ingenieur
wissenschaften
htw saar

Hochschule für
Technik und Wirtschaft
des Saarlandes
University of
Applied Sciences

STL SYSTEMTECHNIKLABOR

Fast Self-Stabilizing Broadcast with 1 Bit

Niko Kleer

Technical Report – STL-TR-2017-04 – ISSN 2364-7167

stl.htwsaar.de

Technische Berichte des Systemtechniklabors (STL) der htw saar
Technical Reports of the System Technology Lab (STL) at htw saar
ISSN 2364-7167

Niko Kleer: Fast Self-Stabilizing Broadcast with 1 Bit
Technical report id: STL-TR-2017-04

First published: October 2017
Last revision: October 2017
Internal review: Klaus Berberich, Emanuele Natale

For the most recent version of this report see: https://stl.htwsaar.de/

Title image source: Flavio Takemoto (flaivoloka), http://www.freeimages.com/photo/1160571

htw saar – Hochschule für Technik und Wirtschaft des Saarlandes (University of Applied Sciences)
Fakultät für Ingenieurwissenschaften (School of Engineering)
STL – Systemtechniklabor (System Technology Lab)
Prof. Dr.-Ing. André Miede (andre.miede@htwsaar.de)
Goebenstraße 40
66117 Saarbrücken, Germany
https://stl.htwsaar.de

FAST SELF-STABILIZING BROADCAST WITH 1 BIT

NIKO KLEER

A thesis submitted for the degree of Bachelor of Science

in

Applied Computer Science

at

University of Applied Sciences, Saarbrücken

Hochschule für Technik und Wirtschaft des Saarlandes

September 2017

DECLARATION

I hereby declare that this thesis has not been previously accepted in substance of any degree and is not being concurrently submitted in candidature for any degree. I state that this thesis is the result of my own independent work/investigation, except where otherwise stated.

Saarbrücken, September 2017

Niko Kleer

Niko Kleer: *Fast Self-Stabilizing Broadcast with* 1 *Bit,* Bachelor of Science, © September 2017

SUPERVISORS:
Prof. Dr. Klaus Berberich
Dr. Emanuele Natale

LOCATION:
Saarbrücken

SUBMISSION:
September 2017

ABSTRACT

Although information propagation as well as self-stabilization represent fundamental problems in Computer Science, there exist only a few time-efficient algorithms for solving the self-stabilizing broadcast problem while restricting an agent to transmit small-sized messages. This thesis provides an extensive analysis of an information propagation protocol for solving the broadcast problem in a self-stabilizing context by transmitting 1-bit messages only. A theoretical investigation of the protocol yields a lower bound of $\Omega(\log n)$ on the convergence time while an experimental analysis via the randomized search heuristic simulated annealing shows the protocol's capabilities to self-stabilize from any arbitrary configuration in $\mathcal{O}(\log n)$ rounds.

ZUSAMMENFASSUNG

Obwohl Informationsverbreitung und Selbststabilisierung fundamentale Probleme der Informatik darstellen, existiert nur eine geringe Anzahl schneller Algorithmen zum Lösen des selbst-stabilisierenden Broadcast-Problems, die einen Vermittler auf die Übertragung kleiner Nachrichten beschränken. Diese Arbeit analysiert einen Algorithmus, der das Broadcast-Problem unter der Voraussetzung, dass jeder Vermittler lediglich über 1-Bit Nachrichten kommunizieren darf, in einem selbst-stabilisierenden Kontext löst. Eine theoretische Untersuchung bezüglich der Konvergenz resultiert in einer unteren Schranke von $\Omega(\log n)$ während eine experimentelle Analyse mit Hilfe des heuristischen Approximationsverfahrens simulierte Abkühlung zeigt, dass der Algorithmus jede beliebige Startkonfiguration in $\mathcal{O}(\log n)$ Runden selbst-stabilisieren kann.

We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

— Donald E. Knuth [27]

ACKNOWLEDGMENTS

I would like to start by expressing my sincere appreciation to my advisor Dr. Emanuele Natale who gave me the opportunity to write this thesis. His patience, encouragement and knowledge have truly been an enrichment to my motivation and the result of this thesis.

I acknowledge my gratitude to my advisor Prof. Klaus Berberich for his continous support throughout the course of this thesis. I am thankful for his guidance and advice in the scientific field.

My thanks also goes to the department for Databases and Information Systems of the Max Planck Institute for Informatics in Saarbrücken for providing me with resources to work on my thesis.

Special recognition to Martin Feick whose support has always been a considerable motivation in overcoming demanding challenges during my studies. I particularly thank him for the fun we had while spending our time on searching and fixing problems together. Thanks to Marek Kohn for his mathematical knowledge and problem solving skills that I was able to extend my knowledge with. I also thank Marius Backes for being a nice companion whenever I was taking the train over the last three years.

CONTENTS

Part I

INTRODUCTION

1 INSPIRATION BIOLOGY

Asking a computer scientist about the origin of many well-known algorithms most likely leads to answers like Edsger Dijkstra who invented the most popular shortest path algorithm [8], Joseph Kruskal who presented his greedy algorithm for finding a minimum spanning tree in a connected weighted graph [29], Thomas Bayes whose theorem considerably contributed to several areas in Computer Science [34] or John von Neumann who introduced the popular sorting algorithm merge sort [28]. Nevertheless, several computational algorithms were initially inspired by a factor that most computer scientists either would not think of in the very first moment or do not know about at all: *biological systems*.
These systems have been able to considerably inspire the conception of computational algorithms in different fields of Computer Science. For example, a concept used in machine learning that has steadily been gaining popularity are neural networks [20, 21]. They underlie fundamental neuroscientific knowledge which is why they were first investigated by neuroscientists and biologists. Genetic algorithms represent a heuristic for finding solutions to optimization problems that was initally inspired by biology as well [18]. Furthermore, by studying the behavior of insects, computer scientists have been able to learn about search optimization [6] as well as understanding network dynamics [13]. There are numerous reasons as to why the investigation of biological systems, to improve the effectiveness of computational algorithms, makes sense. In fact, computational as well as biological systems show quite a few similarities when it comes to their characteristics and requirements [35].

- Biological systems are distributed as they consist of several biological organisms interacting with each other. Distributed systems in Computer Science basically need to be able to do the exact same thing by handling heterogeneity. This means that the system's design needs to consider different hardware architectures, operating systems and programming languages.

- Being able to resist failures and attacks is just as important in biological as in computational systems. In a biological context, a term frequently used describing a system's ability to retain its functionality is robustness [26]. Computer Science also aims to design fault tolerant systems, especially when operated by humans.

- Another factor is communication. Multiple homo- or heterogenous systems that have their internal software components interact with each other are far from being rarities nowadays. Enabling this way of communication mostly increases the usability

of those systems in many cases. This improvement especially applies if communication between humans becomes possible. Networks are used to allow this kind of interaction in the first place. Biological systems make use of networks for communication purposes as well.

- A core principle in software engineering is modularity. Re-using code or certain components e. g. classes in object oriented programming (OOP) is mostly done by implementing design patterns as they guarantee a clean reutilisation [15]. Clean means that the software's maintainability, extensibility and readability does not suffer from that reutilisation. In fact, the opposite is the case as correctly implemented design patterns help to grow an architecture in an appropriate way, especially with respect to large software. Modularity can also be observed when looking at biological networks. They consist of independent modules that are reused as well. Patterns that are observed to be reused in these networks are called network motifs [24].

- Probabilistic algorithms in Computer Science exploit randomness to solve certain problems. Processes in biological systems are also sometimes stochastically influenced. This can be shown at the example of gene cells that have to constantly adapt to changing environments [22].

The similarities listed above clearly point out why the understanding of biological systems and processes has already contributed to several fields in Computer Science.
This chapter is divided into the following sections: we start by introducing a fundamental problem that is not only of considerable importance to Computer Science but can be observed in numerous biological settings as well, namely the broadcast problem. Section 1.2 continues by introducing the condition of self-stabilization which is a fault-tolerance concept in distributed computing for providing robustness in a system. This leads us to the self-stabilizing broadcast problem that emerges from combining both previously mentioned fundamental problems. Finally, Section 1.3 elaborates on the matter of majority consensus that enables decision-making in several biological settings as well as distributed computing under the influence of faulty processes. Consensus is not only of significance with respect to the protocol studied in this thesis but also commonly used for achieving system reliability in distributed computing.

1.1 THE BROADCAST PROBLEM

Computer networks are commonly known for using broadcasts to propagate data to all agents in the network. This is not only a concept used in Computer Science as there are scenarios related to the animal kingdom where propagating information plays a significant role. Ants interacting in theirs nest to recruit other ants to look for

food is only one example of a concrete broadcast in a biological setting [38]. Obviously, scenarios like that seem to be very noisy and highly unpredictable. The field of distributed computing has already introduced numerous communication models for investigating these scenarios computationally. The so-called uniform pull model, later referred as $\mathcal{PULL}$ (see Definition 4.1 in Chapter 4), representing only one of many options, is going to be the communication model considered in this thesis for tackling the broadcast problem. In this model of communication, time proceeds in discrete rounds and an information is supposed to spread among a population of n agents. In each round, each agent observes and copies the information from another uniformly at random chosen agent (including itself).

1.2 SELF-STABILIZING BROADCAST

A condition arising considerable difficulties when solving the broadcast problem is self-stabilization. When solving this problem in a self-stabilizing context, the process has to converge to a valid state from any initial configuration [9]. Even though this is the case, note that an initial configuration must not ignore the definition of the broadcast problem (see Definition 4.2 in Chapter 4) which means that there has to be exactly one source in the initial configuration. Self-stabilization still leads to a decisive issue as an agent can never be sure about whether its current information matches the information propagated by the source. Consequently, an initial configuration that leads to the wrong information to be propagated is unable to converge. Ignoring the self-stabilization constraint, Karp et al. [23] have shown that the problem can be solved in $\mathcal{O}(\log n)$ rounds with high probability by using a certainty bit indicating whether an agent's information corresponds to the information propagated by the source. Since self-stabilization is an essential requirement in our case, we are looking for an algorithm that solves the problem in a self-stabilizing context.

1.3 MAJORITY CONSENSUS

In numerous biological settings, including the example given in Section 1.1, animals and other biological individuals have to come to a decision. Another typical biological setting where this kind of decision making plays a significant role can be observed with respect to fish. They have to quickly reach consensous in their group to agree on what to do when attacked by a predator [40]. The algorithm that we propose in Chapter 3 uses a majority consensus protocol as a subprocedure for solving the self-stabilizing broadcast problem. This protocol proceeds in the following way: each agent has an internal output bit that can be observed by an arbitrary number of agents. In each round of the process, each agent observes the output bit provided by two agents and changes its output bit corresponding to the majority of bits observed by the agent.

1.4 RESULTS

This thesis' main contribution represents a comprehensive analysis regarding the self-stabilizing property of a concrete protocol for solving the self-stabilizing broadcast problem. The protocol that we analyze is of special interest as it restricts agents to communicate via 1-bit messages only. First, we theoretically investigate rigorous bounds regarding the convergence time of the protocol. We do so by moving our focus to the broadcast problem and prove a lower bound of $\Omega(\log n)$ on the convergence time of the problem. After that, we provide an extensive experimental analysis for showing that our protocol for solving the self-stabilizing broadcast problem converges in $\mathcal{O}(\log n)$ rounds.

Part II

RELATED WORK

2

RELATED WORK

In this chapter, we will take a look at the most important related work regarding the topic of this thesis. The first section will present several communication models that have previously been investigated for solving the broadcast problem. After talking about the essentials concerning the concept of self-stabilization in Section 2.2, we will proceed by discussing an aspect that represents a combination of the fundamental problems mentioned before, the self-stabilizing broadcast. Finally, majority consensus and its relation to the self-stabilizing broadcast will be subject of Section 2.4.

2.1 THE BROADCAST PROBLEM

The broadcast problem, sometimes referred as rumor spreading or information dissemination problem, being one of many fundamental problems in computer science, has already been studied extensively. There are several communication models similar to $\mathcal{PULL}$ (see Definition 4.1 in Chapter 4) that have already been considered and analyzed for solving the broadcast problem. A model whose computation efficiency on unknown graph topologies has been investigated is the $\mathcal{GOSSIP}$ model [4] of communication. In this model, information is propagated by allowing each node to establish at most one bidirectional communication with another neighbor in each round. As this model aims to solve the broadcast problem on any graph topology, an additional algorithm is required for choosing a node to communicate with. A different way of communication can be observed in the $\mathcal{LOCAL}$ as well as the $\mathcal{CONGEST}$ model [37]. Both of these models enable each node to propagate its information to all of its neighbors in each round. In contrast to the $\mathcal{GOSSIP}$ model, each node may arbitrarily manipulate the data propagated. Note that only the $\mathcal{CONGEST}$ model restricts the message size that can be transmitted by a node to k-bits where k is usually chosen to be $\log n$. Another similar model investigated by Karp et al. [23] is the random phone call model where each agent u chooses a communication partner v in each round uniformly at random. This model also considers bidirectional communication in which an information is pushed or (not exclusively) pulled, depending on the state of an agent. Since the model assumes agents to be chosen uniformly at random, it can be referrerd to as a uniform $\mathcal{GOSSIP}$ model. The aforementioned as well as many other communication models [14] and well-studied algorithms [7, 10] share the fact that they use randomization and limited communication for providing simplicity and robustness while solving the broadcast problem.

2.2 SELF-STABILIZATION

The most important contribution regarding the self-stabilizing property that requires a system to be in a valid state at any time was published by Edsger Dijkstra [9]. His paper contributed to the area of distributed computing by presenting the first system independent self-stabilizing algorithms. His research effort was later recongnized by Leslie Lamport who emphasized Dijksta's work in 1985 [32]. Even more improvements such as error detection through local checking [41] as well as time-adaptive protocols [30] followed. Another related topic to self-stabilization is superstabilization [12]. In contrast to self-stabilization, superstabilizing algorithms are capable of recovering from changes in the network topology.

