $w.S$ are correct. We have two cases for the node w. If $w.L = \emptyset$ then the node u will be enabled by the Rule $[A]$. In the second case, $w.L \neq \emptyset$, we will have two situations: (i) if the node w points a triangle other than $\{v, u, w\}$. This means that $|\{v\} \cup v.L \cup u.L \cup w.L| > 3$. Then the node v will be enabled by $[W]$ to withdraw the invitation. (ii) if the node w points the same triangle $\{v, u, w\}$ then the node u will be enabled by the Rule $[A]$.

- **Case 2:** $v.L = \{u, w\} \wedge u.L = \{v, x\}$ such that $x \neq w$.
  By Lemma 4.3.2, if $v.L = \{u, w\} \wedge u.L = \{v, x\}$ then $\{v, u, w\}$ and $\{v, u, x\}$ form two adjacent triangles, with common edge (v, u), and by Lemma 4.3.1

$v.S$, $u.S$, $w.S$, $x.S$ are correct. This implies $|\{v\} \cup v.L \cup u.L \cup w.L| > 3$ and $|\{v\} \cup v.L \cup u.L \cup x.L| > 3$, so, at least, the two nodes v and u are enabled by the Rule $[W]$.

- **Case 3:** $v.L = \{u, w\} \wedge u.L = \{x, y\}$ such that $|\{v, u, w, x, y\}| = 5$.
  By Lemma 4.3.2, if $v.L = \{u, w\} \wedge u.L = \{x, y\}$ then $\{v, u, w\}$ and $\{u, x, y\}$ form two adjacent triangles, with common node u, and by Lemma 4.3.1 $v.S$, $u.S$, $w.S$, $x.S$, $y.S$ are correct. This implies $|\{v\} \cup v.L \cup u.L \cup w.L| > 3$, so, at least, the node v is enabled by the Rule $[W]$.

- **Case 4:** $v.L = \{u, w\} \wedge u.L = \{w, x\}$ such that $|\{v, u, w, x\}| = 4$. The proof is similar to that of case 2, but with considering the common edge to be (u, w).

□

Lemma 4.3.4 *In a configuration with no enabled node, if a node v has $v.L = \{u, w\}$ then $\{v, u, w\}$ forms an independent triangle.*

Proof. By Lemma 4.3.2, if $v.L = \{u, w\}$ then $\{v, u, w\}$ forms a triangle in G and by Lemma 4.3.3, if $v.L = \{u, w\}$ then $u.L = \{v, w\}$ and $w.L = \{v, u\}$, so, each node in the graph can belong to at most one triangle. □

Lemma 4.3.5 *In a configuration with no enabled node, each node v with $v.L \neq \emptyset$ forms a triangle with $v.L$. Moreover, the set of all such triangles is an MPT of the graph G.*

Proof. By Lemmas 4.3.2 to 4.3.4, in a configuration with no enabled node, any node $v \in V$ is either belonging to an independent triangle, *i.e* $v.L = \{u, w\}$ or is a single node *i.e* $v.L = \emptyset$. Suppose that the partitioning given by SMPT^c is not maximal. Then there exist three nodes v, u, w such that $v.L = u.L = w.L = \emptyset$ and $triangle_v(u, w)$. All the nodes are enabled by the Rule $[I]$. Contradiction. □

Theorem 4.3.6 *In a configuration with no enabled node, the algorithm SMPT^c gives a maximal graph partitioning into triangles and each node v having $v.L \neq \emptyset$, belongs to the triangle defined by $\{v\} \cup v.L$. Moreover, any remaining node v has $v.L = \emptyset$.*

Proof. This theorem is a direct consequence of Lemmas 4.3.1 to 4.3.5. □

4.4 Convergence proof

The convergence of SMPT^c is proved using the variant function technique. Note that finding a variant function was not trivial (See Section 2.3.6).

In any configuration of the system, the node v could be in exactly one of the following states : (see Figure 4.3)

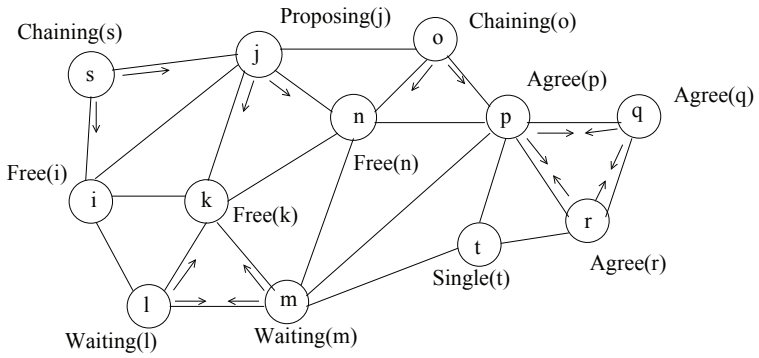


Figure 4.3: States of nodes

- Agree$(v) \equiv v.L = \{u, w\} \wedge u.L = \{v, w\} \wedge w.L = \{v, u\} \wedge triangle_v(u, w)$.
- Single$(v) \equiv v.L = \emptyset \wedge (\forall u, w \in N(v) : triangle_v(u, w) \Rightarrow Agree(u) \vee Agree(w))$.
- Waiting$(v) \equiv v.L = \{u, w\} \wedge u.L = \{v, w\} \wedge w.L = \emptyset$.
- Free$(v) \equiv v.L = \emptyset \wedge (\exists u, w \in N(v) : triangle_v(u, w) \wedge \neg Agree(u) \wedge \neg Agree(w))$.
- Chaining$(v) \equiv v.L = \{u, w\} \wedge (|\{v\} \cup v.L \cup u.L \cup w.L| > 3) \vee \neg triangle_v(u, w))$.
- Proposing$(v) \equiv v.L = \{u, w\} \wedge (u, w \in N(v) : u.L = w.L = \emptyset \wedge triangle_v(u, w))$.

We define also another predicate, called $Correct\_S$(v), which means that $v.S = N[v]$. We note that each node v can be in one of the following states: Agree(v), Single(v), Waiting(v), Free(v), Chaining(v) or Proposing(v) and has $Correct\_v.S$ or $\neg Correct\_v.S$.

Lemma 4.4.1 *Each node v can execute the Rule $[U]$ at most once.*

Proof. The predicate of the Rule $[U]$ depends only on the variable $v.S$ for each node v, and assuming that the closed neighborhood of each node v in the system does not change during the stabilization, and since this rule is mutually exclusive with all other rules, then this rule can be executed at most once at the starting of the system. □

Lemma 4.4.2 *If a node v Agree(v) or Single(v) states then the node v will never change its state.*

Proof. If Agree(v) $\Rightarrow v.L = \{u, w\} \wedge u.L = \{v, w\} \wedge w.L = \{v, u\}$, this implies $|\{v\} \cup v.L \cup u.L \cup w.L| = 3$. By assumption $v.L$, $u.L$ and $w.L$ are valid and $triangle_v(u, w)$, this implies $triangle_u(v, w)$ and $triangle_w(v, u)$. In this case, the three rules ($[I]$,$[W]$,$[A]$) will not enabled for neither v nor u nor w. The only rule that can be enabled is $[U]$ for updating the closed neighborhood for each node when $\neg Correct\_S$(v). Suppose that the node v is in the state Single(v), means that $v.L = \emptyset$ and $\forall u, w \in N(v) : triangle_v(u, w) \Rightarrow Agree(u) \vee Agree(w)$ then v has no available pair of nodes that can form a triangle with it. Since u or w is in state

agree and thus can never change its state, so the node v is disabled and it can never execute any rule ([I],[W],[A]) and can execute only the rule [U] without changing its state. □

Lemma 4.4.3 *If there exists a node v such that $Proposing(v) \vee Waiting(v) \vee Chaining(v) \vee Free(v)$ then there exists at least one enabled node in the system.*

Proof. We prove that in each state, we have at least one enabled node:

1. Proposing(v) means that $v.L = \{u, w\}$ and $u.L = w.L = \emptyset$, then at least the nodes u and w are enabled by the Rule [A].

2. Waiting(v) means that $v.L = \{u, w\}$ and $u.L = \{v, w\}$ and $w.L = \emptyset$, then the node w is enabled by the Rule [A].

3. Chaining(v) means that $v.L = \{u, w\}$ and $\neg triangle_v(u, w)$, then the node v is enabled by [W].

4. Free (v) means that $v.L = \emptyset$ and $(\exists u, w \in N(v) : triangle_v(u, w)$ and $\neg Agree(u) \wedge \neg Agree(w)$. If Free($u$) and Free($w$) then the node v is enabled by [I]. Else, the nodes u and w could be in Proposing, Waiting or Chaining states, and we proved previously that in each state, there is at least one enabled node.

We proved that if there exists a node v in state (Proposing, Waiting, Chaining and Free), then there is at least one enabled node. So, if there exists a node in one of these States, then the configuration of the system is not legitimate. □

Theorem 4.4.4 *The algorithm $SMPT^c$ converges in finite time.*

Proof. We define A, S, W, F, P, C and R as total number of *Agree*, *Single*, *Waiting*, *Free*, *Proposing*, *Chaining* and *Correct_S* nodes, respectively, in a configuration c of the system.

We use the variant function method to prove the convergence of the algorithm SMPT^c. For this, we define the function VF(c) which returns a vector (R, A+S, W, P, F, C). We define lexicographical order between these vectors, for example (3,2,1,4,4,1) is greater than (3,2,1,3,5,1).

Note that every configuration c for which VF(c)=(n,n,0,0,0,0) is a legitimate configuration and once the system reaches a legitimate configuration, no node will move. Hence, by Lemma 4.4.3, in every illegitimate configuration, there exists at least one node that can make a move.

Thus, in the following, we show that every rule increases the value of our function VF:

1. **Update Rule** [U]
   If the node v executes the Rule [U] then the number R increases by one. So, the function is increasing after execution of [U].

2. **Invitation Rule** $[I]$
   If the node v executes the Rule $[I]$ then the node v is not pointed by any neighbor with whom v could form a triangle and $v.L = \emptyset$ (*i.e.* v is a free node) and $Correct\_v.S$. So, after the execution of the Rule $[I]$, the number of proposing nodes (P) increases by one and the number of free nodes (F) decreases by 1.

3. **Withdrawal Rule** $[W]$
   Recall that only chaining nodes are enabled by $[W]$. We have three cases for enabling $[W]$:

   (a) **Case 1:** when v is not pointed by another neighbor with whom v could form a triangle.
   In this case, when the node v executes $[W]$, the number of chaining nodes (C) decreases by 1 and the number of free (F) or single nodes (S) increases by 1. Note, that the node v becomes a single node if all triangles to which it can belong are not available anymore (Formally, $\nexists u, w \in N(v) : triangle_v(u, w) \wedge \neg Agree(u) \wedge \neg Agree(w)$).

   (b) **Case 2:** when v is pointed by another neighbor with whom v could form a triangle.
   In such configuration, since v is pointed and $v.L \neq \emptyset$, and all nodes pointing at v are chaining. Let x be the number of these nodes. Since v is enabled by $[W]$, means that v is also chaining, then, we have $x + 1$ chaining nodes in the closed neighborhood of v. Once v executes $[W]$, a node that was pointing at v will become either proposing or waiting node. Let y be the number of nodes that become waiting and let z be the number of nodes that become proposing. We have $x = y + z$. Hence, when v executes $[W]$, the number of free nodes increases by 1, the number of chaining nodes decreases by $x+1$, the number of proposing and waiting nodes increases, respectively, by y and z.

   (c) **Case 3:** when v points at two neighbors that not form a triangle. Formally, $v.L = \{u, w\} \wedge \neg triangle_v(u, w)$.
   In this case, when the node executes $[W]$, the number of chaining nodes decreases by 1 and the number of free or single node increases by 1.

4. **Acceptation Rule** $[A]$
   If a node v is enabled by $[A]$ then we have two cases:

   (a) **Case 1:** when the node v is pointed by at least 2 waiting nodes u, w which belong to the same triangle. (see Figure 4.4(a))
   In this case, when the node v executes the $[A]$, the number of agree nodes increases by 3, the number of free nodes and waiting nodes decreases, respectively, by 1 and 2. Note, that in this case, we can have other proposing or waiting nodes pointing to the node v that will be chaining after the move of node v. Even in those situations the VF increases.

(b) **Case 2:** when the node v is pointed by x proposing nodes.(see Figure 4.4(b))
We have x triangles to which v may belong. So, when it executes the Rule $[A]$, it will arbitrary choose one triangle among x. In this configuration, the number of proposing nodes reduces by x, the number of free node reduces by 1, and the number of chaining and waiting nodes increase, respectively, by $x-1$ and 2.

(a) Node v is pointed by at least 2 waiting nodes.

(b) Node v is pointed by x proposing nodes.

Figure 4.4: Case when v is enabled by $[A]$.

We conclude that each of these rules increments the value of the function VF. The number of executions is bounded by the number of all possible vector values. So, using Lemma 4.4.1 and the fact that $A + S = n$ and $R = n$, such that n is the number of the nodes in the graph, the system reaches a safe configuration when no increment is possible (i.e VF $(c) = (n, n, 0, 0, 0, 0)$). Hence, by Lemmas 4.4.1, 4.4.2 and 4.4.3 and since each rule only increases the function, the system reaches a legitimate configuration in finite moves. □

Note that the complexity of SMPT^c is bounded by $O(n^4)$ moves and we will improve this result in the following section.

4.5 Complexity analysis

In this section, we compute the maximum number of rule execution of SMPT^c.

First, consider the Rule $[U]$. We proved in Lemma 4.4.1, that this rule can be executed at most once for each node v in the graph G.

We denote Nb_invit_v, Nb_accept_v and Nb_with_v the number of invitations, acceptations and withdrawals moves respectively for a node v. Note that $Nb\_with_v \leqslant Nb\_invit_v + Nb\_accept_v + 1$.

Lemma 4.5.1 *If any node v executes $[A]$ and it is pointed by waiting nodes then the next state of v will be Agree.*

Proof. When the node v is pointed by at least two waiting nodes (recall that waiting nodes are present only in pairs in the graph), see Figure 4.4(a): Even if there exists a proposing node p that point at v for forming triangle $\{v, p, p'\}$, the node v chooses the waiting nodes because ($Max\_P_v(p, p')$ is false). So, assume that the node v choose arbitrary two waiting nodes, called u, w such that $u.L = \{v, w\}$ and $w.L = \{v, u\}$. In this case, when the node v executes $[A]$ for forming triangle $\{v, u, w\}$, the node v will have $v.L = \{u, w\}$. So, $v.L = \{u, w\} \wedge u.L = \{v, w\} \wedge w.L = \{v, u\}$ and by assumption $v.L$ and $u.L$ and $w.L$ are valid, then the nodes v and u and w become agree nodes. □

Lemma 4.5.2 *If a node v executes $[A]$ then the next state of v will be either Waiting or Agree.*

Proof. When the node v is enabled for executing the Rule $[A]$ then $v.L = \emptyset$ and $v.S = N[v]$ and there exist two neighbors u, w of v such that $Max\_P_v(u, w)$ is true. In this situation, we can have two cases:

- The node v is pointed by only proposing nodes (see Figure 4.4(b)): Assume that the node v chooses arbitrary two neighbors, one proposing node, called u such $u.L = \{v, w\}$ and one free node called w ($w.L = \emptyset$). In this case, when the node v executes $[A]$ for accepting the invitation of u, it will have $v.L = \{u, w\}$. So, $v.L = \{u, k\}$ and $u.L = \{v, w\} \wedge w.L = \emptyset$ and by assumption $v.L$ and $u.L$ are valid, then the nodes v and u become waiting nodes.
- The node v is pointed by at least two waiting nodes: Using Lemma 4.5.1, the next state of v will be agree.
  Thus, we proved that if a node executes an acceptation Rule $[A]$, then the next state will be either waiting or agree.

□

Lemma 4.5.3 *A node v can execute at most $d(v)$ times acceptation rule $[A]$ where $d(v)$ is the degree of v in the graph G. In other words, $Nb\_accept_v \leqslant d(v)$.*

Proof. We proved in Lemma 4.5.2 that if a node executes $[A]$ then it will be either agree or waiting. Furthermore, if a node v is pointed by waiting nodes and v executes $[A]$, then the next state must be agree (Lemma 4.5.1). Thus, assume that v is enabled for executing the rule $[A]$.

If the node v is pointed by waiting nodes, then by Lemma 4.5.1 the node v will be agree. Using Lemma 4.4.2, the node v will never change its state. So in this case, the node v executes $[A]$ only once.
If the node v has only proposing nodes that point at v then the worst case is $d(v)$ proposing nodes. Assume that u is one from these proposing nodes. In this situation the node u has $u.L = \{v, w\}$ and $w.L = \emptyset$, so if the node v accepts to belong the triangle $\{v, u, w\}$ by executing $[A]$ then the nodes v and u become waiting nodes and by using Lemma 4.5.1, the next state of w will be agree. Hence, the number of execution of $[A]$ for the node v is at most its degree $d(v)$ times. □

Lemma 4.5.4 *A node v can execute $O(\Delta d(v))$ times invitation rule $[I]$ where Δ is the maximum node degree in the graph G. In other words, $Nb\_invit_v \in O(\Delta d(v))$.*

Proof. Assume that the node v tries to invite the two neighbors u, w by executing the rule $[I]$. The node v executes $[I]$ if $v.L = u.L = w.L = \emptyset$ and $Correct\_v.S$ and $triangle_v(u, w)$. Recall that $triangle_v(u, w) :: \{v, u, w\} \subseteq u.S \cap w.S$ and $|\{v, u, w\}| = 3$. At the starting of the system the values of $u.S$ and $w.S$ can be incorrect and the nodes u and w have not yet executed any rule. Recall also, according to Lemma 4.4.1, the rule $[U]$ can be executed at most once for any node.

Assuming that $u.S$ and $w.S$ are correct. In this situation, when the node v executes $[I]$ for forming the triangle $\{v, u, w\}$, the u and w will be pointed and the only rule can executed is acceptation rule $[A]$. Hence, by using Lemma 4.5.3, each node u or w can execute the acceptation rule at most Δ times where Δ is the maximum node degree in the graph G. So, the worst case, one of nodes u, w accepts another invitation for belonging to adjacent triangle of $\{v, u, w\}$ and the node v will be chaining and withdraws its invitation. Assuming that the node v will be chaining because its neighbor u accepts an adjacent triangle. Using Lemma 4.5.3 again, the node v can invite at most $d(u)$ the same node u. So, for all its neighbors, the node v can make at most $d(v).\Delta$ invitations.
Assuming that $u.S$ and $w.S$ are incorrect. In this situation, the node v can make wrong invitation for inviting the nodes u, w for forming triangle while $\{v, u, w\}$ does not induce a triangle in G. So, in this situation, if the node v invites u, w then v cannot execute any rule, until one of these nodes corrects its value $u.S$ (resp. $w.S$). Thus, when u (resp. w) corrects its value $u.S$ (resp. $w.S$) by executing $[U]$, then node v will never invite again this couple of nodes. So, we can deduce that the maximum execution of $[I]$ for a node v is twice number of execution of $[I]$ when $u.S$ and $w.S$ are correct. Hence, we deduce that $Nb\_invit_v \leqslant 2d(v)\Delta$. □

Lemma 4.5.5 *A node v can execute at most $d(v)(\Delta + 1) + 1$ times the withdrawal rule $[W]$ where Δ is the maximum node degree in the graph G.*

Proof. Observe that $Nb\_with_v \leqslant Nb\_invit_v + Nb\_accept_v + 1$. Then, using Lemma 4.5.4, $Nb\_invit_v \leqslant d(v)\Delta$ and by using Lemma 4.5.3, $Nb\_accept_v \leqslant d(v)$, we obtain $d(v)(\Delta + 1) + 1$. □

Proposition 4.5.6 *The algorithm SMPT^c converges in $O(\Delta m)$ moves.*

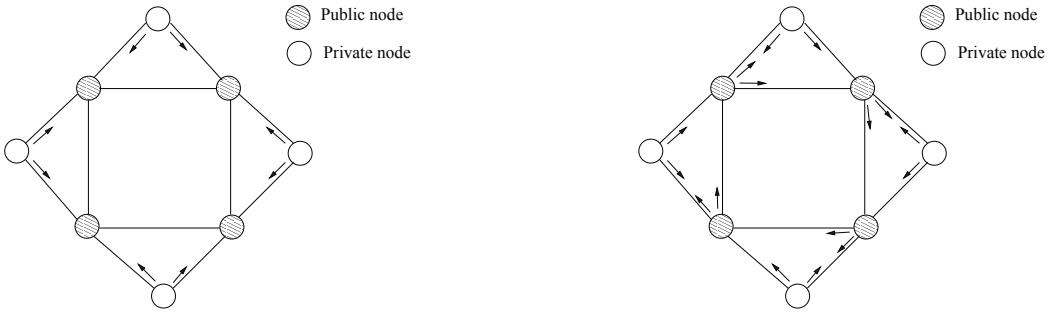
Proof. In addition to Nb_invit_v, Nb_accept_v, Nb_with_v, we also consider $Nb\_Update\_S_v$ which counts the number of execution of the rule $[U]$. By using Lemma 4.4.1, we have $Nb\_Update\_S_v \leqslant 1$. So, the maximum number of moves of the algorithm SMPT^c is $\sum_{v=1}^{n}(Nb\_Update\_S_v + Nb\_invit_v + Nb\_accept_v + Nb\_with_v + 1)$. Thus, by Lemmas 4.5.3, 4.5.4 and 4.5.5, we deduce that the maximum execution of the algorithm SMPT^c is $O(\Delta m)$.

The memory requirement of the algorithm SMPT^c amounts to $O(\Delta \log n)$ per node: Apart of the list of pointer L, a node has to store at most $(\Delta + 1)$ *ids* for S. Thus, each node uses only $O(\Delta \log n)$ memory space. □

Theorem 4.5.7 *SMPT^c is a self-stabilizing algorithm for maximal graph partitioning into disjoint triangles and converges in $O(\Delta m)$ moves under an unfair central daemon and using only $O(\Delta \log n)$ memory space.*

Proof. This theorem is a consequence of Theorem 4.4.4 and Proposition 4.5.6. □

We have to note that the proposed algorithm converges only under the central daemon. Given a graph G composed from a cycle of four adjacent triangles as illustrated in Figure 4.5(a). The nodes with degree 2 and 4 are called *private nodes* and *public nodes*. At initial configuration, each private node points at its two public nodes to form a triangle as presented in Figure 4.5(a). Hence, all public nodes have the same view, then, they have the same behaviour. Moreover, any public node is enabled by the acceptance rule $[A]$. So, if the distributed daemon selects all the public nodes to execute their moves ($[A]$) simultaneously, then all nodes will be chaining and will be enabled by the withdrawal rule $[W]$ (see Figure 4.5(b)). So, if the daemon selects again the same public nodes to execute their moves ($[W]$) simultaneously, then the system reaches the first configuration. Therefore, the system will oscillate between the two configurations and it will never converge to a legitimate (desired) configuration.



(a) Initial configuration (b) Second configuration

Figure 4.5: An infinite execution of SMPT^c under the distributed daemon

4.6 Summary

In this chapter, we proposed a first self-stabilizing algorithm for maximal graph partitioning into triangles (SMPT^c). The algorithm SMPT^c converges in $O(\Delta m)$ moves under the unfair central daemon. However, we show that the algorithm does not converge under the distributed daemon and a transformation is required in order to operate under the distributed daemon. For example, by using the *conflict managers* proposed in [GT07] (*cf.* Chapter 2), the transformed algorithm will stabilize in $O(\Delta^2 m)$ under the unfair distributed daemon.

At this point, it is worth looking at how to adress MPT problem under the distributed daemon without using any transformation. The next chapter deals the MPT problem under the distributed daemon.

Chapter 5

Algorithm for MPT under the Distributed Daemon

Contents

5.1 Introduction

In the previous chapter, we developed the basic algorithm (SMPT^c) for finding an MPT in arbitrary graph that converges in $O(\Delta m)$ moves under the central daemon. We showed how to prove the convergence of an algorithm using the variant function. Thus, we showed how it was difficult to prove an algorithm with such technique as it was the case of the majority of self-stabilizing algorithms. Nevertheless, the complexity $O(\Delta m)$ moves can be very important in large free scale network and the algorithm may not converge under the distributed daemon, then the exention of maximal matching algorithm of Hsu and Huang is not always suitable in this case. For this reason, in the this chapter we develop a new algorithm for computing a maximal partitioning into triangles, called SMPT^D. The new one outperforms previous algorithm SMPT^c on three points: (*i*) the assumption on the validity of pointer list is avoided, (*ii*) the algorithm SMPT^D operates under the unfair distributed daemon while the previous one supports the unfair central daemon only, (*iii*) the move complexity assuming the distributed daemon is considerably reduced from $O(\Delta^2 m)$ to $O(m)$ moves.

For justifying the use of identifiers *id*, we present the impossibility result for solving the maximal partitioning into triangles in anonymous networks.

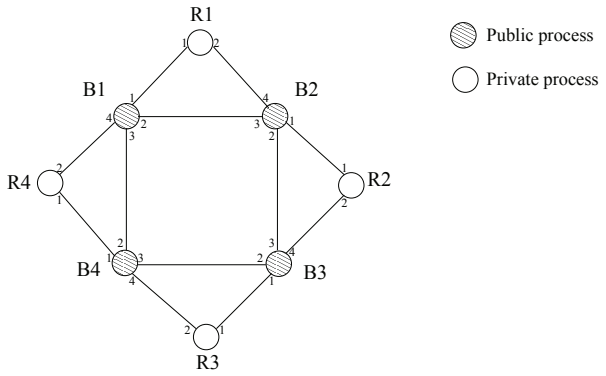


Figure 5.1: An example network

5.2 Impossibility result

In this section, we prove that it is impossible to solve the problem of maximal partitioning into triangles in anonymous networks by providing a counter-example using the synchronous daemon.

Lemma 5.2.1 *There is no deterministic self-stabilizing algorithm for maximal graph partitioning into triangles in anonymous networks under a distributed daemon.*

Proof. Suppose that there exists a self-stabilizing algorithm that computes a maximal graph partitioning into triangles. Apply this algorithm to a network that is composed of a cycle of four processes surrounded by four triangles as shown in Figure 5.1. Processes that belong to the cycle (*i.e.*, B1, B2, B3, B4) are called *public processes*. The remaining processes R1, R2, R3, R4 are called *private processes*. Each *public process* has two adjacent triangles while each *private process* has only one. Note that private (resp. public) processes have equal degree and equal view; and consequently, the same behavior.

