Time- and Space-Efficient Self-Stabilizing Algorithms

Vom Promotionsausschuss der
Technischen Universität Hamburg-Harburg

zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften (Dr. rer. nat.)

genehmigte Dissertation

von

Bernd Hauck

aus

Hamburg

2012

Date of Oral Examination	December 19th, 2012
Chair of Examination Board	Prof. Dr. Sibylle Schupp Institute for Software Systems Hamburg University of Technology
First Examiner	Prof. Dr. Volker Turau Institute of Telematics Hamburg University of Technology
Second Examiner	Prof. Dr.-Ing. Oliver Theel Department of Computer Science University of Oldenburg

Abstract

In a distributed system error handling is inherently more difficult than in conventional systems that have a central control unit. To recover from an erroneous state the nodes have to cooperate and coordinate their actions based on local information only. Self-stabilization is a general approach to make a distributed system tolerate arbitrary transient faults by design. A self-stabilizing algorithm reaches a legitimate configuration in a finite number of steps by itself without any external intervention, regardless of the initial configuration. Furthermore, once having reached legitimacy this property is preserved. An important characteristic of an algorithm is its worst-case runtime and its memory requirements. This thesis presents new time- and space-efficient self-stabilizing algorithms for well-known problems in algorithmic graph theory and provides new complexity analyses for existing algorithms. The main focus is on proof techniques used in the complexity analyses and the design of the algorithms. All algorithms presented in this thesis assume the most general concept with respect to concurrency.

The maximum weight matching problem is a fundamental problem in graph theory with a variety of applications. In 2007, Manne and Mjelde presented the first self-stabilizing algorithm to compute a 2-approximation for this problem. They proved an exponential upper bound on the time complexity until stabilization is reached for both the sequential and the concurrent setting. This thesis presents a new proof technique based on graph reduction to analyze the complexity of self-stabilizing algorithms. It is used to show that the algorithm of Manne and Mjelde in fact stabilizes within polynomial time assuming sequential execution and that a modified version of the algorithm also stabilizes within polynomial time in a concurrent setting.

Connected dominating sets are a vital structure for many applications. By relaxing the connectivity requirement the number of nodes can be reduced significantly. The first self-stabilizing algorithm for the weakly connected minimal dominating set problem was presented by Srimani and Xu in 2007. For the worst-case runtime they proved an exponential upper bound. It remained an open problem whether this limit is sharp. This thesis provides an example that shows that their algorithm indeed has an exponential time complexity. Furthermore, a new self-stabilizing algorithm is presented that stabilizes within polynomial time.

Another classical problem in graph theory is the computation of a minimum vertex cover. Currently, all self-stabilizing algorithms for this problem assume symmetry-breaking mechanisms, such as restricted concurrency, unique identifiers, or randomization. This thesis presents a deterministic self-stabilizing algorithm to compute a $(3 - \frac{2}{\Delta+1})$-approximation of a minimum vertex cover in anonymous networks. It reaches stabilization within polynomial runtime and requires $O(\log n)$ storage per node. For trees the algorithm computes a 2-approximation of a minimum vertex cover.

In 2008, Dong et al. introduced the edge-monitoring problem and provided a

distributed algorithm to solve it. In this thesis the first self-stabilizing algorithm for this problem is developed. Several versions of the edge-monitoring problem are considered. The proposed algorithms have polynomial time complexity.

Table of Contents

TABLE OF CONTENTS

Chapter 1

Introduction

Historically, computer systems were designed using architectures with a single control unit. However, a lot of applications do not necessarily require a central instance that is responsible for all decisions of the system. There are multiple reasons to distribute the control among several entities that can make their own decisions, depending on the context of an application. Just to mention a few: Prices for small and powerful microprocessors continue to decrease, there is permanent and accelerated progress in communication technology, multiple control units facilitate concurrent and parallel processing and increase a system's scalability. Some applications even rely on an inherently distributed setting, such as gathering data via a wireless sensor network. Due to its growing importance, in the late 1970s the analysis of computer systems with several control units became a field of research on its own, called *distributed computing*.

A distributed system consists of several autonomous computational units, so-called *nodes*, that aim to achieve a common goal. The system's topology is represented by a graph composed of the nodes and the communication links between them. In the absence of a shared memory, the nodes communicate with their adjacent nodes by passing messages. *Distributed algorithms* are a class of algorithms that are specifically designed for such settings. Typically, all nodes run the same program concurrently and they only have access to their own and each neighbor's state. There is no central unit that has knowledge of the whole system. Thus, all decisions a node makes are based on local knowledge. This absence of a unit with global knowledge that can steer

the whole system is the basis for a distributed system, and it is also the key challenge of the design of distributed algorithms: Since a node can only observe the behavior of its direct neighbors distant nodes cannot easily coordinate their own actions.

A computer system has to be prepared to deal with errors that may occur. This especially holds for a distributed setting. Several factors may affect the system's state adversely. For instance, a node can fail due to damage or energy depletion, the state of a node can change as a result of memory corruption, or new nodes are added to the system. The lack of a coordinator that has access to the state of all nodes makes it rather difficult to detect faults in the system. Locating the source of an error, replacing or removing an erroneous node or permanently monitoring the whole system to detect faults and perform a global reset as needed can be complex and expensive.

There are two strategies to deal with faults in a computer system: *Masking* solutions hide all errors from the application and the system stays operational without restrictions. However, such an approach is rather expensive as it depends on redundancy and all possible faults have to be known in advance. In case the continuous effective operation of the system is too expensive to guarantee or not essential, a *non-masking* solution is possible. These approaches accept that the application may not work properly for limited time.

Self-Stabilization is a general, non-masking approach to make a distributed system tolerate arbitrary transient faults by design. A distributed system is called self-stabilizing if it reaches a legitimate configuration in a finite number of steps by itself without external intervention and remains legitimate, starting from any possible global configuration. The concept of self-stabilization was presented forty years ago and has attracted a lot of research activity recently.

Most research concentrates on the development of new algorithms to improve the worst-case runtime of a self-stabilizing algorithm for a given problem. Therefore the presentation of a self-stabilizing algorithm is usually followed by an analysis that not only proves its correctness and the self-stabilization property but also provides an upper bound on the time complexity until the algorithm terminates. This analysis is inherently more difficult compared to the analysis of conventional (distributed) algorithms since it is inadmissible to assume one fixed initial state to start from. Hence, several proof techniques were developed to facilitate the analysis of self-stabilizing algorithms.

This thesis contributes new self-stabilizing algorithms for common problems in

graph theory and analyzes their worst-case time complexity. Furthermore, existing algorithms are examined to improve their complexity analysis. In doing so, several proof techniques are demonstrated by applying them to certain algorithms. Beyond that, a new method to determine the worst-case complexity of self-stabilizing algorithms is presented.

The new proof technique represents the main contribution of this thesis. It consists of a mapping from the execution sequence of a graph to that of a reduced graph. This allows to leverage complete induction in the proofs. Along with the use of a potential function this technique is applied to an algorithm by Manne and Mjelde that calculates a 2-approximation for the weighted matching problem. Its time complexity was stated to be exponential. By using the new technique this estimate can be improved significantly: It stabilizes within polynomial runtime.

Furthermore, this thesis completes the analysis of an algorithm by Srimani and Xu that builds up a weakly connected minimal dominating set by providing a lower bound which shows that their algorithm has an exponential runtime. The main disadvantage of the algorithm is identified: The hierarchic structure required by the algorithm gives higher-ranked nodes a superior position that forces all lower-ranked nodes to adapt their states multiple times with just one state update in an adverse setting. A new self-stabilizing algorithm is developed starting from a distance-two design and using a decomposition of the graph to reduce the impact of a single node's state change. The analysis shows that the new algorithm has a polynomial time complexity.

Anonymous networks pose a particular challenge due to the lack of symmetry-breaking mechanisms. There are very few positive results for self-stabilizing algorithms in such a network. In this thesis a new self-stabilizing approximation algorithm for the vertex cover problem in an anonymous network is presented. It is shown that for certain classes of graphs an algorithm with better approximation ratio cannot exist. The design of the algorithm is based on a virtual network which is simulated by the nodes. The complexity analysis studies the local states of the nodes and yields a polynomial result for the time complexity.

The edge-monitoring problem was introduced by Dong et al. recently. It has important applications in wireless network security. The authors proved this problem to be NP-complete and proposed a distributed algorithm to solve it. This thesis presents the first self-stabilizing algorithm for the edge-monitoring problem. Several versions of the problem are considered. The proposed algorithms have polynomial

time complexities.

This thesis is organized as follows: Chapter 2 provides an introduction into self-stabilization and presents the state of the art. The general structure of analyses of self-stabilizing algorithms as well as the most common proof techniques are presented in Chapter 3. The following chapters demonstrate the usage of certain proof techniques with the help of algorithms for the above-mentioned graph problems: Chapter 4 considers the weakly connected minimal dominating set problem and shows how the use of network decomposition leads to a more local analysis. Chapters 5 and 6 both perform an analysis of local states and sequences using different models of computation. The former addresses the calculation of a vertex cover in anonymous networks while the latter considers the edge-monitoring problem. The new proof technique is introduced in Chapter 7. It is demonstrated using an algorithm for the maximum weight matching problem. Chapter 8 summarizes this thesis and discusses future perspectives.

Chapter 2

Self-Stabilization

This chapter provides an introduction to self-stabilizing algorithms and related work. The first section describes conventional distributed algorithms and the models of computation. Section 2.2 starts with the categorization of faults in distributed systems and fault tolerance. It introduces self-stabilization and gives a more formal definition of the terms and concepts used in this thesis. Several methods to measure the complexity of self-stabilizing algorithms are discussed. Section 2.3 presents methods to design a self-stabilizing algorithm. Finally, Section 2.4 provides an overview of self-stabilizing algorithms for classical graph problems. More related work on specific problems can be found in the corresponding chapters.

2.1 Distributed Algorithms

In the literature, different definitions for the term *distributed system* can be found. Tanenbaum and van Steen [TS06] provide a definition that emphasizes the transparency property:

> A distributed system is a collection of independent computers that appears to its users as a single coherent system.

A famous aphorism by Lamport [Lam87] alludes to this property:

> A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

Bal et al. [BST89] characterize a distributed system in a more technical manner:

> A distributed computing system consists of multiple autonomous processors that do not share primary memory, but cooperate by sending messages over a communication network.

They also discuss the disagreement on the term "distributed system" in the literature [BST89]. The definition of Bal et al. will be used throughout this thesis with the understanding that this definition is not limited to physical processors but also considers other autonomous units or *nodes* such as processes. The latter is what Bal et al. call a *logically distributed software system*, but their distinction is not needed on the level of abstraction of this thesis. The communication network mentioned in the definition is considered to be a (connected) graph and only adjacent nodes can communicate with each other directly.

Two main models of distributed systems are distinguished in the literature [Pel00]: The synchronous model and the asynchronous model, the difference being whether there are upper bounds on the time certain processes are allowed to consume. The asynchronous model does not make any assumptions on the duration of a computational step or message delay, apart from being finite. Thus, messages that are sent but not received within a certain time cannot be considered to be lost but may be received later. On the other hand, the synchronous model assumes fixed time intervals for computations and guarantees that any message is received within a given time (which is known to all nodes). Hence, an advantage of synchronous systems is that lost messages can be detected. In this thesis the degree of synchrony of the distributed system is determined by the model used for the *atomicity of communication* and the assumed *scheduler*. These terms will be explained later. More detailed information about distributed systems in general can be found e.g. in [CDK05].

A *distributed algorithm* is an algorithm specifically designed to run in a distributed system. The nodes can operate concurrently and they communicate with each other to achieve a common goal. The most significant difference compared to conventional algorithms is the lack of a central entity that has access to the global state, i.e. the state of each node. All nodes act autonomously and the basis for their decisions is local knowledge only: The nodes hold their own state and can retrieve the state of their neighbors.

It is possible to gather the local state of all nodes by passing a neighbor's state on to the next node until some special node has aggregated the information of the whole system and can send tasks to the other nodes, but that is contrary to the idea of a distributed algorithm. Furthermore this procedure requires time and memory proportional to the size of the graph. The same arguments hold for a similar approach: If all nodes determine the topology of the whole distributed system, they can calculate their final state locally via the execution of an algorithm that is not restricted to local knowledge.

Having a "distributed state" and nodes that execute their algorithm according to local information only, different parts of the system may temporarily veer away from their common goal without knowing it. This also depends on the *locality* of the given problem or algorithm, i.e. to which extent the state of a node far away from a certain node influences its own state. An example for such a dependency is given in Section 4.2. More information about the locality of specific problems can be found in [NS95, MNS95, AGLP89, Suo11].

The atomicity of communication between the nodes can be modeled in miscellaneous ways for distributed algorithms [AW04, Tix09]. Tixeuil [Tix09] emphasizes that most literature in the context of self-stabilizing algorithms uses a high level of atomicity and lists the three most common models:

1. The *state model* (or *shared-memory model with composite atomicity*, [Dij74, Dol00]): In this model, reading the states of all adjacent nodes and updating its own state is considered an atomic action.

2. The *shared-register model* (or *read-write atomicity model*, [DIM93]): This model treats a single read and a single write operation as atomic actions. This model is the more general one, but there are methods for transforming algorithms from one model to the other [Dol00].

3. The *message-passing model* [AB93, DIM97a, KP90]: Here, an atomic step consists of either sending a message to one of the neighboring nodes, or receiving such a message.

The latter model requires to explicitly use the send and receive operation in an algorithm to exchange messages. The first two models simulate a common memory area for two adjacent nodes. In these cases, lower layers realize the information

exchange [Tel01]. Where not explicitly stated otherwise, this thesis assumes the state model for the algorithms. Another model is often used for algorithms in anonymous networks (see below):

4. The *link-register model* with composite atomicity [DIM93]: In this model, a node uses two separate registers for each neighbor (a read and a write register), i.e. a node can only read "its own" segment of its neighbors memory. Reading its registers from all neighbors and updating its own registers is considered one atomic operation. A more formal introduction to the link-register model is provided in Chapter 5.

Distributed algorithms substantially depend on the properties of the underlying network. In a *uniform* network all nodes execute the same algorithm. Non-uniform networks allow the nodes to execute distinct algorithms. A very important property is the availability of a symmetry-breaking mechanism. Such a mechanism is needed e.g. if it is undesirable that two adjacent nodes change their state at the same time. The most common model assumes all nodes to have unique identifiers. These can be used to ensure local mutual exclusion. For instance, in [GT07] the nodes have to set a boolean flag to tell their neighbors in advance when they want to change their state. A node is allowed to change its state only if none of its neighbors with smaller identifier has also set its flag.

Non-uniform networks can use another mechanism to break the symmetry by having a node that takes on a special role. These two network models are equivalent [Dol00]. In uniform networks without unique identifiers it is possible to use randomization to break symmetry. Availing oneself of randomization results in a probabilistic algorithm, though. A network is called *anonymous* if it is uniform and there are no further symmetry breaking mechanisms such as unique identifiers or randomization.

A lot of research has been done in the field of algorithms in anonymous networks. Angluin made the most remarkable publication in that area by proving several impossibility results subject to the different anonymity properties of the network [Ang80]. In particular Angluin showed that it is impossible to break symmetry via a *port numbering* (i.e., an edge ordering, for details see Chapter 5) in general graphs. Most of the algorithms in this thesis assume a uniform network and that all nodes have (locally) unique identifiers. Only in Chapter 5 an anonymous network is assumed.

2.2 Fault Tolerance and Self-Stabilization

In general, it is impossible to guarantee that a system will stay free of faults all the time. Hence, there must be a strategy to handle errors if they occur. Conventional systems may have a central unit that detects errors and decides which measures have to be taken. In a distributed system, error-handling is inherently more difficult: The detection of an error is not as simple due to the lack of a node with global knowledge, and also the nodes have to cooperate and coordinate their actions in order to overcome the erroneous state. Furthermore, there are types of errors that occur more likely in a distributed system. For instance, in a wireless sensor network a node can fail due to a depleted battery or physical damage. Messages can get lost, they may be duplicated or arrive in a different order.

Apart from errors there are other scenarios that can make a distributed system end up in an illegitimate state, e.g. if new nodes are added to the system or some nodes are removed from it. Locating the source of an error, replacing or removing an erroneous node, or permanently monitoring the whole system to detect faults and perform a global reset as needed can be complex and expensive.

If a distributed system does not tolerate any errors the fault of a single node can corrupt the whole system, i.e. if this node exclusively offers an essential service to the other nodes. There are several strategies to deal with faults. They will be discussed after a short classification of faults in distributed systems.

2.2.1 Classification of Faults in Distributed Systems

This section is based on [Tix09]. Another taxonomy of faults and fault-tolerance can be found in [Gär99]. Tixeuil distinguishes the *nature* of a fault, depending on whether it involves the state or the code of a node. *State-related faults* only affect – as the name says – the state of a node, i.e. the node's variables may change their values erroneously. Such errors occur e.g. due to cosmic rays or because of the continuously decreasing transistor size. *Code-related faults* compromise the node's behavior. This category includes crashes, omissions, duplications, desequencing and Byzantine faults [LSP82]. A more detailed description can be found in [Tix09].

Another criterion is the *type* of a fault. This aspect classifies the time span in which faults of arbitrary nature can occur. Three types are distinguished: *Transient faults*

are considered not to occur after a given point in the execution, i.e. there is a "last" transient error. In contrast, *permanent faults* stay permanently after a given point in the execution. *Intermittent faults* have no further limitation. Such faults can hit the system at any time. The latter type of faults is the most general one and subsumes the other two types. However, if intermittent faults do not occur too frequently, it may be sufficient to have a system tolerate transient faults provided that the time interval in which it stays operational is long enough.

A third category in the fault taxonomy of Tixeuil is the *extent* (or *span*) of the faults, describing how many components of the network can get hit by an error. In this thesis the extent of faults is insignificant.

2.2.2 Fault Tolerance and Self-Stabilizing Algorithms

Depending on the application area of the distributed system there are several approaches to deal with faults of nodes. It may be necessary that the functionality is kept up permanently. In this case, a *masking* approach is required. This category of fault tolerance hides all errors from the application, the system stays operational without restrictions. In case the continuous effective operation of the system is too expensive to guarantee or not essential a *non-masking* solution is possible: Such an approach accepts that the system does not work properly for a given time span, it suffices that it will resume its normal behavior when the fault is resolved. These two strategies lead to two major categories of fault tolerant algorithms [Tix09]:

1. *Robust algorithms* have a redundant layout for all critical components or calculations based on the expected error rate. Hence, if the system is hit by a bounded number of faults, the spare components keep the system running. Usually, robust algorithms follow a masking strategy. However, apart from being more expensive than non-masking approaches due to the additional resources for redundancy, robust algorithms require a clear concept of the (number of) errors that may occur. For instance, an algorithm that uses triple modular redundancy [vN56] can only cover up an error on a single component and may not work if another module fails.

2. *Self-stabilizing algorithms* follow a non-masking error strategy and assume all errors to be transient (cf. Section 2.2.1). Hence, no assumptions about their

nature or extent have to be made. An algorithm is self-stabilizing if it can start in any possible configuration, reaches a legitimate configuration in a finite number of steps by itself without any external intervention, and remains in a legitimate configuration [Dij74, Dol00]. Note that being able to start from any configuration implies that a self-stabilizing algorithm cannot rely on explicit initialization of variables.

The self-stabilization approach was presented by Dijkstra in [Dij74]. It did not attract much attention at first but became more and more popular in the late 1980s and has registered an increase in research activity recently [Dol00]. Some important results for classical graph problems are listed in Section 2.4.

The following definition allows to precede the formal introduction to self-stabilization with a real-world example: According to Arora and Gouda, an algorithm is self-stabilizing if the following two properties hold [AG93]:

- Convergence property: After a finite number of moves the system is in a legitimate configuration irrespective of the configuration the algorithm starts with if no further transient error occurs.
- Closure property: If the system is in a legitimate configuration, this property is preserved if no further transient error occurs.

Figure 2.1 demonstrates these properties using a well-known example. A wobbly man fulfills the convergence property since it always returns to its balanced position irrespective of its initial displacement. Having reached its stable state it will not start leaving this position by itself, hence the closure property also holds.

Note that a self-stabilizing algorithm may not be able to establish a legitimate configuration at all if faults occur too frequently, i.e. if the next error occurs before the algorithm has stabilized. Gärtner states that self-stabilizing algorithms can also deal with certain classes of permanent faults, e.g. when there is a sufficiently long error-free period of time [Gär98]. In principle this complies with the assumptions made in most publications about self-stabilization which consider all errors to be transient, i.e. no further error occurs during the stabilization process.

In the literature, two types of self-stabilizing algorithms can be found: *Silent* (or *static*) self-stabilizing algorithms stop when they have reached a legitimate configuration, i.e. no node will change its state with respect to this algorithm until the next

Figure 2.1: A real-world example for self-stabilization: A wobbly man (drawing by Christian Renner) always returns to its balanced position in finite time without external intervention, if no further impulse hits it.

fault occurs. Hence, the wobbly man (Figure 2.1) also serves as an example for a silent algorithm. Most algorithms that establish a structure on the graph, such as e.g. a matching, are silent. All self-stabilizing algorithms presented in this thesis are silent.

A *reactive* (or *dynamic*) algorithm does not terminate at all. However, it is guaranteed that once a legitimate configuration is reached, the set of legitimate configurations cannot be left. A common example for a reactive self-stabilizing algorithm is mutual exclusion [Dij74, DGT04].

2.2.3 Terms and Definitions

This section introduces the technical terminology of the area of self-stabilizing algorithms. A formal model of these terms is required by some of the proofs in this thesis. To establish a balance between mathematical symbols and readability, all terms are illustrated with the help of an intuitive self-stabilizing algorithm.

In a distributed system the communication relation is represented by an undirected graph $G = (V, E)$, with $n = |V|$ and $m = |E|$, where each process is represented by a node in V and two processes v_i and v_j are adjacent if and only if $\langle v_i, v_j \rangle \in E$. The set of neighbors of a node $v \in V$ is denoted by $N(v)$. The closed neighborhood of a node v is denoted by $N[v] = \{v\} \cup N(v)$. The diameter of G is denoted by $\mathcal{D}$ and the maximum degree of G is denoted by Δ.

In [Tur07] Turau presented a self-stabilizing algorithm for the calculation of a maximal independent set of a graph. It is shown in Algorithm 2.1. A subset $S \subseteq V$ forms

an *independent set* if no two nodes of S are adjacent. S is a *maximal independent set* if $S \cup \{v\}$ is not independent for any $v \in V \backslash S$. Figure 2.5 on page 27 shows a maximal independent set. Detailed information on such sets is provided in Section 2.4.1). The technical terms will now be explained one by one.

Algorithm 2.1 Self-Stabilizing Maximal Independent Set

Predicates:
$inNeighbor(v) \equiv \exists w \in N(v) : w.status = \mathsf{IN}$
$waitNeighborWithLowerId(v) \equiv \exists w \in N(v) : w.status = \mathsf{WAIT} \wedge w.id < v.id$
$inNeighborWithLowerId(v) \equiv \exists w \in N(v) : w.status = \mathsf{IN} \wedge w.id < v.id$

Functions:
–

Actions:
$\mathsf{R}_1 :: [status = \mathsf{OUT} \wedge \neg\, inNeighbor(v)]$
$\longrightarrow status := \mathsf{WAIT}$

$\mathsf{R}_2 :: [status = \mathsf{WAIT} \wedge inNeighbor(v)]$
$\longrightarrow status := \mathsf{OUT}$

$\mathsf{R}_3 :: [status = \mathsf{WAIT} \wedge \neg\, inNeighbor(v) \wedge \neg\, waitNeighborWithLowerId(v)]$
$\longrightarrow status := \mathsf{IN}$

$\mathsf{R}_4 :: [status = \mathsf{IN} \wedge inNeighbor(v)]$
$\longrightarrow status := \mathsf{OUT}$

Definition 1 (State)**.** *All nodes $v \in V$ maintain a set $\{var_1, var_2, \ldots, var_k\}_v$ of variables, each of them ranging over a fixed domain of values. The* state s_v *of the node is represented by the values of its variables.*

In the example above, the state of a node consists of a single variable *status*. The values lie in the range of IN, WAIT and OUT. In Figure 2.2 these values correspond to the colors black, gray and white. The values IN and OUT indicate whether a node is part of the maximal independent set or not, WAIT is an intermediate value that indicates that a node wants to change its *status* to IN. When Algorithm 2.1 has terminated, all nodes have their *status* variable set to either IN or OUT. If no ambiguity arises, the assignment of a value to a variable is sometimes written as an

assignment to the node, i.e. in Figure 2.2 node v_0 has the value WAIT. The states of all nodes in V represent the state of the distributed system, also called *configuration*.

Definition 2 (Configuration)**.** *A* configuration *c of the graph G is defined as the n-tuple of all nodes' states: $c = (s_{v_1}, \ldots, s_{v_n})$. The set of all configurations in G is denoted by C^G.*

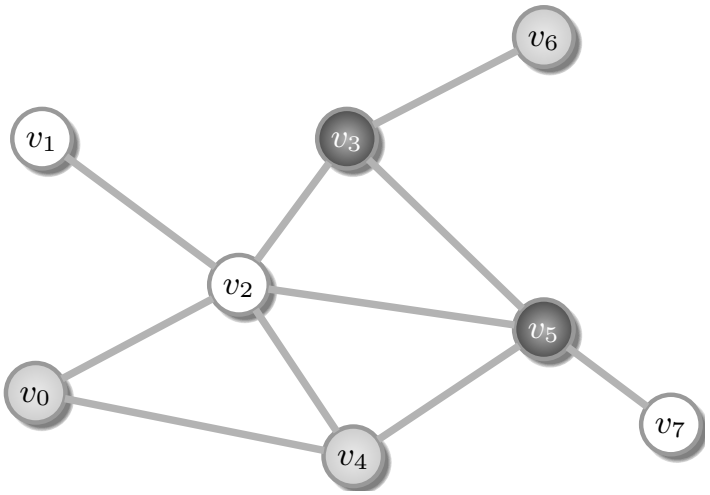


Figure 2.2: Configuration of a graph during the execution of Algorithm 2.1. The colors black, gray and white correspond to the values IN, WAIT and OUT, respectively.

Figure 2.2 shows a configuration of a graph during the execution of Algorithm 2.1. The nodes v_1, v_2 and v_7 have the value OUT assigned to their *status* variable, v_3 and v_5 (resp. the other nodes) have the value IN (resp. WAIT).

The absence of faults can be defined by a predicate $\mathcal{P}$ over the configuration. This motivates the following definition:

Definition 3 (legitimate)**.** *A configuration c is called* legitimate *with respect to $\mathcal{P}$ if c satisfies $\mathcal{P}$. Hence, a legitimate configuration is free of faults. Let $\mathcal{L}_\mathcal{P} \subseteq C^G$ be the set of all legitimate configurations with respect to a predicate $\mathcal{P}$.*

In this case $\mathcal{P}$ must evaluate to *true* if and only if the specified configuration forms a maximal independent set, i.e. for the configuration shown in Figure 2.2 $\mathcal{P}$ is *false* whereas $\mathcal{P}$ is *true* for the configuration depicted in Figure 2.5 (page 27). $\mathcal{L}_\mathcal{P}$ contains all configurations that form an independent set of the graph.

Rules specify the behavior of the nodes. Note that a node can only update its own state.

Definition 4 (Rule)**.** *A* rule *(or* action*) consists of a* name*, a* precondition *(or* guard*) and a* statement*. The precondition of a rule is a Boolean predicate defined on the state of the node itself and its neighbors' states. It decides whether a node is allowed to execute the corresponding statement. The statement describes how a node updates its state.*

The notation of a rule is:

$$\mathsf{Name} :: [\mathit{precondition}] \longrightarrow \mathit{statement}$$

Algorithm 2.1 contains four rules that define in which situations a node has to change the value of its *status* variable.

Definition 5 (Algorithm)**.** *An* algorithm *is a set of rules. It constitutes the program executed on the nodes of the distributed system.*

Definition 6 (enabled)**.** *A rule is called* enabled *in a configuration* c *if its precondition evaluates to* true *in* c*. A node is enabled in a configuration if at least one of its rules is enabled. A rule (resp. node) that is not enabled is called* disabled*.*

If several rules are enabled for a node in a configuration, one rule is nondeterministically chosen for execution. However, algorithms can be designed to guarantee that at most one rule is enabled per node for any configuration. This can be done by extending the guards of the rules to include the negation of the other rules' guards. Hence, without loss of generalization it is assumed that a node is enabled for at most one rule in a given configuration.

In the configuration depicted in Figure 2.2 all nodes are enabled, except for nodes v_2 and v_7. They are disabled since they have a neighbor (e.g. v_5) that is included in the minimal independent set and they themselves are not. Nodes v_4 and v_6 are enabled to execute rule R_2 to set their *status* variable to OUT. The black nodes are neighbors, and hence, both of them are enabled to leave the independent set (rule R_4). Node v_0 could set its *status* to IN via rule R_3 and node v_1 is enabled to execute rule R_1 to set its *status* to WAIT. The execution of a rule by a node is called a *move*.

Definition 7 (Move)**.** *A* move *is a tuple* $(s, s')_v$*, where* s *(resp.* s'*) denotes the state of node* v *before (resp. after) the execution of the statement of an enabled rule.*

If it is clear (or of no relevance) which node executes the move, the subscript will be omitted. If a certain rule is enabled for a given node, the corresponding move is called *enabled* also. An essential property of the system is its *synchrony*. In Figure 2.2 the nodes v_3 and v_5 are both enabled to execute rule R_4. If they make a move simultaneously, both of them set their *status* variable to OUT since they read their neighbors' states at the same time. However, if one of them makes its move first, the other node becomes disabled since it no longer has a black neighbor. The synchrony of a distributed system is modeled by a *scheduler* (or *daemon*). For a given configuration the scheduler chooses which nodes make a move simultaneously.

Definition 8 (Scheduler)**.** *The* scheduler *of a distributed system is a function* $sched : C^G \hookrightarrow 2^V$*, such that* $sched(c)$ *is a nonempty subset of the nodes in* V *that are enabled in configuration* c*.*

The most common schedulers are:

- the *central* scheduler: At any time, only *a single node* makes its move, i.e $\forall c \in C^G : |sched(c)| = 1$.
- the *synchronous* scheduler: *All* enabled nodes make their moves simultaneously.
- the *distributed* scheduler: *Any nonempty subset* of the enabled nodes can make their moves simultaneously.

Although it is easier to prove stabilization for algorithms working under the central scheduler, the synchronous and the distributed scheduler are more suitable for practical implementations. The distributed scheduler allows the nodes to operate with different speed, i.e. not all nodes have to make their move at the same time. Note that the distributed scheduler subsumes the other two types of schedulers and is the most general concept. In general, schedulers have no restrictions on their scheduling policy. However, sometimes it is useful to assume *fairness*:

Definition 9 (Fairness)**.** *A scheduler is called* fair *if it prevents a node being continuously enabled without making a move. Otherwise, the scheduler is called* unfair.

The results presented in this thesis are valid for the unfair distributed scheduler if not explicitly stated otherwise.

Self-stabilizing algorithms operate in *steps*. Intuitively, steps can be seen as time intervals, such that every node can make at most one move within one step and such that all nodes make their move simultaneously. This implies that for any step all nodes read their neighbors' states at the same time.

Definition 10 (Step)**.** *A* step *is a tuple* (c, c')*, where* c, c' *are configurations, such that*

- *all nodes that make a move in this step are enabled in configuration* c*, and*
- c' *is the configuration reached after these nodes have made their move* simultaneously.

When the central scheduler is used, each step consists of the move of a single node only. Thus, if a step consists of the move $\mathsf{m} = (s, s')$ that transforms configuration c_0 into c_1 it is also possible to write $\mathsf{m} = (c_0, c_1)$ and with a slight abuse of notation $\mathsf{m}(c_0) = c_1$. This notation does not introduce any ambiguity when the central scheduler is used, since c_0 and c_1 coincide in all components but one.

Definition 11 (Execution)**.** *An* execution *of an algorithm is a maximal sequence* $c_0, c_1, \ldots$ *of configurations such that for each configuration* c_i *the next configuration* c_{i+1} *is obtained from* c_i *by a single step.*

With these terms and definitions it is possible to describe the two properties *closure* and *convergence* (cf. Section 2.2.2) more formally, which are used to give a formal definition of self-stabilization:

Definition 12. *An algorithm is* self-stabilizing *with respect to* $\mathcal{P}$ *if the following two properties hold:*

- *Closure property: For all configurations* $c_0, c_1 \in C^G$*: If* (c_0, c_1) *is a step with* $c_0 \in \mathcal{L}_\mathcal{P}$*, then* $c_1 \in \mathcal{L}_\mathcal{P}$*.*
- *Convergence property: For every execution* $c_0, c_1, \ldots$ *there is an integer* i *such that* $c_i \in \mathcal{L}_\mathcal{P}$*.*

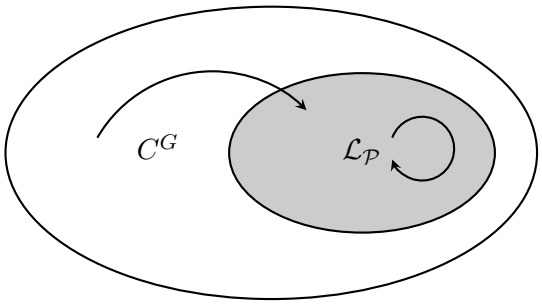


Figure 2.3: Closure and convergence

Definition 12 is illustrated in Figure 2.3: The set $\mathcal{L}_\mathcal{P}$ of legitimate configurations is a subset of C^G, the set of all configurations. Any step starting from a legitimate configuration results in another legitimate configuration. If the initial configuration is not in $\mathcal{L}_\mathcal{P}$, then in a finite number of steps a legitimate configuration is reached.

More details and other elaborative introductions to self-stabilization can be found e.g. in [Dol00], [Tel01], or [Tix09].

2.2.4 Complexity of Self-Stabilizing Algorithms

The *complexity* of an algorithm is a measure for its maximum resource demand. Usually this demand depends on the size of the input or, in case of a distributed algorithm, the number of processors. The considered resources can be time, memory, or the number of messages sent. The latter does not apply in this thesis due to the use of the state model (see Section 2.1) [AW04]. Garey and Johnson contributed the most influential publication on complexity of problems and algorithms [GJ79]. However, they focus on centralized algorithms. The complexity of distributed algorithms with respect to the communication model is discussed e.g. in [AW04]. A detailed introduction to the complexity of self-stabilizing algorithms can be found in [Dol00].

There are several measures for the time complexity of a self-stabilizing algorithm. Note that these measures do not consider local computation of the nodes. This is due to the assumption that the time needed for communication greatly exceeds the time needed for computation, an assumption made for algorithms that consider the computations to be based on local knowledge only. A detailed discussion on this topic can be found in [Tel01]. A standard measure is the *move complexity*.

Definition 13 (Move Complexity)**.** *The (worst-case)* move complexity *of a self-stabilizing algorithm denotes the maximum number of individual moves needed to reach a legitimate configuration irrespective of the initial configuration.*

This upper bound is relevant for many practical applications such as wireless systems with bounded resources. The execution of self-stabilizing algorithms defined for the state model in a wireless setting requires a transformation. The cached sensornet transform (CST) proposed by Herman is a widely used transformation technique [Her04]. It requires that nodes broadcast their state to their neighbors after every move. Since communication is the main consumer of energy, a reduction of the number of broadcasts prolongs the lifetime of a network [TW09].

For the second standard measure for time-complexity of a self-stabilizing algorithm, assume the synchronous scheduler. In this case, in any step *all* enabled nodes make a move. The term *(asynchronous) rounds* tries to extend this idea to match the nature of the central and the distributed scheduler [Dol00]. Starting from a given configuration some nodes may be scheduled several times before all enabled nodes have made a move. Furthermore, since the move of a node can disable other nodes, it does not make sense to require all nodes that were enabled at the beginning of a round to make a move until the round is completed. It also suffices when a node is disabled in between. Note that only for the synchronous scheduler the number of moves per round is limited to the number of nodes, since a round is a single step under this scheduler.

Definition 14 (Round)**.** *A round is a minimal sequence of steps during which any node that was enabled at the beginning of the round has either made a move or has become disabled at least once.*

Definition 15 (Round Complexity)**.** *The (worst-case)* round complexity *of a self-stabilizing algorithm denotes the maximum number of rounds needed to reach a legitimate configuration irrespective of the initial configuration.*

Considering rounds allows to make assumptions on the states of all nodes, e.g. after the first round all nodes have assigned certain values to their variables. The round complexity further permits to ignore scenarios in which a particular node is continuously enabled but does not make a move. The current round does not end unless the node either makes a move or the move of one of its neighbors disables it.

The worst-case number of moves or rounds does not necessarily reflect the time the algorithm needs to stabilize. The number of moves alone does not provide the information whether these moves are equally distributed among all nodes or whether they are performed by a small group of nodes only. Hence, only for the central scheduler, this number conforms exactly with the worst-case stabilization time. On the other hand, a round has no fixed limit for the number of moves contained under the central or the distributed scheduler. Counting the worst-case number of *steps* estimates the time an algorithm needs to stabilize best.

Definition 16 (Step Complexity)**.** *The (worst-case)* step complexity *of a self-stabilizing algorithm denotes the maximum number of steps needed to reach a legitimate configuration irrespective of the initial configuration.*

Note that for the central scheduler the step complexity is equivalent to the move complexity, since this scheduler allows only one move per step. For the synchronous scheduler the step complexity is equivalent to the round complexity, since under this scheduler a round consists of exactly one step. For the distributed scheduler the time a self-stabilizing algorithm needs to reach a legitimate configuration exactly corresponds to the number of steps in the execution. However, since any execution under the central scheduler is also valid for the distributed scheduler, its worst-case number of steps cannot be smaller than the move complexity under the central scheduler. Usually, the step complexity is merely used for the synchronous scheduler to emphasize that the rounds are synchronous.

The last complexity measure considered in this thesis refers to the memory requirement of an algorithm. Often, self-stabilizing algorithms run on very restricted hardware, therefore it is important to use the resources economically.

2.3 Design Methods for Self-Stabilizing Algorithms

The definition of a legitimate configuration for a given problem is usually described by several individual properties that have to hold true. In general, a self-stabilizing algorithm consists of a set of rules that perform a local check whether a precondition of a rule is valid for the executing node and set the state accordingly, if necessary.

However, self-stabilizing algorithms can be designed in very different ways. This section presents common techniques for the development of such algorithms.

As presented in [Mje08], early approaches to find a general mechanism to make any distributed algorithm self-stabilizing aim to detect errors in the global configuration and reset the whole system if required: In [KP90] this is done via a global snapshot, i.e. one node temporarily gathers the state of all nodes. The node decides whether a global reset is necessary or not and informs the other nodes. In [APSVD94] a similar approach is used but the check whether the system is in a legitimate configuration is done locally. These techniques require a lot of time and memory (cf. the following paragraph), and in addition they do not consider the possibility to resolve an error locally, i.e. without restarting the whole network.

