DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Analysis of Population Protocols with Specific Communication Structures

Radu Vintan

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Analysis of Population Protocols with Specific Communication Structures

Analyse von Population Protocols mit Spezifischen Kommunikationsstrukturen

Author:	Radu Vintan
Supervisor:	Prof. Dr. Dr. h.c. Javier Esparza
Advisor:	Dr. Stefan Jaax
Submission Date:	15.07.2020

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.07.2020 Radu Vintan

Acknowledgments

I am grateful to my advisers, Prof. Javier Esparza and Dr. Stefan Jaax, for giving me the opportunity to work on an interesting research problem, and for guiding me along this journey. Their feedback was always constructive, and our weekly meetings were always pleasant. Thanks to them, the process of creating the thesis was a valuable learning experience.

Abstract

Population Protocols are a model used in distributed computing. A population is a collection of mobile agents, which randomly meet each other, while collectively trying to execute a computation, which leads to a yes/no answer. Every agent has a state and behaves like an automaton. Whenever two agents meet, they update their state based on their current state and on the state of their partner. A protocol specifies the possible states and the rules which apply when agents meet. Each state is associated with one of two possible outputs: yes/no. If, after some time, all agents have the same output, the computation is said to have converged and the common output is considered the answer of the computation. Whether two agents can meet or not is specified by an interaction graph. In this thesis, we analyze how fast protocols converge on specific interaction graphs, depending on the population size and on the initial states. We have also implemented a tool which can help us simulate the protocols on any given graph.

Kurzfassung

Population Protocols sind ein Modell, welches für verteilte Systeme verwendet wird. Eine Population ist eine Multimenge von mobilen Agenten, welche sich zufällig zuzweit treffen, und die zusammen versuchen, eine Berechnung durchzuführen. Diese Berechnung endet mit einer Ja/Nein antwort. Jeder Agent hat einen Zustand und verhält sich ähnlich zu einem Automaten. Wenn sich zwei Agenten treffen, werden ihre Zustände aktualisiert, abhängig von dem eigenen Zustand und von dem Zustand ihres Partners. Ein Protokoll beschreibt die möglichen Zustände und die Regeln, die angewendet werden, wenn sich Agenten treffen. Jeder Zustand ist zu einem Ausgang assoziiert. Es gibt zwei mögliche Ausgänge: Ja/Nein. Wenn, nach einiger Zeit, alle Agenten denselben Ausgang besitzen, dann sagen wir, dass die Berechnung konvergiert hat, und der gemeinsame Ausgang wird als Antwort der Berechnung interpretiert. Ob sich zwei Agenten treffen können wird von einem Interaktionsgraphen bestimmt. Wir analysieren in dieser Arbeit wie schnell Protokolle auf spezifische Interaktionsgraphen konvergieren, abhängig von der Größe der Population und der initialen Zustände. Wir haben auch ein Tool implementiert, welches Protokolle auf beliebige Graphen simulieren kann.

Contents

1 Introduction

A population protocol is a model used in distributed computing. The concept has been introduced by Angluin et al. in [1]. A population is a collection of agents, which all behave identically, and which have no identities. The agents are assumed to have very limited computation power and memory resources. They can use their memory to remember their state, i.e. information about themselves and about the population. Agents can meet each other and, when this happens, they change their states depending on their own state and on the state of their partner. They can be regarded as (non-deterministic) automata, which fire transitions whenever they interact with another agent. The way state transitions work is decided *a priori* by the protocol. The protocol also determines the possible states. There is no central component governing this process, and therefore this can be considered a decentralized algorithm.

Every state has an associated output, which is usually either 0 or 1. If, at some point in time, all agents have states that are associated to the same output, then there is a consensus in the population, i.e. all agents have the same "opinion" on some matter. This is an important concept, because it allows us to define a termination condition for the algorithm. It can, however, be the case that a consensus is later broken by some transition that makes an agent "change its mind". We are therefore mostly interested in consensuses which are lasting, i.e. which cannot be reverted. We call the mapping of agents to states a *configuration*. Each computation begins with the agents set to some initial states, and so there is an initial configuration, which will then be altered during the computation.

The objective of the agents is to determine whether their initial configuration has a certain property, which is specified by a given predicate. The purpose of a protocol is to guide the agents in such a way, that their meetings lead to a configuration in which all of them have the same "opinion". Furthermore, this final opinion should be correct: it should tell us whether the initial configuration satisfies the given predicate or not. Designing protocols that "solve" a certain problem is clearly challenging, and much literature has been devoted to this subject.

Different types of network topologies add to the difficulty of designing a protocol. Very often, it is implicitly assumed that any pair of agents can meet, but this assumption can be relaxed. We can generally model this using an undirected graph, which we will call the interaction graph. Its nodes are the agents, and an edge signifies that this pair of agents can meet. It makes sense to assume that these graphs are connected, because a population should not have sub-populations that can never meet each other.

Population protocols have been proven capable of solving some classical problems in distributed computing, most importantly leader election [1] and majority voting [1, 2]. Angluin et al. proved [1, 3] that, surprisingly, the predicates that can be computed by population protocols are exactly the semilinear predicates, or equivalently, the Presburger definable predicates. These are predicates that can be formulated in first-order logic using the universe of natural numbers and a signature containing only addition and natural ordering. The proof uses results from convex geometry [4]. Another independent proof was given by Esparza et al. in [5]. It is based on the theory of vector addition systems [4]. Specific protocols have also been investigated in regard to their convergence speed, i.e. the expected number of interactions that is needed to reach a (correct) consensus. For example, in [6], the authors give an upper bound for the convergence speed of a protocol that solves the majority problem. Automatically verifying the correctness of protocols is also possible [7, 8].

For this thesis, we will investigate what happens if we assume a general (connected) interaction graph. There exists some literature on this topic. In [1], it is proven that some protocols can work correctly on arbitrary (directed) interaction graphs. In [9], special protocols are analyzed, which can deduce information about the topology they run on. For some particular protocols, like the majority protocol, there exist upper bounds for the convergence speed when run on arbitrary connected topologies (including random graphs) [6]. In general, there exist two main concerns. The first one is correctness: given a protocol that works on complete graphs, will it also work on any connected graph? If not, what should we change to make it work? The second one is convergence speed: how fast do protocols converge on arbitrary interaction graphs? How much slower can it get in comparison to using complete graphs? We will try to answer these questions. Furthermore, we have developed a tool that can help us simulate executions of specific protocols on arbitrary interaction graphs. For this, we have used Peregrine, a tool under development at the Chair for Foundations of Software Reliability of the Technical University of Munich [10, 4]. Peregrine already supported simulations on

complete graphs. (Another more simple example of such a tool can be found in [11].) We have extended Peregrine to work on general interaction graphs. As far as we know, it is the first (open-source) tool that allows this.

Outline. The rest of the thesis is structured as follows. We give some preliminary definitions in Chapter 2, then formally define and discuss Population Protocols in Chapter 3. We also explain how protocols can be made to work on arbitrary interaction graphs. Chapter 4 contains our main contribution: it partially answers questions regarding performance, and compares different methods of running protocols on graphs. Chapter 5 explains our implementation of the new features of Peregrine, and gives a tour of this useful tool. It includes an analysis of the speed of convergence for the AVC protocol when run on specific graphs. In Chapter 6, we conclude the work and discuss possibilities of further research.

2 Preliminaries

2.1 Elementary Concepts

Notations:

- $\mathbb{N} :=$ set of natural numbers (including 0)
- $2^A :=$ set of subsets of a set A
- $\binom{A}{k} := \{M \in 2^A : |M| = k\}$ for a set A and a natural number k
- $[i] := \{1, \ldots, i\}$ and $[i]_0 := \{0, \ldots, i\}$ for a natural number i

Definition. A *directed graph* is a tuple $G := (V, E)$, where V is a non-empty, finite set, and $E \subseteq V^2$ is an irreflexive relation on V. An *undirected* graph is a tuple $G := (V, E)$, where V is a non-empty, finite set, and $E \subseteq \binom{V}{2}$. Or, equivalently, $E \subseteq V^2$ is an irreflexive and symmetric relation on V. In all cases, V is called the set of nodes, and E is called the set of edges. We will more often use undirected graphs.

A *path* between two nodes $u, v \in V$ of a (directed or undirected) graph exists iff there exist $n \in \mathbb{N}$, $w_1, \ldots, w_n \in V$, such that $w_1 = u$, $w_n = v$ and $(w_i, w_{i+1}) \in E$ for any $i \in [n-1]$. For directed graphs, we introduce the equivalence relation $\equiv$ on V, where $u \equiv v$ iff there exist paths from u to v and from v to u. An equivalence class of $\equiv$ is called a *strongly connected component (SCC)* of G. If the nodes of a SCC contain no edges to nodes that are outside the SCC, we call it a *bottom SCC*. An undirected graph is called *connected* iff there exists a path for any two nodes in V.

Definition. A *polynomial function* $f : A \to \mathbb{R}$, defined on some non-empty subset A of the real numbers, is a function for which there exist $n \in \mathbb{N}$, $a_n, \cdots, a_0 \in V$, such that $f(x) = \sum_{i=0}^{n} a_i x^i$ for all $x \in A$. A *rational function* $f : A \to \mathbb{R}$, defined on some non-empty subset A of the real numbers, is a function for which there exist polynomial functions g and h (also defined on A), such that:

- $h(x) \neq 0$, for all $x \in A$
- $f(x) = \frac{g(x)}{h(x)}$ for all $x \in A$

In section 4.3, we will use following known property of analysis: if f is a rational function defined on some non-degenerate interval A (i.e. $|A| > 1$), and f is bounded from below (formally, $M := \inf_{x \in A} |f(x)| < \infty$), then there exists some $x_0 \in \overline{A}$, such that $f(x_0) = M$, where $\overline{A}$ is the smallest closed interval that contains A.

2.2 Probability Theory Concepts

We assume that the reader is familiar with basic concepts of probability theory (like random variables, expected value) and knows about classical distributions (exponential, Poisson). We start by introducing the (discrete) *Markov chains*, which we use extensively in this thesis. For more details, we point out to [12].

Definition. A (discrete) *Markov chain* is a tuple $M := (S, P)$, where:

- S is a non-empty, finite set of *states*
- $P : S \times S \to [0, 1]$ is the *transition probability distribution*, and $P(s_1, s_2)$ is the probability of moving to state s_2, assuming the current state is s_1
- $P(s, \cdot)$ is a probability distribution for every $s \in S$, meaning that $\sum_{s' \in S} P(s, s') = 1$

A *run* in the Markov chain is an infinite sequence of states $s_0, s_1, \ldots$, where $P(s_i, s_{i+1}) > 0$ for any $i \geq 0$, and s_0 is called the *initial* state.

Let $M = (S, P)$ be a Markov chain and $s_0 \in S$ be some arbitrary state. We define the random variables $X_0, \ldots, X_i, \ldots$ where $P(X_0 = s_0) = 1$ and, for all $i \geq 0$, $s_1, \ldots, s_{i+1} \in S$, we have:

$$P(X_{i+1} = s_{i+1} | X_i = s_i, \ldots, X_0 = s_0) := P(s_i, s_{i+1})$$

Because $P(s_i, s_{i+1}) = P(X_{i+1} = s_{i+1} | X_i = s_i)$, we get:

$$P(X_{i+1} = s_{i+1} | X_i = s_i, \ldots, X_0 = s_0) = P(X_{i+1} = s_{i+1} | X_i = s_i)$$

This is called the *Markov property*. It is also easy to see that, for any $i_1 < j_1$, $i_2 < j_2$, such that $j_1 - i_1 = j_2 - i_2$, and $s', s \in S$, we have:

$$P(X_{j_1} = s' | X_{i_1} = s) = P(X_{j_2} = s' | X_{i_2} = s)$$

This is called the *stationary property*.

Definition. Let $M = (S, P)$ be a Markov chain. We use the same notations X_i as before. We denote $z_{ij} := E[\min\{n \geq 1 : X_{n+k} = j \text{ if } X_k = i\}$, where $i, j \in S$. Because of the stationary property, this is well defined (does not depend on k). z_{ij} is called the *expected hitting time from i to j*.
We will very often use the **hitting time equations**:

- $z_{ij} = 1 + \sum_{k \neq j} P(i, k) z_{kj}$ for all $i, j \in S$, $i \neq j$
- $z_{jj} = 1 + \sum_{k \neq j} P(j, k) z_{kj}$ for all $j \in S$

The expected hitting time z_{ij} is the number of steps we expect until the state j is reached in the Markov chain, assuming that we start in the state i and do at least one step. The hitting time equations are intuitive: to reach the state j in at least one step, we use the first step to reach a neighbor k of our initial state i, with probability/weight $P(i, k)$, and then recursively require an expected z_{kj} steps to reach j from k, if $k \neq j$. If $k = j$, then no more steps are required, so we only compute the weighted sum over the values $k \neq j$.

We continue with a chain of definitions leading to the *Poisson process*. There are many equivalent characterizations and subtleties associated to this concept, but we will only present what is required to understand the thesis, in particular Section 4.2. For more details, we point to [13] (which is also the source for the following definitions).

Definition. An *arrival process* is a sequence of random variables $S_i : \Omega \to [0, \infty)$, where $i > 0$ and $X_i := S_{i+1} - S_i$ are positive variables, i.e. $F_{X_i}(0) = 0$ for any $i > 0$. We write $0 < S_1 < S_2 < \dots$. The sequence of variables X_i is called the sequence of *interarrival times*. A *renewal process* is an arrival process, such that the sequence of interarrival times X_i consists of random variables which are independent and identically distributed.

Definition. A *Poisson process of rate* λ is a renewal process in which the interarrival intervals have an exponential distribution function, i.e. each X_i has density $f_{X_i}(x) = \lambda \exp(-\lambda x)$ for $x \geq 0$.

3 Population Protocols

3.1 Definitions and Properties

Introducing Protocols

A *Population Protocol* is a tuple $\mathcal{P} = (Q, T, I, O)$, where:

- Q is a non-empty finite set of *states*;
- $T \subseteq Q^4$ is the set of *transitions*;
- $I \subseteq Q$ is the *set of initial states*;
- $O : Q \to \{0, 1\}$ is the *output function*.

If $(A, B, C, D) \in T$, we write $AB \to CD$ and also assume that $(B, A, D, C) \in T$ holds, i.e. $BA \to DC$. Moreover, we assume $AB \to AB$ for all $(A, B) \in Q^2$ (*trivial/silent* transitions).
Note For the next part, $\mathcal{P}$ is a fixed, arbitrary population protocol.

Interaction Graphs and Configurations

An *Interaction Graph* of $\mathcal{P}$ is an undirected, connected graph $G := (V, E)$ where V is the non-empty finite *set of nodes*, and $E \subseteq \binom{V}{2}$ denotes which pairs of nodes are capable of *interacting*. We say that $\mathcal{P}$ *runs* on G.
A *Configuration* is a function $C : V \to Q$ specifying the state of each node.

Schedulers and Interactions

A *scheduler* $M(\mathcal{P}, G)$ is a (potentially probabilistic) algorithm which, given $\mathcal{P}$, an interaction graph $G = (V, E)$, and a configuration C, outputs a configuration C', that is equal to C everywhere except possibly at u and v, where $\{u, v\} \in E$. To suggest that there is a non-zero

probability that the algorithm M outputs C', we write $C \xrightarrow{M} C'$ or, if M is clear from the context, $C \to C'$. We call this a *step* (not to be confused with the transitions of the protocol). If C' is reachable from C in multiple (including 0) steps, we write $C \xrightarrow{*} C'$. We describe some schedulers:
1. M_1: If the interaction graph is complete, i.e. $E = \binom{V}{2}$, then we uniformly choose an edge $\{u, v\}$ of the complete graph. Let $q_1 = C(u), q_2 = C(v)$. We select a transition $q_1q_2 \to p_1p_2$ randomly from T, and apply it. This means setting $C'(u) = p_1, C'(v) = p_2$ and $C'(w) = C(w)$ for all $w \in V \setminus \{u, v\}$. Note that the chosen transition could be the trivial transition (which will then have no effect).
2. $M_2(p)$, where $0 < p < 1$: uniformly choose an edge $\{u, v\}$. With probability p, swap the states q_1, q_2 of nodes u and v. Else, randomly choose a transition $q_1q_2 \to p_1p_2$ from T and apply it.
3. $M_3(p)$, where $0 < p < 1$: uniformly choose an edge $\{u, v\}$ and randomly choose a transition $q_1q_2 \to p_1p_2$ from T, where q_1, q_2 are the states of u, v. Apply the transition on u, v. Afterwards, with probability p, swap the states of nodes u and v.