Consider an execution under the synchronous daemon. Furthermore, regard a starting configuration where all public processes have the same state and all private processes have the same state. Under the synchronous daemon this property is preserved, *i.e.*, nodes in the same group always have the same state. Assume the algorithm stabilizes and computes a maximal graph partitioning into triangles. If a private process belongs to a triangle of the partitioning then all private processes do so. This is impossible. Hence, none of the private nodes belongs to a triangle of the partitioning. Thus, the algorithm produces an empty partitioning. Obviously this is not a maximal graph partitioning into triangles. Contradiction. □

5.3 Algorithm description

This section presents the second self-stabilizing algorithm for computing a maximal partitioning into triangles of an arbitrary graph within the distributed daemon,

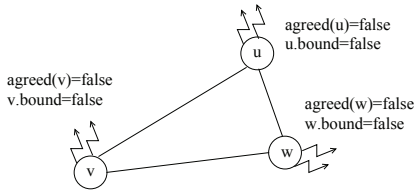
called SMPT^D.

Contrary to the algorithm SMPT^c where each node v maintains a pointer list $v.L$, in the second algorithm SMPT^D, each node v of the graph maintains two variables a and b pointing to different neighbors of v or to $\perp$. Eventually these variables form a pattern that leads to an MPT, *i.e.*, a node v with $v.a \neq \perp$ forms together with the nodes $v.a$ and $v.b$ a triangle in the graph. For technical reasons, the identifier of $v.a$ must always be smaller than that of $v.b$. In order to verify for a node v that it forms a triangle with the nodes $v.a$ and $v.b$, node v needs to known the neighbors of $v.a$ and $v.b$. This is made possible by a variable S through which each node exposes its closed neighborhood. The algorithm is prepared to handle transient faults of this variable.

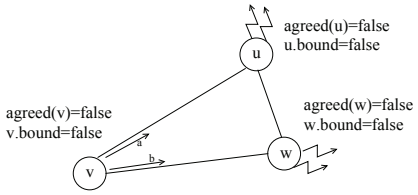
A node that is not already participating in a triangle selects two of its neighbors as candidates for building a triangle. In order not to select neighbors that are already part of a triangle, each node has a Boolean variable *bound*. After joining a triangle, a node sets this variable to *true*. Thus, in selecting a neighbor, a node only considers neighbors with *bound* =*false*. Since the algorithm is supposed to work under the distributed daemon, a mechanism for symmetry breaking is needed (*cf.* Section 5.2). This is based on unique identifiers. A node that starts the formation of a triangle only invites nodes with larger identifiers. That means, a node v selects among its neighbors with larger identifiers two nodes u and w which do not already participate in a triangle (*i.e.*, $u.bound = w.bound = false$), that form a triangle in the graph (based on the information $u.S$ and $w.S$), and which have not already selected neighbors (*i.e.*, $u.a = u.b = w.a = w.b = \perp$). If such neighbors exist then $v.a$ and $v.b$ are updated to u resp. w with $u < w$. The next step will be that the invited node with the smaller identifier (i.e., u) either accepts or denies this invitation. In the first case this node will modify its variables a and b accordingly. Finally, the node with the larger identifier (*i.e.*, w) will also accept the invitation completing the triangle. If u or w decide against accepting v's invitation and choose to accept another invitation or make an invitation themselves then node v resets its variables a and b to $\perp$ and is ready to make or accept another invitation.

After a triangle has been formed, the participating three nodes will updated their variable *bound*. The nodes do this according to their identifiers, beginning with the node having the smallest identifier. The adherence to this order is the cornerstone for proving that the algorithm SMPT^D stabilizes after $O(m)$ moves. Figure 5.2 shows the six steps required to form a triangle. There are also rules that reset the state of a node not fitting into this sequence. This may be due to an incorrect initial configuration or due to concurrent moves of neighboring nodes.

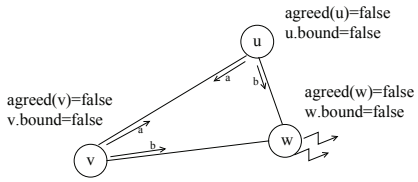
In order to have a concise formulation of the algorithm SMPT^D, a few predicates are introduced (see Figure 5.3 for a formal definition). Predicate $pseudoTriangle(v, u, w)$ is true, if the three nodes form a triangle in the graph from the perspective of node v. Note that node v does not have direct access to $N(u)$ or $N(w)$ but relies on $u.S$ and $w.S$ which might be incorrect. Thus, the validity of $pseudoTriangle(v, u, w)$ does not necessarily imply that v, u and w form a triangle in the graph (nor vice versa).



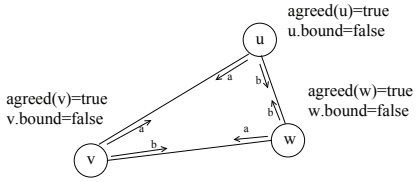
(a) Initial configuration



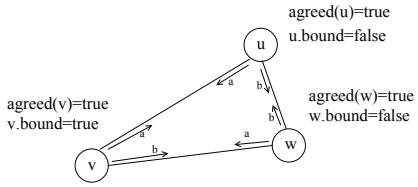
(b) Step 1: v executes `[I1]`



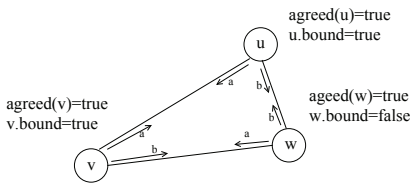
(c) Step 2: u executes `[A1]`



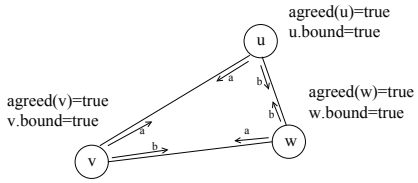
(d) Step 3: w executes `[A2]`



(e) Step 4: v executes `[C2]`



(f) Step 5: u executes `[C2]`



(g) Step 6: w executes `[C2]`

Figure 5.2: A Simple example of an execution of SMPT^D ($v < u < w$) for one triangle

Predicate *semiTriangle*(v, u, w) is true if $u.a$ (resp. $u.b$) points to the node of $\{v, w\}$ with the smaller (resp. larger) identifier. Thus, if *pseudoTriangle*$(v, v.a, v.b)$ and *pseudoTriangle*$(v, v.b, v.a)$ are both true, then $\{v, v.a, v.b\}$ form a triangle using the definition above. Furthermore, we note that *semiTriangle*(v, u, w) is symmetric in v and w, *i.e.*, *semiTriangle*(v, u, w) is true if and only if *semiTriangle*(w, u, v) is true. This is captured by the predicate *agreed*(v). When a node v satisfies *agreed*(v) then the nodes $v.a$ and $v.b$ also satisfy this predicate when their variable S are correct. At that point, the three nodes can begin to set their variable *bound* to *true* to signal that they do not accept further invitations. To enforce that this step succeeds from smallest to largest identifier the predicate *coherent* is introduced. This predicate permits to make a serialization for updating the variable *bound* in one triangle, starting from the node with smaller identifier to the larger one.

$$
\begin{aligned}
\mathit{passive}(v) &\equiv v.a = \bot \ \wedge \ v.b = \bot \ \wedge \ \neg v.\mathit{bound} \\
\mathit{pseudoTriangle}(v,u,w) &\equiv u,w \in N(v) \wedge \{v,u,w\} \subseteq v.S \cap u.S \cap w.S \ \wedge \ |\{v,u,w\}| = 3 \\
\mathit{free}(v,u,w) &\equiv v < u < w \ \wedge \ \mathit{pseudoTriangle}(v,u,w) \ \wedge \ \mathit{passive}(u) \ \wedge \ \mathit{passive}(w) \\
\mathit{semiTriangle}(v,u,w) &\equiv u.a = \min(v,w) \wedge u.b = \max(v,w) \\
\mathit{agreed}(v) &\equiv \mathit{pseudoTriangle}(v, v.a, v.b) \wedge v.a < v.b \wedge \mathit{semiTriangle}(v, v.a, v.b) \wedge \\
&\quad\ \mathit{semiTriangle}(v, v.b, v.a) \\
\mathit{coherent}(v) &\equiv \left\{ \begin{array}{l} \mathit{agreed}(v) \ \wedge \\ [(v < v.a < v.b \ \wedge \ ((\neg v.a.\mathit{bound} \wedge \neg v.b.\mathit{bound}) \vee (v.\mathit{bound} \wedge v.a.\mathit{bound})))\vee \\ (v.a < v < v.b \ \wedge \ ((v.a.\mathit{bound} \wedge \neg v.b.\mathit{bound}) \vee (v.a.\mathit{bound} \wedge v.\mathit{bound})))\vee \\ (v.a < v.b < v \ \wedge \ (v.a.\mathit{bound} \wedge v.b.\mathit{bound}))] \end{array} \right.
\end{aligned}
$$

Figure 5.3: Predicates of the algorithm SMPT^D

The complete set of rules of algorithm SMPT^D is shown in Algorithm 3. The nine rules can be categorized in three groups. Rules of the first group (`[C1]`,`[C2]`) keep the variables S and *bound* up to date. The rules `[A1]`, `[A2]`, and `[I1]` are responsible for creating and accepting invitations. They all require that both variables a and b have the value $\bot$. The actions of these three rules set the variables $v.a$ and $v.b$ such that $v.a < v.b$ holds and such that $\mathit{pseudoTriangle}(v, v.a, v.b)$ is true. If a node declines an invitation, the inviting node does not immediately make a new invitation, instead it first resets its variables. This task is accomplished by the rules of the third group (`[W1]`,`[W2]`,`[W3]`,`[W4]`). The duty of these rules is also to reset a node if an inconsistent state is detected (*i.e.*, $v.b \leq v.a$). Figure 5.2 shows a sequence of rule executions for the stabilization of one triangle. A second example is presented in Figure 5.4 showing an execution of SMPT^D from starting configuration having incorrect $i.S$ for $i \in [1,5]$. In perspective of node 1, this node has six triangles. However, only one triangle exists ($\{1,2,3\}$) and the other triangles are virtual (dashed lines).

5.4 Correctness proof

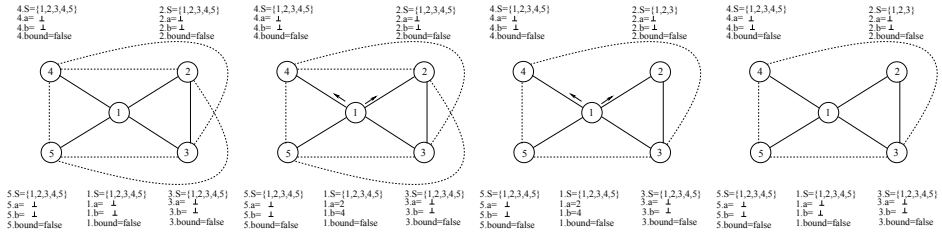
This section proves that in configurations with no enabled nodes, the variables a and b of all nodes induce a maximal partitioning into triangle (MPT).

Lemma 5.4.1 *Let $v \in V$ such that $v.a \neq \bot, v.b \neq \bot$, and $v.a < v.b$. Then, $\mathit{semiTriangle}(v.a, v, v.b) = \mathit{semiTriangle}(v.b, v, v.a) = \mathit{true}$.*

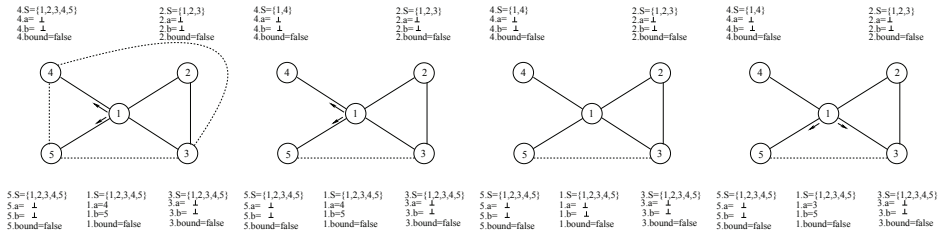
Lemma 5.4.2 *In a configuration with no enabled node, the following properties hold for each $v \in V$.*

(a) $v.S = N[v]$.

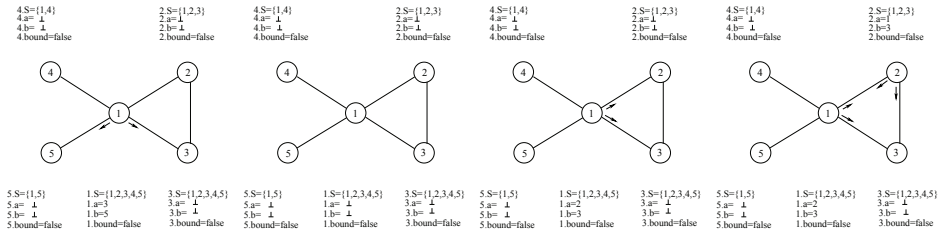
(b) If $v.a \neq \bot$ or $v.b \neq \bot$ then $v.a \neq \bot$, $v.b \neq \bot$, $v.a < v.b$, and $\{v, v.a, v.b\}$ form a triangle.



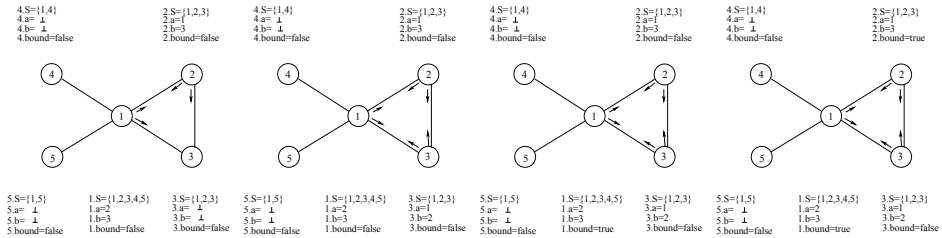
(a) Initial config. (b) 1 executes I1 (c) 2 executes C1 (d) 1 executes W2



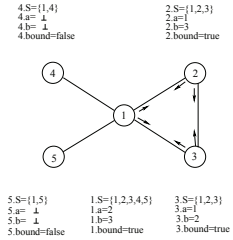
(e) 1 executes I1 (f) 4 executes C1 (g) 1 executes W2 (h) 1 executes I1



(i) 5 executes C1 (j) 1 executes W2 (k) 1 executes I1 (l) 2 executes A1



(m) 3 executes C1 (n) 3 executes A2 (o) 1 executes C2 (p) 2 executes C2



(q) 3 executes C2
Final configuration

Figure 5.4: Example of an execution of SMPT^D

Algorithm 3: Self-stabilizing algorithm for MPT (SMPT^D)

Nodes: v is the current node

$v.S \neq N[v] \quad \longrightarrow v.S := N[v];$ [C1]

$v.bound \neq coherent(v) \quad \longrightarrow v.bound := coherent(v);$ [C2]

if $v.a = \bot \wedge v.b = \bot$ **then**

$\exists(u,w) \in \{u, w \in N(v) \mid u < v < w \wedge semiTriangle(v,u,w) \wedge passive(w) \wedge \neg u.bound \wedge pseudoTriangle(v,u,w)\} \quad \longrightarrow v.a := u; v.b := u.b;$ [A1]

$\exists(u,w) \in \{u, w \in N(v) \mid u < w < v \wedge semiTriangle(v,u,w) \wedge semiTriangle(v,w,u) \wedge \neg u.bound \wedge \neg w.bound \wedge pseudoTriangle(v,u,w)\} \quad \longrightarrow v.a := u; v.b := w;$ [A2]

$\exists(u,w) \in \{u, w \in N(v) \mid free(v,u,w)\} \quad \longrightarrow v.a := u; v.b := w;$ [I1]

else

$v.a = \bot \vee v.b = \bot \vee v.a \geq v.b \quad \longrightarrow v.a := \bot; v.b := \bot;$ [W1]

$\neg pseudoTriangle(v, v.a, v.b) \quad \longrightarrow v.a := \bot; v.b := \bot;$ [W2]

$(v.a.bound \wedge \neg semiTriangle(v, v.a, v.b)) \vee (v.b.bound \wedge \neg semiTriangle(v, v.b, v.a)) \quad \longrightarrow v.a := \bot; v.b := \bot;$ [W3]

$(v.a < v \wedge \neg semiTriangle(v, v.a, v.b)) \vee (v.b < v \wedge \neg semiTriangle(v, v.b, v.a)) \quad \longrightarrow v.a := \bot; v.b := \bot;$ [W4]

(c) If $v.a \neq \bot$ and there exists no node w such that $w > v$ and $w.a \neq \bot$ then $agreed(v)$ is true.

(d) The validity of $agreed(v)$ implies $agreed(v.a)$, $agreed(v.b)$, and $v.bound = true$.

Proof. Property (a) is true since rule [C1] is disabled for all nodes. The first part of Property (b) is true because rule [W1] is disabled. Furthermore, $pseudoTriangle(v, v.a, v.b)$ is true since rule [W2] is disabled. Together with Property (a) this implies $\{v, v.a, v.b\} \subseteq N[v] \cap N[v.a] \cap N[v.b]$. Hence, the set $\{v, v.a, v.b\}$ forms a triangle in the graph.

Let v be the node with the largest identifier such that $v.a \neq \bot$. Then Property (b) implies that $v.a < v.b$ and that $\{v, v.a, v.b\}$ form a triangle. Because of Property (a) this implies that $pseudoTriangle(v, v.b, v.a)$ holds for all permutations of the three nodes. By Lemma 5.4.1 $semiTriangle(v.a, v, v.b)$ is valid. Suppose $v < v.a$. The choice of v implies $v.a.a = v.a.b = v.b.a = v.b.b = \bot$. Then $agreed(v.a) = agreed(v.b) = false$ and thus $v.a.bound = v.b.bound = false$, since rule [C2] is disabled for $v.a$ and $v.b$. Hence, $agreed(v) = false$ and $passive(v.b) = true$. Since rule [A1] is disabled for node $v.a$ this implies $v.bound = true$. This leads to a contradiction since $agreed(v)$ is $false$ and rule [C2] is disabled for v. Hence, $v > v.a$. Then $semiTriangle(v, v.a, v.b)$ since v is disabled for rule [W4].

Suppose $v < v.b$. As above it follows $v.b.a = v.b.b = \bot$ and $v.b.bound = false$. Thus $v.bound = v.a.bound = false$. Furthermore, $semiTriangle(v.b, v.a, v)$ and $semiTriangle(v.b, v, v.a)$ are valid by Lemma 5.4.1. Since rule [A2] is disabled for node $v.b$ this is impossible. This yields $v.a < v.b < v$. Then $semiTriangle(v, v.a, v.b)$

and $\mathit{semiTriangle}(v, v.b, v.a)$ are valid because rule `[W4]` is disabled. Hence, $\mathit{agreed}(v)$ is *true*. This shows Property (c).

If $\mathit{agreed}(v)$ is *true* then $\mathit{semiTriangle}(v, v.a, v.b)$ and $\mathit{semiTriangle}(v, v.b, v.a)$ are valid. This implies that *semiTriangle* is true for all permutations of $v, v.a$, and $v.b$. This also holds for $\mathit{pseudoTriangle}(v, v.a, v.b)$ by Property (a). Thus, $\mathit{agreed}(v.a) = \mathit{agreed}(v.b) = \mathit{true}$ and $v.\mathit{bound} = \mathit{true}$ implying Property (d). □

Lemma 5.4.3 *In a configuration where no node is enabled each node v with $v.a \neq \perp$ forms a triangle with $v.a$ and $v.b$. Moreover, the set of all such triangles is an MPT of the graph G.*

Proof. The proof is by induction on n. Let $n \leq 2$, *i.e.*, G does not contain any triangle. Since $n \leq 2$ predicate $\mathit{pseudoTriangle}(v, u, w)$ is *false* for all $v, u, w \in V$. This yields $v.a = v.b = \perp$ for all $v \in V$ because rule `[W2]` is disabled.

So let $n \geq 3$. First consider the case that $v.a = \perp$ for all $v \in V$. Then $\mathit{free}(v, u, w) = \mathit{false}$ for all $v, u, w \in V$ because rule `[I1]` is disabled. Furthermore, $v.\mathit{bound} = \mathit{false}$ for all $v \in V$ because rule `[C2]` is disabled. This implies $\mathit{passive}(v) = \mathit{true}$ for all $v \in V$. Finally, $\mathit{pseudoTriangle}(v, u, w) = \mathit{false}$ for all $v, u, w \in V$ because $\mathit{free}(v, u, w) = \mathit{false}$. Hence, G does not contain a triangle.

Let v be the node with maximal identifier such that $v.a \neq \perp$. Lemma 5.4.2 (c) and (d) imply $\mathit{agreed}(v) = \mathit{agreed}(v.a) = \mathit{agreed}(v.b) = \mathit{true}$ and $v.\mathit{bound} = v.a.\mathit{bound} = v.b.\mathit{bound} = \mathit{true}$. Since rule `[W3]` is disabled for all $w \in V \setminus \{v, v.a, v.b\}$, we have $\{w.a, w.b\} \cap \{v, v.a, v.b\} = \emptyset$. This means that no node $w \in V \setminus \{v, v.a, v.b\}$ points at $v, v.a, v.b$. Let G' be the graph induced by $V \setminus \{v, v.a, v.b\}$. Since no node of G' is enabled the lemma holds for G' by induction. This implies that the lemma is also true for G. □

5.5 Convergence proof

This section presents the convergence proof of the algorithm SMPT^D by using the analysis of the local states and sequences technique (*cf.* Section 2.3.6). For this, the following lemmas are developed in order to bound the execution of each rule of the algorithm:

Lemma 5.5.1 *Each node makes at most one `[C1]` move. This is always the first move of a node.*

Proof. The precondition of rule `[C1]` only depends on the correctness of $v.S$. Since the neighborhood relation is static, this rule is executed at most once. □

Lemma 5.5.2 *Let $v \in V$ with $\mathit{agreed}(v) = \mathit{true}$. Then $v.a.S = N[v.a]$ or $v.b.S = N[v.b]$ implies $\mathit{agreed}(v.a) = \mathit{agreed}(v.b) = \mathit{true}$.*

Proof. Consider the case $v.a.S = N[v.a]$, the proof of the other case is similar. The assumption $\mathit{agreed}(v) = \mathit{true}$ implies $\mathit{pseudoTriangle}(v, v.a, v.b) = \mathit{true}$ and

$v.a < v.b$. This yields $v.b \in N(v.a)$ and hence the set $\{v, v.a, v.b\}$ forms a triangle in the graph. Clearly this implies $agreed(v.a) = agreed(v.b) = true$. □

Definition 8 *Let $v \in V$ with $agreed(v) = agreed(v.a) = agreed(v.b) = true$. Nodes $v, v.a, v.b$ are said to form to*

- *an ultimate constellation if $v.bound = v.a.bound = v.b.bound = true$ and*
- *to a penultimate constellation if $v.bound = v.a.bound = v.b.bound = false$ or $v.bound = true$, $v.a.bound = v.b.bound = false$ or $v.bound = v.a.bound = true, v.b.bound = false$.*

Steps 3, 4, and 5 of Figure 5.2 show the three penultimate constellations and Step 6 shows an ultimate constellation. Note that any node v, belonging to a penultimate constellation or an ultimate constellation, cannot execute any rule except `[C1]` and `[C2]`.

Lemma 5.5.3 *A node v belonging to an ultimate constellation makes at most one move, a `[C1]` move. A node belonging to a penultimate constellation makes at most two moves, one `[C1]` and one `[C2]` move.*

Proof. Obviously v can only be enabled by rules `[C1]` and `[C2]` without changing pointers. Moreover, v belongs to a constellation means that $agreed(v) = agreed(v.a) = agreed(v.b) = true$. Then, when v (resp. $v.a, v.b$) executes the rules `[C1]` and `[C2]`, the predicate $agreed(v)$ remains true. Since each node makes at most one `[C1]` move (Lemma 5.5.1), it remains to consider `[C2]` moves. Clearly v cannot make a `[C2]` move if it belongs to an ultimate constellation. A node belonging to a penultimate constellation can only make a `[C2]` move if $bound = false$ and the other nodes of the same penultimate constellation with lower identifier have $bound = true$. Thus, each such node can make at most one `[C2]` move. □

Lemma 5.5.4 *Let v be a node that has already executed `[C2]` to set $v.bound$ to $false$. The next time v is enabled by rule `[C2]` it is part of penultimate constellation.*

Proof. Since v is enabled for `[C2]` to set $v.bound$ to $true$, we have $v.bound = false$ and $coherent(v) = true$. This implies $agreed(v) = true$. Furthermore, v cannot be enabled for `[C1]`, thus $v.s = N[v]$. Assume that $agreed(v.a) = false$ or $agreed(v.b) = false$. Then Lemma 5.5.2 implies that nodes $v.a$ and $v.b$ are enabled by rule `[C1]`, *i.e.*, they have not made any move. This implies that the values of $v.a.a$, $v.a.b$, and $v.a.bound$ (resp. of $v.b.a$, $v.b.b$, and $v.b.bound$) have not changed since the start of the execution.

First consider the case $v < v.a$. Then $v.a.bound = v.b.bound = false$ since $coherent(v) = true$. In particular $v.a$ and $v.b$ were never passive and hence node v was never enabled by rule `[I1]` (note $v < v.a$). This yields that the values of $v.a$ and $v.b$ have not changed after v executed `[C2]` to set its $v.bound$

to *false*. Thus, predicate $agreed(v)$ evaluated to *true* when v previously executed `[C2]`. This is impossible, since $v.bound$ was set to *false*. This shows that $agreed(v.a) = agreed(v.b) = true$. Thus, v is part of penultimate constellation.

Next consider the case $v.a < v < v.b$. Then $v.a.bound = true$ and $v.b.bound = false$ since $coherent(v) = true$. This implies that v was never enabled by rule `[A1]`. This yields that the values of $v.a$ and $v.b$ have not changed after v executed `[C2]` to set its $v.bound$ to *false*. As in the first case this yields that v is part of penultimate constellation.

The last case $v.a < v.b < v$ is handled similarly. □

Corollary 5.5.5 *Each node makes at most three `[C2]` moves.*

Lemma 5.5.6 *After predicate $passive(v)$ evaluates to true for a node v, this node makes at most one more `[C2]` move.*

Proof. Consider a configuration c in which v is enabled to make a `[C2]` following a configuration with $passive(v) = true$. Note that $v.S = N[v]$ in c and v cannot execute `[C2]` move if v is not agree with two neighbors $v.a$ and $v.b$ (*i.e.* $agreed(v)$). By Lemma 5.5.3, it suffices to prove that v belongs to a penultimate constellation. Since $passive(v) = true$ implies $v.bound = false$, node v must satisfy $coherent(v) = true$ in c. Hence v must have updated the values of a and b before c. This can only be achieved by v executing a move of type `[I1],[A1]` or `[A2]`.