In Section 2.1 two methods were outlined to turn any sequential algorithm into a distributed algorithm. These techniques can also be extended to suit the self-stabilizing paradigm: Within $O(\mathcal{D})$ rounds any node can gather the state of all other nodes, where $\mathcal{D}$ denotes the diameter of the system. Then, with local computation the nodes can determine and set their target state. Such an approach suffers from the same disadvantages as listed in Section 2.1. Apart from the fact that it requires unique identifiers it leads to a space complexity of $\Omega(m)$ which is undesired. The goal of a local algorithm is to be scalable, e.g. the memory requirement should be in $O(\Delta)$.

In the following sections so-called *transformers* will be used to make algorithms match certain model assumptions. A model is called *weaker* (resp. *stronger*) than another if it is less (resp. more) restrictive than the other. For instance, a system with a central daemon is stronger than a system that assumes a distributed daemon. A *transformer* $\mathcal{T}$ converts a self-stabilizing algorithm $\mathcal{A}$ to a new self-stabilizing algorithm $\mathcal{A}'$, such that $\mathcal{A}'$ runs under a weaker model than $\mathcal{A}$. $\mathcal{T}$ must *preserve* legitimacy, that is to say, $\mathcal{A}$ and $\mathcal{A}'$ share the same set of legitimate configurations. In general, using a transformer to make an algorithm suit a weaker model is attended by a slowdown in stabilization time. This will be explained with more detail in the following sections.

2.3.1 Composition

Often, algorithms are composed of several stages that achieve particular sub-ordinate targets, each of them being the precondition for the next stage. In [Tel01] Tel lists

common examples for the first stage of such composed algorithms, e.g. algorithms may rely on correct routing tables, an elected leader, a snapshot of the system or an acyclic orientation of the graph.

In a classical distributed system these algorithms can be executed one after the other by installing a distributed termination-detection algorithm which ensures that the first stage is completed. Unfortunately, it is impossible to detect termination of the first algorithm in a self-stabilizing manner ([Tel01]). However, it is possible to compose two self-stabilizing algorithms in the following manner ([Her92, Tel01]):

Definition 17 (Composition)**.** *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be self-stabilizing algorithms, such that no variable that is written by $\mathcal{A}_2$ occurs in $\mathcal{A}_1$. The* composition *of $\mathcal{A}_1$ and $\mathcal{A}_2$ is the algorithm that consists of all variables and all actions of both $\mathcal{A}_1$ and $\mathcal{A}_2$.*

Theorem 2.3.1. *The composition of two self-stabilizing algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ is self-stabilizing if the following properties hold:*

- *When Algorithm $\mathcal{A}_1$ has stabilized, property $\mathcal{P}_1$ holds forever.*
- *When property $\mathcal{P}_1$ holds, Algorithm $\mathcal{A}_2$ stabilizes.*
- *Algorithm $\mathcal{A}_1$ does not change any variables Algorithm $\mathcal{A}_2$ reads once $\mathcal{P}_1$ holds (trivial if Algorithm $\mathcal{A}_1$ is a silent algorithm).*
- *The scheduler is fair with respect to both algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$.*

The proof for Theorem 2.3.1 can be found in [Her92, Tel01]. Obviously, the result also holds if both algorithms are silent and stabilize under an unfair scheduler and Algorithm $\mathcal{A}_2$ terminates regardless of the variables set by Algorithm $\mathcal{A}_1$.

The move complexity of a self-stabilizing composed algorithm is the product of the complexities of the individual algorithms [Dol00].

2.3.2 Distance-k Information

According to the model of computation of distributed algorithms, a node has read access only to its own variables and those of its neighbors (*distance-one model*). However, for certain problems it is easier to design an algorithm assuming that a node can even read the variables of nodes that are two or more hops away or assuming that

the values of its neighbor's variables are correct. To make such an algorithm run in a distributed system it has to be transformed. Several transformers can be found in the literature. The functional principle of such a transformer is to provide the nodes with additional variables that gather information about the state of their neighbors. Via these variables the state of a node (or at least parts of its state) can be seen by its neighbors' neighbors. All known transformers require (locally) unique identifiers.

To retrieve distance-two information, in [GGH+04], apart from its own state each node holds a copy of its neighbors' states. Whenever necessary, a node has to update this copy. Furthermore, the node can signal its will to execute a move itself or it can allow one of its neighbors to make a move. A node can execute the algorithm only if all neighbors have given their permission. This way it is guaranteed that whenever a node executes the algorithm, its neighbors have their copies up-to-date. (If a node gives the right to execute a move to one neighbor, no other neighbors can make a move themself.) A major drawback of this approach is the slowdown factor of $O(n^2m)$ moves and the memory overhead of $\Omega(\Delta \log n)$ per node.

In [GHJT08], the approach of [GGH+04] is extended. Its recursive application allows to retrieve distance-k knowledge. The memory requirement and the slowdown factor in moves are both in $n^{O(\log k)}$.

Recently, a new model to access distance-two information was published, the *expression model* [Tur12]. In this model a node does not only have variables but it also holds a set of named expressions. The value of an expression is based on the state of the node and the state of its neighbors. A node cannot directly read the variables of a node two hops away but it can evaluate the expressions of its neighbors. The distance-two model [GGH+04] is a special case of the expression model since it is possible to define an expression that returns the state of all neighbors. Hence, the expression model has the same expressiveness as the distance-two model. An advantage of the expression model is the slowdown factor of only $O(m)$ moves. The memory overhead can be adapted to the given algorithm.

The three models above are discussed in detail in [Tur12]. Since the aggregation of information is attended by a slowdown factor for every hop (see e.g. [GHJT08]), more than 2-hop information is rarely used. However, there are self-stabilizing algorithms that assume a node to have read access to the 4-hop neighborhood, e.g. [UT11, GHJT08].

2.3.3 Scheduler Transformation

Some algorithms that stabilize under the central scheduler do not stabilize under a synchronous scheduler, e.g. the coloring algorithm in [GK93]. In this algorithm the nodes can always choose the same color and hence, it may never terminate (Figure 2.4).

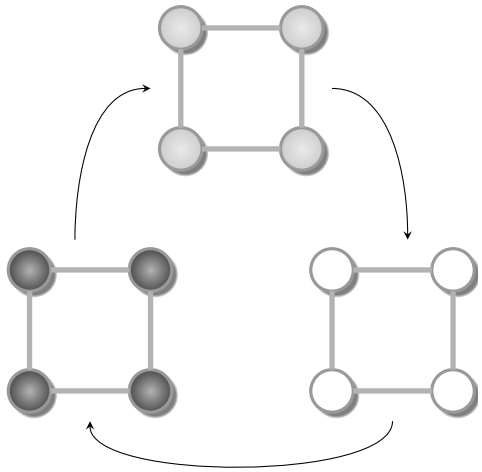


Figure 2.4: The coloring algorithm in [GK93] does not stabilize under the synchronous scheduler if all nodes continually choose the same color.

Assuming a central scheduler is often convenient during the design of a self-stabilizing algorithm. Problems that arise due to the simultaneous execution of a rule by two neighbors do not have to be considered when only one node at a time is allowed to move. Several transformers exist that convert an algorithm designed for the central scheduler into an algorithm that stabilizes under the distributed scheduler. Note that all known transformers for this purpose require the nodes to have (locally) unique identifiers, and hence they are not applicable in anonymous networks. Since the distributed scheduler subsumes all other schedulers, transformations from the distributed scheduler to the central scheduler are not necessary.

In [BDGM00], a self-stabilizing local mutual exclusion algorithm is developed. The authors show that a specific composition scheme can transform an algorithm for the central scheduler into a version that runs under the distributed scheduler through combination with the mutex algorithm. However, the transformation slows down the algorithm by a factor of $O(n^2)$.

The conflict manager in [GT07] basically works in the following way: A node that wants to execute a move indicates this via an extra move that sets a Boolean flag. In the next move the node checks if it is the one with the largest identifier among the nodes that have set their flag. Only in that case it is allowed to execute the move. The conflict manager requires one bit of extra memory per node and leads to a slowdown factor of $O(\Delta)$ moves.

The distance-two transformation of [GGH+04] (resp. the expression model of [Tur12]) can also be used to make a scheduler transformation from the central scheduler to the distributed scheduler since it guarantees mutual exclusion for adjacent nodes. However, the slowdown factor in this case is n^2m moves (resp. $O(m)$ moves). Thus, if the algorithm does not benefit from 2-hop knowledge, the conflict manager of [GT07] is more efficient for scheduler transformation.

2.4 Self-Stabilizing Algorithms for Classical Graph Problems

The first self-stabilizing algorithm was presented by Dijkstra to establish *mutual exclusion* in a ring topology [Dij74]. Mutual exclusion is a fundamental problem of concurrent programming [Dij65]. It assumes that several nodes need to have access to a common resource but only one node is allowed to use it at a time. Hence, in case several nodes want to use the resource concurrently, it must be ensured they access it one after the other.

From the late 1980s on, the field of self-stabilization has attracted a lot of research activity. Self-stabilizing algorithms have been applied to different fields such as device drivers, operating systems and wireless sensor networks [DY06, Yag07, TW09]. The majority of research has focused on distributed algorithms for optimization problems in graph theory such as coloring problems, the minimal dominating set problem and the maximal independent set problem [GT00, Tur07]. This chapter provides a survey of self-stabilizing algorithms for classical graph problems.

Many references can be found in the survey paper of Guellati and Kheddouci [GK10]. They analyze self-stabilizing algorithms for independent sets, dominating sets, colorings and matchings. The survey of Gärtner [Gär03] examines self-stabilizing algorithms for spanning trees. Their results are summarized in this thesis. Further

literature is discussed by Dolev [Dol00]. Tixeuil provides more references [Tix09]. The self-stabilization bibliography by Herman [Her02] lists about 500 self-stabilizing algorithms ordered by several categories (e.g. topology or proof techniques). The results of this section are subsumed using the same details as in [GK10], i.e. the depicted characteristics are result type, required topology, anonymity, daemon type and complexity of an algorithm.

In the context of self-stabilization, anonymous algorithms are very difficult to design. In [SRR94] it is shown that it is impossible to colorize a path with even length with two colors using a deterministic self-stabilizing algorithm under the distributed scheduler. More impossibility results can be found in [SRR95]. Note that several algorithms mentioned below are marked as anonymous since they do not require (locally) unique identifiers, but they assume a *central* scheduler. Such a scheduler trivially breaks symmetry in a distributed system. Hence it is easily possible to generate identifiers by letting a node choose the smallest integer that is not used by its neighbors. However, the algorithms do not use such a mechanism since they do not depend on identifiers. This also implies that these algorithms do not make use of *pointers* from one node to one of its neighbors, which is usually implemented by storing the neighbor's identifier.

2.4.1 Independent Sets

A subset S of vertices of a graph is called *independent* if no two nodes of S are adjacent. S is called *maximal* (MIS) if no further node can be added to S without violating this condition (see Table 2.5). A MIS whose cardinality cannot be increased by removing one node and adding more nodes is called *1-maximal* (1-MIS). If S is a MIS then any node is either in S or has a neighbor in S. Thus, any maximal independent set is also a dominating set (cf. Section 2.4.2). This makes them an important structure for e.g. wireless ad hoc networks [AWF03]. Furthermore, MIS are used to establish mutual exclusion and hence for conflict-avoiding problems such as scheduling. There are several self-stabilizing algorithms in the literature that calculate independent sets. Table 2.1 is taken verbatim from the survey paper by Guellati and Kheddouci [GK10]. It lists self-stabilizing algorithms for the maximal independent set problem classified by several characteristics. A detailed discussion of the algorithms listed in this table can be found in [GK10]. Further details are given in [Tur07].

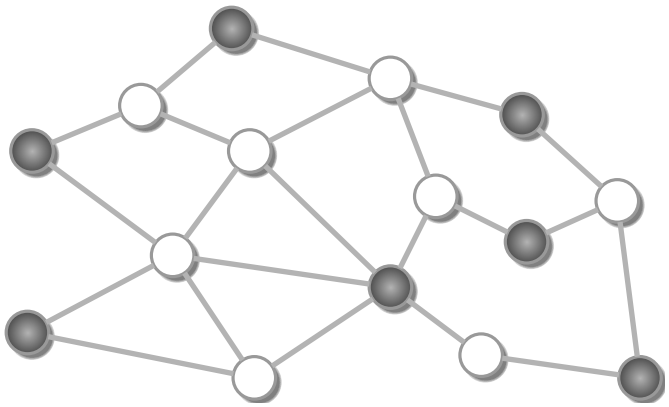


Figure 2.5: Maximal independent set S of a graph. The nodes in S are colored black.

The MIS algorithm of [HHJS03] is identical to that of [SRR95], hence it does not appear in Table 2.1 on its own. Note that in [GK10] the algorithm of [LH03] is mentioned but not included in their table. This algorithm is a fault-containing version of the MIS algorithm in [SRR95], i.e. recovery from a single transient fault is achieved quickly ($O(\Delta)$ moves). However, the stabilization time starting from an arbitrary configuration is not analyzed.

Reference	Result	Req. topology	Anon.	Daemon	Complexity
[SRR95]	MIS	arbitrary	✓	central	$O(n)$ moves
[IKK02]	MIS	arbitrary	–	distributed	$O(n^2)$ moves
[GHJS03d]	MIS	arbitrary	–	synchron.	$O(n)$ rounds
[SGH04]	1-MIS	tree	✓	central	$O(n^2)$ moves
[Tur07]	MIS	arbitrary	–	distributed	$O(n)$ moves
[LH03]	MIS	arbitrary	–	central	unknown

Table 2.1: Self-stabilizing algorithms for the maximal independent set problem. Source: [GK10]

2.4.2 Dominating Sets

A subset S of vertices of a graph $G = (V, E)$ is called *dominating* (DS) if every node in V is either contained in S or it has a neighbor in S. There is a wide variety of domination parameters of a dominating set that can be defined [HL91]. The set

S is called a *total dominating set* (TDS) if every node of the graph has a neighbor in S. The set is *k-dominating* (KDS) if every node has at least k neighbors in S. If a dominating set is connected it is called *connected dominating set* (C-). It is called *weakly connected* (WC-) if the subgraph weakly induced by S, i.e. the graph $(N[S], E \cap (S \times N[S]))$ is connected. A dominating set is *minimal* (M-) if for any node $v \in S$ the set $S \backslash \{v\}$ is not dominating. More details can be found in Chapter 4, where a new algorithm for the WCMDS problem is presented. Figure 2.6 shows a MDS.

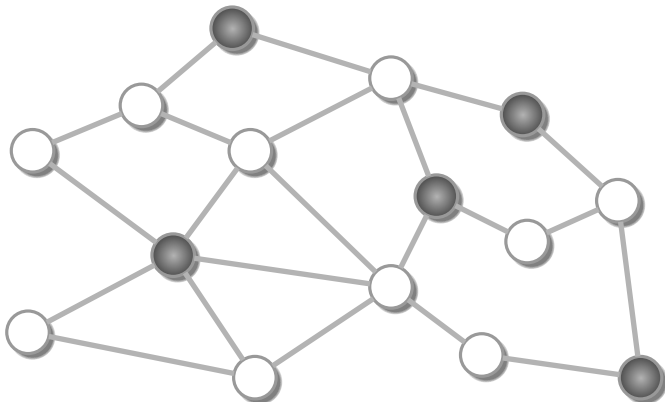


Figure 2.6: Minimal dominating set S of a graph. The nodes in S are colored black.

Dominating sets are an important structure that is often used for efficient communication in wireless and ad hoc networks [AWF03, WL99, UT11]. Hedetniemi and Laskar have gathered more than 300 references for algorithms that calculate various types of dominating sets [HL91]. These algorithms are not self-stabilizing, though.

The survey paper of Guellati and Kheddouci [GK10] considers several self-stabilizing algorithms for the dominating set problem. The upper part of Table 2.2 is taken almost verbatim from this paper, which also includes a detailed discussion of the referenced algorithms. A further discussion of self-stabilizing algorithms for the calculation of a k-dominating set can be found in [Tur12]. Some algorithms for the k-dominating set problem did not appear in [GK10] since they were published later: The algorithm in [DDH+11] is specifically designed to find small k-dominating sets and guarantees an upper bound of at most $\left\lceil \frac{n}{k+1} \right\rceil$ on the size of the calculated set. In [DLV10] Datta et al. present an algorithm with fast stabilization time ($3k + O(1)$ rounds) and little memory overhead ($O(k \log n)$ space per node). Furthermore they prove that no comparison-based algorithm for the k-clustering problem can approx-

imate the optimal solution within $O(\mathcal{D})$ rounds. In [Tur12] Turau introduces the *two-expression model* that assumes distance-two information and a central scheduler. Furthermore he presents a transformation technique to make the algorithms run under the conventional model of computation. Additionally, this paper contains an applications section where a new algorithm for the k-dominating set problem is proposed.

Table 2.2 also contains self-stabilizing algorithms for the calculation of connected, weakly connected and other dominating sets. These parts are new, since these types are not analyzed with the same detail in [GK10].

In the following, the self-stabilizing algorithms for connected and weakly connected minimal dominating sets listed in Table 2.2 are discussed. Note that the algorithm in [KK07b] also guarantees an approximation ratio of $7.6 \cdot \left|D_{opt}\right| + 1.4$ (where D_{opt} is an optimal solution in terms of cardinality) if it runs on a unit disk graph. The algorithm in [KK08] is similar to that in [KK07b] but features *safe convergence*, i.e. the algorithm establishes a particular safe state in short time, and this property holds forever. In this case a dominating set is established after one round. Furthermore the algorithm incorporates the creation of a BFS tree. The same safe convergence property holds for the WCMDS algorithm in [KK07a]. Furthermore, on a unit disk graph an approximation ratio of 5 with respect to the solution with minimum cardinality is guaranteed.

The model of computation in [JG05] assumes a node to have instant read access in its 3-hop neighborhood and write access in its 2-hop neighborhood. The algorithm in [DFG06] also assumes 2-hop read access for the nodes. To make these algorithms run under a more realistic model, a transformer is needed that increases the complexity of the proposed algorithms.

In [RTAS09] a disk graph with bidirectional links (DGB) is assumed. This model is closely related to unit disk graphs but allows the nodes to have different ranges. The authors prove a constant approximation ratio for their algorithm.

The algorithm in [HS11] finds two disjoint minimal dominating sets. The approach identifies the first MDS via the algorithm in [HHJS03]. Then, the remaining set of nodes is reduced to also become a minimal dominating set.

In a *distance-k dominating* set a node is dominating itself or it has a dominating node within its k-hop neighborhood, i.e. the dominating nodes have a larger domination

Reference	Result	Req. topology	Anon.	Daemon	Complexity
Dominating Sets					
[HHJS03]-1	DS	arbitrary	✓	central	$O(n)$ moves
[HHJS03]-2	MDS	arbitrary	✓	central	$O(n^2)$ moves
[XHGS03]	MDS	arbitrary	–	synchron.	$O(n)$ rounds
[GHJS03b]	MTDS	arbitrary	–	central	unknown
[Tur07]	MDS	arbitrary	–	distributed	$O(n)$ moves
[GHJ+08]	MDS	arbitrary	–	distributed	$O(n)$ moves
k-Dominating Sets					
[KK03]-1	MKDS	tree	✓	central	$O(n^2)$ moves
[KK03]-2	MKDS	tree	–	distributed	$O(n^2)$ moves
[GGHJ04]	MKDS	arbitrary	✓	central	$O(kn)$ moves
[HCW08]	M2DS	arbitrary	✓	central	$O(n)$ moves
[KK05]	MKDS	$\delta > k$	–	synchron.	$O(n^2)$ moves
[HLCW07]	M2DS	arbitrary	–	distributed	unknown
[DLV10]	MKDS	arbitrary	–	distributed	$3k + O(1)$ rounds
[DDH+11]	MKDS	arbitrary	–	distributed	$O(n)$ rounds/ $O(\mathcal{D}n^2)$ moves
[Tur12]	MKDS	arbitrary	–	distributed	$O(mn)$ moves
Connected Dominating Sets					
[JG05]	CDS	arbitrary	–	synchron.	$O(n^2)$ rounds
[DFG06]	CDS	arbitrary	–	distributed	$O(n)$ moves
[GS10]	CDS	arbitrary	✓	distributed	unknown
[KK07b]	CMDS	BFS tree	–	central	$O(k)$ rounds, $k =$ depth of BFS tree
[KK08]	CMDS	arbitrary	–	synchron.	$O(n)$ rounds
[RTAS09]	CMDS	DGB	–	central	$O(n^2)$ moves
[SX07]	WCMDS	BFS tree	–	distributed	$O(2^n)$ moves
[KK07a]	WCMDS	arbitrary	–	synchron.	$O(n^2)$ rounds
Algorithm 4.4	WCMDS	BFS tree	–	distributed	$O(mn)$ moves
Other Dominating Sets					
[HS11]	2 disjoint MDS	arbitrary	–	central	$O(n^4)$ moves
[LHWC08]	dist.-2 MDS	arbitrary	–	central	unknown

Table 2.2: Self-stabilizing algorithms for the minimal dominating set problem. Source of the first two parts: [GK10]

range. Such a set is calculated in [LHWC08]. An upper bound on the stabilization time is not given in this paper. The authors believe it to be polynomial.

A revised version of [SX07] was published in [XWS10], however, the algorithm and the included analysis did not change. Hence, it is not included in Table 2.2. This algorithm is discussed in Chapter 4.

2.4.3 Spanning Trees

A connected subgraph T of a graph G is called a *spanning tree* (ST) if it comprises all vertices of G and it does not contain a circle. There are several types of spanning trees which are defined in the following. Note that some kinds of trees require one node to have a special role, called *root*. Let T be a spanning tree of $G = (V, E)$ with root r. If the number of edges between r and all other nodes is minimal, T is called a *breadth-first spanning tree* (BFS). If for any two nodes $v, w \in V$ with $(\langle v, w \rangle \in E$ the path from v to w in T does not contain r, T is a *depth-first spanning tree* (DFS) of G.

Assume the edges of G to have non-negative *weights*. T is a *minimum spanning tree* (MST) if the sum of the weights of its edges is minimal among all spanning trees. If the distance (i.e. the sum of the weights of the edges) between r and all other nodes is minimal, T is called a *shortest-paths spanning tree* (SP-ST). Figure 2.7 shows a simple spanning tree of a graph.

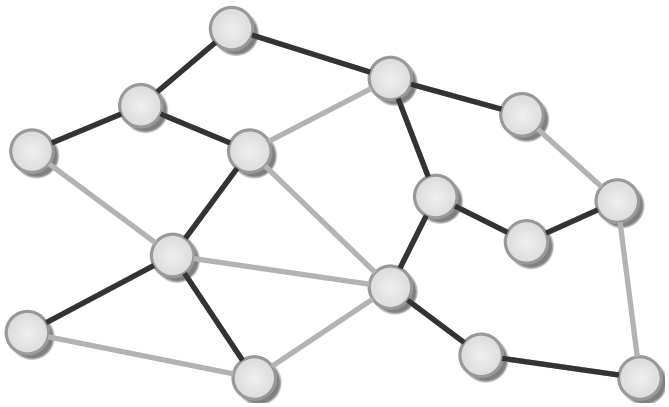


Figure 2.7: Spanning tree T of a graph. The edges of T are colored black.

Many algorithms rely on a spanning tree as an underlying network topology. In [GGKP95] and [BM03] a general technique is presented that transforms any sequential

bottom up dynamic programming algorithm into a self-stabilizing algorithm for a tree network. Gärtner provides a survey and detailed discussions on self-stabilizing algorithms for the construction of spanning trees up to the year 2003 [Gär03]. Table 2.3 categorizes the algorithms of this survey paper and adds the spanning tree algorithms published since then. In the following, the algorithms not discussed in [Gär03] are surveyed. It is noteworthy that a lot of publications for these problems do not provide an upper bound for the time complexity of the proposed algorithms.

There are two new algorithms for the construction of a minimum spanning tree. In [BPBRT10] a general scheme is introduced that allows to develop loop-free and super-stabilizing algorithms for the spanning tree problem that can be adapted for any tree metric, such as e.g. shortest-path tree, minimum spanning tree, maximum-flow tree or minimum-degree spanning tree. This scheme is based on the combination of a new BFS algorithm that is also presented in [BPBRT10]. The resulting algorithm stabilizes in $O(n^3)$ rounds. In the same year, another algorithm for the minimum spanning tree problem was published in [BDPBR10]. It is a self-stabilizing version of the classical distributed algorithm by Gallager, Humblet and Spira [GHS83]. The algorithm requires only $O(n^2)$ rounds.

An algorithm that constructs a spanning tree with any maximizable metric is presented in [DMT11]. Furthermore, the algorithm can deal with Byzantine faults.

In [KKDT10] the first self-stabilizing algorithm specifically designed for the maximum-leaf spanning tree problem is presented. It guarantees an approximation ratio of 3 and stabilizes in at most $O(n^2)$ rounds.

Three self-stabilizing spanning tree algorithms have been published that assume the message-passing model: A solution for the construction of constant-degree spanning trees in large-scale systems is given in [HLP+06]. Note that this algorithm requires a complete graph. A shortest-paths tree is established in [BK07]. This algorithm produces such a tree, rooted at the node with minimal identifier, in $O(\mathcal{D})$ rounds. However, the algorithm does not stabilize in that time. Instead it requires an unknown time to reach stabilization. The first algorithm for the minimum-degree spanning tree problem is presented in [BPBR11]. It guarantees a result within 1 from the optimal degree. The algorithm has a memory complexity of $O(\Delta \log n)$ and works for any topology.

Reference	Result	Req. topology	Anon.	Daemon	Complexity
Simple, Breadth-first, and Depth-first Spanning Trees					
[DIM90, DIM93]	BFS-ST	arbitrary	–	central	$O(\mathcal{D})$ rounds,
[Her92]	DFS-ST	arbitrary	–	central	$O(\mathcal{D}nK)$ rounds
[AKY90]	BFS-ST	arbitrary	–	distributed	$O(n^2)$ rounds
[AG90, AG94]	BFS-ST	arbitrary	–	central	$O(N^2)$ rounds
[CYH91]	ST	arbitrary	–	central	unknown
[SS92]	BFS-ST	arbitrary	–	distributed	unknown
[AS92]	ST	arbitrary	–	central	unknown
[ABB97, ABB98]	D/BFS-ST	arbitrary	–	distributed	$O(n)$ rounds
[AK93]	ST	arbitrary	–	distributed	$O(\mathcal{D})$ rounds
[GGP96]	ST	arbitrary	✓	central	unknown
[DIM97b]	DFS-ST	arbitrary	✓	central	$O(\Delta\mathcal{D})$
Minimum Spanning Trees					
[AS97a]	MST	arbitrary	–	distributed	unknown
[AS98]	MST	symmetric	–	distributed	unknown
[HL01]	MST	arbitrary	–	distributed	unknown
[BDPBR10]	MST	arbitrary	–	distributed	$O(n^2)$ rounds
[BPBRT10]	MST	arbitrary	–	distributed	$O(n^3)$ rounds
Other Spanning Trees					
[BLB95]	min-diam ST	arbitrary	✓	distributed	$O(n\Delta + \mathcal{D}^2 + n\log\log n)$ rnds
[HLP+06]	const-deg ST	complete	–	central	unknown
[BK07]	SP-ST	arbitrary	–	distributed	unknown
[KKDT10]	max-leaf ST	arbitrary	–	distributed	$O(n^2)$ rounds
[BPBR11]	min-deg ST	arbitrary	–	distributed	$O(mn^2\log n)$ rounds
[DMT11]	max-metric ST	arbitrary	–	distributed	unknown

Table 2.3: Self-stabilizing algorithms for the spanning tree problem. Some algorithms require global information: K is an upper bound for Δ, N is an upper bound for n.

2.4.4 Coloring

A *vertex coloring* is a function $color : V \rightarrow C$, where C is a set of colors, such that $color(v) \neq color(w)$ for adjacent nodes v and w. Figure 2.8 shows a vertex coloring of a graph using three colors. Similarly, it is possible to color the edges of a graph, such that adjacent edges do not share the same color. This is called an *edge coloring*. Colorings are used e.g. for conflict avoiding, scheduling [Mar03] or register allocation [Cha82]. Furthermore, colorings are used for the popular Sudoku game [HM07].

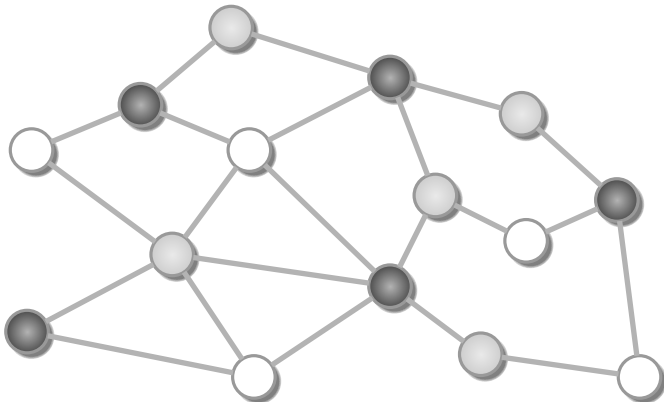


Figure 2.8: Vertex coloring of a graph. Adjacent nodes are not allowed to have the same color.

Guellati and Kheddouci [GK10] illustrate several self-stabilizing coloring algorithms. The upper part of Table 2.4 is taken verbatim from this paper. For a discussion on the vertex coloring algorithms, see [GK10]. A detailed survey of self-stabilizing edge-coloring algorithms is provided by [DDK09] and [HT06]. In the following, the other coloring algorithms (in the lower part of Table 2.4) will be outlined.

The algorithm in [DK08] calculates a *b-coloring* of the system graph. A b-coloring of a graph G is a vertex coloring of G such that for any color there is a vertex that has neighbors in *all* other colors. Note that this algorithm uses distance-two knowledge.

In [BM09] a distance-two coloring is derived, i.e. nodes within distance two are not allowed to have the same color. The algorithms uses at most $\Delta^2 + 1$ colors.

Sun et al. consider a particular NP-complete optimization problem [SEK08]: Assuming the set of colors to be natural numbers, the goal is to find a coloring with a minimal sum of colors. Their algorithm derives a locally minimal color sum which is an upper bound for the optimal solution.

Reference	Req. topology	Anon.	Daemon	Complexity
Vertex Coloring				
[SS93]	bipartite graph	✓	central	unknown
[KK06]-1	bipartite graph	✓	central	$O(mn^3\mathcal{D})$ moves
[KK06]-2	bipartite graph	–	distributed	$O(mn^3\Delta\mathcal{D})$ moves
[GK93]	planar graph	–	distributed	unknown
[HHT05]	planar graph	✓	central	$O(\mathcal{D})$ rounds
[SRR94]-1	odd chain	✓	distributed	unknown
[SRR94]-2	oriented ring	✓	central	unknown
[GT00]-1	arbitrary	✓	central	$O(n\Delta)$ moves
[GT00]-2	arbitrary	–	distributed	$O(n\Delta)$ moves
[GT00]-3	arbitrary	✓	distributed	$O(n\Delta)$ moves
[HJS03]-1	arbitrary	✓	central	$O(m)$ moves
[HJS03]-2	arbitrary	✓	central	$O(n)$ moves
[GHJS04]	arbitrary	✓	central	unknown
Edge Coloring				
[SOM04]	tree	–	central	3 rounds
[KN06]	arbitrary	–	central	$O(\Delta m)$ moves
[MT06]	arbitrary	✓	central	$2\Delta + 2$ rounds
[HT06]	bipartite graph	–	central	$O(n^2m + m)$ moves
[TJH07]	planar, $\Delta \geq 5$	✓	central	$O(n^2)$ moves
[CT07]	arbitrary	–	central	$O(\Delta^2 m)$ moves
[DDK09]	arbitrary	–	central	$O(m(\Delta + n))$ moves
Other Colorings				
[SEK08]	arbitrary	–	central	$O(n\Delta^3)$ moves
[DK08]	arbitrary	–	central	$O(\Delta^2)$ rounds
[BM09]	arbitrary	–	distributed	$O(\Delta^2 m)$ moves
[CT11]	rooted tree	–	central	$O(nh)$ moves, $h =$ height of tree

Table 2.4: Self-stabilizing coloring algorithms. Source of the upper part: [GK10]

Chaudhuri and Thompson present an algorithm for the *L(2, 1)-labeling problem* [CT11]. Here, an ordering of the set of colors is assumed. The colors of the nodes not only have to be unique within distance two, but adjacent nodes also have to choose colors that are at least two apart. For example, if the set of colors is $\{1, 2, \ldots 5\}$, a node with color 3 is not allowed to have a neighbor with colors 2 or 4.

2.4.5 Covering

A subset S of vertices of a graph $G = (V, E)$ is a *vertex cover* if every edge of E is incident to at least one vertex in S. S is a *minimal vertex cover* if there is no vertex cover S' with $S' \subset S$. S is a *minimum vertex cover* if there is no vertex cover with smaller cardinality. Figure 2.9 shows a minimal vertex cover of a graph. The minimal vertex cover problem plays an important role in computer science [NR99] and other disciplines such as bioinformatics, where these structures are used for DNA sequencing [SP07].

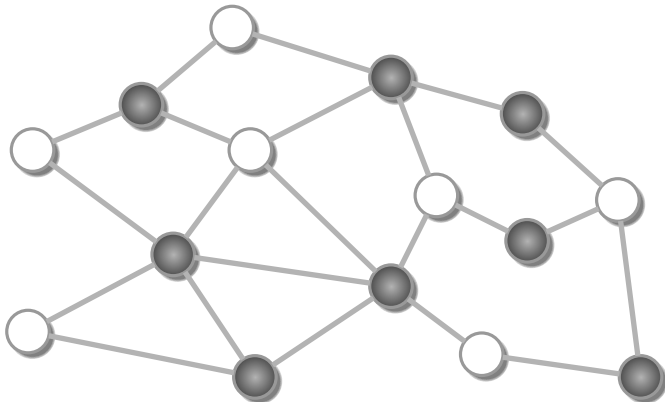


Figure 2.9: Minimal vertex cover S of a graph. The nodes in S are colored black. All edges have at least one node in S.

Since any maximal matching implies a 2-approximation of a minimum vertex cover, all maximal matching algorithms can be used to achieve such a result. Self-stabilizing matching algorithms are presented in Section 2.4.6. Non-self-stabilizing algorithms for the calculation of vertex covers are discussed in Chapter 5.

There are very few self-stabilizing algorithms for the vertex cover problem that do not only compute a maximal matching. Kiniwa presented the first algorithm of this kind in [Kin05]. His algorithm calculates a $(2 - 1/\Delta)$-approximation vertex

Reference	Req. topology	Anon.	Daemon	Complexity
[Kin05]	arbitrary	–	distributed	$\|M\|+2$ rounds, M = calculated matching
Algorithm 5.2	arbitrary	✓	distributed	$O(n+m)$ moves, resp. $O(\Delta)$ rounds

Table 2.5: Self-stabilizing vertex cover algorithms.

cover. Kiniwa combines a greedy method based on a high-degree-first order of vertices with the maximal matching technique. A new algorithm is presented in Chapter 5. Algorithm 5.2 calculates a $(3-2/(\Delta+1))$-approximation vertex cover in anonymous networks.

2.4.6 Matching

Another classical problem in graph theory considers matchings of a graph. Let $G=(V,E)$ be an undirected graph. A set M of independent edges of G is called a *matching* of G. M is a *maximal matching* if there is no matching M' with $M \subset M'$. Figure 2.10 shows a maximal matching of a graph. A matching with largest cardinality is called a *maximum matching*. Let G be a weighted undirected graph. Any edge e has an assigned *weight*, denoted by $w(e) \in \mathbb{R}_+$. The sum of the weights of all edges of a matching M is the weight of M. A matching with heaviest weight is called a *maximum weight matching* of G. Figure 7.1 on page 99 shows a weighted graph and its maximum weight matching.

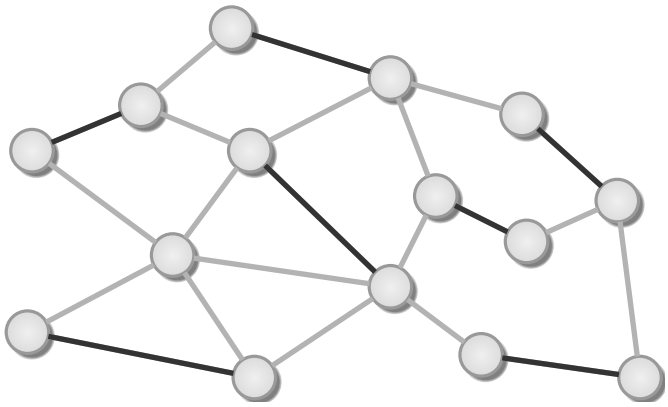


Figure 2.10: Maximal matching M of a graph. The edges in M are colored black.

Maximal matchings are an important structure that is used in diverse fields of application, e.g. for scheduling [SP07] or for multi-channel MAC protocols [HHLL04]. An elaborate discussion about matchings and their applications can be found in [LP86].

Non-self-stabilizing algorithms for the maximal matching problem can be found in Chapter 7. The survey paper by Guellati and Kheddouci [GK10] discusses several self-stabilizing matching algorithms. Their results are listed here. Table 2.6 is taken verbatim from this paper. For details, see [GK10] and Chapter 7.

Reference	Result	Req. topology	Anon.	Daemon	Complexity
[HH92]	maximal	arbitrary	✓	central	$O(m)$ moves
[CHS02]-1	maximal	arbitrary	–	distributed	$O(n)$ rounds
[GHJS03d]	maximal	arbitrary	–	synchron.	$O(n)$ rounds
[GHJS03a]	generalized	arbitrary	✓	central	$O(m)$ moves
[GHS06]	1-maximal	tree	✓	central	$O(n^4)$ moves
[MMPT07], [MMPT09]	maximal	arbitrary	–	distributed	$O(m)$ moves
[KS00]	maximum	tree	✓	central	$O(n^4)$ moves
[CHS02]-2	maximum	bipartite graph	✓	central	$O(n^2)$ rounds
Other matchings					
[MM07]	1/2-approx. max. weight	arbitrary	–	distributed	$O(3^n)$ moves
Alg. 7.3	1/2-approx. max. weight	arbitrary	–	distributed	$O(mn)$ moves
[MMPT08]	2/3-approx. maximum	arbitrary	–	distributed	$O(n^2)$ rounds
[GHJS03c]	strong	arbitrary	–	central	$\geq O(2^{n/3})$ moves

Table 2.6: Self-stabilizing algorithms for the maximal matching problem. Source of the upper part: [GK10]

Currently, there is no self-stabilizing algorithm that calculates a maximum matching on general graphs. There are algorithms for certain classes of graphs, such as trees [KS00] or bipartite graphs [CHS02]. An algorithm for arbitrary graphs is provided in [MMPT08]. It calculates a 2/3-approximation of a maximum matching.