We say that u, v *interact* whenever a transition of the protocol is applied on their states. If this is not the trivial transition, we say that u, v *interact non-trivially*. For M_1, M_3, the chosen nodes u, v always interact, whereas for M_2 they interact with probability $1 - p$. Note that the presented schedulers M_i are somewhat predictable: For all configurations C', such that $C \xrightarrow{M_i} C'$, i.e. C' is obtained from C by applying some M_i-step, there is a well-defined probability $P(C \to C')$ of executing a M_i-step in C that produces C'. From now on, we will assume that all schedulers we discuss have this property.

Transition Graphs and Final Configurations

The *Transition Graph* $\mathcal{G}(\mathcal{P}, G, M)$, where $\mathcal{P}$ is running on the interaction graph G under M, is a directed graph whose nodes are the possible configurations and whose edges are the possible transitions on these nodes. A configuration is *final* iff it belongs to a bottom SCC (see Chapter 2) of the transition graph.

Properties of Configurations

A configuration C is *initial* iff $C(V) \subseteq I$, i.e. it sets all nodes of the graph to initial states. A configuration C is *stable* iff, for every C' such that $C \xrightarrow{*} C'$, we have $O(C'(V)) = O(C(V)) = \{0\}$ or $O(C'(V)) = O(C(V)) = \{1\}$, i.e. all nodes are either true or false nodes, and this property is permanent.
Two configurations C and C' are *equivalent*, written $C \equiv C'$, iff they are the same up to a permutation of the set of nodes, i.e. there exists a permutation π of V, such that $C(w) = C'(\pi(w))$, for all $w \in V$.

Executions and Runs

An *execution under a scheduler* M is a finite sequence $C_0, \ldots, C_l$ of configurations, such that $C_i \xrightarrow{M} C_{i+1}$ for all i. A *run under* M is an infinite sequence $C_0, \ldots, C_l, \ldots$ such that every finite prefix is an execution under M. A run w of M is called *fair* iff, for any step $C \to C'$, if C occurs infinitely often in w, C' also occurs infinitely often in w.

Probability Spaces for Runs

For a given execution $C_0, \ldots, C_l$ under a scheduler M, we use $Run(C_0, \ldots, C_l)$ to denote the set of all runs starting with $C_0, \ldots, C_l$. For every configuration C, we define the probability space $(Run(C), \mathcal{F}, P_C)$, where $\mathcal{F}$ is the σ-algebra generated by all $Run(C_0, \ldots, C_l)$ such that $C_0, \ldots, C_l$ is an execution initiated in C, and P_C is the unique probability measure satisfying $P_C(Run(C_0, \ldots, C_l)) = \prod_{i=0}^{l-1} P(C_i \to C_{i+1})$.
The intuition behind this definition is following: we have a natural way of assigning a measure to executions ("finite runs"), by setting $P(C_0, \ldots, C_l) = \prod_{i=0}^{l-1} P(C_i \to C_{i+1})$ for any execution $C_0, \ldots, C_l$. We then extend this definition to runs ("infinite executions"), by defining the canonical σ-algebra $\mathcal{F}$ on some subset of cylinder sets $Run(C)$, and then using the classical result of probability theory that $\mathcal{F}$ can be equipped with a canonical measure P_C, which "extends" the measures we defined for finite runs (executions).

Agents

Assume $\mathcal{P}$ runs on the interaction graph $G := (V, E)$ under $M \in \{M_2, M_3\}$. Let $A(G)$ be a fixed set, such that $|A(G)| = |V|$ and let $map_G : A(G) \to V$ be a bijection. Let

$w := C_0, \ldots, C_l, \ldots$ be a run. We define $pos_{C_0} : A(G) \to V$, $pos_{C_0} := map_G$. Then, inductively, if $C_i \overset{M}{\to} C_{i+1}$, and a swap of states of some nodes $u, v \in V$ was applied (see definition of schedulers), then define $pos_{C_{i+1}}(u) := pos_{C_i}(v)$, $pos_{C_{i+1}}(v) := pos_{C_i}(u)$, with $pos_{C_{i+1}}$ otherwise equal to pos_{C_i}. Else (if no swap was used), then $pos_{C_{i+1}} := pos_{C_i}$. The set $A(G)$ is called the *Population* or *set of agents*. In this model, the agents "wander around" the interaction graph. The functions pos_{C_i} induced by a run w give us the *positions of the agents*. $C_i(pos_{C_i}(u))$ gives us the *state of the agent* $u \in A(G)$. We will assume that the set $A(G)$ and the function map_G are predefined for any interaction graph G.
We will also trivially use the term *agents* when running under the scheduler M_1 (on complete graphs), as an alias for nodes (forcing a distinction makes no sense, because all nodes of the interaction graph are connected).

Transition Markov Chains

The *Transition Markov Chain* $\Gamma(\mathcal{P}, G, M)$, where $\mathcal{P}$ is running on the interaction graph G under M, "imitates" the Transition Graph: the set of states is the set of possible configurations, a directed edge between configurations C_1 and C_2 exists iff $C_1 \overset{M}{\to} C_2$, and the probability associated to this edge is equal to $P(C_1 \to C_2)$.

Computing Predicates

This definition formalizes the idea that Population Protocols are designed to check whether the initial configuration has a certain property. Given a scheduler M, a unary predicate ψ on initial configurations, and an interaction graph G, we say that $\mathcal{P}$ *computes* ψ iff for any initial configuration C, the probability of reaching a stable configuration, in the probability space $(Run(C), \mathcal{F}, P_C)$, is equal to 1, and for any reachable stable configuration D we have $O(D(V)) = \{1\}$ iff $C \in \psi$.

The next lemmas show that almost all runs are fair. Therefore, proving that a property holds for fair runs is enough to conclude that the property holds for almost all runs. The proof that almost all runs are fair is done by first giving an alternative characterization of fair runs.

Lemma 1. *Given a scheduler M, an interaction graph G, and an initial configuration C_0, let $F \subseteq Run(C_0)$ be the subset of fair runs, and Γ be the associated Transition Markov Chain. Consider*

$R \subseteq Run(C_0)$ the subset of runs $w := C_0, \dots C_l, \dots$ starting in C_0, such that:

- *w eventually reaches a bottom SCC B of Γ*
- *for any state $C \in B$, there are infinitely many $i \geq 0$, such that $C_i = C$, i.e. w visits all the states/configurations contained in B infinitely often.*

Then $F = R$.

Proof. We show that $R \subseteq F$: let w be a run in R, and C be a configuration appearing infinitely often in w. To prove fairness, we assume that C' is a configuration, such that $C \xrightarrow{M} C'$. Because $w \in R$ and C is visited infinitely often, we get $C \in B$, for some bottom SCC B. Because $C \xrightarrow{M} C'$, it must be the case that $C' \in B$. This implies that C' also occurs infinitely often in w, showing that $w \in F$ and $R \subseteq F$.
We show that $F \subseteq R$: let w be a fair run. Let J be the set of configurations occurring infinitely often in w, and G_J be the subgraph of Γ induced by J. Let $C, C' \in J$. Then C' is reachable from C in G_J, else C' would not have appeared infinitely often in w. So G_J is strongly connected. Now assume $C \in J$ and $C \to C'$. Because w is fair, C' appears infinitely often in w, so $C' \in J$. It follows that G_J is a bottom SCC, so every element of J is final. This shows that $w \in R$ and $F \subseteq R$. □

Lemma 2. *Given a scheduler M, an interaction graph G, and an initial configuration C_0, it follows that: the probability of a run π starting in C_0 being fair is equal to 1 in the probability space $(Run(C_0), \mathcal{F}, P_{C_0})$.*

Proof. Let Γ be the associated Transition Markov Chain, and R be defined as in Lemma 1. We apply Theorem 10.27 from [12] on Γ to obtain that R has measure 1. Because $R = F$ (Lemma 1), it follows that the set of fair runs also has measure 1.

□

3.2 Running Protocols on Graphs

Example. Consider the protocol $\mathcal{P}$ with states $\{B, R\}$, non-trivial transitions $\{(B, B, R, R)\}$, initial states $\{B, R\}$, and output function given by $O(B) = 0$, $O(R) = 1$, running on the interaction graph $G := (\{1, 2, 3\}, \{\{1, 2\}, \{2, 3\}\})$. We denote a configuration C by the tuple

$(C(1), C(2), C(3))$. Figure 3.1 shows the initial configuration $C_0 := (B, R, B)$. (B stands for blue, R stands for red.)

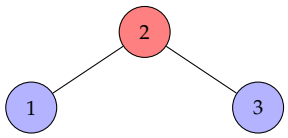


Figure 3.1: Initial Configuration Example

If we would run $\mathcal{P}$ under M_1, we would never reach stable consensus, because the nodes 1, 3 are not connected, so they cannot interact. However, with $M_2(p)$, where $0 < p < 1$, stable consensus will be reached: Eventually, we will swap along one of the two edges. Assume w.l.o.g. that we swap along $\{1, 2\}$. The new configuration is $C_1 := (R, B, B)$. A new swap along $\{1, 2\}$ would take us back to the initial configuration (so eventually back to C_1), and a swap along $\{2, 3\}$ would not change anything. When the edge $\{2, 3\}$ is eventually selected for an interaction, we reach the stable consensus $C_2 := (R, R, R)$, with output 1. Running under M_3 would lead to the same result. This example shows why swapping helps protocols run correctly on arbitrary graphs.

We now prove that protocols that correctly calculate predicates on complete graphs (under M_1) also compute the same predicates on arbitrary interaction graphs (under M_2 or M_3). This is not a fully original result: lemmas similar to the ones from this section appear, for example, in [1]. However, the authors use directed graphs for their model, which leads to more complicated proofs, and do not consider different schedulers as we do.

Fix a protocol $\mathcal{P}$, a set of nodes V and a number $0 < p < 1$. On V, we consider the complete interaction graph G_1, and an arbitrary interaction graph G_2. We use M_1 to run $\mathcal{P}$ on G_1 and $M_i(p)$ ($i \in \{2, 3\}$) to run $\mathcal{P}$ on G_2. Note that, when using M_i on a given configuration, it is always possible (i.e. with probability > 0) to swap the states of any two nodes, without altering their states, and also possible to apply a (usable) transition from T on any two connected nodes, without swapping their states. When using M_1, it is possible to apply a (usable) transition from T on any two nodes; however, swapping the states of two arbitrary nodes is, in general, impossible.

We show:

Lemma 3. *Let $i \in \{2,3\}$. Let $C \equiv C'$, where C is reachable in G_1 (under M_1) and C' is reachable in G_2 (under M_i). Suppose $C' \xrightarrow{M_i*} D'$ in G_2. Then $C \xrightarrow{M_1} D$ and $D \equiv D'$ for some G_1-configuration D.*

Proof sketch. Per induction on the length k of the execution $C' \xrightarrow{*} D'$. Idea: if the step is a swap of two states, then $D' \equiv C' \equiv C$ and we are done, because we can take $D := C$. Else, the step consists of applying a transition from T on nodes u and v, and then possibly swapping the states of u and v (in case of M_3). Let π be a permutation of V such that $C(\pi(w)) = C'(w), \forall w \in V$. Applying the transition from T is also possible in G_1, for the nodes $\pi(u)$ and $\pi(v)$ (they are certainly connected, because G_1 is a complete graph): let D be the configuration obtained by doing this. Then $D \equiv D'$.

□

Lemma 4. *Let $i \in \{2,3\}$. Let C' be reachable in G_2 (under M_i) and C reachable in G_1 (under M_1), such that $C \equiv C'$. Assume $C \to D$ is a M_1-step. Then $C' \xrightarrow{M_i*} D'$ and $D \equiv D'$ for some G_2-configuration D'.*

Proof sketch. Let $\{u, v\}$ be the edge chosen for the step $C \to D$ and π be a permutation of V such that $C(w) = C'(\pi(w)), \forall w \in V$. Let $a, b \in A(G_2)$ be the agents situated on nodes $\pi(u)$, $\pi(v)$ (see definition of agents). If $\pi(u)$ and $\pi(v)$ are neighbors in G_2, then take $D' = D$. Else, we have to find an execution to bring a and b "together", without altering their states, to allow them to interact. To do this, we use the fact that G_2 is connected: we select a fixed path from $\pi(u)$ and $\pi(v)$ and use swaps to reach a configuration C'', such that $\{pos_{C''}(a), pos_{C''}(b)\}$ is an edge in G_2 (i.e. we bring a next to b). For M_3, we apply the trivial transitions at every step to avoid altering their states. The reached configuration C'' is equivalent to C and C', i.e. $C \equiv C' \equiv C''$. Then, we allow a and b to interact. They will also possibly swap positions after interacting (in case of M_3). In both cases, this will lead to a configuration D', and $D' \equiv D$.

□

Lemma 5. *Let $i \in \{2,3\}$. Let $C \equiv C'$, where C and C' are reachable configurations of G_1 (under M_1) and G_2 (under M_i), respectively, and C' is final in G_2. Then C is final in G_1.*

Proof. Let $H_1 := \mathcal{G}(\mathcal{P}, G_1, M_1), H_2 := \mathcal{G}(\mathcal{P}, G_2, M_i)$ be the transition graphs. Let H_2' be the bottom SCC of H_2 containing C'. Let H_1' be the set of reachable G_1-configurations D such

that $D \equiv D'$ for some $D' \in H_2'$. It is clear that $C \in H_1'$. We show that if $C_1 \in H_1'$ and $C_1 \xrightarrow{*} C_2$, then $C_2 \in H_1'$ and $C_2 \xrightarrow{*} C_1$. Thus we will have proven that H_1' is a union of bottom SCCs of H_1, and it will follow that $C \in H_1'$ is final.
By definition of H_1', there exists $C_1' \in H_2'$ such that $C_1' \equiv C_1$. By (repeatedly) applying Lemma 4, we find a transition $C_2' \equiv C_2$ such that $C_1' \xrightarrow{*} C_2'$. Since H_2' is final, $C_2' \in H_2'$ and $C_2' \xrightarrow{*} C_1'$. By Lemma 3, we find a transition $\overline{C_1}$, such that $C_2 \xrightarrow{*} \overline{C_1}$ and $\overline{C_1} \equiv C_1'$. If $\overline{C_1} = C_1$, we are done. Else, we have that $C_1 \equiv C_1' \equiv \overline{C_1}$ and $C_1 \xrightarrow{*} \overline{C_1}$. Let π be a permutation of V such that $\overline{C_1}(w) = C_1(\pi(w)), \forall w \in V$. We therefore know that $C_1 \xrightarrow{*} C_1 \circ \pi$. Applying the same steps over and over, we get that $C_1 \xrightarrow{*} C_1 \circ \pi \xrightarrow{*} C_1 \circ \pi^2 \xrightarrow{*} \ldots \xrightarrow{*} C_1 \circ \pi^{ord(\pi)} = C_1$, where $ord(\pi)$ is the order of π in the permutation group S_V.
In particular, we have $\overline{C_1} = C_1 \circ \pi \xrightarrow{*} C_1$, which shows that $C_2 \xrightarrow{*} C_1$. This completes the proof. □

Note that the converse holds as well, i.e. if C is final in G_1, then C' is final in G_2. The proof is similar to the one we have given (swap the roles of H_1 and H_2, swap uses of Lemma 3 and Lemma 4).