2.3 SELF-STABILIZING BROADCAST

By combining the fundamental problems previously discussed, we get a problem that has steadily been gaining attention over the last few years, namely the self-stabilizing broadcast. There have already been theoretical investigations for each communication model in Section 2.1 regarding the broadcast problem. Consequently, lower as well as upper bounds on the convergence resulted from those investigations. Unfortunately, the self-stabilization condition arises several difficulties such as making sure that agents are unable to lead the system into an invalid state by propagating wrong information. That is why protocols solving the broadcast problem in a self-stabilizing context mostly rely on synchronization measures [31]. In contrast to previously investigated protocols, Boczkowski et al. [3] present the first algorithm that focuses on restricting the message size to 3 bits. This thesis aims to minimize the message size even further and investigates an algorithm that uses 1-bit messages only.

2.4 MAJORITY CONSENSOUS

Distributed as well as multi-agent systems have to agree on data for making a decision in many cases. However, those systems are vulnerable to failures such as crash, timing, omission or Byzantine failures [33]. Since achieving system reliability is a crucial task in distributed computing, consensus represents another fundamental problem in computer science that has already enjoyed comprehensive research. Consensus protocols have to be fault-tolerant and aim to be fast in terms of running time [1, 2]. Minimizing message traffic plays a significant role as well which is why the majority consensus protocol that is used as sub-procedure for solving the self-stabilizing broadcast problem in this thesis focuses on transmitting 1-bit messages.

Part III

PROBLEM STATEMENT

3

THE BSF-PROTOCOL

An algorithm that solves the self-stabilizing broadcast problem should fulfill three requirements.

- The algorithm should be as **simple** as possible. This can not be measured since simplicity is usually based on common sense. Consequently, there should not be the need for a measure to decide whether an algorithm is simple or not.
- **Runtime** plays a significant role. In other words: the amount of time needed by the algorithm to converge to a valid state. We can do this by measuring an algorithm's runtime complexity.
- The **message size** transmitted by the algorithm in relation to each agent should be as small as possible.The reason for this is directly related to computer networks. Transmitting large messages is normally done by splitting the data into packets. A fundamental problem that applies in biological settings as well is the following: how can one prevent data loss during the transmission as there has to be a confirmation on whether each part of the message reached its destination. That is why an algorithm should restrict agents to transfer small messages only.

A concrete candidate for solving the self-stabilizing broadcast problem is the *Boosting-Susceptible-Frozen* (BSF) protocol provided by [3]. The protocol satisfies each requirement mentioned above. Moreover, it defines the following three states and includes the parameters l and k.

- Agents in the **boosting** state simply perform the majority consensus rule. Moreover, each agent has an individual counter T that keeps track of the number of rounds an agent has not changed its opinion. If an agent has observed only agents with the same opinion b for l rounds, the agent becomes susceptible to $1 - b$.
- Agents **susceptible** to b change their opinion as soon as they observe an agent with opinion b. This turns them into Frozen-b agents.
- **Frozen**-b agents keep their opinion b for k rounds. After this duration, they restart boosting.

Pseudocode for applying the algorithm just presented on a set of n agents where a_k represents the agent currently observed by the algorithm is given below in Algorithm 1.

Algorithm 1 BSF-Protocol

```
1: procedure BSF(l,k)
2:     if a_k is boosting then
3:         observe two agents
4:         if both agents agree with a_k on the same opinion then
5:             T_b ← T_b + 1
6:             if T_b ⩾ l then
7:                 a_k becomes susceptible to 1 − b
8:                 T_b ← 0
9:             end if
10:        else if both agents disagree with a_k then
11:            a_k's opinion changes to 1 − b
12:        else
13:            T_b ← 0
14:        end if
15:    else if a_k is susceptible then
16:        observe one agent
17:        if the agent disagrees then
18:            a_k's opinion changes to 1 − b and a_k freezes
19:        end if
20:    else if a_k is frozen then
21:        T_f ← T_f + 1
22:        if T_f ⩾ k then
23:            a_k starts boosting
24:            T_f ← 0
25:        end if
26:    end if
27: end procedure
```

3.1 THESIS OBJECTIVE

The objective of this bachelor thesis is divided into a theoretical as well as an experimental part.

1. **Theoretical part:** The first part of this thesis investigates a lower bound regarding the convergence time of the algorithm presented in this chapter.

2. **Experimental part:** Completing this thesis, the experimental part includes implementing an efficient framework that allows to test the self-stabilizing property of the BFS-protocol by searching for initial configurations that give the algorithm a hard time to converge. The programming language C++ is used to implement the framework while searching for hard initial configurations will be done by implementing the randomized search heuristic simulated annealing. By doing so, we want to show that the algorithm takes no longer than $\mathcal{O}(\log n)$ rounds to converge to a valid state.

Part IV

THEORETICAL INSIGHTS

4

ON THE CONVERGENCE OF THE ALGORITHM

From a theoretical point of view, we want to analyze the convergence of the previously presented candidate algorithm. Therefore, we are interested in finding bounds concerning the minimal as well as the maximal number of rounds the algorithm needs to converge. We can distinguish between two scenarios: a configuration either favors the convergence of the algorithm or not. These scenarios are subsequently explained.

- **Optimistic**: Initial configurations that favor the convergence of the algorithm are expected to converge to a valid state right away.
- **Pessimistic**: On the other hand, when starting in an unfavorable configuration, the population is expected to diverge in such a way that the entire population disagrees with the source's opinion at some point. While the population keeps on boosting, all agents become susceptible to the opinion propagated by the source after some time. Agents taking on that opinion become frozen which increases the probability for other agents to freeze and accelerates the spreading process with every single agent. This behavior should eventually lead to a quickly converging algorithm.

However, before we can give any mathematical details, the communication model [11] needs to be defined.

Definition 4.1. *The uniform pull model, denoted as $\mathcal{PULL}$, defines a model of communication on a population of n agents in which time proceeds in discrete rounds. There is a specific part of each agent's memory that will be referred as the observable part. In each round, each agent u observes one uniformly at random chosen agent v from the entire population, including herself. Observing an agent v means that u may look at the observable part of v's memory. In case there are multiple agents $u_1, ..., u_k$ with $1 < k \leqslant n$ observing an agent v in the same round, they observe the exact same memory.*

In this chapter we aim to determine a lower bound concerning the convergence time of the BSF-ALGORITHM. Since analytically investigating the algorithm is rather complicated as we would be required to consider any possible scenario, we will continue by considering another problem that resembles the pessimistic scenario described at the beginning of this chapter, namely the broadcast problem. We will consider this more general problem for providing a lower bound showing that the algorithm requires logarithmic time to converge. The reason of the broadcast problem being relevant in relation to the convergence of the BSF-ALGORITHM is the following: Consider a population in which there are $\frac{n}{2}$ agents agreeing as well as disagreeing with the

source's opinion. In case this population fails to agree on the opinion propagated by the source, all agents will disagree at some point. Following the BSF-ALGORITHM, this will lead all agents to become susceptible to the source's opinion. Finally, all of them will freeze. Choosing the parameter k large enough results in all agents to be frozen at the same time. Since the broadcast problem is the problem of showing the amount of time required for an information to spread among a population, the configuration just described is identical to the starting configuration required by the broadcast problem. Therefore, this problem can be considered to be a likely case during the execution of the BSF-ALGORITHM. We define the broadcast problem in Definition 4.2.

Definition 4.2. *The following process in $\mathcal{PULL}$ defines the broadcast problem. At the beginning of the process, the population comprises a set of uninformed agents $U_t = \{u_1, u_2, ..., u_n\}$ as well as an informed agent that represents the population's source S. Whenever an uninformed agent $u_i \in U_t$ observes an informed one, it turns informed. Furthermore, an informed agent keeps on being informed meaning that she is not influenced by future observations. The process has not converged and keeps on proceeding as long as $U_t \neq \emptyset$.*

In contrast to the algorithm, an analytical investigation of the broadcast problem is much easier as we only concentrate on a specific behavior instead of any possible case. Consequently, an analysis of the broadcast problem allows gaining insights on the convergence of the BSF-ALGORITHM.

4.1 INVESTIGATING A LOWER BOUND

Theorem 4.1. *The expected convergence time of the broadcast problem is $\Omega(\log n)$.*

Proof. In order to prove, we assume that each uninformed agent has opinion 0. Definition 4.2 implies that the probability for any agent $u_i \in U_t$ to take on S opinion in the first round equals $\frac{1}{n}$ since S is the only informed agent. Moreover, an informed agent is not capable of taking on opinion 0. Let $Y_t(u_i)$ denote a boolean random variable that takes on value 1 if the agent u_i sees an informed agent in round t. By considering the aforementioned facts, we can say that the expected value for any $u_i \in U_t$ to turn informed in round one is given by

$$\begin{aligned} E[Y_1(u_i)] &= P(Y_1(u_i) = 1) + 0 \cdot P(Y_1(u_i) = 0) \\ &= \frac{1}{n}. \end{aligned} \tag{1}$$

Let X_t represent the fraction of agents that is informed in round t. By using Equation 1, we can determine the expected fraction of agents that turns informed in round one with respect to the entire population. Since $E[X_1]$ has to consider each uninformed agent's probability $P(Y_1(u_i) = 1)\ \forall u_i \in U_t$, we have to sum up the probabilities and

divide the result by the number of agents leading to the following equation:

$$\begin{aligned} E[X_1] &= E[\frac{1}{n} \sum_{u_i \in U_t} Y_1(u_i)] \\ &= \frac{1}{n} \sum_{u_i \in U_t} E[Y_1(u_i)] \\ &= \frac{1}{n} \sum_{u_i \in U_t} P(Y_1(u_i) = 1) \\ &= \frac{1}{n} \sum_{u_i \in U_t} \frac{1}{n}. \\ &= \frac{1}{n}. \end{aligned} \tag{2}$$

We can adapt Equation 2 to determine the expected fraction of agents that turns informed in round t. Note that the value of $E[X_t]$ is based on every single round that has previously been executed. Since the broadcast problem proceeds in discrete rounds, we can look at $E[X_t]$ based on the number of agents that have already been informed up until round $t-1$, namely X_{t-1}. Therefore, we may use the conditional expectation $E[X_t|X_{t-1}]$. This means that we need to add the number of agents previously informed to the number that is expected to turn informed in round t. In contrast to Equation 2, we now need to consider the probabilities of all agents to turn informed based on round $t-1$. Keep in mind that the summation that considers any $u_i \in U_t$ divided by n ends up being $1 - X_{t-1}$ which is why we can transform the summation. As a result,

$$\begin{aligned} E[X_t|X_{t-1}] &= X_{t-1} + \frac{1}{n} \sum_{u_i \in U_t} E[Y_t(u_i)|X_{t-1}] \\ &= X_{t-1} + \frac{1}{n} \sum_{u_i \in U_t} P(Y_t(u_i) = 1|X_{t-1}) \\ &= X_{t-1} + \frac{1}{n} \sum_{u_i \in U_t} X_{t-1} \\ &= X_{t-1} + \frac{1}{n}|U_t| \cdot X_{t-1} \\ &= X_{t-1} + (1 - X_{t-1})X_{t-1} \\ &= 2X_{t-1} - X_{t-1}^2 \end{aligned} \tag{3}$$

can be concluded to be the expected fraction of agents turning informed in round t given the information that we have from round $t-1$. By applying the law of total expectation, stating that $E[X_t] = E[E[X_t|X_{t-1}]]$, we can get an upper bound for $E[X_t]$.

$$\begin{aligned} E[X_t] &= E[E[X_t|X_{t-1}]] \\ &= E[2X_{t-1} - X_{t-1}^2] \\ &\leqslant 2E[X_{t-1}] \leqslant 4E[X_{t-2}] \leqslant \ldots \leqslant 2^t E[X_1] \\ &= \frac{2^t}{n}. \end{aligned} \tag{4}$$

Since the upper bound only assumes positive values, we are allowed to use Markov's Inequality

$$P(X \geqslant a) \leqslant \frac{E[X]}{a} \quad \forall a > 0$$

for a real-valued positive random variable to determine a lower bound on the probability for the broadcast problem to converge in $\Omega(\log n)$. Evaluating Markov's Inequality for $t = \frac{1}{2} \log n$ and $a = 1$ leads to

$$\begin{aligned} P(X_{\frac{1}{2}\log n} \geqslant 1) &\leqslant \frac{2^{\frac{1}{2}\log n}}{n} \\ &\leqslant \frac{1}{\sqrt{n}} \end{aligned}$$

showing that $\Omega(\log n)$ is indeed a lower bound for the convergence time of the broadcast problem with high probability. □

Part V

IMPLEMENTATION

5 REQUIREMENTS

The BSF-ALGORITHM's implementation can be considered to be rather simple as neither a graphical user interface nor a complex software architecture need to be implemented. Nevertheless, there are some basic requirements that should be taken into account.

1. **Keep it simple:** Just like the algorithm itself, the implementation is supposed to be kept as simple as possible. This is obviously not only a desirable aspect in this particular case but in software development in general. There might also be cases in which a seemingly difficult function has to be implemented. In situations like that, implementing the first idea about solving a problem right away often turns out to be a bad approach. However, reconsidering that idea mostly leads to finding a simplified variation. As a result, simplifying complicated procedures helps to write code that can easily be understood, maintained and documented.

2. **Optimized runtime:** Being able to investigate the algorithm's behavior in relation to networks with millions of agents is another desirable aspect. This can not be achieved by simply using high-end hardware components like a 16-core CPU and 128 GB of RAM. To enable these large scale network investigations, optimizing the algorithm's runtime by keeping the amount of unnecessary code as small as possible and implementing mechanisms that accelerate its execution always have to be considered. A mechanism frequently used to achieve runtime improvements is *multi-threading*.

This chapter provides a section that covers each aspect of the implementation. The first section gives an overview on the self-stabilizing broadcast problem's implemented architecture. After that, we take a more detailed look at each component's functions and data structures. Finishing the algorithm's basic implementation, Section 5.3 gives more detailed explanations on the randomized search heuristic *simulated annealing* (SA). Finally, Section 5.4 elaborates on several threading examples and takes into consideration whether they are beneficial or not.