The self-stabilizing algorithm in [GHJS03c] computes a *strong matching*. This is a more restrictive version of the matching problem that specifies that a matched node is not allowed to have any matched neighbors apart from the node it itself is matched to.

In [MM07] Manne and Mjelde proposed an algorithm for the maximum weight matching problem that guarantees an approximation ratio of $1/2$. The move complexity of this algorithm was stated to be exponential for both the central and the distributed scheduler. In Chapter 7 it is shown that the algorithm indeed stabilizes after $O(mn)$ moves under the central scheduler. Furthermore, a modified version of this algorithm (Algorithm 7.3) also stabilizes after $O(mn)$ moves under the distributed scheduler.

Chapter 3

Analysis of Self-Stabilizing Algorithms

Proving self-stabilization of an algorithm and determining its complexity is a challenging task. A famous example is the mutual exclusion protocol in [Dij74]: Its proof of correctness appeared a total of twelve years later in [Dij86].

The algorithm and the choice of methods to derive the worst-case behavior have to be a good match. Some proof techniques may be less effective on certain algorithms and can lead to results that leave much room for improvement. Thus, deciding which proof method(s) are most suitable for a given algorithm is a crucial factor for the quality of the result.

This chapter presents several approaches to determine the worst-case complexity of a self-stabilizing algorithm. The first section illustrates the necessary and optional parts of the analysis. Section 3.2 describes the proof techniques.

3.1 Elements of the Analysis

The analysis of a self-stabilizing algorithm consists of several parts that can be treated separately. The two properties *closure* and *convergence* are mandatory to give evidence that an algorithm is indeed self-stabilizing (cf. Section 2.2.2). Most authors incorporate the complexity analysis into the proof of convergence. Apart from these elements it is

helpful to provide a worst-case example. In the following sections these components are described in detail.

3.1.1 Closure

To make the closure property hold true it has to be guaranteed that if the system is in a legitimate configuration, it is impossible to leave the set of legitimate configurations by executing the algorithm. For silent self-stabilizing algorithms (cf. Section 2.2.2) the nodes do not make another move when a legitimate configuration is reached. Hence, the closure property holds trivially.

Since a reactive algorithm does not stop after reaching a certain configuration, it is much more difficult to show that a reactive self-stabilizing algorithm fulfills the closure property. A common approach is to first assume an arbitrary legitimate configuration and then show that any next step leads to another legitimate configuration. Often this can be shown by considering only the local states of a single node and its neighbors.

The self-stabilizing algorithms considered in this thesis are all silent. Thus, the closure property does not have to be proven explicitly for them.

3.1.2 Convergence

The convergence property demands a self-stabilizing algorithm to reach a legitimate configuration after a finite number of moves irrespective of the configuration the algorithm starts with. Assuming a silent self-stabilizing algorithm, the convergence property can be verified by proving the following two properties:

- Termination: The algorithm stops after a finite number of moves.
- Correctness: Every final configuration is legitimate.

The proof of correctness verifies that the analyzed algorithm indeed calculates what it was developed for, i.e. it does not stop at an illegitimate configuration. Due to the design of self-stabilizing algorithms the correctness property is usually not very difficult to show: The predicate $\mathcal{P}$ that evaluates the correctness of a global configuration is based on the local states of the nodes. Hence, if the system is not in a legitimate configuration, there must be at least one enabled node.

The second property to be proven is termination, i.e. the algorithm cannot make an infinite number of moves. Most publications not only provide evidence that the proposed algorithms eventually reach a final configuration but also prove an upper bound for the maximum runtime. Section 3.2 describes the different methods to determine the worst-case number of moves in detail.

The worst-case move complexity for an algorithm depends on the used scheduler (cf. Section 2.3.3). Most self-stabilizing algorithms have different complexity results for each scheduler. The choice of a scheduler has great influence on the analysis, too: Assuming the central scheduler is used, the analyst does not have to consider effects caused by adjacent nodes executing at the same time. Nevertheless, it is often easier to determine the move complexity for an algorithm when a synchronous scheduler is assumed: There are no enabled nodes that do not execute a rule for a considerable number of steps as it can be the case under the central or distributed scheduler. In many cases the synchronous scheduler makes an algorithm stabilize quickly. On the other hand, certain algorithms do not stabilize at all under this scheduler due to the lack of symmetry-breaking mechanisms [IJ90, GT00, Ang80]. Since the distributed scheduler is the most general one which subsumes all other schedulers, all complexity results for this scheduler trivially also hold for the others.

3.1.3 Worst-Case Example

To verify that the derived complexity is a sharp limit and not just any arbitrary upper bound, the next step is to give an example that demonstrates the analyzed algorithm indeed requires the derived number of moves. This step is often skipped, especially for algorithms with a low complexity. Unfortunately, it is also often omitted even in cases where the result is not obvious, e.g. [SX07]. If the lower bound provided by the example and the calculated upper bound are far apart, further research can be pursued to reduce this gap by either finding an example that requires more moves until stabilization or by improving the complexity analysis.

3.2 Proof Methods for the Complexity Analysis

There is no such a thing as *the* perfect proof method that is applicable to all algorithms to verify their self-stabilization property or to determine their worst-case complexity.

This section introduces the techniques most commonly used in the literature. In Section 3.2.4, a new technique is presented that allows to make use of complete induction on the number of edges of a graph.

Often, the analysis of a self-stabilizing algorithm cannot be done by using only one proof method. Instead a combination of several techniques has to be applied. As the result of the analysis heavily depends on the methods used, it is a crucial step to decide which approach is probably best suited for a given algorithm.

3.2.1 Global State Analysis

The configuration of the whole distributed system is not visible to single nodes. However, the "view from above" can be used in the complexity analysis to prove termination. For instance, it may be possible to prove that no configuration can occur twice. Especially in algorithms that make use of a hierarchical structure it may be possible to show that local decisions made by one node cannot be undone unless a higher-ranked neighbor makes a decision itself that contradicts the state of this node. This yields the result due to the finite number of nodes.

The system's configuration is also the basis for the following analysis: For some algorithms it is possible to prove that, provided some nodes or parts of the system have already reached their final state, there is a limit on the number of moves for at least one further node. The rest may follow by induction if it is possible to find a node that stops after a few moves, e.g. the highest-ranked node.

A disadvantage of the analysis of the system's configuration to limit the number of moves until stabilization usually is the quality of the result. Having the guarantee that any configuration can occur at most once can be enough to provide evidence that the algorithm terminates at all if the set of configurations is finite. But since for most algorithms a node can take on several local states, the total number of configurations is at least exponential. Hence, this technique may not be the method of choice when the goal is to prove the efficiency of an algorithm.

In [SX07], Srimani and Xu present an algorithm for the calculation of a weakly connected minimal dominating set (cf. Section 4.2). They prove that no configuration can occur twice in order to show that their algorithm eventually stops. It remained an open problem whether this is a sharp bound or not. In this thesis it is shown that

there are examples for which their algorithm indeed needs $O(2^n)$ moves to establish a legitimate configuration under a central scheduler.

An example for the second approach mentioned in this section is the minimum weight matching algorithm by Manne and Mjelde [MM07]. They proved that if there is at least one inactive node, then there always exists a node that makes at most two more moves. Via induction this results in an upper bound of $O(3^n)$ moves. Section 3.2.4 outlines a new proof method that can be applied to this algorithm and which shows that the algorithm indeed stabilizes after at most $O(mn)$ moves. A more detailed description of this technique and the complete analysis of Manne and Mjelde's algorithm can be found in Chapter 7.

3.2.2 Analysis of Local States, Properties and Sequences

The analysis of a self-stabilizing algorithm does not necessarily have to consider the states of all nodes. A very common approach is the analysis of the state of a single node and its neighbors. Some algorithms have the property that nodes become disabled permanently after executing a certain move. An intuitive example is a simple node-coloring algorithm under a central scheduler, such as Algorithm 1 of [GK93]: If a node has set its color to a value that does not occur in its neighborhood, it is guaranteed that none of its neighbors can choose the same color. Hence, the node will not become enabled again.

Another version of this property is a limitation on the number of moves a node (or its neighbors) can make after it has executed a particular move. For instance, in the *maximal independent set* algorithm by Turau the node with locally highest identifier can enter the set if none of its neighbors is already included. After that all its neighbors will make at most one further move [Tur07]. This paper also shows how the analysis of sequences of moves can be used to determine the move complexity of a self-stabilizing algorithm: The dependencies between the rules the algorithm consists of and the states of a node's neighbors lead to a specific order of the moves a node can make. Since all these sequences either contain a move that disables a node itself permanently or a move that requires a neighbor to have made a move that limits all its neighbor's moves, an upper bound on the total number of moves can be derived.

There are proof methods that combine the analysis of local and global properties. Showing that a certain move e.g. disables the executing node permanently can be

combined with deriving an upper limit on the number of moves that can be made in total *without* an execution of this move. As a simple example, consider an algorithm consisting of two rules to color a node black and gray respectively (c.f. Algorithm 4.2). Obviously, after n moves of one type all nodes have the same color and the corresponding rule is disabled.

3.2.3 Potential Functions and Convergence Stairs

Potential functions have been used for proving self-stabilization e.g. in [Dol00] or [Kes88]. A potential function (or *variant function*) measures the "progress" an algorithm makes during its execution. This can be done e.g. by counting nodes with a certain property. It may be possible to measure how close the current configuration is to the final configuration in a system that converges to a unique legitimate configuration.

Potential functions are usually monotonic functions with a given limit for the value. It has to be shown that the value converges to this limit during execution, i.e. the execution of at least one rule increases this function while the other rules at least do not decrease it (or vice versa). The monotonicity is not a mandatory property but it simplifies the proofs significantly since the convergence property trivially holds true if it can be shown that the value increases (resp. decreases) regularly.

There are very simple examples for potential functions, for instance the growing number of permanently disabled nodes. However, for some algorithms only very complex potential functions were found. Such an example is given in Chapter 7. The major challenge of this proof method is the search for a suitable function. In [The00] Theel writes:

> The difficulty of this verification strategy lies in the fact that finding such a variant function for a given system requires experience and inspiration since the function must in itself bear the "essence of convergence" of the system. Thus, deriving a variant function for arbitrary systems is regarded as an art rather than a craft.

In [TG99] and [The00] a relation between self-stabilization and control theory in the engineering domain is shown. According to them, Ljapunov's "Second Method" [Lja07] can be used to more easily identify variant-like functions.

A proof method similar to potential functions uses *convergence stairs* [GM91]. A convergence stair consists of several global predicates $R_1, \ldots, R_k$ with the following conditions:

- *Boundary:* $R_1 = \text{true}$ and R_k corresponds to the definition of a legitimate configuration of the system.
- *Closure:* Each R_i is closed, i.e. if $R_i = \text{true}$ for a given configuration, all subsequent configurations also satisfy R_i.
- *Convergence:* Each R_i converges to R_{i+1} for $i < k$.

A monotonic potential function ϕ for a self-stabilizing algorithm is a special case of a convergence stair since it is possible to define a predicate for each value of ϕ.

3.2.4 Graph Reduction and Induction

In Chapter 7, a new approach to determine the worst-case complexity of self-stabilizing algorithms is presented. Its basic idea is to create a mapping from the algorithm's execution sequence of a graph to that of a reduced graph. This allows to use complete induction in the proofs.

In particular, the original graph is reduced by removing the lightest edge. This choice is due to the analyzed algorithm that makes use of edge weights. Then, two mappings are defined: Configurations that are not valid on the reduced graph have to be altered, and the same holds for moves that cannot be performed by the adjacent nodes. These moves are either changed to suit the new topology or just omitted if there is no reasonable substitute. It is shown that via these two mappings any execution sequence of the original graph can be transformed to a valid execution sequence of the reduced graph. Now the number of omitted moves due to the deletion of the edge has to be determined. In this case it is possible to retrieve an upper bound for these moves. The product of this number and the number of edges of the original graph yield the result by induction. For more details see Chapter 7.

3.2.5 Invariancy-Ranking

Recently, Köhler and Turau developed a generic technique for proving stabilization under the distributed scheduler, provided the algorithm in focus stabilizes under the central scheduler [KT10] .

The basic idea of their work is to define a *rank* for each enabled node in a given configuration. By sorting the moves of a step based on their rank it is aimed to obtain a *serialization*, i.e. a sequence of moves for the central scheduler that produces the same global configuration as executing these moves in a single step of the distributed scheduler. If the rank of the nodes does not change during the execution of the obtained sequence, the ranking function is called an *invariancy-ranking*. It is shown that for an algorithm with an invariancy-ranking, every set of enabled instances is serializable in any configuration. Furthermore they prove that for any execution e under the distributed scheduler of an algorithm with an invariancy-ranking there is an execution e' under the central scheduler such that e is a subsequence of e'. This yields the main result of [KT10]: If it is possible to find an invariancy-ranking for a given algorithm, the complexity under the central scheduler is also an upper bound for the complexity under the distributed scheduler. There is no slowdown as no scheduler transformation has to be applied (cf. Section 2.3.3).

Chapter 4

Distance-Two Knowledge and Network Decomposition

Many self-stabilizing algorithms aim to achieve a configuration that depends on a fixed hierarchic structure with respect to e.g. edge weights, node identifiers or distance to a root node. Usually, in such a case, a node only takes the states of its higher-ranked neighbors as a basis for its move. The state of a lower-ranked node is ignored, these nodes have to adapt their state to the state of their higher-ranked neighbors. Since a higher-ranked node does not adjust its state according to *all* its neighbors, any of its moves can force all its lower-ranked neighbors to change their states. In an adverse setting, this may not only cause a significant number of nodes to get affected by a single transient error (e.g. at the highest-ranked node) starting from a legitimate configuration. Also, the complexity of an algorithm can be very high if a node can activate other parts of the network again and again by a single move. An example for such a case that leads to an exponential number of moves will be presented in Section 4.2.1.

The idea of network decomposition is to "gain more locality" by limiting the area a node can control just by its rank. This chapter uses this technique to develop a distributed self-stabilizing algorithm for the weakly connected minimal dominating set problem. It assumes a self-stabilizing algorithm to compute a breadth-first tree. Using an unfair distributed scheduler the algorithm stabilizes in at most $O(nmA)$ moves,

where A is the number of moves to construct a breadth-first tree, i.e. $O(A) \leq O(n^3)$ [SX07]. All previously known algorithms required an exponential number of moves.

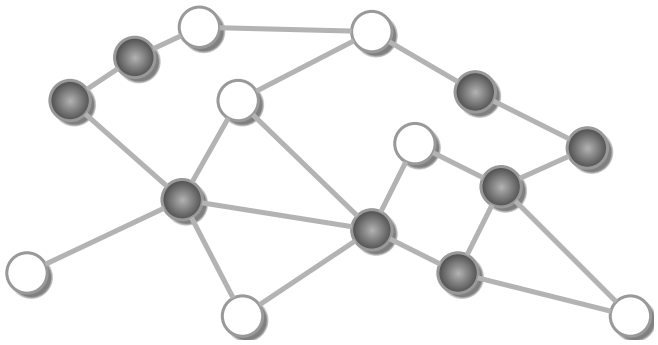
This chapter is structured as follows: Section 4.1 introduces the weakly connected minimal dominating set problem and overviews related work. In Section 4.2 the first self-stabilizing algorithm for this problem, which was developed by Srimani and Xu [SX07], is presented. Their complexity analysis is completed by proving that the algorithm needs an exponential number of moves in the worst case. Section 4.3 explains how the disadvantages of the aforementioned algorithm can be resolved by dividing the graph into several subnetworks. In Section 4.4 the new algorithm is presented assuming a central scheduler. The design is made with a distance-two setting which is later transformed to the common model that only allows distance-one knowledge. The complexity analysis makes use of a potential function. Section 4.5 demonstrates that it is possible to transform the algorithm to run under a distributed scheduler with only constant slowdown. The chapter ends with a short conclusion.

4.1 Example: Weakly Connected Minimal Dominating Set

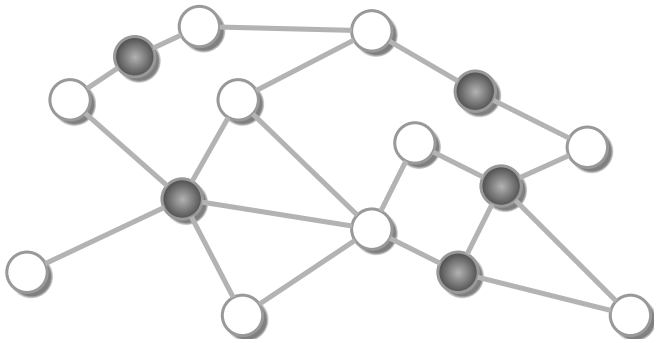
4.1.1 Introduction

A classical problem of graph theory is the calculation of dominating sets. Dominating sets are an important structure for many applications, e.g. as a backbone network for efficient communication in a distributed system [WL99]. Depending on the application it is possible to define many distinct domination parameters [HL91]. For the sake of coherence, the definitions of Section 2.4.2 are repeated here:

Let $G = (V, E)$ be an undirected graph. A dominating set S of G is a subset of V such that each $v \in V \backslash S$ has at least one neighbor in S. S is a *minimal* dominating set if for any node $v \in S$ the set $S \backslash \{v\}$ is not dominating. If S is connected, it is called *connected dominating set*. A dominating set S is called *weakly connected* if the subgraph weakly induced by S, i.e. the graph $(N[S], E \cap (S \times N[S]))$ is connected (Figure 4.1).



(a) Connected dominating set



(b) Weakly connected dominating set

Figure 4.1: A weakly connected dominating set requires significantly less nodes than a connected dominating set

4.1.2 Related Work

The construction of minimal dominating sets in distributed systems has attracted a lot of research due to the importance of the concept in the field of wireless communication [AWF03]. Many applications use dominating sets as a structure to route messages or for clustering. Often, the dominating sets are required to be connected. However, connected dominating sets can be rather large. By relaxing the connectivity requirement the number of nodes can be reduced significantly ([CL02], see Figure 4.1). This led to the notion of a weakly connected minimal dominating set [DGH+97, PH06]. Several distributed algorithms for approximating such sets have been proposed [AWF03, DMP+03].

Self-stabilizing algorithms for dominating sets exist for several dominating parameters. A survey paper by Guellati and Kheddouci discusses several self-stabilizing algorithms for the construction of minimal dominating sets and minimal k-dominating

sets [GK10]. Section 2.4.2 of this thesis provides a summary of their results and further lists algorithms for connected and weakly connected dominating sets.

Recently, Srimani and Xu presented the first self-stabilizing algorithm to construct a weakly connected dominating set (WCMDS) using the central scheduler [SX07]. A revised version of this paper [XHGS03] contains more details but the results are the same: The algorithm is based on a breadth-first tree and stabilizes after $O(2^n)$ moves. This algorithm will be the subject of Section 4.2.

Kamei and Kakugawa published a self-stabilizing algorithm for the calculation of a WCMDS under the synchronous scheduler [KK07a]. On a unit disk graph this algorithm guarantees an approximation ratio of 5 with respect to the solution with minimum cardinality. Furthermore, a dominating set is established after the first round. The total stabilization time of the algorithm is $O(n^2)$ rounds.

A new algorithm for the construction of a weakly connected minimal dominating set under the distributed scheduler will be presented in Section 4.4. It stabilizes after $O(mnA)$ moves, where A is the number of moves necessary for the construction of a breadth-first spanning tree.

4.2 Algorithm of Srimani and Xu

The first self-stabilizing algorithm to compute a weakly connected minimal dominating set was presented by Srimani and Xu [SX07]. Their algorithm requires a breadth-first spanning tree of the given graph. A self-stabilizing algorithm that establishes such a tree is also presented in [SX07] (within at most $O(n^3)$ moves). It assumes unique node identifiers and the node with maximum id is chosen to be the root of the spanning tree. The spanning tree algorithm initializes the variable $v.p$ that stores the *parent node* of a node v, and $v.l$ which holds the *level*, i.e. the distance in hops to the root node. The root node r has $r.p = r$ and $r.l = 0$. The boolean flag $v.f$ denotes, whether node v is a member of the weakly connected minimal dominating set or not. The set of rules is shown in Algorithm 4.1.

Via rule R_1 the root node enters the WCMDS, if it is not included already. Rule R_2 makes a node leave the set if an adjacent node with lower level is included. Otherwise it enters the set itself. A more detailed description and the proofs of correctness and termination can be found in [SX07].

Algorithm 4.1 WCMDS Algorithm of Srimani and Xu

Functions:
hasLowerLevelIncludedNeighbor(v) :
return $(\exists u \in N(v)$ *s.t.* $u.l \leq v.l \wedge u.f = true)$

Actions:
(root node)
$\mathsf{R}_1 :: [(v.p = v) \wedge (v.f = false)]$
$\longrightarrow v.f := true$

(non-root nodes)
$\mathsf{R}_2 :: [(v.p \neq v) \wedge (hasLowerLevelIncludedNeighbor(v) = v.f)]$
$\longrightarrow v.f := \neg hasLowerLevelIncludedNeighbor(v)$

4.2.1 Complexity Analysis

Srimani and Xu prove that Algorithm 4.1 stabilizes by performing an analysis of the global configuration. In particular they show that no configuration can occur twice. This leads to an upper bound of at most $O(2^n)$ moves. Since they did not provide a lower bound, i.e. a worst-case study that indeed uses $O(2^n)$ moves, it remained an open question whether this limit is sharp or not. The following example provides a lower bound for Algorithm 4.1.

Consider a circle C which consists of eight nodes. It contains a nodes v_s and a node v_t with distance 4 (Figure 4.2). Assume node v_s to have the lowest level in C and hence, v_t has the highest level of all nodes in C. Obviously, if node v_s sets its flag then all nodes of C with even distance will finally have assigned the same state and the other nodes take on the opposite value.

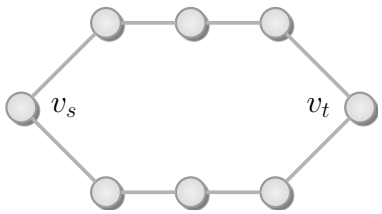


Figure 4.2: Circle C with node v_s having the lowest level

In the following it will be shown that a certain initial configuration and an adverse sequence of moves allows node v_t to make four moves when v_s only moves twice.

This is made possible because the nodes of the two paths that connect v_s and v_t not necessarily execute their rules synchronously. Hence it is possible to "store a decision" from a higher-ranked node for a finite time and pass it on with delay. Figure 4.3 shows an execution of Algorithm 4.1 on C that demonstrates this behavior (this example does not consider why node v_s changes its state at all, configurations that allow for such an assumption will be presented later in this section). Node v_s changes its state twice and v_t changes its state four times. Nodes v with $v.f = true$ are colored black, the others are white.

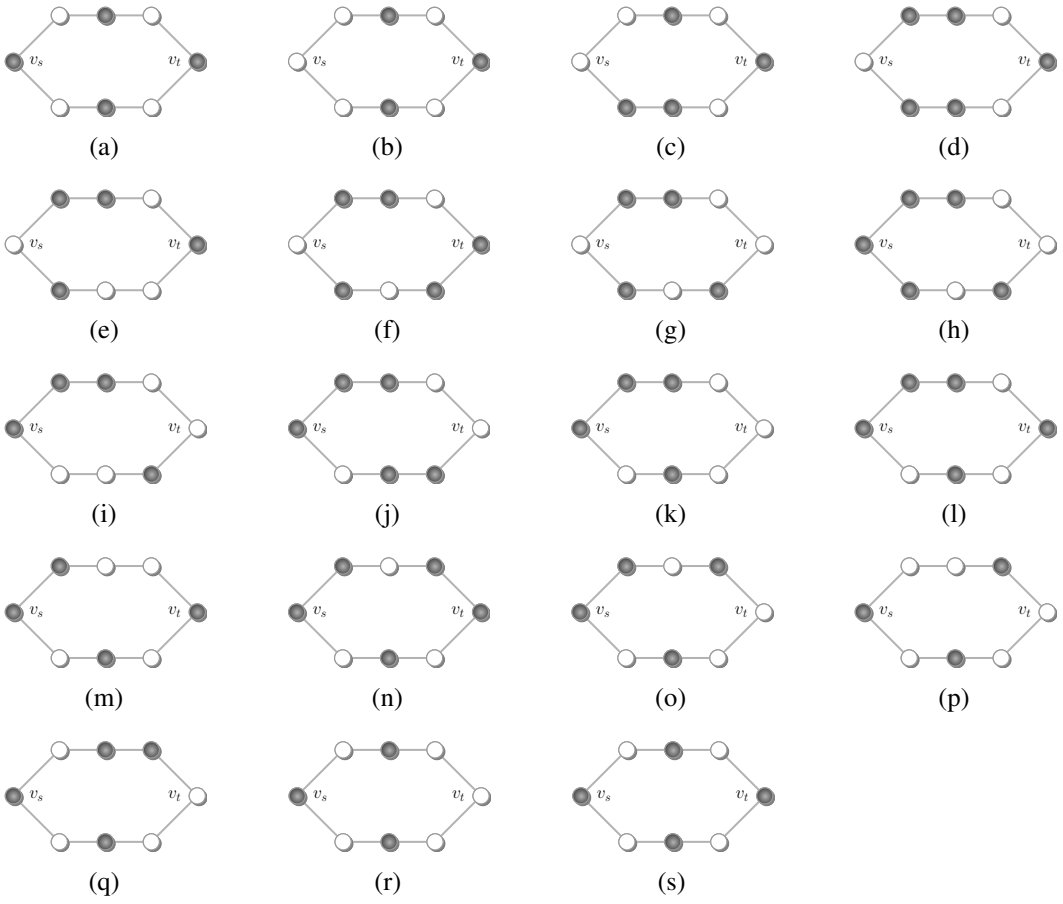


Figure 4.3: Adverse execution of Algorithm 4.1 on a circle when node v_s moves twice: Node v_t makes four moves!

Let G_k be a graph that is composed of two nodes, one of them regarded as root, and k circles $C_1, \ldots C_k$ of the same type as described above and arranged in a row, such

that every circle shares a particular node with its neighboring circle, i.e. for all circles C_i and C_{i+1}: $v_{t,i} = v_{s,i+1}$. Figure 4.4 shows G_3.

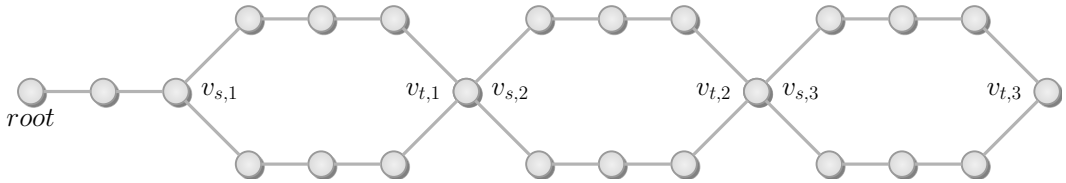


Figure 4.4: Graph G_3

According to the considerations of a single circle it is now possible to construct a graph and an initial configuration that leads to an exponential number of moves until stabilization: Consider the initial configuration given in Figure 4.5:

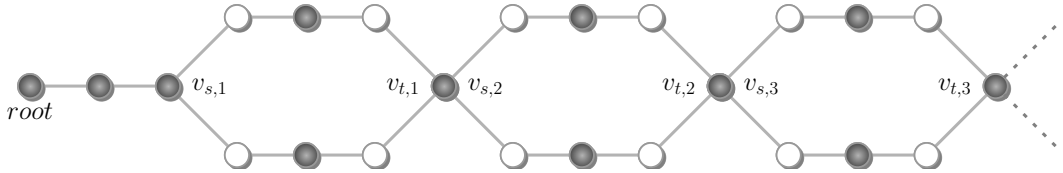


Figure 4.5: Adverse initial configuration of the WCMDS algorithm on Graph G_k

As shown above, $v_{t,i}$ can make twice as much moves as $v_{s,i}$, if $v_{s,i}$ gets enabled by a node with higher level. In an adversarial execution the nodes of a circle C_i do not perform a move if a node of a circle C_j with $i < j$ is enabled. Furthermore, the node between $v_{s,1}$ and *root* only moves if no other node is enabled. Consequently, node $v_{s,1}$ will move twice. Thus, there is an execution of Algorithm 4.1 of G_k (consisting of $7k+2$ nodes) in which node $v_{t,k}$ can make 2^k moves. Hence, $O(2^n)$ is also a lower bound for Algorithm 4.1.

Remark 1. The behavior of the system in case of a single transient error after having reached stability heavily depends on the rank of the node in the hierarchy, i.e. the distance of the erroneous node to the root node. If a node with no lower-ranked neighbors in the breadth-first tree fails, no other node gets enabled to execute a rule. The node will reset its status and the system regains legitimacy after a single move. However, if the memory of the root node gets corrupted, the same situation as described in the worst-case example can occur, i.e. all nodes can make a move and the system stabilizes after $O(2^n)$ moves.

4.3 Network Decomposition

The previous section showed that some algorithms allow the decisions of a single node to decide the final state of any other node. The local storage of an impulse given by a higher-ranked node can lead to an exponential number of moves.

There are several ways to prevent this behavior. One possibility is to build up a certain level of synchrony by using synchronizers [Awe85, AKM+07]. However, these approaches usually come along with a slowdown of the stabilization time. Furthermore they require a-priori knowledge, e.g. the number of nodes of the system. In this section a new, polynomial algorithm for the weakly connected minimal dominating set problem is presented that uses a network decomposition [AGLP89], or *layers*, of a graph to limit the influence of a single node's move. The new algorithm has two things in common with the algorithm of Srimani and Xu:

1. It is assumed that nodes have globally unique identifiers and that a distinguished node, called *root*, has been selected, e.g. the node with the smallest identifier.

2. The proposed algorithm is based on a self-stabilizing algorithm $\mathcal{A}$ to construct a breadth-first tree for the given root.

The only requirement for $\mathcal{A}$ is that each node maintains a variable d measuring the distance in hops to the root of the tree. Algorithms of this type are described in [HC92, SX07, SS92]. The breadth-first tree algorithm in [SX07] stabilizes after at most $O(n^3)$ moves. The other references lack a detailed complexity analysis.

Instead of a boolean flag to indicate whether a node is considered to be part of the weakly connected dominating set or not, the new algorithm will use a variable *status*. This variable may assume any of the three values *black*, *gray* and *white*. The code of the algorithm to build a breadth-first tree is modified as follows: Every time the variable d is changed the following statement is executed:

if $(d \ mod \ 2 = 1)$ **then**
    $status := white$
else if $(status = white)$ **then**
    $status := black$

Additionally, a similar rule is introduced to color a node, in case the color and the value of d are inconsistent. These changes do not increase the worst-case run time

of the algorithm. When $\mathcal{A}$ stabilizes, all nodes with an odd distance to the root have been assigned the value *white* (see Figure 4.6). All other nodes are either *black* or *gray*. The root always gets assigned the value *black*. For every node v there is a path to the root where every second node is white and the other nodes are either black or gray. Note that the variable *status* does not appear in any of the guards of $\mathcal{A}$. Let $N_b(v), N_w(v), N_g(v)$ denote the sets of black, white and gray neighbors of v.

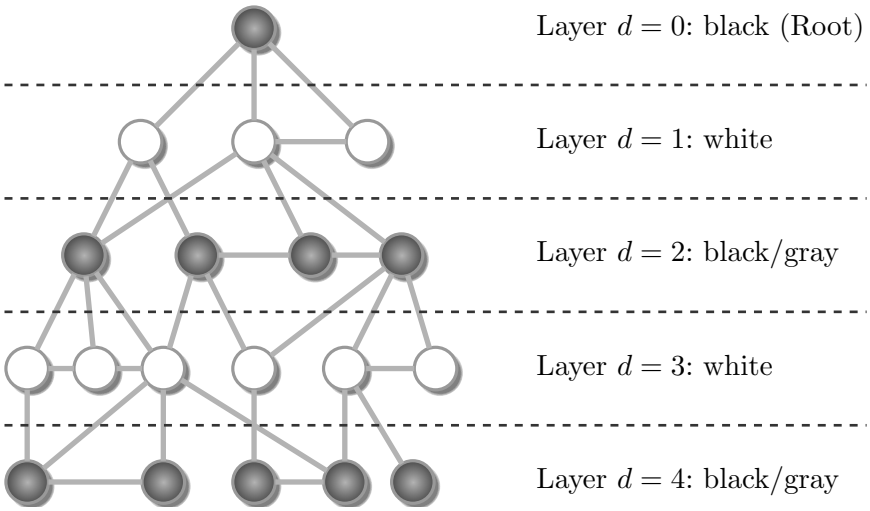


Figure 4.6: Network decomposition for the WCMDS Algorithm. The white nodes will never change their *status* variable. Therefore, the move of a black/gray node has only local effects.

The algorithm to compute a weakly connected minimal dominating set will be the composition of $\mathcal{A}$ and an algorithm to be presented in the following sections. This algorithm uses the variable *status* in its guards and also changes the value of this variable. It will be shown that it stabilizes for any assignment of the variable *status* (cf. Section 2.3.1). When both algorithms have stabilized, the black nodes will form a weakly connected minimal dominating set. Moreover, the worst case number of moves of the composed algorithm is the product of the worst case number of moves of both algorithms (cf. Section 2.3.1).

4.4 Central Scheduler

In order to devise an algorithm to compute a weakly connected minimal dominating set it is first assumed that each node has instant access to the states of all nodes

within distance two (*distance-two model*, cf. Section 2.3.2). Later this assumption will be removed. Algorithm 4.2 is formulated using this premise. Intuitively, a black node changes its color to gray if it has a black neighbor and if all its white and gray neighbors have another black neighbor. A gray node changes its color to black if it has no black neighbor or if it has a gray or white neighbor with no black neighbor. Note that the state of white nodes never changes (see Figure 4.6).

Algorithm 4.2 MDS with Network Decomposition, Distance-Two Algorithm

Actions:

$\mathsf{R}_1 :: \left[v.status = black \wedge N_b(v) \neq \emptyset \wedge \nexists\, w \in N_g(v) \cup N_w(v) \,:\, |N_b(w)| = 1\right]$
$\longrightarrow v.status = gray$

$\mathsf{R}_2 :: \left[v.status = gray \wedge (N_b(v) = \emptyset \vee \exists\, w \in N_g(v) \cup N_w(v) \,:\, |N_b(w)| = 0)\right]$
$\longrightarrow v.status = black$

Lemma 4.4.1. *When no node is enabled for Algorithm 4.2 in the distance-two model then the black nodes form a minimal dominating set for the graph induced by all black and gray nodes and white nodes with a black or gray neighbor.*

Proof. Since no node is enabled for rule R_2, every gray node must have a black neighbor. Suppose there exits a white node v with $N_b(v) = \emptyset$. Then there exists a gray node $w \in N(v)$. But then w would be enabled with respect to rule R_2. Thus, the black nodes are dominating.

Suppose there exists a black node v such that the remaining black nodes are dominating. Then $N_b(v) \neq \emptyset$ and $|N_b(w)| > 1$ for every white or gray node $w \in N(v)$. Thus, node v is enabled for rule R_1. This contradiction completes the proof. □

Lemma 4.4.2. *Algorithm 4.2 stabilizes after at most* $3n$ *moves in the distance-two model.*

Proof. For a configuration c let $\phi(c)$ denote the number of nodes that are either black or have a black neighbor in c. Obviously, $0 \leq \phi(c) \leq n$. Rule R_2 increases ϕ by at least one and rule R_1 does not decrease ϕ. Thus, there are at most n executions of rule $R2$. Furthermore, a node cannot execute rule R_1 twice without executing rule R_2 in-between. This concludes the proof. □

Goddard et al. developed a general mechanism to transform self-stabilizing algorithms having distance-k knowledge into algorithms that operate in the normal distance-one model [GHJT08] (cf. Section 2.3.2). This transformation will be used in the following for the case $k = 2$. For this purpose each node will be reconfigured to have two additional variables. The variable bn holds the number of black neighbors of a node, i.e. $v.bn = |N_b(v)|$. The variable p can hold the identifier of a neighbor or the special value *null*. This variable is used to ensure that a node v can execute an action of the original algorithm only if the values of the variable bn of all neighbors of v are up-to-date.

To formally define the transformed algorithm some definitions are needed. For each node v let

$$minN[v] = \min\{w \in N[v] \mid w.p = w\}.$$

If $w.p \neq w$ for all nodes $w \in N[v]$ then $minN[v] = null$. A node v is called *correct*, if $v.bn$ has the correct value. A correct node v is called *prepared* if $w.p = v$ for all $w \in N[v]$ and it is called *neutral* if $w.p = null$ for all $w \in N[v]$. Finally, v is called *willing* if it is enabled with respect to Algorithm 4.2. This property is based on the values of the variable bn of all neighbors. Note that these values do not necessarily reflect the current state of the nodes. The complete set of rules is as follows:

Algorithm 4.3 WCMDS with Network Decomposition, Central Scheduler

Actions:

UPDATE :: [v *is not correct*]
$\longrightarrow$ *update* $v.bn$

ASK :: [v *is willing and neutral*]
$\longrightarrow$ $v.p := v$

RESET :: [v *is correct and* $v.p \neq minN[v]$]
$\longrightarrow$ $v.p := minN[v]$

CHANGE :: [v *is prepared*]
$\longrightarrow$ *if* (v *is willing*)
execute Algorithm 4.2
$v.p := null$

Theorem 1 of [GHJT08] with $k = 2$ yields that Algorithm 4.3 stabilizes under

a central scheduler after at most $O(n^2m)$ moves. In the following it is shown that $O(nm)$ moves suffice.

A move of a node v is called *correct* if it is based on correct values $w.bn$ of all neighbors $w \in N(v)$ and incorrect otherwise. In the following, an execution of rule CHANGE will be referred to as a $\mathsf{REAL}-\mathsf{CHANGE}$ move if the node really executes Algorithm 4.2.

Lemma 4.4.3. *If a node v makes an* ASK *move and then later a* CHANGE *move with no intervening* RESET *move by v, then the* CHANGE *move is correct.*

Proof. At the time of the ASK move, all neighbors w of v point to $null$. Later, at the time of the CHANGE move, each w points to v. At the moment when a neighbor w changes its pointer back to v with a RESET move, its bn value is correct. From that point on, since $w.p = v$, no other neighbor of w can make a CHANGE move, and so $w.bn$ remains correct. □

Lemma 4.4.4. *There are at most $O(n)$ correct and incorrect* $\mathsf{REAL}-\mathsf{CHANGE}$ *moves.*

Proof. By the preceding lemma an incorrect $\mathsf{REAL}-\mathsf{CHANGE}$ can only occur as the first CHANGE move of a node because subsequent CHANGE moves must be preceded by an ASK move. An incorrect $\mathsf{REAL}-\mathsf{CHANGE}$ move using rule R_2 of Algorithm 4.2 will not decrease the value of the function $\phi(c)$ as introduced in Lemma 4.4.2. An incorrect $\mathsf{REAL}-\mathsf{CHANGE}$ move of a node v using rule R_1 of Algorithm 4.2 can decrease the value of $\phi(c)$ by at most $d_v - 1$. If a node makes an incorrect $\mathsf{REAL}-\mathsf{CHANGE}$ move, then all $\mathsf{REAL}-\mathsf{CHANGE}$ moves by any of its neighbors are correct. Hence, all incorrect $\mathsf{REAL}-\mathsf{CHANGE}$ moves together can decrease the value of $\phi(c)$ by at most n. Thus, repeating the arguments contained in the proof of Lemma 4.4.2 gives rise to the upper limit of $O(n)$. □

Lemma 4.4.5. *Algorithm 4.3 stabilizes under a central scheduler after at most $O(nm)$ moves. After stabilization the black nodes form a minimal dominating set for the graph induced by all black and gray nodes and those white nodes that have a black or gray neighbor.*

Proof. During an interval without any $\mathsf{REAL}-\mathsf{CHANGE}$ moves all nodes together make at most $O(m)$ moves. This follows from Lemma 8 in [GHJT08] applied for the case $k = 2$. Applying Lemma 4.4.4 yields the desired bound.