We are now ready to prove the first important result:

Theorem 1. *Let $i \in \{2, 3\}$. If $\mathcal{P}$ computes a predicate ψ on G_1 under M_1, then it computes ψ on G_2 under M_i.*

Proof. Let C_0 be an initial configuration. We consider a fair run $w := C_0, \ldots, C_l, \ldots$ in G_2. Because there are only finitely many configurations, there is a configuration C' occurring infinitely often in w. By Lemma 1, C' is final. By Lemma 3, there is a reachable (from C_0) G_1-configuration C such that $C' \equiv C$. By Lemma 5, C is final in G_1. Because $\mathcal{P}$ computes ψ on G_1, we know that $O(C(V)) = \{\psi(C_0)\}$. Because $C' \equiv C$, we get that $O(C'(V)) = \{\psi(C_0)\}$. The configurations in the bottom SCC of C have the same output as C (else $\mathcal{P}$ would not compute ψ correctly on G_1), so C is stable. This implies that C' is stable, because any configuration reachable from C' is equivalent to a configuration reachable from C, by Lemma 3. We have shown that the fair runs on G_2 under M_i reach a correct stable consensus. Because almost all runs are fair (Lemma 2), we conclude that $\mathcal{P}$ correctly computes ψ on G_2. □

Note that the converse holds as well, i.e. if $\mathcal{P}$ computes ψ on G_2, then it computes ψ on G_1. The proof is analogous to the one above (swap the roles of G_1 and G_2, swap uses of Lemma 3 and Lemma 4, use the reciprocal of Lemma 5 instead of Lemma 5).

4 Performance

4.1 Convergence Speed

4.1.1 Preamble

We are given a predicate ψ, a scheduler M, and an interaction graph G. Can we find some protocol $\mathcal{P}$, that computes ψ, and for which we can determine the expected number of steps until the protocol reaches a stable consensus?

A partial answer is known from Angluin et al. (Theorem 22 [1]): if the scheduler is M_1 (and so the interaction graph is the complete graph), then there exists a protocol $\mathcal{P}(\psi)$, which calculates ψ, and whose expected time until the population reaches a stable consensus is $\mathcal{O}(n^2 \log(n))$, where $n := |V|$ is the number of agents/nodes. To prove the existence, the authors use a constructive approach, which we will analyze and discuss in this section. We call a protocol $\mathcal{P}(\psi)$ obtained from the construction in [1] the **canonical** protocol for the predicate ψ.

In the following definitions, we formalize the idea that a protocol makes irrevesible progress towards the goal of reaching stable consensus:

Stage Vectors

Consider a scheduler M, an interaction graph G, and a protocol $\mathcal{P}$ running on G under M, with initial configuration C_1. Assume that there exists a finite tuple $\beta := (\phi_1, \ldots, \phi_r)$ ($r \in \mathbb{N}$), of unary predicates on configurations of $\mathcal{P}$, such that:

- $C_1 \in \phi_1$
- $\phi_{i+1} \subseteq \phi_i$ for all $i \in [r-1]$

- for all $i \in [r]$, if $C_1 \xrightarrow{*} C_i$ for some $C_i \in \phi_i$, then for all configurations $C_i \xrightarrow{*} C$ we have that $C \in \phi_i$, i.e. if the protocol reaches a configuration fulfilling ϕ_i, then ϕ_i will remain valid for all future configurations
- for all $i \in [r]$, the set of runs that eventually reach a configuration $C_i \in \phi_i$ has measure 1 in $Run(C_1)$, i.e. a run reaches a ϕ_i-fulfilling configuration almost surely

We call ϕ_i a *stage*. If ϕ_r is included in the set of stable consensuses, then we say that ϕ_r is a *final stage* of $\mathcal{P}$. β is called the *stage vector* of $\mathcal{P}$ when run on G under M starting in C_1. A more refined definition of similar concepts can be found in [14], where *stage graphs* are introduced. The given definition of stage vectors is precise enough for our purposes.
For a configuration $C_i \in \phi_i$ ($i \in [r-1]$), let $\Pi(C_i)$ be the expected number of steps, until a configuration $C_{i+1} \in \phi_{i+1}$ is reached. Because of the last condition in the definition, we know that $\Pi(C_i) < \infty$, which allows us to define $\Pi(i) := \max\{\Pi(C_i) : C_i \in \phi_i\}$ and $\Pi(\beta) := \max\{\Pi(i) : i \in [r-1]\}$. Because the number of configurations is finite, we have $\Pi(i) < \infty$ and therefore $\Pi(\beta) < \infty$.

Proposition. If ϕ_r is a final stage, we know that the expected number of steps until $\mathcal{P}$ reaches stable consensus (from C_1) is bounded by $\mathcal{O}(r\Pi(\beta))$.

Using the previously introduced terminology, we will discuss the results from [1] and show how to find bounds for the expected time of reaching stable consensus for canonical protocols running on arbitrary interaction graphs:

Threshold and Remainder

Two classical problems that can be solved with Population Protocols are Threshold and Remainder. We are given parameters $m \geq 2$, a_i ($1 \leq i \leq n$), and $c \geq 1$, where n is the size of the population. The initial configurations can be seen as a set of vectors $(x_i) \in [s]^n$ of length n, where $s := \max\{|c| + 1, m, \max a_i\}$. The predicates to compute for Threshold and Remainder are, respectively:

- $\sum_{i=1}^{n} a_i x_i < c$

- $\sum_{i=1}^{n} a_i x_i \equiv c (\text{mod } m)$

The authors construct canonical protocols $\mathcal{P}_1, \mathcal{P}_2$ for computing these two predicates on complete graphs. Let C_1 be a fixed initial configuration. Both protocols admit a stage vector β of length $2n$ (starting in C_1), i.e. use $2n$ stages to reach stable consensus, which we informally describe:

- *Election stages*: during the first n stages, there exist special agents (i.e. agents that have a certain bit set to 1 in their state), which we call *leaders*. The stages $\phi_1, \ldots, \phi_n$ of this vector are $\phi_i :=$ set of configurations C_i with exactly $n - i + 1$ leaders. The construction in [1] guarantees that these are indeed stages (i.e. the number of leaders never increases). Whenever two leaders l_1, l_2 with states q_1, q_2 non-trivially interact (i.e. a non-trivial transition $q_1 q_2 \to p_1 p_2$ is applied), exactly one of the both agents becomes a non-leader (its leader bit is set to 0), so the number of leaders decreases and, therefore, we advance to the next stage.

- *Collection stages*: during the next n stages, there is a special agent x (the unique leader). The other agents always form a (X, Y, Z) partition. X is the set of *active* agents, i.e. if x interacts non-trivially with $y \in X$, the stage changes, and y becomes *consumed*. Y is the set of *inactive* agents, i.e. if x interacts non-trivially with $y \in Y$, the stage remains invariant, but at a later point in time y might become active. Z is the set of *consumed* agents, i.e. no interaction of x with any of them can (ever) change the stage. The stages $\phi'_1, \ldots, \phi'_n$ of this vector are $\phi'_i :=$ set of configurations with $|Z| \geq i - 1$, i.e. at least $i - 1$ consumed agents. The construction in [1] guarantees that these are indeed stages (i.e. a consumed agent stays consumed and, as long as $|Z| < n - 1$, we have $|X| > 0$), and ϕ'_n is a final stage (if $|Z| = n - 1$, stable consensus has been reached).

Fix an interaction graph G and a scheduler $M \in \{M_2, M_3\}$. We first prove the following:

Proposition. β is a stage vector (starting in C_1) for $\mathcal{P}_1/\mathcal{P}_2$ even if the protocols run on G under M.

Proof. We have to prove the third and fourth conditions from the definition of stage vectors. For that, first fix a non-final stage ζ, and a configuration $C_i \in \zeta$. If there is a configuration C', such that $C_1 \overset{*M}{\to} C'$ and $C' \notin \zeta$, then by Lemma 3 there is also a reachable configuration

$C'' \equiv C'$, such that $C_i \overset{*M_1}{\to} C''$. Because $C'' \equiv C'$, it follows that $C'' \notin \zeta$ (easy to see that all stages of β are closed under equivalence), which contradicts the fact that ζ is a stage of $\mathcal{P}_1/\mathcal{P}_2$ when run under M_1 (on a complete graph). This proves the third condition of the definition. Next, recall Lemma 2: the set of fair runs has measure 1 in $Run(C_1)$. For the election stages, every fair run guarantees that all pairs of leaders will eventually interact non-trivially, thus leading to just one leader: so the set of runs eventually reaching a configuration from ψ_i has measure 1 for all $i \in [n]$. For the collection stages, when we have just one leader, fairness guarantees that the leader will eventually consume all other agents: so the set of runs eventually reaching a configuration from ψ_i' also has measure 1 for all $i \in [n]$. This proves the fourth condition of the definition, so β **remains a stage vector for** $\mathcal{P}_1/\mathcal{P}_2$ **even if the protocols run on** G **under** M. □

There are $2n$ stages. Assume that we know $\Pi(\beta)$ (under the assumption that we run on G under M). Then, by the last observation in the definition of stage vectors, the expected time until reaching consensus from any initial configuration (for $\mathcal{P}_1/\mathcal{P}_2$) is bounded by $\mathcal{O}(n\Pi(\beta))$. We want an upper bound for $\Pi(\beta)$ for the protocols $\mathcal{P}_1/\mathcal{P}_2$ running on G under M. Let $A(G)$ be the set of agents of G (see definition of agents). Because the probability of swapping at any step is $0 < p < 1$, every 2 agents will almost surely interact non-trivially infinitely often.

Let $\mathcal{T}(u, v)$ be the expected time until two agents situated on nodes $u, v \in V$ interact. Of course, $\mathcal{T}(u, v)$ depends only on the parameter p of the scheduler M and on the interaction graph G, and not on the particular protocol, configuration or scheduler. Define $\mathcal{T} := \max\{\mathcal{T}(u, v) : u, v \in V\}$. $\mathcal{O}(\mathcal{T})$ is an upper bound for the expected time until two agents interact non-trivially (assuming that their states allow non-trivial transitions), no matter their initial positions. It is easy to see that $\Pi(\beta) \in \mathcal{O}(\mathcal{T})$: every stage change occurs for the protocol $\mathcal{P}_1/\mathcal{P}_2$ exactly when two "chosen" agents interact non-trivially, and this expectedly cannot take more than $\mathcal{O}(\mathcal{T})$ steps. We have proven the following:

Lemma 6. *Let G be an interaction graph. Let $\mathcal{T}$ be defined as before. Running the canonical protocols $\mathcal{P}_1/\mathcal{P}_2$ using $M \in \{M_2, M_3\}$ on G requires an expected time of no longer that $\mathcal{O}(n\mathcal{T})$ steps to reach a stable consensus.*

General Predicates

Let ψ be an arbitrary (computable) predicate. For the construction of the canonical protocol $\mathcal{P} := \mathcal{P}(\psi)$, the authors prove the following result [1]: There exist *base predicates* (i.e. Threshold/Remainder) $\psi_1, \ldots, \psi_k$, such that computing ψ can be done by:

- Running the canonical *base protocols* $\mathcal{P}_1, \ldots, \mathcal{P}_k$ for the base predicates in parallel, until they all reach stable consensus.
- Collecting the results of the computations of $\mathcal{P}_1, \ldots, \mathcal{P}_k$.

To compute protocols in parallel, the authors use a product construction, similar to the product construction for DFAs (Deterministic Finite Automata), i.e. by designing a protocol whose transitions correspond to firing a transition to $\mathcal{P}_1, \ldots, \mathcal{P}_k$ simultaneously, thus leading to k parallel computations.

Formally, this corresponds to a stage vector β of length $k + n$ (starting in some initial configuration C_1) for $\mathcal{P}$, if $\mathcal{P}$ runs on complete graphs. We describe the stages:

- *Waiting stages*: during the first k stages, we wait for the base protocols $\mathcal{P}_1, \ldots, \mathcal{P}_k$ (which are run in parallel) to reach stable consensus. The stages $\phi_1, \ldots, \phi_k$ are $\phi_i :=$ set of configurations C_i for which at least i of the base protocols reach stable consensus.
- *Collection stages*: like before (for Threshold/Remainder), the unique agent x must collect the results from all other agents for (global) stable consensus to be reached.

We fix an interaction graph G (with set of nodes V, $|V| = n$) and a scheduler $M \in \{M_2, M_3\}$. Like in the analysis of Threshold/Remainder, it is easy to see that β remains a stage vector (starting in C_1) for $\mathcal{P}$ even if it runs on G under M. For the waiting stages, this follows because the canonical protocols $\mathcal{P}_i$ eventually reach stable consensus (the set of runs where their parallel computation reaches stable consensus has measure 1), and their stable consensus is irreversible. For the collection stages, the argumentation is identical to the one we gave for Threshold/Remainder predicates.

Let $E[T_i]$ be the expected time until $\mathcal{P}_i$ reaches stable consensus. From Lemma 6, we know that $E[T_i] = \mathcal{O}(n\mathcal{T})$, where $\mathcal{T} := \max\{\mathcal{T}(u, v) : u, v \in V\}$, and $\mathcal{T}(u, v) :=$ expected time until

two agents situated on nodes $u, v \in V$ interact. The expected time for computing $\mathcal{P}_1, \ldots, \mathcal{P}_k$ in parallel is bounded by $E[\sum_{i=1}^{k} T_i] = \sum_{i=1}^{k} E[T_i] = \mathcal{O}(kn\mathcal{T}) = \mathcal{O}(n\mathcal{T})$, because k is a constant, depending only on the predicate ψ. Then, arguing as for the Threshold/Remainder predicates, the expected time for the collection stages is also bounded by $\mathcal{O}(n\mathcal{T})$.

Thus, we conclude:

Lemma 7. *Let G be an interaction graph. Let $\mathcal{T}$ be defined as before. Running the canonical protocol $\mathcal{P}(\psi)$ of a predicate ψ using $M \in \{M_2, M_3\}$ on G requires an expected time of no longer than $\mathcal{O}(n\mathcal{T})$ steps to reach a stable consensus.*

4.1.2 Results

Our objective is therefore to bound the expected number of steps until any two agents interact. For this, **we will use the scheduler** M_2. The calculations for M_3 will prove to be very similar. Let d be the *diameter* of the interaction graph G (i.e. $\max\{dist(u, v) : u, v \in V\}$). We pessimistically fix two agents $u, v \in A(G)$, whose initial positions in the graph G are at distance d (maximum possible distance) apart. Denote the expected number of steps until u, v interact for the first time with $\mathcal{T}_d := \mathcal{T}(u, v)$. Let p be the probability of swapping and $q := 1/|E|$. We assume $d \geq 2$ and $|E| \geq 5$. To calculate the expected time (i.e. number of steps) until u and v interact we construct a pessimistic discrete Markov Chain Γ, whose states are $i \in [d]_0$. State 0 is absorbing, it means that u and v already interacted. State $i > 0$ means that u and v have not yet interacted, and their current positions in the graph $pos(u)$ and $pos(v)$ are at distance i apart. We will obtain the following chain Γ:

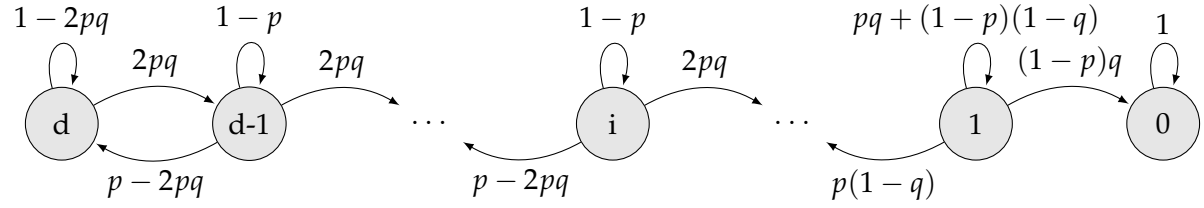


Figure 4.1: Chain modeling distance between two nodes

We justify that our chain is indeed pessimistic, i.e. the probabilities we used for $P(i \to i-1)$

are lower bounds of the real ones and the probabilities we used for $P(i \to i+1)$ are upper bounds of the real ones.