5.1 SOFTWARE ARCHITECTURE

As previously mentioned, implementing an efficient framework to test the self-stabilizing property of the BSF-ALGORITHM does not require a complex software architecture. In fact, the problem can be translated into a really simple architecture using only three major classes, excluding `Main`, as shown by Figure 1.

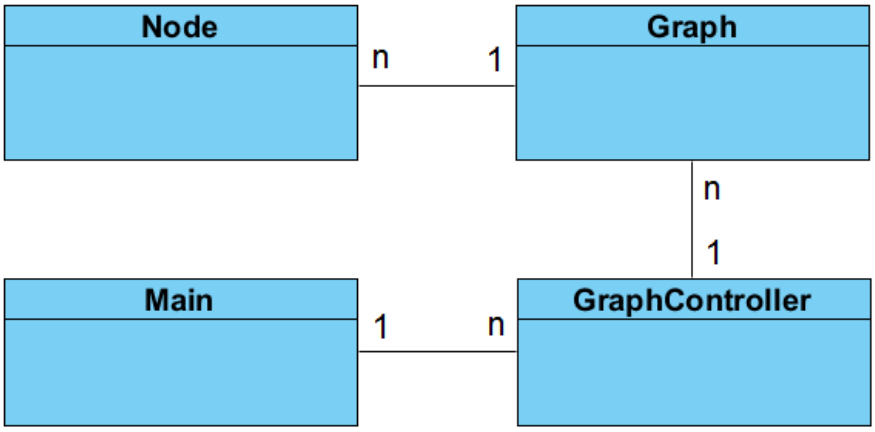

Figure 1: Shows the implementation's class diagram

- `Node`: A really simple class representing an agent in the self-stabilizing broadcast problem where each agent belongs to exactly one `Graph`.
- `Graph`: This class represents the self-stabilizing broadcast problem's model including a population of n `Nodes` as well as the BSF-ALGORITHM. A `Graph` object is furthermore created by precisely one `GraphController`.
- `GraphController`: While the classes just mentioned are responsible for providing the core functionality of this framework, a class that enables the execution of additional procedures is needed as well. Moreover, we also want to be able to work with the data resulting from testing the broadcast problem's self-stabilizing property. The most important part of this class is therefore going to be an implementation of the randomized search heuristic simulated annealing.

Since this section has merely introduced the application's components superficially, the next section gives more detailed explanations on how these classes are implemented.

5.2 IMPLEMENTING THE BSF-PROTOCOL

Each of the presented classes implements properties or certain data structures to manage their data. This section takes a closer look at these implementations and gives explanations as to why specific procedures have been chosen to be implemented as they are.

5.2.1 *The Node Class*

Being able to implement this class appropriately requires recalling the BSF-ALGORITHM described in Chapter 3. First off, a `Node`'s properties include an opinion as well as state that corresponds to the BSF-protocol. Furthermore, a `Node` needs to keep track of how many rounds it has already been boosting or frozen. Besides that, a `Node`

also needs to be able to observe two agents as implied by the majority consensus protocol. The opinion and individual counters T_b and T_f are rather small numbers which is why using an integer is absolutely sufficient. Using a `char` for storing a `Node`'s state is reasonable as it improves code readability in contrast to a different data type. Finally, `Nodes` currently observed are obviously pointers. For the purpose of simplicity, Table 1 lists the aforementioned properties.

Table 1: Lists and describes a `Node`'s properties

Property	*Datatype*	*Description*
state	`char`	There are three different possibilities indicating a `Node`'s state: b: boosting s: susceptible f: frozen
opinion	`integer`	A binary property that can be either 0 or 1
boostCounter	`integer`	Counts how many rounds a `Node` has already been boosting
frozenCounter	`integer`	Counts how many rounds a `Node` has already been frozen
firstNeighbour	`Node*`	Represents the first `Node` currently observed
secondNeighbour	`Node*`	Represents the second `Node` currently observed

This class does not need to provide any substantial functionalities as the `Graph` class takes care of that part.

5.2.2 *Appropriately Implementing the Graph Class*

The `Graph` class is supposed to provide the core functionalities for executing computational experiments. Before we take a closer look at the implementation, let us first consider how a `Graph` is commonly implemented.

- **Adjacency List**: A very common approach to implement a `Graph` includes using an adjacency list that stores each `Node` and its corresponding neighbors. As this implementation only requires $\mathcal{O}(V + E)$ memory space where V represents the number of vertices and E the number of edges in the `Graph`, it is recommendable in many scenarios that require a `Graph` to be implemented.
- **Adjacency Matrix**: Using an adjacency matrix containing entries about the `Graph`'s topology represents an alternative. Since the matrix is quadratic, storing only the topology already requires $\mathcal{O}(V^2)$ memory space. As a result, implementing a `Graph`

this way might be worse if there is no particular reason requiring an adjacency matrix.

However, the broadcast problem is a **complete graph**. Without any doubt, using an associative array to store the `Nodes` is sufficient as each `Node` has a uniform chance to observe any other `Node` in the `Graph`. Consequently, a data structure for storing a specific topology is not required leaving the implementation at $\mathcal{O}(V)$ memory usage.
So far, only the `Graph`'s implementation itself has been clarified. Obviously, data extracted from the self-stabilization process require data structures as well. Since these experiments aim to find hard initial configurations, we optimally want to gather all data about these configurations. This includes the number of individual opinions, states and counter values T_b and T_f. Moreover, this class has to define the value of the parameters l and k that are needed to execute the BSF-ALGORITHM. Storing the number of individual values can be done elegantly by using maps as they enable storing tuples. An individual value and its quantity also represent a tuple which is why all previously mentioned individual values can be stored in a map. Similar to a `Node`'s individual counter values, integers serve as data types to store l and k. Structuring everything that has been said in this section by now, Table 2 lists the `Graph`'s properties. Note that this class heavily relies on *standard template library* (STL) containers that provide an easy way for managing memory.
Completing the `Graph`'s implementation, let us move on to its functionalities. Since this class is responsible for providing the core functionalities of this application, there is a large sequence of procedures that need to be executed to successfully run computational experiments.

Table 2: Lists and describes a `Graph`'s properties

Property	*Datatype*	*Description*
nodes	`vector<Node>`	Stores the `Graph`
nodeStates	`map<char,int>`	Stores the quantity of individual states
nodeOpinions	`map<int,int>`	Stores the quantity of individual opinions
boostCounters	`map<int,int>`	Stores the quantity of individual T_b
frozenCounters	`map<int,int>`	Stores the quantity of individual T_f
maxBoostCounter	`integer`	Corresponds l
maxFrozenCounter	`integer`	Corresponds k

1. First, a `Graph` is instantiated by passing n, the number of `Node`'s, to the constructor. After that, the constructor needs to initialize the `Graph`'s properties. Filling the `Graph` with `Nodes` is

done by generating randomly determined opinions based on a Bernoulli distribution. Furthermore, a `Node` always finds itself in the boosting state after its initialization. As long as the `Graph` is not fully initialized, determining `Nodes` that are being observed is neither required nor guarantees a uniform `Node` determination at this point. During the initialization, the constructor also has to increase the generated opinion's and chosen state's individual number in the corresponding map. Algorithm 2 shows how the `Graph` initialization could be implemented. Note that the `initProperties` function is trivially implemented by simply initializing each data structure with 0. Therefore, the function will not be described in greater detail.

Algorithm 2 `Graph` initialization

```
1:  procedure GRAPH(n)
2:      Seed random number generator
3:      Call initProperties(n)
4:      i ← 0
5:      while i < n do
6:          b ← unif{0,1}                          ▷ Generate an opinion
7:          Node n_i ← Node(b,'b',0,0)             ▷ Create Node
8:          f(b) ← f(b) +1                         ▷ Update opinion map
9:          f('b') ← f('b') +1                     ▷ Update state map
10:         V[i] = n_i                             ▷ Insert Node
11:     end while
12: end procedure
```

2. After initializing the `Graph`'s properties, the next step includes implementing the BSF-ALGORITHM. Before discussing the details, there is one more important information that we need to be careful about. Recall that a `Graph` contains exactly one source whose opinion is supposed to be propagated and set to a fixed value. This problem is solved by skipping V[0] in each round of the algorithm's execution as the implementation always expects the source to be there. Consequently, the very first `Node`'s opinion in the `Graph` never changes. After skipping the source, the implementation samples two `Nodes` chosen uniformly at random for each `Node` in the `Graph`. As soon as each `Node`'s neighbors are set, the BSF-ALGORITHM can be executed.
   In case a `Node` is boosting, there are only two major conditions that need to be checked. Consider `Node` n_i to have opinion b. n_i's opinion changes, if the majority of output bits currently observed does not equal b. This means that n_i's neighbors both need to have the same opinion $1-b$. On the other hand, n_i's boost duration T_b increases if both neighbors have opinion b. Any other case results in T_b being reset to 0. Moreover, T_b must not exceed l meaning that as soon as T_b equals l, n_i becomes susceptible to $1-b$. `Nodes` susceptible to b change their opinion and become frozen as soon as they see opinion b. In this case, `Nodes` only look at exactly one of their neighbors

opinions. If that particular opinion equals b, the corresponding `Node` freezes. Concerning frozen `Nodes`, this part of the algorithm is called as long as the freeze duration T_f does not exceed k. While this is not the case, T_f is increased. As soon as T_f equals k, n_i restarts boosting. In addition to the algorithm itself, at any point of its execution, we want to track the quantity of individual states and opinions in the `Graph`. This means that each map needs to be updated appropriately. Some auxiliary functions can be used to accomplish these changes as well as retain code readability. Since these function's implementations are trivial taking an old and a new value as their parameters, they do not require any explanations. Listing all the above mentioned operations would be too complicated. That is why the notations $B(n_i)$, $S(n_i)$ and $F(n_i)$ used in Algorithm 3 are supposed to denote all operations related to n_i's corresponding state that have to be executed for n_i to complete one round of the BSF-ALGORITHM's execution.

Algorithm 3 BSF-ALGORITHM

```
1:  procedure SELFSTABILIZE
2:      c ← 0
3:      i ← 1                                   ▷ Skip the Zealot
4:      while f(Opinion(V[0])) < n do           ▷ Check if self-stabilized
5:          Call sample()                       ▷ Sample neighbors
6:          while i < n do                      ▷ Iterate over Nodes
7:              if State(n_i) == 'b' then
8:                  B(V[i])
9:              else if State(n_i) == 's' then
10:                 S(V[i])
11:             else if State(n_i) == 'f' then
12:                 F(V[i])
13:             end if
14:             i ← i + 1
15:         end while
16:         c ← c + 1                           ▷ Completed one round
17:         i ← 1                               ▷ Restart with first Node
18:     end while
19:     return c
20: end procedure
```

3. Finally, the number of rounds resulting from Algorithm 3 can be used to make a statement on the convergence time of the algorithm. However, running the application more than once leads to the same initial configurations to be self-stabilized over and over. With respect to these configurations, each result is based on a uniform opinion distribution and exclusively boosting `Nodes`. This means that the next step towards finding hard initial configurations requires implementing a procedure that automatically searches for these configurations by utilizing this implementation.

5.2.3 *Preparing the GraphController Class*

At this point of the implementation, this class only needs to instantiate a `Graph` and run the self-stabilization algorithm. One could argue that a task this simple does not require to be executed in an additional class. The argument is reasonable as this could be done in the `Main` module as well. However, as mentioned in Section 5.2.2, this implementation still requires a procedure that considerably increases its benefit. As this procedure is going to be rather complicated, providing another class for its implementation is more reasonable. Nevertheless, neither the properties nor any specific functions needed by this class can be determined yet. The next section does not only focus on the aforementioned procedure but gives more detailed explanations on the requirements of this class as well.

5.3 SIMULATED ANNEALING

Optimization problems in computer science mostly require a strategy that provides a decent approximation of an optimal solution. That is because a brute-force solution is infeasible in most cases, especially with respect to problems that can not be solved in polynomial runtime. Let C denote the number of possible configurations, note that self-stabilizing any initial arbitrary configuration using our candidate algorithm would require the self-stabilization of

$$C = 2(6\lfloor \log^2(n) \rfloor)^{n-1} \tag{5}$$

configurations, assuming that l and k are both set to $\log(n)$.

Proof. Each `Node` n in `Graph` G can either have opinion 0 or 1 and be in one out of three states. Moreover, each n's individual counter values T_b and T_f can only be set to a value α where $0 \leqslant \alpha < \lfloor \log(n) \rfloor \ \forall \alpha \in \mathbb{N}$. Applying basic combinatorics, this configuration space leads to $(2 \cdot 3\lfloor \log^2(n) \rfloor)^n$ possible configurations. However, setting the source's state and counter values is neither required nor sensible in the first place. Consequently, only its opinion needs to be considered which leads to $n-1$ `Nodes` with the exact same number of parameters that can arbitrarily be combined. Furthermore, only the source ends up with exactly two configurations from which we can conclude the final result. □

Consequently, a brute-force solution is clearly not an option. Simulated annealing [25] is a randomized search heuristic that allows approximating optimization problems where considering every single solution is computationally infeasible. The following steps describe the idea behind the heuristic.

1. Choose an initial energy state T_0, also called *temperature*. This value highly depends on the problem and has a considerable impact on the entire simulation. Since determining the temperature requires running the self-stabilization algorithm, chapter

6 is going to show how T_0 is determined in relation to this particular problem.

2. Calculate a randomly generated solution s. The function returning this solution is called cost function $c(x)$. Concerning our problem, the BSF-ALGORITHM represents the cost function. Furthermore, when speaking of a solution, we always refer to the number of rounds resulting from the algorithm.

3. Generate a neighbor configuration and calculate solution s' resulting from $c'(x)$. As it is impossible to give a concrete definition regarding the term neighbor configuration in general, the process of generating such a solution is not trivial at all. For example, a neighbor configuration could involve the number of individual opinions, states or counter values to be changed. This means that the challenge in generating a new configuration lies in changing an appropriate quantity of `Nodes` while keeping the process computable in terms of the algorithm's runtime.