Consider a configuration in which no node is enabled with respect to Algorithm 4.3. Then obviously all nodes are correct. Suppose that there exits a node that points to itself and let v be the node with the smallest identifier having this property. If there would be a neighbor w of v with $w.p \neq v$, then v would be enabled with respect to rule RESET. This would imply that v is prepared and could execute rule CHANGE. This yields that no node points to itself. Therefore, all nodes are neutral. Because no node is enabled to execute rule ASK, no node can be enabled with respect to Algorithm 4.2. Hence, the second statement follows from Lemma 4.4.1. □

Let $\mathcal{A}$ be a self-stabilizing algorithm to compute a breadth-first tree that is enhanced as described above and that stabilizes after $O(\mathcal{A})$ moves under a central scheduler. In order for Algorithm 4.3 to produce a *weakly connected* minimal dominating set of the graph a small modification is required. In Algorithm 4.2 the set $N_w(v)$ needs to be restricted to white nodes w that satisfy $w.d = v.d + 1$. With this modification the corresponding statement of Lemma 4.4.5 still holds. Denote by $\mathcal{D}$ the composition of $\mathcal{A}$ and Algorithm 4.3. Intuitively $\mathcal{A}$ separates the nodes of G into layers. The nodes in these layers are beginning with the root alternately colored black and white. Due to the modification described above Algorithm 4.3 basically works independently on every set of nodes consisting of a white layer and the black layer above. This leads to the following theorem.

Theorem 4.4.1. *$\mathcal{D}$ is a self-stabilizing algorithm to compute a weakly connected minimal dominating set. Under a central scheduler $\mathcal{D}$ stabilizes after at most $O(nmA)$ moves.*

Proof. According to Lemma 4.4.5, Algorithm 4.3 makes at most $O(nm)$ moves in between two moves of algorithm $\mathcal{A}$. Hence, $\mathcal{D}$ makes at most $O(nmA)$ moves. When algorithm $\mathcal{D}$ has stabilized, every white node has a black or a gray neighbor. According to Lemma 4.4.5 the set S of black nodes form a minimal dominating set. To prove that the subgraph weakly induced by S is connected, note that each white node has a black neighbor in the layer above and each gray node has a black neighbor in the same layer. Furthermore, each black node except the root has a white neighbor in the layer above. Hence, every node is connected with the root via a path, where every second node is a black node. This implies that the black nodes form a weakly connected minimal dominating set. □

4.5 Distributed Scheduler

Gairing et al. provide a mechanism to transform self-stabilizing algorithms having distance-two knowledge for the distributed scheduler [GGH+04]. This transformation leads to a slowdown factor of $O(n^2)$ moves. Another option is to use a general transformer to convert an algorithm stabilizing under a central scheduler into one that stabilizes under a distributed scheduler. Usually this is achieved by avoiding simultaneous rule executions of neighboring nodes. Then the transformed schedule is equivalent to a serial schedule. In the best cases this leads to a slowdown factor of $O(\Delta)$ moves [GT07]. In the following, Algorithm 4.3 is modified such that it stabilizes under a distributed scheduler with only a constant slowdown. The key observation is that it is not necessary to avoid all simultaneous rule executions in order to preserve the stabilization property. Furthermore, the proof makes use of a special characteristic of Algorithm 4.2.

Algorithm 4.3 already prevents neighboring nodes to execute rule CHANGE simultaneously since neighboring nodes cannot be prepared at the same time. A problem arises when a node v executes rule RESET at the same time as the node w it is pointing to performs a CHANGE move (Figure 4.7). Let u be the node that v points to after

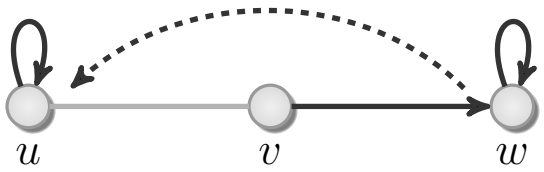


Figure 4.7: Node v executes rule RESET at the same time as node w makes a CHANGE move. Now u can execute rule CHANGE despite being in an inconsistent state!

the RESET move. Note that at this moment the value $u.bn$ is not consistent. If u would make a CHANGE move in this situation, then this move would not be legal with respect to Algorithm 4.2. To avoid such situations, the rules RESET and CHANGE are revised so that a node can detect whether the neighbor it is currently pointing to wants to perform a CHANGE: A node wishing to execute a CHANGE move has to set its *wtc* ("want to change") flag first using rule LOCK.

Another modification addresses the ASK rule. Allowing neighboring nodes to perform ASK moves simultaneously can result in $O(n^2)$ ASK moves between CHANGE

moves. To avoid this to happen another flag *wta* ("want to ask") is introduced. A node v is allowed to perform an ASK move if and only if it is the node with the smallest identifier in $N[v]$ that has its *wta* flag set to *true*. Another rule is added to reset this flag if a node cannot execute an ASK due to not being enabled with respect to Algorithm 4.2 anymore. The complete set of rules is shown in Algorithm 4.4.

Algorithm 4.4 WCMDS with Network Decomposition, Distributed Scheduler

Actions:

UPDATE :: [v *is not correct*]
$\longrightarrow$ *update* $v.bn$

ASK :: [v *is willing and neutral and* $\forall w \in N(v) : (w > v$ *or* $w.wta = false)$]
$\longrightarrow$ **if** $(v.wta = false)$
then $v.wta := true$
else $v.p := v$
$v.wta := false$

RESET :: [v *is correct and* $v.p \neq minN[v]$ *and* $v.p.wtc = false$]
$\longrightarrow v.p := minN[v]$

LOCK :: [v *is prepared and* $v.wtc = false$]
$\longrightarrow v.wtc := true$

CHANGE:: [v *is correct and* $v.wtc = true$]
$\longrightarrow$ **if** $(v$ *is prepared and willing*$)$
then *execute Algorithm 4.2*
$v.p := null$
$v.wtc := false$

UNASK :: [v *is correct and not willing and* $v.wta = true$]
$\longrightarrow v.wta := false$

In the following, an execution of rule ASK for node v will be referred to as a $\mathsf{REAL}-\mathsf{ASK}$ move if v points to itself after the move.

Lemma 4.5.1. *The number of* RESET *moves of nodes that point to themselves before execution is limited to one per node.*

Proof. A node v can only point to itself via performing $\mathsf{REAL}-\mathsf{ASK}$, which implies that all neighbors point to $null$. Afterwards no neighbor can perform a $\mathsf{REAL}-\mathsf{ASK}$

and thus, v cannot perform a RESET. Therefore this situation can only appear once for each node, namely in the initial configuration. □

Lemma 4.5.2. *During an interval without* REAL − CHANGE *moves all nodes together make at most* $O(n+m)$ *moves.*

Proof. a) Let v be any node. Obviously v makes at most one UPDATE move during such an interval. After a CHANGE move of v, $v.wtc$ (resp. $v.p$) has value *false* (resp. *null*). For v to make another CHANGE move, it must first execute a LOCK move requiring v to be prepared. Thus, v must execute a REAL − ASK move and for that to happen it must be neutral. Hence, at the time of the LOCK move all neighbors of v must have moved, and their bn values are correct at the time of the LOCK move. Thus, v is enabled with respect to Algorithm 4.2 and cannot execute a CHANGE move without this being a REAL − CHANGE.

Between every two LOCK moves of v there must be at least one CHANGE move. Thus, there are at most two LOCK moves per node. If v performs a REAL − ASK move then all its neighbors point to $null$. Obviously the next move of v cannot be an UPDATE, ASK, LOCK, or UNASK move. If its next move is a RESET move there must be a neighbor pointing to itself. Therefore this neighbor must have executed rule REAL − ASK itself, which is impossible, because v does not point to $null$. Thus, the next move of v is a CHANGE move. Since this rule can be executed only once, there cannot be more than two REAL − ASK moves.

Hence, v makes at most $O(n)$ REAL − ASK, LOCK, CHANGE, and UPDATE moves.

b) In between two successive RESET moves the set of self-pointing neighbors must change. By the result of the last paragraph and Lemma 4.5.1 each neighbor can enter and leave the set of self-pointing nodes only twice. Thus, it follows that a node v can execute at most $5d_v$ RESET moves.

A node can execute rule UNASK only if it has executed rule ASK before (except for the very first time). At this time it was enabled with respect to Algorithm 4.2. If it is enabled to perform an UNASK this must have changed, and thus, one of its neighbors must have performed an UPDATE. Every neighbor can execute at most one UPDATE move, so the number of UNASK moves of a node v is bounded by $d_v + 1$.

Obviously, a node v cannot execute rule ASK twice without having executed rule UNASK or REAL − ASK in between. Thus, v can make at most $d_v + 4$ ASK moves.

Hence, there are at most $O(m)$ moves of the types ASK, UNASK and RESET. □

Lemma 4.5.3. *Algorithm 4.4 stabilizes after at most $O(nm)$ moves under the distributed scheduler. After stabilization the black nodes form a minimal dominating set for the graph induced by all black and gray nodes and those white nodes that have a black or gray neighbor.*

Proof. Lemma 4.4.3 holds for Algorithm 4.4 under a distributed scheduler, since neighboring nodes cannot simultaneously execute a CHANGE and a RESET move. This yields that Lemma 4.4.4 is also valid for Algorithm 4.4 using the distributed scheduler. Note that even under this scheduler neighboring nodes cannot execute a CHANGE move concurrently. Thus, there will be no more than $O(n)$ moves of type $\mathsf{REAL} - \mathsf{CHANGE}$. By Lemma 4.5.2 the algorithm stabilizes after at most $O(nm)$ moves.

When no node is enabled for Algorithm 4.4, then all nodes are correct and the variable *wtc* has value *false* for all nodes. Furthermore, all non-willing nodes have set *wta* to *false*. Suppose there is a node that points to itself. Then the node with the smallest identifier pointing to itself would be prepared and thus rule LOCK would be enabled. Hence, all nodes are neutral. Suppose there is a node that is willing. Among all such nodes, let v be the node with the smallest identifier. Then rule ASK would be enabled for v. This contradiction proves, that no node is enabled with respect to Algorithm 4.2. Hence, the second statement follows from Lemma 4.4.1. □

Let $\mathcal{A}$ be a self-stabilizing algorithm to compute a breadth-first tree that is enhanced as described above and that stabilizes after $O(\mathcal{A})$ moves under an unfair distributed scheduler. Denote by $\mathcal{D}$ the composition of $\mathcal{A}$ and Algorithm 4.4. This leads to the main result of this chapter.

Theorem 4.5.1. *$\mathcal{D}$ is a self-stabilizing algorithm to compute a weakly connected minimal dominating set. Under an unfair distributed scheduler $\mathcal{D}$ stabilizes after at most $O(nm\mathcal{A})$ moves.*

Proof. The proof is similar to that of Theorem 4.4.1, using Lemma 4.5.3 instead of Lemma 4.4.5. □

4.6 Conclusion

This chapter completed the analysis of the worst case complexity for an algorithm by Srimani and Xu that computes a weakly connected minimal dominating set [SX07]. By giving an example it verified that $O(2^n)$ is not only an upper bound but also a lower bound for the number of moves until stabilization. Furthermore, this chapter presented another self-stabilizing algorithm for this problem. Via network decomposition the area a node can control by its state is significantly reduced. Thus, the new algorithm only requires a polynomial number of moves to stabilize assuming an unfair distributed scheduler.

Chapter 5

Analysis of Local States and Sequences

The local state analysis (see Section 3.2.2) is a convenient method to show self-stabilization. If it can be applied, the proofs of termination are usually less complex compared to other methods, such as e.g. graph reduction which will be presented in Chapter 7. This chapter makes use of several techniques for the design and the analysis of a self-stabilizing algorithm that runs in an anonymous network.

Designing self-stabilizing algorithms for anonymous networks is inherently difficult. There are only a few positive results related to anonymous networks [AAFJ08]. The reason is the lack of a mechanism for breaking symmetry (cf. Section 2.1). This is best seen when considering completely symmetric graphs such as rings. When a self-stabilizing algorithm starts in a configuration where all nodes have the same initial state, then either no nodes or all nodes are enabled. In the latter case, under a synchronous scheduler, all nodes will make the same move and will assume a common state again. Thus, at any time, all nodes are in the same state and no node or edge stands out. This implies that under this assumption it is impossible e.g. to compute a minimum vertex cover or a maximal matching. At best it is possible to compute approximations of the optimal solution.

In the following, a self-stabilizing algorithm for the minimum vertex cover problem in an anonymous network is presented. It simulates a bipartite graph to calculate its result and its complexity is determined by analyzing the local states of the nodes.

The chapter is structured as follows: The next section introduces the vertex cover problem and summarizes related work for both this problem and self-stabilizing algorithms in an anonymous network in general. Section 5.2 introduces a basic version of the proposed algorithm with approximation ratio 3. This algorithm is extended in Section 5.3 to achieve an approximation of $3 - \frac{2}{\Delta+1}$. Section 5.4 concludes this chapter.

5.1 Example: Vertex Cover Approximation in Anonymous Networks

5.1.1 Introduction

A vertex cover of a graph is a set S of vertices such that each edge of the graph is adjacent to at least one vertex of S (see Figure 5.1).

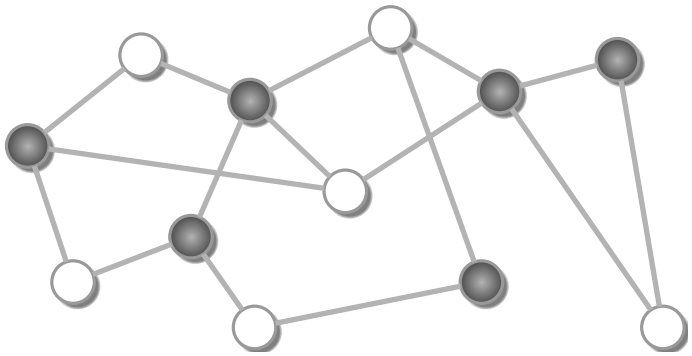


Figure 5.1: Minimal vertex cover S of a graph. The nodes in S are colored black.

Finding a minimum vertex cover is a classical optimization problem and is an example of an NP-hard problem. In a centralized setting, a simple polynomial-time 2-approximation algorithm is well-known. Distributed algorithms with an approximation ratio of less than two do exist, as long as a mechanism for symmetry breaking is available. This chapter considers distributed self-stabilizing algorithms in anonymous networks for the minimum vertex cover problem.

5.1.2 Related Work

A well-known procedure to compute a 2-approximation of a minimum vertex cover is based on the following observation: Any maximal matching (c.f. Section 2.4.6) implies a 2-approximation vertex cover by selecting all nodes incident to matched edges. Unfortunately it is impossible to establish a maximal matching with a distributed algorithm in general anonymous networks. In such networks a minimum vertex cover cannot be approximated with a ratio better than two (see Lemma 5.2.1). Polishchuk and Suomela developed a local algorithm (not self-stabilizing) that finds a 3-approximation vertex cover in anonymous networks [PS09]. So the question is whether there exist k-approximation algorithms with $k < 3$. In [ÅS10] Åstrand and Suomela compare several fast distributed algorithms with smaller approximation ratio but without the self-stabilization property. The contribution of this chapter is a fault-containing self-stabilizing $(3 - \frac{2}{\Delta+1})$-approximation algorithm for a minimum vertex cover in anonymous networks with port numbering using the distributed scheduler and the link-register model with composite atomicity. Note that the model used - deterministic self-stabilizing algorithms in anonymous networks with port numbers - is indeed a very weak model of distributed computing.

Self-stabilizing algorithms work with different schedulers. The central scheduler non-deterministically selects in every step a single node to make a move. Thus, this type of scheduler inherently provides a mechanism for symmetry breaking. Hence, algorithms running under this type of scheduler often work in anonymous networks. This includes the algorithm in [HH92] for solving the maximum matching problem and that in [BDGT09] for the coloring problem. Even under a central scheduler some problems cannot be solved at all under these premises. Shukla et al. proved that there is no self-stabilizing algorithm for coloring an arbitrary anonymous odd-degree bipartite network, using a central scheduler [SRR95]. This reference contains more impossibility results for the central scheduler scenario. Some special classes of anonymous networks such as trees or planar graphs allow self-stabilizing algorithms [AS97b, GHS06, SGH04, TJH07, XS05].

The situation is much more difficult for the distributed scheduler, because no symmetry breaking mechanism is available. Many graph optimization problems cannot be solved at all under this premise, in some cases even computing a good approximation is impossible. In general it is impossible to compute a 2-approximation

of a minimum vertex cover in anonymous networks (see Lemma 5.2.1).

Recently Polishchuk and Suomela developed a local algorithm that finds a 3-approximation vertex cover in anonymous networks [PS09]. Their algorithm is not self-stabilizing, though. Even though there exist generic techniques to transform synchronous distributed algorithms into the asynchronous world of self-stabilization such as synchronizers [Awe85], the transformed algorithms are usually complicated, require more memory or have a bad performance. Therefore, in this chapter a self-stabilizing algorithm for the minimum vertex cover problem with approximation ratio $3 - 2/(\Delta + 1)$ extending this concept is proposed.

The remaining part of this section reviews related work for networks with unique identifiers. Approximation algorithms for the minimum vertex cover problem have been studied extensively under this assumption. In a sequential setting algorithms with approximation ratios less than 2 are known to exist. However, Håstad showed that for any $\delta > 0$ it is NP-hard to approximate vertex cover within $(7/6 - \delta)$ [Hås01].

In a distributed setting, a 2-approximation can be achieved if a maximal matching is available. Hańćkowiak et al.'s distributed algorithm finds a maximal matching in $O(\log^4 n)$ rounds [HKP98], and Panconesi and Rizzi's algorithm finds a maximal matching in $O(\Delta + \log^* n)$ rounds [PR01]. Chattopadhyay et al. developed a self-stabilizing algorithm that stabilizes in $O(n^2)$ steps with a maximal matching for a fair distributed scheduler using the shared memory model with read/write atomicity [CHS02]. Later, Manne et al. presented an algorithm that stabilizes in $O(m)$ steps using an unfair distributed scheduler and the shared memory model with composite atomicity [MMPT09]. Both algorithms lead to a 2-approximation of a minimum vertex cover. Kiniwa's self-stabilizing algorithm calculates a $(2 - 1/\Delta)$-approximation vertex cover using the shared memory model with composite atomicity and the distributed scheduler [Kin05]. Kiniwa combined a greedy method based on a high-degree-first order of vertices with the maximal matching technique.

5.2 Basic Algorithm

5.2.1 Preliminaries

The link-register model with composite atomicity is used as the communication model [DIM93]. An overview of this model was provided in Section 2.1. More formally,

in the link-register model with composite atomicity a node v communicates with neighbors p and q using separate registers: r_{vp} is written by v and read by p, whereas r_{qv} is written by q and read by v. Reading the registers r_{qv} of all neighbors and updating all registers r_{vp} is one atomic operation. It is assumed that each node can distinguish the different edges that are incident to it, i.e., for each $v \in V$ there exists a bijection between the neighbors of v in G and $\{1,\ldots,d_v\}$. The numbers associated by each vertex to its neighbors are called port-numbers and the bijections are called a port-numbering of G. Port numbers are fixed and no assumption on the order of the port numbers is made.

In [Ang80] Angluin showed that it is impossible to break symmetry via a port numbering. The following well-known lemma shows that there can be no self-stabilizing algorithm computing a minimum vertex cover with an approximation ratio less than 2 in anonymous networks even when port numbering is available.

Lemma 5.2.1. *In a uniform anonymous network with port numbering a non-probabilistic self-stabilizing algorithm that calculates a k-approximation minimum vertex cover with $k<2$ for arbitrary graphs cannot exist under a distributed scheduler.*

Proof. Consider a ring with n nodes, where n is even and all nodes have the same state. Furthermore, assume all nodes to have the same port number for their left (resp. right) neighbor. If the scheduler always schedules all enabled nodes, then the states of all nodes will always coincide. Thus, all of them will be selected for the vertex cover. An optimal vertex cover only contains every second node. □

Let $G = (V,E)$ be an undirected graph with maximal degree Δ, $|V| = n$ and $|E| = m$. Let $S \subset V$ and $E_S = \{\langle u,v\rangle \in E \mid u,v \in S\}$, then $G|_S = (S,E_S)$ is called the subgraph of G induced by S. A vertex cover of G is a subset S of V such that each $e \in E$ is incident to at least one node of S. A vertex cover is called *minimum* if there is no other vertex cover that contains less nodes. $\gamma(G)$ denotes the size of a minimum vertex cover of G. For a line graph P with $|P|$ nodes $\gamma(P) = |P|/2$ if $|P|$ is even and $(|P|-1)/2$ if $|P|$ is odd. Let S be a vertex cover of G, then $r_G(S) = |S|/\gamma(G)$ denotes the approximation ratio of S. If U is a subgraph of G with vertex set U_V, then $r_U(S) = |S \cap U_V|/\gamma(U)$.

Lemma 5.2.2. *Let S be a vertex cover of G and $G_i = (V_i, E_i)$ for $i = 1, \ldots, s$ vertex disjoint subgraphs of G with $|E_i| > 0$ and $S \subseteq \cup_{i=1}^{s} V_i$. Then $r_G(S) \leq \max_{i=1,\ldots,s} r_{G_i}(S)$.*

Proof.

$$r_G(S) = \frac{|S|}{\gamma(G)} \leq \frac{\sum_i^s |S \cap V_i|}{\sum_i^s \gamma(G_i)} \leq \max_{i=1,\ldots,s} \frac{|S \cap V_i|}{\gamma(G_i)} = \max_{i=1,\ldots,s} r_{G_i}(S)$$

□

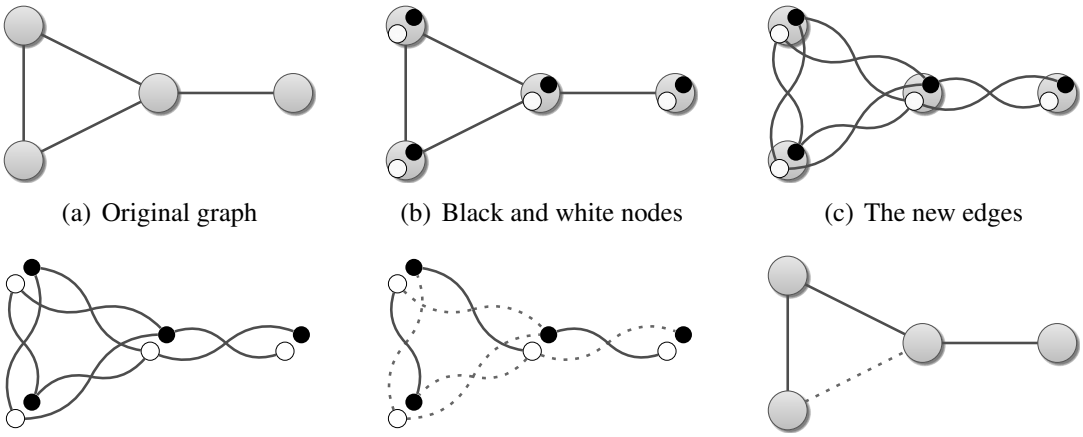
A matching is a subset M of independent edges of G. M is a *maximal matching* if there is no matching M' with $M \subset M'$. Clearly, the vertices of the edges of a maximal matching form a vertex cover which contains at most twice as many vertices as an optimal vertex cover. This observation does not lead to a 2-approximation algorithm for a vertex cover, since it is impossible to compute a maximal matching in an anonymous network. To circumvent this dilemma, this thesis introduces the concept of a *generalized matching*; this is a vertex disjoint set of subgraphs which are either paths or rings. A generalized matching M of G is called maximal, if every edge of G is incident to an edge belonging to a subgraph of M. The following result is crucial for the new approximation algorithm.

Lemma 5.2.3. *The vertices of a maximal generalized matching M of G form a vertex cover with at most three times as many nodes as an optimal vertex cover.*

Proof. Clearly, the maximality of M implies that the set S of vertices of the edges of M form a vertex cover of G. Furthermore, $r_U(S) \leq 3$ for each subgraph $U \in M$. Equality holds in case U is a path of length 3. Lemma 5.2.2 implies that S is a 3-approximation of a minimum vertex cover. □

5.2.2 Algorithm Description

To leverage the last lemma to find a 3-approximation of a minimal vertex cover, it is necessary to compute a maximal generalized matching in an anonymous network. This section presents a self-stabilizing algorithm for this purpose based on the *Kronecker double cover* $K(G)$ of a graph G. First the construction of $K(G) = G \otimes K_2$ is reviewed. $K(G)$ is constructed by making two copies of the vertex set of G (black and white nodes) and adding edges $\langle x_w, y_b \rangle$ and $\langle y_w, x_b \rangle$ for every edge $\langle x, y \rangle$ of G.



(a) Original graph (b) Black and white nodes (c) The new edges

(d) Kronecker double cover (e) Maximal matching of Kronecker double cover (f) Maximal generalized matching of original graph

Figure 5.2: The Kronecker Double Cover

Each edge of $K(G)$ can be uniquely associated with an edge of G. Figures 5.2(a) to 5.2(d) illustrate the construction of $K(G)$ for a graph with four nodes. The next lemma is straightforward to prove. It reveals an important property of $K(G)$, see also Figures 5.2(e) and 5.2(f).

Lemma 5.2.4. *Let $\mathcal{M}$ be a matching of $K(G)$ and M the set of edges of G that are associated with the edges of $\mathcal{M}$. If $\mathcal{M}$ is a maximal matching of $K(G)$, then M is a maximal generalized matching of G.*

The key to compute a maximal matching of $K(G)$ in anonymous networks is the fact that $K(G)$ is a bicolored graph. Hańćkowiak et al. have developed an algorithm that can be used for this end [HKP98]. Their algorithm works in synchronous networks, therefore it is necessary to rewrite the algorithm such that it is self-stabilizing and can be executed on G (as opposed to $K(G)$). The idea of the algorithm of Hańćkowiak is simple. Assume the nodes are colored black and white. Black nodes make offers to unmatched white nodes and white nodes choose one of the "offering" black nodes. This process is repeated until no more offers can be made. Edges corresponding to accepted offers form a maximal matching. Then Lemma 5.2.4 and 5.2.3 lead to a 3-approximation algorithm. The same approach is used by Polishchuk and Suomela to develop a local algorithm that finds a 3-approximation vertex cover in anonymous networks [PS09].

This section introduces the details of the algorithm as outlined above. It will be refined in Section 5.3 to achieve the stated approximation ratio. The graph $K(G)$ will not be represented explicitly by the algorithm, only the edges belonging to the matching of $K(G)$ will be explicitly stored. For this purpose each node v of G defines two variables that store the port numbers of neighbors: $v.black$ and $v.white$. It is assumed that these variables contain at any time a valid port number of a neighbor. They will be referred to as the black and white pointer of a node. In the context of a node v the names of neighboring nodes are identified with their corresponding port numbers. Thus, the expression $v.white = w$ is true, if $w \in N(v)$ and $v.white$ equals the port number of w from v's point of view.

Algorithm 5.1 (page 75) works as follows: Vertices try to have their black (resp. white) pointer to point to a neighbor whose white (resp. black) pointer points back to them. Each pair of nodes pointing to each other in this sense corresponds to a matched edge in $K(G)$. The following two predicates characterize situations in which a node v participates in such a matching.

- $blackMatched(v) \equiv v.black \neq v \wedge v.black.white = v$
- $whiteMatched(v) \equiv v.white \neq v \wedge v.white.black = v$

The term $v.black \neq v$ (resp. $v.white \neq v$) is necessary to exclude nodes that point to themselves. The following notation is introduced for such pointers: The black (resp. white) pointer of a node v is said to be *free* if $v.black = v$ (resp. $v.white = v$). A *node* is called free if both its pointers are free. A pointer *is freed* if it is set to point to the node itself. A node v can assign to its black (resp. white) pointer only a neighbor x with $x.white = x$ (resp. $x.black = v$). The following two functions select a neighbor to point to according to these rules. The function $select(S)$ selects one element of the specified set S in a deterministic manner. If nodes are equipped with port numbers this operator can be implemented by the minimum function. For the algorithm it is irrelevant *which* node is chosen. The definition of the select operation for a node v is extended, such that $v.select(\emptyset) = v$.

- $selectWhite(v) = v.select\{x \in N(v) \mid x.white = x\}$
- $selectBlack(v) = v.select\{x \in N(v) \mid x.black = v\}$

Algorithm 5.1 Maximal Generalized Matching / 3-Approximation Vertex Cover

Predicates:
$blackMatched(v) \equiv v.black \neq v \wedge v.black.white = v$
$whiteMatched(v) \equiv v.white \neq v \wedge v.white.black = v$

Functions:
$selectWhite(v) = v.select\{x \in N(v) \mid x.white = x\}$
$selectBlack(v) = v.select\{x \in N(v) \mid x.black = v\}$

Actions:
$\mathsf{R}_1 :: [\neg whiteMatched(v) \wedge (v.white \neq v \vee v.white \neq selectBlack(v))]$
$\longrightarrow$ **if** $(v.white \neq v)$ **then**
$v.white := v$
else
$v.white := selectBlack(v)$

$\mathsf{R}_2 :: [\neg blackMatched(v) \wedge (v.black = v \vee v.black.white \neq v.black) \wedge$
$v.black \neq selectWhite(v)]$
$\longrightarrow v.black := selectWhite(v)$

Rule R_1 sets variable *white* of a node. If it points to a neighbor that does not point back to it with its black pointer, the variable is freed. If a node has a free white pointer and a neighbor points with its black pointer towards it, it sets its white pointer to this neighbor. Freeing the white pointer ensures that a node executes rule R_1 at most twice.

Rule R_2 controls a node's black pointer. If it points to a node which points with its white pointer to another node itself, the pointer will be set to a neighboring node that has a free white pointer, or it is freed if there is no such neighbor.

An execution of rule R_1 (resp. R_2) will also be referred to as *white* (resp. *black*) *move*. Rule R_1 has higher priority than R_2, i.e. if a node is enabled to make a white and a black move, it will only make the white move.

5.2.3 Analysis

Lemma 5.2.5. *If all nodes are disabled with respect to Algorithm 5.1 then all pointers are either matched or free and the set of matched edges forms a maximal generalized matching.*

Proof. If a node's white (resp. black) pointer is neither matched nor free, rule R_1 (resp. R_2) is enabled for this node. Let $\langle u, v \rangle$ be an edge. Since all nodes are disabled at least one of the nodes u and v is matched via at least one of its pointers. Otherwise, because rule R_1 is not enabled, u and v would both have a free white pointer and hence, rule R_2 would be enabled for both nodes. Because no node is enabled for rule R_2 there cannot be a pair u, v of adjacent nodes such that u's white and v's black pointer is free. Hence, the black and white pointers represent a maximal matching of $K(G)$. According to Lemma 5.2.4 the set of matched edges forms a maximal generalized matching. □

Lemma 5.2.6. *While executing Algorithm 5.1 a node directs its black pointer towards a given neighbor at most once.*

Proof. According to Algorithm 5.1, for u being enabled to point to v, v must have a free white pointer. Hence, the black pointer of u (and of all other nodes that point towards v) is disabled until v moves its white pointer. After that v's white pointer is matched with one of the nodes that were pointing at it and will not make a move again. Besides, from that time on, a node cannot point towards v with its black pointer since v does not free its white pointer. □

Let S be the set of nodes that are matched via their black or white pointer.

Theorem 5.2.1. *Algorithm 5.1 stabilizes after $O(n+m)$ moves under the distributed scheduler and the set S is a 3-approximation of a minimum vertex cover.*

Proof. There are at most two white moves per node. A white pointer can first be freed if it is not free already and then it accepts an offer from a black pointer. This black pointer cannot move unless the white pointer is directed to another node, thus, there will be no further white moves. Hence, there are at most $2n$ white moves in total. According to Lemma 5.2.6 a node v can point to any node only once via its black pointer. Thus, its black moves are limited by $2d(v)$. Hence, the total number of moves is at most $2n + 2\sum_{v \in V} d(v) = 2(n+2m)$. Lemma 5.2.5 implies that the matched edges of G form a maximal generalized matching of G. Thus, Lemma 5.2.3 implies that S is a 3-approximation of a minimum vertex cover. □

The following example proves that the bound on the time complexity is tight.

Example 5.2.1. *Let n be an even number and consider the complete Graph K_n. Initially, all nodes have set their white pointer to any node and all black pointers are free. A distributed scheduler is assumed and the execution is broken into* phases. *The first phase consists of one step: All nodes free their white pointer. Now $n/2$ phases follow that consist of the following moves each:*

1. *Let $U = \{v \mid \neg blackMatched(v)\}$. Choose any node x from U. Every node $y \in U \backslash \{x\}$ points with its black pointer towards x and the black pointer of x points to an arbitrary node y in $y \in U \backslash \{x\}$.*

2. *x and y let their white pointers point towards each other.*

Thus, after every phase two more nodes are matched via both pointers. This results in $2n$ white and $\sum_{i=0}^{n/2}(n-2i)$ black moves and hence, $O(n+m)$ moves in total.

5.3 Approximation Ratio Improvement

Algorithm 5.1 computes a vertex cover S consisting of all nodes that have at least one pointer matched. The idea of the improved algorithm is to check whether a node v that is matched with exactly one pointer has only neighbors that have both their pointers matched. If this is the case $S \backslash \{v\}$ is still a vertex cover since all of v's neighbors are in S. If two neighboring nodes have the same pointer matched, or if they are both matched with both pointers, it is impossible to remove exactly one of them from S due to the impracticality of symmetry breaking. Figure 5.3 provides an example for this approach. To mark nodes that belong to the vertex cover a Boolean variable vc is introduced.

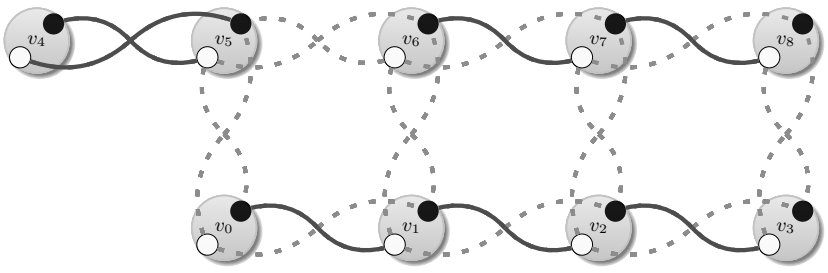


Figure 5.3: Nodes v_0 and v_6 can be excluded from the vertex cover S. The adjacent nodes v_3 and v_8 both have their white pointer matched. Hence, they cannot be excluded from S.

The following predicate characterizes vertices that are candidates to be excluded from the cover.

- $candidate(v) \equiv ((blackMatched(v) \vee v.black = v) \wedge v.white = v) \vee$
  $((whiteMatched(v) \vee v.white = v) \wedge v.black = v)$

For a node it is impossible to determine whether a neighbor has both pointers matched. Therefore, the above stated criteria cannot be assessed by the node itself. To overcome this problem a second Boolean variable dm is introduced. The purpose of dm is to signal to a node's neighbors that both pointers are matched (*doubly matched*). Algorithm 5.2 consists of the rules of Algorithm 5.1 and additionally four rules that change the values of the Boolean variables vc and dm. Rule $\mathsf{R_i}$ has a higher priority than rule $\mathsf{R_j}$ for $i < j$.

Algorithm 5.2 Vertex Cover with Approximation Ratio $3 - 2/(\Delta + 1)$

Predicates and Functions:
All predicates and functions of Algorithm 5.1

$candidate(v) \equiv ((blackMatched(v) \vee v.black = v) \wedge v.white = v) \vee$
$((whiteMatched(v) \vee v.white = v) \wedge v.black = v)$

Actions:
$\mathsf{R_1}$ and $\mathsf{R_2}$ as in Algorithm 5.1

$\mathsf{R_3} :: [whiteMatched(v) \wedge blackMatched(v) \wedge v.dm = false]$
$\longrightarrow v.dm := true$

$\mathsf{R_4} :: [(\neg whiteMatched(v) \vee \neg blackMatched(v)) \wedge v.dm = true]$
$\longrightarrow v.dm := false$

$\mathsf{R_5} :: [(\forall x \in N(v) : x.dm = true) \wedge candidate(v) \wedge v.vc = true]$
$\longrightarrow v.vc := false$

$\mathsf{R_6} :: [((\exists x \in N(v) : x.dm = false) \vee \neg candidate(v)) \wedge v.vc = false]$
$\longrightarrow v.vc := true$

Lemma 5.3.1. *Algorithm 5.2 stabilizes after $O(n + m)$ moves resp. $O(\Delta)$ rounds under the distributed scheduler.*

Proof. If a node is *whiteMatched* (resp. *blackMatched*) it remains so forever as long as no transient error occurs. Hence, rules R_3 and R_4 are executed $O(n)$ times in total. Thus, if a node sets its dm variable to *true* then this assignment is never changed. The predicate $candidate(v)$ can change its value at most $d(v)$ times. Furthermore, the expression $\forall x \in N(v) : x.dm = true$ can also change its value at most $2d(v)$ times. This implies that rules R_5 and R_6 are executed $O(m)$ times in total for G. Theorem 5.2.1 now implies that Algorithm 5.2 stabilizes after $O(n+m)$ moves in total.

In the following it will be shown that after the first round for every node v the number of neighbors with unmatched white pointers is reduced at least every second round, or v matches its own black pointer during that time. Note that due to the priority of R_1 over R_2, after the first round all white pointers are either matched or free. Therefore any node will make at most one further white move and this move will match the white pointer.

Since all unmatched white pointers are free, a node v that has an unmatched black pointer and at least one neighbor with an unmatched white pointer is enabled to execute rule R_1. Let a *phase* of v consist of two rounds:

1. In the first round node v points towards the white pointer of a neighbor w via its black pointer.

2. In the second round node w matches its white pointer with one of its neighbor's black pointer.

Note that a phase can be prolonged by one further round in which node v matches its own white pointer, but only once. Hence, after every phase the number of neighbors that have an unmatched white pointer is reduced by at least 1 or v has matched its own black pointer during that time. As a result, after $d(v)$ phases all neighbors of v have their white pointer matched or v has matched its own black pointer. Thus, after $2\Delta + 2$ rounds all white pointers are matched. There may be one further round in which some unmatched black pointers have to be freed. Finally, there may be two additional rounds in which nodes set their dm and their vc variables. Hence, after at most $2\Delta + 5$ rounds Algorithm 5.2 has stabilized. □

The following lemma shows that Algorithm 5.2 computes a vertex cover. Let $S = \{v \in V \mid v.vc = true\}$.