- For $i = d \geq 2$: there is at least one u-v-path in G, and every path has at least 2 edges, so there must be at least 2 edges which, if a swap is applied on them, bring u and v closer together. Swapping along these 2 edges happens with a probability $2pq$.
- For $2 \leq i \leq d-1$. Like in the previous case, the probability of getting closer is at least $2pq$. If the 2 "good" edges are not selected, or if no swap happens (probability $1-p$), then we make the (very pessimistic) assumption that, whatever happens, the distance between u and v rises. So we use $P(i \to i+1) \leq 1 - 2pq - (1-p) = p - 2pq$.
- For $i = 1$: u and v interact iff the edge connecting them is chosen, and a transition is applied. This happens with probability $q(1-p)$. We pessimistically assume that the distance between them grows if any other edge is selected for a swap (this happens if all edges of G are connected to u or v), and so $P(1 \to 2) \leq (1-q)p$.

There is one obvious improvement which we could make, by observing that, if u and v are at distance $i \geq 2$ apart, then at least $i-2$ edges produce no ill effects if we swap along them. Unfortunately, we did not succeed in deriving a better bound by using this more sane version. We therefore stick to the presented chain.

The next step is solving the chain. For this, let X_i ($i \geq 0$) be random variables indicating the current state in the chain after i steps and $z_i := E[\min\{n \geq 0 : X_{n+k} = 0 \text{ if } X_k = i\}]$ ($i \in [d]$) be the expected hitting times from state i to state 0. (This is well defined because of the stationary property, see Chapter 2.) By the definition of $\mathcal{T}_d$, we have that:

$$\mathcal{T}_d = z_d \tag{4.1}$$

Define $a_{i-1} := z_i - z_{i-1}$ for $i \in [d]$. We use the known hitting time equations for Markov chains (see Chapter 2). For $i = d$, $z_d = 1 + (1-2pq)z_d + 2pqz_{d-1}$, so:

$$z_d = z_{d-1} + \frac{1}{2pq},\ a_{d-1} = \frac{1}{2pq}$$

For $2 \leq i \leq d-1$: $z_i = 1 + (1-p)z_i + (p-2pq)z_{i+1} + 2pqz_{i-1}$. Substituting $z_i + a_i$ for z_{i+1},

we get $2pqz_i = 1 + 2pqz_{i-1} + (p - 2pq)a_i$, so:

$$z_i = z_{i-1} + \frac{1 + (p - 2pq)a_i}{2pq},\ a_{i-1} = \frac{1 + (p - 2pq)a_i}{2pq}$$

Define $b_i := a_{d-1-i}$, $0 \leq i \leq d - 1$. We have shown that:

$$b_0 = \frac{1}{2pq},\ b_{i+1} = \left(\frac{1}{2q} - 1\right) b_i + \frac{1}{2pq}$$

(Expression is well defined because our assumption $|E| \geq 5$ implies $q \leq 1/5$).
Solving for b_m ($m \geq 0$) gives:

$$b_m = \left(\frac{1}{2q} - 1\right)^m \frac{1}{2pq} \left(1 + \sum_{l=0}^{m-1} \frac{1}{(\frac{1}{2q} - 1)^{l+1}}\right)$$

Using $1 + \sum_{l=0}^{m-1} \frac{1}{(\frac{1}{2q}-1)^{l+1}} \leq \sum_{l=0}^{\infty} \frac{1}{(\frac{1}{2q}-1)^l} = \frac{1-2q}{1-4q}$, we get:

$$b_m \leq \frac{1}{p(1-4q)} \left(\frac{1}{2q} - 1\right)^{m+1} \tag{4.2}$$

Now let $c_i := b_0 + \cdots + b_i$. Using the previous inequality, we get:

$$c_m = \frac{1}{p(1-4q)} \sum_{i=0}^{m} \left(\frac{1}{2q} - 1\right)^{i+1} = \frac{1}{p(1-4q)} \cdot \left(\frac{1}{2q} - 1\right) \frac{\left(\frac{1}{2q} - 1\right)^{m+1} - 1}{\frac{1}{2q} - 2} \leq$$

$$\leq \frac{1}{p(1-4q)} \cdot \left(\frac{1}{2q} - 1\right) \frac{\left(\frac{1}{2q} - 1\right)^{m+1}}{\frac{1}{2q} - 2} = \frac{1-2q}{p(1-4q)^2} \left(\frac{1}{2q} - 1\right)^{m+1} \leq \frac{2}{p(1-4q)} \left(\frac{1}{2q} - 1\right)^{m+1}$$

For the last inequality we used $(1 - 2q)/(1 - 4q) \leq 2$. The bound is:

$$c_m \leq \frac{2}{p(1-4q)} \left(\frac{1}{2q} - 1\right)^{m+1} \tag{4.3}$$

The hitting time equation for $i = 1$ is $(p + q - 2pq)z_1 = 1 + p(1 - q)z_2$. Substituting $z_1 + b_{d-2}$ for z_2, we get $(q - pq)z_1 = 1 + p(1 - q)b_{d-2}$. Using (4.2), this leads to:

$$z_1 \leq \frac{1}{q(1-p)} + \frac{2(1-q)}{(1-2q)(1-p)(1-4q)} \left(\frac{1}{2q} - 1\right)^d$$

Because $(1-q)/(1-2q) \leq 2$, we can write:

$$z_1 \leq \frac{1}{q(1-p)} + \frac{4}{(1-p)(1-4q)} \left(\frac{1}{2q} - 1\right)^d \tag{4.4}$$

We have that $z_d = z_{d-1} + c_0 = z_{d-2} + c_1 = \cdots = z_1 + c_{d-2}$. Inequalities (4.4) and (4.3) give us bounds for both z_1 and c_{d-2}. Plugging them in, we get:

$$z_d \leq \frac{1}{q(1-p)} + \frac{4}{(1-p)(1-4q)} \left(\frac{1}{2q} - 1\right)^d + \frac{2}{p(1-4q)} \left(\frac{1}{2q} - 1\right)^{d-1} \tag{4.5}$$

From (4.1) we know that $\mathcal{T}_d = z_d$. We get:

$$\mathcal{T}_d \leq \frac{1}{q(1-p)} + \frac{4}{(1-p)(1-4q)} \left(\frac{1}{2q} - 1\right)^d + \frac{2}{p(1-4q)} \left(\frac{1}{2q} - 1\right)^{d-1} \tag{4.6}$$

Recall that these calculations hold for M_2. What about M_3? The pessimistic Markov chain for M_3 is almost identical to Γ: the only difference in the chain is that $P(1 \to 0) = q$, because M_3 guarantees that a transition is applied, and swaps are only done after applying transitions. For this scheduler we get:

$$z_1 \leq \frac{1}{q} + \frac{4}{1-4q} \left(\frac{1}{2q} - 1\right)^d,\ z_d \leq z_1 + \frac{2}{p(1-4q)} \left(\frac{1}{2q} - 1\right)^{d-1} \tag{4.7}$$

and thus:

$$\mathcal{T}_d \leq \frac{1}{q} + \frac{4}{1-4q} \left(\frac{1}{2q} - 1\right)^d + \frac{2}{p(1-4q)} \left(\frac{1}{2q} - 1\right)^{d-1} \tag{4.8}$$

We use Lemma 7 to derive an upper bound for the schedulers M_2/M_3. It is easy to see that, with the notations from the Lemma, we have that $\mathcal{T} = \mathcal{T}_d$. Recall that $1/q = |E|$, where E is the set of edges of the interaction graph G. Using the inequalities (4.6), (4.8), the fact that $\frac{1}{1-4q} \leq 2$, and treating p as a constant, we obtain:

Theorem 2. *The canonical protocol $\mathcal{P}(\psi)$ of a predicate ψ expectedly reaches a stable consensus after at most*

$$\mathcal{O}\left(\frac{n|E|^d}{2^d}\right)$$

steps on M_2/M_3, where d is the diameter of the interaction graph.

Let us analyze the more precise bounds from inequalities (4.6) and (4.8). For M_2, if $p \to 1$, then we only do swaps, and the agents are no longer allowed to interact, i.e. no transitions are applied. This obviously leads to bad performance, as the bound suggests. However, for M_3, this is not a problem, because swapping is only done after transitioning, and making $p \to 1$, i.e. always swapping, seems to be optimizing performance in the bound. For both M_2 and M_3, making $p \to 0$ leads to catastrophic results, because if nodes are not swapped then there will be pairs of agents that will never "reach" each other. So our bounds are sensitive to changes of the scheduling parameter p. Let us look at an example to show that the bound from Theorem 2 can be very imprecise.

Consider the following line interaction graph $G := (V, E)$, where $|V| := [n]$ and $E := \{\{i, i+1\} : i \in [n-1]\}$. So $d = n - 1$. We calculate an upper bound for the expected time $\mathcal{T}_d$ until two fixed agents $u, v \in A(G)$ at distance d interact (when using the scheduler M_2) by constructing a pessimistic Markov chain, whereby the states have the same meaning as before. The following chain is pessimistic:

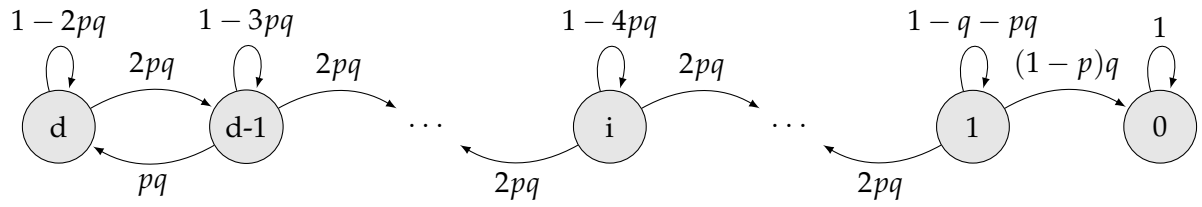


Figure 4.2: Chain for Line Graphs

We denote, again, z_i the hitting time from state i to state 0 for $i \geq 1$. By the choice of u,v, similarly to equation (4.1), we can write:

$$\mathcal{T} = \mathcal{T}_d = z_d$$

With the known notations, this time we obtain: $b_0 = \frac{1}{2pq}$, $b_1 = \frac{3}{4pq}$, $b_{i+1} = b_i + \frac{1}{2pq}$ for $i \geq 1$, so for $m \geq 1$:

$$b_m = b_1 + \frac{m-1}{pq} = \frac{4m-1}{4pq} \leq \frac{m}{pq}, \quad c_m \leq \frac{m(m+1)+1}{2pq}$$

and then, because $(q - pq)z_1 = 1 + pqb_{d-2} \leq d - 1$ and $z_d = z_1 + c_{d-2}$:

$$z_1 \leq \frac{d-1}{q(1-p)}, \;\; z_d \leq \frac{d-1}{q(1-p)} + \frac{d^2 - 3d + 3}{2pq}$$

and thus, remembering that $\frac{1}{q} = d = |E| = n - 1$, and treating p as a constant:

$$\mathcal{T} = \mathcal{O}(|E|^3)$$

We obtain (Lemma 7) a running time of $\mathcal{O}(n|E|^3) = \mathcal{O}(|E|^4)$. This is very far away from the pessimism of our bound from Theorem 2, which would indicate $\mathcal{O}(|E|^{|E|+1})$. The same problem arises if we do the calculations for M_3 (as expected, again only the factor $1/(1-p)$ will disappear). We will now present a second proof, which leads to a bound that, except for some (non-trivial) edge cases, is better than the one from Theorem 2:

Recall the notation $q := 1/|E|$ (meaning that $1/q = \mathcal{O}(n^2)$) and the (connected) interaction graph $G := (V, E)$ of diameter d. We find a bound (for the scheduler M_2) for $\mathcal{T}$ (again, using the notation known since Lemma 7). For this proof, let $deg(i)$ be the degree of $i \in V$ in G, and $N(i)$ be the set of neighbors of i. We also set, for $2 \leq k \leq d$, $V_k := \{\{i,j\} : i,j \in V,\ dist(i,j) = k\}$, and analogously $V_{\geq k}$. For $1 \leq k \leq d$, define $\mathcal{T}_k$ to be the expected time until two agents interact, assuming that their distance in the interaction graph is k. Also define γ_{ijk} to be the cardinality of the set $\{l \in N(i) : dist(l,j) = k-1\} \cup \{l \in N(j) : dist(i,l) = k-1\}$.

For the beginning, we consider the Markov chain given by the states $\{i,j\} \in \binom{V}{2}$ and 0, where state $\{i,j\}$ signifies that the two agents are on the nodes i and j of the graph, and the absorbing state 0 signifies that the two agents interacted. The transitions of this chain are simple: if l is a neighbor of i, different from j, then $P(\{i,j\} \to \{l,j\}) = pq$, because the (single) edge connecting i and l must be chosen (probability q), and we need to swap (probability p). If $\{i,j\} \in E$, then we also set $P(\{i,j\} \to 0) = (1-p)q$, because an interaction means that the edge connecting i and j is chosen (probability q), and we need to not swap (probability $1-p$). Let X_i be random variables indicating the current state in the chain after i steps. We use, for $i,j \in V$, $i \neq j$, the notations $z_{ij} := E[\min\{n \geq 0 : X_{k+n} = 0 \text{ if } X_k = \{i,j\}\}]$ for the expected hitting times from state $\{i,j\}$ to state 0.

Let us analyze the hitting time equations. For $\{i,j\} \in E$, we get:

$$((deg(i) + deg(j) - 2)pq + (1-p)q))z_{ij} = 1 + pq \sum_{l \in N(i), l \neq j} z_{lj} + pq \sum_{l \in N(j), l \neq i} z_{il} \tag{4.9}$$

and for $\{i,j\} \in \binom{V}{2} \setminus E$:

$$(deg(i) + deg(j))pqz_{ij} = 1 + pq \sum_{l \in N(i), l \neq j} z_{lj} + pq \sum_{l \in N(j), l \neq i} z_{il} \tag{4.10}$$

Adding up all the $\binom{n}{2}$ equations from (4.9) and (4.10), we obtain:

$$\sum_{\{i,j\} \in E} (1-p)qz_{ij} = \binom{n}{2}$$

which implies that, for any $\{i,j\} \in E$, we have:

$$z_{ij} \leq \binom{n}{2} \frac{1}{(1-p)q}$$

so, remembering that that $\mathcal{T}_1 \leq \max\{z_{ij} : \{i,j\} \in E\}$:

$$\mathcal{T}_1 \leq \binom{n}{2} \frac{1}{(1-p)q} \tag{4.11}$$

Let $2 \leq k \leq d$. A similar trick will prove useful for bounding $\mathcal{T}_k$. Indeed, we now take the Markov chain given by the states $\{i,j\} \in V_{\geq k}$ and 0, where state $\{i,j\}$ signifies that the two agents are on the nodes i and j of the graph, and the absorbing state 0 signifies that the two agents are at distance $< k$ from each other. The transitions of this chain are very similar to the previous case: if l is a neighbor of i, such that $\{l,j\} \in V_{\geq k}$, then $P(\{i,j\} \to \{l,j\}) = pq$. If $\{i,j\} \in V_k$, then we also set $P(\{i,j\} \to 0) = \gamma_{ijk}pq$, because γ_{ijk} is defined as the number of distinct edges that bring i and j closer. We again use the notations z_{ij} for the expected hitting times from state $\{i,j\}$ to state 0.