(i) If v executed move `[I1]` then $v < v.a < v.b$. After this move of v node $v.a$ must have executed `[A1]` and $v.b$ must have executed `[A2]` for v to satisfy $coherent(v) = true$. Hence, v belongs to a penultimate constellation. Note that during execution of move `[I1]` by v, the nodes $v.a$ and $v.b$ can execute other moves of type `[I1],[A1]` or `[A2]` for inviting or accepting an adjacent triangle. In this case, the node v will be not agree and therefore v cannot execute $[C2]$ move. (ii) If v executed move `[A1]` then $v.a < v < v.b$. This requires $semiTriangle(v, v.a, v.b)$ and $passive(v.b)$. After this move of v node $v.b$ must have executed `[A2]` for v to satisfy $coherent(v) = true$. Hence, v also belongs to a penultimate constellation. (iii) The case that v executed a `[A2]` is treated similarly. □

Lemma 5.5.7 *Each node v makes at most one `[W1]` move.*

Proof. The rules of the algorithm set variables $v.a$ and $v.b$ of a node v either both to $\perp$ or both to a value different from $\perp$ such that $v.a < v.b$. Hence, each node u having $u.a \geqslant u.b$ or $u.a \neq \perp$ and $u.b = \perp$ (resp. $u.a = \perp$ and $u.b \neq \perp$) means that u has pointers from starting configuration and not from the execution of the Algorithm 3. Then each node makes at most one `[W1]` move. □

Lemma 5.5.8 *Each node v makes at most $d(v)$ `[W2]` moves.*

Proof. A node v making a `[W2]` move is not enabled with respect to `[W1]` or `[C1]`, *i.e.*, $v.a \neq \perp$, $v.b \neq \perp$, $v.a < v.b$, and $v.s = N[v]$. In between two `[W2]`

moves node v must make a [I1], [A1], or [A2] move. Each such move requires $pseudoTriangle(v, v.a, v.b)$ to be true. Thus, this predicate is invalidated between any two [W2] moves. This can only be caused by $v.a$ or $v.b$ making move [C1]. Since each neighbor can do this only once, the proof is complete. □

Lemma 5.5.9 *Each node v makes at most $2d(v)$ [W3] moves.*

Proof. A node v makes a [W3] move only if it is not enabled for [W2] and [W1]. By Corollary 5.5.5 each node changes its variable *bound* at most twice to *true*. Moreover, the values of $v.a$ and $v.b$ created by rules [I1], [A1], [A2] always satisfy $v.a.bound = false$ and $v.b.bound = false$. Thus, v withdraws its variables $v.a$, $v.b$ at most $2d(v)$ times. □

Lemma 5.5.10 *Each node makes at most one [W4] move.*

Proof. Observe that a node v having $v < v.a$ cannot execute [W4]. Since rule [W4] can be enabled if and only if [W1], [W2], [W3] are disabled, we have $v.a < v.b$, $pseudoTriangle(v, v.a, v.b)$ and $\neg v.a.bound \vee semiTriangle(v, v.a, v.b)$ and $\neg v.b.bound \vee semiTriangle(v, v.b, v.a)$. Thus, only two situations are possible, either $v.a < v < v.b$ or $v.a < v.b < v$.

Except for initial configurations, a node v can have pointers $v.a, v.b$ such that $v.a < v < v.b$ (resp. $v.a < v.b < v$) only by executing rule [A1] (resp. [A2]). Hence, we prove in the following that if v executes [A1] or [A2] then v will never be enabled by [W4]:

Claim 1: If v executes [A1], then v will never be enabled by [W4].

Proof. Recall that v executes [A1] if and only if v has two neighbors u, w such that $u.a = v$ and $u.b = w$ and $pseudoTriangle(v, u, w)$ and $passive(w)$ and $u < v < w$. So, during the [A1] move of v, the node u is disabled for [W1] because $u.a \neq \perp$ and $u.b \neq \perp$ and $u.a < u.b$ and it is disabled for [W3] because $u.a.bound = false$ and $u.b.bound = false$. u is also disabled for [W4] because $u < u.a$. Moreover, if $pseudoTriangle(u, v, w)$ is valid, rule [W2] will also be disabled for u. This makes v ineligible for [W4]. Nevertheless, if $pseudoTriangle(u, v, w)$ is *false*, then $u.S$ and $w.S$ are incorrect. Node u will be enabled by [C1]. So, when u executes [C1] for updating $u.S$ then $pseudoTriangle(v, u, w)$ will be *false* for v and v will be enabled for [W2] and not for [W4]. In addition, if w changes $w.bound$ to *true*, then, by Lemma 5.5.6, $w.bound$ will always keep this value. Hence, nodes v and u will be both enabled by [W3] and not by [W4]. So, we deduce that after execution [A1], v will never be enabled by [W4]. □

Claim 2: If v executes [A2], then v will never be enabled by [W4].

Proof. Recall that node v executes [A2] if and only if there exist two neighbors u, w such that $u < w < v$ and $w.a = u$, $w.b = v$, $u.a = w$, $u.b = v$, $\neg u.bound$

and $\neg w.bound$ and $pseudoTriangle(v, u, w)$. According to the value of variable S at nodes u and w, there are two cases:

Case 1: $pseudoTriangle(u, v, w) = false$. Then the predicates $pseudoTriangle(v, u, w)$ and $\neg pseudoTriangle(u, v, w)$ are *true*. This implies that $u.S$ and $w.S$ are incorrect. In this case, nodes u and w are enabled by `[C1]` and when at least one of them executes `[C1]` during or after the move of v, node v will have $pseudoTriangle(v, u, w) = false$. Hence, node v will enabled by `[W2]` and not by `[W4]`.

Case 2: $pseudoTriangle(u, v, w) = true$. When v executes `[A2]` then v, u, w will be in a penultimate constellation with $agreed(v)$, $agreed(u)$, and $agreed(w)$ being *true*. Thus, using Lemma 5.5.3, any of the three nodes v, u and w can change the values of variables a and b anymore and the only rules that may be executed are `[C1]` and `[C2]`. □

So, we deduce that if any node v executes `[I]`, `[A1]` or `[A2]`, then v will never be enabled by `[W4]`. In summary, rule `[W4]` is only executed only at an initial configuration of a node. □

Theorem 5.5.11 *The algorithm SMPT^D converges after $O(m)$ moves under the unfair distributed daemon using $O(\Delta \log n)$ memory.*

Proof. Observe that only rules `[I1]`,`[A1]` and `[A2]` set the values of variables a and b to values different from $\perp$, whereas rules `[W1]`,`[W2]`,`[W3]` and `[W4]` set these variables to $\perp$. Hence, the number of executions of rules from the first group is at most the number of rules of the second group plus one. Using Lemmas 5.5.1, 5.5.7-5.5.10 and Corollary 5.5.5, it follows that each node $v \in V$ makes at most $6d(v) + 11$ moves. Thus, for a connected graph, Algorithm 3 makes $O(m)$ moves.

The memory requirement of the algorithm SMPT^D amounts to $O(\Delta \log n)$ per node: Apart of the boolean variable *bound*, a node has to store two *ids* for its variables a and b and at most $(\Delta + 1)$ *ids* for S. Thus, each node uses only $O(\Delta \log n)$ memory space. □

$O(m)$ is a tight bound ?

In the following a graph G will be presented demonstrating that the worst-case number of moves of the algorithm is at least m. The structure of G is depicted in Figure 5.5. G_1 is the subgraph induced by the nodes labeled 1 to 5. G_2 and G_3 are also induced subgraphs isomorphic to G_1. Each node v satisfies $v.a = v.b = \perp$ and $v.bound = false$. Nodes 1 to 5 are all enabled with respect to rule `[C1]` because variable S does not contain the correct closed neighborhoods (see Figure 5.5). The dashed lines in this figure indicate edges which the nodes believe to exist based on the values of S. Observe that from the perspective of node 1 it is adjacent to four triangles. Thus, $pseudoTriangle(1, 2, 4)$, $pseudoTriangle(1, 4, 5)$, $pseudoTriangle(1, 3, 5)$, and $pseudoTriangle(1, 2, 3)$ are all satisfied from node's 1 point of view. Hence node 1 first executes rule `[C1]` and afterwards executes rule `[I1]` four times before it finally forms the triangle $\{1, 4, 5\}$. Thus, each edge (v, u)

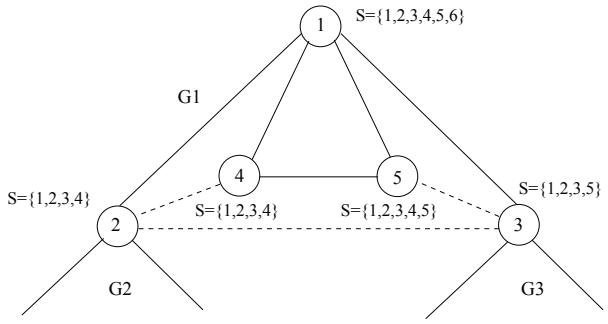


Figure 5.5: An initial configuration of a graph requiring m moves.

of the subgraph $G1$ induces at least one move. The same argument can be repeated for the induced subgraphs G_2 and G_3. Note that by attaching more copies of G_1 the graph may grow arbitrarily. In summary, the algorithm SMPT^D requires at least m moves for this graph.

5.6 Summary

In this chapter, we proved that finding a deterministic self-stabilizing algorithm for maximal partitioning into triangles is impossible in anonymous graphs under the distributed daemon. Moreover, assuming distinct local identifiers for breaking symmetry between nodes, we developed a new self-stabilizing algorithm (SMPT^D) for such problem, improving the previous one (SMPT^c) proposed in Chapter 4. The algorithm SMPT^D operates under the unfair distributed daemon and stabilizes within $O(m)$ moves where m is the number of edges in the graph G. Furthermore, we showed that this complexity is a tight bound for SMPT^D.

5.7 Conclusion

In this first part, we study the problem of maximal partitioning into triangles of general graphs. This partitioning is a generalization of maximal matching problem in graphs. We showed that finding a deterministic self-stabilizing algorithm for such problem is impossible in anonymous graphs. Moreover, we gave approximation of lower bound for this maximal partitioning, comparing with the maximum one. Furthermore, assuming distinct local identifiers, a first self-stabilizing algorithm for maximal graph partitioning into triangles is presented in Chapter 4. The first algorithm (SMPT^c) converges in polynomial moves under the central daemon only. Then, a second algorithm (SMPT^D) is developed in Chapter 5 in order to avoid the strong assumptions used in the first version. The second algorithm operates under the unfair distributed daemon and stabilizes in linear moves.

Part II

p-Star Decomposition (MSD)

Chapter 6

Introduction and motivation of part II

Contents

6.1 Introduction

In Part I, the study of maximal partitioning into triangles is developed and some of its applications are provided. Moreover, two self-stabilizing algorithms were presented for such partitioning. This partitioning into triangles can be seen as decomposition into patterns where each pattern is a triangle. Part II introduces another decomposition into other types of patterns where each pattern is a star. A star is a tree with one center node and leaf nodes (see Figure 6.1).

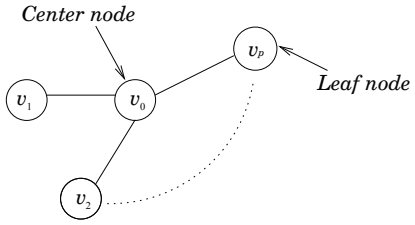
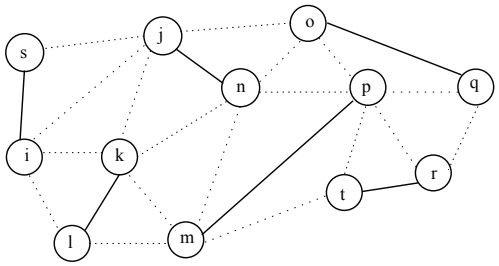


Figure 6.1: A star

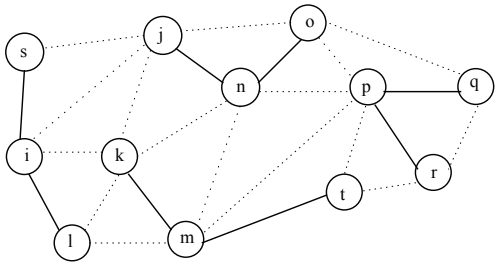
Star deomposition is one of the well studied graph decomposition problem, also called *star partitions* in graph theory [Cai74, SW93, LS96, BEZE01, LL05, MG12]. Note that the terms of partitioning and decomposition have the same meaning. Thus, the two terms will be used interchangeably throughout this part.

The star decomposition describes a graph as the union of disjoint stars [BEZE01]. An uniform decomposition into stars is the one in which all stars have equal size. A p-star is a complete bipartite graph $K_{1,p}$ with one center node and p leaves where $p \geq 1$ (see Figure 6.1). A p-star decomposition subdivides a graph into disjoint

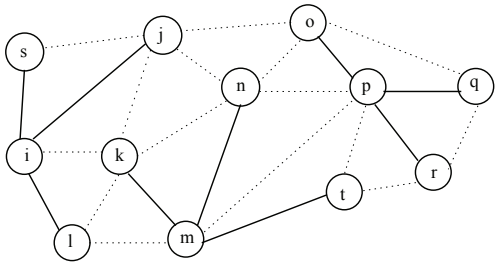
p-stars [Cai74, LL05]. This variant belongs to the class of generalized matchings and subgraph-decomposition problems that were proved to be NP-complete [KH78a, KH83, KH78b]. Figure 6.2 illustrates an example of p-star decomposition of a given graph. Note that a 1-star decomposition and 2-star decomposition are equivalent to a matching in graphs (Figure 6.2(a)) and path decomposition of graphs where length paths is 2 (Figure 6.2(b)) respectively. Figure 6.2(c) illustrates an example of p-star decomposition where $p = 3$.



(a) 1-star decomposition



(b) 2-star decomposition



(c) 3-star decomposition

Figure 6.2: Examples of p-decomposition of a graph (The depicted edges form the p-star decomposition)

Since, the question of the existence or not of p-star decomposition of a given graph is NP-Complete [KH78b]. Moreover, a perfect decomposition into p-stars does not always exist for general graphs. Therefore, we consider the problem *Maximal p-Star Decomposition* defining a local maximization property. A p-star decomposition of the graph is maximal if it cannot be extended by p-star using only nodes that are not already in the decomposition. More formal definitions of this decomposition are given in the following section.

6.2 Definitions

Let p be a positive integer ($p \geq 1$).

Definition 9 (p-Star) *A graph $G = (V, E)$ is called a p-star if $|V| = p + 1$ and $\exists v \in V$ such that $E = \{(v, u) : u \neq v\}$. Node v is called the center of the p-star and a node $u \neq v$ is called a leaf (see Figure 6.1).*

The problem of maximal p-star decomposition (MSD) of general graphs is defined as follows.

Definition 10 (Maximal p-Star Decomposition) *A p-star Decomposition of a graph $G = (V, E)$ is a set SD of disjoint subsets of V such that each subset $U \in SD$ satisfies that $|U| = p + 1$ and $G[U]$ contains a p-star as a subgraph. The decomposition SD is called maximal (MSD) if no subgraph of $G[V \setminus \bigcup_{U \in SD} U]$ is a p-star.*

6.3 Motivation

As p-star decomposition is a generalization of matching problem ($p = 1$), many applications of maximal matching can also be applyed (*cf.* Chapter 3).

Moreover, star decompositions have several applications in areas such as scientific computing, scheduling, load balancing and parallel computing [AR04, Pot97]. Furthermore, they have been used for studying the robustness of social networks [LHK11, LHK13]. In addition to applications in distributed systems, the decomposition into p-stars is also used in the field of parallel computing and programming. This decomposition offers similar paradigm as the *Master-Slaves* (a.k.a *Master-Workers*) paradigm used in grid [MMT07] and P2P infrastructures [BMT09] and Wireless Sensors Networks [DXW09]. The Master-Slaves paradigm distinguishes between two entities: masters and slaves. A master is responsible for decomposing the problem into different tasks and distributes the tasks on its slaves and collects results in order to produce the final result of the computation.

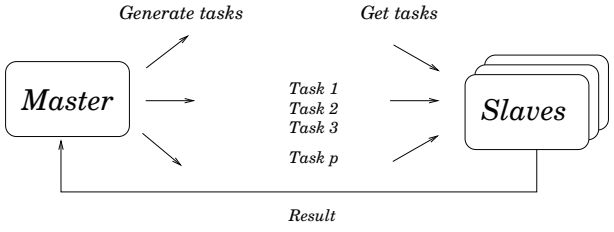


Figure 6.3: Master/Slaves model

The contribution of the second part of this thesis is the study of the *Maximal p-Star Decomposition* (MSD) of arbitrary graphs and developing different self-stabilizing algorithms for finding such decomposition using the distributed daemon

and distance-1 knowledge model. Chapter 7 presents a first self-stabilizing algorithm for finding a maximal p-star decomposition (SMSD^1) that converges in linear rounds under the distributed daemon. Formal proofs of its correctness and its convergence using graph reduction and induction technique are also presented. Moreover, based on the impossibility result of maximal matching presented in [MMPT09], an impossibility proof for finding a deterministic self-stabilizing algorithm for such decomposition in anonymous system is also deduced. Unfortunately, even the proposed algorithm converges in linear rounds, the simulations show that the first algorithm may provide an exponential number of moves under the distributed daemon. Moreover, SMSD^1 offers a unique legitimate configuration and it may consider a configuration with correct maximal p-star decomposition as not legitimate if it does not match with the unique configuration expected by SMSD^1. This is why, a second algorithm (called SMSD^2) is developed in Chapter 8. Hence, SMSD^2 converges in polynomial moves and considers all maximal p-star decompositions to be legitimate. Section 8.6 concludes this second part.

Chapter 7

Algorithm for MSD with unique legitimate configuration

Contents

7.1 Introduction

In the previous chapter, we introduced the problem of *Maximal p-star Decomposition* (MSD) of general graphs and we presented some applications of this parameter in distributed systems. In this chapter, we present a first self-stabilizing algorithm for finding an MSD of an arbitrary graph, called SMSD[1]. The algorithm works under the unfair distributed daemon and converges in linear rounds. This work is published in [NTHK13].

This chapter is organized as follows: In Section 7.2, we present an impossibility result for finding a deterministic self-stabilizing algorithm for MSD in anonymous graphs in order to justify the use of identifiers (*id*). Section 7.3 describes the first self-stabilizing algorithm (SMSD[1]) for MSD problem Formal proofs of its correctness and convergence are developed in Section 7.4 and 7.5 respectively. The analysis of the complexity of the proposed algorithm using graph reduction and induction technique is provided in section 7.6. Section 7.7 summarizes this chapter.

7.2 Impossibility result

In [MMPT09], Manne et al. proved that there is no deterministic self-stabilizing algorithm for the maximal matching problem that operates under the synchronous daemon and performs in arbitrary anonymous graphs. Their proof idea is as follows:

Assume that for each node in the graph, its local state is either *unmatched* or *proposed matched* with one of its neighbors. A configuration is legitimate for matching problem if every pair of nodes is consistent in their mutual relationship. Consider that a graph is a cycle of size at least 3. At starting, each node has the same state, and since the local view of each node is identical, this means that all nodes have the same behavior. Note that every node has exactly two neighbors, and a node in state *proposed matched* may either be directed clockwise or counter-clockwise. Thus, each node is either (1) unmatched, (2) clockwise proposed matched, or (3) counter-clockwise proposed matched. Since the local state of every node is identical and no node is matched, the initial configuration is not a maximal matching. Note that synchronous daemon is a special case of a distributed daemon, this means that finding a deterministic self-stabilizing algorithm for the maximal matching problem that operates under the distributed daemon is also impossible in anonymous graphs.

Since it is impossible to find a deterministic self-stabilizing algorithm for maximal matching in anonymous graph under a distributed daemon [MMPT09], and since the p-star decomposition is a generalization of the matching problem for which $p = 1$, then the impossibility result remains valid for p-star decomposition for all $p \geq 1$. Hence, any self-stabilizing algorithm requires a mechanism for symmetry breaking. Thus, the distinct node identifiers *ids* are used for such mechanism.

7.3 Algorithm description

This section presents a first self-stabilizing algorithm (called SMSD^1) for computing a maximal p-star decomposition of an arbitrary graph. Assume that all nodes have unique identifiers. We say that a node v_1 is smaller than a node v_2 (denoted by $v_1 < v_2$) if v_1's identifier is smaller than that of v_2. For notational convenience we assume $v < null$ for each node v.

The general idea of the proposed algorithm SMSD^1 is as follows: A node becomes a leaf node by selecting the smallest possible node as a center node. A node v becomes center node only if all nodes smaller than v are either center node or have decided not to become center node. In other words, the node v with the smallest identifier having at least p neighbors becomes center node. The p neighbors $v_1, \ldots, v_p$ of v with the smallest identifiers become the leaf nodes of v. This procedure is recursively repeated for the subgraph of G consisting of all nodes except $v, v_1, \ldots v_p$. The challenge is to design an efficient distributed version of this algorithm.

Let X be a set and p is positive integer. The algorithm SMSD^1 uses two operators X^p and $\min X$ that are defined as follows:

$$X^p = \begin{cases} \emptyset & \text{if } |X| < p \\ \text{the } p \text{ smallest elements of } X & \text{otherwise.} \end{cases}$$

$$\min X = \begin{cases} null & \text{if } |X| = \emptyset \\ \text{the smallest element of } X & \text{otherwise.} \end{cases}$$

Each node v of G maintains two variables m and s. Variable s of v contains the list of pointers to its p leaves and the variable m contains the pointer to the selected center node. If a node has not selected a center node then $m = null$ and if it has not selected any leaves then $s = \emptyset$.

Note that during the execution of the algorithm, a node v can be a member of the set of leaves of many neighbors. For a node v, the set of such neighbors is denote by $M(v)$. Formally, $M(v) = \{w \in N(v) \mid v \in w.s\}$.

Moreover, the set of potential leaves of a node v is denoted by $S(v)$. This set contains all neighbors w of v such that w is either a center node (*i.e.* $w.s \neq \emptyset$) and its identifier is bigger than v (*i.e.* $w > v$) or w is not a center node (*i.e.* $w.s = \emptyset$) and w points at *null* or to a center node bigger or equal to v (*i.e.* $w.m \geqslant v$). Formally,

$$S(v) = \{w \in N(v) \mid (w.s = \emptyset \wedge w.m \geqslant v) \vee (w.s \neq \emptyset \wedge w > v)\}.$$

For each node v two cases have to be distinguished. Case (1): v can not be a center node (*i.e.* $S(v)^p = \emptyset$) or v is pointed by a center node having a smaller identifier than v (*i.e.* $minM(v) < v$). In this case the correct values for $v.s$ and $v.m$ are $\emptyset$ and $\min M(v)$ respectively. This means that v becomes a leaf and v selects the smallest possible center node of v. These values are denoted by $v.s_{new}$ and $v.m_{new}$ respectively.

Case (2): v can be center node (*i.e.* $S(v)^p \neq \emptyset$) and v is pointed by center node with larger identifier than v or v is not pointed by any center node (*i.e.* $\min M(v) > v$). In this case the correct values for $v.s$ and $v.m$ are $S(v)^p$ and *null* respectively. This means that v becomes a center node and v selects the nodes in $S(v)^p$ as leaves. These values are also denoted by $v.s_{new}$ and $v.m_{new}$ respectively.

Formally, the algorithm SMSD^1 uses the following code permitting a node v to compute its new values of s_{new} and m_{new}.

if $(\min M(v) < v \vee S(v)^p = \emptyset)$ **then**
    $v.s_{new} := \emptyset$; $v.m_{new} := \min M(v)$;
else
    $v.s_{new} := S(v)^p$; $v.m_{new} := null$;

The proposed algorithm SMSD^1 consists of Rule `[R]` only. A node v is enabled if and only if $v.m \neq v.m_{new}$ or $v.s \neq v.s_{new}$. When v executes the rule `[R]`, v updates the values of s and m.

Algorithm 4: Self-stabilizing algorithm for MSD (SMSD^1)

Nodes: v is the current node

$v.m \neq v.m_{new} \vee v.s \neq v.s_{new} \longrightarrow v.m := v.m_{new}$; $v.s := v.s_{new}$; `[R]`

Consider a G a complete graph with a starting configuration, where each node v has $v.m = null$ and $v.s = \emptyset$, Figure 7.1 shows an example of the execution of

the algorithm SMSD[1] for such a graph with nine nodes. Using the synchronous daemon, the proposed algorithm finds two 3-stars and one single node after three rounds. The edges of the resulting two 3-stars are depicted in bold.

7.4 Correctness proof

This section proves that in configuration with no enabled node, the stars induced by all nodes v with $v.s \neq \emptyset$ form a maximal p-star decomposition of G.

Lemma 7.4.1 *In a configuration with no node is enabled, the following properties hold for each $v \in V$.*

(a) If $v.s \neq \emptyset$ then $v.s \subseteq N(v)$ and $|v.s| = p$ and $v.m = null$.

(b) If $v.m \neq null$ then $v.m \in N(v)$.

(c) If $v \in w.s$ then $v.m = w$ and $v.s = \emptyset$.

Proof. Let $v.s \neq \emptyset$. Then $v.s = v.s_{new} = S(v)^p$ since rule `[R]` is disabled. Thus, $v.s \neq \emptyset$ implies $v.s \subseteq N(v)$ and $|v.s| = p$. Moreover, if $v.s_{new} = S(v)^p$ then $v.m = v.m_{new} = null$. This proves property (a). Let $v.m \neq null$ then $v.m = v.m_{new} = \min M(v)$ since rule `[R]` is disabled. Thus, $\min M(v) \in N(v)$ implies $v.m \in N(v)$. This proves property (b).

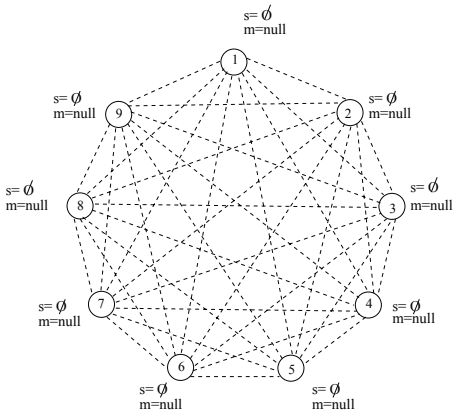
Property (c) is proven by contradiction. Suppose there exists $v, w \in V$ such that $v \in w.s$ and $v.s \neq \emptyset$ or $v.m \neq w$. First assume $v.s \neq \emptyset$. Since $v \in w.s$, $w.s \neq \emptyset$. Then $v.s = v.s_{new} = S(v)^p$ since rule `[R]` is disabled for v. They are two cases to consider.
Case $v < w$. $v.s \neq \emptyset$ and $v \in S(w)^p$ implies $v > w$. Contradiction.
Case $v > w$. Then $v \in w.s$ implies $w \in M(v)$. Furthermore, $\min M(v) < v$ since $w < v$ and $w \in M(v)$. Thus, $v.s = v.s_{new} = \emptyset$. Contradiction.