Lemma 5.3.2. *If all nodes of G are disabled with respect to Algorithm 5.2 then S is a vertex cover of G.*

Proof. Let v be a node with $x.vc = \mathit{false}$. It suffices to prove that all neighbors of v have their vc variable set to *true*. Assume, there is a node $x \in N(v)$ with $x.vc = \mathit{false}$. According to the assumption, rules R_1 and R_2 are not enabled for both nodes. If one of the two nodes has both pointers matched then it is enabled to execute R_5, hence both nodes have at least one free pointer. If one of the two nodes has two free pointers then one of them is enabled to execute R_2, so assume both nodes to have exactly one free pointer. If these pointers have the same color then rule R_5 is enabled, otherwise the node that has a free black pointer is enabled to point to the other one via rule R_2. This contradiction concludes the proof. □

Theorem 5.3.1. *Algorithm 5.2 computes a $(3 - \frac{2}{\Delta+1})$-approximation vertex cover.*

Proof. Consider a configuration where no node is enabled with respect to algorithm 5.2. The idea of the proof is to construct a set $\mathcal{C}$ of subgraphs of G, such that S is a subset of the union of the vertices of these subgraphs and $r_P(S) \leq 3 - \frac{2}{\Delta+1}$ for all $P \in \mathcal{C}$. Then $r_G(S) \leq 3 - \frac{2}{\Delta+1}$ by Lemma 5.3.2 and 5.2.2 and the proof is complete.

According to Lemma 5.2.5 the set of matched edges forms a maximal generalized matching. Let $\mathcal{P}$ denote the set of connected subgraphs of this matching. Insert all subgraphs of $\mathcal{P}$ that are rings, paths of length two or paths with an endpoint v with $v.vc = \mathit{false}$ and $v.white = v$ into $\mathcal{C}$. Clearly $r_P(S) \leq 2 < 3 - \frac{2}{\Delta+1}$ for all $P \in \mathcal{C}$. Note that vertices of rings and paths of length two have no free pointer.

Each $P \in \mathcal{P} \backslash \mathcal{C}$ is a path of length at least three and has an endpoint v_P with a free white pointer and $v_P.vc = \mathit{true}$. The black pointer of the other endpoint of P is also free. The following notation is used throughout the proof: If v is a node with a free white pointer and $v.vc = \mathit{true}$, then P_v denotes the corresponding path in $P \in \mathcal{P} \backslash \mathcal{C}$. Conversely, if $P \in \mathcal{P} \backslash \mathcal{C}$ then v_P denotes the endpoint of P with v_P having a free white pointer and $v_P.vc = \mathit{true}$.

Let $D = \{v_P \mid P \in \mathcal{P} \backslash \mathcal{C}\}$ and let $G|_D$ be the subgraph of G induced by D. Note that each $v \in D$ has at least one neighbor in D, otherwise $v.vc$ would be *false*. Let M be a maximal matching of G_D. If M contains an edge (a, b) such that P_a has odd and P_b has even length, and such that a has an unmatched neighbor $u \in D$ where P_u

has odd length, then replace edge $\langle a, b\rangle$ of M by $\langle a, u\rangle$. Thus, one may safely assume that if for an edge $\langle a, b\rangle \in M$ the path P_a has odd length and P_b has even length, then a has no unmatched neighbor $u \in D$ such that P_u has odd length.

Uniquely associate with each unmatched node $u \in D$ for which P_u has odd length a matched node of G_D. For each matched node v denote by U_v the set of unmatched nodes that are associated with v. Furthermore, for each matched node x let $\mathcal{P}_x$ be the set of paths P from $\mathcal{P}\backslash\mathcal{C}$ with $v_P \in U_x$. Note that all paths in U_x have odd length. Add each even length path P from $\mathcal{P}$ where v_P is not matched to the set $\mathcal{C}$. Note that $r_P(S) \leq 2$ for such P.

For each $m = \langle a, b\rangle \in M$ a subgraph G_m of G will be constructed. G_m consists of the paths P_a, P_b, all paths in $\mathcal{P}_a \cup \mathcal{P}_b$ and the additional edges $\langle a, x\rangle$ for $x \in U_a$, $\langle y, b\rangle$ for $y \in U_b$, and $\langle a, b\rangle$. Denote by δ_x the degree of node x in G_m. Clearly, $\delta_a = |U_a| + 2$ and $\delta_b = |U_b| + 2$. Figure 5.4 shows the general structure of G_m, edges belonging to $\mathcal{M}$ are depicted in bold. Add the subgraphs in $\{G_m \mid m \in \mathcal{M}\}$ into $\mathcal{C}$. Note that the graphs contained in $\mathcal{C}$ are vertex disjoint and that S is contained in the union of the vertex sets of these graphs.

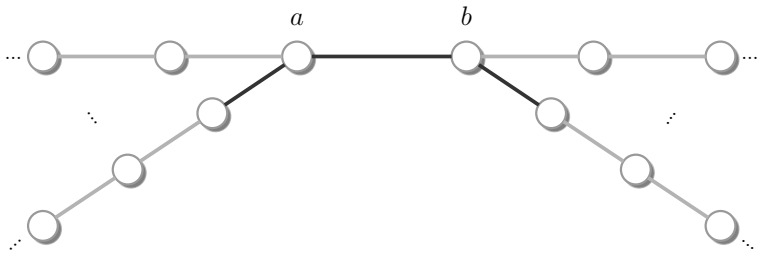


Figure 5.4: General structure of the graph G_m for $m \in \mathcal{M}$

Let $m = \langle a, b\rangle$ be an edge of M. In the following it will be shown that $r_{G_m}(S) \leq 3 - \frac{2}{\Delta+1}$ holds for all $m \in M$. Note that the vertices of G_m form a subset of S of size

$$s = |P_a| + |P_b| + \sum_{x \in U_a} |P_x| + \sum_{x \in U_b} |P_x|.$$

Denote with P_{ab} the path resulting from joining the paths P_a and P_b with the edge $\langle a, b\rangle$. The union of minimum vertex covers of the paths in $\mathcal{P}_a \cup \mathcal{P}_b$ and a particular vertex cover $\mathcal{C}_{ab}$ of P_{ab} forms a minimum vertex cover of G_m. If $\mathcal{P}_a \neq \emptyset$ (resp. $\mathcal{P}_b \neq \emptyset$) then a (resp. b) must be in $\mathcal{C}_{ab}$. $\mathcal{C}_{ab}$ is a minimum vertex cover of G_m

with respect to this constraint. To have a general term for $\gamma(G_m)$ the size of $\mathcal{C}_{ab}$ is expressed with the help of a parameter ϵ:

$$|\mathcal{C}_{ab}| = (|P_a| + |P_b| + \epsilon)/2$$

The value of ϵ will be determined through a case by case analysis. These considerations lead to the following expression for $\gamma(G_m)$:

$$\begin{aligned}\gamma(G_m) &= \left(|P_a| + |P_b| + \epsilon + \sum_{x \in U_a} (|P_x| - 1) + \sum_{x \in U_b} (|P_x| - 1)\right)/2 \\ &= (s - (\delta_a + \delta_b - 4 - \epsilon))/2\end{aligned}$$

Thus,

$$r_{G_m}(S) = \frac{s}{\gamma(G_m)} = \frac{2s}{s - (\delta_a + \delta_b - 4 - \epsilon)}$$

To derive an upper bound for $r_{G_m}(S)$, this expression will be analyzed with respect to monotonicity. $r_{G_m}(S)$ is strictly monotonic decreasing with respect to s if $\delta_a + \delta_b \geq 4 + \epsilon$. Thus, if s_{min} is a minimal value for s and $\delta_a + \delta_b \geq 4 + \epsilon$ then

$$r_{G_m}(S) \leq \frac{2s_{min}}{s_{min} - (\delta_a + \delta_b - 4 - \epsilon)} \qquad (*)$$

The set $\mathcal{C}_{ab}$ will be determined by looking at three different cases.

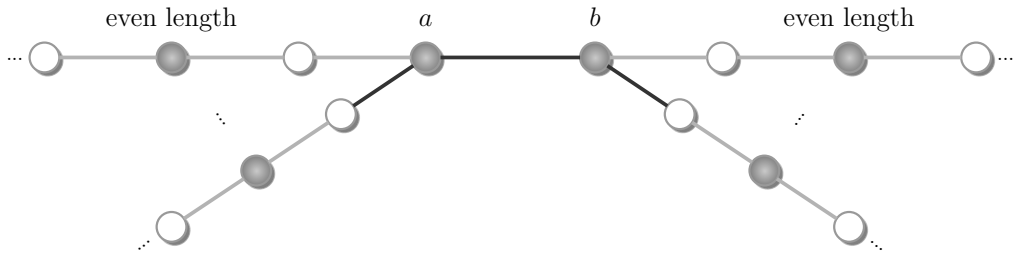


Figure 5.5: Structure of G_m in case P_a and P_b have even length.

Case 1: Both P_a and P_b have even length (see Figure 5.5).
In this case $\mathcal{C}_{ab}$ is equal to the minimum vertex cover for P_{ab} that includes nodes a and b, thus $\epsilon = 0$ and $\delta_a + \delta_b \geq 4 + \epsilon$. The bold vertices in Figure 5.5 indicate the set

$\mathcal{C}_{ab}$. The value of s is minimized for $|P_a| = |P_b| = 4$ and $|P_x| = 3$ for $x \in U_a \cup U_b$. Hence, $s_{min} = 3(\delta_a + \delta_a) - 4$. Using $(*)$ the following holds:

$$r_{G_m}(S) \leq \frac{2(3(\delta_a + \delta_b) - 4)}{3(\delta_a + \delta_b) - 4 - (\delta_a + \delta_b - 4)} = \frac{3(\delta_a + \delta_b) - 4}{\delta_a + \delta_b} < 3 - \frac{2}{\Delta + 1}$$

Case 2: Both P_a and P_b have odd length (see Figure 5.6).
First the subcase that both a and b have a degree strictly larger than 2 is considered. Then $\mathcal{C}_{ab}$ is equal to the minimum vertex cover for P_{ab} that includes nodes a and b, thus $\epsilon = 2$ and $\delta_a + \delta_b \geq 4 + \epsilon$. The bold vertices in Figure 5.6 indicate the set $\mathcal{C}_{ab}$. Furthermore, the value of s is minimized for $|P_a| = |P_b| = 3$ and $|P_x| = 3$ for $x \in U_a \cup U_b$. Hence, $s_{min} = 3(\delta_a + \delta_a) - 6$. Using $(*)$ the following holds:

$$r_{G_m}(S) \leq \frac{2(3(\delta_a + \delta_a) - 6)}{3(\delta_a + \delta_a) - 6 - (\delta_a + \delta_b - 4 - 2)} = \frac{3(\delta_a + \delta_b) - 6}{\delta_a + \delta_b} < 3 - \frac{2}{\Delta + 1}$$

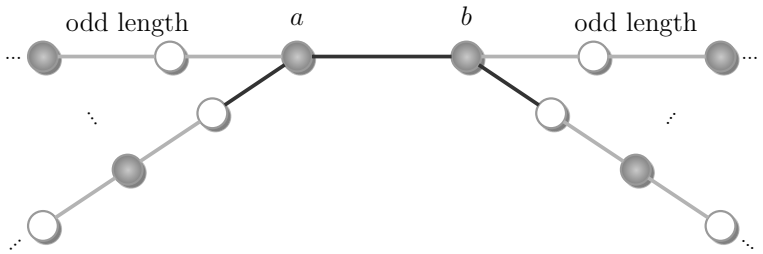


Figure 5.6: Structure of G_m in case P_a and P_b have odd length.

If $\delta_a = 2$ and $\delta_b > 2$ then $\mathcal{C}_{ab}$ is equal to the minimum vertex cover for P_{ab} that includes node b, thus $\epsilon = 0$ and $\delta_a + \delta_b \geq 4 + \epsilon$. The value of s is minimized for $|P_a| = |P_b| = 3$ and $|P_x| = 3$ for $x \in U_b$. Hence, $s_{min} = 3\delta_b$. Using $(*)$ the following holds:

$$r_{G_m}(S) \leq \frac{6\delta_b}{3\delta_b - (2 + \delta_b - 4)} = \frac{3\delta_b}{\delta_b + 1} < 3 - \frac{2}{\Delta + 1}$$

Case 3: P_a has odd and P_b has even length, $\delta_a = 2$, and $\delta_b \geq 2$ (see Figure 5.7).
In this case $\mathcal{C}_{ab}$ is equal to the minimum vertex cover for P_{ab} that includes node b, thus $\epsilon = -1$ and $\delta_a + \delta_b \geq 4 + \epsilon$. The bold vertices in Figure 5.7 indicate the set

$\mathcal{C}_{ab}$. Furthermore, the value of s is minimized for $|P_a| = 3$, $|P_b| = 4$ and $|P_x| = 3$ for $x \in U_b$. Hence, $s_{min} = 3\delta_b + 1$. Using $(*)$ the following holds:

$$r_{G_m}(S) \leq \frac{6\delta_b + 2}{3\delta_b + 1 - (2 + \delta_b - 4 + 1)} = \frac{3\delta_b + 1}{\delta_b + 1} \leq 3 - \frac{2}{\Delta + 1}$$

□

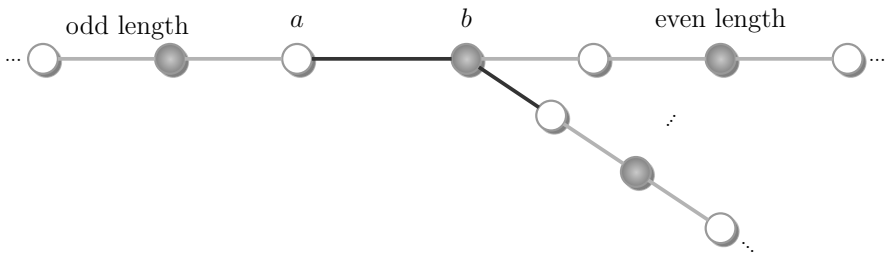


Figure 5.7: Structure of G_m in case that P_a has odd and P_b has even length and $\delta_a = 2$

The following example demonstrates that the approximation ratio of Algorithm 5.2 is no better than $3 - \frac{3}{\Delta+1}$.

Example 5.3.1. *Let G_k be the graph depicted in Figure 5.8. The optimal vertex cover consists of nodes l, r, and $m_1, \ldots, m_k$. Hence, $\gamma(G_k) = k + 2$. However, in the worst case scenario Algorithm 5.2 selects all nodes of G_k. Figure 5.8 displays such a configuration: Nodes m_i and t have both pointers matched and nodes t and m_i have their white (resp. black) pointers matched with l, l_i (resp. r, r_i). Any node that has an unmatched pointer has a neighbor with an unmatched pointer of the same color. Thus, the node is not enabled to set its vc variable to false. Hence, the approximation ratio is $3 - 3/(\Delta + 1)$. This example corresponds to the second subcase of case 2 of Theorem 5.3.1.*

Algorithm 5.2 achieves approximation ratio two if applied to trees. The following lemma is well-known. The proof is included for reasons of completeness.

Lemma 5.3.3. *Let $T = (V, E)$ be a tree with $|V| > 2$ and I the set of inner nodes, i.e. $I = \{v \in V \mid d(v) > 1\}$. Let M be a maximum cardinality matching of T. Then I is a vertex cover of T and $|I| < 2|M|$.*

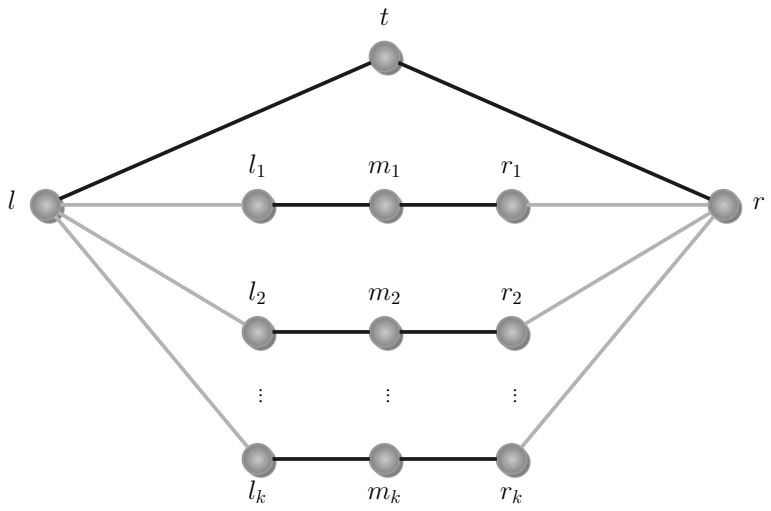


Figure 5.8: Worst case example for Algorithm 5.2

Proof. I is a vertex cover of T. The rest is shown by induction on the number of nodes. The statement holds for $|V| = 3$. Let $|V| > 3$, and x be a leaf of T. The neighbor of x is denoted by y. Two cases are distinguished.

Case $d(y) > 2$: Let $T' = T \backslash \{x\}$. The set of inner nodes is not changed by removing node x, i.e. $I' = I$. By induction $|I'| < 2\,|M'|$, where M' is a maximum cardinality matching of T'. Thus, $|I| = |I'| < 2\,|M'| \leq 2\,|M|$.

Case $d(y) = 2$: Let $T' = T \backslash \{x, y\}$. Node $y \notin I'$ and maybe its other neighbor is a leaf in T', hence $|I'| \leq |I| + 2$. By induction $|I'| < 2\,|M'|$, where M' is a maximum cardinality matching of T'. $M' \cup \{\langle x, y \rangle\}$ is a maximum cardinality matching of T, thus, $|M'| + 1 = |M|$. This yields $|I| = |I'| + 2 < 2\,|M'| + 2 \leq 2\,|M|$. □

Theorem 5.3.2. *For trees Algorithm 5.2 calculates a 2-approximation vertex cover.*

Proof. Let $T = (V, E)$ be a tree. Lemma 5.3.2 yields that S is a vertex cover. Let M_1 be the set of nodes that are matched to one neighbor via both pointers each, i.e. $M_1 = \{v \in V \mid v.black = v.white \neq v\}$.

Let $T' = T \backslash M_1$. Assume there exists a leaf x of T' such that $x.vc = true$. Then, x must be matched with a node y via at least one of its pointers. If its other pointer is not free, then it must have a second neighbor in T' and therefore it is not a leaf. Note that a vertex of T' cannot be matched with the same neighbor via both its pointers. If y's second pointer is free one of the two nodes is enabled to execute rule R_2, hence y is also matched with another node. All other neighbors of x in T belong to M_1 anyway,

thus rule R_3 must be enabled for x. This contradiction shows that $x.vc = \mathit{false}$ for all leaves x of T'.

Let I' be the set of nodes of T' without the leaves and let M_2 be a maximum cardinality matching of T'. From Lemma 5.3.3 follows that $I' < 2\,|M_2|$. The set $M_1 \cup M_2$ is a maximal matching of T. Hence, König's theorem yields: $|S| \leq 2\,|M_1| + |I'| < 2\,|M_1| + 2\,|M_2| = 2\,|M_1 \cup M_2| \leq 2\,|S_{opt}|$. □

It is an open question whether Theorem 5.3.2 can be extended to other classes of graphs. Example 5.3.1 shows that the theorem does not hold for bipartite graphs.

5.4 Conclusion

This chapter presented a self-stabilizing algorithm for a $(3 - \frac{2}{\Delta+1})$-approximation minimum vertex cover in anonymous networks using the distributed scheduler and the link-register model with composite atomicity. It stabilizes after $O(n+m)$ moves resp. $O(\Delta)$ rounds. The algorithm achieves a 2-approximation if it is executed on a tree.

Finding a linear self-stabilizing algorithm requiring only $O(\log n)$ storage per node with a better approximation remains an open problem.

Chapter 6

Analysis of Local States and Sequences (II)

Like Chapter 5, this chapter will perform an analysis based on local information. Since this chapter assumes unique identifiers instead of an anonymous network it is possible to design the algorithms in a different way: The identifiers facilitate the use of distance-k information (c.f. Section 2.3.2). The expression model of [Tur12] will be used to make 2-hop knowledge available to the nodes. Furthermore, this model allows to *design* the algorithms for the central scheduler while they can be *executed* under the distributed scheduler via the transformer of [Tur12].

This chapter presents the first self-stabilizing algorithms for the *edge-monitoring problem*. It has important applications e.g. in network security. Two versions of this problem are considered in this chapter: The minimal edge-monitoring problem, which corresponds to the original problem definition by Dong et al. ([DLL08]), and an edge-aware version, which yields a result more suitable for wireless networks. Using the distributed scheduler both algorithms stabilize after at most $O(mn^2)$ moves.

This chapter is structured as follows: At first, a definition of the edge-monitoring problem is given and related work is reviewed. Section 6.2 presents self-stabilizing algorithms for the above mentioned versions of the problem. The results are summarized in Section 6.3.

6.1 Example: Edge Monitoring

6.1.1 Introduction

A rather new problem in graph theory is the finding of a minimal edge monitoring. A node can *monitor* an edge if it is a neighbor of both of its adjacent nodes. The minimal edge-monitoring problem consists of identifying a minimal set of nodes that are able to monitor a given subset of the edges of a graph (see Figure 6.1). Dong et al. defined the minimal edge-monitoring problem and proved it to be *NP*-complete in case the set is required to have minimum *cardinality* [DLL08, DLL+11].

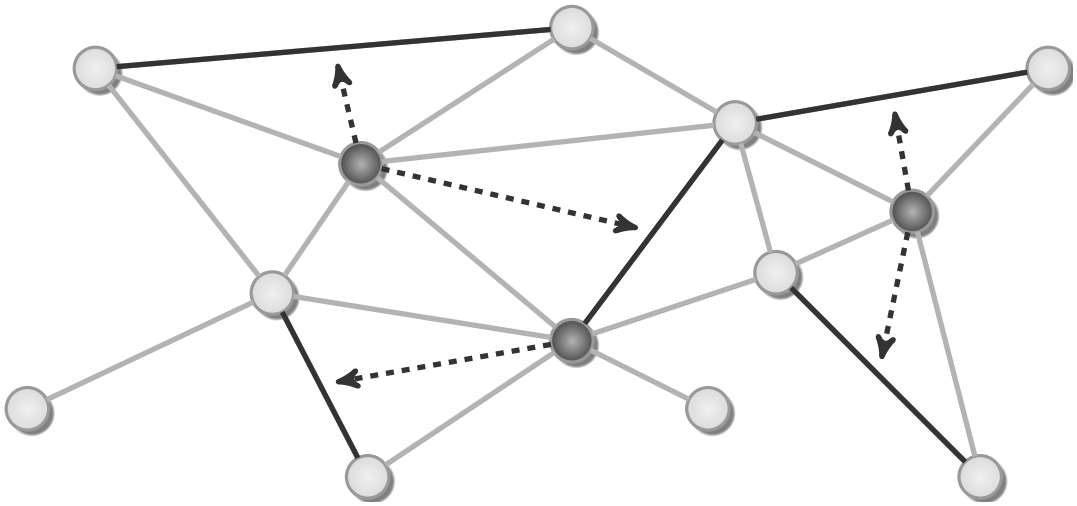


Figure 6.1: Edge monitoring of a graph. The arrows indicate the edges that a node monitors.

To be more formal, let $G = (V, E)$ be an undirected graph. Given an edge $e = \langle y, z \rangle$, node x can *monitor* e if $\langle x, y \rangle, \langle x, z \rangle \in E$. A set of nodes $V' \subseteq V$ defines a *k-monitoring* of a set of edges $E' \subseteq E$ if all edges of E' are monitored by at least k different nodes in V'. A k-monitoring V' of E' is *minimal* if no subset of V' is a k-monitoring of E'. Note that V' is minimal with respect to the described property, it does not have to be the set with minimum cardinality.

6.1.2 Related Work

Wireless sensor networks consist of many nodes that communicate via radio. They facilitate a lot of applications e.g. military, health or environmental surveillance

applications [ASSC02]. Due to their wireless communication they are prone to various attacks, such as interception, subversion, falsification, denial of service or physical corruption [GSRV08]. Analyzing traffic flow is one method to secure the network against compromised nodes. Dong et al. were the first to describe the *minimal edge-monitoring problem* to retrieve a minimal set of nodes that are able to observe particular communication links of a network [DLL08, DLL+11]. They provided two distributed algorithms to solve it. Dong et al. also suggested using *local* monitors for the edges since it reflects the broadcast nature of wireless communication: As illustrated in Figure 6.2, a node within the communication range of both a sending and a receiving node can monitor their interactions.

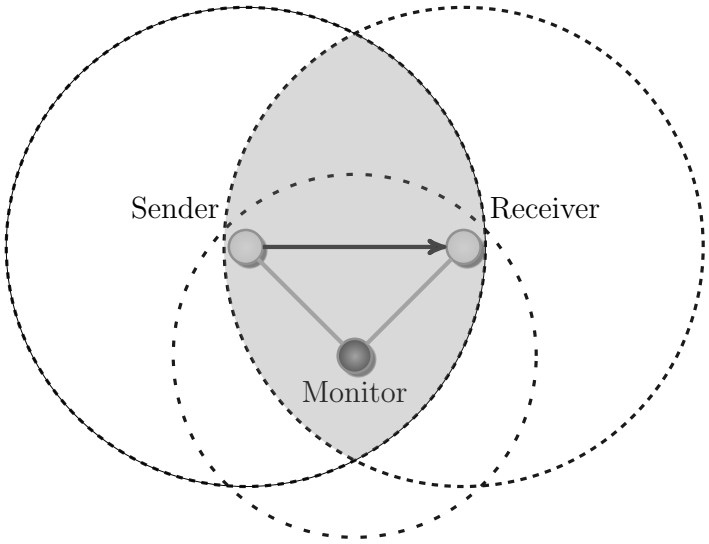


Figure 6.2: Monitor of an edge

Wireless network diagnosis is a field of research that has been extensively studied. Most approaches assume a central sink that gathers information from the whole network to decide which measures have to be taken, e.g. [LLL10]. Distributed edge-monitoring algorithms that allow local decisions can be found in [DLL+11] and [DLL08]. Hsin and Liu developed a distributed mechanism to monitor nodes instead of communication links [HL06]. The surveillance of nodes is also the object of the self-protection problem [WZL07, WLZ08]. It consists of selecting a set of "protecting" nodes and is closely related to the dominating set problem [HL91, Tur07]. Giruka et al. give an overview of security in wireless sensor networks [GSRV08]. More references

on this topic can be found in [DLL08]. Akyildiz et al. provide a general survey on wireless sensor networks [ASSC02].

6.2 Basic Algorithm

This section proposes the first *self-stabilizing* algorithms for the edge-monitoring problem. Several aspects of this problem are considered: The first algorithm simply establishes a minimal monitoring. The second one provides the nodes with more information about the edges they have to monitor.

6.2.1 Preliminaries

The algorithms presented in this chapter run under the distributed scheduler assuming the distance-one model, i.e. a node can see its own state and the state of its neighbors. The nodes have unique identifiers and the shared memory model with composite atomicity (see Section 2.1) is assumed for communication. The algorithms are *modeled* in the *expression-two model* presented in [Tur12] (cf. Section 2.3.2) and using the central scheduler. In order to run under the distributed scheduler in the distance-one model the transformation of [Tur12] is applied which results in a slowdown factor of $O(m)$ moves.

As in [DLL+11] for all edges that are supposed to be monitored a specific monitoring demand $desired(e) \geq 0$ for each edge e is defined. This number describes how many nodes are supposed to monitor e. The adjacent nodes are aware of the edge's *desired* value. The set of edges to be monitored is denoted by E_S, i.e. $E_S = \{e \in E \mid desired(e) > 0\}$.

6.2.2 Simple Edge Monitoring Algorithm

In this section the first algorithm that calculates a minimal edge monitoring is described. All nodes hold a Boolean variable *state* that can assume the values IN or OUT. A node in state IN monitors all edges that it can monitor whereas a node in state OUT does not monitor any edge. Let V_{in} (resp. V_{out}) denote the set of nodes that are in state IN (resp. OUT) in a given configuration. Furthermore, the nodes hold a variable *neigh* for the set of its neighbors. A node is called *correct* if this variable indeed

holds the node's set of neighbors. This can be verified by the node's neighbors via a boolean expression. A node is *safe* if it itself and all its neighbors hold the correct set of neighbors in their *neigh* variable. Only a node itself can check whether it is safe or not. With the *neigh* variables each node x can determine the set $monEdges(x)$ of edges that it can monitor, i.e.

$$monEdges(x) = \{\langle y,z\rangle \mid y,z \in N(x) \wedge y \in z.neigh \wedge z \in y.neigh\}.$$

Let edge $e = \langle x,y\rangle \in E_S$. The following sets can be calculated by node x via accessing $y.neigh$: The set of nodes that *are able to* monitor edge $\langle x,y\rangle$ is denoted by $monCandidates(\langle x,y\rangle)$, that is to say:

$$monCandidates(\langle x,y\rangle) = \{N(x) \cap y.neigh\}.$$

Note that $monCandidates(\langle x,y\rangle)$ (and hence, all terms that use this set) may not be correct, if node x is not safe. The subset of nodes in $monCandidates(\langle x,y\rangle)$ that actually do monitor edge $\langle x,y\rangle$ is denoted by $monNodes(\langle x,y\rangle)$, i.e.

$$monNodes(\langle x,y\rangle) = \{z \in monCandidates(\langle x,y\rangle) \mid z.state = IN\}.$$

Correspondingly, via the *neigh* variable of node x, node y can calculate the sets $monCandidates(\langle y,x\rangle)$ and $monNodes(\langle y,x\rangle)$. The two sets $monCandidates(\langle x,y\rangle)$ and $monCandidates(\langle y,x\rangle)$ (resp. $monNodes(\langle x,y\rangle)$ and $monNodes(\langle y,x\rangle)$) denote the same sets if all variables and expressions are up-to-date, however, a distinction is necessary to clarify which node is enabled to calculate the set. If it is clear which node calculates the set, also the notation $monCandidates(e)$ and $monNodes(e)$ is used. The same notation is used in the analysis to denote the set of nodes that can monitor an edge e and its subset of nodes that currently monitor e.

The *administrator* of an edge is the adjacent node with smaller identifier, i.e. $admin(e) = \min\{x,y\}$. Apart from x and y each node in $N(x) \cap N(y)$, and hence any node that can monitor e, can determine its administrator. The administrator provides an expression $demand_e$ that denotes the difference between the desired number of monitoring nodes and the current number, i.e. $demand_e = desired(e) - |monNodes(e)|$. The set $adminEdges(x)$ contains all edges node x administrates. It cannot be determined by other nodes.

The first algorithm consists of three simple rules: A node has to set its *neigh* variable if it does not hold its set of neighbors. Note that a node can execute this rule at most once, and it cannot make any other move before all nodes in its closed neighborhood have the correct value for their *neigh* variable. Any node x in state OUT which observes that there are not enough nodes in state IN for an edge e contained in $monEdges(x)$, i.e. $admin(e).demand_e > 0$, enters state IN itself. Conversely, a node enters state OUT if all edges it can monitor are monitored by sufficient nodes even without itself, i.e. $\forall e \in monEdges(x) : admin(e).demand_e < 0$. The complete set of rules is given in Algorithm 6.1. The execution of rule In (resp. rule Out) is called an IN-*move* (resp. OUT-*move*).

Algorithm 6.1 Expression-Two Algorithm for Edge Monitoring

Expressions:
$correct :: (x.neigh = N(x))$
$demand_e :: desired(e) - |\{z \in N(x) \cap y.neigh \mid z.state = \mathsf{IN}\}|$
for all edges $e = \langle x, y \rangle \in adminEdges(x)$

Functions:
$adminEdges(x) = \{\langle x, y \rangle \mid y \in N(x) \wedge x < y\}$
$monEdges(x) = \{\langle y, z \rangle \mid y, z \in N(x) \wedge y \in z.neigh \wedge z \in y.neigh\}$
$admin(\langle y, z \rangle) = \min\{y, z\}$
$safe(x) = \forall y \in N[x] : y.correct$

Actions:
Neigh :: $[x.neigh \neq N(x)]$
$\longrightarrow \quad x.neigh := N(x)$

In :: $[safe(x) \wedge x.state = \mathsf{OUT} \wedge (\exists e \in monEdges(x) : admin(e).demand_e > 0)]$
$\longrightarrow \quad x.state := \mathsf{IN}$

Out :: $[safe(x) \wedge x.state = \mathsf{IN} \wedge (\forall e \in monEdges(x) : admin(e).demand_e < 0)]$
$\longrightarrow x.state := \mathsf{OUT}$

If there is an edge e that has a $desired(e)$ value larger than the number of nodes that are able to monitor e, an edge monitoring cannot be achieved since e cannot be

monitored sufficiently. The *monitoring gap*

$$gap = \sum_{\substack{e \in E_S, \\ demand_e > 0}} demand_e$$

is a measure for the quality of an edge-monitoring approximation.

Lemma 6.2.1. *If all nodes are disabled with respect to Algorithm 6.1, V_{in} constitutes a minimal edge monitoring of E_S. If G does not allow a sufficient edge monitoring of E_S, V_{in} constitutes an edge-monitoring approximation with minimal monitoring gap.*

Proof. Assume all nodes to be disabled with respect to Algorithm 6.1 and let $e \in E_S$. If $demand_e > 0$ and there is a node $x \in V_{out}$ that can monitor e then x is enabled to execute rule In. If there is a node $x \in V_{in}$ with $demand_e < 0$ for all edges $e \in monEdges(x)$ then x is enabled to execute rule Out. This contradiction proves the lemma. □

Lemma 6.2.2. *Let Algorithm 6.1 be executed under the central scheduler and the expression model. Assume that rule* Neigh *is not executed by any node. Then any node can make at most one* OUT*-move and this is its last move.*

Proof. Let $e \in E_S$ be an arbitrary edge. When a node x is enabled to leave state IN, all edges $e \in monEdges(x)$ are monitored by at least one node more than necessary according to the administrator *admin* of e. Note that *admin* may not be able to calculate the correct set $monNodes(e)$ due to an incorrect neighbor (cf. Figure 6.3). However, since the nodes do not execute rule Neigh, the administrator's perspective whether a neighbor is able to monitor an edge or not does not change in this scenario. Furthermore neither the incorrect neighbor nor any of its neighbors can execute an IN- or an OUT-move under these circumstances.

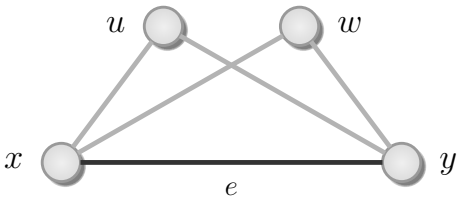


Figure 6.3: Let x be the administrator of e. If $w \notin y.neigh$ or $y \notin w.neigh$ then x cannot calculate the correct set $monNodes(e)$.

Under the central scheduler, only one monitor of e can leave state IN at any moment. Hence, after node x has left state IN all edges that were monitored by x are still indicated to be monitored sufficiently by its administrators. Since a node cannot leave state IN without all affected administrators' permission node x will never become enabled to enter state IN again. □

Corollary 6.2.1. *Assuming the expression model, Algorithm 6.1 stabilizes after at most $2n^2$ moves under the central scheduler.*

Proof. Rule Neigh can only be executed once per node (as the node's first move). The rest follows directly from Lemma 6.2.2 since a node cannot enter state IN twice without entering state OUT in between. This results in at most $n2n$ moves. □

The memory requirement of Algorithm 6.1 amounts to $O(\Delta \log n)$ per node: Apart from its neighbors' ids a node has to store its boolean state and at most Δ *demand* variables (there are at most m *demand* variables in the whole network, one for every monitored edge).

Let Algorithm 6.1^T denote the resulting algorithm if the transformation of [Tur12] is applied to Algorithm 6.1. From Theorem 3.2 of [Tur12] and Corollary 6.2.1 immediately follows:

Theorem 6.2.1. *Algorithm 6.1^T establishes an edge monitoring with minimal monitoring gap. It stabilizes after at most $O(mn^2)$ moves under the distributed scheduler in the distance-one model.*

6.2.3 Knowledge about Monitored Edges

Using Algorithm 6.1 a node x in state IN is expected to monitor *all* edges in $monEdges(x)$. This means that an edge may be monitored by more nodes than actually needed. Since checking a communication link is an active process it makes more sense to provide the nodes with information about which edges they actually *have* to monitor. The algorithm has to be changed slightly to meet this requirement: Instead of the boolean variable *state* each node x is equipped with a list $x.mon$ of edges which it is currently monitoring. In accordance, for an edge $e = \langle x, y \rangle$ the set $monNodes(\langle x, y \rangle)$ has to be redefined to contain the nodes that have edge e in their *mon* list, i.e. for node x $monNodes(\langle x, y \rangle) = \{z \in N(x) \cap y.neigh \mid \langle x, y \rangle \in z.mon\}$. Thus, the

expression to determine the demand of an edge has to be adapted to the new definition of $monNodes(e)$. Correspondingly, rule In and rule Out have to be adjusted: Instead of entering or leaving state *IN* the edge is added to or removed from the node's *mon* set. The complete set of rules is given in Algorithm 6.2.

Algorithm 6.2 Expression-Two Algorithm for Edge Monitoring with Monitoring Knowledge

Expressions:
$correct :: (x.neigh = N(x))$
$demand_e :: desired(e) - |\{z \in N(x) \cap y.neigh \mid e \in z.mon\}$
for all edges $e = \langle x, y \rangle \in adminEdges(x)$

Functions:
$adminEdges(x) = \{\langle x, y \rangle \mid y \in N(x) \wedge x < y\}$
$monEdges(x) = \{\langle y, z \rangle \mid y, z \in N(x) \wedge y \in z.neigh \wedge z \in y.neigh\}$
$admin(\langle y, z \rangle) = \min\{y, z\}$
$safe(x) = \forall y \in N[x] : y.correct$

Actions:
Neigh :: $[x.neigh \neq N(x)]$
$\longrightarrow$ $x.neigh := N(x)$

In :: $[safe(x) \wedge \exists e \in monEdges(x) : ((admin(e).demand_e > 0) \wedge (e \notin x.mon))]$
$\longrightarrow$ $\forall e \in monEdges(x) : ((admin(e).demand_e > 0) \wedge (e \notin x.mon))$
$x.mon := x.mon \cup \{e\}$

Out :: $[safe(x) \wedge \exists e \in x.mon : (admin(e).demand_e < 0)]$
$\longrightarrow$ $\forall e \in x.mon : (admin(e).demand_e < 0)$
$x.mon := x.mon \backslash \{e\}$

Lemma 6.2.1 and 6.2.2 also hold for Algorithm 6.2 under the central scheduler and the expression-two model. As in Algorithm 6.1, rule Out can be performed at most once per node and the same holds for rule In since all edges in $monEdges(x)$ that have positive demand are monitored by node x after the first execution of rule In and will never be excluded from $x.mon$ as long as none of the nodes execute rule Neigh. Hence, the following holds:

Corollary 6.2.2. *Algorithm 6.2 calculates an edge monitoring with minimal monitoring gap. Assuming a central scheduler and the expression-two model it terminates after at most* $2n^2$ *moves.*

The transformation of [Tur12] applied to Algorithm 6.2 results in Algorithm 6.2^T. The following theorem holds due to Theorem 3.2 of [Tur12] and 6.2.2:

Theorem 6.2.2. *Algorithm* 6.2^T *calculates an edge monitoring with minimal monitoring gap. Assuming the distance-one model and a distributed scheduler, Algorithm* 6.2^T *terminates after* $O(mn^2)$ *moves.*

6.3 Conclusion

This chapter presented the first two self-stabilizing algorithms for the edge-monitoring problem. While the first algorithm finds a minimal edge monitoring that assumes the selected nodes to monitor *all* edges they can monitor, the second algorithm is more suitable for real-world applications since no edge has a negative monitoring demand when the algorithm has stabilized. Both algorithms yield a result with minimal monitoring gap. The algorithms are designed using the expression model. Using the transformer of [Tur12] they stabilize after $O(mn^2)$ moves assuming the distance-one model and the distributed scheduler.