For $\{i,j\} \in V_{\geq k}$, we get:

$$(deg(i) + deg(j))pqz_{ij} = 1 + pq \sum_{l \in N(i), \{l,j\} \in V_{\geq k}} z_{lj} + pq \sum_{l \in N(j), \{i,l\} \in V_{\geq k}} z_{il}$$

If $\{i,j\} \in V_k$, then z_{ij} appears on the right hand side of these equations exactly $deg(i) + deg(j) - \gamma_{ijk}$ times. Else, it appears exactly $deg(i) + deg(j)$ times. Therefore, adding up all equations leads to:

$$\sum_{\{i,j\} \in V_k} \gamma_{ijk} pq z_{ij} = |V_{\geq k}|$$

Let $\{i,j\} \in V_k$ be fixed. Considering that $\gamma_{ijk} \geq 2$ and $|V_{\geq k}| < \binom{n}{2}$, the previous equation implies:

$$z_{ij} \leq \binom{n}{2} \frac{1}{2pq}$$

so, remembering that $\mathcal{T}_k \leq \max\{z_{ij} : \{i,j\} \in V_k\} + \mathcal{T}_{k-1}$:

$$\mathcal{T}_k \leq \binom{n}{2} \frac{1}{2pq} + \mathcal{T}_{k-1} \tag{4.12}$$

Inequalities (4.11) and (4.12) lead to:

$$\mathcal{T}_d \leq (d-1)\binom{n}{2} \frac{1}{2pq} + \binom{n}{2} \frac{1}{(1-p)q} \tag{4.13}$$

Using the fact that $\mathcal{T} = \mathcal{T}_d$, we get:

$$\mathcal{T} \leq (d-1)\binom{n}{2} \frac{1}{2pq} + \binom{n}{2} \frac{1}{(1-p)q} \tag{4.14}$$

For the scheduler M_3, again, everything is identical, except that the factor $1/(1-p)$ disappears. Treating p as a constant, and remembering that $1/q = |E|$, we obtain the following as a consequence of Lemma 7:

Theorem 3. *The canonical protocol $\mathcal{P}(\psi)$ of a predicate ψ expectedly reaches a stable consensus after at most*

$$\mathcal{O}(dn^3|E|)$$

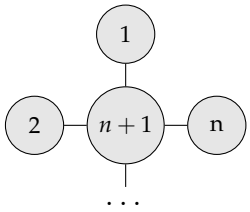
steps on M_2/M_3, where d is the diameter of the interaction graph.

Note that the bound from Theorem 3 is not always better than the one from Theorem 2. For example, if $|E| \in \Theta(n)$ and $d = 2$, Theorem 2 gives $\mathcal{O}(n^3)$, while Theorem 3 gives $\mathcal{O}(n^4)$. However, Theorem 3 gives, in general, better bounds; for the line example, it gives a bound of $\mathcal{O}(n^5)$, which is much closer to the cubic bound we found than what Theorem 2 indicates. For the examples that follow, we choose the method from the proof of Theorem 2, which is easier to use for concrete graphs. Let us investigate whether there are any graph families for which the bounds from the Theorems 2 and 3 are tight. We will analyze three family of graphs for the scheduler M_2, and do some *exact* calculations. The calculations for M_3 are very similar and differ only by a factor of $1/(1-p)$, so we will not reproduce them.

4.1.3 Examples

Star Graph

Consider the interaction graph $G := (V, E)$, where $V := [n+1]$ and $E := \{\{i, n+1\} : i \in [n]\}$. It looks like this:



The corresponding *exact* Markov chain for the distance between the two selected agents is given in Figure 4.3.

Again, we use the hitting time equations. For $i = 2$, we get $z_2 = z_1 + \frac{1}{2pq}$. It follows, for $i = 1$:

$$(q + (n-2)pq)z_1 = 1 + (n-1)pq(z_1 + \frac{1}{2pq})$$

and therefore:

$$z_1 = \frac{n+1}{2q(1-p)}, \quad z_2 = \frac{n+1}{2q(1-p)} + \frac{1}{2pq}$$

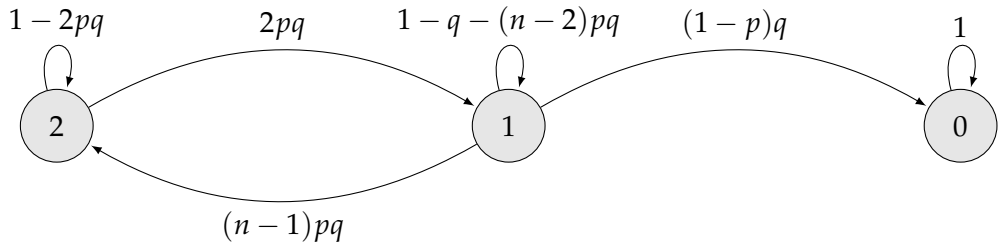


Figure 4.3: Chain for Star Graph

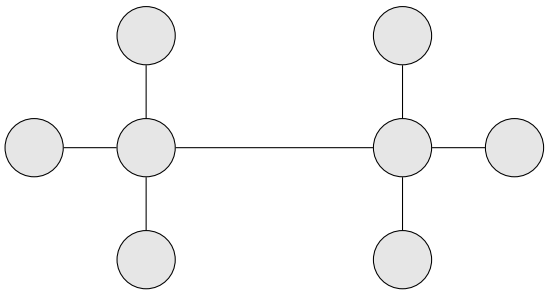
Remembering that $\frac{1}{q} = |E| = n$ and using $\mathcal{T} = z_2$:

$$\mathcal{T} = \mathcal{O}(n^2)$$

leading to a running time (Lemma 7) of $\mathcal{O}(n^3)$, and is close to our bound in Theorem 3, which gives $\mathcal{O}(n^4)$ (because $d = 2$). The bound of Theorem 2 is $\mathcal{O}(n^3)$ is tight. Will this also work for higher d? Let us extend this graph for $d = 3$.

Extended Star Graph

The idea behind this version is apparent in the following picture, which shows an example for $n = 8$:



The graph is built up from 2 star components, each consisting of $k := n/2$ nodes. The two "central" nodes are directly connected, i.e. the central nodes each have k neighbors: the $k-1$ other nodes from their own component, and the other central node.

Constructing an *exact* Markov chain for this graph is more complicated than before. We call a non-central node a *trapped* node, i.e. our graph has $2k - 2 = n - 2$ trapped nodes. If an

agent is positioned on a trapped node, it cannot move anywhere except the central node from the same component. For the Markov chain, we use the following convention for our state notations: state i means that the agents are at distance i, and both located on central nodes; state i' implies distance i, but exactly one agent is on a trapped node; state i'' implies distance i, and both agents are on trapped nodes. We obtain the chain in Figure 4.4.

Using $z_3'' = z_2' + \frac{1}{2pq}$, we get:

$$(k+1)pqz_2' = 1 + (k-1)pq\left(z_2' + \frac{1}{2pq}\right) + pqz_1 + pqz_1',$$

$$z_2' = \frac{k+1}{4pq} + \frac{z_1}{2} + \frac{z_1'}{2} \tag{4.15}$$

Now we can express z_1 using only z_1':

$$(q + (2k-3)pq)z_1 = 1 + 2(k-1)pq\left(\frac{k+1}{4pq} + \frac{z_1}{2} + \frac{z_1'}{2}\right),$$

$$z_1 = \frac{k^2+1}{2q(1+(k-2)p)} + \frac{(k-1)p}{1+(k-2)p}z_1' \tag{4.16}$$

Using $z_2'' = z_1' + \frac{1}{2pq}$ and (4.9), the hitting time equation for z_1' reveals:

$$(q + (k-1)pq)z_1' = 1 + (k-1)pqz_2'' + pqz_2',$$

$$(q + (k-1)pq)z_1' = 1 + (k-1)pq\left(z_1' + \frac{1}{2pq}\right) + pq\left(\frac{k+1}{4pq} + \frac{z_1}{2} + \frac{z_1'}{2}\right),$$

which, after solving for z_1', leads to:

$$z_1' = \frac{3(k+1)}{q(2-p)} + \frac{pz_1}{2-p} \tag{4.17}$$

Because for this family of graphs we have $k = n/2 \in \Theta(n)$ and $1/q = n - 1 = \Theta(n)$, we easily see that $z_1' = \Theta(n^2) + \Theta(1)z_1$ and $z_1 = \Theta(n^2) + \Theta(1)z_1'$, so $z_1' = \Theta(n^2)$, $z_1 = \Theta(n^2)$. Recalling (4.15), this leads to $z_2' = \Theta(n^2)$ and finally $z_3'' = \Theta(n^2)$. This implies $\mathcal{T} = \Theta(n^2)$ and a running time (Lemma 7) of $\mathcal{O}(n^3)$. What does this mean? By extending our star graph from

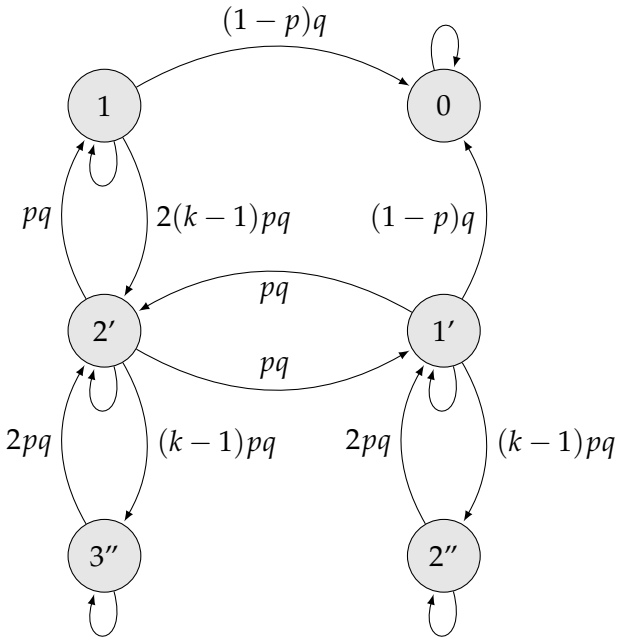
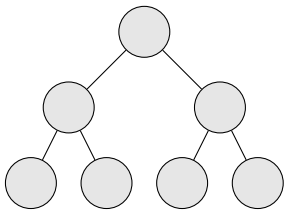


Figure 4.4: Chain for Extended Star Graph

$d = 2$ to $d = 3$, we have seen that the bound remains in $\mathcal{O}(n^3)$. Our bounds from Theorems 2 and 3 would indicate $\mathcal{O}(n^4)$, so the bounds are no longer tight.

Binary Tree

In the following we analyze a perfect binary tree with $n := 2^{d+1} - 1$ nodes, i.e. all interior nodes have two children and all leaves have the same *depth* (= distance from root) $d \geq 1$. Below you can see an example for $d = 2$:



Assume that the two agents are located on leaves of opposite sides. Determining how long it takes until these two agents interact is quite complicated, but we definitely know that at

least one agent needs to reach the root for such an interaction to become possible. We will calculate how long it takes for one agent located in a leaf to reach the root. This will lead to a lower bound for $\mathcal{T}$. The *exact* Markov chain for this new problem is easy to design:

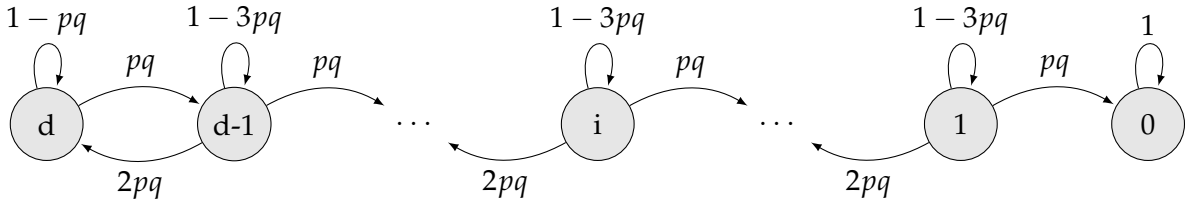


Figure 4.5: Chain for Binary Graph

The states signify the distance from the agent to the root of the tree. We are interested in z_d. We recall the notations $a_{i-1} := z_i - z_{i-1}$ for $i \in [d]$, $b_i := a_{d-1-i}$ for $0 \leq i \leq d-1$, and $c_i := b_0 + \cdots + b_i$. It is easy to see, from the hitting time equations, that $z_d = z_{d-1} + \frac{1}{pq}$, i.e. $b_0 = a_{d-1} = \frac{1}{pq}$ and $z_i = z_{i-1} + (2a_i + \frac{1}{pq})$, i.e. $b_{i+1} = 2b_i + \frac{1}{pq}$. Solving for b_m ($m \geq 0$) gives:

$$b_m = \frac{2^{m+1} - 1}{pq},\ c_m = \frac{2^{m+3-(m+3)}}{pq}$$

and therefore $z_d = z_1 + c_{d-2} = z_1 + \frac{2^{d+1}-(d+1)}{pq}$. Substituting $z_2 = z_1 + b_{d-2}$ in the hitting time equation for $i = 1$ leads to $3pqz_1 = 1 + 2pq(z_1 + b_{d-2})$, and we get:

$$z_1 = \frac{2^d - 1}{pq},\ z_d = \frac{3 \cdot 2^d - (d+2)}{pq}$$

Because $2^d = 2^{d+1}/2 = (n+1)/2$ and $1/q = n - 1$, we have obtained $z_d = \Theta(n^2)$, so $\mathcal{T} = \Omega(n^2)$ (as discussed, z_d is, this time, a lower bound for $\mathcal{T}$). The bound from Theorem 3 gives $\mathcal{T} = \mathcal{O}(dn^2|E|) = \mathcal{O}(n^4)$, while the bound from Theorem 2 is not even polynomial.

4.2 Asynchronous Model

To analyze the speed of convergence of our protocols, we introduced *schedulers*, which decide how the current configuration changes in one step, and we counted the expected number of steps until all nodes lead to the same output, i.e. reach *stable consensus*. This model leads to

what we might call the *sequential expected convergence time*. However, an alternative model mentioned in the literature works in the following way: Each edge of the interaction graph has an associated *clock* that ticks at the times of a rate 1 Poisson process (defined in Chapter 2). The clocks of distinct edges are assumed to be independent. When a clock ticks, we use a scheduler to decide how the configuration changes. This is called the *Asynchronous Model*. An interesting question is: given a protocol $\mathcal{P}$, a scheduler M and an interaction graph G, what is the expected time until we reach a stable consensus, under the asynchronous model? Fortunately, this can be easily reduced to what we did for the previous model:

Theorem 4. *Assume G has $m := |E| \geq 1$ edges. Let Z_k be random variables describing the time at which the k-th tick happens, $k \geq 1$. Then $E(Z_k) = \frac{k}{m}$.*

Corollary. *Assume $\mathcal{P}$ expectedly needs $t_{\mathcal{P}}$ steps to compute a predicate ψ in the sequential model. Then the asynchronous model expectedly needs $t_{\mathcal{P}}/m$ time. We call this the parallel expected convergence time.*

To prove Theorem 4, we need a few preliminaries. The following properties are known from probability theory:

- In a Poisson process of rate λ, the interarrival times are exponentially distributed with rate λ.

- Let X be exponentially distributed. Then for all $t, s \geq 0$ it holds that:

$$\frac{P(X > t + s)}{P(X > t)} = P(X > s)$$

  We say that the exponential distribution is *memoryless*.

- Let T_i be exponentially distributed, independent random variables with rates λ_i, for $i \in [n]$, where $n \geq 1$. Then $\min\{T_i : i \in [n]\}$ is exponentially distributed with rate $\sum_{i=1}^{n} \lambda_i$.

Now the proof of Theorem 4. Assume that either $t = 0$ and no tick ever happened, or that at time $t \geq 0$ the clock of an edge ticked. W.l.o.g. assume for this case that this was the first clock. Let T_i ($i \in [m]$) be random variables denoting how much time passes, beginning at t, until the i-th clock ticks. Because the clocks tick at the times of a rate 1 Poisson process, their inter-tick times are exponentially distributed with rate 1. We directly get that $T_1 \sim exp(1)$.