This yields that $v \in w.s$ implies $v.s = \emptyset$.

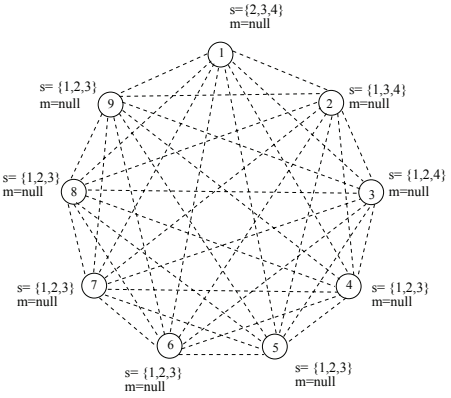
Next consider the remaining case $v.m \neq w$. Using the previous result, if $v \in w.s$ then $v.s = \emptyset$. Hence, $v.m \neq w$ implies $v.m = null$ or $v.m = u$ such that $u \neq w$. By assumption, $v \in w.s$, *i.e.* $w \in M(v)$. This implies $\min M(v) \neq null$. To obtain a final contradiction the remaining proof is split into two cases for v:

1. If $\min M(v) < v$ or $S(v)^p = \emptyset$ then $v.s_{new} = \emptyset$ and $v.m_{new} = \min M(v)$. Further analysis depends on the value of $v.m$. If $v.m = null$ then we have $\min M(v) \neq null$ and $v.m = null$, this implies that $v.m \neq v.m_{new}$. Contradiction. On the other hand if $v.m = u$ and $u \neq w$ then $v \in u.s$. Assume that $u < v$ (resp. $u > v$) and by assumption $v \in S(w)$, this implies that $w.s \neq S(w)^p$. So, rule `[R]` is enabled for w (resp. for u). Contradiction.

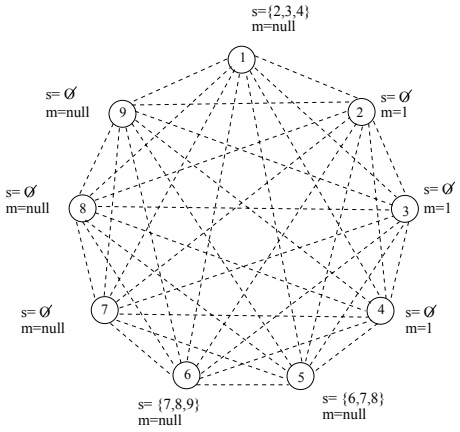
2. If $\min M(v) \geq v$ and $S(v)^p \neq \emptyset$ then $v.s_{new} = S(v)^p$ and $v.m_{new} = null$. Node v is disabled by rule `[R]`, *i.e.* $v.m = v.m_{new} = null$. So, based on the previous result, if $v \in w.s$ then $v.s = \emptyset$. Hence, $v.s = \emptyset$ and we have $S(v)^p \neq \emptyset$. This implies $v.s \neq v.s_{new}$ and rule `[R]` is enabled for v. Contradiction.



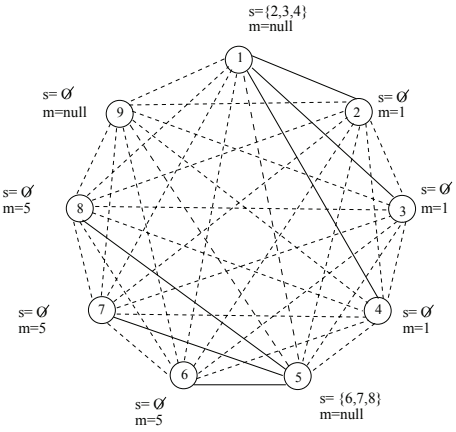
(a) Initial configuration



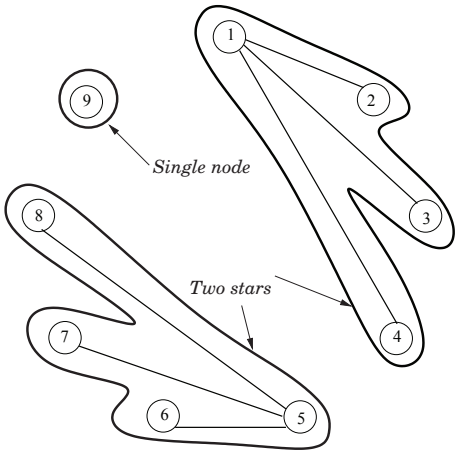
(b) Round 1



(c) Round 2



(d) Round 3



(e) Final configuration

Figure 7.1: Example of executing SMSD^1 under the synchronous daemon ($p = 3$).

We conclude that if $v \in w.s$ then $v.m = w$ and $v.s = \emptyset$. This completes the proof of property (c). □

Consider a configuration with no enabled node. Let S be the set of all nodes $v \in V$ with $v.s \neq \emptyset$. By Lemma 7.4.1, each node v of S together with the p nodes in $v.s$ forms a star in G. These stars do not overlap.

Lemma 7.4.2 *In a configuration with no enabled node the stars induced by all nodes v with $v.s \neq \emptyset$ form a maximal p-star decomposition of G.*

Proof. It is sufficient to prove that the decomposition is maximal. Let $v \in V$ such that $v.s = \emptyset$ and $v.m = null$. Assume that v is the center of a star with p leaves that are neither contained in S nor leaves of a center node contained in S. Then $w.m = null$ and $w.s = \emptyset$ for every leaf w of v. This implies that all p leaves of v are contained in $S(v)$. This is impossible because otherwise node v would be enabled. □

7.5 Convergence proof

In this section, the convergence of the algorithm SMSD^1 under the unfair distributed daemon is proved. The time complexity of the algorithm is measured in rounds. Note that in general a round under an unfair distributed daemon may consist of an infinite number of moves. Thus, it is not sufficient to prove that the algorithm stabilizes after a finite number of rounds. For this reason, we first prove in Theorem 7.5.4 that the algorithm SMSD^1 requires only a finite number of moves which implies its convergence. In the following the usage of the unfair distributed daemon is assumed.

A move of a node v is called m-move (resp. s-move) if v executes rule `[R]` and assigns a new value to $v.m$ (resp. $v.s$). Thus, a move can be an m-move and an s-move at the same time.

Lemma 7.5.1 *Let $v \in V$ and e be an execution of SMSD^1 such that no node u with $u < v$ makes an s-move in e. Then v makes at most $d(v) + 2$ s-moves in e.*

Proof. We prove that each node $u \in N(v)$ may enter or leave the set $S(v)$ at most once during e. Since between two s-moves of v the set $S(v)$ must change the result as follows. Let $w \in N(v)$ and c be the first configuration in e where w makes a move. Denote by e_c the remaining execution, *i.e.*, the suffix of e beginning in c. So, there are two possible cases for w:

Case $\min M(w) < v$. Let $u = \min M(w)$. Since $u < v$, by assumption $u.s$ will never change, thus $u \in M(w)$ holds forever and hence $\min M(w) < v$ holds forever. If $w > v$ then $\min M(w) < w$ holds forever, this implies that $w.s = \emptyset$ will be satisfied from now on. Hence, w will never be part of $S(v)$ in the future. If $w < v$ then because $\min M(w) < v$ holds forever, w will also never be part of $S(v)$

in the future. In summary, if $\min M(w) < v$, node w will at most once drop out of $S(v)$.

Case $\min M(w) \geq v$. Again by assumption $\min M(w) \geq v$ holds for the rest of the execution. Consider the case $w > v$. If there exists $u < v$ with $w \in u.s$ then w will never be part of $S(v)$ in the future. If $w \notin u.s$ for all $u < v$, then w will be forever in $S(v)$. Next let $w < v$. Then $w.s$ will never change by assumption. Then as in the previous case, node w will never be part of $S(v)$ in the future or will be forever contained in $S(v)$. In summary, if $\min M(w) \geq v$, node w will at most once drop out of $S(v)$ or will be inserted at most once into $S(v)$.

Hence, each neighbor of w induces at most one change of $S(v)$. Furthermore, all nodes u with $u < v$ cause together at most one change of $S(v)$. In total we have at most $d(v) + 2$ s-moves of v. □

Lemma 7.5.2 *The total number of s-moves in any execution of SMSD^1 is finite.*

Proof. The proof is by induction on the identifier of the nodes. The node v with the smallest identifier makes at most $d(v)+2$ s-moves by Lemma 7.5.1. Let $w \in V$, with $w \neq v$. By induction there exists a number C such that all nodes with identifiers less than w make together at most C s-moves. Then Lemma 7.5.1 implies that w makes at most $(C + 1)(d(w) + 2)$ s-moves. This completes the proof. □

Lemma 7.5.3 *Let Δ be the maximum node degree in the graph G. The total number of m-moves in any execution of SMSD^1 is at most $\Delta C + n$, here C denotes the total number of s-moves.*

Proof. If a node v makes a m-move then the set $M(v)$ has changed since the last m-move of v or it is v's first m-move. The set $M(v)$ changes if the membership of v in $w.s$ for a node $w \in N(v)$ changes. This is caused by an s-move of w. In the worst case a s-move of a node u changes the sets $w.s$ of all neighbors w of u. This completes the proof. □

Theorem 7.5.4 *The algorithm SMSD^1 is a self-stabilizing algorithm for computing a maximal p-star decomposition and stabilizes in finite time under the unfair distributed daemon.*

Proof. The convergence property of SMSD^1 follows from Lemmas 7.5.2 and 7.5.3. The correctness property was shown in Lemma 7.4.2. □

7.6 Complexity analysis

In the following, we analyze the round complexity of the algorithm SMSD^1 under the unfair distributed daemon.

Lemma 7.6.1 *After round r_0 and in all following rounds, each node $v \in V$ satisfies the following properties.*

(a) $v.m = null$ *or* $v.m \in N(v)$.

(b) if $v.s \neq \emptyset$ *then* $|v.s| = p \wedge v.s \subseteq N(v) \wedge d(v) \geq p \wedge v.m = null$.

Proof. It is obvious that any node $v \in V$ that does not satisfy properties (a) and (b) is enabled and when v executes rule `[R]` during r_0, v will satisfy both of these properties because $v.m_{new} = null$ or $v.m_{new} \in N(v)$ and $v.s_{new} = \emptyset$ or $|v.s_{new}| = p$. Note that $v.s_{new} \subseteq N(v)$. □

Lemma 7.6.2 *After round* r_1 *and in all following rounds, each node* $v \in V$ *with* $v.m = u$ *satisfies* $d(u) \geq p$ *and* $v.s = \emptyset$.

Proof. After the first round r_0, if a node u has $d(u) < p$ then $u.s = \emptyset$ (Lemma 7.6.1). Moreover, u will never have $u.s \neq \emptyset$ because $S(u)^p = \emptyset$ independently of $\min M(u)$ and $S(u)^p$. Hence, u keeps its value $u.s = \emptyset$. So, after round r_1 and for all following rounds, we have $u \notin M(v)$ for all $v \in V$. This completes the proof. □

Lemma 7.6.3 *Let* v^* *be the smallest node in* G *such that* $d(v^*) \geq p$. *Then,*

(a) after round r_2 *and in all following rounds,* $v^*.m = null$ *and* $v^*.s = N(v^*)^p$.

(b) Let be $S^* = (v^* \cup v^*.s)$. *After round* r_3 *and in all following rounds,* $v.m \notin S^*$ *and* $v.s \cap S^* = \emptyset$ *for all* $v \in V(G) \setminus S^*$.

Proof. For proving property (a), it is sufficient to prove that during round r_2 and in all following rounds, we have $v^*.m_{new} = null$ and $v^*.s_{new} = S(v^*)^p$. This implies that $\min M(v^*) \geq v^* \wedge S(v^*)^p \neq \emptyset$.

By assumption and according to Lemmas 7.6.1 and 7.6.2, v^* is the smallest node such that $v^*.s \neq \emptyset$. Hence, after r_1, we have $\min M(v^*) > v^*$. Now, we prove that during round r_2, we always have $S(v^*)^p \neq \emptyset$.

By definition, $S(v^*) = \{w \in N(v^*) \mid (w.s = \emptyset \wedge w.m \geq v^*) \vee (w.s \neq \emptyset \wedge w > v^*)\}$. Now, we show that after round r_1, any neighbor w of v^* belongs to $S(v^*)$. For a node w two cases have to be considered.

Case $w.s \neq \emptyset$. Then using Lemma 7.6.1, $w > v^*$. This implies that $w \in S(v^*)$.

Case $w.s = \emptyset$. Then node w can have $w.m = null$ or $w.m \neq null$. If $w.m \neq null$ then by Lemma 7.6.1, $w.s = \emptyset$. By assumption and using Lemma 7.6.2, we have $w.m > v^*$ and $w.s = \emptyset$, this implies that $w \in S(v^*)$. If on the other hand $w.m = null$ then this yields $w.s = \emptyset$ and $w.m = null > v^*$. This implies that $w \in S(v^*)$.

We deduce that any neighbor w of the node v^* belongs of $S(v^*)$ independent of the values of $w.m$ or $w.s$. Hence, $S(v^*) = N(v^*) \neq \emptyset$. So, during round r_2 and in all following rounds, we have $S(v^*) = N(v^*) \neq \emptyset$. This implies $v^*.m_{new} = null$ and $v^*.s_{new} = N(v^*)^p$. Thus, if $v^*.m \neq v^*.m_{new}$ or $v^*.s \neq v^*.s_{new}$ then v^* executes rule `[R]` and updates its variables such that $v^*.m = null$ and $v^*.s = N(v^*)^p$ after round r_2 and v^* will never make a move again.

Property (b) means that after round r_3, there is no node $v \in V \setminus S^*$ depending on the star S^* formed by $v^* \cup v^*.s$. As previously shown, after round r_2 and in all following rounds, node v^* satisfies $v^*.m = null$ and $v^*.s = N(v^*)^p$. Hence, after round r_2, each node w not belonging to star S^* that satisfies $w.m \in S^*$ or $w.s \cap S^* \neq \emptyset$ will be enabled by rule `[R]` and must execute this rule before the end of round r_3. Thus, after round r_3, any node w not belonging to S^* will have $w.m \notin S^*$ and $w.s \cap S^* = \emptyset$. □

Lemma 7.6.4 *The algorithm SMSD^1 stabilizes after at most $2\lfloor \frac{n}{p+1} \rfloor + 2$ rounds.*

Proof. The proof is by induction. Consider the first two rounds r_0 and r_1. Each node satisfies the properties stated in Lemmas 7.6.1 and 7.6.2. Let be v^* the node with the smallest identifier in G with degree at least p (*i.e.* $d(v^*) \geq p$). Using Lemma 7.6.3, the star S^*, which contains the node v^* as a center node and $v^*.s$ as leaf nodes, will stabilize after at most two successive rounds and any node belonging to this star (*i.e.* nodes in $\{v^*\} \cup v^*.s$) will never make a move again. Let G' be the graph obtained by removing the nodes of S^* from G. The argument given above can be repeated. Hence, by induction, each star stabilizes after at most two more rounds. Since G contains at most $\lfloor \frac{n}{p+1} \rfloor$ stars, SMSD^1 will stabilize after at most $2\lfloor \frac{n}{p+1} \rfloor + 2$ rounds. □

Theorem 7.6.5 *SMSD^1 is self-stabilizing algorithm for maximal p-star decomposition and converges after at most $2\lfloor \frac{n}{p+1} \rfloor + 2$ rounds under the unfair distributed daemon using $O(p \log n)$ memory.*

Proof. The result is a direct consequence of Theorem 7.5.4 and Lemma 7.6.4. □

7.7 Summary and Discussions

In this chapter, we presented a first self-stabilizing algorithm for graph decomposition into disjoint p-stars, called SMSD^1. The algorithm operates under the unfair distributed daemon and stabilizes after at most $2\lfloor \frac{n}{p+1} \rfloor + 2$ rounds using $O(p \log n)$ memory where n is the number of nodes in the graph G and p is a positive integer.

However, the proposed algorithm SMSD^1 has a weak point which cannot be ignored. SMSD^1 may consider a configuration with correct maximal p-star decomposition as an illegitimate configuration if it does not match with the unique configuration expected by the algorithm. Let us show this point by a simple example. Consider a graph G as path of four nodes v_1, v_2, v_3, v_4 such that $v_1 < v_2 < v_3 < v_4$ and we fixe $p = 1$. This means that the 1-star decomposition provides a maximal matching in the graph G. Figure 7.2 depicts an example of SMSD^1 execution under the synchronous daemon.

The starting configuration is a correct maximal 1-star decomposition where the node v_1 (resp. v_2) is a center node of a star and has v_4 (resp. v_3) as a leaf node, as illustrated in Figure 7.2(a). However, this configuration is not legitimate because

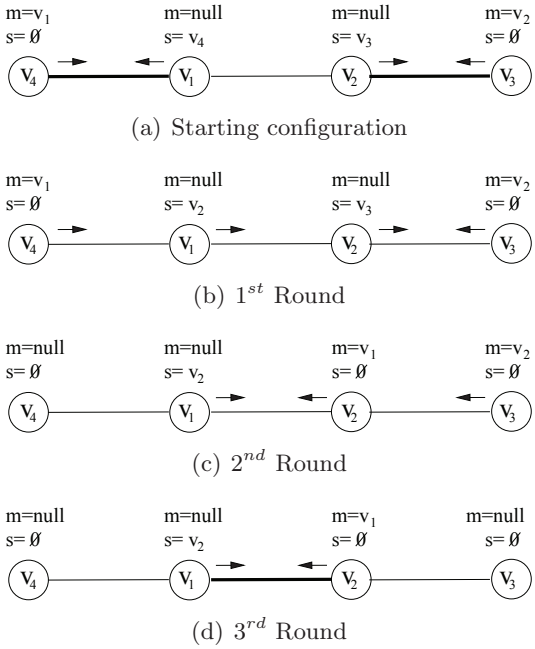


Figure 7.2: Example of executing SMSD^1 under the synchronous daemon ($p = 1$).

the node v_1 is enabled ($v_1.s_{new} = \{v_2\}$ and $v_1.s = \{v_4\}$). So, after the first round v_1 executes `[R]` and it will have $v_1.s = \{v_2\}$ and $v_1.m = null$ (See Figure 7.2(b)). In the 2^{nd} configuration, the node v_2 is enabled by `[R]` because $min\ M(v_2) = v_1 < v_2$ and v_4 is also enabled because $S(v_4) = \emptyset$ and $v_4.m \neq minM(v_4)$. So, after the 2^{nd} round, the nodes v_2 and v_4 will have $v_2.s = v_4.s = \emptyset$ and $v_2.m = v_1$ and $v_4.m = null$ (See Figure 7.2(c)). In this configuration, the node v_3 is enabled and after the 3^{rd} round, v_3 will have $v_3.s = \emptyset$ and $v_3.m = v_4.m = null$ (Figure 7.2(d)). We conclude that for any starting configuration, the algorithm SMSD^1 always leads to a unique configuration that is p-star decomposition.

Concerning the complexity of SMSD^1, the number of moves seems to be exponential even if its rounds complexity is linear. So, we simulate the algorithm SMSD^1 on a complete graph with initially values $v.m = null$ and $v.s = \emptyset$ for every node v. We assume the unfair central daemon that selects the enabled node that have the highest *id*. The following tables 7.1 and 7.2 present the number of moves for the stabilization with respect to the number of nodes n. Then, we remark that the number of moves needed for the stabilization get huge very quickly. However, we are not aware of an example where SMSD^1 requires an exponential number of moves.

n	3	4	16	20	21	22	23
Moves	8	13	8911	65072	106686	174755	285989

Table 7.1: Simulation on complete graph with n nodes and $p = 2$.

n	3	4	16	20	21	22	23
Moves	0	13	6176	44757	73403	120260	196904

Table 7.2: Simulation on complete graph with n nodes and $p = 3$.

Chapter 8

Algorithm for MSD with multi-legitimate configurations

Contents

8.1 Introduction

In the previous chapter, we developed a first algorithm ($SMSD^1$) for maximal p-star decomposition of general graphs. The algorithm converges in linear rounds under the unfair distributed daemon. We proved that moves complexity of $SMSD^1$ is bounded but simulations showed its exponentiality. Moreover, $SMSD^1$ provides a unique legitimate configuration. In this chapter, a second self-stabilizing algorithm for MSD, called $SMSD^2$, is developed in order to outperform the first algorithm on two points: (i) $SMSD^2$ offers more than one legitimate configuration instead of a unique one for the previous one, *i.e.* $SMSD^2$ considers all maximal p-star decompositions to be legitimate. (ii) $SMSD^2$ has a polynomial move complexity under the unfair distributed daemon.

This chapter is organized as follows: In Section 8.2, we present the second algorithm $SMSD^2$. Then, we give formal proofs of its correctness and convergence in Section 8.3 and Section 8.4 respectively.

8.2 Algorithm description

In this section, we present a second self-stabilizing algorithm for maximal p-star decomposition of an arbitrary graph G, called $SMSD^2$. The algorithm $SMSD^2$ can be summarized as follows: each node v that does not belong to a p-star invites the smallest neighbor to be its center node of a p-star. If a node v has at least p neighbors that point at only v, then v accepts their invitations by pointing at the

p smallest nodes of these nodes. If a node v points at a neighbor u and u already belongs to a p-star then v withdraws its invitation or invites another neighbor.

Let X be a set of nodes and p a positive integer. SMSD^2 uses the same operator X^p as defined for SMSD^1. Recall that X^p returns at most the p smallest nodes of X and it is defined as follows :

$$X^p = \begin{cases} \emptyset & \text{if } |X| < p \\ \text{the } p \text{ smallest elements of } X & \text{otherwise.} \end{cases}$$

Each node v, in the graph G, maintains a list of pointers $v.L$ that defines the neighbors of v in the p-star to which node v may belong. We say $v.L$ is *coherent*, if $|v.L| \in \{0, 1, p\}$ and $v.L \subseteq N(v)$.

Note that $v.L$ contains only pointers (i.e. *id*) of v's neighbors. So, it is possible that in the starting configuration, some nodes can have incoherent pointers list. For this reason, we use the predicate $incoherent(v)$ in the first Rule `[U1]`. Formally, $incoherent(v) \equiv (|v.L| \notin \{0, 1, p\}) \vee (v.L \nsubseteq N(v))$.

In addition to the pointers list $v.L$, SMSD^2 uses two boolean variables $v.inStar$ and $v.leaf$. A node v uses the variable $v.inStar$ to inform its neighbors whether v belongs to a p-star ($v.inStar = true$) or not ($v.inStar = false$). The second variable $v.leaf$ is used to inform neighbors whether v can be a center node ($v.leaf = false$) or not ($v.leaf = true$). The new value $v.leaf$ is computed using the function $S(v)$. Thus, $S(v)$ returns *true* if the number of neighbors of v that do not belong yet to any p-star is less than p else $S(v)$ returns $false$. Formally, $S(v) = true$ if $|\{u \in N(v) : u.inStar = false\}| < p$ else $S(v) = false$.

We say that the node v becomes *agreeing* to be a center node of a p-star if v points at p neighbors defined by $v.L$ and each node u from this list has $u.L = \{v\}$. However, a node v becomes *agreeing* to be a leaf node of a p-star if v points at a neighbor u such that $|u.L| = p$ and $u.inStar = true$ and $v \in u.L$. So, any node v is agreeing if and only if the predicate $agreed(v) = true$. Formally, *agreed* is defined as follows:

$$agreed(v) \equiv \begin{cases} (v.L = \{u\} \wedge u.inStar = true \wedge |u.L| = p \wedge v \in u.L) \vee \\ (\forall u \in v.L : u.L = \{v\} \wedge |v.L| = p) \end{cases}$$

Note that during the execution of SMSD^2, a node v can have more than one neighbor that can be its center node. The set of such neighbors is denoted by $C(v)$, formally $C(v) = \{u \in N(v) : u.inStar = false \wedge u.leaf = false \wedge |u.L| \leqslant 1\}$. Moreover, a node v can be pointed by many neighbors that point at only v. Then, the set of such nodes is defined by $A(v)$. Formally, $A(v) = \{u \in N(v) : u.L = \{v\}\}$.

Considering the unfair distributed daemon, some enabled nodes can make their moves simultaneously. Hence, we make a serialization technique for the execution of critical moves in order to reduce the moves complexity of the algorithm. If the nodes can change their pointers list $v.L$ simultaneously, this induces a higher moves complexity. Hence, the trick is to use a Boolean flag, called $v.flag$ for each node $v \in V$ in order to inform its neighbors that v wants to execute a critical move. The updating of this flag uses the predicate $want\_to\_change(v)$, defined as follows:

$want\_to\_change(v) \equiv v.instar = false \wedge (A(v)^p \neq \emptyset \vee v.L \neq C(v)^1)$.
Hence, the smallest node in the neighborhood having this flag to *true* executes its critical move for changing its pointer list. This can happen by checking the predicate $Smallest(v)$. Formally, $smallest(v) \equiv v.flag = true \wedge \nexists u \in N(v) : u.flag = true \wedge u < v$.

The complete set of rules of SMSD^2 is shown in Algorithm 5. The six rules can be categorized in two groups. We assume an order between rules as presented in the algorithm, for example Rule `[U2]` is executed only if `[U1]` is disabled and so on. Considering a node v, the Rules of the first group (`[U1,U2,U3,U4]`) keep the variables $v.L$, $v.leaf$, $v.flag$ and $v.instar$ up to date. The rules of the second group (`[A,I]`) are responsible for creating and accepting invitations (critical moves). The Rule `[A]` permits to a node v for accepting invitations from its p neighbors and the Rule `[I]` for inviting the smallest possible neighbor to be a center node or to $\emptyset$ where $C(v)^1 = \emptyset$ (Recall that $C(v)^1$ returns the smallest node in $C(v)$). Note that the rules of the second group require that $v.inStar = false$. This means that if a node v is *agreeing* then v cannot execute any rule of this group (Since `[U3]` is disabled). Moreover, only one node can execute an acceptation or an invitation move in the same neighborhood, by checking if the current node v has the smallest $v.flag = true$ within its neighborhood (using predicate $smallest(v)$).