Chapter 7

Potential Function and Induction via Graph Reduction

This chapter uses a combination of a classical approach and a new technique to determine the complexity of a self-stabilizing algorithm for the maximum weight matching problem (see below): A complex potential function (cf. Section 3.2.3) is constructed to assess the algorithm's progress with respect to its convergence. An important contribution of this chapter is the new technique to compute the move complexity of self-stabilizing algorithms. The main idea is to map an execution sequence for a graph to that of a given subgraph. This mapping allows to derive an upper limit for the difference between the numbers of moves for both execution sequences - for the original graph and for the subgraph. Hence, the total number of moves required for the original graph is bound by the sum of this limit and the number of moves required for the subgraph. Since the subgraph has less edges or nodes, the latter number can be determined by induction.

The new graph reduction technique forms the basis for a new analysis of an algorithm by Manne and Mjelde that computes a 2-approximation for the maximum weight matching problem. In [MM07] they established that their algorithm stabilizes after $O(2^n)$ (resp. $O(3^n)$) moves under a central (resp. distributed) scheduler. The precise determination of the stabilization time of this algorithm remained an open problem. An example for which the algorithm requires an exponential number of moves was not provided. The use of the new technique improves these bounds considerably.

In particular it is shown that the algorithm stabilizes after $O(nm)$ moves under the central scheduler and that a modified version of the algorithm also stabilizes after $O(nm)$ moves under the distributed scheduler.

This chapter is organized as follows: The first section defines the problem and reviews the state of the art of distributed and self-stabilizing algorithms for matching problems. Section 7.2 presents the basic algorithm of Manne and Mjelde. Its worst-case complexity assuming a synchronous scheduler is analyzed in Section 7.3. The other schedulers are treated in the following sections. In Section 7.4 the specific potential function and the new proof method for the analysis of self-stabilizing algorithms is presented and applied to Manne and Mjelde's algorithm under the assumption of a central scheduler. Section 7.5 shows how the results of the preceding section can be carried over to work under a distributed scheduler. Section 7.6 concludes this chapter.

7.1 Example: Weighted Matching with Approximation Ratio 2

7.1.1 Introduction

The maximum weight matching problem is a fundamental problem in graph theory with a variety of important applications. Let $G = (V, E)$ be an undirected graph, with $n = |V|$ and $m = |E|$. A set M of independent edges of G is called a *matching* of G. M is a *maximal matching* if there is no matching M' with $M \subset M'$. A matching M is a *maximum matching* if there is no matching with cardinality larger than $|M|$. Let G be a weighted undirected graph. The *weight* of an edge e is denoted by $w(e) \in \mathbb{R}_+$. The weight of a matching M is the sum of the weights of all edges of M. A matching is called a *maximum weight matching* if its weight is the maximum among all matchings of G. Figure 7.1 shows a weighted graph and its maximum weight matching.

The algorithm considered in this chapter computes a *maximum weighted matching* of a graph with an approximation ratio of 2, i.e. the weight of the computed matching is at least half the weight of the maximum weight [MM07].

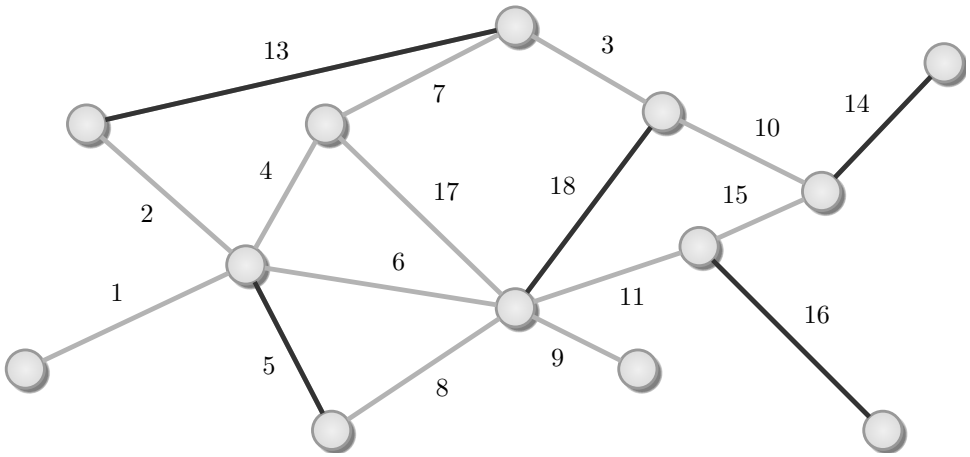


Figure 7.1: A maximum weight matching with weight 66.

7.1.2 Related Work

Algorithms solving the maximum matching problem received a lot of attention since the early work of Edmonds [Edm65]. This research has been carried out for bipartite and general graphs both in the weighted and unweighted setting. While there are many sequential algorithms, only a small number of distributed algorithms for matching have been proposed [WW04]. In fact, I am not aware of any distributed algorithm that solves the maximum matching problem optimally, except for special graph classes such as bipartite graphs. Therefore, research has concentrated on finding maximal matchings and on approximating maximum matchings, the weighted and the unweighted case. In the following, related work is classified into synchronous and asynchronous systems.

First, distributed algorithms to approximate maximum weighted matchings in synchronous systems are considered. Wattenhofer et al. present a randomized 5-approximation algorithm taking $O(\log n)$ rounds [WW04]. Nieberg's algorithm computes a $(1+\epsilon)$-approximation in $O(\log n)$ rounds [Nie08]. The unweighted maximum matching problem received considerably more attention. The currently best algorithms for finding approximately optimal matchings are due to Lotker et al. [LPSR09]. For any $\epsilon > 0$ they give a randomized distributed $(4+\epsilon)$-approximation algorithm for maximum weighted matching, whose running time is $O(\log n)$ and for unweighted dynamic graphs, they give a distributed algorithm that maintains a $(1+\epsilon)$-approximation in $O(1/\epsilon)$ time for each node insertion or deletion. The focus

of this thesis lies on asynchronous systems. The state of the art for synchronous systems is surveyed by Elkin [Elk04].

With respect to asynchronous systems at first the unweighted case is considered. Early work concentrated on the maximal matching problem. The history of self-stabilizing algorithms for the unweighted maximal matching problem goes back to Hsu and Huang [HH92]. Their algorithm assumes the shared memory model with composite atomicity and a central scheduler and requires $O(n^3)$ moves. Later the analysis of the algorithm was improved and lower bounds were proven $O(n^2)$ [Tel94], $O(m)$ [HJS02]. Note that the algorithm does not work with a distributed scheduler. Chattopadhyay et al. developed an algorithm that stabilizes in $O(n^2)$ steps for a fair distributed scheduler using the shared memory model with read/write atomicity [CHS02]. Later Manne et al. [MMPT09] presented an algorithm that stabilizes in $O(m)$ steps using an unfair distributed scheduler and the shared memory model with composite atomicity.

Whereas the algorithm of Hsu and Huang assumed an anonymous network, these two algorithms require node identifiers that are unique within distance 2. A deterministic self-stabilizing algorithm for the maximal matching problem in anonymous networks is impossible under a synchronous scheduler. For example, let G be a cyclic graph where all edges are of the same weight and the nodes do not have identifiers. If all nodes are in the same state initially, this property will hold after every step. Thus, when an algorithm stabilizes no node and no edge stands out. Chattopadhyay et al. presented a randomized algorithm for the maximal matching problem in anonymous network with read/write atomicity [CHS02]. Several methods for transforming algorithms using strong model assumptions to algorithms using weaker assumptions are described in the literature: From a fair scheduler to an unfair scheduler [Kar01], from a central to a distributed scheduler [GT07] and for atomicity refinement [NA02, CDP03]. In general, algorithms developed for a specific model are superior to transformed algorithms in terms of complexity.

Approximation of a maximum matching for the unweighted case also received some attention. Manne et al. presented the first self-stabilizing algorithm for finding a $3/2$-approximation to this problem using at most exponential time under a distributed adversarial scheduler [MMPT08]. This work is done for the shared memory model with composite atomicity.

Finally, the case of maximum weight matchings in asynchronous systems is considered. The work in this area is sparse. Manne and Mjelde developed the first self-stabilizing 2-approximation algorithm for the maximum weight matching problem [MM07]. The authors showed that their algorithm stabilizes after $O(2^n)$ (resp. $O(3^n)$) moves under a central (resp. distributed) scheduler. They assume unique identifiers and the shared memory model with composite atomicity. The following sections contribute a new analysis of Manne and Mjelde's algorithm and limit the number of moves to $O(nm)$ for the central scheduler. Furthermore, a modified version of this algorithm is presented, requiring $O(nm)$ moves for the unfair distributed scheduler. The survey of Guellati and Kheddouci [GK10] contains more references for self-stabilizing algorithms solving the matching problem (c.f. Section 2.4.6).

7.2 Algorithm Description

Manne and Mjelde's algorithm is based on the classical algorithm, shown in Algorithm 7.1. It is a sequential greedy algorithm that calculates a matching M_{greedy} with at least half the weight of the maximum weight [Pre99]. The idea is to start with an empty set and then add the remaining heaviest edges each time. M_{greedy} is unique if the edges' weights are pairwise different. The algorithm of Manne and Mjelde computes M_{greedy} [MM07].

Algorithm 7.1 Greedy 2-Approximation of Maximum Weight Matching

$M = \emptyset$
for $\langle u, v \rangle$ **in** E *in descending order with respect to their weight*
  if *neither* u *nor* v *are incident to an edge in* M
    then $M := M \cup \{\langle u, v \rangle\}$

Having unique node identifiers permits the assumption that all edge weights are different. This can be achieved by the following simple definition of a total order on the set of edges. Let $\langle u_1, u_2 \rangle$ and $\langle v_1, v_2 \rangle$ be two edges then $\langle u_1, u_2 \rangle < \langle v_1, v_2 \rangle$ if and only if

- $w(\langle u_1, u_2 \rangle) < w(\langle v_1, v_2 \rangle)$ or
- $w(\langle u_1, u_2 \rangle) = w(\langle v_1, v_2 \rangle) \wedge \min(u_1, u_2) < \min(v_1, v_2)$ or

- $w(\langle u_1, u_2\rangle) = w(\langle v_1, v_2\rangle) \wedge \min(u_1, u_2) = \min(v_1, v_2) \wedge$
  $\max(u_1, u_2) < \max(v_1, v_2)$.

For the rest of this chapter it is assumed that the weights of all edges are pairwise different.

In the following the self-stabilizing algorithm of Manne and Mjelde is presented with a slightly different notation. The state $s_v = (v.p, v.w)$ of a node v is defined by two variables p and w. The intention of these variables is as follows: p stores a pointer (i.e. the identifier) to a neighbor of v or $null$, and w stores the weight of the edge $\langle v, v.p\rangle$, i.e., $w(\langle v, v.p\rangle)$. The definition of $w(\langle\cdot,\cdot\rangle)$ is extended such that $w(\langle v, null\rangle) = 0$. To express that $v.p = u$, it is said that *node v points to node u* or synonymously *node v points to edge* $\langle v, u\rangle$.

Let $C_v = \{v_i \in N(v) \mid w(\langle v_i, v\rangle) \geq v_i.w \vee v_i.p \in \{v, null\}\}$. A node $v_{max} \in C_v$ is called *maximal* if $w(\langle v_{max}, v\rangle) \geq w(\langle v_i, v\rangle)\, \forall v_i \in C_v$. If $C_v \neq \emptyset$ then denote by $maxC_v$ the unique maximal node of C_v. The complete algorithm from [MM07] is depicted below. Note that the definition of C_v has been slightly altered compared to the original paper. It is straightforward to see that the results of [MM07] still hold and the calculated matching is the same.

Algorithm 7.2 Self-Stabilizing 2-Approximation Maximum Weight Matching

Functions:

$BestMatch(v)$:
  if $C_v \neq \emptyset$ **then return** $maxC_v$
  else return $null$

Actions:

$\mathsf{R}_1 :: [v.p \neq BestMatch(v) \vee v.w \neq w(\langle v, v.p\rangle)]$
  $\longrightarrow v.p := BestMatch(v)$
    $v.w := w(\langle v, v.p\rangle)$

Two nodes are called *matched*, if they both point at each other. An edge $\langle v, w\rangle$ is matched, if v and w are matched. Denote by M the set of all matched edges of a configuration. A node v is called in *sync*, if $v.w = w(\langle v, v.p\rangle)$. In [MM07] it is proven that Algorithm 7.2 stabilizes under a distributed scheduler with at most $O(3^n)$ moves and that in a configuration of G, where no node is enabled, M is a

2-approximation of the maximum weight matching of G. The following sections prove that the move complexity of Algorithm 7.2 is in fact polynomial.

A configuration c satisfies $\mathcal{P}$ if all nodes are in sync, all nodes not contributing to the matching point to $null$ and the matching defined by c is M_{greedy}.

Lemma 7.2.1. *There is a unique configuration in which all nodes are disabled with respect to Algorithm 7.2 and this configuration satisfies $\mathcal{P}$.*

Proof. Consider a configuration c where all nodes are disabled with respect to Algorithm 7.2. Since rule R_1 is not enabled, all nodes are in sync. Consider a node v with $v.p = u$ but $u.p \neq v$. If $u.p = null$ or $w(\langle u, u.p\rangle) < w(\langle v, u\rangle)$ then u is enabled to point to v. Hence, $w(\langle u, u.p\rangle) > w(\langle v, u\rangle)$ and thus, v is enabled. This contradiction shows that all nodes not contributing to the matching point to $null$ and c defines a matching M_c.

Assume $M_c \neq M_{greedy}$. Let $e_1 = \langle u_1, v_1\rangle$ be the heaviest edge with $e_1 \in M_{greedy}$ and $e_1 \notin M_c$. Then, in configuration c node u_1 and v_1 do not point towards each other but also they do not point to heavier edges, since there are no heavier edges leading towards nodes that are not matched via even heavier edges already. Therefore u_1 and v_1 are enabled, contradicting the assumption. □

A disadvantage of the presented algorithms must be mentioned. A configuration that represents a maximum weight matching is not necessarily a legitimate configuration, i.e. it does not necessarily satisfy $\mathcal{P}$. For this purpose consider a graph consisting of the nodes v_0, v_1, v_2, and v_3 and the three edges $\langle v_0, v_1\rangle$, $\langle v_1, v_2\rangle$ and $\langle v_2, v_3\rangle$ with weights $1, 1+\epsilon$, and 1 where $\epsilon > 0$, as shown in Figure 7.2. A maximum weight matching consists of the edges $\langle v_0, v_1\rangle$ and $\langle v_2, v_3\rangle$ with a total weight of 2. Even if initialized with this matching, Algorithm 7.2 will stabilize with the non-maximum weight matching consisting of the edge $\langle v_1, v_2\rangle$ with a total weight of $1+\epsilon$.

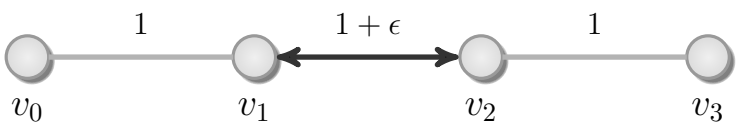


Figure 7.2: Disadvantage of Algorithms 7.1 and 7.2: They choose edge $\langle v_1, v_2\rangle$ instead of $\langle v_0, v_1\rangle$ and $\langle v_2, v_3\rangle$.

7.3 Synchronous Scheduler

An upper bound of $O(n^2)$ moves until stabilization for Algorithm 7.2 executed under a synchronous scheduler directly follows from Theorem 3 of [MM07]: This theorem proves that assuming a fair scheduler Algorithm 7.2 converges after at most $2|M|+1$ rounds, where M is the final matching found by the algorithm. The proof analyzes the local states of the nodes and shows that after each second round the heaviest edge possible (with respect to the matching under construction) is irreversibly added to M (cf. Algorithm 7.1). Both adjacent nodes will not make a move afterwards. The result follows since a matching cannot contain more than $n/2$ edges and under the synchronous scheduler any node can make at most one move per round.

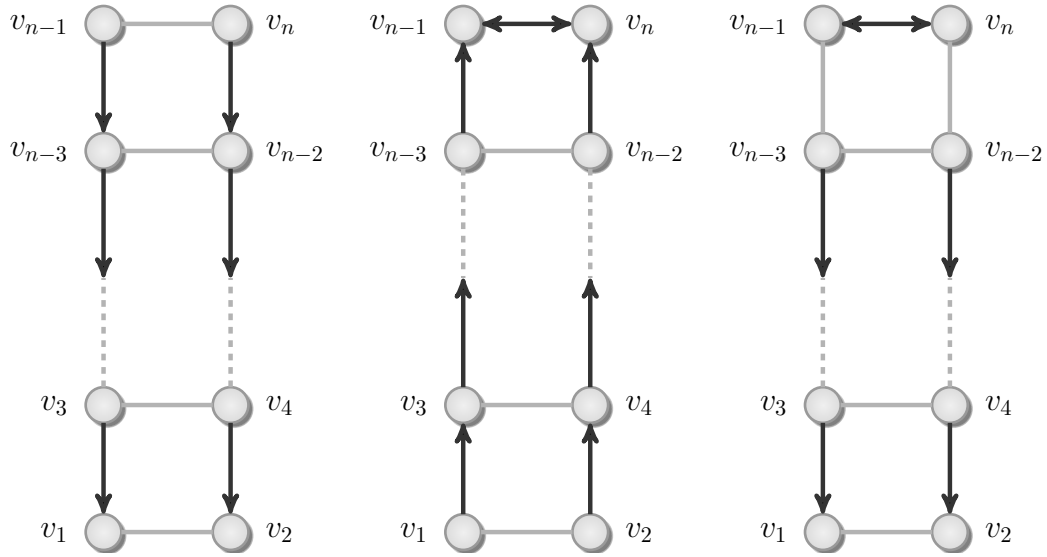
This section provides an example that shows that $O(n^2)$ is also a lower bound. For this purpose consider the *ladder graph* L with n nodes: Let n be even and $\langle v_i, v_j \rangle \in E$ if $j = i+2$, or $j = i+1$ and i is odd. The weights of the edges satisfy the relation $w(\langle v_{i_1}, v_{j_1} \rangle) < w(\langle v_{i_2}, v_{j_2} \rangle)$ if

- $min(i_1, j_1) < min(i_2, j_2)$, or
- $min(i_1, j_1) = min(i_2, j_2) \wedge max(i_1, j_1) < max(i_2, j_2)$.

Figure 7.3 gives an initial configuration and shows the first two steps of an execution of Algorithm 7.2. The nodes' pointers are indicated by arrows. In the final configuration the matching consists of all edges $\langle v_i, v_j \rangle$, where i is odd and $j = i+1$. This configuration is reached after $n^2/2$ moves.

7.4 Central Scheduler

To analyze the complexity of Algorithm 7.2 under the central scheduler, two methods are used: The following section introduces a potential function Φ for this algorithm that measures its "progress". This function has two important properties: It is monotonically increasing and it has an upper limit. Section 7.4.2 presents a mapping of an execution sequence of Algorithm 7.2 on a given graph to a closely-related execution sequence on a certain subgraph. Via this mapping an upper bound for the difference between the number of moves for both sequences can be determined by showing that the potential function must be increased in some situations. Given this result, the move complexity of the algorithm can be derived by induction.



(a) Start configuration (b) After the first round (c) After the second round

Figure 7.3: Start of the execution of Algorithm 7.2 under the synchronous scheduler for graph L

7.4.1 Potential Function

Let the function $\Phi : C^G \longrightarrow \mathbb{R}_+$ be defined as the sum of the weights of all edges $\langle x, x.p \rangle$ that meet the following conditions:

- x is disabled with respect to Algorithm 7.2, and
- if $x.p$ is enabled, then its move will not enable x.

In other words: An edge contributes to Φ if and only if either both nodes are disabled and they are pointing to each other or they are exactly one move away from becoming so. An edge $\langle x, x.p \rangle$ contributes to Φ only once, even in case $x.p$ points to x itself and the rules above also apply to the edge $\langle x.p, x \rangle$. Edges that contribute to the value of function Φ will be called Φ-*edges*. This status of an edge $\langle v, w \rangle$ with respect to being a Φ-edge can only change after a move of a neighbor of v or w.

Figure 7.4 illustrates the concept of Φ-edges. The nodes' pointers are indicated by arrows, the values of their weight variables are in brackets. Edge $\langle a, c \rangle$ is a Φ-edge, because a points to c and is disabled while c is enabled to point to a; g and h are matched and disabled and thus, $\langle g, h \rangle$ also contributes to Φ. Nodes d and e are both

disabled and point towards f. However, f is enabled to point to e, so $\langle d, f \rangle$ is not a Φ-edge, but $\langle e, f \rangle$ is.

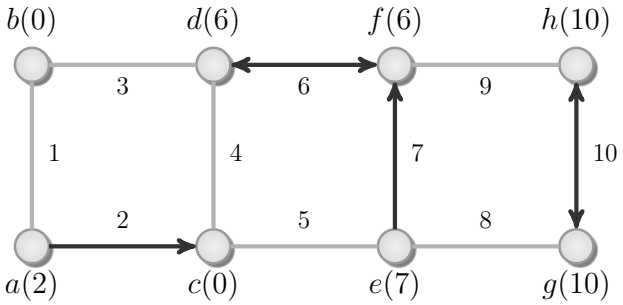


Figure 7.4: Illustration of the potential function: Edges $\langle a, c \rangle$, $\langle e, f \rangle$ and $\langle g, h \rangle$ are Φ-edges.

Lemma 7.4.1. *Φ is monotonically increasing and so is the number of Φ-edges.*

Proof. The value of Φ may change due to the nodes' moves. The impact of a node's move on Φ is analyzed one by one. Let x be any node that is enabled with respect to Algorithm 7.2. By executing its move node x changes its pointer from $x.p_{old}$ to $x.p_{new}$.

If $x.p_{old} = x.p_{new}$ only the weight variable of x changed but its pointer did not. Any neighbor $y \neq x.p_{new}$ of x with $y.p = x$ gets enabled and thus, $\langle y, x \rangle$ did not contribute to Φ before. If $x.p_{new}$ is enabled to point to x after that move, $\langle x, x.p_{new} \rangle$ is a new Φ-edge. So let $x.p_{old} \neq x.p_{new}$. There are two distinct cases that are to be applied after the move of x:

Case 1: $x.p_{new} = null$

Φ remains unchanged, since x was enabled before and all its neighbors are pointing to heavier edges than $\langle x, x.p_{old} \rangle$, otherwise x would not have been enabled to perform this move. None of the neighbors gets enabled or disabled by x's move.

Case 2a: $x.p_{new} \neq null$ and $x.p_{new}$ is disabled.

Clearly $x.p_{new}$ was pointing to x and $\langle x.p_{new}, x \rangle$ was a Φ-edge before x's move. Hence, even though $\langle x, x.p_{new} \rangle$ meets all conditions to be a Φ-edge it does not increase the Φ-value. Nodes that are pointing towards x have a lower weight than $w(\langle x.p_{new}, x \rangle)$. So they did not contribute to Φ anyway. However, Φ can increase, if, due to the move of x, a neighbor $y \in N(x)$, that also pointed to x before (e.g.

$y = x.p_{old}$ or $x.p_{old} = null$), becomes enabled to point to a disabled node z that on its part already points to y.

Case 2b: $x.p_{new} \neq null$, $x.p_{new}$ is enabled, and its move would not enable x. Note that $\langle x, x.p_{new} \rangle$ is a Φ-edge. The goal is to show that the weight of this edge is heavier than the weight of an edge that previously was a Φ-edge but lost this status due to the move of x. Two cases are distinguished:

- $w(\langle x, x.p_{new} \rangle) > w(\langle x, x.p_{old} \rangle)$
  If there have been other nodes pointing at x resp. $x.p_{new}$ before x's move, their edges would have been of lower weight than $\langle x, x.p_{new} \rangle$ and they would not have been taken into account for Φ, since the move of x resp. $x.p_{new}$ enables them. Other nodes do not become enabled. Thus, no Φ-edge loses this status. Since $\langle x, x.p_{new} \rangle$ is a new Φ-edge, the value of Φ increases.

- $w(\langle x, x.p_{new} \rangle) < w(\langle x, x.p_{old} \rangle)$
  $x.p_{old}$ was pointing to another node with higher weight and so $w(\langle x, x.p_{old} \rangle)$ did not account for Φ. Now x is the node that points to $x.p_{new}$ with heaviest weight. If there have been other nodes pointing at x before x's move, their edges would have been of lower weight than $\langle x, x.p_{new} \rangle$ and they would not have been taken into account for Φ, since the move of x enables them.

  If there are other nodes pointing at $x.p_{new}$, one of its edges could have been a Φ-edge before the move. Due to the move of x this edge loses this status, since $x.p_{new}$ is enabled to point to x. However, the value of Φ increases, because the new Φ-edge, $\langle x, x.p_{new} \rangle$ has a higher weight than the former Φ-edge.

Case 2c: $x.p_{new} \neq null$, $x.p_{new}$ is enabled, and its move would enable x. Note that $\langle x, x.p_{new} \rangle$ is not a Φ-edge. $x.p_{new}$ was enabled to point to an edge of weight greater than $\langle x, x.p_{new} \rangle$ before the move by x. So $x.p_{new}$ is not affected by the move of x nor are the neighbors of $x.p_{new}$. x obviously was enabled before its move and so are all nodes that are pointing towards x. So their weight did not contribute to Φ before and hence Φ does not decrease.

So, there is no move that decreases the value of Φ. The second statement also follows from the preceding proof, since there is no move that decreases the number of Φ-edges. □

Lemma 7.4.2. *At any time the number of* Φ*-edges is at most* $n/2$.

Proof. Let e_1, e_2 be two incident edges. Without loss of generality $w(e_1) > w(e_2)$. Assume, both, $e_1 = \langle v_1, v_2 \rangle$ and $e_2 = \langle v_2, v_3 \rangle$ contribute to function Φ. If v_2 points to neither v_1 nor to v_3 then both v_1 and v_3 have to point to v_2, and the move of v_2 would enable v_3, contradicting the assumption. If $v_2.p = v_1$ then e_2 can only be a Φ-edge if v_1 points to a heavier edge than e_1. So v_2 is enabled and v_1 does not point to v_2. Therefore e_1 is not a Φ-edge. If $v_2.p = v_3$ than v_2 is enabled to point to v_1 unless v_1 itself points to a heavier edge. In the latter case there is no node pointing towards e_1 and therefore it cannot be a Φ-edge. If v_2 is enabled to point to v_1, e_2 cannot be a Φ-edge. Therefore the set of Φ-edges always forms a matching of the underlying graph. □

7.4.2 Graph Reduction and Induction

This section proves that Algorithm 7.2 stabilizes in $O(nm)$ steps under the central scheduler. For this purpose the graph G' that is obtained by removing the lightest edge from G is examined. It is shown that any execution of Algorithm 7.2 for G can be mapped to a closely related valid execution for G'. In particular a bound for the number of additional moves for G in comparison to G' is established. This allows to leverage induction on the number of edges. More precisely, it is proven that certain sequences of moves on the lightest edge increase the number of Φ-edges. Furthermore, it is shown that particular moves can only occur in the context of these sequences. Then via Lemma 7.4.1 and 7.4.2 an upper bound for the number of moves can be derived.

For the upcoming analysis the formal model for self-stabilizing algorithms, as described in Section 2.2.3, is enhanced by the symbol $\perp$, which denotes the *empty move*. This move does not change the state of any node. Every node is at any time enabled with respect to the empty move. It is not part of the algorithm under consideration, it is a convenience for the proof. Furthermore, the wildcard symbol $*$ will be used to denote an arbitrary value, e.g. if it is irrelevant whether a node's weight variable is in sync with its pointer.

Let $\langle a, b \rangle$ be the lightest edge of G and without loss of generality $a < b$ with respect to the fixed order of the nodes. Let $G' := G \setminus \{\langle a, b \rangle\}$. Let π_c define a transformation that converts a configuration of C^G into a configuration of $C^{G'}$, i.e. the states of the

nodes a and b are changed to ensure they are not pointing towards each other and they do not store the weight of edge $\langle a, b \rangle$.

$$\pi_c : \begin{cases} C^G \longrightarrow C^{G'}, \\ (s_{v_1}, \ldots, s_{v_n}) \mapsto (s'_{v_1}, \ldots, s'_{v_n}), \text{ where} \end{cases}$$

$$s'_{v_i} = s_{v_i}, \text{ if } v_i \notin \{a, b\}$$

$$s'_a = \begin{cases} s_a, & \text{if } s_a.p \neq b \\ (null, 0), & \text{if } s_a.p = b \wedge s_a.w = w(\langle a, b \rangle) \\ (null, s_a.w), & \text{if } s_a.p = b \wedge s_a.w \neq w(\langle a, b \rangle) \end{cases}$$

$$s'_b = \begin{cases} s_b, & \text{if } s_b.p \neq a \\ (null, 0), & \text{if } s_b.p = a \wedge s_b.w = w(\langle b, a \rangle) \\ (null, s_b.w), & \text{if } s_b.p = a \wedge s_b.w \neq w(\langle b, a \rangle) \end{cases}$$

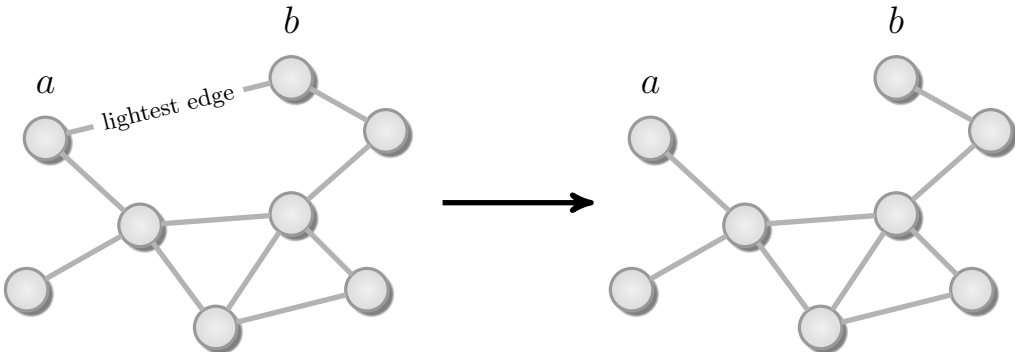


Figure 7.5: Graph G and the reduced graph G'. Some valid moves on G, e.g. node a setting its pointer to node b, are no longer possible on G'.

There are four types of moves that may appear in an execution of Algorithm 7.2 for G, which cannot be executed for G' (cf. Figure 7.5, Table 7.1). Therefore, a mapping from the moves that are executable in G to the moves executable in G' is defined. To facilitate readability the following shorthand notation is used throughout the rest of the chapter. For $x \in \{a, b\}$ let

$$\perp_0^x = \begin{cases} \perp, & \text{if } s_x.w = w(\langle x, s_x.p \rangle) \\ ((null, *)_x, (null, 0)_x), & \text{if } s_x.w \neq w(\langle x, s_x.p \rangle) \end{cases}$$

Note that a node cannot be out of sync after its first move, so with the possible exception of the node's very first move $\perp_0^x = \perp$.

Denote by $\mathcal{M}^G$ the set of all moves of the algorithm on graph G. Let $\mathcal{M}_\perp^{G'} := \mathcal{M}^{G'} \cup \{\perp\}$. Every move of $\mathcal{M}^G$ is mapped to a move in $\mathcal{M}_\perp^{G'}$. The only moves that will be changed are related to edge $\langle a, b \rangle$. Move m_{ix} represents the move of type $i \in \{1, .., 4\}$ performed by node $x \in \{a, b\}$. It is mapped to move m'_{ix}. Table 7.1 exemplifies the respective moves of node a. A formal definition of the four types of moves is given in Table 7.2.

Move in $\mathcal{M}^G$		**Move in $\mathcal{M}_\perp^{G'}$**	
Name	Move	Name	Move
m_{1a}	a pointed to *null* before, then points to b	m'_{1a}	(empty move)
m_{2a}	a pointed to any node before, then points to b	m'_{2a}	a sets its pointer to *null*
m_{3a}	a pointed to b before, then points to *null*	m'_{3a}	(empty move)
m_{4a}	a pointed to b before, then points to any other node	m'_{4a}	a pointed to *null* before, then points to any other node

Table 7.1: Informal description of the moves in G and corresponding moves in G' using the example of node a.

All other moves remain unchanged. Note that this does not only include the moves by nodes other than a and b. Furthermore moves such as node a setting its pointer from a node $x \neq b$ towards a node $y \neq b$ will not be altered. So formally the following mapping is defined:

$$\pi_{\mathsf{m}} : \begin{cases} \mathcal{M}^G \longrightarrow \mathcal{M}_\perp^{G'}, \\ \mathsf{m} \mapsto \mathsf{m}', \text{ where} \end{cases}$$

$$\mathsf{m}' = \begin{cases} \mathsf{m}'_{ix}, & \text{if } \mathsf{m} = \mathsf{m}_{ix}, \text{ for } x \in \{a, b\} \wedge i \in \{1, \ldots, 4\} \\ \mathsf{m}, & \text{in all other cases.} \end{cases}$$

The moves m_{4a} and m_{4b} will receive a special treatment, the moves referred to as the *list moves* are those of type 1, 2 and 3 only.

Move in $\mathcal{M}^G$		Move in $\mathcal{M}^{G'}_{\perp}$	
Name	Move	Name	Move
m_{1a}	$((null, *), (b, w(\langle a, b\rangle)))$	m'_{1a}	$\perp^a_0$
m_{1b}	$((null, *), (a, w(\langle a, b\rangle)))$	m'_{1b}	$\perp^b_0$
m_{2a}	$((y \neq null, *), (b, w(\langle a, b\rangle)))$	m'_{2a}	$((*, *), (null, 0))$
m_{2b}	$((y \neq null, *), (a, w(\langle a, b\rangle)))$	m'_{2b}	$((*, *), (null, 0))$
m_{3a}	$((b, *), (null, 0))$	m'_{3a}	$\perp^a_0$
m_{3b}	$((a, *), (null, 0))$	m'_{3b}	$\perp^b_0$
m_{4a}	$((b, *), (y \neq b, w(\langle a, y\rangle)))$	m'_{4a}	$((null, *), (y, w(\langle a, y\rangle)))$
m_{4b}	$((a, *), (y \neq a, w(\langle b, y\rangle)))$	m'_{4b}	$((null, *), (y, w(\langle b, y\rangle)))$

Table 7.2: Formal definition of the moves in G and corresponding moves in G'

Lemma 7.4.3. *If a node v in configuration c of G is enabled to perform move m, then v is enabled to perform move $\pi_m(\mathsf{m})$ in configuration $\pi_c(c)$ of G'.*

Proof. There are several distinguishable possibilities for move m. Let $\mathsf{m}' = \pi_{\mathsf{m}}(\mathsf{m})$ and $c' = \pi_c(c)$.

Case 1: $\mathsf{m} = \mathsf{m}_{2a}$: All neighbors of a point to other nodes via heavier edges, so a has no edge left to point to but $\langle a, b\rangle$. π_c does not change anything about that, but G' does not contain edge $\langle a, b\rangle$, so a is enabled to set its pointer to $null$ instead.

Case 2: $\mathsf{m} \in \{\mathsf{m}_{1a}, \mathsf{m}_{3a}\}$: $\mathsf{m}' = \perp^a_0$. Every node is enabled at any time with respect to the empty move, and a node that is not in sync is enabled as well.

Case 3: $\mathsf{m} = \mathsf{m}_{4a}$: In configuration c move m_{4a} was enabled, i.e. a points to b and wants to turn its pointer towards a node x via a heavier edge. π_c sets $a.p$ to $null$. From a's point of view this does not change anything else, so move m'_{4a} is enabled.

Case 4: m is a move that is not contained in the list above: $\mathsf{m}' = \mathsf{m}$. π_c does not affect moves that are not related to edge $\langle a, b\rangle$.

The corresponding moves of node b can be treated alike. □

Lemma 7.4.4. *Let c_0 be a configuration of G and m a move enabled in this configuration with respect to Algorithm 7.2. Then $\pi_c(\mathsf{m}(c_0)) = \mathsf{m}'(\pi_c(c_0))$, where $\mathsf{m}' = \pi_m(\mathsf{m})$. In other words the diagram shown in Fig 7.6 is commutative.*

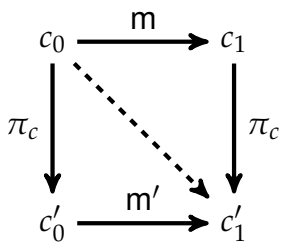


Figure 7.6: Visualization of Lemma 7.4.4: Commutativity of the order of move and state transformation.

Proof. Four cases have to be considered :

1. Node $x \notin \{a, b\}$ performs move m, thus $\mathsf{m}' = \mathsf{m}$:

$$\begin{aligned}
\pi_c(\mathsf{m}(c_0)) &= \pi_c(\mathsf{m}((s_{v_1}, \ldots, s_a, \ldots, s_b, \ldots, s_x, \ldots, s_{v_n}))) \\
&= \pi_c((s_{v_1}, \ldots, s_a, \ldots, s_b, \ldots, \mathsf{m}(s_x), \ldots, s_{v_n})) \\
&= (s_{v_1}, \ldots, s'_a, \ldots, s'_b, \ldots, \mathsf{m}(s_x), \ldots, s_{v_n}) \\
&= (s_{v_1}, \ldots, s'_a, \ldots, s'_b, \ldots, \mathsf{m}'(s_x), \ldots, s_{v_n}) \\
&= \mathsf{m}'((s_{v_1}, \ldots, s_a, \ldots, s_b, \ldots, s_x, \ldots, s_{v_n})) \\
&= \mathsf{m}'(\pi_c(c_0))
\end{aligned}$$

Without loss of generality, for the rest of this proof assume node a to be the node that performs move m.