Let $i \in [m]$, $i > 1$. If the i-th clock ever ticked, let $t_i \leq t$ be the time at which this last happened, else let $t_i := 0$. Let S_i be the random variable denoting how much time passes, beginning at t_i, until the next tick of the i-th clock. As before, we know that $S_i \sim exp(1)$. Moreover, using the fact that the exponential distribution is memoryless, we get for any $d \geq 0$: $P(T_i > d) = P(S_i > t - t_i + d \mid S_i > t - t_i) = P(S_i > d)$. For $d < 0$, obviously $P(T_i > d) = P(S_i > d) = 1$. We deduce that $T_i \sim S_i \sim exp(1)$.

Let T be the time that passes, beginning at t, until the next tick happens. Then by definition $T := \min\{T_i : i \in [n]\}$. Because $T_i \sim exp(1)$ for all $i \in [m]$, we get that $T \sim exp(m)$ and, therefore, $E(T) = \frac{1}{m}$, which is independent of t. We have implicitly shown that $Z_j - Z_{j-1} \sim T \sim exp(m)$ and $E(Z_j - Z_{j-1}) = \frac{1}{m}$ for any $j \geq 1$, whereby we use the convention $Z_0 := 0$. Using the linearity of the expectation, we get $E(Z_k) = \sum_{j=1}^{k} E(Z_j - Z_{j-1}) = \sum_{j=1}^{k} \frac{1}{m} = \frac{k}{m}$.

□

4.3 Optimizing Schedulers

We have seen in the proofs of the Theorems 2 and 3 that the performance of the schedulers M_2 and M_3 is greatly influenced by their parameter $0 < p < 1$. In particular, M_2 can be very inefficient if $p \to 0$ or $p \to 1$, and M_3 can suffer from similar problems for $p \to 0$. We investigate, for each of these schedulers, what conditions we need, such that the existence of an optimal p is guaranteed. Maybe there exists an optimal p that works for any protocol and any initial configuration? Or maybe we can find an optimal p only after we fix a protocol? We will answer these questions. But, first, we will show:

Theorem 5. *Fix a scheduler M_i ($i \in \{2, 3\}$) and a protocol $\mathcal{P}$, such that (at least) for $0 < p < 1$, the expected time until reaching stable consensus is finite. Then, for a fixed initial configuration C,*

there indeed exists an optimal $p \in [0,1]$, i.e. a value of p which minimizes the expected time until the computation reaches a stable consensus.

The careful reader might notice that the optimal p is allowed to be 0 or 1, which contradicts the definition of schedulers, where we assumed $0 < p < 1$. The definition uses this constraint, because it is needed for the proof of Theorem 1 (correctness of schedulers M_i). This is in general necessary for Theorem 1 to hold: e.g. for M_3 and $p = 0$, one can imagine a protocol where a special agent must meet all the others for the computation to end in consensus; if the interaction graph is not complete, and because no swaps ever happen, the computation will never finish; however, if the interaction graph is complete, the computation will definitely succeed. It is not hard to see that this contradicts Theorem 1. But there are other particular cases, where $p = 0$ or $p = 1$ is well defined, and sometimes it is even the optimal value. (We will confirm this by examples later.)

Proof sketch of Theorem 5. By assumption, it is clear that the expected number of steps until a stable consensus is reached, denoted $\mathcal{T}(p)$, is a well defined function from $(0,1)$ to $\mathbb{R}_{\geq 0}$. W.l.o.g. assume that C is not already a stable consensus, else any p is optimal. We justify that $\mathcal{T}$ is a *rational function* in p (see Chapter 2). Consider a Markov chain which imitates the transition graph, i.e. the set of states Q is the set of possible configurations, a directed edge between configurations C_1 and C_2 exists iff $C_1 \overset{M_i}{\to} C_2$, and the probability of this edge is equal to $P(C_1 \to C_2)$. (See definition of Transition Markov Chains.) Let A be the set of states corresponding to stable consensuses. Then, $\mathcal{T}(p)$ is the expected hitting time from C to A. Our schedulers M_i have the nice property, that if $C_1 \overset{M_i}{\to} C_2$, then $P(C_1 \to C_2) = a + bp$, where $a, b \in \mathbb{Q}$. Therefore, writing all the hitting time equations from C' to A for the configurations C' in $Q \setminus A$, results in a linear system with $|Q| - |A| > 0$ equations, $|Q| - |A|$ unknowns, and coefficients which are first-degree (rational) polynomials in p. This system has a unique solution, because expected hitting times are well defined. $\mathcal{T}(p)$ is a component of the solution of this linear system; it therefore must be a rational function in p. Because $\mathcal{T} \geq 0$, this implies the existence of a minimum for some $p \in [0,1]$ (this has been discussed in Chapter 2). (The existence of a maximum does not follow, but we are luckily not interested in this).

□

What if we just fix a protocol, but allow the initial configuration to vary? We will show that this weakening of the assumptions of Theorem 5 does, in general, no longer suffice to

assert the existence of an optimal p. We first give an example for M_2. Consider the protocol with states $\{x, 0, 1\}$ and non-trivial transitions $x0 \to x1$, $0x \to 1x$. We consider all states to be initial states. We output states x and 1 to 1, and state 0 to 0. We use the topology given by a line graph with three nodes v_i, $i \in [3]$. A configuration C is denoted by the tuple $(C(v_1), C(v_2), C(v_3))$. In the following, we will use a simplified version of the Markov chain described in the proof of Theorem 5, where a state of the Markov chain is a set of configurations. The state 0 includes the configurations corresponding to stable consensuses. The states of the other nodes are encoded by strings $s_1s_2s_3 \in \Sigma^3$, where $\Sigma := \{x, 0, 1\}$, and the string $s_1s_2s_3$ includes both the configurations (s_1, s_2, s_3) and (s_3, s_2, s_1) (symmetry). The states $S := \{0, x01, x10, 0x1, x00, 0x0\}$ describe all possible configurations. For two sets of configurations $X, Y \in S$, we define $P(X \to Y) := P(x \to y)$, where $x \in X$, $y \in Y$. This is well defined: it is easy to check that for all configurations $x_1, x_2 \in X$, $y_1, y_2 \in Y$, we have $P(x_1 \to y_1) = P(x_2 \to y_2)$.

The states $\{0, x01, x10, 0x1\}$ form a subchain, which will prove to be enough for our purposes:

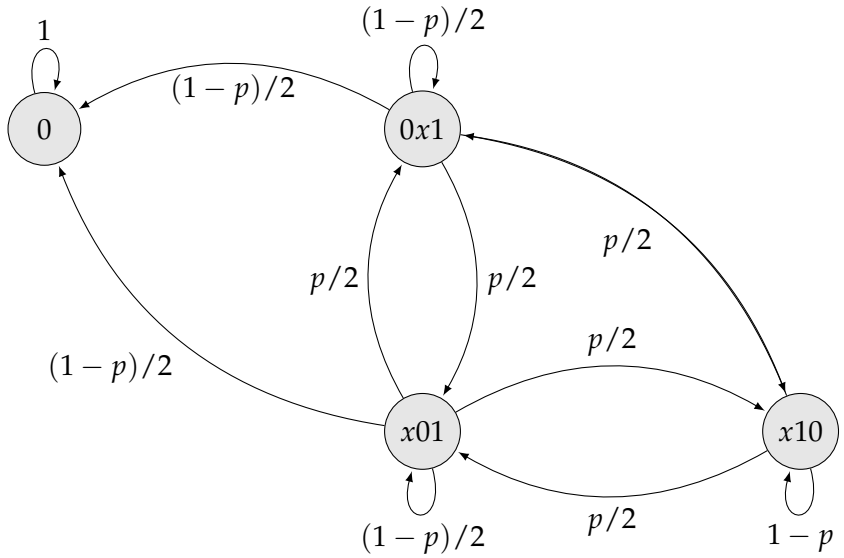


Figure 4.6: Chain for Example for M_2

Solving for the expected hitting times to 0 (using the well known z-notations) gives $z_1 := z_{0x1} = z_{x01} = \frac{3}{1-p}$ and $z_2 := z_{x10} = \frac{2p+1}{p(1-p)}$. This implies, among other things, that the initial

configurations $C_1 := (x, 0, 1)$ and $C_2 := (x, 1, 0)$ satisfy the hypothesis of Theorem 5 (i.e. the expected hitting times towards stable consensus are finite for any $0 < p < 1$). For C_1, the optimum is reached when minimizing z_1, so when taking $p = 0$. For C_2 this is definitely not the case, as z_2 goes to ∞ for $p \to 0$ (it can be shown that the minimum is obtained for $p = (\sqrt{3} - 1)/2$, so the optimal p is not always 0 or 1). We have shown that for M_2, different initial configurations (satisfying the hypothesis of Theorem 5) can lead to different optimal p. So no general optimal p exists.

What happens if we use M_3? Unfortunately, the previous example no longer works: it can be shown that the expected hitting times for all initial configurations are optimized by taking $p = 1$. However, we can still construct an example. We keep the topology identical (as well as the description of the states of our Markov chain), but include two new transitions for the protocol: $1x \to 0x$, $x1 \to x0$. This time, the Markov chain no longer has any subchains we can use, so we draw it in its entirety. The Markov chain is given in Figure 4.7.

Solving the equations for the hitting times (for example, by using WolframAlpha), we get:

$$z_1 := z_{x01} = \frac{18p^2 - 13p - 10}{4p^2 - 2p - 3},\ z_2 := z_{x10} = \frac{18p^3 - 3p^2 - 19p - 4}{p(4p^2 - 2p - 3)}$$

One could analytically (or numerically) try to find the minima of these functions, but we present the plots in Figure 4.8, which are already enlightening. Both z_1 and z_2 are well defined for all $0 < p < 1$ (the initial configurations C_1 and C_2 therefore fulfill the hypothesis of Theorem 5), but z_1 is optimal for $p = 1$, whereas z_2 is optimal for some value $0 < p < 1$ (again showing that there doesn't exist a single optimal p, but also that the optimal p for an initial configuration need not be 0 or 1).

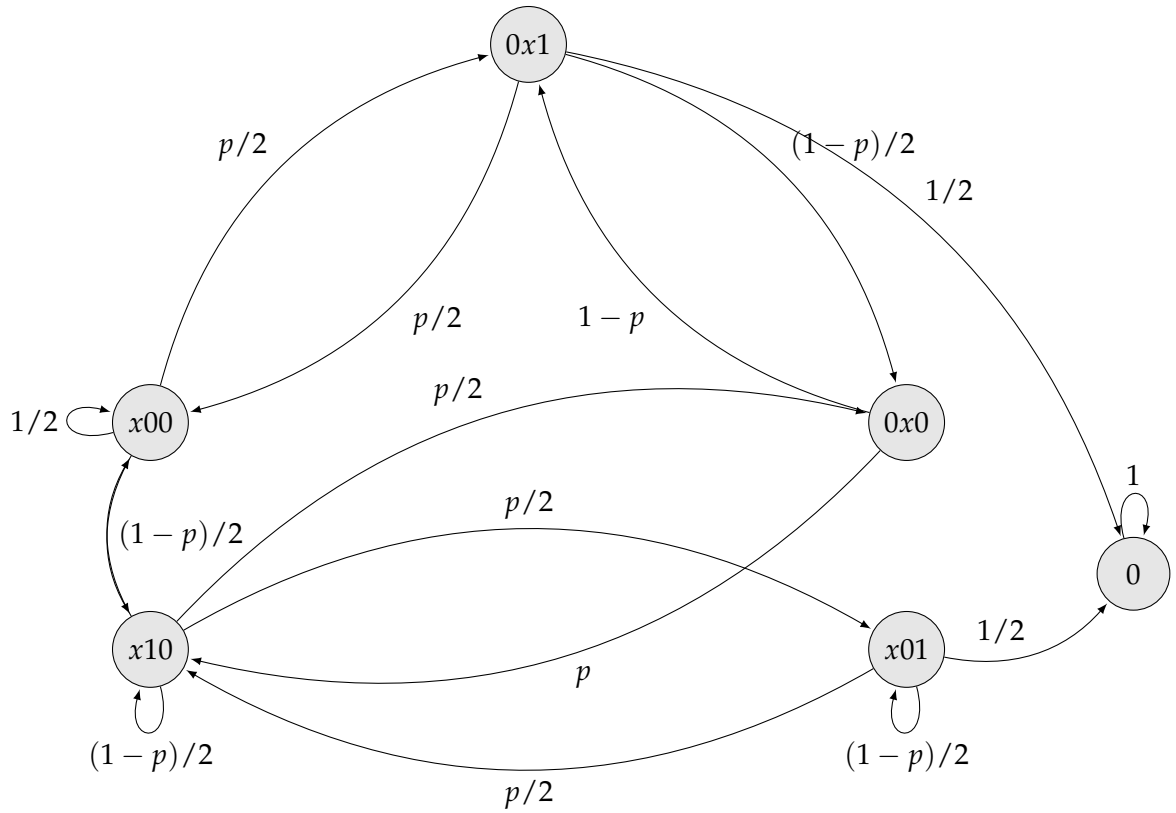


Figure 4.7: Chain for Example for M_3

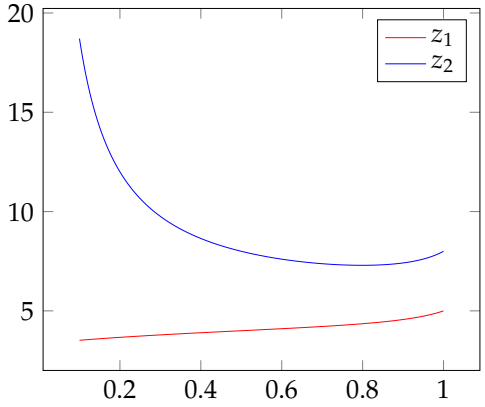


Figure 4.8: Comparing z_1 and z_2

5 Peregrine

Peregrine can be (currently) accessed from `https://peregrine.model.in.tum.de/`. It is a tool developed at the Chair for Foundations of Software Reliability of the Technical University of Munich [10, 4], and works in the browser. Peregrine is open source: the code can be (currently) found at `https://gitlab.lrz.de/ga97cer/peregrine`. This repository includes our implementation, which can be found on the branch *peregrine-graphs*. The tool allows users to design their own protocols, to simulate them, and to gather statistics about certain properties, like convergence speed. The tool can also verify the correctness of protocols automatically: if the protocol correctly computes the given predicate, Peregrine confirms this; if not, it provides a counterexample. It uses the SMT solver *Z*3 [15] to test satisfiability of Presburger formulas, and the model checker LoLa [16] to exhibit counterexamples. For more details on how Peregrine works and what it can do, we point to [10] or to the documentation, which is included on the website. In this chapter, our aim is to discuss (only) the new features we implemented. Indeed, there are some features Peregrine lacks: it only allows simulations, generation of statistics, and verification for protocols that run on complete graphs under the scheduler M_1 (see definition of schedulers).

5.1 Objectives

The main objective of our implementation is to extend Peregrine to work on arbitrary (connected) interaction graphs. More precisely, the implementation should offer following key functionality:

Creating Graphs. The user can construct a specific graph, either by manually drawing it on a canvas, or by using a graph generator which spawns special, predefined graphs (like the "Star Graphs" or "Line Graphs" from Section 4.1.3). For drawing graphs, the user is able to move

nodes around, to add/delete nodes/edges and to choose an initial configuration, manually or randomly.

Visualizing Simulations. The user can visualize runs of a protocol (starting in the chosen initial configuration) under specific schedulers on the chosen graph. Whenever two agents interact, the edge connecting their positions is highlighted, and their new states appear on the screen.

Storing Graphs. Similarly to how Peregrine already supports creating/saving/deleting protocols, the user is able to persistently save a specific graph on the server for later use. Stored graphs can be identified by their name; they can be edited or deleted.

Statistics. Generating statistics can be done for protocols that run on a specific graphs under a specific scheduler.