4. If $s' > s$, we found a better[1] solution and s is set to s'. If this is not the case, we check the so-called acceptance probability $e^{\frac{s'-s}{T_k}}$ where T_k denotes the temperature in round k that determines whether s' is still accepted as the new solution. Should the resulting probability be greater than a randomly generated value $r \in [0, 1]$, s' is accepted even though there is no improvement in the new result. Since there is a chance that the simulation could be stuck in local minima at some point, this mechanism is supposed to provide a way for escaping them.

5. Update the temperature. There are several strategies [36] that can be applied to reduce its value. Optimally, the process updating T_k should guarantee a well scaling temperature with respect to the acceptance probability. More precisely, at the beginning of the simulation, there should be a high probability for bad solutions to be determined as the new solution. However, as the simulation proceeds, the probability should converge to 0 at some point. As this process is at least as complex as determining an appropriate temperature, Chapter 6 provides a separate section dedicated to present several ways on how to deal with this issue.

6. Repeat step three to five as long as T_k is non-negative or has not reached $0 + c$ for some problem-dependent constant c.

Algorithm 4 shows the entire procedure explained above. This section continues by presenting strategies for generating neighbor configurations. After that, Section 5.3.2 discusses how the simulation's basic concept needs to be adjusted for calculating sensible results. Since the difficulties arising from determining the initial temperature and updating T_k require the execution of experiments, neither of them will be addressed in this section.

1 If this was a minimization problem, a better solution would correspond to $s' < s$

Algorithm 4 Simulated annealing

```
1: procedure SIMULATE
2:     Set T_k
3:     s = c(x)                                  ▷ Calculate solution
4:     while T_k > 0 + c do
5:         Generate neighbor
6:         s' = c'(x)                            ▷ Calculate neighbor solution
7:         if s' > s then                        ▷ Compare solutions
8:             s = s'
9:         else if e^((s'−s)/T_k) > rand(0,1) then   ▷ Check acceptance prob.
10:            s = s'
11:        end if
12:        Reduce T_k
13:    end while
14:    return s
15: end procedure
```

5.3.1 *Generating Neighbor Configurations*

First off, the process of generating neighbor configurations requires strategies for adjusting a `Graph`'s parameters. In our case, this includes the individual number of opinions, states and counter values. In an optimal scenario, strategies adjusting these parameters cover a well distributed configuration space that can be explored by the simulation. This can be achieved by using and combining several probability distributions.

Starting with the opinion parameter, remember that there are exactly $n = f(0) + f(1)$ opinions in the `Graph`. Let opinion $b \in \{0, 1\}$. Appropriately adjusting the individual number of opinions can be done by either reducing or increasing $f(b)$ which, at the same time, leads to change in the quantity of $f(1-b)$. A probability distribution that is perfectly suitable for achieving such a change is the *normal distribution*. A concrete strategy for changing the number of opinion b in Graph G where b is randomly determined is the following: The number of b's added or subtracted is based on a value generated by a normal distribution $\mathcal{N}(\mu = 0, \sigma)$ where each random number r is generated around the mean μ with standard deviation σ. Figure 2 shows such a distribution. Note that r must neither under- nor overflow $f(b)$ as this would cause the application to crash. Also, recall that the source's opinion must not change. By setting $\mu = 0$, there is no need to worry about a value to be added or subtracted. Otherwise, an additional mechanism for making that

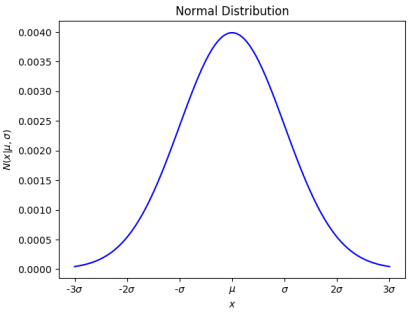

Figure 2: Shows a normal distribution around the mean $\mu = 0$ with standard deviation $\sigma = 10^3$

decision would be required. This shift in the number of opinions will be denoted as SHIFT(G) where G represents the `Graph` in which a random number of opinions is changed.
Moving on to the individual counter values, this is another case where using a probability distribution results in covering numerous configurations. Even though the exact values of l and k have not yet been determined, it can already be stated that these parameters are not going to be set to a constant value for any $n \in \mathbb{N}$. Boczkowski et al. [3] state that they probably scale around $\log(n)$. Since T_b and T_f can be set to any value α where $0 \leqslant \alpha < \lfloor \log(n) \rfloor \ \forall \alpha \in \mathbb{N}$, a distribution that is capable of dealing with different quantities of values is needed. This condition can easily be satisfied by a *binomial distribution* $\mathcal{B}(t = \lfloor \log(n) \rfloor - 1, p)$ where t represents the number of trials with probability of success equal to p. Figure 3 shows a typical binomial distribution. Obviously, the number of trials must not exceed l or k which is why t is either set to $l-1$ when randomizing T_b or $k-1$ in case of T_f. Note that setting t to l or k might cause problems. Doing so leads T_b to reach $l+1$ in some cases and T_f to reach $k+1$. Since we do not unnecessarily want these values to be increased to an invalid value, the maximum value generated by the distribution has to be $\lfloor \log(n) - 1 \rfloor$. The idea behind using this distribution is to randomize the counter values T_b and T_f of k `Nodes` according to random values generated by the distribution. Since $p \in [0, 1]$, these values can arbitrarily be scaled which results in a large configuration space to be covered. Randomly determining the number of counter values that are randomized is an option as well. Moreover, both counter values do not need to be randomized according to a binomial distribution with the same probability of success. Always choosing an identical p for randomizing both values results in numerous configurations to be neglected. This strategy will be denoted by RANDC(G,k) where G represents the `Graph` in which k `Node`'s counter values are changed.

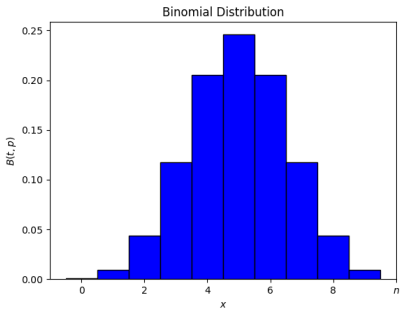

Figure 3: Shows a binomial distribution where $t = 10$ with probability of success $p = 0.5$

Finally, the states also require a randomization strategy so that the simulation has the opportunity to search its way through a completely randomized configuration space. In theory, the strategy presented for randomizing the number of opinions could be used in this case as well. However, using a normal distribution to determine the number of individual states changing is rather inappropriate. The main reason for that is that a `Node`'s state is a ternary instead of a binary value as $n = f('b') + f('s') + f('f')$. Assume that a normal distribution is used to determine a change in the number of state i where $i \in \{b, s, f\}$. While applying this strategy to adjust the number of opinions does not require an additional strategy, the fact that states are ternary values complicates this situation. Find-

ing an appropriate system for changing $f('b') + f('s') + f('f') - f('i')$ is obligatory. As a result, guaranteeing a properly randomized search space becomes rather difficult. Fortunately, we can simply utilize the fact that states are ternary values. In the strategy we will apply, we *flip* the state of k randomly chosen `Nodes`. Flipping a `Node`'s state means that its state changes corresponding to the BSF-ALGORITHM. Without checking any conditions, a boosting `Node` becomes susceptible, susceptible `Nodes` freeze and whenever a frozen `Node` is chosen, it restarts boosting. This way, there is no need to worry about the requirement of any additional randomization strategies. Even though probability distributions do not provide a good way for randomizing this parameter, a normal distribution may be used to determine the position of k `Nodes` whose states are flipped. FLIP(G,k) will serve as notation for denoting this strategy.
As the strategies for generating neighbor configurations have been clarified, it is necessary to combine the simulation's algorithm with these strategies. Moreover, the `GraphController` class can be completed now.

5.3.2 *Simulation Implementation*

Explanations on everything that is required to implement the simulation, excluding T_k, have been given. Still, the simulation's general algorithm presented at the beginning of this section, Algorithm 4, needs to be adapted to the self-stabilizing broadcast problem. Let us think about what kind of changes need to be made while using operations directly related to our problem. For this explanation, assume that temperature $T_k \in \mathbb{N}^+$. After setting T_k, obtaining a solution requires initializing and self-stabilizing a Graph G_0. Note that self-stabilizing a `Graph` is highly probabilistic which means that we cannot ignore the variance in this process. That is why a solution will be based on the mean that results from self-stabilizing the same `Graph` m-times where $m \in \mathbb{N}^+$. It is also important to notice that, once a `Graph` self-stabilized, that particular `Graph` can neither be reproduced nor self-stabilized again. Procedures for allowing these operations have not been implemented. `Graph` $G_{s'}$ is supposed to be based on a neighbor configuration of G_s though. Fortunately, this issue can simply be avoided by copying G_0 before its self-stabilization. That allows reusing G_0 as many times as required. Continuing with the implementation, we can start with the annealing. Before applying the strategies presented in Section 5.3.1, it is necessary to create another copy of G_s. This copy is required for resetting G_s after its self-stabilization in case neither $s' > s$ nor $e^{\frac{s'-s}{T_k}} > \text{rand}(0,1)$. After creating a copy, our strategies can be applied. At this point, a last copy of the new `Graph` $G_{s'}$ has to be created for resetting $G_{s'}$ to its initial state as well. Otherwise, there is no way for continuing with the new `Graph` in case we find a new solution. Determining s' by taking the mean over m self-stabilizations of $G_{s'}$, validating s' and reducing the temperature are the only operations left completing the

Algorithm 5 Adapted simulated annealing

```
1: procedure SIMULATE(n,k)
2:     Set T_k
3:     G_0 ← Graph(n)
4:     G_s ← G_0                                   ▷ Create the first backup
5:     m ← 10                                      ▷ Choose m relatively small
6:     s ← (1/m) Σ_{k=1}^{m} selfstabilize(G_0)
7:     while T_k > 0 + c do
8:         G_b ← G_s                               ▷ Copy old configuration
9:         SHIFT(G_s)
10:        FLIP(G_s,k)
11:        RANDC(G_s,k)
12:        G_s' ← G_s                              ▷ Copy new configuration
13:        s' ← (1/m) Σ_{k=1}^{m} selfstabilize(G_s)
14:        if (s' > s) or (e^{(s'-s)/T_k} > rand(0,1)) then
15:            s = s'
16:            G_s ← G_s'                          ▷ Override old Graph
17:        else                                    ▷ Otherwise
18:            G_s ← G_b                           ▷ Discard neighbor configuration
19:        end if
20:        Reduce T_k
21:    end while
22:    return s
23: end procedure
```

simulation's one round of the annealing. As soon as T_k reaches $0 + c$ for some problem-dependent constant c, the simulation is finished. Finally, s' can be returned and $G_{s'}$'s data optionally be printed. Algorithm 5 shows the adapted algorithm described above. This version also suggests to check s' and the acceptance probability in only one condition as they result in the same operations to be executed.

The algorithm's implementable adaptation also reveals that the `GraphController` class does not need to implement any properties. That is because, at any point of the algorithm's execution, all data can simply be extracted from an arbitrary `Graph` as long as the access is not restricted. An additional function that takes a `Graph` as its parameter for retrieving and printing the data should be implemented though.

There is one more aspect that needs to be analyzed. As this implementation is ideally capable of finding hard initial configurations for $n \geqslant 10^6$, runtime optimization has already been mentioned as an important requirement. Therefore, a mechanism directly related to that matter is going to be subject of this chapter's last section.

5.4 MULTI-THREADING

Most computers have to be able to handle the execution of multiple applications at the same time. Besides that, applications usually have

to perform several actions concurrently as well. While each running application handled by an operating system is called a *process*, an action performed by a process runs in a so-called *thread*. Moreover, the term *multi-threading* refers to the concurrent execution of multiple threads. In order to enable multi-threading, a multi-core *central processing unit* (CPU) is required as the number of threads that can concurrently be executed is equal to the number of cores provided by a processor. That arises the question of how single core CPUs manage multiple threads as they still seem to guarantee parallelism to some extent. The reason as to why one core is capable of creating the illusion of concurrent thread execution lies in how CPUs treat threads in general. A core does not keep on executing the exact same thread but switches between several other threads, called *context switches*. As a result, even a single core CPU seems to execute multiple threads at the same time. Moreover, context switching implies that a thread can be in different states as there are assigned as well as non-assigned threads. In fact, threads can switch between the three following states [39]:

- **Ready**: Whenever a thread is invoked and ready for its execution, it starts in this state. Threads in this state need to be assigned to a CPU's core. Note that, in case there are more threads ready for their execution than cores available, the operating system is in charge of choosing the next thread for its assignment. This kind of thread management is called *scheduling*. As soon as the operating system assigns a thread to a CPU's core, it switches into the **running** state.

- **Running**: A thread in the **running** state can progress in three different ways.

  1. The thread **terminates**.
  2. The thread **resigns** back into the **ready** state after a specific amount of time to allow the execution of other threads i. e. to simulate parallelism.
  3. The thread encounters a condition that needs to be satisfied for the thread to continue which leads the thread to transition into the **blocked** state.

- **Blocked**: As soon as a thread's blocking condition is satisfied, it returns to the **ready** state. The condition may also include in- and output operations.

That being said, multi-threading sounds like a great way to improve an application's runtime. Especially when the application heavily relies on calculations. However, there is a major issue in relation to multi-threaded applications that demands a correctly implemented threading library, namely *synchronization*. Imagine a scenario in which multiple threads concurrently change a certain value. Situations like that must not be allowed by an implementation as they lead to inconsistent data. In this case, a mechanism for making sure that only one

thread at a time may access shared memory, *mutual exclusion* (mutex), has to be implemented. Code that may only be run by exactly one thread is called *critical section*.
One more danger a programmer has to be careful about when implementing multi-threading are *deadlocks*. In relation to multi-threaded applications, a deadlock describes a state in which an application is unable to proceed as there are two threads that wait for each other to release its resources. The following four conditions [5] need to be satisfied for allowing deadlocks:

- **Mutual exclusion**: This condition's meaning has already been stated above.
- **Hold and Wait**: Describes a state in which a thread requests additional resources to the ones it already holds.
- **No preemption**: In case a thread acquires a resource, no preemption means that the resource must not be removed and can only be released by the thread itself.
- **Circular wait**: Means that each thread waits for another thread to release its resources.