2. $\mathsf{m} \in \{\mathsf{m}_{1a}, \mathsf{m}_{3a}\}$, thus $\mathsf{m}' = \bot_0^a$. So a either points to *null* after its move, which will not be changed by the application of π_c, or it points to b; in this case it will point to *null* after the application of π_c:

$$\begin{aligned}
\pi_c(\mathsf{m}(c_0)) &= \pi_c(\mathsf{m}((s_{v_1}, \ldots, s_a, \ldots, s_b, \ldots, s_{v_n}))) \\
&= \pi_c((s_{v_1}, \ldots, \mathsf{m}(s_a), \ldots, s_b, \ldots, s_{v_n})) \\
&= (s_{v_1}, \ldots, \pi_c(\mathsf{m}(s_a)), \ldots, \pi_c(s_b), \ldots, s_{v_n}) \\
&= (s_{v_1}, \ldots, \pi_c(\mathsf{m}(s_a.p, s_a.w)), \ldots, s'_b, \ldots, s_{v_n}) \\
&= (s_{v_1}, \ldots, (null, 0)_a, \ldots, s'_b, \ldots, s_{v_n})
\end{aligned}$$

If node a is enabled to perform the moves m_{1a} or m_{3a} in configuration c_0, it must be pointing to *null* or b before its move. Hence $\pi_c(s_a) = (null, 0)$, if a had its weight and its pointer in sync.

$$\begin{aligned}
\mathsf{m}'(\pi_c(c_0)) &= \bot_0^a((s_{v_1}, \ldots, \pi_c(s_a), \ldots, \pi_c(s_b), \ldots, s_{v_n})) \\
&= \bot_0^a((s_{v_1}, \ldots, (null, 0)_a, \ldots, s'_b, \ldots, s_{v_n})) \\
&= (s_{v_1}, \ldots, \bot_0^a((null, 0)_a), \ldots, s'_b, \ldots, s_{v_n}) \\
&= (s_{v_1}, \ldots, (null, 0)_a, \ldots, s'_b, \ldots, s_{v_n})
\end{aligned}$$

If a did not perform a move before, then this node may be out of sync. In this case $\pi_c(s_a) = (null, s_a.w)$ and move m' is not the empty move but it sets the weight of a to 0.

$$\begin{aligned}
\mathsf{m}'(\pi_c(c_0)) &= \bot_0^a((s_{v_1}, \ldots, \pi_c(s_a), \ldots, \pi_c(s_b), \ldots, s_{v_n})) \\
&= \bot_0^a((s_{v_1}, \ldots, (null, s_a.w)_a, \ldots, s'_b, \ldots, s_{v_n})) \\
&= (s_{v_1}, \ldots, \bot_0^a((null, s_a.w)_a), \ldots, s'_b, \ldots, s_{v_n}) \\
&= (s_{v_1}, \ldots, (null, 0)_a, \ldots, s'_b, \ldots, s_{v_n})
\end{aligned}$$

3. $\mathsf{m} = \mathsf{m}_{4a}$:

$$\begin{aligned}
\pi_c(\mathsf{m}(c_0)) &= \pi_c(\mathsf{m}((s_{v_1}, \ldots, (b, *)_a, \ldots, s_b, \ldots, s_{v_n}))) \\
&= \pi_c((s_{v_1}, \ldots, \mathsf{m}((b, *)_a), \ldots, s_b, \ldots, s_{v_n})) \\
&= (s_{v_1}, \ldots, \pi_c(\mathsf{m}((b, *)_a)), \ldots, \pi_c(s_b), \ldots, s_{v_n}) \\
&= (s_{v_1}, \ldots, \pi_c((x \neq b, w(\langle a, x \rangle)_a)), \ldots, s'_b, \ldots s_{v_n}) \\
&= (s_{v_1}, \ldots, (x \neq b, w(\langle a, x \rangle)_a, \ldots, s'_b, \ldots, s_{v_n})
\end{aligned}$$

and

$$\begin{aligned}
\mathsf{m}'(\pi_c(c_0)) &= \mathsf{m}'((s_{v_1}, \ldots, \pi_c((b, *)_a), \ldots, \pi_c(s_b), \ldots, s_{v_n})) \\
&= \mathsf{m}'((s_{v_1}, \ldots, (null, 0)_a, \ldots, s'_b, \ldots, s_{v_n})) \\
&= (s_{v_1}, \ldots, \mathsf{m}'((null, 0)_a), \ldots, s'_b, \ldots, s_{v_n}) \\
&= (s_{v_1}, \ldots, (x \neq b, w(\langle a, x \rangle)_a, \ldots, s'_b, \ldots, s_{v_n})
\end{aligned}$$

4. At last, $\mathsf{m} = \mathsf{m}_{2a}$.

$$\begin{aligned}
\pi_c(\mathsf{m}(c_0)) &= \pi_c(\mathsf{m}((s_{v_1}, \ldots, s_a, \ldots, s_b, \ldots, s_{v_n}))) \\
&= \pi_c((s_{v_1}, \ldots, \mathsf{m}(s_a), \ldots, s_b, \ldots, s_{v_n})) \\
&= (s_{v_1}, \ldots, \pi_c(\mathsf{m}(s_a)), \ldots, \pi_c(s_b), \ldots, s_{v_n}) \\
&= (s_{v_1}, \ldots, \pi_c(\mathsf{m}(s_a.p, s_a.w)), \ldots, s'_b, \ldots, s_{v_n}) \\
&= (s_{v_1}, \ldots, \pi_c((b, w(\langle a, b\rangle)_a), \ldots, s'_b, \ldots, s_{v_n}) \\
&= (s_{v_1}, \ldots, (null, 0)_a, \ldots, s'_b, \ldots, s_{v_n})
\end{aligned}$$

If the configuration is transformed first it should be pointed out that a is enabled to perform move m_{2a} in configuration c_0 if it points to a node other than b before. Hence, π_c does not affect s_a and m' is the move that lets a point to $null$. This yields:

$$\begin{aligned}
\mathsf{m}'(\pi_c(c_0)) &= \mathsf{m}'((s_{v_1}, \ldots, \pi_c(s_a), \ldots, \pi_c(s_b), \ldots, s_{v_n})) \\
&= \mathsf{m}'((s_{v_1}, \ldots, s_a, \ldots, s'_b, \ldots, s_{v_n})) \\
&= (s_{v_1}, \ldots, \mathsf{m}'(s_a), \ldots, s'_b, \ldots, s_{v_n}) \\
&= (s_{v_1}, \ldots, (null, 0)_a, \ldots, s'_b, \ldots, s_{v_n})
\end{aligned}$$

The same arguments hold for node b executing move m. Thus, in all cases $\pi_c(\mathsf{m}(c_0)) = \mathsf{m}'(\pi_c(c_0))$. □

Lemma 7.4.5. *If c is a legitimate configuration of Algorithm 7.2 for G, then $\pi_c(c)$ is a legitimate configuration for G'.*

Proof. Let c be a legitimate configuration for G. π_c cannot alter the state of any nodes but a and b. Furthermore, these nodes are only changed if one of them points to the other. Let x be a node of G. Three cases are considered:

Case 1: $x.p = y$ and $\{x, y\} \neq \{a, b\}$.

Since c is legitimate, y points at x. None of them gets enabled by applying π_c.

Case 2: $x.p = null$.

x cannot be a neighbor of a, unless a points at a heavier edge. In none of these cases one of the involved nodes gets activated by π_c. The same holds for the neighbors of b.

Case 3: $x \in \{a, b\}$, a and b are pointing at each other.
π_c sets both nodes' pointers to $null$. Since $\langle a, b\rangle$ is not contained in G', a and b cannot point to each other. If, without loss of generality, a is enabled in configuration $\pi_c(c)$, there must be a neighbor $z \in N(a)$ in G' that points to $null$ or to a. Since z remains unaffected by π_c, it must have been pointing towards a in c as well. Thus, a was enabled in c in contradiction to the assumption. □

Hence, from an execution for G it is possible to derive an execution for G' that differs from the original execution only in the list moves. Both executions result in legitimate configurations that are related via π_c. For G the algorithm will need at most as many additional steps as there are moves replaced by move $\perp$ in G'. The latter only applies to moves of type 1 and 3.

Let $\#(\mathsf{m}_{ix})$ denote the number of executed moves of type i by node x. Besides, let $\#(G)$ denote the number of executed moves of a given execution for graph G. This yields:

$$\#(G) \leq \#(G') + \#(\mathsf{m}_{1a}) + \#(\mathsf{m}_{1b}) + \#(\mathsf{m}_{3a}) + \#(\mathsf{m}_{3b})$$

To analyze how often the moves in question can be executed, it is shown that these moves increase the number of Φ-edges in certain situations. Lemma 7.4.2 limits the number of such edges to at most $n/2$. Therefore the executions of these moves can be bounded.

At first, it should be noted that nodes a and b cannot point to $null$ at the same time, except for the initial configuration. As soon as one of them executes a move, this situation will not occur again. a and b cannot perform any of the moves of the list as long as both of them point to other nodes (the edges to these nodes are heavier). For instance, move m_{2b} cannot be executed after m_{2a} without having another move of b in between that makes it point to a heavier edge first.

The list moves cannot be executed in arbitrary order. For example, move m_{3a} cannot be executed twice, without having move m_{1a} or m_{2a} in between them, since it requires a to point to b.

*L*et $\mathsf{m}_0, \mathsf{m}_1, \ldots, \mathsf{m}_k$ be a sequence of moves corresponding to an execution of Algorithm 7.2 for graph G. If m_i and m_j are list moves and m_l is not a list move for all l with $i < l < j$, then $[\mathsf{m}_i, \mathsf{m}_j]$ is called a *list free sequence*.

Table 7.3 shows the possible list free sequences according to Algorithm 7.2. Each row represents the list free sequences that start with the specified move in the first

column and end with any of the moves in the following columns. The initial, nonrecurring situation in which m_{1a} and m_{1b} could follow after each other without one of the other list moves in between is not considered. Table 7.3 already contains the results of Lemma 7.4.7. The cases, in which a new Φ-edge is created before the second list move becomes enabled, are marked with a Φ.

Move	m_{1a}	m_{1b}	m_{2a}	m_{2b}	m_{3a}	m_{3b}
m_{1a}	–	–	✓	✓	✓	✓
m_{1b}	–	–	✓	✓	✓	✓
m_{2a}	–	✓Φ	✓Φ	✓Φ	✓	✓
m_{2b}	✓Φ	–	✓Φ	✓Φ	✓	✓
m_{3a}	–	–	✓Φ	✓Φ	–	–
m_{3b}	–	–	✓Φ	✓Φ	–	–

Table 7.3: Possible list free sequences. The cases, in which a new Φ-edge is created before the second list move becomes enabled, are marked with a Φ.

In order that two moves of type 1 can be executed, there must be an intermediate move of type 2. The same holds for moves of type 3. In the following it will be shown that the number of Φ-edges increases during each of the list free sequences $[m_{2a}, m_{1b}]$, $[m_{2a}, m_{2a}]$, $[m_{2a}, m_{2b}]$, $[m_{3a}, m_{2a}]$ and $[m_{3a}, m_{2b}]$. By symmetry the results also hold for $[m_{2b}, m_{1a}]$, $[m_{2b}, m_{2b}]$, $[m_{2b}, m_{2a}]$, $[m_{3b}, m_{2a}]$ and $[m_{3b}, m_{2b}]$, respectively. Two sets of configurations that will play a major role in the upcoming proofs are defined in advance:

C^1: In these configurations, node a points at b, a is disabled, i.e. all neighbors of a, except for b, point to other nodes via heavier edges. Node b points to a or to *null* and is enabled. The heaviest edge b could point to is $\langle b, z\rangle$, where z on its part points to b or *null*. Furthermore, node z has a neighbor t. Figure 7.7 shows a configuration of C^1. Note that node b (resp. z) may point to *null* or a (resp. b), which is indicated by the dotted arrows.

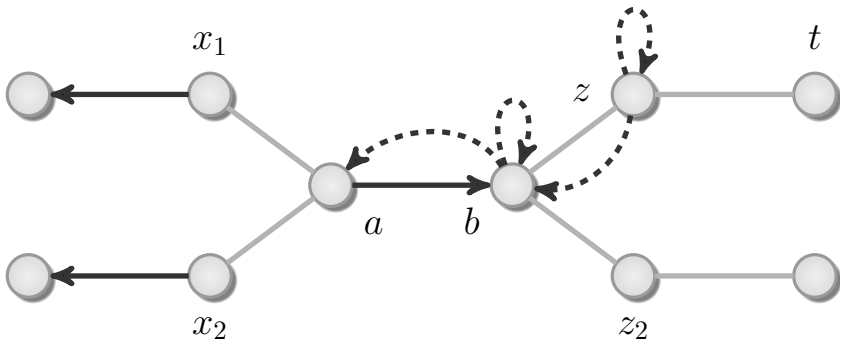


Figure 7.7: Configurations of C^1

C^2: In these configurations, nodes a and b are pointing at each other and are disabled, i.e. all neighbors of a and b point to other nodes via heavier edges. A configuration of C^2 is shown in Figure 7.8.

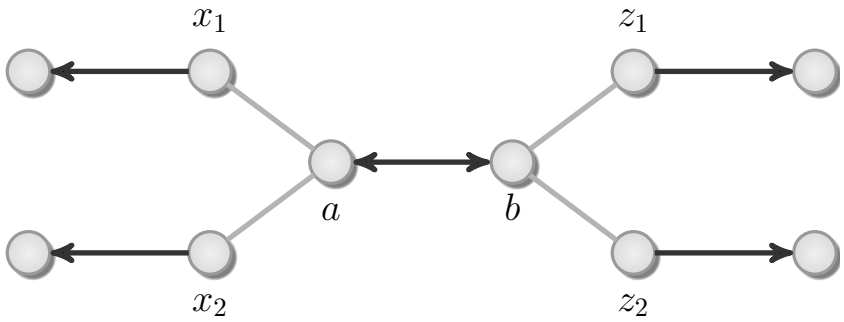


Figure 7.8: Configurations of C^2

Remark 2. The upcoming proofs involve many nodes. Sometimes there may be several possibilities for nodes z, x, and t. For example, in Figure 7.7 there are nodes x_1 and x_2. Node a might be pointing to x_1 first and then it switches to x_2 later. These moves are not of relevance – only the list moves are addressed here – so they are not considered at this point. The node that is considered to be "node x" in these cases is the one that can point to a via the heaviest edge.

Lemma 7.4.6. *If for a configuration $c \in C^1 \cup C^2$ the next list move is m_{2a} (resp. m_{2b}), then the number of Φ-edges increases before a (resp. b) becomes enabled to perform this move.*

Proof. Let $c \in C^1$. Initially, $\langle a, b\rangle$ is not a Φ-edge. As long as this is not the case, the following holds: If an $x \in N(a)$ points towards a (necessary for m_{2a} ever being executed again), then $\langle x, a\rangle$ becomes a new Φ-edge. Thus, a points at b all the time (m_{3a} is not allowed), no other $x \in N(a)$ points at a and the next list move is m_{2b}. In order to let move m_{2b} ever be executed again, b first has to point to another neighbor z. As soon as all neighbors of b (distinct from a) point to heavier edges (necessary so that m_{2b} can become enabled), $\langle a, b\rangle$ becomes a new Φ-edge.

Now let $c \in C^2$. Nodes a and b are disabled and $\langle a, b\rangle$ is a Φ-edge. a resp. b will not become enabled unless a neighbor $x \in N(a)$ resp. $z \in N(b)$ performs a move and points towards a resp. b. In doing so, the Φ-edge will be moved in the corresponding direction, i.e. $\langle x, a\rangle$ (resp. $\langle z, b\rangle$) becomes a Φ-edge and $\langle a, b\rangle$ is no more a Φ-edge, and the resulting configuration is contained in C^1 (possibly with exchanged roles of a and b). The result follows from the first case. □

Lemma 7.4.7. *During each of the list free sequences [m_{2a}, m_{1b}], [m_{2a}, m_{2a}], [m_{2a}, m_{2b}], [m_{3a}, m_{2a}] and [m_{3a}, m_{2b}] as well as [m_{2b}, m_{1a}], [m_{2b}, m_{2b}], [m_{2b}, m_{2a}], [m_{3b}, m_{2a}] and [m_{3b}, m_{2b}] respectively the number of Φ-edges increases.*

Proof. The sequences will be analyzed one by one.

Case **[m_{2a}, m_{1b}]:**
Move m_{2a} lets a point to b, i.e. all other neighbors of a point to heavier edges. Node b on its part cannot have been pointing to another node immediately before this move, otherwise m_{2a} would not have been possible. Since b does not perform move m_{3b} next, it must have been pointing at $null$. Assume there is no $x \in N(b)\backslash\{a\}$ that points to b or $null$, then via move m_{2a} edge $\langle a, b\rangle$ would have become a new Φ-edge. So let $x \in N(b)\backslash\{a\}$ with $x.p = b$ or $x.p = null$. b cannot execute m_{1b} until all its neighbors point at a heavier edge. But in that case, again, edge $\langle a, b\rangle$ would have become a new Φ-edge. If b points to x previously, it cannot set its pointer to $null$ (which is necessary for move m_{1b}) before a performs a move. Since other list moves are excluded, the only possibility is that a first points at a heavier edge and later to $null$. But a cannot direct its pointer to a heavier edge, unless the other node of this edge that on its part was pointing to a heavier edge previously, points to a. In doing so, a new Φ-edge is created.

Cases **[m_{2a}, m_{2a}]** and **[m_{2a}, m_{2b}]:**
Node a points at b, all other neighbors of a are pointing at heavier edges. b points at a or at $null$.

- If there is a $z \in N(b) \setminus \{a\}$, which enables b, then the configuration is contained in C^1 and the rest follows from Lemma 7.4.6.
- All neighbors of b (except for a) are pointing at heavier edges. If b points at $null$, then via the first move, m_{2a}, $\langle a, b \rangle$ already became a new Φ-edge. If b points at a, then the configuration is contained in C^2 and the rest follows from Lemma 7.4.6.

Case **[m_{3a}, m_{2a}]:**
Initially a points at b, b on its part (and all other neighbors of a) has selected another node. Hence, a sets its pointer to $null$ via executing move m_{3a}. In order that m_{2a} can be the next list move, b has to point to $null$ first (b cannot point towards a without performing a list move). This is impossible, until a directs its pointer to a heavier edge first. Therefore there must be an $x \in N(a)$ that points at a afore. This makes edge $\langle x, a \rangle$ a new Φ-edge.

Case **[m_{3a}, m_{2b}]:**
Again, initially a points at b, b on its part (and all other neighbors of a) has selected another node. Hence, a sets its pointer to $null$ via executing move m_{3a}. In order that m_{2b} can be the next list move, a has to point to $null$ at that time. Before, the node could point to a heavier edge and back to $null$. Since this requires a node x to point towards a via a heavier edge (this node therefore would be disabled after this move) this would make $\langle x, a \rangle$ a new Φ-edge. So from now on let a point to $null$ and let no other neighbor enable it. If all neighbors of b (except for a) point to a heavier edge (this is required for m_{2b} being enabled), edge $\langle a, b \rangle$ becomes a new Φ-edge.

By symmetry Lemma 7.4.7 also holds for the list free sequences [m_{2b}, m_{1a}], [m_{2b}, m_{2b}], [m_{2b}, m_{2a}], [m_{3b}, m_{2a}] and [m_{3b}, m_{2b}]. □

Figure 7.9 illustrates the possible list move sequences. According to Lemma 7.4.7 (c.f. Table 7.3) edges that increase the number of Φ-edges are weighted 1, other edges have weight 0. The basic idea of the following proof is to determine at which length a path in the depicted graph necessarily exceeds the cost of $n/2$.

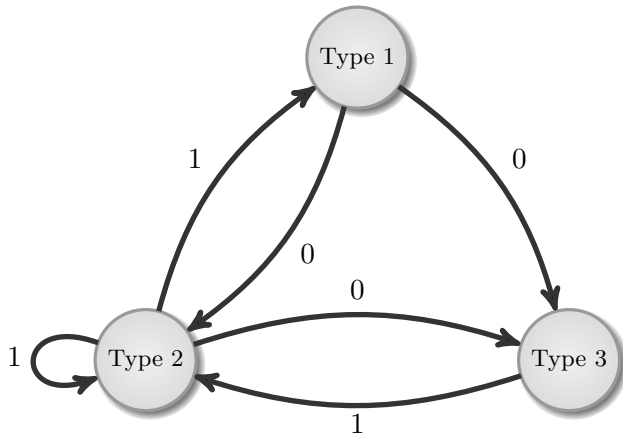


Figure 7.9: Graph of list free sequences

Theorem 7.4.1. *Running under a central scheduler Algorithm 7.2 stabilizes after at most* $(n+3)m$ *moves, if* $m > 0$.

Proof. To make use of induction it is necessary to determine an upper bound on the number of moves of type 1 and 3. Considering the graph depicted in Figure 7.9 this means to identify a path not exceeding $n/2$ cost containing as much occurrences of type 1 and 3 as possible.

Since initially both nodes, a and b, could point to *null* for once, it is possible that m_{1a} and m_{1b} are executed without another intermediate list move. In this case the number of Φ-edges does not necessarily increase (c.f. page 115). After that the list moves follow the graph of Figure 7.9. The next move of type 3 can be executed without cost increases. The following list move will be attended by an increase of Φ and so will at least every second subsequent list move.

Therefore $\#(\mathsf{m}_{1a}) + \#(\mathsf{m}_{1b}) + \#(\mathsf{m}_{3a}) + \#(\mathsf{m}_{3b}) \leq \#(\textit{all list moves}) \leq n+3$. So, the total number of moves the algorithm needs to stabilize in G is greater than the number of moves in G' by at most $n+3$ moves. Clearly, for a graph that contains only one edge Algorithm 7.2 stabilizes after at most n moves. By induction over the edges of G this results in: $\#(G) \leq (n+3)m$. □

In the following a lower bound for the number of moves for Algorithm 7.2 of $\Omega(n^2)$ for the central scheduler is provided. For this purpose consider the *line graph* with n nodes: The nodes are arranged along a line with ascending weighted edges from left

to right. Initially, let all nodes point to their left neighbor, except for the first and the last node that point to *null*. Consider two phases:

1. From left to right, all enabled nodes, one after the other, point to their right neighbor, the last one points to its left neighbor.

2. From left to right, all enabled nodes, one after the other, point to their left neighbor, the first one points to *null*.

These two phases alternate until the algorithm stabilizes. The final configuration is reached after $n^2/2 - n/2 + 1$ moves.

7.5 Distributed Scheduler

If Algorithm 7.2 is executed under the distributed scheduler it is possible that two neighboring nodes make a move at the same time. Hence the results of the last section do not necessarily hold true.

In [GT07] Tixeuil and Gradinariu showed how to transform an algorithm that stabilizes under a central scheduler into an algorithm that stabilizes under a distributed scheduler, provided unique node identifiers exist. Their method is now applied to Algorithm 7.2. After that it will be shown that the resulting algorithm also stabilizes after at most $O(nm)$ moves, although the transformation of [GT07] usually comes with a slowdown factor of $O(\Delta)$ moves.

The basic idea is to provide the nodes with an additional variable *want_to_act*, indicating, whether a node is enabled with respect to the original algorithm, and a predicate *allowed_to_act*, that is used to guarantee that no two neighboring enabled nodes execute a move simultaneously. In particular, only the node with the highest identifier among the neighboring nodes having their *want_to_act* variable set to true is allowed to execute a rule of the original algorithm.

One important property of Algorithm 7.2 is that a node is always disabled immediately after its move, if the algorithm is running under a central scheduler. The transformation of [GT07] is changed in order to preserve this property: When a node executes a move of the original algorithm, simultaneously the variable *want_to_act* is set to false. This simplifies the analysis of Algorithm 7.3.

Algorithm 7.3 Self-Stabilizing 2-Approximation Maximum Weight Matching under a Distributed Scheduler

Predicates:

$v.allowed\_to\_act$:
$v.want\_to\_act \wedge v > \max\{x \in N(v) \mid x.want\_to\_act\}$

Functions:

$BestMatch(v)$:
if $C_v \neq \emptyset$ **then return** $maxC_v$
else return $null$

Actions:

$\mathsf{R}_1 :: [v.want\_to\_act \neq (v.p \neq BestMatch(v) \vee v.w \neq w(\langle v, v.p\rangle))]$
$\longrightarrow v.want\_to\_act := (v.p \neq BestMatch(v) \vee v.w \neq w(\langle v, v.p\rangle))$

$\mathsf{R}_2 :: [v.allowed\_to\_act]$
$\longrightarrow v.p := BestMatch(v)$
$v.w := w(\langle v, v.p\rangle)$
$v.want\_to\_act :=$ false

Using Algorithm 7.3 neighboring nodes cannot both execute a move of type R_2 in the same step. Therefore the results of Table 7.3 still hold true under a distributed scheduler. To prove this, arrange all moves that are executed simultaneously in an arbitrary sequential order and execute them one by one. Since none of these executing nodes are neighbors, their moves do not influence each other. Therefore, the moves of Algorithm 7.3 using the distributed scheduler can be regarded as being executed under the central scheduler. This allows to carry over the definitions introduced in Section 7.4.2.

As for the central scheduler, the moves of Algorithm 7.3 for G will be mapped to moves for G'. A move in which a node x executes rule R_1 (resp. R_2) will be called u_x (resp. m_x). Since the moves of types m_a and m_b cannot be executed simultaneously in G, all moves $\mathsf{u}_x, \mathsf{m}_x$ are mapped to themselves for $x \notin \{a, b\}$. The moves m_a and m_b will be transformed as in Section 7.4.2.

For $x \in \{a, b\}$ the move according to rule R_1 setting $want\_to\_act_x$ to true (resp. false) is denoted by u_{x+} (resp. u_{x-}). The moves u_{a+} and u_{b+} have to be subjected to a detailed review. For example node a can be enabled to perform move m_{1a} for

G based on a previous execution of u_{a+}, which is illegal for G' (except for a not in sync), since a is not enabled to perform any move in G' in that case. Thus, u_{a+} will be mapped to $\perp$, if a performs this move in G, because it wants to perform m_{1a} or m_{3a} and is in sync. If the next move of a is u_{a-}, m_{1a} or m_{3a}, this move will also be mapped to $\perp$. If the next move m_{next} of a is any other move, then it is mapped to a move consisting of a combination of u_{a+} and m_{next}. Node b is treated alike.

The following analyzes how often node a (resp. node b) can be *enabled* to perform move m_{1a} or m_{3a} (resp. m_{1b} or m_{3b}). In combination with the number of *executions* of the moves m_{1a}, m_{3a}, and the corresponding moves of node b, the number of moves mapped to $\perp$ in G' can be determined.

Lemma 7.5.1. *If node a or node b is enabled to perform a type 1 move (resp. a type 3 move) and later it again gets enabled to perform the same move, then there will either be an intermediate execution of a list move (resp. of a list move type 2) or the number of Φ-edges increases.*

Proof. The two types are analyzed individually. Consider move m_{1a} first. Initially, let node a be enabled to perform move m_{1a}, i.e. a points at $null$, b points at a or to $null$, and all other neighbors of a are pointing towards heavier edges. In order for node a to become enabled to perform move m_{1a} again without executing it (in this case the number of Φ-edges increases, see Table 7.3), m_{1a} has to be disabled in the first place. This can be realized, as either b points at a heavier edge, or a neighbor of a points at a via a heavier edge. In the former case, a cannot become enabled to perform move m_{1a}, unless b performs move m_{1b} or m_{2b} first.

So let b point to a or to $null$ constantly. If a neighbor $x \in N(a)$ points at a, then $\langle x, a\rangle$ is a Φ-edge. If b was enabled before this move, this is a new Φ-edge. In particular this is the case if b points to $null$. So assume b disabled. If a neighbor $z \in N(b)$ should point at b then $\langle z, b\rangle$ would become a new Φ-edge. If x should point at a heavier edge later (required for a being enabled to perform move m_{1a}), then $\langle a, b\rangle$ will become a new Φ-edge.

Next, the move m_{3a} is considered. Let node a be enabled to perform move m_{3a}, that is a points at b, all neighbors of a, including b are pointing at heavier edges. So that node a becomes enabled to perform move m_{3a} again, without executing this move, first of all m_{3a} has to be disabled. This can be realized, as either b or another neighbor

of a points at a. If b directs its pointer towards a now, this is move m_{2b}. So let b constantly point to heavier edges in the following. Two cases are considered:

Case 1: Edge $\langle a,b\rangle$ is a Φ-edge.
If a neighbor $x \in N(a)\backslash\{b\}$ points at a, edge $\langle x,a\rangle$ replaces $\langle a,b\rangle$ as a Φ-edge. In order to enable m_{3a} anew x has to point to a heavier edge again. In doing so, this edge replaces $\langle x,a\rangle$ as a ϕ-edge and $\langle a,b\rangle$ becomes a new Φ-edge.

Case 2: Edge $\langle a,b\rangle$ is not a Φ-edge.
As long as $\langle a,b\rangle$ is not a Φ-edge, the following holds true: If a neighbor $x \in N(a)\backslash\{b\}$ points at a, $\langle x,a\rangle$ becomes a new Φ-edge.

The same arguments hold for moves m_{1b} and m_{3b}. □

Lemma 7.5.2. *Node a (resp. node b) will not be enabled to perform move m_{1a} or m_{3a} (resp. m_{1b} or m_{3b}) more than $n/2+1$ times each.*

Proof. Before a becomes enabled to perform move m_{1a} (resp. move m_{3a}) – except for the very first execution – the number of Φ-edges increases or a list move of type 1 or 2 (resp. type 2) will be executed (Lemma 7.5.1). Lemma 7.4.7 yields that the number of Φ-edges increases before a new move of type 1 (resp. type 3) is possible in that situation. Thus, a will be enabled to perform move m_{1a} (resp. m_{3a}) at most $n/2+1$ times. The same holds for moves m_{1b} and m_{3b}. □

Theorem 7.5.1. *Running under a distributed unfair scheduler Algorithm 7.3 stabilizes after at most $(4n+8)m$ moves.*

Proof. As in Section 7.4.2 a bound for the possible additional moves for graph G compared to graph G' is calculated. In the distributed case for this purpose the number of moves of type u_{a+} and u_{b+} that are mapped to $\perp$ have to be counted, each with at most one consecutive move, respectively. $\#(\mathsf{u}_{a+}) \leq (n/2+1)+(n/2+1) = n+2$ (Lemma 7.5.2). In the worst case every time the consecutive move will be mapped to $\perp$ as well. That makes a total of $2n+4$ moves. The same number has to be added for u_{b+} respectively. This yields: $\#(G) \leq \#(G')+2(2n+4)$. The theorem is now easily proved by induction on the number of edges of G. □

7.6 Conclusion

This chapter presented a new analysis of the time complexity of a self-stabilizing algorithm that computes a 2-approximation for the maximum weight matching problem [MM07]. The analysis is based on a new proof technique. It is shown that the original algorithm requires $O(nm)$ moves under the central scheduler and a modified version $O(nm)$ moves under the distributed scheduler. Previously known bounds were exponential. The following two conjectures are considered for future research:

Conjecture 1. *Algorithm 7.2 stabilizes after at most $O(n^2)$ moves under the central scheduler. This bound also holds for Algorithm 7.3 using the distributed scheduler.*

Conjecture 2. *Algorithm 7.2 stabilizes after at most $O(n^2)$ moves under the distributed scheduler even without the transformation of [GT07].*

The example presented at the end of Section 7.4 only requires $O(n^2)$ moves and I was unable to find an example requiring $O(nm)$ moves, where m is not in the order of n. It seems that in order to prove any of these conjectures, a different approach is required. Induction on the number of edges is no longer possible.

Chapter 8

Conclusion

This chapter summarizes the results of this thesis and highlights perspectives for future research.

8.1 Summary

This thesis provided new self-stabilizing algorithms for several problems in algorithmic graph theory. Additionally, several existing algorithms were analyzed with respect to their time- and space complexity.

The main contribution of this thesis is a new proof technique for the complexity analysis of self-stabilizing algorithms. It is based on graph reduction and allows to leverage complete induction in the proofs. This novel approach was used for a new analysis of the time complexity of a self-stabilizing algorithm that computes a 2-approximation for the maximum weight matching problem [MM07]. While the upper bound on its time complexity was previously stated to be exponential, the new technique proved that the algorithm indeed stabilizes after $O(nm)$ moves under the central scheduler. Furthermore, a modified version of the algorithm could be shown to stabilize after $O(nm)$ moves under the distributed scheduler.

A self-stabilizing algorithm for the calculation of a weakly connected minimal dominating set by Srimani and Xu ([SX07]) is proven to have an exponential runtime in an adverse setting. While it was known that this algorithm has an exponential upper

bound on the time complexity, it was an open problem whether this limit is sharp. In this thesis a new self-stabilizing algorithm for the construction of weakly connected minimal dominating sets was presented that requires only a polynomial number of moves under the distributed scheduler.

A self-stabilizing algorithm for a $3-2/(\Delta+1)$-approximation minimum vertex cover in anonymous networks was presented. It assumes the link-register model with composite atomicity and a distributed scheduler. Stabilization is reached after $O(n+m)$ moves resp. $O(\Delta)$ rounds. Furthermore, the algorithm requires only $O(\log n)$ storage per node. Note that unlike all previously known self-stabilizing algorithms for the vertex cover problem this approach does not utilize symmetry-breaking mechanisms such as restricted concurrency, unique identifiers, or randomization. The algorithm achieves a 2-approximation vertex cover if it is executed on a tree.

This thesis presented the first two self-stabilizing algorithms for the edge-monitoring problem. Two versions of this problem were considered: The first algorithm calculates a minimal edge monitoring assuming that the selected nodes actually monitor *all* edges they can. Using the second algorithm, the nodes obtain information about the edges they actually have to monitor. Under the distributed scheduler both algorithms stabilize after $O(mn^2)$ moves.

8.2 Future Perspectives

The self-stabilizing algorithm for the weakly connected minimal dominating set problem presented in Chapter 4 is based on a spanning tree. The same holds for all other currently known self-stabilizing algorithms for this problem. It would be interesting to investigate if it is possible to create such a structure without a tree as a basis.

In Chapter 5 a self-stabilizing approximation algorithm for the calculation of a $3-2/(\Delta+1)$-approximation of a vertex cover in anonymous networks was presented. It is an open question whether there is a self-stabilizing algorithm with better approximation ratio that also requires only $O(\log n)$ storage per node.

The algorithms for the edge-monitoring problem (Chapter 6) assume that the nodes do not have a limit on the number of edges they are allowed to monitor. Such a limit makes the calculation of a valid monitoring significantly more difficult. Developing

a self-stabilizing algorithm with polynomial runtime for this variation of the edge-monitoring problem would be a challenging task.

In Chapter 7, a new proof method was presented that was applied to an algorithm for the maximum weight matching problem [MM07]. While the previous best known upper bound was exponential it was shown that the algorithm indeed stabilizes within polynomial time. A starting point for future research is the use of this method for the analysis of other algorithms.

On the one hand, the complexity of Algorithm 7.2 could be proven to be limited to at most $O(mn)$ moves, but on the other hand this thesis could not provide an example that exceeds $O(n^2)$ moves. Hence it remains an open question, whether Conjectures 1 and 2 of Chapter 7 indeed hold true or can be proven wrong. The latter could be done by presenting such an example. To prove that Algorithm 7.2 indeed has a quadratic move complexity, it appears that the method presented in Chapter 7 does not suffice since it makes use of induction on the number of edges.

Section 2.4 discussed self-stabilizing algorithms for classical problems in algorithmic graph theory. However, such an overview is necessarily subject to an "aging process". Due to the intense research activity in the field of self-stabilization there soon will be new algorithms for the mentioned problems and it is worthwhile to create supplementary lists every now and then. Furthermore, there are self-stabilizing algorithms for many other problems, not necessarily related to graph theory. A list with these algorithms can be included in a more application-oriented work.

As mentioned in Section 3.2, there is no perfect proof method that is applicable to all algorithms to verify their self-stabilization property or to determine their worst-case complexity. It is rather an experience-based guessing which method may lead to a good result. Hence, it is possible to further investigate the systematic application of proof methods as well as to look for new proof methods.

List of Algorithms

List of Figures

Bibliography

[AAFJ08] D. Angluin, J. Aspnes, M. J. Fischer, and H. Jiang. Self-stabilizing population protocols. *Autonomous and Adaptive Systems, ACM Trans.*, 3(4):1–28, 2008.

[AB93] Y. Afek and G. M. Brown. Self-stabilization over unreliable communication media. *Distributed Computing*, 7:27–34, 1993.

[ABB97] Y. Afek and A. Bremler-Barr. Self-stabilizing unidirectional network algorithms by power-supply (extended abstract). In *Proc. Discrete Algorithms (SODA), 8^{th} Ann. ACM-SIAM Symp., New Orleans, LA, USA, 1997*, pages 111–120. ACM/SIAM, 1997.

[ABB98] Y. Afek and A. Bremler-Barr. Self-stabilizing unidirectional network algorithms by power supply. *Theoretical Computer Science, Chicago J.*, 1998, 1998.

[AG90] A. Arora and M. G. Gouda. Distributed reset (extended abstract). In *Proc. Foundations of Software Technology and Theoretical Computer Science (FSTTCS), 10^{th} Conf., Bangalore, India, 1990*, volume 472 of *LNCS*, pages 316–331. Springer, 1990.

[AG93] A. Arora and M. G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *Software Engineering, IEEE Trans.*, 19:1015–1027, 1993.

[AG94] A. Arora and M. G. Gouda. Distributed reset. *Computers, IEEE Trans.*, 43(9):1026–1038, 1994.

[AGLP89] B. Awerbuch, A. V. Goldberg, M. Luby, and S. A. Plotkin. Network decomposition and locality in distributed computation. In *Proc. Foundations of Computer Science (FOCS), 30^{th} Ann. Symp., Research Triangle Park, NC, USA, 1989*, pages 364–369. IEEE Computer Society, 1989.

[AK93] S. Aggarwal and S. Kutten. Time optimal self-stabilizing spanning tree algorithms. In *Proc. Foundations of Software Technology and Theoretical Computer Science (FSTTCS), 13^{th} Conf., Bombay, India, 1993*, volume 761 of *LNCS*, pages 400–410. Springer, 1993.

[AKM+07] B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, and G. Varghese. A time-optimal self-stabilizing synchronizer using a phase clock. *Dependable and Secure Computing, IEEE Trans.*, 4(3):180–190, 2007.

[AKY90] Y. Afek, S. Kutten, and M. Yung. Memory-efficient self stabilizing protocols for general networks. In *Distributed Algorithms (WDAG), 4^{th} Int. Workshop, Bari, Italy, 1990*, volume 486 of *LNCS*, pages 15–28. Springer, 1990.