Like most web applications, Peregrine consists of a backend and frontend. The frontend is implemented in JavaScript (abbreviated JS), using the framework React. JS is a multi-paradigm, dynamically typed, object-oriented language, which is often used for frontends of web applications [17]. React is a (currently popular) JS library for building user interfaces on the web. It uses *components*, which describe what should be rendered on the screen. A component can use other components to specify this, thus allowing for modular design. For more details about React, we point to the official website, which (at the time of writing) is `https://reactjs.org/`. The backend of Peregrine written in Haskell, which is a statically typed, purely functional programming language, that infers types and uses lazy evaluation [18]. For our implementation, we have mostly worked in the frontend, but backend programming was required to implement persistent storage for graphs.

Note that we have not implemented a verification tool for protocols running on arbitrary graphs. This is because, thanks to Theorem 1, correctness for runs on complete graphs (under M_1) is equivalent to correctness for runs on arbitrary graphs (under M_2/M_3). Moreover, verifying correctness on complete graphs is possible for all protocols, as long as we know what predicates they compute. An exception to this rule is given by some special protocols, that are not only designed to run on arbitrary graphs, but which also try to determine information

about the graph itself (e.g. whether the interaction graph is a cycle). More information can be found in [9]. Formulating a Presburger predicate for these protocols seems to be impossible, and we do not know how we could verify their correctness. But, for our purposes, the already implemented verification tool is powerful enough.

5.2 Implementation

For graph drawing, we use the React library *react-d3-graph* by Daniel Caldas, which is based on the powerful JS library *d3*. This library suits our purposes, because it is easy to use and is specialized for graphs. It offers an API that can be used to control what events should be triggered, depending on where/how the user clicks on the screen. For more details regarding the implementation of the library, we point to the GitHub repository: `https://github.com/danielcaldas/react-d3-graph`. Using this library, we have implemented some of the features we discussed in the previous section. More precisely:

- The library sets up a canvas where graphs can be drawn, or where the users can visualize the graphs they generate/load.
- Adding a node is done by left-clicking on the canvas background; the node appears on a random position on the screen. To add an edge, the user must select a node first. Selecting a node is done by clicking on it. If this is done correctly, the node gets bigger, to signify that it has been selected. Then, after clicking on the node to which the user wants to connect the selected node, the edge appears. Removing a node/edge is done by right-clicking on it. Removing a node also removes its incident edges.

These features are accessible via the *Graph Simulation tab*. There exist other tabs as well; each of them includes a tool that can be used for the selected protocol. For example, the tab *Details* gives a short description of the protocol, the tab *Simulation* allows simulations on complete graphs under M_1, and the tab *Statistics* contains the statistics generator.

5.2.1 Graph Simulation

The *Graph Simulation tab* allows the user to set up the initial states for each node. For this, the user first selects the state from a drop-down list, and then he double-clicks on a node to assign it the selected state. Figure 5.1 shows an example of a drawn graph in the *Graph Simulation tab*, for a protocol whose initial states are called Y (blue) and N (red). This is a protocol that solves the majority problem, which we will discuss more extensively in Section 5.3. Concretely, the protocol is designed to output 1 (*true*) iff the number of red nodes is not greater than the number of red nodes. Above the drawn graph, there exist two drop-down menus, called *Mode* and *Swap*. Using the terminology of the thesis (definition of schedulers), these are the scheduler and the parameter p respectively. We implemented two schedulers, called *Manual* (as in the figure) and *Automatic*. The *Manual* mode is just the scheduler $M_3(p)$, and *Swap* is the parameter p. The computation stops when no interaction between two agents (recall definition of agents) could change the output of the current configuration. Concretely (and equivalently), the algorithm checks, at every step, whether any non-trivial transition is enabled under clique-topology. If this is no longer the case, the computation stops and the user is informed whether a consensus has been reached.

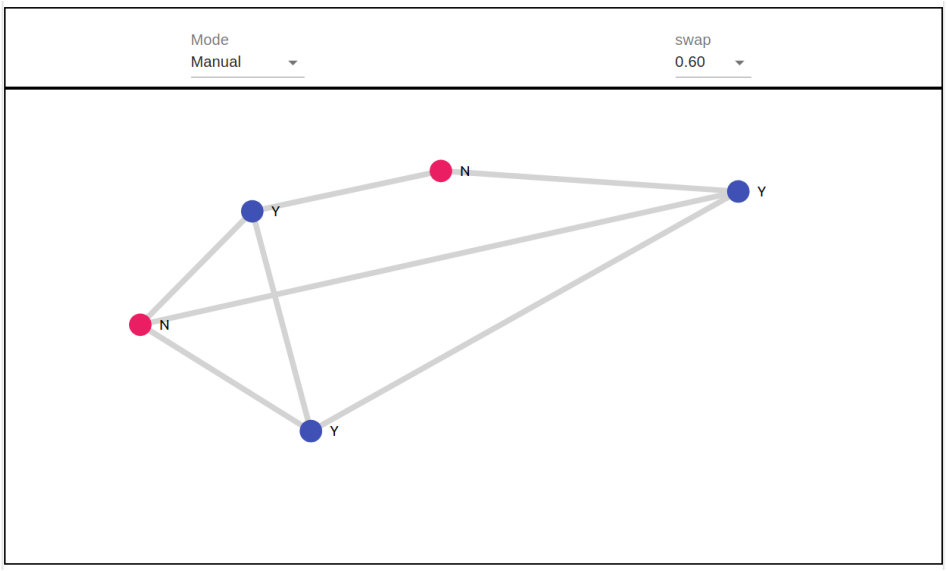


Figure 5.1: Example of drawn graph in *Graph Simulation tab*

The *Automatic* mode has been included to make computations somewhat faster than plain M_3; at every step, the algorithm tries to randomly pick an edge between two nodes u and v, such that a non-silent transition can be applied. If it succeeds after n times, where n is the size of the population, it applies the transition, and also swaps the states of u and v, with probability $1/2$ (like $M_3(1/2)$). If such an edge cannot be found, two arbitrary connected nodes are chosen and their states are swapped. The algorithm stops under the same conditions as the manual mode. The automatic mode is somewhat non-local, because it triggers additional swaps when it sees that they lead to progress, which the population normally would not be able to know. We have not implemented the scheduler M_2, because it is empirically slower than M_3, yet exploits the same idea of swapping states.

After constructing the graph, setting up the initial states, and choosing the scheduler, the user can click on the *Play* button. If the graph is not connected (contradicting the definition we have given for interaction graphs), the simulation will not start and the user will be warned. Else, the simulation starts: At every step, the chosen edge is highlighted and the states of the interacting nodes are updated. It is also possible to *Pause* the simulation, or to change the speed by using a slider. For higher speeds, more steps are applied at once. To check out the execution step by step, the user can pause, and then use the buttons *Previous* and *Next*. When the computation ends, the user is informed whether consensus has been reached. Figure 5.2 shows this behavior; the control buttons and the speed slider can also be seen. Because there were more blue nodes than red nodes initially (see Figure 5.1), the stable consensus outputs 1, and all nodes are colored blue.

We have also included the ability to generate special interaction graphs. There are multiple parameterized families of graphs the implementation supports. We give a description:

- *Circular Graphs with parameter n*: $G := (V, E)$, where $V := [n]$ and $E := \{\{i, (i \bmod n) + 1\}, i \in [n]\}$, i.e. a cycle of length $n \geq 1$.
- *Grid Graphs with parameters m, n*: $G := (V, E)$, where $V := [m] \times [n]$ and $E := \{\{(i, j), (k, l)\} : |i - k| + |j - l| = 1\}$, i.e. graphs that look like a $m \times n$ grid.
- *Star Graphs with parameter n*: see section 4.1.3.
- *Erdős Graphs with parameters n, p*: random graphs with n nodes. They are generated by

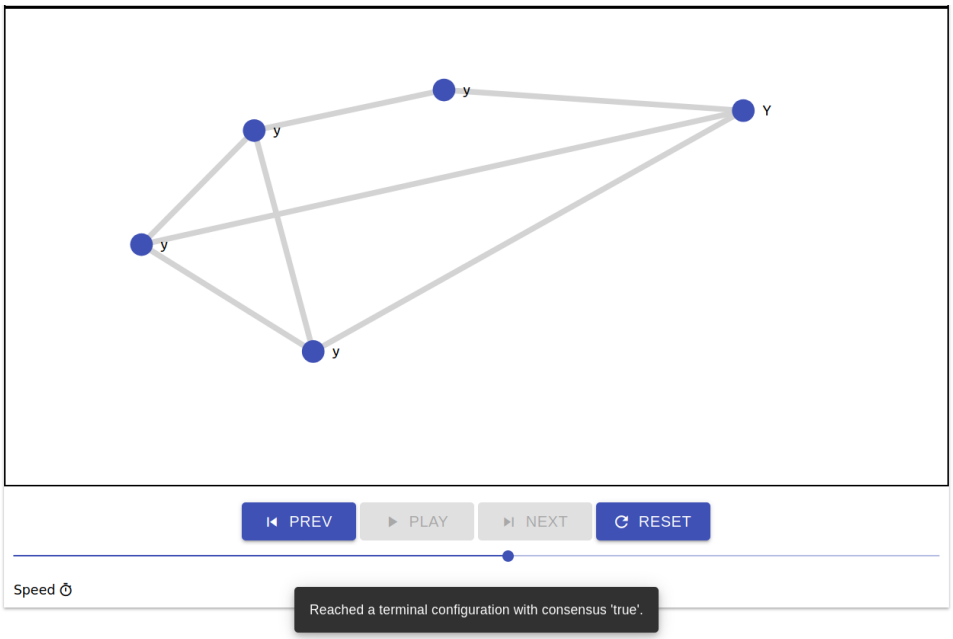


Figure 5.2: Example of consensus in *Graph Simulation tab*

the following algorithm: an edge is constructed between i and j with probability p, for all $i, j \in V$. The generator tries to generate such a graph which is also connected. If, after a 100.000 tries, the generator fails to find such a graph, it will not draw anything and report the failure in a message explaining that p appears to be too small. Indeed, the smaller the probability of constructing edges, the smaller the chance of the resulting Erdős graph being connected.

- *Preferential Attachment Graphs with parameters* $n, k_0 \geq k$: random graphs with n nodes. They are generated by the following algorithm: The construction starts with k_0 nodes forming a circular graph. Then, the other $n - k_0$ nodes are constructed iteratively. Every such new node x is connected to k already existing nodes: the chance of a node y becoming a neighbor of x is hereby directly proportional to the number of neighbors y already has in the existing graph. Preferential Attachment Graphs have the advantage over Erdős Graphs that they are certainly connected.

Algorithm 1 (written in pseudocode) describes how each of the $n - k_0$ nodes are connected to the existing nodes for the Preferential Attachment Graphs.

Algorithm 1: Constructing Preferential Attachment Graphs

Assume that $i \geq k_0$ nodes have already been included. We add the node $i+1$.
$d :=$ a vector where every of the i nodes appears $deg(i)$ times.
for $ct \in [k]$ **do**
  uniformly at random select an index r of d
  connect $i+1$ to $d[r]$
  $d :=$ eliminate all appearances of $d[r]$ from d
end

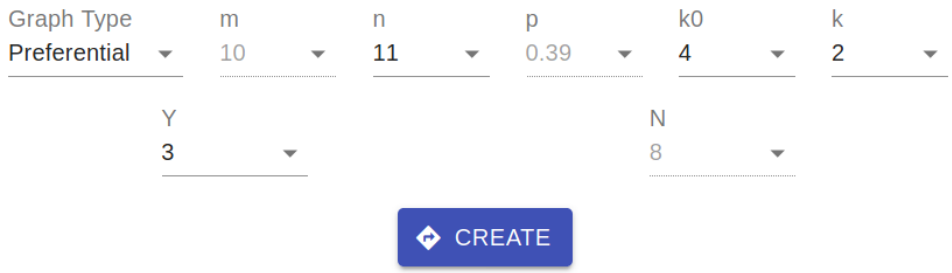


Figure 5.3: Generating graphs

Figure 5.3 shows the controls that can be used for the generation. The user selects the *Graph Type*, the suitable parameters, and then clicks on *Create* to spawn the graph. There is also the possibility of specifying the distribution of the initial states (see the fields Y and N in the figure). In general, for any protocol $\mathcal{P}$ with initial states $Q_0 := \{q_1, \ldots, q_l\}$ ($l \geq 1$), the user can specify how many of the n nodes should have the initial state q_i, for all $i \in [l]$. Assume that the user selects the values $n_1, \ldots, n_l$ ($\sum_{i=1}^{l} n_i = n$). Let I be the set of the initial configurations of size n, that assign the state q_i to precisely n_i nodes, for all $i \in [l]$. The implementation uses Algorithm 2 to assign the initial states to the nodes. The algorithm guarantees that the initial configuration is selected uniformly from the set I.

Algorithm 2: Assigning initial states

Assume n nodes, initial states q_i, inputs n_i ($i \in [l]$).
$d :=$ the set of n nodes.
for $i \in [l]$ **do**
  for $j \in [n_i]$ **do**
    uniformly at random select an index r of d
    set the state of node $d[r]$ to q_i
    $d :=$ eliminate $d[r]$ from d
  end
end

It is also possible to load a graph from the list of saved graphs. For this, there is a drop-down list where the user can select a graph by its name, and the graph will spawn after clicking on *Load*. Details about storing graphs follow in the next section. The implementation of most features described in this section can be found in the frontend files *GraphSimulation.js*, which mostly deals with the visual parts, *GraphUtils.js*, which implements the algorithms dealing with graph generation, and *RandomRunner.js*, which executes the simulation. (See GitLab repository.)

5.2.2 Graph Editor

The *Graph Editor* is a tool for constructing graphs that, unlike the ones in the *Graph Simulation tab*, are not tied to a specific protocol. It allows drawing/generating/loading graphs as before, but doing simulations or assigning states to nodes is no longer possible (because we are no longer tied to a protocol). However, the editor adds functionality for saving and deleting graphs. It can be accessed from the main menu of Peregrine. Figure 5.4 shows the specific components. Above the 10×10 drawn grid, we can see the drop-down list *Graph to Load*, which contains the names of the saved graphs. After selecting a graph by its name, the user can load it or delete it. Under the canvas, the field *Graph Name* asks for a name, and after clicking on *Save* the currently drawn graph will be persistently saved on the server under that name. The user can later load it in the *Graph Editor* or in the *Graph Simulation tab* of a

protocol.

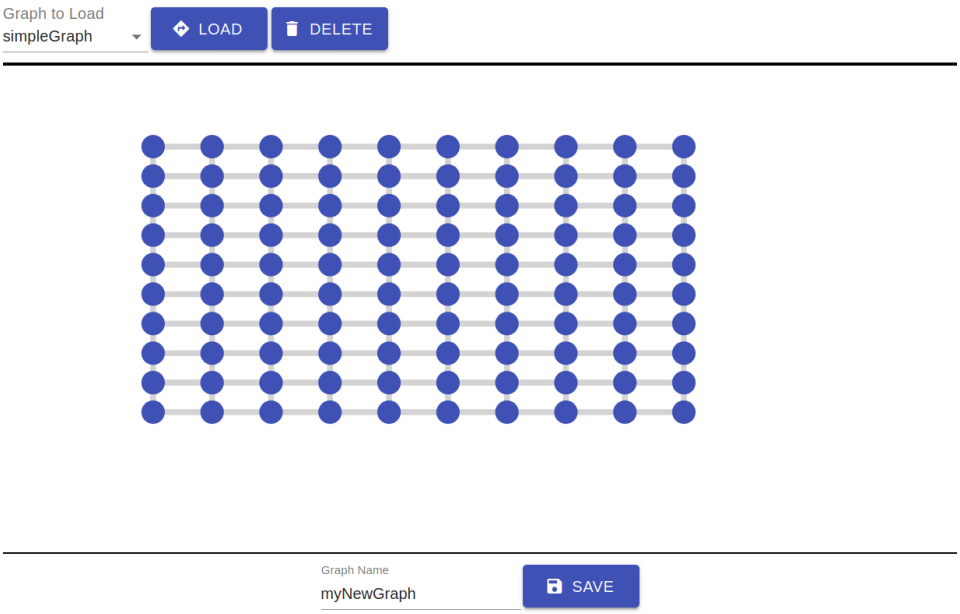


Figure 5.4: The *Graph Editor*

The implementation is done both in the frontend and the backend. When the user clicks the corresponding buttons, the frontend makes requests to the server, whose API has been extended to support operations such as: retrieving the list of currently saved graphs, adding a graph into the list, deleting a graph from the list, and returning the list of node coordinates & edges for a graph. Most of the relevant backend (Haskell) code can be found in the files *Actions.hs* and *Lib.hs* (see GitLab repository).