Algorithm 5: Self-stabilizing algorithm for MSD (SMSD^2)

$incoherent(v) \longrightarrow v.L := \emptyset$; $v.inStar := false$;	`[U1]`
$v.leaf \neq S(v) \wedge v.inStar = false \longrightarrow v.leaf := S(v)$;	`[U2]`
$v.inStar \neq agreed(v) \longrightarrow v.inStar := agreed(v)$;	`[U3]`
$v.flag \neq want\_to\_change(v) \longrightarrow v.flag := want\_to\_change(v)$;	`[U4]`
$v.inStar = false \wedge A(v)^p \neq \emptyset \wedge smallest(v) \longrightarrow v.L := A(v)^p$; $v.inStar := true$;	`[A]`
$v.inStar = false \wedge v.L \neq C(v)^1 \wedge smallest(v) \longrightarrow v.L := C(v)^1$;	`[I]`

The intuitive idea of SMSD^2 is as follows: each node v which is agreeing for belonging to a p-star has $v.inStar = true$. Otherwise, v points by $v.L$ at the smallest possible center, defined by $C(v)^1$. Thus, if a node v has at least p neighbors that point at only v (i.e. $A(v)^p \neq \emptyset$) and v is the smallest node for executing acceptation or invitation moves then v accepts the invitations of its p smallest neighbors, by executing the Rule `[A]`. Observe that any neighbor u that points at v cannot change their pointer list because Rules `[A]` and `[I]` are disabled ($smallest(u) = false$) during v's move.

8.3 Correctness proof

First, we prove that in a configuration where no node is enabled, each node having $|v.L| = p$ is a center of a p-star, defined by $\{v\} \cup v.L$. Moreover, the union of such

p-stars forms a maximal p-star decomposition of the graph G. Recall that a node v is enabled if one rule of the algorithm SMSD (`[U1-U4]`, `[I]` or `[A]`) is enabled.

Lemma 8.3.1 *In configuration with no enabled node, the following properties holds for each node $v \in V$:*

(a) $v.L$ is coherent, i.e. $|v.L| \in \{0, 1, p\}$ and $v.L \subseteq N(v)$.

(b) if $v.inStar = false$ then $v.leaf = S(v)$.

(c) if $|v.L| = p$ then $v.inStar = agreed(v) = true$.

(d) if $|v.L| = p$ then $u.inStar = agreed(u) = true$ for any $u \in v.L$.

Proof. Since Rule `[U1]` is disabled for $v \in V$, we have $incoherent(v) = false$. This means that $|v.L| \in \{0, 1, p\}$ and $v.L \subseteq N(v)$. This proves Property (a).

Since Rule `[U2]` is disabled and using Property (a) then $v.leaf = S(v)$ for any v with $v.inStar = false$. This proves Property (b).

Considering the first part of Property (c), i.e. if $|v.L| = p$ then $agreed(v) = true$. Assume in a configuration where no node is enabled, there exists a node $v \in V$ with $|v.L| = p$ and $agreed(v) = false$. Using Property (a), we have $v.L \subseteq N(v)$. So, there are two cases: (i) if $v.inStar = true$ then `[U3]` is enabled because $v.inStar \neq agreed(v)$. Contradiction with assumption. (ii) if $v.inStar = false$ then $v.leaf = S(v)$ (Property (b)) and $agreed(v) = v.inStar = false$ (because Rule `[U3]` is disabled). Since Rule `[U4]` is disabled then $v.flag = want\_to\_change(v)$ for any $v \in V$. Without loss of generality, let v be the smallest node having $|v.L| = p$. So, if $A(v)^p \neq \emptyset$ then v is enabled by the Rule `[A]`, contradiction. In case where $A(v)^p = \emptyset$ (i.e. no neighbor points at only v), two situations are distinguished:

1. If $p = 1$ then we have $|v.L| = 1$ and $agreed(v) = false$ by assumption. Since `[U3]` is disabled at v, $v.inStar = agreed(v) = false$. Let $v.L = \{u\}$. Using Property (a), we have $u \in N(v)$. This situation means that v points at a neighbor u which may have two cases:

   (a) If $agreed(u) = true$ then $u.inStar = agreed(u) = true$ since `[U3]` is disabled at u. We have $v.L = \{u\}$ and $u.inStar = true$ and $agreed(v) = false$, then $v.L \neq C(v)^1$ because $u \notin C(v)$. This implies Rule `[I]` is enabled at v. Contradiction.

   (b) If $agreed(u) = false$ then $u.inStar = agreed(u) = false$ since `[U3]` is disabled at u. We have $v.L = \{u\}$ and $u.inStar = false$ and $agreed(v) = false$, then at least Rule `[A]` is enabled at u because $A(u)^1 \neq \emptyset$. Contradiction.

2. If $p \geqslant 2$ then we have $|v.L| \geqslant 2$ and $agreed(v) = false$ by assumption. Then $v.L \neq C(v)^1$ because $|v.L| \neq |C(v)^1|$. This implies that Rule `[I]` is enabled at v. Contradiction.

So, we proved that if $|v.L| = p$ then $agreed(v) = true$ for any integer $p \geqslant 1$. Furthermore, Rule [U3] is disabled for v and $agreed(v) = true$, then $v.inStar = agreed(v) = true$. This proves Property (c).

Considering Property (d). We have $|v.L| = p$ and using Property (c) and definition of $agreed(v)$, then $agreed(u) = true$ for any $u \in v.L$. Moreover, $u.inStar = agreed(u) = true$ because Rule [U3] is disabled for any $u \in v.L$. This proves Property (d). □

In the following lemma, we prove that SMSD always finds a maximal p-star decomposition in legitimate configuration.

Lemma 8.3.2 *In configuration with no enabled node, each node v having $|v.L| = p$ forms a p-star with $v.L$ as leaves. Moreover, the set of such p-stars forms a Maximal p-star Decomposition of the graph G.*

Proof. The proof is by induction on n.

Let $n \leqslant p$, i.e. G does not contains any p-star. Using Property (a) of Lemma 8.3.1 and $n \leqslant p$ then $|v.L| \leqslant 1$ for any $v \in V$. This implies that $v.inStar = agreed(v) = false$ for $\forall v \in V$ because Rule [U3] is disabled. Using Property (b) of Lemma 8.3.1 and $v.inStar = false$, implies that $v.leaf = S(v)$ for any $v \in V$. Hence, $n \leqslant p$ and $v.leaf = S(v)$ and $v.inStar = false$ for any $v \in V$, then $v.leaf = true$ for any $v \in V$. Furthermore, since Rule [I] is disabled and $|v.L| \leqslant 1$ for all nodes $v \in V$, then $v.L = \emptyset$ because $C(v)^1 = \emptyset$. Hence, in configuration where no node is enabled if $n \leqslant p$ then $v.L = \emptyset$ and $v.inStar = false$ for any $v \in V$.

Let $n > p$. First, consider that there is no node $v \in V : |v.L| = p$. This means that $|v.L| < p$ for any $v \in V$. Moreover, since Rules [U1-U4] are disabled and $|v.L| < p$ for any $v \in V$ then $v.inStar = agreed(v) = false$ for any $v \in V$. Since Rules [A],[I] are disabled, then $A(v)^p = \emptyset$ and $v.L = C(v)^1$ for any $v \in V$. This implies that $v.leaf = true$ for any $v \in V$. Hence, we deduce that G does not contain a p-star. Furthermore, since Rule [I] is disabled for $v \in V$, then $v.L = \emptyset$ because $C(v)^1 = \emptyset$ for any $v \in V$.

Second, consider that there exists a node $v \in V : |v.L| = p$. Let v be the smallest identifier such that $|v.L| = p$. We have $|v.L| = p$ and using Property (c) of Lemma 8.3.1, then $v.inStar = agreed(v) = true$. Moreover, using Property (d) of Lemma 8.3.1, we have $u.inStar = agreed(u) = true$ for any $u \in v.L$. Moreover, since Rule [I] is disabled for all nodes, in particular for any node $w \in V \setminus \{v \cup v.L\}$, we have $w.L \cap \{v \cup v.L\} = \emptyset$ (because $u.inStar = true$ for every $u \in \{v\} \cup v.L$ -Properties (c) and (d) of Lemma 8.3.1-). This means that no node $w \in V \setminus \{v \cup v.L\}$ points at $\{v \cup v.L\}$. Let G' be the graph induced by $V \setminus \{v \cup v.L\}$. Since no node of G' is enabled then the lemma holds for G' by induction. This implies that the Lemma is also true for the graph G. □

8.4 Convergence proof

In this section, we prove that the algorithm SMSD^2 converges within polynomial moves under the unfair distributed daemon. The following lemma shows that when a node executes an acceptation rule `[A]`, then it will never make a move again.

Lemma 8.4.1 *If a node v executes Rule `[A]` then it will never make a move again.*

Proof. Recall that only the Rules `[U1]`, `[A]` and `[I]` can change the pointer list of the node v. Furthermore, `[A]` is executed at v if Rules `[U1-U4]` are disabled.

Consider the Rule `[A]`. A node v executes the Rule `[A]`, meaning that there are at least p neighbors pointing at v to become a center node and v is the smallest node having $v.flag = true$ within its neighborhood. So, when the node v executes `[A]`, the value of $v.inStar$ is updated ($v.inStar = true$) and the node v chooses p smallest neighbors from nodes that pointed it, say $u_1, u_2, \ldots, u_p$, in order to form a p-star. Consider the node u_1 and the reasoning proof is the same for $u_2, \ldots, u_p$. Observe that during the time-step where v executes `[A]`, u_1 cannot change its $u_1.L$ by executing `[I]` or `[A]` because $smallest(u_1) = false$. Furthermore, u_1 cannot execute Rule `[U1]` because $incoherent(u_1) = false$. So, u_1 cannot change its pointer list $u_1.L$ during this time-step. Hence, after the acceptation move of v, we have $v.L = \{u_1, u_2, \ldots, u_p\}$ and $v.inStar = true$ and $u_1.L = \{v\}$. Then, the node u_1 has $agreed(u_1) = true$ and the only Rules that u_1 can execute are `[U2]` and `[U3]` without changing $u_1.L$. Observe that when u_1 excutes `[U3]` then u_1 will have $u_1.inStar = true$. So, u_1 will not execute neither `[A]` nor `[I]`. Thereby, v will never execute any rule again. □

We have to note that if a node v has $agreed(v) = true$ then it is either agreeing to be: (i) a *center node i.e* v points at p neighbors to be its leaves and every leaf of its list $v.L$ points at only v, formally $|v.L| = p$ and $\forall u \in v.L : u.L = v$ or (ii) a *leaf node* of some center node u if v points at only u and $v \in u.L$ and $|u.L| = p$ and $u.inStar = true$.

Thus, if v has $agreed(v) = true$ as a center node then any v's leaf u has $agreed(u) = true$. However, if v has $agreed(v) = true$ as a leaf node pointing at some neighbor u to be a center then $agreed(u)$ of the node u is not necessarily $true$. This is why we distinguish the two different situations in the proof of Lemma 8.4.2. The aim of this lemma is to prove that the maximum sequence of the value $inStar$ for each node in the graph G is bounded by a constant.

Lemma 8.4.2 *The sequence '$false \rightarrow true \rightarrow false \rightarrow true$' is the maximum possible sequence of $v.inStar$ for each node v during the execution of SMSD using an unfair distributed daemon.*

Proof. A node v updates its variable $v.inStar$ to $true$ if and only if $agreed(v) = true$. This means that the node v is either a center node of a star or a leaf node.

Assuming that the node v will be a center node. It is clear that the only rules that can change the value of $v.inStar$ from $false$ to $true$ are `[A]` and `[U3]`. We proved in Lemma 8.4.1 that the node v never makes a move after the execution of `[A]`. Moreover, a node v executes `[U3]` to be a center node if $agreed(v) = true$, this means that any node $u \in v.L$ has $u.L = \{v\}$. This situation can only exist in a starting configuration. So, when v updates its $v.inStar$ to $true$ then some nodes $u \in v.L$ may change their pointer list to invite or to accept other nodes. Hence, v may make wrong decision by executing `[U3]` and it will be not agreeing. However, the next updating of v to be a center node will be only by executing Rule `[A]`. Thus, the max sequence for a center node starting from $v.inStar = false$ is '$false \rightarrow true \rightarrow false \rightarrow true$'.

If a node v starts with $v.inStar = true$ and v will change its $v.inStar$ to $false$, this means that $agreed(v) = false$. This implies that the value $v.inStar$ is an initial value, not obtained by an execution of Rule `[A]`. Then $v.inStar$ will be set to $false$ by `[U1]` if $v.L$ is incoherent (*i.e.* $incoherent(v) = true$). Otherwise, v executes `[U3]`. In both cases, the next update of $v.inStar$ will be by executing the Rule `[A]` that makes v never move again (Lemma 8.4.1). Thus, the maximum sequence for a center node starting from $v.inStar = true$ is '$true \rightarrow false \rightarrow true$'.

Assuming that the node v will be a leaf node. It is useful to note that a leaf node v changes its value $v.inStar$ to $true$ (resp. $false$) if its center node u has already $u.inStar = true$ (resp. $false$). In other words, a leaf node stabilizes only if its center node has already stabilized. Consider that, initially, v has $v.inStar = false$, so in order to change its $v.inStar$ to true by executing the Rule `[U3]`, v must have $agreed(v) = true$. This means that in perspective of v, the node u is a center node because $|u.L| = p$ and $u.inStar = true$ and $v \in u.L$. In this situation, the node u can have two situations $agreed(u) = true$ or $agreed(u) = false$.

(i) if $agreed(u) = true$, means that $|u.L| = p$ and $u.inStar = true$ and $\forall v \in u.L : v.L = \{u\}$, then the node u is a real center node and keeps its $u.inStar = true$, and as consequence, any u's leaf node v will also keeping its value $v.inStar$ to $true$.

(ii) if $agreed(u) = false$, means that u is not a real center node. In this situation, the node v may take a wrong decision by executing `[U3]` to change its $v.inStar$ to $true$ because $agreed(v) = true$. But in this configuration, $agreed(u) = false$ and $u.inStar = true$. Then, u executes `[U3]` for updating $u.inStar$ to $false$. This pushes the leaf node v to change again its $v.inStar$ to $false$. After these executions, v may withdraw its invitation and invites another center node if u is not the smallest possible center (*i.e.* $\{u\} \neq C(v)^1$), else v keeps $v.L = \{u\}$. Thus, the next updating of $v.inStar$ must be definitive because the next invitation by executing Rule `[I]` allows v to invite only a node u with $u.inStar = false$ and $|u.L| \leqslant 1$ and only move to be a center node for u (creating p pointers and updating $inStar = true$) is the Rule `[A]`.

So, we conclude that for any node $v \in V$, then '$false \rightarrow true \rightarrow false \rightarrow true$' is the maximum sequence for $v.inStar$. □

Lemma 8.4.3 *Under the unfair distributed daemon, SMSD*^{2} *converges within* $O(\Delta^2 m)$

moves where Δ and m are respectively the maximum node degree and the number of edges in the graph G.

Proof. We denote by $|R|$ the number of executions of a Rule $[R]$ by one node. We give an upper bound for each $|R|$ of a node v.

- For the Rule `[U1]`: Note that any rule which modifies $v.L$, it provides a list of pointer such that $incoherent(v) = false$. So, $|U1| \leqslant 1$.
- For the Rule `[U2]`: It is clear that the variable $v.leaf$ depends only on the number of neighbors which have their variable $u.inStar = false$ and by Lemma 8.4.2, each neighbor u can change its variable $u.inStar$ at most 3 times. Thus, in the worst case, the $v.leaf$ can be changed at most $3d(v) + 1$. We deduce that $|U2| \leqslant 3d(v) + 1$ times.
- For the Rule `[U3]`: Using Lemma 8.4.2, each neighbor v can change its variable $v.inStar$ at most 3 times. So, $|U3| \leqslant 3$.
- For the Rule `[A]`: Using Lemma 8.4.1, node v can execute `[A]` at most once and never move again. So, $|A| \leqslant 1$.
- For the Rule `[I]`: The new value of $v.L$ depends on the smallest available center node defined by $C(v)^1$. Let be $u = C(v)^1$. So, $C(v)^1$ depends on the values of $u.inStar$ and $u.leaf$ and $|u.L|$. By the algorithm, the node u can have $|u.L| = p$ by executing only Rule `[A]`. Furthermore, u will have $v.inStar = true$ and it will never move again (Lemma 8.4.1). So, v can change its pointer at most once for each neighbor u when u executes `[A]`. Moreover, $C(v)^1$ also depends on $u.inStar$ and $u.leaf$. We have $u.inStar$ can changed at most 3 times (Lemma 8.4.2) and $u.leaf$ can changed at most $3d(u) + 1$ times. Then, $C(v)^1$ can change at most $3d(u) + 4$ times for each neighbor $u \in N(v)$. We deduce that for all neighbors of v, the value of $C(v)^1$ can change at most $d(v)(3\Delta + 4)$ times where Δ is the maximum node degree in G. We conclude that $|I| \leqslant d(v)(3\Delta + 4) + 1$.
- For the Rule `[U4]`: This rule depends on the predicate $want\_to\_change(v)$ which depends on $v.instar$, $A(v)^p$ and $C(v)^1$. We have $v.inStar$ can change at most 3 times (Lemma 8.4.2), $C(v)^1$ can change $d(v)(3\Delta + 4) + 1$ times (*cf.* previous result) and $A(v)^p$ can change $d(v)(\Delta(3\Delta + 4) + 1)$ times (because each v's neighbor u can make at most $d(u)(3\Delta + 4) + 1$ invitations or one acceptation). Then, we conclude that $|U4| \leqslant (3\Delta^2 + 7\Delta + 5)d(v) + 4$.

So, for any node v in the system, the number of executions of all rules is $\sum |R|$. Since $\sum_{v=1}^{n} d(v) = 2m$, where m is the number of edges and n the number of nodes, then we deduce that the complexity is $O(\Delta^2 m)$ moves. □

Theorem 8.4.4 *$SMSD^2$ is a self-stabilizing algorithm for computing a maximal p-star decomposition and stabilizes within $O(\Delta^2 m)$ moves under an unfair distributed daemon.*

Proof. By Lemma 8.3.2, the algorithm is correct and by Lemma 8.4.3, we conclude that SMSD^2 is self-stabilizing algorithm for maximal p-star decomposition and terminates in $O(\Delta^2 m)$ moves under the unfair distributed daemon for any connected graph. □

8.5 Summary

In this chapter, a second self-stabilizing algorithm for graph decomposition into disjoint p-stars (SMSD^2) is developed. The algorithm operates under the unfair distributed daemon and stabilizes in $O(\Delta^2 m)$ moves where m is the number of edges and Δ is maximum node degree in the graph. This complexity is also an upper bound for round complexity. Moreover, SMSD^2 considers all maximal p-star decomposition to be legitimate configurations.

8.6 Conclusion

In this part, we study the problem of maximal p-star decomposition of arbitrary graphs. This decomposition is a generalization of maximal matching problem in graphs. We showed that finding a deterministic self-stabilizing algorithm for such problem is impossible in anonymous graphs. Moreover, assuming unique locally distinct node identifiers, two self-stabilizing algorithms are developed for MSD problem.

A first algorithm, called SMSD^1, is presented in Chapter 7. The first algorithm operates under the unfair distributed daemon and stabilizes in $O(n)$ rounds. If $p = 1$ then SMSD^1 provides a maximal matching in graph. Furthermore, the time complexity in rounds of SMSD^1 is the same order as the best known self-stabilizing algorithm for maximal matching under the synchronous daemon [GHJS03c] or the distributed daemon [MMPT09]. Using the synchronous daemon, the algorithm SMSD^1 requires at most $O(n^2/p)$ moves. Moreover, we shown that for any starting configuration, SMSD^1 always leads to a unique configuration that is p-star decomposition.

Unfortunately, the simulations show that SMSD^1 may provide an exponential number of moves under the distributed daemon. Furthermore, SMSD^1 offers only a unique legitimate configuration which can be considered as a weak point of this algorithm. Then, a second algorithm, called SMSD^2, is presented in Chapter 8 in order to outperform the first one. The second algorithm SMSD^2 converges in polynomial moves and considers all maximal p-star decompositions to be legitimate.

Part III

Edge Monitoring Set (EMS)& Independent Strong Dominating Set (ISD-set)

Chapter 9

Introduction and motivation of part III

Contents

9.1 Introduction

Unlike wired networks, wireless networks have no predefined connection structure. Then, the communications between two resources (eg. sensors) provided with omni-directional antennas, are directly made when they are within communication range (neighbor), or through other intermediate resources. Moreover, these networks are obviously vulnerable in hostile environments.

Nowadays, wireless networks are often deployed in a random way by using a huge number of resources. To have high-level structures for the control and the monitoring of these resources, it is necessary to take into account the locality of the resources and the closeness between them. This can be made in an auto-organized way by awarding specific roles for certain resources of the network. For example, a set of resources can be selected to play the server role and the remaining resources play the client role, by being near a server.

Since it is natural to model a network by a graph, where resources and links are represented by nodes and edges of the graph respectively, several algorithms for graph parameters have been proposed in the literature for designing efficient protocols in wireless sensor and ad-hoc networks. For example, self-stabilizing algorithms for finding minimal dominating sets, maximal matchings, independent sets (see Section 2.4 for more details).

In this third part of the thesis, we focus on two variants of dominating sets: *Edge Monitoring Sets* and *Independent Strong Dominating Sets*. The goal of these parameters is the selection of certain nodes (a.k.a. dominants or monitors) for dominating some nodes or edges.

Edge Monitoring Sets is a simple and effective mechanism for the security of wireless networks, especially to cope with compromised nodes in wireless sensors

networks (WSNs). A node v can monitor (or dominate) an edge e if both end-nodes of e are neighbors of v; *i.e.*, e and v together form a triangle in the graph. Moreover, some edges need more than one monitor. Finding a set of monitoring nodes satisfying all monitoring constraints is called the edge-monitoring problem. The minimum edge-monitoring problem is known to be NP-complete [DLL08, DLL+11]. In this part, we present a novel polynomial self-stabilizing algorithm for computing a minimal edge-monitoring set which operates under the unfair distributed daemon. More details can be found in Chapter 10.

The second parameter, called Independent Strong Dominating Sets (ISD-set), is an interesting variant of dominating sets. In addition to its domination and independence properties, the ISD-set considers also nodes degrees that make it very useful in practical applications. This variant was introduced by Sampathkumar et al. in [SL96]. In this part, we propose the first self-stabilizing algorithm for computing an ISD-set of an arbitrary graph that operates under the distributed daemon. Moreover, performed simulations and comparisons with well-known self-stabilizing algorithms for dominating sets and Independent sets problems showed the efficiency of the proposed algorithm for ISD-set. More details can be found in Chapter 11.

9.2 Overview and definitions

We consider networks in which all communications are bidirectional. We model the network by a graph G where resources (eg. sensors) are represented by nodes, defined by the set V and their communications by edges, defined by the set E. Recall that we denote by n and m the number of nodes and edges in G respectively.

Definition 11 (A monitor) *Given an edge $e = \langle u, w\rangle$, a node v can* monitor *e, if $\langle u, v\rangle$, $\langle v, w\rangle \in E$,* i.e. *the three nodes v, u, w form a triangle in G.*

Let $\omega(e)$ be the weight of the edge $e \in E$. This weight describes the number of nodes that are supposed to monitor e. The set of edges that have to be monitored is denoted by E_s. Formally, $E_s = \{e \in E | \omega(e) > 0\}$.

Definition 12 (A minimal Edge Monitoring problem) *Minimal Edge Monitoring problem consists in identifying a minimal set of nodes D that are able to monitor a given subset of edges E_s of the global edges E.*

Definition 13 (k-monitoring) *A set of nodes $D \subseteq V$ is* k-monitoring *of a set of edges $E_s \subseteq E$ if all edges of E_s are monitored by at least k different nodes in D. A k-monitoring D of E_s is* minimal *if no subset of D is k-monitoring of E_s.*

Definition 14 (A Dominating Set) *A set of nodes D is a dominating set (DS) of G if every node $v \in V - D$ has a neighbor in D. D is a minimal dominating set (MDS) if any of its proper subsets is not a dominating set.*

Definition 15 (An Independent Set) *A set of nodes D is an independent set (IS) of G if $\forall u, v \in D, (u, v) \notin E$. D is a maximal independent set (MIS) if D is also a dominating set.*

Definition 16 (Strong & Weak domination) *Let $d(v)$ be the degree of v. A node v strongly dominates a node u and u weakly dominates v if $(u, v) \in E$ and $d(v) \geqslant d(u)$. We say that v is stronger than u.*

Definition 17 (An Independent Strong Dominating Set) *A set $D \subseteq V$ is an independent strong dominating set (ISD-set) of G if D is an independent set (IS) and every node in $V - D$ is strongly dominated by at least one node in D.*

By the last definition, observe that any ISD-set is at the same time a minimal dominating set and a maximal independent set. Hence, the minimality and the maximality are implicite in this case.

9.3 Motivation

In wireless networks with randomly deployed sensor nodes, the selection of a minimal monitoring set of nodes is a challenging task, especially for large scale networks using only distance-one knowledge. Consider for example the deployment in Figure 9.1. The black nodes can monitor all communication links depicted in bold. In [DLL+11, DLL08], *Dong et al.* proved that finding a minimum set of monitoring nodes is NP-complete. The authors also proposed two distributed polynomial algorithms with provable approximation ratio. However, the algorithms assume a synchronous model and distance-two knowledge. Furthermore, distance-two knowledge is not a realistic solution in WSN. In Chapter 10, we assume the most general model that is asynchronous model with distance-one knowledge.

To the best of our knowledge, there is only one work proposed by Hauck in [Hau12] where the author presented the first self-stabilizing algorithm for the edge monitoring problem. His algorithm uses the expression model [Tur12] and converges in $O(n^2)$ moves under the central daemon. Using the transformer proposed by Turau in [Tur12], the transformed algorithm converges in $O(mn^2)$ moves under the unfair distributed daemon.

In this thesis, we improve the previous work by proposing a novel algorithm that operates under the distributed daemon without using any transformer as it the case with Hauck's work. Moreover, our algorithm converges in $O(\Delta^2 m)$ moves where Δ is the maximum node degree in graph. Thus, in particular for networks with low maximal node degree our algorithm converges much faster. This led us to considerate the first parameter (Edge Monitoring Set Problem). This work is published in [NHTK14].