Deadlocks can easily be avoided by simply making sure that at least one of these conditions does not apply. The only condition that can impossibly be ignored when threads need to access shared memory segments is mutual exclusion. Strategies for specifically neglecting any other condition have to be considered based on the respective problem.
This section will present and analyze four concrete multi-threading strategies. As this mechanism's implementation differs for every programming language, there will be listings showing actual C++ code instead of pseudocode. Please note that the strategies presented below are merely considerations that are going to be evaluated in Section 6.1.4.

5.4.1 *Multi-threaded* BSF-ALGORITHM

A naïve approach for reducing our problem's runtime could be concurrently adjusting each `Node`'s opinion during the BSF-ALGORITHM's execution. More precisely, threading the operations $B(n_i)$, $S(n_i)$ and $F(n_i)$ depending on n_i's opinion. A `Graph` with n `Nodes` implies that this implementation requires **invoking** and **joining** n threads in each round of the algorithm's execution. As mentioned above, invoking a thread leads the thread to start in the **ready** state. Joining the threads after their completion is obligatory because of two reasons. Firstly, the variable that keeps track of how many rounds the algorithm has already required to self-stabilize may only be increased once. Secondly, starting a new round requires the previous round's completion which means that each thread has to terminate before progressing to the next round. Listing 1 shows this strategy's implementation.

Listing 1: Shows a multi-threaded BSF-ALGORITHM implementation

```
std::vector<std::thread> threads;
int rounds = 0;

// Enter the self-stabilization loop
while(nodeOpinions[nodes.at(0).opinion] < nodes.size()){
   sample();

   // Invoke a thread for each node a put them into a vector
   for(int i=1; i<nodes.size();i++){
      // Parameters: function, instance of the corresponding
      // class and arguments passed to the function
      threads.push_back(std::thread(&Graph::executeOperation,
      this,std::ref(nodes.at(i))));
   }

   // Join the threads
   for(auto& thread : threads){
      thread.join();
   }

   // Remove all threads from the vector
   threads.clear();
   rounds++;
}
return rounds;
```

Please note that removing all threads from the vector is mandatory as the application crashes otherwise. The reason for that is the following: as soon as all threads complete their corresponding operations in the second round, the application tries to join the first thread in the vector. However, a thread that has already been joined must not be joined again which causes the application to crash.
As this strategy does not involve threads that read and write data which could potentially lead to inconsistency, mutual exclusion is not required. As a result, there is no need to worry about the application to run into a deadlock either. Every single aspect of this approach seemed favorable so far. Keep in mind that invoking and joining αn threads where α represents the number of rounds the algorithm needs to self-stabilize results in a massive overhead though. Moreover, the operations per thread i. e. the number of operations executed in $B(n_i)$, $S(n_i)$ and $F(n_i)$ is rather small. Consequently, this strategy is highly likely to not improve the algorithm's runtime. Section 6.1.4 will evaluate this conclusion.

5.4.2 *Concurrent Configuration Adjustments*

Moving to the adapted simulated annealing, operations that are perfectly suitable for applying a multi-threading strategy are SHIFT(G_s), FLIP(G_s,k) and RANDC(G_s,k). These operations do not compete for any

data as each of them randomizes different properties. This strategy's implementation is equally to the one previously presented as shown by Listing 2.

Listing 2: Shows a multi-threaded neighbor configuration generation

```
std::vector<std::thread> threads;

// Invoke a thread for each function where a threads
// requires the function, an instance of the corresponding
// class and the function's parameters
threads.push_back(std::thread(&Graph::shift,
std::ref(pullModel)));
threads.push_back(std::thread(&Graph::flip,
std::ref(pullModel),k));
threads.push_back(std::thread(&Graph::randC,
std::ref(pullModel),k));

// Join the threads
for(auto& thread : threads){
  thread.join();
}
```

Using a vector for thread management is obviously optional as joining them requires three lines of code even if each thread is manually invoked. Note that each function's randomization aspect plays a significant role in this implementation. When applying this strategy, it is crucial to make sure that `std::rand` is not used by any thread for generating random numbers. That is because of a problem mainly related to smaller `Graphs`. When running the application for smaller n, the simulation will need less than a second for executing one round where n depends on a system's respective hardware. This will lead to `std::rand` generating the exact same number over and over. Not to mention that `std::rand` is not thread-safe either as described in its documentation. That is why this implementation uses `std::default_random_engine` for generating time-independent and thread-safe random numbers.
This strategy's potential for improving the application's runtime is considerably higher compared to the approach presented in 5.4.1. The number of operations executed in each thread is not only noticeably higher but the overhead arising from thread creation and invocation is significantly lower as well. Evaluating this strategy will show whether there is an in- or decrease in the application's runtime.

5.4.3 *Achieving a Multi-threaded Mean Calculation*

Staying in the adapted simulated annealing, the next strategy refers to determining the mean that is taken over m `Graph` self-stabilizations. This turns out to be rather difficult as this is the first time we need to deal with a value returned by a function that we want to run in a thread. More precisely, calculating the mean requires summing

up m self-stabilization results where each self-stabilization runs in a seperate thread. In fact, `std::thread` is unable to provide this functionality by itself. Listing 3 shows an auxilliary function that uses `std::future` for guaranteing a multi-threaded mean calculation.

Listing 3: Shows a multi-threaded mean calculation

```
int m = 10;
double sum = 0;
std::vector<std::thread> threads;
std::vector<std::future<int>> futures;

// Invoke m threads and store their values in a promise
// Note that Graph g is passed to this function
for(int i = 0; i < m; i++){
  std::promise<int> p;
  futures.push_back(promise.get_future());
  threads.push_back(std::thread(&Graph::selfstabilize,
  std::move(g), std::move(p)));
}

// Join the threads
for(auto& thread : threads){
  thread.join();
}

for(int i = 0; i < m; i++){
  sum = sum + futures.at(i).get();
}

return (sum/m);
```

Let us take a look at the first loop as it represents the most complicated part of this implementation. First, an `std::promise` p is declared. This namespace allows storing values that are not yet available but will be determined in the future. However, a `promise` has to be connected to an `std::future` to aquire its value. Using a vector once again provides an elegant solution for managing m `future` objects. After inserting a `future` into the vector, the `future`'s corresponding thread can be invoked. Note that this invocation does not only require passing the function and a reference that is able to call the function but a `promise` for storing the function's result as well. This results in the following adjustments that have to be implemented in Algorithm 3:

- Algorithm 3 must take a `promise` as its parameter.
- Receiving legitimate results demands reseeding each `Graph`'s random number generator. Not doing so leads to producing the exact same result over and over as g's random number generator was initialzed in its constructor.
- Before leaving Algorithm 3, the `promise` that is passed to the function needs to be set to the function's result which would

normally be returned. In case computing single results is no longer required, returning a value is unnecessary as well.

Fortunately, these adjustments are easily implemented as shown by Listing 4. This strategy might fail at improving the application's runtime for smaller n. On the other hand, sequentially determining the mean is highly likely going to be too slow in case of large scale simulations. The next chapter will give further insights on this matter.

Listing 4: Shows the adjustments required in Algorithm 3

```
// Pass a promise to the function
void Graph::selfstabilize(std::promise<int> p){
// Reseed the random number generator
int seed = std::chrono::system_clock::now().
time_since_epoch().count();
generator.seed(seed);

/*
*  Insert the rest of the algorithm
*/

// Pass the result to the promise at the end of the function
p.set_value(result);
```

5.4.4 *Running each Simulation in a Thread*

Considering that the data generated in Chapter 6 is going to be based on numerous experiments, running multiple simulations at the same time would be desirable. Since each simulation instantiates its `Graphs` locally, i. e. it runs independently, there is no need to implement additional synchronization measures. Furthermore, this strategy's implementation does not differ compared to the others, excluding the mean calculation, as shown by Listing 5.

Listing 5: Shows how to run each simulation in a thread

```
std::vector<std::thread> threads;

// Invoke s threads where s is passed as a parameter
for(int i=0; i<s; ++i){
  threads.push_back(std::thread(&GraphController::simulate,
  this,n,k));
}

// Join the threads
for(auto& thread : threads){
  thread.join();
}
```

As a simulation captures the largest number of operations in the entire application, this strategy is not only simple but enormously powerful as well. Based on the fact that choosing $n \geqslant 10^6$ results in accessing tens of millions of `Nodes`, this implementation might even be required to allow large scale simulations. The absolute runtime difference will be presented in Section 6.1.4.

Part VI

EXPERIMENTS

6 RUNNING THE ALGORITHM

All experiments that are going to be executed in this chapter are either required for determining parameters, measuring runtimes or finding hard initial configurations. That is why this chapter is divided into three sections. The first section provides comprehensive explanations on how each parameter that explicitly requires experiments is determined. After choosing parameters that look optimal to the algorithm, we will continue by attempting to find correlations in hard configurations. Finally, Section 6.3 will take on checking the conjecture that applying the BSF-ALGORITHM leads to any initial configuration to self-stabilize in $\mathcal{O}(\log n)$.

6.1 FINDING APPROPRIATE PARAMETERS

Several parts of the implementation remained incomplete as they required one or multiple parameters. Since most of them have a considerable impact on the application's runtime and the convergence of the BSF-ALGORITHM, their choices have to be based on concrete data. This section will start off by presenting the measure used for evaluating the data retrieved in this chapter. After that, this section proceeds by determining the following parameters:

1. First, it is crucial to find optimal values concerning the BSF-ALGORITHM's parameters as they highly influence each round of the simulation. In this case, optimal means that the number of rounds required by the algorithm to self-stabilize should be as small as possible. The insights provided by Boczkowski et al. [3] will be used for choosing these parameters in Section 6.1.1.

2. The next step includes optimizing the simulation. Doing so requires choosing a starting temperature T_0. Section 5.3 has already mentioned that this parameter's value depends on the respective problem. That is why Section 6.1.2 will attempt to find an optimal value for T_0.

3. After finding an appropriate starting temperature, experimentally determining a well scaling schedule for reducing T_k is a crucial step for escaping local and discovering global minima. Therefore, Section 6.1.3 will test several cooling schedules [16, 25, 36] for optimizing the simulation.

4. Finally, Section 6.1.4 will evaluate each threading strategy presented in Section 5.4. This includes evaluating the usage of several strategies at the same time.

6.1.1 *Optimally Choosing l and k*

The most important parameters for showing that the BSF-ALGORITHM converges in $\mathcal{O}(\log n)$ for any arbitrary configuration are l and k. Unwisely setting their values might lead to conclusions that can be disproved. Fortunately, Boczkowski et al. [3] mention that both parameters are likely to scale around $\log n$. Because of this, the following experiments will be executed: in each experiment, the parameters l and k are set to a value $\alpha \log n$ where $\alpha \in \mathbb{Q}^+$. Furthermore, we calculate a value by summing up the results of 10^2 independent experiments on the same configuration. This allows to draw a conclusion based on similar configurations. Moreover, we also calculate the mean over all experiments to reduce the amount of variance. The last factor influencing the results of our experiments is the number of `Nodes` n that we initialize a `Graph` with. In these experiments, we will use decimal powers starting with 10^2 and ending with 10^6. Table 3 starts off by only considering parameter combinations in which $l = k$.

Table 3: Shows the mean values related to experiments with $l = k$ where each result is based on 10^2 experiments

	$n =$				
$l \wedge k$	10^2	10^3	10^4	10^5	10^6
$\lfloor \log_{10} n \rfloor$	17	23	28	36	42
$\lfloor \frac{1}{2} \log_2 n \rfloor$	8	23	31	35	52
$\lfloor \log_2 n \rfloor$	10	16	20	27	28
$\lfloor 2 \log_2 n \rfloor$	13	16	28	33	38

These results show that setting l and k to $\lfloor \log_2 n \rfloor$ is, on average, the best option in most cases. They also reveal that there is no need to test $l \wedge k > \lfloor 2 \log_2 n \rfloor$ as a considerable increase in the number of rounds becomes visible as soon as we move to larger `Graphs`. Note that an attempt to find appropriate parameters also has to consider combinations where $l \neq k$. Remember the beginning of Chapter 4 where we distinguished between two following two scenarios regarding the convergence of the algorithm:

- The configuration of the `Graph` directly leads to a converging behavior.
- First, the algorithm diverges and reaches a point where the number of disagreeing `Nodes` in the `Graph` comes close to n. After the `Nodes` restart boosting, the fact that most of them are going to be sensitive to the opposite opinion leads to a quickly converging behavior with high probability.

Considering these scenarios, neither in- nor decreasing the number of rounds in the boosting period should lead to an improvement in the number of rounds a `Graph` needs to converge. Shortening the period results in the `Node`'s opinions to fluctuate too much. On the

other hand, extending the period leads to the `Nodes` requiring too many rounds to agree on an opinion. However, rescaling the time frame for frozen `Nodes` might have a positive effect on the convergence of the algorithm. Table 4 shows whether these assumptions are true. Note that we only consider combinations that include $\lfloor \log_2 n \rfloor$ because of our previous results. Besides that, in case a combination starts producing results that are highly unlikely to lead to an improvement, the parameters will not be tested any further.