[Ang80] D. Angluin. Local and global properties in networks of processors (extended abstract). In *Proc. Theory of Computing (STOC), 12th Ann. ACM Symp., Los Angeles, CA, USA, 1980*, pages 82–93. ACM, 1980.

[APSVD94] B. Awerbuch, B. Patt-Shamir, G. Varghese, and S. Dolev. Self-stabilization by local checking and global reset (extended abstract). In *Proc. Distributed Algorithms (WDAG), 8th Int. Workshop, Terschelling, The Netherlands, 1994*, volume 857 of *LNCS*, pages 326–339, Terschelling, Netherlands. Springer, 1994.

[AS92] G. Antonoiu and P. K. Srimani. A self-stabilizing distributed algorithm to construct an arbitrary spanning tree of a connected graph. *Computers and Mathematics with Applications*, 30, 1992.

[AS97a] G. Antonoiu and P. K. Srimani. Distributed self-stabilizing algorithm for minimum spanning tree construction. In *Parallel Processing (Euro-Par), 3rd Europ. Conf., Passau, Germany, 1997*, volume 1300 of *LNCS*, pages 480–487. Springer, 1997. 10.1007/BFb0002773.

[AS97b] G. Antonoiu and P. K. Srimani. A self-stabilizing distributed algorithm to find the center of a tree graph. *Parallel Algorithms and Applications*, 10(3-4):237–248, 1997.

[AS98] G. Antonoiu and P. K. Srimani. A self-stabilizing distributed algorithm for minimal spanning tree problem in a symmetric graph. *Computers and Mathematics with Applications*, 35(10):15–23, 1998.

[ÅS10] M. Åstrand and J. Suomela. Fast distributed approximation algorithms for vertex cover and set cover in anonymous networks. In *Proc. Parallelism in Algorithms and Architectures (SPAA), 22nd ACM Symp., Santorini, Greece, 2010*, pages 294–302, Santorini, Greece. ACM, 2010.

[ASSC02] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38:393–422, 2002.

[AW04] H. Attiya and J. L. Welch. *Distributed computing: fundamentals, simulations, and advanced topics*. Wiley Series on Par. and Distr. Comput. Wiley, 2004.

[Awe85] B. Awerbuch. Complexity of network synchronization. *J. ACM*, 32(4):804–823, 1985.

[AWF03] K. M. Alzoubi, P.-J. Wan, and O. Frieder. Independent Set, Weakly-Connected Dominating Set, and Induced Spanners in Wireless Ad Hoc Networks. *Foundations of Computer Science, Int. J.*, 14(2):287–303, 2003.

[BDGM00] J. Beauquier, A. K. Datta, M. Gradinariu, and F. Magniette. Self-stabilizing local mutual exclusion and daemon refinement. In *Proc. Distributed Computing (DISC), 14th Int. Conf., Toledo, Spain, 2000*, volume 1914 of *LNCS*, pages 223–237. Springer, 2000.

[BDGT09] S. Bernard, S. Devismes, M. Gradinariu, and S. Tixeuil. Optimal deterministic self-stabilizing vertex coloring in unidirectional anonymous networks. In *Proc. Parallel and Distributed Processing (IPDPS), IEEE Int. Symp., Rome, Italy, 2009*, pages 1–8. IEEE Computer Society, 2009.

[BDPBR10] L. Blin, S. Dolev, M. Gradinariu Potop-Butucaru, and S. Rovedakis. Fast self-stabilizing minimum spanning tree construction - using compact nearest common ancestor labeling scheme. In *Proc. Distributed Computing (DISC), 24th Int. Symp., Cambridge, MA, USA, 2010*, volume 6343 of *LNCS*, pages 480–494. Springer, 2010.

[BK07] J. Burman and S. Kutten. Time-optimal asynchronous self-stabilizing spanning tree. In *Proc. Distributed Computing (DISC), 21st Int. Symp., Lemesos, Cyprus, 2007*, volume 4731 of *LNCS*, pages 92–107. Springer, 2007.

[BLB95] F. Butelle, C. Lavault, and M. Bui. A uniform self-stabilizing minimum diameter tree algorithm (extended abstract). In *Proc. Distributed Algorithms (WDAG), 9th Int. Workshop, Le Mont-Saint-Michel, France, 1995*, volume 972 of *LNCS*, pages 257–272. Springer, 1995.

[BM03] J. R. S. Blair and F. Manne. Efficient self-stabilizing algorithms for tree network. In *Proc. Distributed Computing Systems (ICDCS), 23rd Int. Conf., Providence, RI, USA, 2003*. IEEE Computer Society, 2003.

[BM09] J. R. S. Blair and F. Manne. An efficient self-stabilizing distance-2 coloring algorithm. In *Proc. Structural Information and Communication Complexity (SIROCCO), 16th Int. Coll., Piran, Slovenia, 2009, Revised Selected Papers*, volume 5869 of *LNCS*, pages 237–251. Springer, 2009.

[BPBR11] L. Blin, M. Gradinariu Potop-Butucaru, and S. Rovedakis. Self-stabilizing minimum degree spanning tree within one from the optimal degree. *Parallel and Distributed Computing, J.*, 71(3):438–449, 2011.

[BPBRT10] L. Blin, M. Gradinariu Potop-Butucaru, S. Rovedakis, and S. Tixeuil. Loop-free super-stabilizing spanning tree construction. In *Proc. Stabilization, Safety, and Security of Distributed Systems (SSS), 12th Int. Symp., New York, NY, USA, 2010*, volume 6366 of *LNCS*, pages 50–64. Springer, 2010.

[BST89] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, 1989.

[CDK05] G. F. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design (4th Edition) (International Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., 2005.

[CDP03] S. Cantarell, A. K. Datta, and F. Petit. Self-stabilizing atomicity refinement allowing neighborhood concurrency. In *Proc. Stabilization, Safety, and Security of Distributed Systems (SSS), 6th Int. Symp., San Francisco, CA, USA, 2003*, volume 2704 of *LNCS*, pages 102–112. Springer, 2003.

[Cha82] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proc. Compiler construction, SIGPLAN Symp., Boston, MA, USA, 1982*, pages 98–105. ACM, 1982.

[CHS02] S. Chattopadhyay, L. Higham, and K. Seyffarth. Dynamic and self-stabilizing distributed matching. In *Principles of Distributed Computing (PODC), 21st Ann. Symp., Monterey, CA, USA, 2002*, pages 290–297, Monterey, California. ACM, 2002.

[CL02] Y. P. Chen and A. L. Liestman. Approximating minimum size weakly-connected dominating sets for clustering mobile ad hoc networks. In *Proc. Mobile Ad Hoc Networking and Computing (MobiHoc), 3rd Int. Symp., Lausanne, Switzerland, 2002*, pages 165–172, 2002.

[CT07] P. Chaudhuri and H. Thompson. A self-stabilizing distributed algorithm for edge-coloring general graphs. *Combinatorics, Australasian J.*, 38:237–247, 2007.

[CT11] P. Chaudhuri and H. Thompson. Improved self-stabilizing algorithms for l(2, 1)-labeling tree networks. *Mathematics in Computer Science*, 5(1):27–39, 2011.

[CYH91] N.-S. Chen, H.-P. Yu, and S.-T. Huang. A self-stabilizing algorithm for constructing spanning trees. *Information Processing Letters*, 39:147–151, 1991.

[DDH$^+$11] A. K. Datta, S. Devismes, K. Heurtefeux, L. L. Larmore, and Y. Rivierre. Self-stabilizing small k-dominating sets. In *Networking and Computing (ICNC), 2nd Int. Conf., Osaka, Japan, 2011*, pages 30–39. Conference Publishing Service, 2011. Best Paper Award.

[DDK09] K. Drira, L. Dekar, and H. Kheddouci. A self-stabilizing $(\Delta + 1)$-Edge-Coloring algorithm of arbitrary graphs. In *Proc. Parallel and Distributed Computing, Applications and Technologies (PDCAT), 10th Int. Conf., Hiroshima, Japan, 2009*, pages 312–317, 2009.

[DFG06] V. Drabkin, R. Friedman, and M. Gradinariu. Self-stabilizing wireless connected overlays. In *Proc. Principles of Distributed Systems (OPODIS), 10th Int. Conf., Bordeaux, France, 2006*, volume 4305 of *LNCS*, pages 425–439. Springer, 2006.

[DGH$^+$97] J. E. Dunbar, J. W. Grossman, J. H. Hattingh, S. T. Hedetniemi, and A. A. McRae. On weakly connected domination in graphs. *Discrete Mathematics*, 167/168:261–269, 1997.

[DGT04] A. K. Datta, M. Gradinariu, and S. Tixeuil. Self-stabilizing mutual exclusion under arbitrary scheduler. *Computer J.*, 47(3):289–298, 2004.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569–, 1965.

[Dij74] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[Dij86] E. W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1:5–6, 1986.

[DIM90] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. In *Proc. Principles of Distributed Computing (PODC), 9th Ann. ACM Symp., Quebec City, QC, Canada, 1990*, pages 103–117, 1990.

[DIM93] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7(1):3–16, 1993.

[DIM97a] S. Dolev, A. Israeli, and S. Moran. Resource bounds for self-stabilizing message-driven protocols. *Computing, SIAM J.*, 26:273–290, 1997.

[DIM97b] S. Dolev, A. Israeli, and S. Moran. Uniform dynamic self-stabilizing leader election. *Parallel and Distributed Systems, IEEE Trans.*, 8(4):424–440, 1997.

[DK08] L. Dekar and H. Kheddouci. Distance-2 self-stabilizing algorithm for a b-coloring of graphs. In *Proc. Stabilization, Safety, and Security of Distributed Systems (SSS), 10th Int. Symp., Detroit, MI, USA, 2010*, volume 5340 of *LNCS*, pages 19–31. Springer, 2008.

[DLL08] D. Dong, Y. Liu, and X. Liao. Self-monitoring for sensor networks. In *Proc. Mobile Ad Hoc Networking and Computing (MobiHoc), 9th ACM Int. Symp. Hong Kong, China, 2008*, pages 431–440. ACM, 2008.

[DLL+11] D. Dong, X. Liao, Y. Liu, C. Shen, and X. Wang. Edge self-monitoring for wireless sensor networks. *Parallel and Distributed Systems, IEEE Trans.*, 22(3):514–527, 2011.

[DLV10] A. K. Datta, L. L. Larmore, and P. Vemula. A self-stabilizing o(k)-time k-clustering algorithm. *Computer J.*, 53(3):342–350, 2010.

[DMP+03] D. P. Dubhashi, A. Mei, A. Panconesi, J. Radhakrishnan, and A. Srinivasan. Fast distributed algorithms for (weakly) connected dominating sets and linear-size skeletons. In *Proc. Discrete algorithms (SODA), 14th Ann. ACM-SIAM Symp., Baltimore, MD, USA, 2003*, pages 717–724, 2003.

[DMT11] S. Dubois, T. Masuzawa, and S. Tixeuil. Maximum metric spanning tree made byzantine tolerant. In *Proc. Distributed Computing (DISC), 25th Int. Symp., Rome, Italy, 2011*, volume 6950 of *LNCS*, pages 150–164. Springer, 2011.

[Dol00] S. Dolev. *Self-stabilization*. MIT Press, 2000.

[DY06] S. Dolev and R. Yagel. Self-stabilizing device drivers. In *Proc. Stabilization, Safety, and Security of Distributed Systems (SSS), 8th Int. Symp., Dallas, TX, USA, 2006*, volume 4280 of *LNCS*, pages 276–289. Springer, 2006.

[Edm65] J. Edmonds. Paths, trees, and flowers. *Mathematics, Canadian J.*, 17:449–467, 1965.

[Elk04] M. Elkin. Distributed approximation: A survey. *Algorithms and Computation Theory, ACM SIG News*, 35(4):40–57, 2004.

[Gär98] F. C. Gärtner. On the relationship between self-stabilization and fault tolerance. In *Proc. Self-Stabilization, Dagstuhl Sem. 98331, Dagstuhl, Germany, 1998*, page 9, 1998.

[Gär99] F. C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31:1–26, 1999.

[Gär03] F. C. Gärtner. A Survey of Self-Stabilizing Spanning-Tree Construction Algorithms. Techn. Rep. IC/2003/38, Swiss Federal Institute of Technology, Lausanne, 2003.

[GGH+04] M. Gairing, W. Goddard, S. T. Hedetniemi, P. Kristiansen, and A. A. McRae. Distance-two information in self-stabilizing algorithms. *Parallel Processing Letters*, 14(3-4):387–398, 2004.

[GGHJ04] M. Gairing, W. Goddard, S. T. Hedetniemi, and D. P. Jacobs. Self-stabilizing maximal k-dependent sets in linear time. *Parallel Processing Letters*, 14(1):75–82, 2004.

[GGKP95] S. Ghosh, A. Gupta, M. H. Karaata, and S. V. Pemmaraju. Self-stabilizing dynamic programming algorithms on trees. In *Proc. Self-Stabilizing Systems, 2^{nd} Workshop, Las Vegas, NV, USA, 1995*, pages 11.1–11.15, 1995.

[GGP96] S. Ghosh, A. Gupta, and S. V. Pemmaraju. A fault-containing self-stabilizing spanning tree algorithm. *Computing and Information, J.*, 2(1):332–338, 1996.

[GHJ+08] W. Goddard, S. T. Hedetniemi, D. P. Jacobs, P. K. Srimani, and Z. Xu. Self-stabilizing graph protocols. *Parallel Processing Letters*, 18(1):189–199, 2008.

[GHJS03a] W. Goddard, S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani. A robust distributed generalized matching protocol that stabilizes in linear time. In *Proc. Distributed Computing Systems (ICDCSW), 23^{rd} Int. Conf. Workshop, Providence, RI, USA, 2003*, pages 461–. IEEE Computer Society, 2003.

[GHJS03b] W. Goddard, S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani. A self-stabilizing distributed algorithm for minimal total domination in an arbitrary system graph. In *Proc. Parallel and Distributed Processing (IPDPS), 17^{th} Int. Symp., Nice, France, 2003*, pages 240.2–. IEEE Computer Society, 2003.

[GHJS03c] W. Goddard, S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani. Self-stabilizing distributed algorithm for strong matching in a system graph. In *Proc. High Performance Conputing (HiPC), 10^{th} Int. Conf., Hyderabad, India, 2003*, volume 2913 of *LNCS*, pages 66–73. Springer, 2003.

[GHJS03d] W. Goddard, S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani. Self-stabilizing protocols for maximal matching and maximal independent sets for ad hoc networks. In *Proc. Parallel and Distributed Processing (IPDPS), 17th Int. Symp., Nice, France, 2003*, pages 162.2–. IEEE Computer Society, 2003.

[GHJS04] W. Goddard, S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani. Fault tolerant algorithms for orderings and colorings. In *Proc. Parallel and Distributed Processing (IPDPS), 18th Int. Symp., Santa Fe, NM, USA, 2004*. IEEE Computer Society, 2004.

[GHJT08] W. Goddard, S. T. Hedetniemi, D. P. Jacobs, and V. Trevisan. Distance-k knowledge in self-stabilizing algorithms. *Theoretical Computer Science*, 399(1-2):118–127, 2008.

[GHS83] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *Programming Languages and Systems, ACM Trans.*, 5(1):66–77, 1983.

[GHS06] W. Goddard, S. T. Hedetniemi, and Z. Shi. An anonymous self-stabilizing algorithm for 1-maximal matching in trees. In *Proc. Parallel and Distributed Processing Techniques and Applications (PDPTA), Int. Conf., Las Vegas, NV, USA, 2006*, pages 797–803. CSREA Press, 2006.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[GK93] S. Ghosh and M. H. Karaata. A self-stabilizing algorithm for coloring planar graphs. *Distributed Computing*, 7:55–59, 1993.

[GK10] N. Guellati and H. Kheddouci. A survey on self-stabilizing algorithms for independence, domination, coloring, and matching in graphs. *Parallel and Distributed Computing, J.*, 70:406–415, 2010.

[GM91] M. G. Gouda and N. J. Multari. Stabilizing communication protocols. *Computers, IEEE Trans.*, 40:448–458, 1991.

[GS10] W. Goddard and P. K. Srimani. Anonymous Self-Stabilizing Distributed Algorithms for Connected Dominating Set in a Network Graph. In *Proc. Complexity, Informatics and Cybernetics (IMCIC), 1st Int. Multi-Conf., Orlando, FL, USA, 2010*, 2010.

[GSRV08] V. C. Giruka, M. Singhal, J. Royalty, and S. Varanasi. Security in wireless sensor networks: Research articles. *Wireless Communications and Mobile Computing*, 8:1–24, 2008.

[GT00] M. Gradinariu and S. Tixeuil. Self-stabilizing vertex coloration and arbitrary graphs. In *Proc. Principles of Distributed Systems (OPODIS), 4th Int. Conf., Paris, France, 2000*, Studia Informatica Universalis, pages 55–70. Suger, Saint-Denis, rue Catulienne, France, 2000.

[GT07] M. Gradinariu and S. Tixeuil. Conflict managers for self-stabilization without fairness assumption. In *Proc. Distributed Computing Systems (ICDCS), 27th Int. Conf., Toronto, Canada, 2007*, page 46. IEEE Computer Society, 2007.

[Hås01] J. Håstad. Some optimal inapproximability results. *J. ACM*, 48(4):798–859, 2001.

[HC92] S.-T. Huang and N.-S. Chen. A self-stabilizing algorithm for constructing breadth first trees. *Information Processing Letters*, 41:109–117, 1992.

[HCW08] T. C. Huang, C.-Y. Chen, and C.-P. Wang. A linear-time self-stabilizing algorithm for the minimal 2-dominating set problem in general networks. *Information Science and Engineering, J.*, 24(1):175–187, 2008.

[Her92] T. R. Herman. *Adaptivity through distributed convergence*. PhD thesis, Department of Computer Science, University of Texas at Austin, 1992. UMI Order No. GAX92-12547.

[Her02] T. R. Herman. A comprehensive bibliography on self-stabilization. *Theoretical Computer Science, Chicago J.*, 2002.

[Her04] T. R. Herman. Models of self-stabilization and sensor networks. In *Distributed Computing (IWDC), 5th Int. Workshop, Kolkata, India, 2003*, volume 2918 of *LNCS*, pages 836–836. Springer, 2004.

[HH92] S.-C. Hsu and S.-T. Huang. A self-stabilizing algorithm for maximal matching. *Information Processing Letters*, 43(2):77–81, 1992.

[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*, 46:805–811, 2003.

[HHLL04] S.-H. Hsu, C.-C. Hsu, S.-S. Lin, and F.-C. Lin. A multi-channel mac protocol using maximal matching for ad hoc networks. In *Proc. Distributed Computing Systems (ICDCSW), 24th Int. Conf. Workshop, Tokyo, Japan, 2004*, volume 7, pages 505–510. IEEE Computer Society, 2004.

[HHT05] S-T. Huang, S.-S. Hung, and C.-H. Tzeng. Self-stabilizing coloration in anonymous planar networks. *Information Processing Letters*, 95(1):307–312, 2005.

[HJS02] S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani. Maximal matching stabilizes in time O(m). *Information Processing Letters*, 80(5):221–223, 2002.

[HJS03] S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani. Linear time self-stabilizing colorings. *Information Processing Letters*, 87:251–255, 2003.

[HKP98] M. Hańćkowiak, M. Karoński, and A. Panconesi. On the distributed complexity of computing maximal matchings. In *Proc. Discrete Algorithms, 9th Ann. ACM-SIAM Symp., San Francisco, CA, USA, 1998*, pages 219–225, 1998.

[HL91] S. T. Hedetniemi and R. C. Laskar. Bibliography on domination in graphs and some basic definitions of domination parameters. *Discrete Mathematics*, 86:257–277, 1991.

[HL01] L. Higham and Z. Liang. Self-stabilizing minimum spanning tree construction on message-passing networks. In *Proc. Distributed Computing (DISC), 15th Int. Conf., Lisbon, Portugal, 2001*, volume 2180 of *LNCS*, pages 194–208. Springer, 2001.

[HL06] C.-F. Hsin and M. Liu. Self-monitoring of wireless sensor networks. *Computer Communications*, 29(4):462–476, 2006.

[HLCW07] T. C. Huang, J.-C. Lin, C.-Y. Chen, and C.-P. Wang. A self-stabilizing algorithm for finding a minimal 2-dominating set assuming the distributed demon model. *Computers and Mathematics with Applications*, 54:350–356, 2007.

[HLP$^+$06] T. Hérault, P. Lemarinier, O. Peres, L. Pilard, and J. Beauquier. Self-stabilizing spanning tree algorithm for large scale systems. Technical Report 1457, Laboratoire de Recherche en Informatique, 2006.

[HM07] A. M. Herzberg and R. M. Murty. Sudoku Squares and Chromatic Polynomials. *Notices of the AMS*, 54(6):708–717, 2007.

[HS11] G. Hong and P. K. Srimani. A self-stabilizing algorithm for two disjoint minimal dominating sets in an arbitrary graph. In *Proc. Southeast Regional, 49th Ann. Conf. Kennesaw, GA, USA, 2011*, pages 306–307. ACM, 2011.

[HT06] S.-T. Huang and C.-H. Tzeng. Distributed edge coloration for bipartite networks. In *Proc. Stabilization, Safety, and Security of Distributed Systems (SSS), 8th Int. Symp., Dallas, TX, USA, 2006*, pages 363–377. Springer, 2006.

[IJ90] A. Israeli and M. Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. In *Proc. Principles of distributed computing (PODC), 9th Ann. ACM Symp., Quebec City, QC, Canada, 1990*, pages 119–131. ACM, 1990.

[IKK02] M. Ikeda, S. Kamei, and H. Kakugawa. A space-optimal self-stabilizing algorithm for the maximal independent set problem. In *Proc. Parallel and Distributed Computing, Applications and Technologies (PDCAT), 3rd Int. Conf., Kanazawa, Japan, 2002*, pages 70–74, 2002.

[JG05] A. Jain and A. Gupta. A distributed self-stabilizing algorithm for finding a connected dominating set in a graph. In *Proc. Parallel and Distributed Computing, Applications and Technologies (PDCAT), 6th Int. Conf., Dalian, China, 2005*, pages 615–619. IEEE Computer Society, 2005.

[Kar01] M. H. Karaata. Self-stabilizing strong fairness under weak fairness. *Parallel and Distributed Systems, IEEE Trans.*, 12:337–345, 2001.

[Kes88] J. L. W. Kessels. An exercise in proving self-stabilization with a variant function. *Information Processing Letters*, 29:39–42, 1988.

[Kin05] J. Kiniwa. Approximation of self-stabilizing vertex cover less than 2. In *Proc. Stabilization, Safety, and Security of Distributed Systems (SSS), 7th Int. Symp., Barcelona, Spain, 2005*, volume 3764 of *LNCS*, pages 171–182. Springer, 2005.

[KK03] S. Kamei and H. Kakugawa. A self-stabilizing algorithm for the distributed minimal k-redundant dominating set problem in tree networks. In *Proc. Parallel and Distributed Computing, Applications and Technologies (PDCAT), 4th Int. Conf., Chengdu, China, 2003*, pages 720–724, 2003.

[KK05] S. Kamei and H. Kakugawa. A self-stabilizing approximation algorithm for the distributed minimum k-domination. *Fundamentals of Electronics, Communications and Computer Sciences, IEICE Trans.*, E88-A(5):1109–1116, 2005.

[KK06] Adrian Kosowski and Lukasz Kuszner. Self-stabilizing algorithms for graph coloring with improved performance guarantees. In *Proc. Artificial Intelligence and Soft Computing (ICAISC), 8th Int. Conf., Zakopane, Poland, 2006*, volume 4029 of *LNCS*, pages 1150–1159. Springer, 2006.

[KK07a] S. Kamei and H. Kakugawa. A self-stabilizing approximation algorithm for the minimum weakly connected dominating set with safe convergence. In *Proc. Reliability, Availability, and Security (WRAS), 1st Int. Workshop, Paris, France, 2007*, pages 57–66, 2007.

[KK07b] S. Kamei and H. Kakugawa. A self-stabilizing distributed approximation algorithm for the minimum connected dominating set. In *Proc. Parallel and Distributed Processing (IPDPS), 21st Int. Symp., Long Beach, CA, USA, 2007*, pages 1–8. IEEE, 2007.

[KK08] S. Kamei and H. Kakugawa. A self-stabilizing approximation for the minimum connected dominating set with safe convergence. In *Proc. Principles of Distributed Systems (OPODIS), 12th Int. Conf., Luxor, Egypt, 2008*, volume 5401 of *LNCS*, pages 496–511. Springer, 2008. 10.1007/978-3-540-92221-6_31.

[KKDT10] S. Kamei, H. Kakugawa, S. Devismes, and S. Tixeuil. A self-stabilizing 3-approximation for the maximum leaf spanning tree problem in arbitrary networks. In *Proc. Computing and Combinatorics (COCOON), 16th Ann. Int. Conf., Nha Trang, Vietnam, 2010*, volume 6196 of *LNCS*, pages 80–89. Springer, 2010.

[KN06] L. Kuszner and A. Nadolski. Self-stabilizing algorithm for edge-coloring of graphs. *Foundations of Computing and Decision Sciences*, 31(2):157–167, 2006.

[KP90] S. Katz and K. Perry. Self-stabilizing extensions for message-passing systems. In *Proc. Principles of distributed computing (PODC), 9th Ann. ACM Symp., Quebec City, QC, Canada, 1990*, pages 91–101. ACM, 1990.

[KS00] M. H. Karaata and K. A. Saleh. A distributed self-stabilizing algorithm for finding maximum matching in trees. *Computer Systems Science and Engineering*, 15(3):175–180, 2000.

[KT10] S. Köhler and V. Turau. A new technique for proving self-stabilization under the distributed scheduler. In *Proc. Stabilization, Safety, and Security of Distributed Systems (SSS), 12th Int. Symp., New York, NY, USA, 2010*, volume 6366 of *LNCS*, pages 65–79. Springer, 2010.

[Lam87] L. Lamport. Distribution, May 1987. E-mail correspondence. Message-Id: <8705281923.AA09105@jumbo.dec.com>.

[LH03] J.-C. Lin and T. C. Huang. An efficient fault-containing self-stabilizing algorithm for finding a maximal independent set. *Parallel and Distributed Systems, IEEE Trans.*, 14(8):742–754, 2003.

[LHWC08] J.-C. Lin, T. C. Huang, C.-P. Wang, and C.-Y. Chen. A self-stabilizing algorithm for finding a minimal distance-2 dominating set in distributed systems. *Information Science and Engineering, J.*, 24(6):1709–1718, 2008.

[Lja07] A. M. Ljapunow. Problème général de la stabilité du mouvement. *Annales de la faculté des sciences de Toulouse*, 9(2):203–474, 1907.

[LLL10] Y. Liu, K. Liu, and M. Li. Passive diagnosis for wireless sensor networks. *Networking, IEEE/ACM Trans.*, 18(4):1132 –1144, 2010.

[LP86] L. Lovász and M. D. Plummer. *Matching theory*. Annals of discrete mathematics. North-Holland, 1986.

[LPSR09] Z. Lotker, B. Patt-Shamir, and A. Rosén. Distributed approximate matching. *Computing, SIAM J.*, 39(2):445–460, 2009.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *Programming Languages and Systems, ACM Trans.*, 4:382–401, 1982.

[Mar03] D. Marx. Graph colouring problems and their applications in scheduling. *Periodica Polytechnica, Electrical Engineering*, 48:11–16, 2003.

[Mje08] M. Mjelde. *New Results on Self-Stabilizing Algorithms, and on Protocols for Wireless Sensor Networks*. PhD thesis, University of Bergen, Norway, 2008.

[MM07] F. Manne and M. Mjelde. A self-stabilizing weighted matching algorithm. In *Proc. Stabilization, Safety, and Security of Distributed Systems (SSS), 9th Int. Symp., Lyon, France, 2007*, pages 383–393, 2007.

[MMPT07] F. Manne, M. Mjelde, L. Pilard, and S. Tixeuil. A new self-stabilizing maximal matching algorithm. In *Structural Information and Communication Complexity (SIROCCO), 14th Int. Coll., Castiglioncello, Italy, 2007*, pages 96–108, 2007.

[MMPT08] F. Manne, M. Mjelde, L. Pilard, and S. Tixeuil. A self-stabilizing 2/3-approximation algorithm for the maximum matching problem. In *Proc. Stabilization, Safety, and Security of Distributed Systems (SSS), 10th Int. Symp., Detroit, MI, USA, 2008*, pages 94–108. Springer, 2008.

[MMPT09] F. Manne, M. Mjelde, L. Pilard, and S. Tixeuil. A new self-stabilizing maximal matching algorithm. *Theoretical Computer Science*, 410(14):1336–13450, 2009.

[MNS95] A. Mayer, M. Naor, and L. Stockmeyer. Local computations on static and dynamic graphs. Technical report, Weizmann Science Press of Israel, Jerusalem, Israel, 1995.

[MT06] T. Masuzawa and S. Tixeuil. A self-stabilizing link-coloring protocol resilient to unbounded byzantine faults in arbitrary networks. In *Proc. Principles of Distributed Systems (OPODIS), 9th Int. Conf., Pisa, Italy, 2005*, volume 3974 of *LNCS*, pages 118–129. Springer, 2006.

[NA02] M. Nesterenko and A. Arora. Stabilization-preserving atomicity refinement. *Parallel and Distributed Computing, J.*, 62(5):766–791, 2002.

[Nie08] T. Nieberg. Local, distributed weighted matching on general and wireless topologies. In *Proc. Foundations of Mobile Computing (FOMC), 5th Int. Workshop, Toronto, Canada, 2008*, pages 87–92. ACM, 2008.

[NR99] R. Niedermeier and P. Rossmanith. Upper bounds for vertex cover further improved. In *Proc. Theoretical Aspects of Computer Science (STACS), 16th Ann. Symp., Trier, Germany, 1999*, volume 1563 of *LNCS*, pages 561–570. Springer, 1999.

[NS95] M. Naor and L. Stockmeyer. What can be computed locally? *Computing, SIAM J.*, 24(6):1259–1277, 1995.

[Pel00] D. Peleg. *Distributed computing: a locality-sensitive approach*. Society for Industrial and Applied Mathematics, 2000.

[PH06] A.-S. K. Pathan and C. S. Hong. A Key-Predistribution-Based Weakly Connected Dominating Set for Secure Clustering in DSN. In *Proc. High Performance Computing and Communications, 2nd Int. Conf., Munich, Germany, 2006*, pages 270–279. Springer, 2006.

[PR01] A. Panconesi and R. Rizzi. Some simple distributed algorithms for sparse networks. *Distributed Computing*, 14(2):97–100, 2001.

[Pre99] R. Preis. Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In *Proc. Theoretical Aspects of Computer Science, 16th Ann. Symp., Trier, Germany, 1999*, volume 1563 of *LNCS*, pages 259–269. Springer, 1999.

[PS09] V. Polishchuk and J. Suomela. A simple local 3-approximation algorithm for vertex cover. *Information Processing Letters*, 109(12):642–645, 2009.

[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 *Proc. Advanced Communication Technology (ICACT), 11th Int. Conf., Gangwon-Do, South Korea, 2009*, volume 1, pages 526–530. IEEE Press, 2009.

[SEK08] H. Sun, B. Effantin, and H. Kheddouci. A self-stabilizing algorithm for the minimum color sum of a graph. In *Proc. Distributed Computing and Networking (ICDCN), 9th Int. Conf., Kolkata, India, 2008*, volume 4904 of *LNCS*, pages 209–214. Springer, 2008.

[SGH04] Z. Shi, W. Goddard, and S. T. Hedetniemi. An anonymous self-stabilizing algorithm for 1-maximal independent set in trees. *Information Processing Letters*, 91:77–83, 2004.

[SOM04] Y. Sakurai, F. Ooshita, and T. Masuzawa. A self-stabilizing link-coloring protocol resilient to byzantine faults in tree networks. In *Proc. Principles of Distributed Systems (OPODIS), 8th Int. Conf., Grenoble, France, 2004, Revised Selected Papers*, volume 3544 of *LNCS*, pages 283–298. Springer, 2004.

[SP07] A. Dharwadker S. Pirzada. Applications of graph theory. *Korean Society for Industrial and Applied Mathematics, J.*, 11(4):19–38, 2007.

[SRR94] S. K. Shukla, D. J. Rosenkrantz, and S. S. Ravi. Developing self-stabilizing coloring algorithms via systematic randomization. In *Proc. Parallel Processing (IWPP), 1st Int. Workshop, Bangalore, India, 1994*, pages 668–673, 1994.

[SRR95] S. K. Shukla, D. J. Rosenkrantz, and S. S. Ravi. Observations on self-stabilizing graph algorithms for anonymous networks. In *Proc. Self-Stabilizing Systems (WSS), 2nd Workshop, Las Vegas, Nevada, 1995*, pages 7.1–7.15, 1995.

[SS92] S. Sur and P. K. Srimani. A Self-Stabilizing Distributed Algorithm to Construct BFS Spanning Trees of a Symmetric Graph. *Parallel Processing Letters*, 2:171–179, 1992.

[SS93] S Sur and P. K. Srimani. A self-stabilizing algorithm for coloring bipartite graphs. *Information Sciences*, 69:219–227, 1993.

[Suo11] J. Suomela. Survey of local algorithms. *ACM Computing Surveys*, 2011. To appear.

[SX07] P. K. Srimani and Z. Xu. Self-stabilizing algorithms of constructing spanning tree and weakly connected minimal dominating set. In *Proc. Distributed Computing Systems (ICDCSW), 27th Int. Conf. Workshop, Toronto, Canada, 2007*, page 3, 2007.

[Tel94] G. Tel. Maximal matching stabilizes in quadratic time. *Information Processing Letters*, 49(6):271–272, 1994.

[Tel01] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2nd edition, 2001.

[TG99] O. E. Theel and F. C. Gärtner. An exercise in proving convergence through transfer functions. In *Proc. Distributed Computing Systems (ICDCSW), 4^{th} Int. Conf. Workshop, Austin, Texas, 1999*, pages 41–47. IEEE Computer Society, 1999.

[The00] O. E. Theel. Exploitation of ljapunov theory for verifying self-stabilizing algorithms. In *Proc. Distributed Computing (DISC), 14^{th} Symp.,Toledo, Spain, 2000*, volume 1914 of *LNCS*, pages 209–222. Springer, 2000.

[Tix09] S. Tixeuil. *Algorithms and Theory of Computation Handbook, 2^{nd} Edition*, chapter Self-stabilizing Algorithms, pages 26.1–26.45. CRC Press, Taylor & Francis Group, 2009.

[TJH07] C.-H. Tzeng, J.-R. Jiang, and S.-T. Huang. A self-stabilizing $(\Delta + 4)$-edge-coloring algorithm for planar graphs in anonymous uniform systems. *Information Processing Letters*, 101(4):168–173, 2007.

[TS06] A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., 2006.

[Tur07] V. Turau. Linear self-stabilizing algorithms for the independent and dominating set problems using an unfair distributed scheduler. *Information Processing Letters*, 103(3):88–93, 2007.

[Tur12] V. Turau. Efficient transformation of distance-2 self-stabilizing algorithms. *Parallel and Distributed Computing, J.*, 2012.

[TW09] V. Turau and C. Weyer. Fault tolerance in wireless sensor networks through self-stabilization. *Communication Networks and Distributed Systems, J.*, 2(1):78–98, 2009.

[UT11] S. Unterschütz and V. Turau. Construction of connected dominating sets in large-scale manets exploiting self-stabilization. In *Proc. Localized Algorithms and Protocols for Wireless Sensor Networks (LOCALGOS), 5^{th} Int. Workshop, Barcelona, Spain, 2011*, 2011.

[vN56] J. von Neumann. Probabilistic logics and synthesis of reliable organisms from unreliable components. In *Automata Studies*, pages 43–98. Princeton University Press, 1956.

[WL99] J. Wu and H. Li. On calculating connected dominating set for efficient routing in ad hoc wireless networks. In *Proc. Discrete Algorithms and Methods for Mobile Computing and Communications (DIALM), 3^{rd} Int. Workshop, Seattle, WA, USA, 1999*, pages 7–14. ACM, 1999.

[WLZ08] Y. Wang, X.-Y. Li, and Q. Zhang. Efficient algorithms for p-self-protection problem in static wireless sensor networks. *Parallel and Distributed Systems, IEEE Trans.*, 19(10):1426–1438, 2008.

[WW04] M. Wattenhofer and R. Wattenhofer. Distributed Weighted Matching. In *Distributed Computing (DISC), 18th Ann. Conf., Amsterdam, The Netherlands, 2004*, pages 335–348, 2004.

[WZL07] D. Wang, Q. Zhang, and J. Liu. The self-protection problem in wireless sensor networks. *Sensor Networks, ACM Trans.*, 3, 2007.

[XHGS03] Z. Xu, S. T. Hedetniemi, W. Goddard, and P. K. Srimani. A synchronous self-stabilizing minimal domination protocol in an arbitrary network graph. In *Proc. Distributed Computing (IWDC), 5th Int. Workshop, Kolkata, India, 2003*, volume 2918 of *LNCS*, pages 26–32. Springer, 2003.

[XS05] Z. Xu and P. K. Srimani. Self-stabilizing anonymous leader election in a tree. In *Proc. Parallel and Distributed Processing (IPDPS), 19th IEEE Int. Symp., Denver, CO, USA, 2005*, page 207.1. IEEE Computer Society, 2005.

[XWS10] Z. Xu, J. Wang, and P. K. Srimani. Distributed fault tolerant computation of weakly connected dominating set in ad hoc networks. *Supercomputing, J.*, 53(1):182–195, 2010.

[Yag07] R. Yagel. *Self-Stabilizing Operating Systems*. PhD thesis, Ben-Gurion University of the Negev, Beersheva, Israel, 2007.

Author's Publications

[Hau08] Bernd Hauck. Worst-case analysis of a self-stabilizing algorithm computing a weakly connected minimal dominating set. Technical Report urn:nbn:de:gbv:830-tubdok-5126, Hamburg University of Technology, Hamburg, Germany, 2008.

[HT09a] Bernd Hauck and Volker Turau. A self-stabilizing algorithm for constructing weakly connected minimal dominating sets. *Information Processing Letters*, 109(14):763–767, 2009.

[HT09b] Bernd Hauck and Volker Turau. A self-stabilizing approximation algorithm for vertex cover in anonymous networks. In *Proc. Stabilization, Safety, and Security of Distributed Systems (SSS), 11th Int. Symp., Lyon, France, 2009*, volume 5873 of *LNCS*, pages 341–353. Springer, 2009.

[HT11a] Bernd Hauck and Volker Turau. A fault-containing self-stabilizing (3 - 2/(delta+1))-approximation algorithm for vertex cover in anonymous networks. *Theoretical Computer Science*, 412(33):4361–4371, 2011.

[HT11b] Bernd Hauck and Volker Turau. A new analysis of a self-stabilizing maximum weight matching algorithm with approximation ratio 2. *Theoretical Computer Science*, 412(40):5527–5540, 2011.