Regarding what led us to consider the second parameter (ISD-set), this choice is justified by several points. The first one is essentially related on the property of this parameter which constitutes a combination of Minimal Dominating set (MDS)

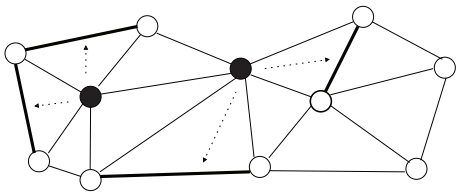


Figure 9.1: Edge monitoring of a graph. The black nodes can monitor the bold communication links.

and Maximal Independent Set (MIS). These two parameters have been extensively studied on both theoretical and practical (algorithmic) aspects. Therefore, several self-stabilizing algorithms have been proposed for MDS and MIS problems (see the survey in section 2.4), however no one considers the degree of nodes and the distance between dominating nodes at the same time. Hence, the nodes belonging to the dominating set can be in the same neighborhood and some nodes can be dominated by neighbors that have smaller degrees. Figure 9.2 illustrates a possible configuration of a Minimal dominating set (MDS) and an ISD-set in a star graph, composed of one center node and four leaves. The white and black nodes denote the dominated and the dominating nodes respectively. So, we note that an MDS may select the leaf nodes as dominating nodes and the center node as dominated (Figure 9.2(a)). However, ISD-set problem provides only the center node as dominating node (Figure 9.2(b)). So, we note that ISD-set is more suitable than MDS to reduce the cardinality of the dominating set. Therefore, ISD-set combines two properties and advantages of MDS and MIS, that make ISD-set a very convenient approach for a better network covering with minimum number of nodes in large-scale networks.

(a) MDS (b) ISD-set

Figure 9.2: MDS vs ISD-set.

Furthermore, the existing self-stabilizing algorithms for MDS and MIS has different complexities (steps, rounds, moves) and there is no simulation comparison of these algorithms. In this part, we propose the first self-stabilizing algorithm for computing an ISD-set in an arbitrary graph. Moreover, performed simulations and comparisons with well-known self-stabilizing algorithms for MDS and MIS problems showed the efficiency of the proposed algorithm for ISD-set. More descriptions of this contribution can be found Chapter 11.

Chapter 10

Algorithm for EMS problem

Contents

10.1 Introduction

A sensor network is a wireless ad-hoc network with a large number of nodes that are micro-sensors to collect and transmit environmental data autonomously. Usually, the deployment of these sensors is done in a random manner. These networks find many applications such as military surveillance (detection intrusion, weapons location and vehicles), forest fire control, industrial process control, machine health monitoring, and so on.

The power limitation in wireless sensor networks (WSN) and hostile environments in which they could be deployed are factors that make this type of networks very vulnerable. Since, the security of these networks is very important, especially for sensitive and critical applications.

One of the most difficult threats in WSN is compromised nodes. Several attacks may use the compromised nodes to divert the proper functioning of the networks. Considering the real challenges to design security mechanisms against these attacks, many approaches have been proposed based on local monitoring technique (*a.k.a* watchdog) [KBNR05, KBS05, LC06, GBS08].

The basic idea of local monitoring is assigning monitoring role to some sensors in the network [MGLB00]. Usually, these monitors are placed in the middle of communication between the sender (S) and the receiver nodes (R). Figure 10.1 illustrates the case where the nodes $M1$ and $M2$ monitor the communication between nodes S and R. We have to note that both of $M1$ and $M2$ are located in the transmission range of nodes S and R.

The sensors are deployed in random and dense manner making the selection of minimal monitor nodes harder, especially for large scale WSN and using only 1-hop neighborhood knowledge. For example, as shown in Figure 10.2, the black

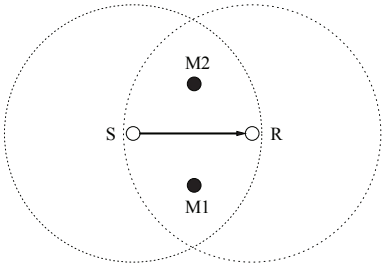


Figure 10.1: Local monitoring.

nodes denote the monitors and the depicted edges denote the communications that must be monitored, *i.e.* there exists at least one monitor node for each depicted edge. In [DLL08, DLL+11], *Dong et al.* proved that finding minimum nodes for such problem is NP-complete and they proposed two distributed polynomial algorithms for provable approximation ratio to this issue. Their algorithms operate in synchronous model and assume distance-two knowledge.

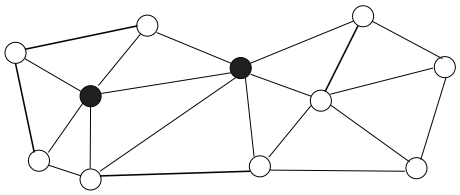


Figure 10.2: Edge monitoring set of a graph. The black nodes are the monitors of the depicted edges.

Using transient fault-tolerance aspect of self-stabilizing systems, Hauck developed the first self-stabilizing algorithm for computing a minimal set for edge monitoring problem [Hau12]. His algorithm uses the expression model defined by Turau in [Tur12] and converges in $O(n^2)$ moves under the unfair central daemon. Using the transformer in [Tur12], the transformed Hauck's algorithm converges in $O(mn^2)$ moves under the unfair distributed daemon. In this thesis, we propose a new self-stabilizing algorithm for edge monitoring problem (called *SEMS*) with a lower complexity than Hauck's algorithm. Moreover, our algorithm assumes distance-one knowledge, *i.e.* each node has local view and knows only the adjacent edges. All algorithms presented for edge-monitoring problem are summarized in Table 10.1.

Ref.	**Dist. know.**	**Com. model**	**Self-stab.**	**Comp.**	**Transformer.**
[DLL+11]	Distance-two	Synchronous	No	$O(\Delta)$	Yes
[Hau12]	Expression model	Asynchronous	Yes	$O(n^2m)$	Yes
SEMS	Distance-one	Asynchronous	Yes	$O(\Delta^2m)$	No

Table 10.1: Distributed algorithms for edge-monitoring problem.

The following section presents the self-stabilizing algorithm *SEMS* for computing a minimal edge-monitoring set for a general graph G with edge weight function ω

as introduced above (see Section 9.2). In this algorithm, each node v maintains a variable *state* with range $\{In, Wait, Out\}$. This variable indicates whether v belongs to the monitoring set or not. A node is called a *monitor* if its variable *state* has value IN. Thus, the edge-monitoring set D of G is defined by $D = \{v \in V : v.state = In\}$. The state $Wait$ is an intermediate state from state In to Out required for symmetry breaking. It is used to inform neighbors that this node is not required to be a monitor and can change its state to Out.

10.2 Algorithm description

The monitors of an edge are administered by the end node with the smaller identifier. Neighbors of v that are either monitors or potential monitors of an edge adjacent to v are called *target monitors*. Thus, a node v maintains a set of target monitors for each of its adjacent edges which it is responsible for. For an edge $\langle v, u\rangle$, this includes all current monitors, *i.e.*, all common neighbors of v and u with state In or $Wait$. If the number of these nodes is not sufficient (*i.e.*, less than $\omega(v, u)$) then this set is supplemented by the smallest common neighbors of v and u with state Out until this set has $\omega(v, u)$ elements. If on the other hand the number of these nodes exceeds $\omega(v, u)$ then the set of target monitors is empty. Thus, the edge does not need this node as a monitor. The union of target monitors of all adjacent edges of a responsible node is called the *"target monitoring set"* of the node.

Note that there is one small drawback with the following notion: A node does not know the set of neighbors for each of its neighbors. This information is necessary to compute the target monitoring set of a node. However, a node can avoid this pitfall by exposing the set of neighbors in a variable and neighbors can use this variable for their computations. Since this variable can be corrupted by a transient fault, the target monitoring set may be faulty for some time.

The algorithm *SEMS* works as follows. Nodes keep a target monitoring set as well as the exposed set of neighbors always up-to-date. A node with state In that is not a target monitor for any of its neighbors will change its state. In order to avoid an oscillating behavior such a node does not immediately change its state to Out. It first transits into state $Wait$. In order to transit into state Out, all neighbors must give permission to do such transition. A node only gives this permission to the neighbor with state $Wait$ that has the smallest identifier among these nodes. This is realized by a public variable containing the identifier of the neighbor that can be removed from its monitoring set. So, only after all neighbors give this permission, a node may transit from state $Wait$ to state Out. If a node with state $Wait$ becomes a member of the target monitoring set of a neighbor then it transits back to state In. There is also a rule for changing the state from Out to In. The precondition for this rule is that the node is a target monitor of a neighbor and none of its neighbors is currently giving this node the above discussed permission.

Technically, the algorithm *SEMS* uses the following variables for each node v:

- S :: contains the open neighborhood of v.

- TM :: the set of target monitors. It is a set of neighbors that are either monitors or potential monitors of an edge adjacent to v. TM will contain a sufficient number of nodes to satisfy the monitor demands of all adjacent edges. Note that $|TM| \leq \Delta$.
- PO :: used to give permissions to change state to Out. It either contains the smallest identifier of all neighbors in state $Wait$ not contained in TM or $null$.

If $v.PO = u$ (resp. $u \in v.TM$) then we say v points at u to leave (resp. to enter) the monitoring set.

For a set X of node identifiers and a positive integer p denote by X^p the set of the p smallest identifiers contained in X. If $|X| \leq p$ then $X^p = X$. Note that this definition is slightly different to those used in previous chapters. Formally, the operator X^p of this algorithm is defined as follows:

$$X^p = \begin{cases} X & \text{if } |X| \leq p \\ \text{the } p \text{ smallest elements of } X & \text{otherwise.} \end{cases}$$

In algorithm *SEMS* a node v uses the three functions $Mon(v,u)$, $Candidate(v,u)$, and $TM_e(v,u)$, defined for all neighboring nodes $v, u \in V$. Function $Mon(v,u)$ returns the set of nodes that are supposingly monitoring edge $\langle v,u \rangle$. These are neighbors of v and most likely also of u that have state In or $Wait$. Formally,

$$Mon(v,u) = \{z \in N(v) \cap u.S \mid z.state = In \vee z.state = Wait\}$$

Function $Candidate(v,u)$ returns the set of nodes that are supposingly new candidates to monitor edge $\langle v,u \rangle$. These are neighbors of v and most likely also of u that have state Out. Formally,

$$Candidate(v,u) = \{z \in N(v) \cap u.S \mid z.state = Out\}$$

Function $TM_e(v,u)$ uses the first two functions to compute a target set of monitors for edge $\langle v,u \rangle$. It is used to keep $v.TM$ up-to-date. Formally,

if $(|Mon(v,u)| \leqslant \omega(v,u) \wedge v < u)$ **then**
    $TM_e(v,u) = Mon(v,u) \cup Candidate(v,u)^{\omega(v,u)-|Mon(v,u)|}$;
else
    $TM_e(v,u) = \emptyset$;

Note that $TM_e(v,u) = \emptyset$ for an edge $\langle v,u \rangle$ if $v > u$.

Algorithm *SEMS* is specified by six rules that are divided into two categories. Rules `[R1]` and `[R2]` belong to the first category. They are used to update the values of the variables TM and PO.

The remaining four rules of the second category maintain variable *state*. If more than one rule is enabled, we assume that the rule with the smallest number is executed.

Algorithm 6: Algorithm *SEMS*: Maintaining TM, PO and S

Nodes: v is the current node

$S \neq N(v) \quad \longrightarrow S := N(v);$ [R1]

$TM \neq \bigcup_{u \in N(v)} TM_e(v,u) \vee PO \neq min\{u \in N(v) \mid u.state = Wait \wedge u \notin TM\} \quad \longrightarrow$
$TM := \bigcup_{u \in N(v)} TM_e(v,u);$
$PO := min\{u \in N(v) \mid u.state = Wait \wedge u \notin TM\}$; [R2]

Algorithm *SEMS*: Maintaining *state*

Nodes: v is the current node

$state = Out \wedge \exists u \in N(v) : v \in u.TM \wedge \forall w \in N(v) : v \neq w.PO$
$\longrightarrow state := In;$ [R3]

$state = In \wedge \forall u \in N(v) : v \notin u.TM \quad \longrightarrow state := Wait;$ [R4]

$state = Wait \wedge \exists u \in N(v) : v \in u.TM \quad \longrightarrow state := In;$ [R5]

$state = Wait \wedge \forall u \in N(v) : v = u.PO \quad \longrightarrow state := Out;$ [R6]

Figure 10.3 shows an execution of Algorithm *SEMS* under the synchronous daemon for a graph with six nodes. Two of the edges require each one a monitor. In the initial configuration, all nodes are in state *Out* and the values of variable S are consistent with the neighborhood relation. Furthermore, we assume $v.TM = \emptyset$ and $v.PO = null$ for each node v.

10.3 Correctness proof

First, we prove that in a configuration where no node is enabled, the set D forms a minimal edge monitoring set with respect to ω.

Lemma 10.3.1 *In a configuration with no enabled node, the following properties hold for each $v \in V$.*

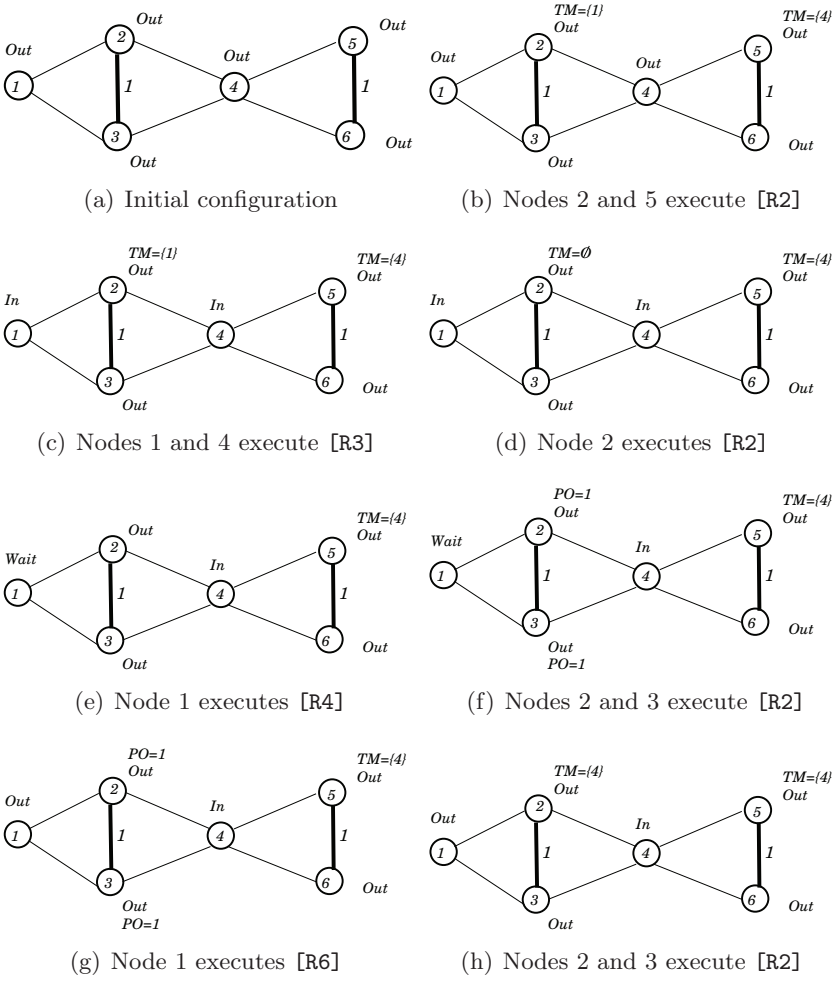
(a) $v.S = N(v)$,

(b) if $v.state = Wait$ *then* $v \notin u.TM$ *for all* $u \in N(v)$,

(c) if $v.state = Out$ *then* $v \neq u.PO$ *for all* $u \in N(v)$,

(d) $v.state \in \{In, Out\}$.

Proof. Properties (a) and (b) are satisfied because rules [R1] and [R5] are disabled. Note that $v.PO = \{u \in N(v) : u.state = Wait \wedge u \notin v.TM\}$ since rule [R2] is disabled for each node $v \in V$. Thus, $u.PO = null$ or $u.PO.state = Wait$. Hence, $v \neq u.PO$ since $v.state = Out$. This proves property (c).



(a) Initial configuration (b) Nodes 2 and 5 execute [R2]

(c) Nodes 1 and 4 execute [R3] (d) Node 2 executes [R2]

(e) Node 1 executes [R4] (f) Nodes 2 and 3 execute [R2]

(g) Node 1 executes [R6] (h) Nodes 2 and 3 execute [R2]

Figure 10.3: Example of an execution of Algorithm *SEMS*

Assume Property (d) is false. Among all nodes violating this property choose a node v with a minimal identifier. Then $v.state = Wait$. By minimality of v, if $v \notin u.TM$ for a node $u \in N(v)$ then $v = u.PO$. Since rule [R6] is disabled there exists a node $u \in N(v)$ such that $v \neq u.PO$. Hence, $v \in u.TM$ and rule [R5] is enabled. Contradiction. □

Lemma 10.3.2 *In a configuration with no enabled node any edge has sufficiently many monitors,* i.e. $|Mon(v,u)| \geqslant \omega(v,u)$ *for each* $\langle v,u\rangle \in E$.

Proof. The proof is by contradiction. Assume that there exists an edge $\langle v,u\rangle$ such that $|Mon(v,u)| < \omega(v,u)$. Without loss of generality, let $v < u$. By definition, $Mon(v,u) = \{z \in N(v) \cap u.S \mid z.state \in \{In, Wait\}\}$. Using properties (d) and (a) of Lemma 10.3.1, we have

$$Mon(v,u) = \{z \in N(v) \cap N(u) \mid z.state = In\}.$$

Since $|Mon(v,u)| < \omega(v,u)$ the set have $Candidate(v,u)^{\omega(v,u)-|Mon(v,u)|}$ is not empty (otherwise no solution would exist). Moreover, since rule [R2] is disabled for v the following holds:

$$\emptyset \neq Candidate(v,u)^{\omega(v,u)-|Mon(v,u)|} \subseteq TM_e(v,u) \subseteq v.TM$$

This shows that there exists a node $z \in v.TM$ with $z.state = Out$. Also $z \neq w.PO$ for all $w \in N(z)$ by property (c) of Lemma 10.3.1. This yields that rule [R3] is enabled for node z. Contradiction. □

Lemma 10.3.3 *In a configuration with no enabled node, the set* $D = \{v \in V \mid state(v) = In\}$ *forms a minimal edge-monitoring set with respect to* ω.

Proof. According to Lemma 10.3.2, D is an edge-monitoring set. Thus, it is sufficient to prove that D is minimal. Assume there exists a node $v \in D$ such that $D' = D - \{v\}$ is an edge monitoring set of G with respect to ω (see Figure 10.4 for an example). So $v.state = In$. Then for any pair $u_1, u_2 \in N(v)$ with $u_1 < u_2$ edge $\langle u_1, u_2 \rangle$ has more than $\omega(u_1, u_2)$ monitors, *i.e.* $|Mon(u_1,u_2)| > \omega(u_1,u_2)$. Thus, $TM_e(u_1,u_2) = TM_e(u_2,u_1) = \emptyset$. Now, $v \notin u_1.TM$ and $v \notin u_2.TM$ since rule [R2] is disabled for u_1 and u_2. Let $u_1 \in N(v)$ such that $N(u_1) \cap N(v) = \emptyset$. Then $v \notin u_1.TM$ by definition of $u_1.TM$ (note rules [R1] and [R2] are not enabled). Hence, $v \notin u.TM$ for any $u \in N(v)$. This implies that rule [R4] is enabled for v. Contradiction. □

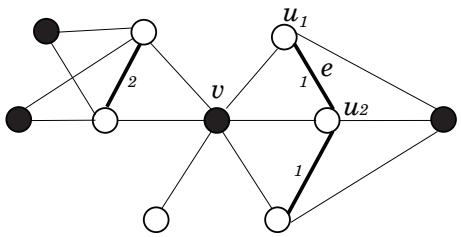


Figure 10.4: Non-minimal edge-monitoring set. Monitoring nodes are depicted in bold and the edge labels denote ω. Node v is not needed as a monitor.

10.4 Convergence proof

In the previous section, we proved the correctness of SEMS algorithm. Then, it remains to prove that *SEMS* stabilizes in finite time for any starting configuration under the unfair distributed daemon. Figure 10.5 shows all transitions of a node with respect to variable *state* that can occur during an execution of Algorithm *SEMS*.

Observe that nodes do not enter or leave the set TM if they change their state from $Wait$ to In or conversely.

Recall that the following lemma follows from the convention that rules with a higher priority have precedence.

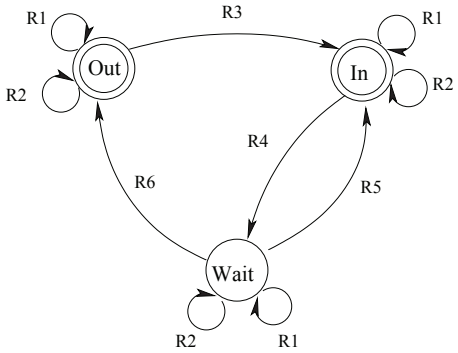


Figure 10.5: State Transition Diagram of Algorithm *SEMS*

Lemma 10.4.1 *Each node executes rule* `[R1]` *at most once. If a node does execute* `[R1]` *then in its first move.*

This lemma implies that if a node v executes rules `[R2]` to `[R6]` then $v.S = N(v)$.

A node v can change its state from In via $Wait$ to Out because neighboring nodes signal to v that all their edges are sufficiently monitored. This information can be false because some neighbor u of v wrongly assumed that its neighbor u_1 could monitor edge $\langle u, u_2\rangle$. The reason for such a wrong assumption is that $u_2 \in u_1.S$ but $u_2 \notin N(u_1)$. Once u_1 executes rule `[R1]` node u will realize this and u can now consider v as a target monitor and include it into $u.TM$. This could then prompt v to change its state to In again. Now the situation is different, all neighbors of v have executed a rule in the mean time. Because of priority of rules then $u.S = N(u)$ holds for all $u \in N(v)$. If node v changes its state again to Out with rule `[R6]` then it is because all neighbors indicated with their variable PO that their edges have a sufficient number of monitors without v. Since this number never will fall again under the value given by ω, node v will never move to state In again. This behavior is formally proved in the following two lemmas.

Lemma 10.4.2 *Each node executes* `[R6]` *at most twice,* i.e. *it changes from state Wait to state Out at most twice.*

Proof. Let c be a configuration in which a node $v \in V$ has state *Wait* and executes rule `[R6]`. For v to execute rule `[R6]` again it must first change its state back to *Wait*. This can only be achieved by first changing to state *In* with rule `[R3]` and then to state *Wait* with rule `[R4]`. Note that $v = u.PO$ for all $u \in N(v)$ when v executed rule `[R6]`. For v to be enabled for rule `[R3]` it is required that $v \neq u.PO$ for all $u \in N(v)$. Thus, all neighbors of v must have executed rule `[R2]` before v can execute rule `[R3]` again. A node executing rule `[R2]` cannot be enabled for rule `[R1]`. Thus, each neighbor u of v satisfies $u.S = N(u)$ when u executes rule `[R2]`. Hence, those neighbors of v that are responsible for edges that v can monitor have all finally determined that v is not required as a monitor, *i.e.* v will never enter $u.TM$ for a neighbor u. Hence v will never change its state to In again. □

Lemma 10.4.3 *Each node executes* `[R3]` *at most three times,* i.e. *it changes from state Out to state In at most three times.*

Proof. A node executing rule `[R3]` four times would execute rule `[R6]` at least three times. This contradicts Lemma 10.4.2.

□

Lemma 10.4.4 *Each node executes* `[R4]` *at most* $6\Delta d(v)$ *times,* i.e. *it changes from state In to state Wait at most* $6\Delta d(v)$ *times.*

Proof. A node v with state In executes rule `[R4]` if v is not a target monitor of any of its neighbors, *i.e.* $v \notin u.TM$ for all $u \in N(v)$. In order to reenter state In at least one of v's neighbors must declare v as a target monitor, *i.e.* there must be a node $u \in N(v)$ with $v \in u.TM$. Note that for u to change its set of target monitors, a neighbor of u must change its state from Out to In or from $Wait$ to Out or execute rule `[R1]`. According to Lemmas 10.4.1 to 10.4.3, each neighbor of u can do this at most 6 times. Hence, node u can update $u.TM$ at most $6d(u)$ times. This implies that node v changes its state to $Wait$ at most $6\Delta d(v)$. □

Lemma 10.4.5 *Each node executes* `[R5]` *at most* $6\Delta d(v)+1$ *times,* i.e. *it changes from state Wait to state In at most* $6\Delta d(v) + 1$ *times.*

Proof. By Lemma 10.4.4, a node v can change its state from In to $Wait$ at most $6\Delta d(v)$ and using State Transition Diagram of v, then v can change from state $Wait$ to state In at most $6\Delta d(v) + 1$ times. □

Lemma 10.4.6 *Any node v can execute* `[R2]` *at most* $(6\Delta^2 + 9)d(v)$ *times.*

Proof. Consider a node v. The execution of rule `[R2]` depends on the values of $v.TM$ and $v.PO$. By definition, the value of $v.TM$ itself depends on $TM_e(v, u)$ for each neighbor u of v. $Mon(v, u)$ depends on the neighbors w of v which are in state *Wait* or *In*. Note that node w can change its value from state *Out* to *Wait* at most three times (Lemma 10.4.3) and from state *Wait* to *Out* at most twice (Lemma 10.4.2). Thus, each neighbor w of v changes $Mon(v, u)$ at most five times and once if $w.S$ is incorrect. So, for each of v's neighbor u, $TM_e(v, u)$ can change at most 6 times. Hence, we deduce that $v.TM$ can change at most $6d(v)$ times for each neighbor of v.

Next we consider variable $v.PO$. By definition, PO depends on the neighbors that have state *Wait*. Using Lemmas 10.4.2 and 10.4.5, each neighbor u of v changes its state from *Wait* to state *In* or *Out* at most $6\Delta d(u) + 3$ times. Thus, for each neighbor of v, the value of $v.PO$ can change at most $d(v)(6\Delta^2 + 3)$ times.