Table 4: Shows the mean values related to experiments with $l \neq k$ where each result is based on 10^2 experiments

		$n =$				
l	k	10^2	10^3	10^4	10^5	10^6
$\lfloor \log_2 n \rfloor$	$\lfloor \log_{10} n \rfloor$	7	12	<u>21</u>	<u>25</u>	<u>32</u>
$\lfloor \log_2 n \rfloor$	$\lfloor \frac{1}{2} \log_2 n \rfloor$	<u>5</u>	13	<u>21</u>	28	<u>32</u>
$\lfloor \log_2 n \rfloor$	$\lfloor 2 \log_2 n \rfloor$	7	<u>11</u>	22	28	<u>32</u>
$\lfloor \log_{10} n \rfloor$	$\lfloor \log_2 n \rfloor$	13	23	37	X	X
$\lfloor \frac{1}{2} \log_2 n \rfloor$	$\lfloor \log_2 n \rfloor$	6	26	36	X	X
$\lfloor 2 \log_2 n \rfloor$	$\lfloor \log_2 n \rfloor$	6	13	27	35	42

Even though rescaling k leads to results similar to $l \wedge k = \lfloor \log_2 n \rfloor$, parameter combinations with $l \neq k$ do not lead to a consistent improvement. While smaller `Graph`s tend to converge slightly faster, $n \geqslant 10^4$ neither appears to profit from shortening nor extending the period in which `Nodes` are frozen. This behavior can not be observed with respect to the boosting period. In fact, these results confirm our previous assumption. Rescaling l leads to a tendency towards worse results when setting $l < k$ as well. As these experiments do not achieve an improvement, any further experiment that is executed subsequently will have the algorithm's parameters l and k set to $\lfloor \log_2 n \rfloor$.

6.1.2 *Temperature Determination*

Carefully determining the inital energy state T_0 represents an essential step towards optimizing the simulation. The main reason for that lies in the acceptance probability $e^{\frac{s'-s}{T_k}}$ where we recall s' to be the number of rounds configuration c' needed to self-stabilize while c' represents a neighbor configuration based on configuration c in round k (see Chapter 5 Section 5.3). To show that T_0 has a considerable impact, let us analyze the acceptance probability's behavior in relation to s, s' and T_k. Therefore, consider the following two scenarios where Δs corresponds to $s' - s$:

- $|\Delta s| > T_0$: Choosing a starting temperature that is highly likely to be lower than most $|\Delta s|$ that are expected to be generated at the beginning of the simulation is an unreasonable decision. That is because this condition implies that T_0 is lower than al-

most any $|\Delta s|$ during the entire simulation. Therefore, the acceptance probability $e^{\frac{-\Delta s}{T_0}} \xrightarrow[T_0 \to |\Delta s|]{} 0.3675$. Consequently, the probability for accepting a solution that does not improve the previous result is $\leqslant 36.75\%$ in most cases. Since T_0 progresses towards $0 + c$ for some problem-dependent constant c, the percentage will constantly decrease. This leads to a low probability for escaping local minima which does not fullfil the purpose of the acceptance probability.

- $|\Delta s| < T_0$: In contrast to the previous scenario, choosing an initial energy state T_0 that is greater than $|\Delta s|$ for a specific amount of time during the simulation leads $e^{\frac{-\Delta s}{T_0}} \xrightarrow[T_0 \to \infty]{} 1$. Because of T_0's decreasing behavior, this leads to a well scaling acceptance probability. Note that choosing a starting temperature that is too large leads to nearly any result being accepted for a long time during the simulation.

The previous explanations show that the effectiveness of the simulation heavily relies on determining an appropriate starting temperature. That is why we are especially interested in the behavior of $|\Delta s|$. In order to gain knowledge about $|\Delta s|$, we will run multiple simulations where each simulation runs for 100 rounds. Since we are looking for the mean over all values of $|\Delta s|$, we will set T_0 to a large value which leads to each round's result to be accepted as the new result. Doing so allows gaining an extensive insight in how much $|\Delta s|$ fluctuates. Table 5 shows the results produced in several simulations. Note that these results use $k = \frac{n}{10}$ for generating neighbor configurations.

Table 5: Shows the mean values of $|\Delta s|$ where each result is based on one simulation running for 10^2 rounds

	$n =$				
Simulation	10^2	10^3	10^4	10^5	10^6
1	24	24	23	25	29
2	18	22	17	27	35
3	16	20	23	28	32
4	17	25	25	30	33
5	19	23	23	34	30

First, we notice that $|\Delta s|$ does not fluctuate within a given quantity of `Nodes`. Even more interesting is the fact that increasing n does not lead to a considerable increase in $|\Delta s|$ either. Combining the knowledge we gained from Section 6.1.1 and the results provided in Table 5, we can roughly determine the smallest value required for achieving a decent acceptance probability when starting a simulation. Nevertheless, we will still set T_0 to multiple values when executing experiments in the following sections. Since the process we investigate to

have a high variance, we can neither guarantee nor even proof that setting T_0 to a specific value leads to an optimal result.

6.1.3 *Cooling Schedules*

After determining parameters that look optimal to the algorithm and getting decent insights regarding T_0, we need to find a way for progressively reducing the temperature during the simulation. Therefore, we will test three popular cooling schedules commonly used in relation to simulated annealing in this section. The linear cooling schedule

$$T_k = T_0 - c \tag{6}$$

for some constant $c > 0$ as well as the exponential schedule

$$T_k = T_0 \alpha^k \tag{7}$$

with $0 < \alpha < 1$ represent very simple schedules for leading T_0 to the state of equilibrium after a certain number of rounds k [25]. Another schedule that we will test in this section is the logarithmic cooling schedule

$$T_k = \frac{d}{\log(1+k)} \tag{8}$$

introduced by Geman and Geman [16] where d is independent of k and optimally set to the largest energy state that has to be overcome by the simulation. In contrast to (6) and (7), the logarithmic schedule has been proven to lead a system to the global minimum state in the limit of infinite time [19]. A way for comparing these schedules that has been used on a simple system as well as an NP-hard problem by Nourani et al. [36] is to capture the best-so-far-energy (BSFE) reached in round k [17]. Recall that we are still interested in finding configurations that require the maximum number of rounds for the BSF-ALGORITHM to converge to a valid state. It is important to notice that, in contrast to the NP-hard problem, we do not define the BSFE to be a negative value. This is why, in our case, the BSFE corresponds to the largest instead of the smallest value seen in round k. Using this method will not only educate us about the cooling schedule that we should use, but we will learn about the number of rounds a simulation needs until there is no more improvement in the result determined by the simulation. Consequently, we will execute the following independent experiments for getting the information we are looking for: we subsequently test each cooling schedule's performance on several `Node` populations ranging from 10^3 to 10^5 by observing the BSFE for a large number of iterations N. We choose $c = \frac{k \cdot T_0}{N}$, $\alpha = 0.95$ and $d = T_0$ for allowing a slow cooling process in case of each schedule. Moreover, we once again use $k = \frac{n}{10}$ for generating neighbor configurations.

The first thing that we learn from Figure 4 is that the number of iterations required for reaching the BSFE is not large. This is why we

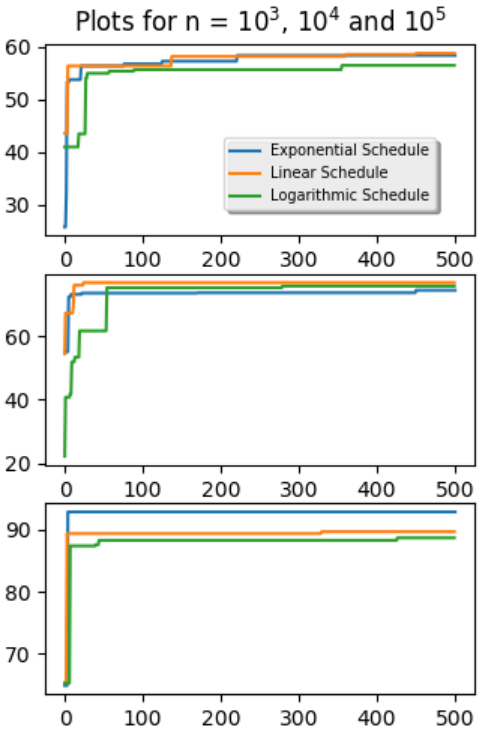

Figure 4: Shows the best-so-far-energies determined for each cooling schedule. The y-axis shows the BSFE while the x-axis shows the number of iterations.

will, similarly to simulations in earlier sections, keep on running any future simulations for 10^2 iterations. Unfortunately, this comparison does not deliver a result that allows to sensibly choose an appropriate cooling schedule. The main reason for this is likely to be the fact that the BSF-ALGORITHM is influenced by multiple factors including our strategies for generating neighbor configurations. As a result, observing the BSFE resulting from only one simulation might cover a configuration space that is insufficient for choosing an appropriate schedule. Therefore, we will base our decision on experiments that take a larger variety of factors influencing the BSF-ALGORITHM into account. We subsequently test each cooling schedule in ten independent simulations where we run each simulation with different parameters, i. e. we change T_0, k, the mean σ that we use for determining the number of `Nodes` changing in each round and the probability of success p for randomizing the number of individual counter values. After running these simulations, we take the mean over all best-so-far-energies for determining the cooling schedule that was able to produce the best results on average. The data points can be seen in Figure 5 as well as the resulting mean values in Table 6.

Without looking at Table 6, we discover that the logarithmic cooling schedule seems to perform slightly better than the others with respect to lower quantities. Moreover, we can observe a considerable difference in case of the largest population measured. The mean values that we computed for each quantity confirm these observations. This also matches the results from Nourani et al. [36]. Note that Fig-

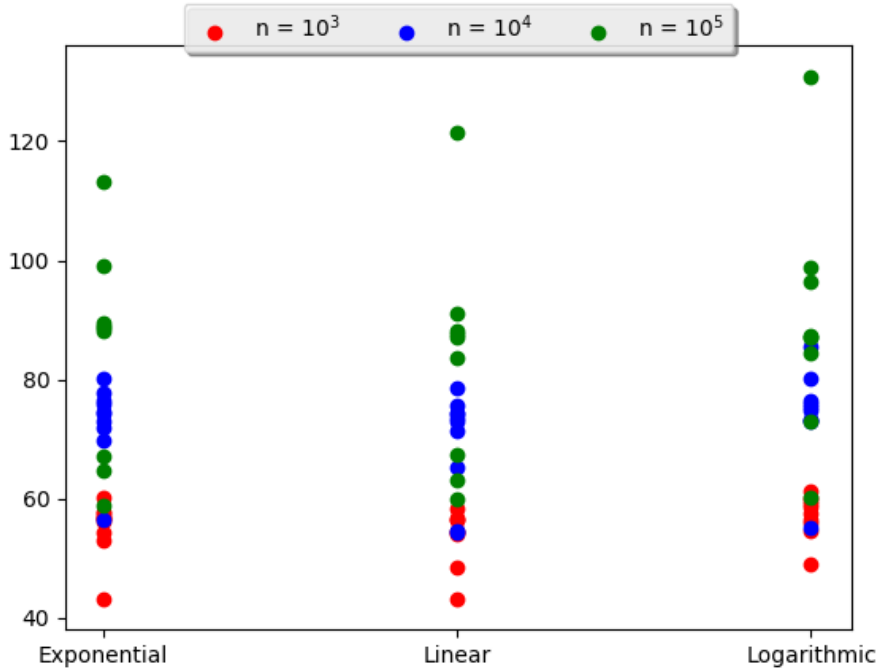

Figure 5: Shows the best-so-far-energies measured when considering a large variety of parameter combinations. The y-axis shows the BSFE while the x-axis shows the corresponding cooling schedule.

ure 5 also points to a problem that we will dedicate our attention to in Section 6.2: the data points still exhibit a substantial spreading behavior within a given quantity of `Nodes`. Preferably, we would like to be able to find hard initial configurations more consistently. Therefore, eliminating convergence favoring configurations is going to be the next task after evaluating our multi-threading strategies. Since the logarithmic cooling schedule outperforms the others, future simulations will be based only on this schedule.

Table 6: Shows the mean values and standard deviations in relation to the data points from Figure 5

	n =		
Schedule	10^3	10^4	10^5
Exponential	55.2	73.04	84.88
Linear	53.96	69.51	83.79
Logarithmic	57.25	74.3	92.57
Standard Deviation	4.37	7.68	18.22

6.1.4 *Multi-Threading Strategies Evaluation*

Before we move to optimizing the search space, we still need to evaluate the multi-threading strategies that were considered in Section 5.4. As all parameters have been determined, we are now capable of test-

ing each strategy with respect to the same conditions that are going to apply when computing the final results. The following abbreviations will be used for referring to each strategy.

- We use **multi-threaded** BSF for referring to a multi-threaded BSF-ALGORITHM.
- **multi-threaded neighbor** will be used when applying a multi-threaded neighbor configuration generation.
- When using `std::future` as well as `std::promise` for calculating the mean during a simulation, we use **multi-threaded mean**.

Before running several simulations concurrently, we will test the strategies mentioned above. This will allow us to implement and use favorable strategies when running multiple large scale simulations. Therefore, we will start by executing the following independent experiments: we test each strategy on a given quantity of `Nodes` ranging from 10^2 to 10^6 separately in exactly one simulation where we do not change the parameters within a given quantity. Also, we purposely choose $k = \frac{n}{2}$ for straining the neighbor configuration generation.

Table 7: Shows the runtimes measured for each multi-threading strategy in seconds using a 4-core 3.5 GHz CPU

	$n =$				
Strategy	10^2	10^3	10^4	10^5	10^6
None	0.07	1.33	18.15	269	> 6600
Multi-threaded BSF	68.47	X	X	X	X
Multi-threaded neighbor	0.11	1.31	17.38	206	> 6000
Multi-threaded mean	0.08	0.61	8.14	111	> 2400

Table 7 shows the performance of each individual strategy. We notice that executing large scale simulations for $n \geqslant 10^6$ indeed requires a considerable amount of time, especially when passing on multi-threading strategies. Fortunately, multi-threaded neighbor as well as multi-threaded mean allow accelerating the simulation in each relevant case compared to the single-threaded variation. Also, the anticipated overhead when multi-threading the BSF-ALGORITHM can be confirmed. There is no point in testing multi-threaded BSF even further as only the overhead increases which would lead to even more meaningless runtimes. As a result of these experiments, we will use both runtime favoring strategies in any future simulations.
Completing our evaluation, we are still interested in the improvement that we can achieve by running multiple simulations concurrently. Note that our strategies already exhaust a noticeable number of cores, especially with respect to the mean calculation. Therefore, achieving a runtime optimization requires a substantial number of cores at this

point. There is no doubt that this strategy leads to a runtime improvement when running numerous large scale simulations with more than four cores. However, this requires tremendous calculation power. In fact, running ten simulations that underlie the same conditions as our previous experiments concurrently does not achieve a runtime optimization when using the same hardware. This can be concluded from Table 8.