In summary, v can execute rule `[R2]` at most $d(v)(6\Delta^2 + 9)$ times. □

Lemma 10.4.7 *Algorithm* SEMS *terminates in* $O(\Delta^2 m)$ *moves under the unfair distributed daemon.*

Proof. Lemmas 10.4.1 to 10.4.6 stated upper bounds on the number of executions for each rule on each node. In the worst case these moves all occur sequentially. This gives the following upper bound for the total number of moves:

$$n + \sum_{v \in V}(6\Delta^2 + 9)d(v) + 3n + \sum_{v \in V} 6\Delta d(v) + \sum_{v \in V}(6\Delta d(v) + 1) + 2n \in O(\Delta^2 m)$$

□

Theorem 10.4.8 *Algorithm* SEMS *is self-stabilizing algorithm for finding a minimal edge monitoring set for a given set of monitoring requirements of a general graph. It uses $O(\Delta log\ n)$ memory space per node and stabilizes in $O(\Delta^2 m)$ moves under the unfair distributed daemon.*

Proof. Theorem 10.4.8 is a direct consequence of Lemmas 10.3.3 and 10.4.7. □

10.5 Summary

In this chapter, we presented a new self-stabilizing algorithm to find minimal edge-monitoring sets in general graphs. The presented algorithm *SEMS* converges in $O(\Delta^2 m)$ moves under the unfair distributed daemon and assumes the most general model (Distance-one Knowklege). Consequently, this result improves Hauck's work [Hau12] by proposing a lower move complexity without using any transformer.

Chapter 11

Algorithm for ISD-set problem

Contents

11.1 Introduction

Dominating sets and Independent sets are very important class of problems with several theoretical and practical applications. These problems have attracted many theoretical researches, therefore many results have been proposed and different variants have been identified by the graph community. In practical side, the dominating sets (DS) and independent sets (IS) gave a special interest to distributed systems field due to their importance for several applications. The structure of DS and IS can be useful as virtual overlays in computer networks. These structures are often used for designing efficient protocols in wireless sensor and ad-hoc networks [GHJ+08, UT11, YKR06, BDTC05, AWF03, KMW04], for example clustering approaches in wireless sensor networks for load balancing and extending the network lifetime [YKR06]. A survey of different node clustering approaches can be found in [YKR06].

Usually, the nodes having higher degrees in graphs play important roles for clustering in wireless networks [YKR06], for providing stable cluster structures [KMW04] and for studying communities structure in p2p networks [LHK13], while the majority of the distributed algorithms for minimal dominating set (MDS) and maximal independent set (MIS) problems do not consider this aspect. This problem has been studied in graph theory and it is called *strong* and *weak domination*. These concepts were introduced by Sampathkumar et al. in [SL96]. Moreover, finding the

minimum *independent strong dominating set* (ISD-set) is NP-hard even in bipartite graphs [DHMU02].

Given a graph $G = (V, E)$, a set $D \subseteq V$ is an independent set (IS) if no two nodes of D are neighbors (adjacent). let $d(v)$ be the degree of v in graph G. A node v strongly dominates a node u and u weakly dominates v if $uv \in E$ and $d(v) \geqslant d(u)$. A set $D \subseteq V$ is an ISD-set of G if D is an independent set and every node in $V - D$ is strongly dominated by at least one node in D.

In this chapter, we propose the first self-stabilizing algorithm for a minimal ISD-set. The algorithm operates under the unfair distributed daemon.

11.2 Algorithm description

This section describes the self-stabilizing algorithm for computing a minimal ISD-set in general graphs (called *ISDS*). We assume that every node $v \in V$ has a distinct local identifier denoted by id.

The approach of *ISDS* is based on greedy approach. The general idea of this algorithm is as follows: A node v becomes a dominating node if there is no dominating neighbor that is stronger than v. In other words, the node v with the largest both degree and id becomes a dominating node. Thereby, all of its neighbors will be strongly dominated. This procedure is recursively repeated for the sub-graph of G consisting of all nodes except v and its neighbors.

In algorithm *ISDS*, each node v maintains a variable $state$ with range $\{In, Out\}$. This Boolean variable indicates whether v belongs to the strong dominating set or not. Thus, the ISD-set is defined by $D = \{v \in V : v.state = In\}$. The algorithm uses a second variable d that is supposed to contain the degree of a node in the graph G. We define a lexicographical strong order between nodes of G, denoted by $\succ$, that considers their degrees and ids. Thus, the nodes are first ranked by their degree variable and if two nodes have the same degree variable, they are ranked by the highest id. Then, we say that v is stronger than u, denoted by $v \succ u$, if $v.d > u.d$ or $v.d = u.d \wedge v.id > u.id$.

In addition to the two variables $v.state$ and $v.d$ for each node $v \in V$, ISDS uses a local function, denoted by $I(v)$. The latter permits to compute the new value of $v.state$ as follows:

$$I(v) = \begin{cases} Out & \text{if } \exists u \in N(v) : u \succ v \wedge u.state = In \\ In & \text{otherwise.} \end{cases}$$

The proposed Algorithm *ISDS* is composed of two rules $[R1]$ and $[R2]$. The first rule $[R1]$ permits to update the variable d and reset $state$ to Out if d is not up-to-date (*i.e.* d is not equal to the true degree of the concerned node). The second rule $[R2]$ updates the variable $v.state$. We assume that there is an order between rules, i.e. if the two rules are enabled, we assume that the rule with the smallest number is executed. The details of these rules are presented in the following algorithm.

Algorithm 8: Self-stabilizing algorithm for ISD-set (ISDS)

Nodes: v is the current node

$v.d \neq d(v) \quad \longrightarrow v.d := d(v);\ v.state := Out;$ [R1]

$v.state \neq I(v) \quad \longrightarrow\ v.state := I(v);$ [R2]

Note that the definition of the lexicographical strong order $\succ$, can easily be generalized for weighted graphs where the weight of any node v represents its importance in G. A weight of a node can be its degree, its remaining energy or other network parameters. Thereby, the ISDS rules still valid for such generalization to weighted graphs.

11.3 Correctness proof

First, we prove that in a configuration where no node is enabled, the set $D = \{v \in V, v.state = In\}$ forms an ISD-set.

Lemma 11.3.1 *In a configuration where no enabled node, the following properties hold:*

(a) any node $v \in V$ has $v.d = d(v)$;

(b) any node $v \in V - D$ is strongly dominated by a node $u \in D$, i.e. *$u \succ v$ and $u.state = In$;*

(c) there are no two adjacent nodes in D.

Proof.
Property (a) is ensured by Rule $[R1]$.
Property (b) is proved by contradiction. Assume that there exists a node $v \in V - D$ which is not strongly dominated by any neighbor $u \in D$, this means that there is no neighbor u such that $u \succ v$ and $u.state = In$. In this case, we have $I(v) = In$ and $v.state = Out$ and using property (a) $v.d = d(v)$. Then $[R2]$ is enabled at v. Contradiction.
Property (c) is proved by contradiction. Assume that there exist two nodes $v, u \in D$ such that $u \in N(v)$. By definition, $v, u \in D$ means that $v.state = u.state = In$. Without loss of generality, let $u \succ v$ then $I(v) = Out$. This implies that $I(v) = Out$ and $v.state = In$. Then, $[R2]$ is enabled at v. Contradiction. □

Lemma 11.3.2 *In configuration where no enabled node, the set $D = \{v \in V, v.state = In\}$ is an independent strong dominating set of the graph G.*

Proof. Using property (b) of Lemma 11.3.1, we have $\forall v \in V - D$ is strongly dominated by a node $u \in D$. Moreover, using property (c) of the same lemma, D is an independent set. Then, we deduce that D is an ISD-set of a general graph G.
□

11.4 Convergence & complexity analysis

In this section, the convergence of ISDS under the unfair distributed daemon is proved. The time complexity of the algorithm is analyzed in terms of rounds. Recall that in general a round under an unfair distributed daemon may consist of an infinite number of moves. Therefore, in the following section we bound the number of moves for all rounds.

11.4.1 Convergence proof

First, we prove in this section that ISDS requires only a finite number of moves.

Definition 18 *A move of a node v is called* in-move *if v executes rule $[R2]$ and assigns a value In to $v.state$.*

Lemma 11.4.1 *The Rule $[R1]$ can be executed at most once for any node $v \in V$.*

Proof. Since the open neighborhood of any node $v \in V$ does not change during the stabilization of the system, then its degree does not change too. □

Lemma 11.4.2 *Let a node $v \in V$ and suppose that during an interval of time $[t_1, t_2]$, there is no node u with $u \succ v$ makes an in-move. Then v makes at most one in-move for any execution of ISDS algorithm.*

Proof. Recall that a node $v \in V$ can make an in-move only if Rule $[R1]$ is disabled. This implies that any node v must have a correct value of $v.d$ (*i.e.* $v.d = d(v)$) before executing an in-move.

Furthermore, a node v, having $v.state = Out$, makes an in-move if $I(v) = In$, *i.e.* any neighbor u of v has $u.state = Out$ or $u.state = In$ and $v \succ u$. By assumption no neighbor u executes an in-move during the time interval $[t_1, t_2]$ such that $u \succ v$ (Note that if $d(u) > d(v)$ and u has an incorrect $u.d$ then u may execute $[R1]$ but can not execute $[R2]$ during $[t_1, t_2]$), hence, when v executes an in-move, then no neighbor u executes an in-move and therefore v remains $v.state = In$. □

Lemma 11.4.3 *The total number of in-moves of ISDS is finite.*

Proof. The proof is by induction on the order between nodes. The strongest node v makes at most one in-move by Lemma 11.4.2. During each of the two times intervals, when v is not making an in-move, using Lemma 11.4.2 again, the second strongest node $u \in V - \{v\}$ makes at most one in-move. Therefore, the same situation can be repeated for the rest of nodes, showing that all nodes make only a finite number of in-moves. This completes the proof. □

Lemma 11.4.4 *The total number of moves of ISDS is finite.*

Proof. Since the number of $[R2]$ executions is bounded by total number of in-moves plus n (since each node can make at most one out-move more than its number of in-move) and using Lemma 11.4.3, then the number of $[R2]$ executions is finite. This implies that the number of $[R1]$ (Lemma 11.4.1) and $[R2]$ executions is finite. □

Theorem 11.4.5 *The algorithm ISDS always stabilizes, and finds an independent strong dominating set.*

Proof. The algorithm ISDS is correct (Lemma 11.3.2) and makes a finite number of moves (Lemma 11.4.4), then we deduce that ISDS is a self-stabilizing algorithm and it provides an independent strong dominating set. □

11.4.2 Complexity analysis

In the following, we prove that after at most $(n + 1)$ rounds, the algorithm ISDS stabilizes.

Lemma 11.4.6 *After round r_1 and in all following rounds, each node $v \in V$ has a correct value of $v.d$, i.e. $v.d = d(v)$.*

Proof. It is obvious that any node $v \in V$ that has an incorrect value $v.d$ is enabled and when v executes rule $[R1]$ during r_1, v will have $v.d = d(v)$ during all the following rounds. □

Lemma 11.4.7 *Let v^* be the strongest node in G $(\forall v \in V, v^* \succ v)$. Then,*

(a) after round r_2 and in all following rounds, $v^*.state = In$.

(b) after round r_3 and in all following rounds, $v.state = Out$ for all $v \in N(v^*)$.

Proof. For proving property (a), it is sufficient to prove that during round r_2 and in all following rounds, we have $I(v^*) = In$.

By assumption and Lemma 11.4.6, we have $\forall v \in N(v^*), v^* \succ v$, this means that there are no neighbors stronger than v^* in state In, this implies $I(v^*) = In$. So, if $v^*.state = Out$ after round r_1 then v^* will execute rule $[R2]$ for updating $v^*.state$ to In and v^* will never make a move again.

Property (b) means that after round r_3, any node $v \in N(v^*)$ has $v.state = Out$. As previously shown, after round r_2 and in all following rounds, node v^* maintains $v^*.state = In$ (property (a)). By assumption any neighbor v of v^*, $v^* \succ v$, this implies $I(v) = Out$. So, any neighbor v of v^* has $v.state = In$ after round r_2, will be enabled by rule $[R2]$ and must execute this rule before the end of round r_3. Thus, after round r_3, any node $v \in N(v^*)$ will have $v.state = Out$ and will never move again. □

Lemma 11.4.8 *Algorithm ISDS stabilizes after at most $(n + 1)$ rounds.*

Proof. The proof is by induction. Consider the first round r_1, each node has a correct $v.d$. Let be v^* the strongest node in G. Using Lemma 11.4.7, the subgraph which contains the node v^* and its neighbors $N(v^*)$ will stabilize after at most two successive rounds. Let G' be the graph obtained by removing the first stabilized subgraph from G. The argument given above can be repeated. Hence, by induction, each strong node and its neighbors stabilize after at most two more rounds. Since G contains at most n nodes and using Lemma 11.4.6, Algorithm ISDS will stabilize after at most $n+1$ rounds. □

The following theorem summarizes the main result of this section.

Theorem 11.4.9 *ISDS is a self-stabilizing algorithm for computing an ISD-Set and converges after at most* $(n+1)$ *rounds under the unfair distributed daemon using* $O(\log n)$ *memory space.*

Proof. Using Theorem 11.4.5 and Lemma 11.4.8, we deduce that ISDS is a self-stabilizing algorithm for computing an ISD-set and converges after at most $(n+1)$ rounds.

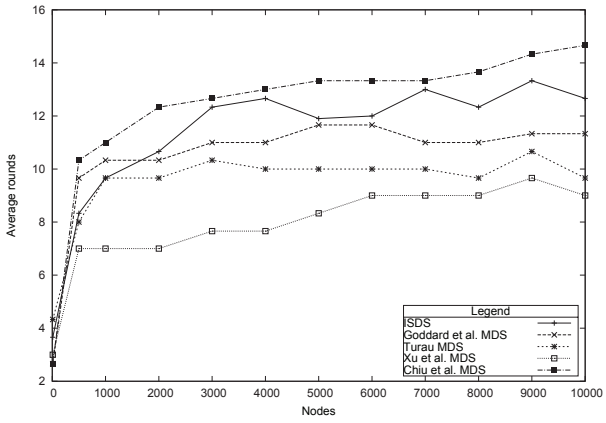
The memory requirement of ISDS amounts to $O(\log n)$ per node: Apart of the boolean variable *state*, a node has to store the variable d for its degree. Thus, each node uses only $O(\log n)$ memory space. □

11.5 Some simulations and performance analysis

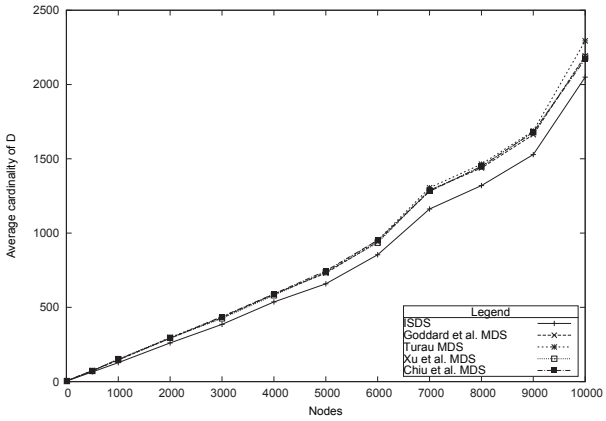
In the previous section, we showed that our algorithm ISDS has the same round complexity as the best self-stabilizing algorithms for MIS and MDS problems under the distributed daemon. In this section, we study the performance of ISDS by using simulations. It has to be mentioned that this is the first work that compares the different linear self-stablizing algorithms for MDS and MIS problems.

In our simulation, we used arbitrary undirected graphs with no loops or multiple edges between nodes. Also, the graphs used are sparse. We chose the graph density such that the graphs remain sparse and the maximum node degree is between 20 and 30. We used the synchronous daemon for all the algorithms. Our implementation is based on the source code developed by Lukasz Kuszner using the JAVA language [Luk05].

Considering an arbitrary graph G with n nodes, Figure 11.1 and Figure 11.2 show the performance of our algorithm ISDS in terms of average rounds and average cardinality of the set D comparing to other well-known self-stabilizing algorithms for MDS and MIS problems. The number n is varied from 0 to 10000 nodes. Thence, we compare our algorithm ISDS with the algorithm proposed by Turau for MDS in [Tur07], the algorithm proposed by Xu et al. in [XHGS03] and the algorithm proposed by Goddard et al. in [GHJ+08] and the algorithm proposed by Chiu et al. in [CCT14]. We also compare our algorithm with the algorithm proposed by Turau for MIS in [Tur07] and the algorithm proposed by Goddard et al. in [GHJS03c].



(a) Convergence time

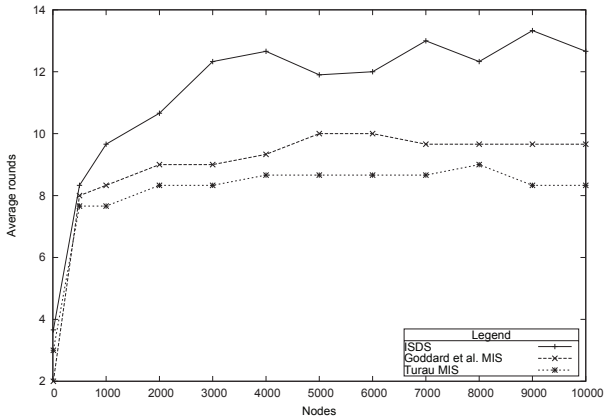


(b) Cardinality of the set D

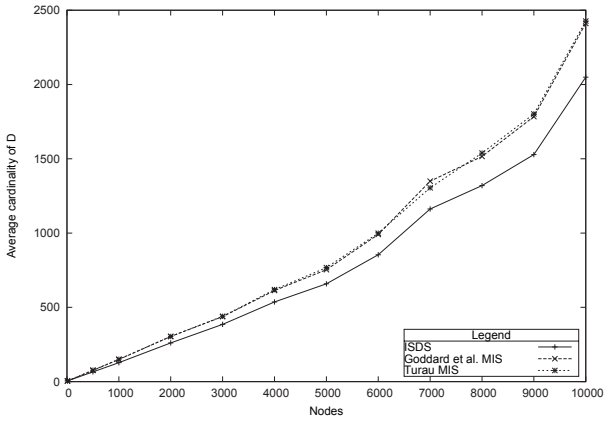
Figure 11.1: Comparison of ISDS with some MDS algorithms.

From the simulation results we can see that our algorithm has a convergence speed that is close to other MDS and MIS algorithms, as illustrated by Figure 11.1(a) and Figure 11.2(a), respectively. In fact, we observe that the round complexity does not depend on the number of nodes in large scale graphs. Surprisingly, the fast algorithm of Chiu et al [CCT14] in term of moves needs more rounds for its stabilization than others algorithms. This is because Chui et al. algorithm uses four states for each node.

Concerning our algorithm *ISDS*, the number of nodes selected to belong to D is appreciably fewer than MDS and MIS algorithms (up to 15%) especially for large graphs, as illustrated by Figure 11.1(b) and Figure 11.2(b). Moreover, observe that other algorithms give similar results for the cardinality of the set D while ours always gives smallest cardinalities.



(a) Convergence time



(b) Cardinality of the set D

Figure 11.2: Comparison of ISDS with some MIS algorithms.

11.6 Summary

In this section, we consider the problem of independent strong dominating set (ISD-set). Therefore, we proposed the first self-stabilizing algorithm for finding an ISD-set in general graphs. Using only $O(\log n)$ memory space per node, the algorithm operates under the unfair distributed daemon and stabilizes after at most $(n + 1)$ rounds.

Apart from the fact that ISDS algorithm theoretically converges in linear rounds, we evaluate the practical performance of this algorithm with well-known self-stabilizing algorithms for MIS and MDS problems. The simulations show that ISDS convergence speed is close to the other algorithms. However, the number of dominating nodes selected by our algorithm is always smaller than those given by other algorithms, especially for large graphs. This makes ISDS very suitable for improving the performance of wireless networks when the dominating or independent sets are needed to be as smaller as possible.

Furthermore, the algorithm ISDS can be generalized to weighted graphs by changing only the definition of the lexicographical order. Thus, the weight of nodes can be very useful in wireless networks where the weight of dominating nodes may represent remaining energy, mobility, signal strength, or average distance to neighbors. The dominated nodes may be associated with dominating nodes having the highest weight.

11.7 Conclusion

In this third part, we studied two parameters, called Edge Monitoring sets and Independent Dominating Sets in general graphs. Both parameters can be considered as variants of dominating sets problems. Then, we first proposed a self-stabilizing algorithm to find minimal edge-monitoring sets in general graphs. Such sets provide a valuable tool to implement a simple and effective mechanism for building secure wireless sensor networks. The algorithm has a lower move complexity than existing self-stabilizing algorithm.

About the second parameter, we proposed a self-stabilizing algorithm (ISDS) for finding an ISD-set in general graphs. The algorithm ISDS converges in linear rounds, while having the same order of rounds complexity as the best self-stabilizing algorithm for MDS and MIS problems. Furthermore, the simulations showed the efficiency of ISDS for reducing the cardinality of the dominating set.

Chapter 12

Conclusions and Perspectives

In this thesis, the algorithmic aspects and applications of four variants of graph decompositions and dominating sets are investigated. This concluding chapter summarizes the results presented in the previous chapters and discusses future work to each contribution.

At the beginning of this thesis, we presented basic concepts regarding the self-stabilization paradigm and discussed self-stabilizing algorithms for some classical graph problems. Then, we presented the four main contributions of this work that are divided into three parts:

In the first part, we discussed the problem of maximal partitioning into triangles (MPT) of arbitrary graphs. We gave an approximation ratio for this maximal partitioning. Moreover, we proved that finding a deterministic self-stabilizing algorithm for MPT problem under the distributed daemon is impossible in anonymous graphs.

We considered MPT problem as a generalization of maximal matching problem in graphs. Then, we proposed to start from the Hsu and Huang's algorithm for maximal matching in arbitrary graph and extend it in such a way to face the MPT problem by proposing the first self-stabilizing algorithm. Throughout the analysis of the behavior of the first proposed algorithm, we showed how it was difficult to prove its convergence using the variant function technique. Thus, we proved that the first algorithm converges in polynomial number of moves and a transformation is required in order to operate under the distributed daemon. Hence, a second self-stabilizing algorithm is developed in order to improve the first version. We showed that the improved algorithm operates under the unfair distributed daemon and stabilizes in linear moves.

As future work for MPT problem treated in this first part, we plan to focus on the following issues:

- The proposed algorithms for MPT problem provide a 3-approximation for maximum triangle partitions in general graphs. The natural question is how can we improve this approximation?

- Generalize these proposed algorithms for weighted graphs (for nodes and for edges).

The second part of this thesis was devoted to study the problem of maximal p-star decomposition (MSD) of arbitrary graphs. This decomposition also considered as a generalization of maximal matching problem in graphs when $p = 1$. We showed

that finding a deterministic self-stabilizing algorithm for such problem is impossible in anonymous graphs. Therefore, assuming distinct local identifiers, we presented two self-stabilizing algorithms for MSD problem.

The first algorithm converges in linear rounds under the unfair distributed daemon. Hence, its time complexity in rounds is the same order as the best known self-stabilizing algorithm for maximal matching under the synchronous daemon or the distributed daemon. Moreover, we showed that for any starting configuration, the algorithm always leads to a unique configuration that is a maximal p-star decomposition.

Afterwards, a second algorithm was developed that operates also under the unfair distributed daemon and considers all maximal p-star decomposition configurations to be legitimate. Even though this part contains two efficient self-stabilizing algorithms for MSD problem, a number of issues need to be further investigated.

- The first algorithm proposed for MSD stabilizes within linear rounds. However, its move complexity is not analyzed and it seems to be exponential using simulations. Therefore, we are not aware of an example where the first algorithm requires an exponential number of moves. For this, it would be interesting to show the existence of a polynomial bound for the number of moves of the first algorithm or to propose a new algorithm with linear moves complexity.

- The second algorithm is analyzed using moves only. Thus it would be interesting to find a formal proof for rounds complexity.

- All graphs considered in this part are not weighted, thus it would be interesting to generalize these algorithms for weighted graphs.

The last part of this thesis was devoted to study the edge monitoring and independent strong dominating set problems. Then, we presented a novel self-stabilizing algorithm for edge monitoring problem, improving the existing self-stabilizing algorithm for such problem. Our algorithm converges in polynomial moves and operates under the unfair distributed daemon without using any transformer.

We would like to investigate whether the move complexity of the proposed algorithm for this problem under the distributed daemon could be improved. Also, it would be of interest to prove the approximation ratio of the number of monitor nodes defined by the algorithm.

Concerning the second variant of dominating sets, we presented a greedy self-stabilizing algorithm for finding a minimal independent strong dominating set in arbitrary graphs. Moreover, we evaluated the practical performance of this algorithm with well-known self-stabilizing algorithms for MIS and MDS problems. The simulations shown that the proposed algorithm convergence speed is close to known algorithms. However, the number of dominating nodes selected by our algorithm is always smaller than those given by other algorithms, especially for large graphs. This makes our algorithm for ISD-set problem very suitable for improving

the performance of wireless networks when the cardinality of the dominating or independent sets are needed to be as smaller as possible.

Even though the proposed algorithm converges in linear rounds, finding a linear moves algorithm for ISD-set problem is still an open problem.

Bibliography

[AB93] Yehuda Afek and Geoffrey M. Brown. Self-stabilization over unreliable communication media. *Distributed Computing*, 7:27–34, 1993. (Cited on pages 8 and 20.)

[AG93] A. Arora and M. Gouda. Closure and convergence: a foundation of fault-tolerant computing. *Software Engineering, IEEE Transactions on*, 19(11):1015–1027, 1993. (Cited on page 12.)

[APSVD94] Baruch Awerbuch, Boaz Patt-Shamir, George Varghese, and Shlomi Dolev. Self-stabilization by local checking and global reset (extended abstract). In *WDAG*, pages 326–339, 1994. (Cited on page 19.)

[AR04] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '04, pages 120–124, 2004. (Cited on page 69.)

[AS88] Baruch Awerbuch and Michael Sipser. Dynamic networks are as fast as static networks. In *Foundations of Computer Science, 1988., 29th Annual Symposium on*, pages 206–219. IEEE, 1988. (Cited on page 19.)

[AWF03] 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(02):287–303, 2003. (Cited on pages 28 and 109.)

[BBCD02] Fatima Belkouch, Marc Bui, Liming Chen, and Ajoy Kumar Datta. Self-stabilizing deterministic network decomposition. *J. Parallel Distrib. Comput.*, 62(4):696–714, 2002. (Cited on page 33.)

[BDJV05] Doina Bein, Ajoy Kumar Datta, Chakradhar R. Jagganagari, and Vincent Villain. A self-stabilizing link-cluster algorithm in mobile ad hoc networks. In *ISPAN*, pages 436–441, 2005. (Cited on page 33.)

[BDTC05] Jeremy Blum, Min Ding, Andrew Thaeler, and Xiuzhen Cheng. Connected dominating set in sensor networks and manets. In Ding-Zhu Du and PanosM. Pardalos, editors, *Handbook of Combinatorial Optimization*, pages 329–369. Springer US, 2005. (Cited on pages 25 and 109.)