Table 8: Shows the runtimes measured when self-stabilizing ten configurations sequentially as well as concurrently in seconds using a 4-core 3.5 GHz CPU

	$n =$				
Execution	10^2	10^3	10^4	10^5	10^6
Sequentially	0.58	4.88	54	955	X
Concurrently	0.58	4.5	54	945	X

Since these results include both strategies that were previously confirmed to improve the runtime of the algorithm, we still get the information that implementing them at the same time improves the performance even further.

6.2 SEARCH SPACE OPTIMIZATIONS

Currently running a simulation leads to randomly moving in the configuration space. Doing so results in a search where we need to hope that the simulation finds hard initial configurations. Because of this, our results still exhibit a considerable spreading behavior as we were able to discover in Section 6.1.3 while evaluating several cooling schedules. Therefore, the last step before presenting the final results requires minimizing the search space by eliminating configurations that favor the convergence of the BSF-ALGORITHM. Optimally, we want to minimize the number of possible configurations to the point where we can consistently find hard initial configurations. For discovering parameter correlations in these configurations, we subsequently dedicate a Section to each parameter. Section 6.2.1 analyzes how the most influential parameter, the number of informed `Nodes`, affects the convergence. After gaining those insights, we move to the number of individual states in Section 6.2.2. Finally, based on previous results, we also attempt to find correlations regarding the individual counter values T_b and T_f in Section 6.2.3.

6.2.1 *Informed vs. Uninformed Populations*

The number of informed `Nodes` in a population represents a crucial factor regarding the convergence of the BSF-ALGORITHM. When starting in an initial configuration where a few `Nodes` are informed, we might experience the scenario where the population fails to agree on the opinion propagated by the source leading to all agents to dis-

agree at some point. We also investigated this case in Chapter 4. On the other hand, informed populations where only a small fraction of Nodes propagates wrong information are more likely to converge directly. However, we are unaware as to how informed a population may be when searching for hard initial configurations. This is why we execute the following independent experiments for gaining more precise insights regarding this problem: before running a simulation, we initialize the number of informed Nodes with progressively increasing numbers $i \in I = \{1^1, \frac{n}{10}, \frac{2n}{10}, ..., \frac{9n}{10}\}$. Furthermore, we take the mean over ten simulations for each quantity $i \in I$. Similarly to Section 6.1.3, we consider Node populations ranging from 10^3 to 10^5 and do not change parameters within a given quantity of Nodes. The most important step towards useful insights lies in how we generate neighbor configurations as we completely pass on changing the number of informed Nodes. This way, we focus on exploring a search space that is directly related to the respective fraction of informed Nodes.

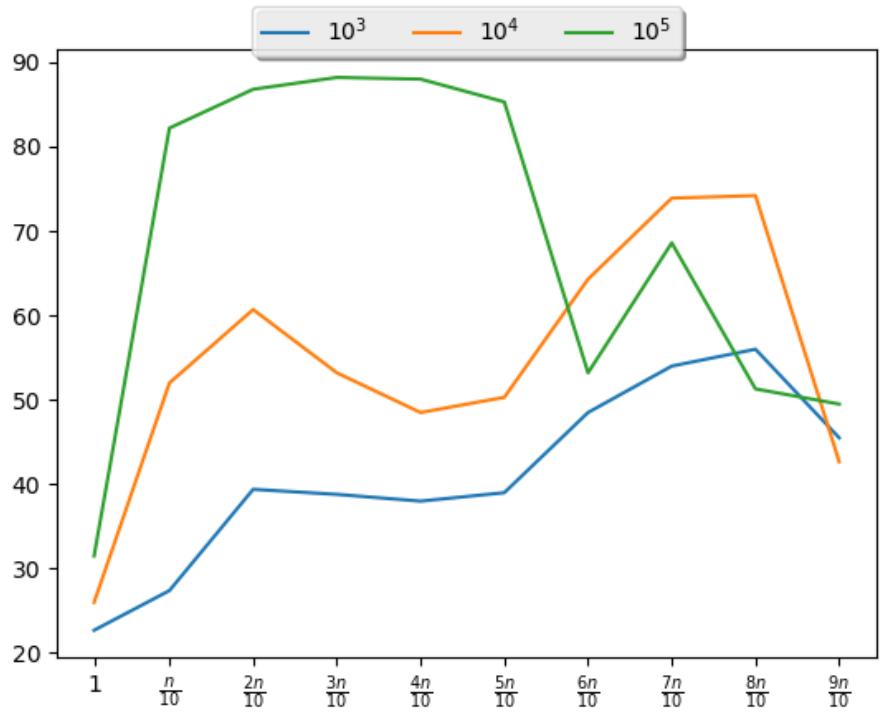

Figure 6: Shows the results determined by the simulation on the y-axis while the x-axis shows the quantity of informed Nodes a population was initialized with.

There is indeed a crucial difference regarding the convergence when comparing uninformed to informed populations as we can see in Figure 6. Configurations where less than half of the population is informed appear to favor the convergence in case of smaller populations. Furthermore, as soon as the number of informed Nodes exceeds that fraction, we notice a steady increase in the number of rounds our algorithm requires to converge. The increase stops after informing too many Nodes as configurations tend to converge di-

1 This Node represents the Zealot

rectly with higher probability. This situation changes when observing larger populations. As long as the informed fraction remains below 50%, the simulated-annealing search discovers the hardest initial configurations. Based on these experiments, there is no doubt that we may limit the search space of our simulation to the interval that we discovered to lead to the hardest initial configurations. Therefore, we will not allow the fraction of informed `Nodes` to drop below $\frac{6n}{10}$ or exceed $\frac{9n}{10}$ in case of $n = 10^3$ and $n = 10^4$ as this range led to the hardest initial configurations to be found. Since we need to keep that fraction within a different range with respect to the largest population, we will not allow the fraction to be lower than $\frac{n}{10}$ or higher than $\frac{5n}{10}$. By restricting the search space for a certain quantity, we will attempt to find more precise correlations regarding the individual number of states and counter values T_b and T_f in the following two sections.

6.2.2 *The Impact of State Shifting*

We just showed that the informed fraction of `Nodes` plays a significant role regarding the convergence of the BSF-ALGORITHM. To find correlations with respect to the number of individual states among a population, we will proceed similarly. Since our strategy for generating a neighbor configuration does not provide a good way for initializing a population with a certain state combination as we do not use a probability distribution, we will use a binomial distribution instead. We execute the following independent experiments for discovering correlations between the fraction of informed `Nodes` and the number of individual states: before running a simulation, we initialize the number of individual states corresponding to a binomial distribution with probability of success $p \in P = \{0.1, 0.2, ..., 0.9\}$. We once again compute the mean over ten simulations for each probability of success $p \in P$. Furthermore, we restrict a given quantity to the fraction of informed `Nodes` that turned out to result in the hardest configurations. Similar to the previous section, we completely pass on flipping states when generating a neighbor configuration. Figure 7 shows which probabilities lead to the hardest initial configurations based on all insights that we gained from Section 6.2.1.

A logical consequence from restricting the fraction of informed agents to what we determined to be the worst range is that we are able to find harder configurations more consistently. Even though our results still fluctuate, their range is considerably smaller than in Figure 6. Furthermore, the fact that the convergence time becomes even worse in some cases represents another achievement. Knowing the probability that leads to the worst configurations on average, we will initialize the individual number of states corresponding to the respective probability in any future simulations. By doing so, our simulation is more likely to find and self-stabilize hard initial configuration when generating a neighbor configuration. Based on these insights, we will

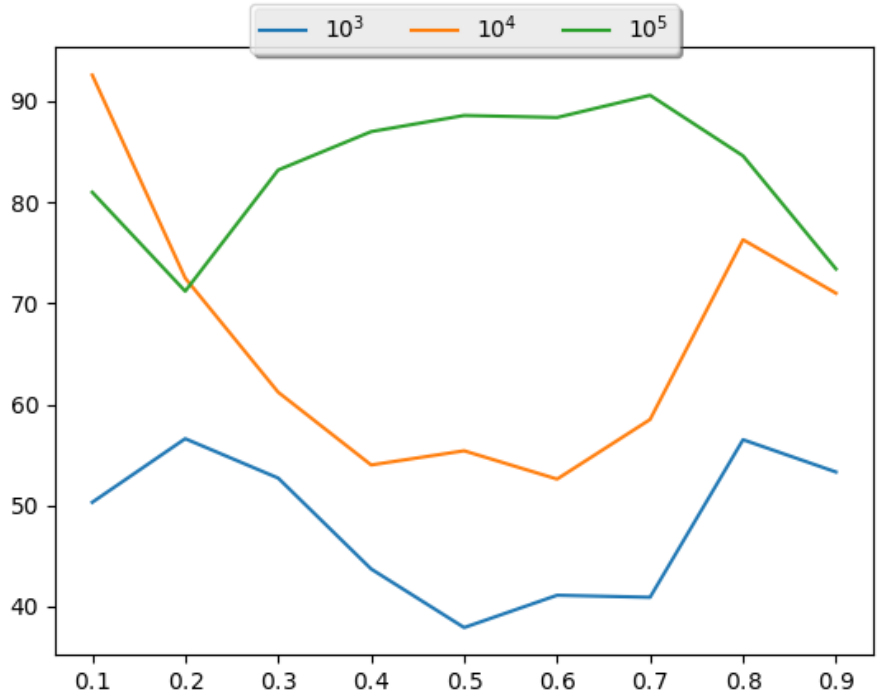

Figure 7: Shows the results determined by the simulation on the y-axis while the x-axis shows the probability of success p that was used for initializing a state combination.

attempt to find correlations regarding a `Node`'s counter values T_b and T_f in the next section.

6.2.3 *Boost and Freezing Duration Based Correlations*

Moving on, we focus our attention to finding correlations regarding a `Node`'s boost and freezing duration. In contrast to our strategy for shifting the number of individual states, we already use a binomial distribution for randomizing these values in each round of a simulation. However, when generating a neighbor configuration, the distribution requires a probability of success p. Finding correlations regarding the quantity of p in relation to previous insights is our last task before moving to the last section. The independent experiments that we are going to execute for discovering these correlations only differ in one aspect compared to the previous section. This time, we do not only adjust the number of informed `Nodes` corresponding to the range that we determined in Section 6.2.1 but we initialize a population with a state combination that we learned to lead to the hardest initial configurations on average. Furthermore, we once again rely on a binomial distribution with probability of success $p \in P = \{0.1, 0.2, ..., 0.9\}$ for initializing each `Node`'s boost duration T_b and freezing duration T_f. The mean over ten simulations for each quantity $p \in P$ will serve as a measure regarding the convergence in the respective duration combination. Note that adjusting these values is not allowed during any simulation as we are interested in knowing how the algorithm performs with respect to given boost and freezing

durations based on the binomial distribution. This will educate us about the probability of success p that leads to the hardest initial configurations on average. Figure 8 shows the impact of each probability based on previous insights.

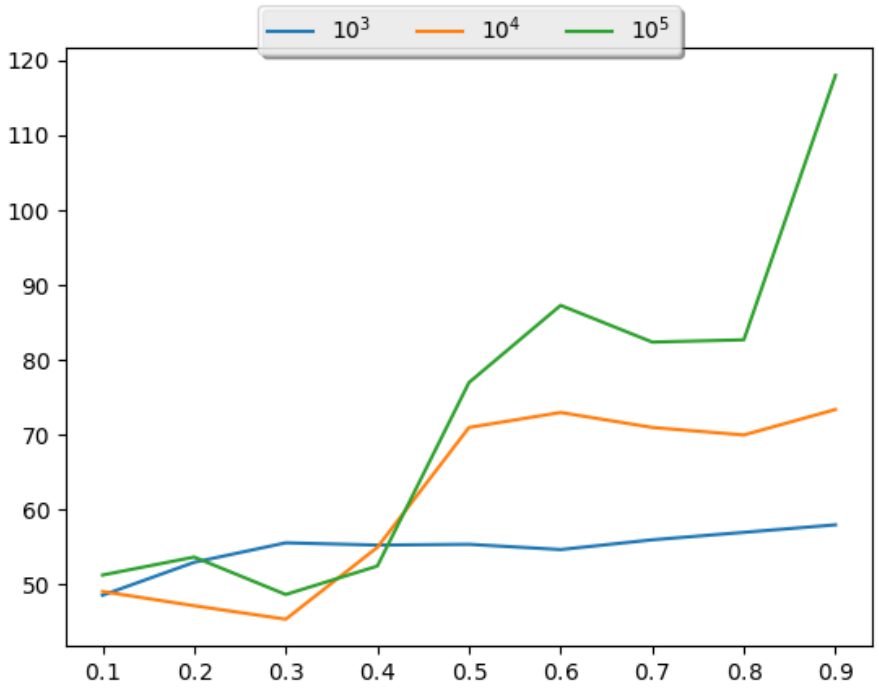

Figure 8: Shows the results determined by the simulation on the y-axis while the x-axis shows the probability of success p that was used for initializing a duration combination.

Unlike Figure 6 and Figure 7 where we were partially able to observe different behaviors in relation to different quantities, these results show noticeable similarities. Each population requires the highest number of rounds to converge when initialized with $p \geqslant 0.5$ while $p = 0.9$ appears to lead to the hardest initial configurations on average. We will consider these insights in the last section and randomize the boost as well as the freezing duration with the probability that favors the convergence the least.

6.3 RESULTS

After putting effort into the determination of parameters that look optimal to the algorithm in Section 6.1 and searching for correlations in hard initial configurations in the previous section, we finally move to the results. Therefore, we will proceed by executing the following independent experiments for the last time: we restrict the fraction of informed `Nodes`, initialize the state combination and use probability p for randomizing each `Node`'s boost and freezing duration corresponding to values that were previously determined to lead the simulated-annealing search into finding the hardest initial configurations on average. This condition applies in every single simulation that we execute. Since our strategies are still sensitive to the number

of `Nodes` k that change in each round of the simulation, we will test different quantities. The data points in Figure 9 as well as the mean values and standard deviations presented in Table 9 refer to the quantity that resulted in the simulation that determined the hardest initial configurations on average for the respective quantity of `Nodes`.