[BEZE01] D. Bryant, S. El-Zanati, and Ch. Eynden. Star factorizations of graph products. *J. Graph Theory*, 36(2):59–66, February 2001. (Cited on page 67.)

[BG09] A. Berns and S. Ghosh. Dissecting self-* properties. In *Self-Adaptive and Self-Organizing Systems, 2009. SASO '09. Third IEEE International Conference on*, pages 10–19, 2009. (Cited on page 13.)

[BM12] Jean R.S. Blair and Fredrik Manne. An efficient self-stabilizing distance-2 coloring algorithm. *Theoretical Computer Science*, 444(0):28 – 39, 2012. (Cited on page 21.)

[BMT09] A. Bendjoudi, N. Melab, and E.-G. Talbi. P2p design and implementation of a parallel branch and bound algorithm for grids. *Int. J. Grid Util. Comput.*, 1(2):159–168, 2009. (Cited on page 69.)

[BNBJ+08] Amotz Bar-Noy, Theodore Brown, Matthew P Johnson, Thomas La Porta, Ou Liu, and Hosam Rowaihy. Assigning sensors to missions with demands. In Mirosaw Kutyowski, Jacek Cicho, and Przemysaw Kubiak, editors, *Algorithmic Aspects of Wireless Sensor Networks*, volume 4837 of *Lecture Notes in Computer Science*, pages 114–125. Springer Berlin Heidelberg, 2008. (Cited on pages 22 and 35.)

[BPV04] Christian Boulinier, Franck Petit, and Vincent Villain. When graph theory helps self-stabilization. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 150–159. ACM, 2004. (Cited on page 18.)

[BYK14] Yacine Belhoul, Sad Yahiaoui, and Hamamache Kheddouci. Efficient self-stabilizing algorithms for minimal total k-dominating sets in graphs. *Information Processing Letters*, 114(7):339–343, 2014. (Cited on pages 25 and 28.)

[Cai74] P. Cain. Decomposition of complete graphs into stars. *Bull. Austral. Math. Soc.*, 10:23–30, 1974. (Cited on pages 67 and 68.)

[CCT14] 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. (Cited on pages 17, 25, 28, 114 and 115.)

[CDDL09] Eddy Caron, Ajoy Kumar Datta, Benjamin Depardon, and Lawrence L. Larmore. A self-stabilizing k-clustering algorithm using an arbitrary metric. In *Euro-Par*, pages 602–614, 2009. (Cited on page 33.)

[CHS02] Subhendu Chattopadhyay, Lisa Higham, and Karen Seyffarth. Dynamic and self-stabilizing distributed matching. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, PODC '02, pages 290–297, New York, NY, USA, 2002. ACM. (Cited on pages 22, 23 and 24.)

[CR02] Alberto Caprara and Romeo Rizzi. Packing triangles in bounded degree graphs. *Inf. Process. Lett.*, 84(4):175–180, November 2002. (Cited on page 34.)

[DFG06] Vadim Drabkin, Roy Friedman, and Maria Gradinariu. Self-stabilizing wireless connected overlays. In *Proceedings of the 10th International Conference on Principles of Distributed Systems*, OPODIS'06, pages 425–439, Berlin, Heidelberg, 2006. Springer-Verlag. (Cited on pages 26 and 28.)

[DHMU02] Gayla S Domke, Johannes H Hattingh, Lisa R Markus, and Elna Ungerer. On parameters related to strong and weak domination in graphs. *Discrete Mathematics*, 258(1):1–11, 2002. (Cited on page 110.)

[DHR+11] S. Devismes, K. Heurtefeux, Y. Rivierre, A.K. Datta, and L.L. Larmore. Self-stabilizing small k-dominating sets. In *Networking and Computing (ICNC), 2011 Second International Conference on*, pages 30–39, 2011. (Cited on page 28.)

[Dij74] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974. (Cited on pages 8, 11 and 15.)

[DIM93] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7(1):3–16, 1993. (Cited on page 8.)

[DLL08] Dezun Dong, Yunhao Liu, and Xiangke Liao. Self-monitoring for sensor networks. In *Proceedings of the 9th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, MobiHoc '08, pages 431–440, New York, NY, USA, 2008. ACM. (Cited on pages 96, 97 and 100.)

[DLL+11] Dezun Dong, Xiangke Liao, Yunhao Liu, Changxiang Shen, and Xinbing Wang. Edge self-monitoring for wireless sensor networks. *Parallel and Distributed Systems, IEEE Transactions on*, 22(3):514–527, 2011. (Cited on pages 96, 97 and 100.)

[DLV10] Ajoy K. Datta, Lawrence L. Larmore, and Priyanka Vemula. A self-stabilizing o(k)-time k-clustering algorithm. *Comput. J.*, 53(3):342–350, March 2010. (Cited on page 28.)

[DNB06] V. Delouille, R. Neelamani, and R.G. Baraniuk. Robust distributed estimation using the embedded subgraphs algorithm. *Signal Processing, IEEE Transactions on*, 54(8):2998–3010, 2006. (Cited on page 36.)

[DNCB03] V. Delouille, R. Neelamani, V. Chandrasekaran, and R.G. Baraniuk. The embedded triangles algorithm for distributed estimation in sensor

networks. In *Statistical Signal Processing, 2003 IEEE Workshop on*, pages 371–374, 2003. (Cited on page 36.)

[Dol00] Shlomi Dolev. *Self-stabilization*. MIT Press, 2000. (Cited on pages 8, 11, 17 and 20.)

[DT11] Swan Dubois and Sébastien Tixeuil. A taxonomy of daemons in self-stabilization. *CoRR*, abs/1110.0334, 2011. (Cited on page 15.)

[DWS14] Yihua Ding, James Z Wang, and Pradip K Srimani. A linear time self-stabilizing algorithm for minimal weakly connected dominating sets. *International Journal of Parallel Programming*, pages 1–12, 2014. (Cited on pages 27 and 28.)

[DWS15] Yihua Ding, James Wang, and Pradip K Srimani. Self-stabilizing algorithms for maximal 2-packing and general k-packing ($k \geq 2$) with safe convergence in an arbitrary graph. *International Journal of Networking and Computing*, 5(1):105–121, 2015. (Cited on page 21.)

[DXW09] Xuxing Ding, Fangfang Xie, and Qing Wu. Energy-balanced clustering with master/slave method for wireless sensor networks. In *Electronic Measurement Instruments, 2009. ICEMI '09. 9th International Conference on*, pages 3–20, Aug 2009. (Cited on page 69.)

[FPSV09] P. Foggia, G. Percannella, C. Sansone, and M. Vento. Benchmarking graph-based clustering algorithms. *Image and Vision Computing*, 27(7):979–988, 2009. (Cited on page 33.)

[Gär03] Felix C. Gärtner. A survey of self-stabilizing spanning-tree construction algorithms. Technical report, EPFL, October 2003. (Cited on page 21.)

[GBS08] Saurabh Ganeriwal, Laura K. Balzano, and Mani B. Srivastava. Reputation-based framework for high integrity sensor networks. *ACM Trans. Sen. Netw.*, 4(3):15:1–15:37, June 2008. (Cited on page 99.)

[GGH+04] Martin Gairing, Wayne Goddard, Stephen T. Hedetniemi, Petter Kristiansen, and Alice A. McRae. Distance-two information in self-stabilizing algorithms. *Parallel Processing Letters*, 14(03n04):387–398, 2004. (Cited on pages 18 and 19.)

[GHJ+08] Wayne Goddard, Stephen T. Hedetniemi, David Pokrass Jacobs, Pradip K. Srimani, and Zhenyu Xu. Self-stabilizing graph protocols. *Parallel Processing Letters*, 18(1):189–199, 2008. (Cited on pages 20, 25, 28, 109 and 114.)

[GHJS03a] 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. *Computers and*

Mathematics with Applications, 35:240–243, 2003. (Cited on pages 25 and 28.)

[GHJS03b] Wayne Goddard, Stephen T. Hedetniemi, David Pokrass Jacobs, and Pradip K. Srimani. A robust distributed generalized matching protocol that stabilizes in linear time. In *ICDCS Workshops*, pages 461–465, 2003. (Cited on pages 23 and 24.)

[GHJS03c] Wayne Goddard, Stephen T. Hedetniemi, David Pokrass Jacobs, and Pradip K. Srimani. Self-stabilizing protocols for maximal matching and maximal independent sets for ad hoc networks. In *IPDPS*, page 162, 2003. (Cited on pages 22, 24, 28, 29, 91 and 114.)

[GHJS05] Wayne Goddard, Stephen T. Hedetniemi, David Pokrass Jacobs, and Pradip K. Srimani. Self-stabilizing global optimization algorithms for large network graphs. *IJDSN*, 1(3&4):329–344, 2005. (Cited on page 25.)

[GHJT08] Wayne Goddard, Stephen T. Hedetniemi, David P. Jacobs, and Vilmar Trevisan. Distance- k knowledge in self-stabilizing algorithms. *Theor. Comput. Sci.*, 399(1-2):118–127, June 2008. (Cited on page 18.)

[GHS06] Wayne Goddard, Stephen T. Hedetniemi, and Zhengnan Shi. An anonymous selfstabilizing algorithm for 1-maximal matching in trees. *Information Processing Letters*, 91:797–803, 2006. (Cited on pages 22 and 24.)

[Gib85] Alan Gibbons. *Algorithmic Graph Theory.* Cambridge University Press, 1985. (Cited on page 22.)

[GJ79] 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. (Cited on page 34.)

[GK10] Nabil Guellati and Hamamache Kheddouci. A survey on self-stabilizing algorithms for independence, domination, coloring, and matching in graphs. *J. Parallel Distrib. Comput.*, 70(4):406–415, April 2010. (Cited on pages 12 and 21.)

[GS10] W. Goddard and P. K. Srimani. Anonymous self-stabilizing distributed algorithms for connected dominating set in a network graph. In *Proceedings of the international multi-conference on complexity, informatics and cybernetics*, IMCIC, 2010. (Cited on pages 26 and 28.)

[GT07] Maria Gradinariu and Sebastien Tixeuil. Conflict managers for self-stabilization without fairness assumption. In *Proceedings of the 27th International Conference on Distributed Computing Systems*, ICDCS

'07, pages 46–, Washington, DC, USA, 2007. (Cited on pages 19 and 49.)

[Hau12] Bernd Hauck. *Time- and Space-Efficient Self-Stabilizing Algorithms*. Dissertation, University of Hamburg-Harburg, 2012. (Cited on pages 20, 97, 100 and 108.)

[HCW08] 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. *J. Inf. Sci. Eng.*, 24(1):175–187, 2008. (Cited on pages 26 and 28.)

[Her02] T.R. Herman. A comprehensive bibliography on self-stabilization. *Theoretical Computer Science, Chicago J.*, 2002. (Cited on page 21.)

[HH92] Su-Chu Hsu and Shing-Tsaan Huang. A self-stabilizing algorithm for maximal matching. *Inf. Process. Lett.*, 43(2):77–81, 1992. (Cited on pages 18, 22, 23, 24 and 37.)

[HHJS03] S.M. Hedetniemi, S.T. Hedetniemi, D.P. Jacobs, and P.K. Srimani. Self-stabilizing algorithms for minimal dominating sets and maximal independent sets. *Computers and Mathematics with Applications*, 46(56):805–811, 2003. (Cited on pages 25 and 28.)

[HJS01] Stephen T. Hedetniemi, David Pokrass Jacobs, and Pradip K. Srimani. Maximal matching stabilizes in time o(m). *Inf. Process. Lett.*, 80(5):221–223, 2001. (Cited on page 22.)

[HLCW07] 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, 2007. (Cited on pages 26 and 28.)

[IK02] Hiroko Ishii and Hirotsugu Kakugawa. A self-stabilizing algorithm for finding cliques in distributed systems. *Reliable Distributed Systems, IEEE Symposium on*, 0:390, 2002. (Cited on page 33.)

[IKK02] M. Ikeda, S. Kamei, and H. Kakugawa. A space-optimal self-stabilizing algorithm for the maximal independent set problem. In *in: Proc. 3rd International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 70–74, 2002. (Cited on pages 28 and 29.)

[Ioa02] Kleoni Ioannidou. Transformations of self-stabilizing algorithms. In Dahlia Malkhi, editor, *Distributed Computing (DISC)*, volume 2508 of *Lecture Notes in Computer Science*, pages 103–117. Springer Berlin Heidelberg, 2002. (Cited on page 17.)

[JG05] Ankur Jain and A. Gupta. A distributed self-stabilizing algorithm for finding a connected dominating set in a graph. In *Parallel and Distributed Computing, Applications and Technologies, 2005. PDCAT 2005. Sixth International Conference on*, pages 615–619, 2005. (Cited on pages 26 and 28.)

[JM11] C. Johnen and F. Mekhaldi. Self-stabilizing computation and preservation of knowledge of neighbor clusters. In *Self-Adaptive and Self-Organizing Systems (SASO)*, pages 41–50, 2011. (Cited on page 20.)

[JM14] Colette Johnen and Fouzi Mekhaldi. Self-stabilizing with service guarantee construction of 1-hop weight-based bounded size clusters. *J. Parallel Distrib. Comput.*, 74(1):1900–1913, 2014. (Cited on pages 20 and 33.)

[Joh97] Colette Johnen. Memory efficient, self-stabilizing algorithm to construct bfs spanning trees. In *PODC*, page 288, 1997. (Cited on page 21.)

[KBNR05] I. Khalil, S. Bagchi, and C. Nina-Rotaru. Dicas: Detection, diagnosis and isolation of control attacks in sensor networks. In *Security and Privacy for Emerging Areas in Communications Networks, 2005. SecureComm 2005. First International Conference on*, pages 89–100, 2005. (Cited on page 99.)

[KBS05] Issa Khalil, Saurabh Bagchi, and Ness B. Shroff. Liteworp: A lightweight countermeasure for the wormhole attack. In *in Multihop Wireless Network. In the International Conference on Dependable Systems and Networks (DSN*, pages 612–621, 2005. (Cited on page 99.)

[Kes88] J. L. W. Kessels. An exercise in proving self-stabilization with a variant function. *Inf. Process. Lett.*, 29(1):39–42, September 1988. (Cited on pages 19 and 20.)

[KH78a] D. Kirkpatrick and P. Hell. On the completeness of a generalized matching problem. In *STOC*, pages 240–245, New York, USA, 1978. ACM. (Cited on page 68.)

[KH78b] David G. Kirkpatrick and Pavol Hell. On the completeness of a generalized matching problem. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, pages 240–245, New York, NY, USA, 1978. ACM. (Cited on page 68.)

[KH83] D. Kirkpatrick and P. Hell. On the complexity of general graph factor problems. *SIAM Journal on Computing*, 12(3):601–609, 1983. (Cited on page 68.)

[KK03] S. Kamei and H. Kakugawa. A self-stabilizing algorithm for the distributed minimal k-redundant dominating set problem in tree networks. In *Parallel and Distributed Computing, Applications and Technologies, 2003. PDCAT'2003. Proceedings of the Fourth International Conference on*, pages 720–724, 2003. (Cited on pages 26 and 28.)

[KK05] Sayaka Kamei and Hirotsugu Kakugawa. A self-stabilizing approximation algorithm for the distributed minimum k-domination. *IEICE Transactions*, 88-A(5):1109–1116, 2005. (Cited on pages 26 and 28.)

[KK07] S. Kamei and H. Kakugawa. A self-stabilizing approximation algorithm for the minimum weakly connected dominating set with safe convergence. In *Proceedings of the 1st International Workshop on Reliability, Availability, and Security (WRAS)*, page 5766, 2007. (Cited on pages 27 and 28.)

[KK08] Sayaka Kamei and Hirotsugu Kakugawa. A self-stabilizing approximation for the minimum connected dominating set with safe convergence. In TheodoreP. Baker, Alain Bui, and Sbastien Tixeuil, editors, *Principles of Distributed Systems*, volume 5401 of *Lecture Notes in Computer Science*, pages 496–511. Springer Berlin Heidelberg, 2008. (Cited on pages 26 and 28.)

[KK10] Sayaka Kamei and Hirotsugu Kakugawa. A self-stabilizing distributed approximation algorithm for the minimum connected dominating set. *Int. J. Found. Comput. Sci.*, 21(3):459–476, 2010. (Cited on pages 26 and 28.)

[KM08] Hirotsugu Kakugawa and Toshimitsu Masuzawa. Convergence time analysis of self-stabilizing algorithms in wireless sensor networks with unreliable links. In Sandeep Kulkarni and Andr Schiper, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 5340 of *Lecture Notes in Computer Science*, pages 173–187. Springer Berlin Heidelberg, 2008. (Cited on page 20.)

[KMW04] Fabian Kuhn, Thomas Moscibroda, and Rogert Wattenhofer. Initializing newly deployed ad hoc and sensor networks. In *Proceedings of the 10th annual international conference on Mobile computing and networking*, pages 260–274. ACM, 2004. (Cited on pages 27 and 109.)

[KP93] Shmuel Katz and Kenneth J. Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7(1):17–26, 1993. (Cited on page 19.)

[KS00] M.H. Karaata and K.A. Saleh. A distributed self-stabilizing algorithm for finding maximum matching. *Computer Systems Science and Engineering*, 3:175–180, 2000. (Cited on pages 23 and 24.)

[KS08] Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, New York, NY, USA, 1 edition, 2008. (Cited on page 7.)

[Lam84] L. Lamport. Solved problems unsolved problems and non-problems in concurrency. In *PODC*, pages 1–11, 1984. (Cited on page 11.)

[LC06] Suk-Bok Lee and Yoon-Hwa Choi. A resilient packet-forwarding scheme against maliciously packet-dropping nodes in sensor networks. In *Proceedings of the Fourth ACM Workshop on Security of Ad Hoc and Sensor Networks*, SASN '06, pages 59–70, New York, NY, USA, 2006. ACM. (Cited on page 99.)

[LHK11] Slimane Lemmouchi, Mohammed Haddad, and Hamamache Kheddouci. Study of robustness of community emerged from exchanges in networks communication. In *MEDES*, pages 189–196, 2011. (Cited on page 69.)

[LHK13] Slimane Lemmouchi, Mohammed Haddad, and Hamamache Kheddouci. Robustness study of emerged communities from exchanges in peer-to-peer networks. *Computer Communications*, 36(10-11):1145–1158, 2013. (Cited on pages 27, 36, 69 and 109.)

[LL05] H. Lee and Ch. Lin. Balanced star decompositions of regular multigraphs and λ-fold complete bipartite graphs. *Discrete Mathematics*, 301(2-3):195–206, 2005. (Cited on pages 67 and 68.)

[LS96] Ch. Lin and T. Shyu. A necessary and sufficient condition for the star decomposition of complete graphs. *J. Graph Theory*, 23(4):361–364, December 1996. (Cited on page 67.)

[Luk05] Lukasz Kuszner. Source code homepage. *http://kaims.pl/ kuszner*, 2005. (Cited on page 114.)

[MG12] E. Ebin Raja Merly and N. Gnanadhas. Linear star decomposition of lobster. *Int. J. of Contemp. Math. Sciences*, 7(6):251–261, 2012. (Cited on page 67.)

[MGLB00] Sergio Marti, Thomas J Giuli, Kevin Lai, and Mary Baker. Mitigating routing misbehavior in mobile ad hoc networks. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 255–265. ACM, 2000. (Cited on page 99.)

[MM07] Fredrik Manne and Morten Mjelde. A self-stabilizing weighted matching algorithm. In *SSS*, pages 383–393, 2007. (Cited on pages 23 and 24.)

[MMPT07] Fredrik Manne, Morten Mjelde, Laurence Pilard, and Sébastien Tixeuil. A new self-stabilizing maximal matching algorithm. In *SIROCCO*, pages 96–108, 2007. (Cited on page 23.)

[MMPT09] Fredrik Manne, Morten Mjelde, Laurence Pilard, and Sébastien Tixeuil. A new self-stabilizing maximal matching algorithm. *Theor. Comput. Sci.*, 410(14):1336–1345, March 2009. (Cited on pages 21, 23, 24, 70, 71, 72 and 91.)

[MMT07] M. Mezmaz, N. Melab, and E-G. Talbi. A Grid-based Parallel Approach of the Multi-Objective Branch and Bound. In *Proceedings 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, PDP, pages 23–30, 2007. (Cited on page 69.)

[NHK12] Brahim Neggazi, Mohammed Haddad, and Hamamache Kheddouci. Self-stabilizing algorithm for maximal graph partitioning into triangles. In *Proceedings of the 14th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, SSS, pages 31–42, 2012. (Cited on page 37.)

[NHTK14] Brahim Neggazi, Mohammed Haddad, Volker Turau, and Hamamache Kheddouci. A self-stabilizing algorithm for edge monitoring problem. In *Stabilization, Safety, and Security of Distributed Systems*, pages 93–105. Springer, 2014. (Cited on page 97.)

[NTHK13] Brahim Neggazi, Volker Turau, Mohammed Haddad, and Hamamache Kheddouci. A self-stabilizing algorithm for maximal p-star decomposition of general graphs. In *SSS*, pages 74–85, 2013. (Cited on page 71.)

[OR09] Sibel Ozkan and C.A. Rodger. Hamilton decompositions of graphs with primitive complements. *Discrete Mathematics*, 309(14):4883 – 4888, 2009. (Cited on page 33.)

[Pap94] C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, 1994. (Cited on page 34.)

[Pot97] Alex Pothen. Graph partitioning algorithms with applications to scientific computing. Technical report, Norfolk, VA, USA, 1997. (Cited on page 69.)

[REJ+07] H. Rowaihy, S. Eswaran, M. Johnson, D. Verma, A. Bar-Noy, T. Brown, and T. La Porta. A survey of sensor selection schemes in wireless sensor networks. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 6562 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, 2007. (Cited on pages 22 and 35.)

[RTAS09] H. Raei, M. Tabibzadeh, B. Ahmadipoor, and S. Saei. A self-stabilizing distributed algorithm for minimum connected dominating sets in wireless sensor networks with different transmission ranges. In *Advanced*

Communication Technology, 11th International Conference on, volume 01, pages 526–530, 2009. (Cited on pages 26 and 28.)

[SAG11] Belkacem Serrour, Alex Arenas, and Sergio Gomez. Detecting communities of triangles in complex networks using spectral optimization. *Computer Communications*, 34(5):629–634, 2011. (Cited on page 36.)

[SGH04] Z. Shi, W. Goddard, and S. T. Hedetniemi. An anonymous self-stabilizing algorithm for 1-maximal independent set in trees. *Information Processing Letters*, 91(2):77–83, 2004. (Cited on pages 28 and 29.)

[Shy10] Tay-Woei Shyu. Decomposition of complete graphs into paths and stars. *Discrete Mathematics*, 310(1516):2164–2169, 2010. (Cited on page 33.)

[SL96] E Sampathkumar and L Pushpa Latha. Strong weak domination and domination balance in a graph. *Discrete Mathematics*, 161(1):235–242, 1996. (Cited on pages 27, 96 and 109.)

[SRR+95] Sandeep K Shukla, Daniel J Rosenkrantz, SS Ravi, et al. Observations on self-stabilizing graph algorithms for anonymous networks. In *Proceedings of the second workshop on self-stabilizing systems*, volume 7, page 15, 1995. (Cited on pages 28 and 29.)

[SW93] Akira Saito and Manoru Watanbe. Graph partitioning into induced stars. *Ars Combinatoria*, 36:3–6, 1993. (Cited on page 67.)

[SX07] P.K. Srimani and Zhenyu Xu. Self-stabilizing algorithms of constructing spanning tree and weakly connected minimal dominating set. In *Distributed Computing Systems Workshops, 2007. ICDCSW '07. 27th International Conference on*, pages 3–3, 2007. (Cited on pages 20, 27 and 28.)

[Tel94a] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, New York, NY, USA, 1994. (Cited on page 16.)

[Tel94b] Gerard Tel. Maximal matching stabilizes in quadratic time. *Information Processing Letters*, 49(6):271 – 272, 1994. (Cited on pages 20 and 22.)

[TH09] Volker Turau and Bernd Hauck. A self-stabilizing algorithm for constructing weakly connected minimal dominating sets. *Inf. Process. Lett.*, 109(14):763–767, 2009. (Cited on pages 27 and 28.)

[TH11] Volker Turau and Bernd Hauck. A new analysis of a self-stabilizing maximum weight matching algorithm with approximation ratio 2. *Theor. Comput. Sci.*, 412(40):5527–5540, 2011. (Cited on pages 20, 21, 23 and 24.)

[The00] Oliver Theel. Exploitation of ljapunov theory for verifying self-stabilizing algorithms. In Maurice Herlihy, editor, *Distributed Computing (DISC)*, volume 1914 of *Lecture Notes in Computer Science*, pages 209–222. Springer Berlin Heidelberg, 2000. (Cited on page 20.)

[Tix09] Sébastien Tixeuil. *Algorithms and Theory of Computation Handbook, Second Edition, chapter Self-stabilizing Algorithms*. CRC Press, 2 edition, 2009. (Cited on pages 9 and 10.)

[Tur07] Volker Turau. Linear self-stabilizing algorithms for the independent and dominating set problems using an unfair distributed scheduler. *Inf. Process. Lett.*, 103(3):88–93, 2007. (Cited on pages 20, 25, 26, 28, 29 and 114.)

[Tur12] Volker Turau. Efficient transformation of distance-2 self-stabilizing algorithms. *Journal of Parallel and Distributed Computing*, 72(4):603–612, 2012. (Cited on pages 18, 19, 25, 26, 28, 97 and 100.)

[UT11] S. Unterschutz and V. Turau. Construction of connected dominating sets in large-scale manets exploiting self-stabilization. In *Distributed Computing in Sensor Systems and Workshops (DCOSS), 2011 International Conference on*, pages 1–6, 2011. (Cited on pages 25 and 109.)

[WS05] Xinzhou Wu and R. Srikant. Regulated maximal matching: A distributed scheduling algorithm for multi-hop wireless networks with node-exclusive spectrum sharing. In *Decision and Control, 2005 and 2005 European Control Conference. CDC-ECC '05. 44th IEEE Conference on*, pages 5342–5347, Dec 2005. (Cited on pages 22 and 35.)

[XHGS03] Zhenyu Xu, Stephen T. Hedetniemi, Wayne Goddard, and Pradip K. Srimani. A synchronous self-stabilizing minimal domination protocol in an arbitrary network graph. In *IWDC*, pages 26–32, 2003. (Cited on pages 25, 28 and 114.)

[YKR06] Ossama Younis, Marwan Krunz, and Srinivasan Ramasubramanian. Node clustering in wireless sensor networks: Recent developments and deployment challenges. *Network, IEEE*, 20(3):20–25, 2006. (Cited on pages 27 and 109.)

List of Figures

List of Algorithms