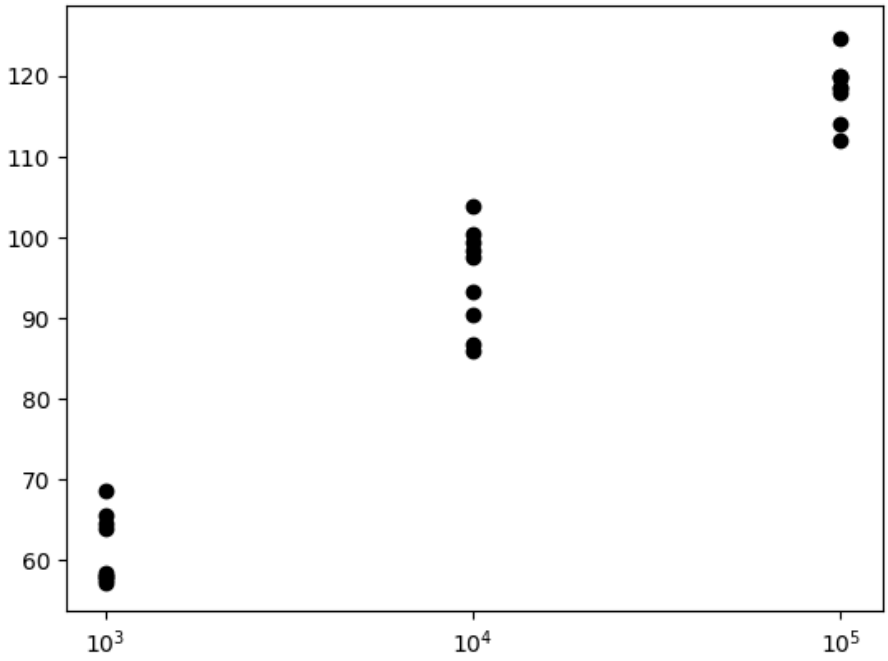

Figure 9: Shows the final results determined by the simulation on the y-axis while the x-axis shows the quantity of the population.

These results show that the effort that we put into detecting correlations considerably contributes to optimizing the search. We have not only managed to reduce the standard deviation but to push the maximum number of rounds to even higher values in case of 10^3 and 10^4. Also, all data points that we determined for the largest quantity now reach values similar to the hardest configuration visualized in Figure 5. The most important information that we can conclude based on these results is that any arbitrary configuration self-stabilizes in poly-logarithmic runtime. Finally, it remains to conclude the results of this thesis in the last chapter.

Table 9: Shows the mean values and standard deviations in relation to the data point from 9

Population	*Mean*	*Standard Deviation*
10^3	61	4.26
10^4	95.7	6.23
10^5	118.4	3.31

Part VII

CONCLUSION

7 CONCLUSION

The intention of this thesis was to provide an extensive analysis of the BSF-protocol for showing that the algorithm is capable of self-stabilizing from any arbitrary configuration in $\mathcal{O}(\log n)$ rounds. A theoretical investigation of the broadcast process, which is closely related to the algorithm, has shown that we can bound the minimum number of rounds by $\Omega(\log n)$. Furthermore, an experimental analysis via the randomized search heuristic simulated annealing was supposed to provide more precise insights on the behavior and the convergence of the algorithm. Determining parameters that look optimal to the protocol has shown that the algorithm exhibits high sensitivity regarding the boosting state. In case the boosting period is too small, agents may adjust their opinion too frequently. On the other hand, longer periods have shown to be problematic as agents take too long to agree on an information. As a result of these observations, setting the algorithms' parameters l and k that are responsible for fixing the maximum number of rounds an agent may boost or be frozen to $\lfloor \log_2 n \rfloor$ turned out to be the best choice in terms of the convergence of the algorithm. Running simulations for determining the starting temperature T_0 revealed that the number of rounds a neighbor configuration requires to self-stabilize neither fluctuates within a given quantity of agents nor shows noteworthy increases when self-stabilizing larger populations. For optimizing the simulation even further, we compared the performance of a linear, exponential and logarithmic cooling schedule. Our results show that the logarithmic schedule achieves superior results compared to the others as the schedule finds hard initial configurations, on average, more consistently. Furthermore, we were able to achieve substantial runtime improvements by exploiting multithreading strategies. After putting considerable effort in optimizing the parameters of the simulated annealing, we showed existing correlations in hard initial configurations that vary based on the size of a population. When we considered the correlations we discovered in our final results, the simulation was able to find hard initial configurations a lot more consistently. Finally, our ultimate results confirmed a convergence time of $\mathcal{O}(\log n)$.

BIBLIOGRAPHY

[1] Dan Alistarh, Rati Gelashvili, and Milan Vojnović. "Fast and Exact Majority in Population Protocols." In: *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*. PODC '15. Donostia-San Sebastián, Spain: ACM, 2015, pp. 47–56. ISBN: 978-1-4503-3617-8. DOI: 10.1145/2767386.2767429. URL: http://doi.acm.org/10.1145/2767386.2767429.

[2] Dana Angluin, James Aspnes, and David Eisenstat. "A simple population protocol for fast robust approximate majority." In: *Distributed Computing* 21.2 (2008), pp. 87–102. ISSN: 1432-0452. DOI: 10.1007/s00446-008-0059-z. URL: https://doi.org/10.1007/s00446-008-0059-z.

[3] Lucas Boczkowski, Amos Korman, and Emanuele Natale. "Brief Announcement: Self-stabilizing Clock Synchronization with 3-bit Messages." In: *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*. PODC '16. Chicago, Illinois, USA: ACM, 2016, pp. 207–209. ISBN: 978-1-4503-3964-3. DOI: 10.1145/2933057.2933075.

[4] Keren Censor-Hillel, Bernhard Haeupler, Jonathan Kelner, and Petar Maymounkov. "Global Computation in a Poorly Connected World: Fast Rumor Spreading with No Dependence on Conductance." In: *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing*. STOC '12. New York, New York, USA: ACM, 2012, pp. 961–970. ISBN: 978-1-4503-1245-5. DOI: 10.1145/2213977.2214064. URL: http://doi.acm.org/10.1145/2213977.2214064.

[5] E. G. Coffman, M. Elphick, and A. Shoshani. "System Deadlocks." In: *ACM Comput. Surv.* 3.2 (June 1971), pp. 67–78. ISSN: 0360-0300. DOI: 10.1145/356586.356588.

[6] National Research Council et al. *Catalyzing inquiry at the interface of computing and biology*. National Academies Press, 2006.

[7] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. "Epidemic Algorithms for Replicated Database Maintenance." In: *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*. PODC '87. Vancouver, British Columbia, Canada: ACM, 1987, pp. 1–12. ISBN: 0-89791-239-X. DOI: 10.1145/41840.41841. URL: http://doi.acm.org/10.1145/41840.41841.

[8] Edsger W. Dijkstra. "A Note on Two Problems in Connexion with Graphs." In: *Numer. Math.* 1.1 (Dec. 1959), pp. 269–271. ISSN: 0029-599X. DOI: 10.1007/BF01386390. URL: http://dx.doi.org/10.1007/BF01386390.

[9] Edsger W. Dijkstra. "Self-stabilizing Systems in Spite of Distributed Control." In: *Commun. ACM* 17.11 (Nov. 1974), pp. 643–644. ISSN: 0001-0782. DOI: 10.1145/361179.361202.

[10] Benjamin Doerr and Mahmoud Fouz. "Asymptotically Optimal Randomized Rumor Spreading." In: *Automata, Languages and Programming: 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part II*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 502–513. ISBN: 978-3-642-22012-8. DOI: 10.1007/978-3-642-22012-8_40.

[11] Benjamin Doerr, Leslie Ann Goldberg, Lorenz Minder, Thomas Sauerwald, and Christian Scheideler. "Stabilizing Consensus with the Power of Two Choices." In: *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '11. San Jose, California, USA: ACM, 2011, pp. 149–158. ISBN: 978-1-4503-0743-7. DOI: 10.1145/1989493.1989516. URL: http://doi.acm.org/10.1145/1989493.1989516.

[12] Shlomi Dolev and Ted Herman. *Superstabilizing Protocols for Dynamic Distributed Systems*. Tech. rep. 1997.

[13] Jennifer H Fewell. "Social insect networks." In: *Science* 301.5641 (2003), pp. 1867–1870.

[14] Alan M. Frieze and Geoffrey R. Grimmett. "The shortest-path problem for graphs with random arc-lengths." In: *Discrete Applied Mathematics* 10 (1985), pp. 57–77.

[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[16] Stuart Geman and Donald Geman. "Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images." In: *IEEE Trans. Pattern Anal. Mach. Intell.* 6.6 (Nov. 1984), pp. 721–741. ISSN: 0162-8828. DOI: 10.1109/TPAMI.1984.4767596. URL: http://dx.doi.org/10.1109/TPAMI.1984.4767596.

[17] George Ruppeiner, Jacob Mørch Pedersen, and Peter Salamon. "Ensemble approach to simulated annealing." In: *J. Phys. I France* 1.4 (1991), pp. 455–470. DOI: 10.1051/jp1:1991146.

[18] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989. ISBN: 0201157675.

[19] Bruce Hajek. "Cooling Schedules for Optimal Annealing." In: *Math. Oper. Res.* 13.2 (May 1988), pp. 311–329. ISSN: 0364-765X. DOI: 10.1287/moor.13.2.311. URL: http://dx.doi.org/10.1287/moor.13.2.311.

[20] Donald O. Hebb. *The organization of behavior: A neuropsychological theory*. New York: Wiley, 1949. ISBN: 0-8058-4300-0.

[21] J. J. Hopfield. "Neurocomputing: Foundations of Research." In: ed. by James A. Anderson and Edward Rosenfeld. Cambridge, MA, USA: MIT Press, 1988. Chap. Neural Networks and Physical Systems with Emergent Collective Computational Abilities, pp. 457–464. ISBN: 0-262-01097-6.

[22] Mads Kærn, Timothy C Elston, William J Blake, and James J Collins. "Stochasticity in gene expression: from theories to phenotypes." In: *Nature Reviews Genetics* 6.6 (2005), pp. 451–464.

[23] Richard Karp, Christian Schindelhauer, Scott Shenker, and Berthold Vöcking. "Randomized Rumor Spreading." In: *41th Annual Symposium on Foundations of Computer Science (FOCS-00)*. IEEE Computer Society. Redondo Beach, USA: IEEE, 2000, pp. 565–574. ISBN: 0-7695-0852-9.

[24] Nadav Kashtan and Uri Alon. "Spontaneous evolution of modularity and network motifs." In: *Proceedings of the National Academy of Sciences* 102.39 (2005), pp. 13773–13778. DOI: doi:10.1073/pnas.0503610102.

[25] Scott Kirkpatrick, C Daniel Gelatt, Mario P Vecchi, et al. "Optimization by simulated annealing." In: *science* 220.4598 (1983), pp. 671–680.

[26] Hiroaki Kitano. "Biological robustness." In: *Nature Reviews Genetics* 5.11 (2004), pp. 826–837.

[27] Donald E. Knuth. "Computer Programming as an Art." In: *Communications of the ACM* 17.12 (1974), pp. 667–673.

[28] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998. ISBN: 0-201-89685-0.

[29] Joseph B Kruskal. "On the shortest spanning subtree of a graph and the traveling salesman problem." In: *Proceedings of the American Mathematical society* 7.1 (1956), pp. 48–50.

[30] Shay Kutten and Boaz Patt-Shamir. "Time-adaptive Self Stabilization." In: *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '97. Santa Barbara, California, USA: ACM, 1997, pp. 149–158. ISBN: 0-89791-952-1. DOI: 10.1145/259380.259435. URL: http://doi.acm.org/10.1145/259380.259435.

[31] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System." In: *Commun. ACM* 21.7 (July 1978), pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563.

[32] Leslie Lamport. "Solved Problems, Unsolved Problems and Nonproblems in Concurrency." In: *SIGOPS Oper. Syst. Rev.* 19.4 (Oct. 1985), pp. 34–44. ISSN: 0163-5980. DOI: 10.1145/858336.858339. URL: http://doi.acm.org/10.1145/858336.858339.

[33] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine Generals Problem." In: *ACM Trans. Program. Lang. Syst.* 4.3 (July 1982), pp. 382–401. ISSN: 0164-0925. DOI: 10.1145/357172.357176. URL: http://doi.acm.org/10.1145/357172.357176.

[34] Sharon Bertsch McGrayne. *The theory that would not die: how Bayes' rule cracked the enigma code, hunted down Russian submarines, & emerged triumphant from two centuries of controversy*. Yale University Press, 2011.

[35] Saket Navlakha and Ziv Bar-Joseph. "Algorithms in nature: the convergence of systems biology and computational thinking." In: *Molecular Systems Biology* 7.1 (2011). ISSN: 1744-4292. DOI: 10.1038/msb.2011.78.

[36] Yaghout Nourani and Bjarne Andresen. "A comparison of simulated annealing cooling strategies." In: *Journal of Physics A: Mathematical and General* 31.41 (1998), p. 8373.

[37] David Peleg. *Distributed Computing: A Locality-sensitive Approach*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2000. ISBN: 0-89871-464-8.

[38] Nitzan Razin, Jean-Pierre Eckmann, and Ofer Feinerman. "Desert ants achieve reliable recruitment across noisy interactions." In: *Journal of the Royal Society Interface* 10.82 (2013), p. 20130079.

[39] Jerome Howard Saltzer. "Traffic control in a multiplexed computer system." PhD thesis. Massachusetts Institute of Technology, 1966.

[40] David JT Sumpter, Jens Krause, Richard James, Iain D Couzin, and Ashley JW Ward. "Consensus decision making by fish." In: *Current Biology* 18.22 (2008), pp. 1773–1777.

[41] G. Varghese. *SELF-STABILIZATION BY LOCAL CHECKING AND CORRECTION*. Tech. rep. Cambridge, MA, USA, 1992.

LIST OF FIGURES

LIST OF TABLES

LISTINGS

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/