5.2.3 Statistics

We have implemented a tool for generating statistics about sets of simulations on arbitrary interaction graphs. Such a tool was already included in Peregrine, but only for complete graphs. Using this new functionality is possible, after selecting a protocol, in the *Graph Statistics tab*. The tab includes a canvas, where graphs can be generated or loaded using the known menus (but not drawn). The user selects the scheduler he wants to use (*Manual*

or *Automatic*), the *Number of steps*, which we denote with k, and the number of simulations, which we denote with N. Clicking on the button *Generate* activates Algorithm 3.

Algorithm 3: Generating Statistics

Assume n nodes, initial states q_i, inputs k and N.
$d :=$ the set of n nodes.
for $i \in [N]$ **do**
  // Generating initial configuration
  for $v \in d$ **do**
    choose initial state q uniformly at random
    assign state q to node v
  end
  execute $\leq k$ steps of simulation (stop if consensus is reached)
end

For every of the N simulations, the algorithm remembers the initial configuration, the last reached configuration, and the number of steps which were executed. We call this the *raw data*. The output consists of:

- Discussion of consensus: the program gives the numbers of simulations that led to no consensus, that led to correct consensus (i.e. correctly calculating the predicate the protocol is supposed to compute), and that lead to incorrect consensus.
- Average number of steps: the program gives the average number of steps for the simulations that reached consensus. This can be filtered to include only those computations that reached correct/incorrect consensus.
- The raw data.

5.3 Case Study: AVC

We will now analyze the behavior of a concrete protocol on specific (interaction) graphs. A very important problem in distributed computing is the *majority problem*. We give an informal

description: The n agents of the population are assumed to be initially either blue or red. Their purpose is to determine whether there are more blue agents than red agents. If this is the case, then the computation should eventually stabilize to a positive consensus, otherwise to a negative consensus. Much literature has been devoted to finding and analyzing protocols that solve this problem. For example, Vojnović et al. find upper bounds for the expected convergence time of a specific protocol that solves the majority problem [6]. Their bounds are very general, they depend on the chosen interaction graph. There have been efforts both in trying to find very simple protocols that solve majority, and in trying to optimize performance. Most simple protocols are inefficient, and most performant protocols are not simple. This raises the question whether this trade-off can be eliminated. Are there protocols that are both simple and fast, and which solve majority?

In [2], Vojnović et al. present the *Average and Conquer* (AVC) protocol as a positive answer to this question. The protocol has two parameters: an odd number $m > 0$ and an integer $d > 0$. There are three kinds of states: the strong states $\{-m, -m+2, \ldots, -3, 3, 5, \ldots, m-2, m\}$, the intermediate states $\{-1_1, \ldots, -1_d, 1_d, \ldots, 1_1\}$, and the weak states $\{-0, +0\}$. Intuitively, the *sign* of the state ($+$ or $-$) indicates the "opinion" of the node, i.e. whether it believes that the majority is blue or red. The *value* of the state (its absolute value) suggests how sure it is of its opinion. At the beginning, blue nodes have state $+m$, while red nodes have state $-m$. The idea of the protocol is hidden in the title: when two nodes meet, their values are *averaged*. The strong states can "convince" other nodes of changing their sign (opinion). The weak states $+0$ and -0 are called like that because they do not influence the sign (opinion) of their partner. The intermediate states are a technical way of preventing the nodes from becoming weak too quickly. For a complete and accurate description of the protocol, we point to [2]. The authors also prove upper and lower bounds for AVC, which they confirm empirically. However, these bounds only hold for complete interaction graphs. In this section, we will analyze the AVC protocol experimentally on specific interaction graphs and showcase its performance.

Methodology. We already described Peregrine and our implementation, which allows simulating protocols on arbitrary topologies. We will use this implementation (with some convenient modifications) to gather statistics about the AVC protocol, which is already in-

cluded in Peregrine. We use the scheduler M_3, because it appears to be somewhat faster than M_2. We will do simulations for certain graph families, i.e. sets of interaction graphs of the form $\{G_n : n \in \mathbb{N}\}$. The AVC protocol does not handle ties, so for simplicity we will settle on odd values of n. For every chosen graph G_n, we do 10 simulations, using various initial configurations, and average the results. We could do more simulations per graph, but we observed that this does not significantly alter the result. There are many possibilities to choose initial configurations; for example, one could choose randomly (with probability $1/2$) whether a node is blue or red. However, this is not a very good idea; for bigger n, the number of blue nodes will tend to be very close to the number of red nodes, which will result in a much worse runtime than what we can generally expect. This is because, as most majority protocols, AVC's performance is worst when the initial configuration is close to a tie. On the other hand, including initial configurations where there is a very solid majority is not interesting, because most protocols (and especially AVC) tend to be deceivingly fast when this is the case. We have decided on using configurations that linearly vary from "almost tie" to "majority of 75%". Algorithm 4 describes our method.

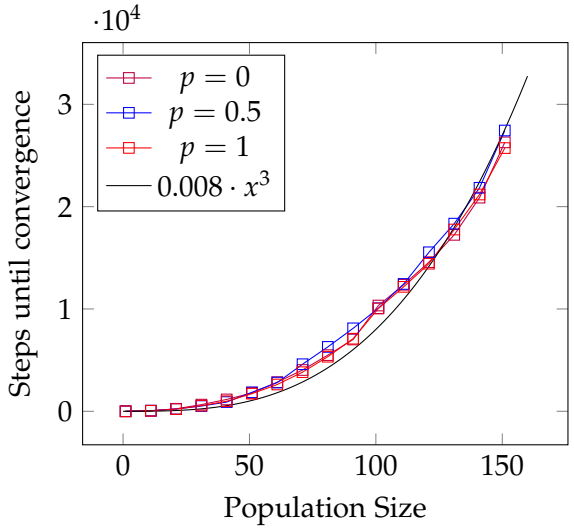
Algorithm 4: Running a set of 10 simulations

$a := \frac{n/2-1}{9}$
$b := 1 - a$
for $i \in [10]$ **do**
  $f_i := ai + b$
  $y := \lfloor (f_i + n)/2 \rfloor$
  generate Population with y blue nodes (and $n - y$ red nodes) on the graph G_n
  use Peregrine to execute the simulation using M_3
end

Generating a population with y blue nodes is done in a random fashion, as described in the implementation of Peregrine (see Section 5.2.1). Observe that for $i = 1$ we get $y = \lfloor (n+1)/2 \rfloor$, whereas for $i = 10$ we get $y = \lfloor 3n/4 \rfloor$, and the jump from the first value to the last value is done in a linear way. Therefore, this corresponds to what we intended. We execute the simulation until we reach a consensus (no limit of steps). Because AVC works correctly on complete interaction graphs, and because we use M_3, Theorem 1 guarantees that every

reached stable consensus is also correct. We do all simulations using the two values $p = 0.5$ and $p = 1$ of the swap parameter p. For the start, we assume the parameters of the protocol $m = 3$ and $d = 1$. For the values of n, we take $n_j := 11 + 10j$, where $0 \leq j \leq 14$. Let $n_j(p)$ be the number of steps until convergence we obtain for $n = n_i$ if we use the parameter p.

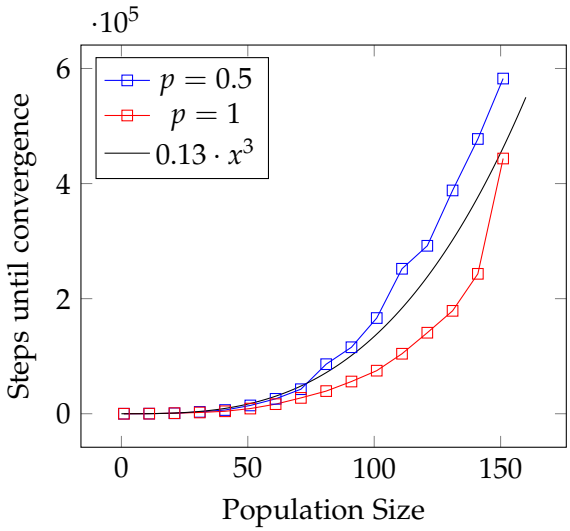
We will also try to guess the asymptotic speed for each topology. To do that, we first select a natural number $b > 0$, and then use the least squares method to find the polynomial function of type mx^b ($m > 0$) which best fits the data. More precisely, we take the column vector A, where $A_i := n_i^b$, the column vector y, where y_i is the average of $n_j(0.5)$, $n_j(1)$, and calculate the least squares solution to the equation $Am = b$. (For this, we have used Matlab). The solution is a scalar m_0, which leads to the polynomial $m_0 x^b$. We then compare the optimal polynomials we obtained for different values of b. In all cases, we noticed that there is one b which stands out (i.e. it leads to the best approximation), and each time we assumed that the complexity is therefore most likely $\mathcal{O}(n^b)$. The plots that follow contain this best fitting polynomial to visually convince the reader of our guess. Of course, there are other methods which could be used. For example, one could try fitting a general polynomial of degree b for different values of b, or using more refined techniques, such as spline interpolation. However, for our purposes, this method seems to be good enough.



n	Steps(p = 0.5)	Steps(p = 1)
11	66	52
31	511	549
51	1856	1722
71	4599	3823
91	8107	7095
111	12443	12162
121	15540	14423
131	18339	17765
141	21842	21187
151	27443	25725

Figure 5.5: Complete Graphs

We begin with Complete Graphs. Figure 5.5 shows the convergence behavior. The value of the swap parameter p does not seem to affect performance. We include the result for $p = 0$ (in the plot), because M_3 with $p = 0$ is equivalent to the scheduler M_1, and it shows that, for complete graphs, the schedulers M_1 and M_3 lead to equal performance. In this case, the data is well approximated by a third degree polynomial, which suggests that the complexity is cubic. This is not surprising: in the mentioned paper [2], the authors prove a cubic bound for complete graphs.

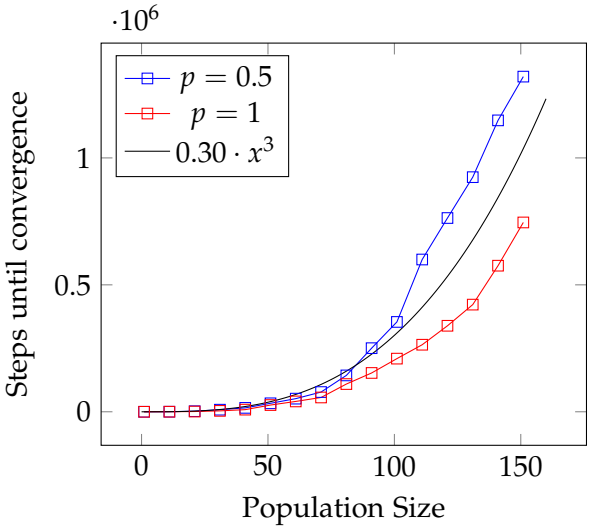


n	Steps(p = 0.5)	Steps(p = 1)
11	156	120
31	3206	2195
51	14390	8987
71	42604	27511
91	115550	55988
111	251927	104249
121	291870	140792
131	388068	178828
141	477525	243149
151	582499	443546

Figure 5.6: Circular Graphs

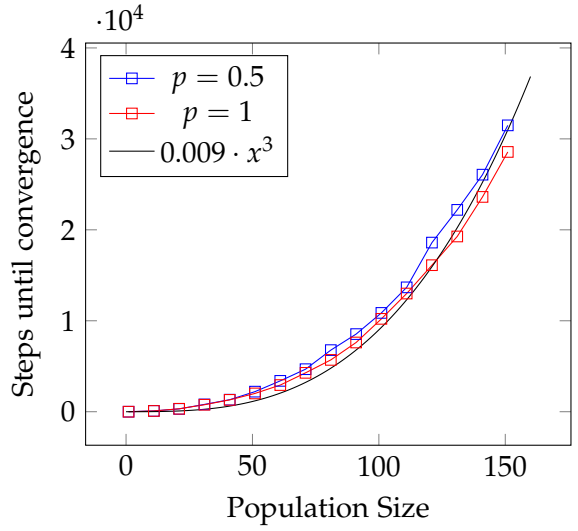
Figure 5.6 shows the convergence behavior for *Circular Graphs* (as described in Section 5.2.1). There is no significant difference between the two swap values, and the convergence speed seems to be cubic.

Figure 5.7 shows a very similar cubic trend for *Line Graphs*, but the values are definitely greater than those we had for Circular Graphs. This was to be expected, because Line Graphs have a diameter twice as big, and indeed the convergence seems to be twice as slow.



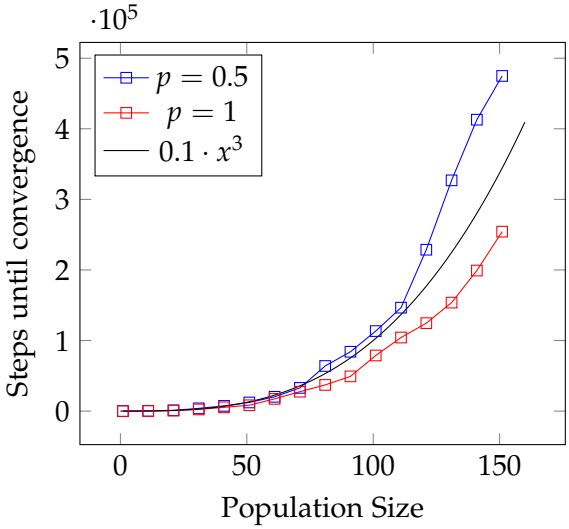
n	Steps(p = 0.5)	Steps(p = 1)
11	278	244
31	8238	3597
51	33412	26561
71	77725	56381
91	250557	152902
111	599800	264114
121	763222	338764
131	924162	422118
141	1147876	575441
151	1320221	745768

Figure 5.7: Line Graphs



n	Steps(p = 0.5)	Steps(p = 1)
11	89	84
31	804	766
51	2207	2010
71	4976	4274
91	8535	7611
111	13674	12992
121	18595	16112
131	22194	19270
141	26064	23618
151	31487	28563

Figure 5.8: Star Graphs



n	Steps(p = 0.5)	Steps(p = 1)
11	292	170
31	3913	2355
51	12241	8296
71	32882	27514
91	84103	49209
111	146542	104265
121	228681	124630
131	326961	153726
141	412911	199141
151	474902	254288

Figure 5.9: Preferential Attachment(2,1)

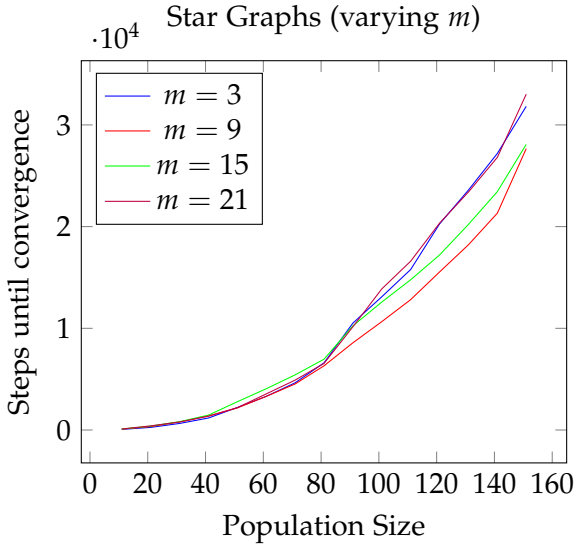


Figure 5.10: Varying m

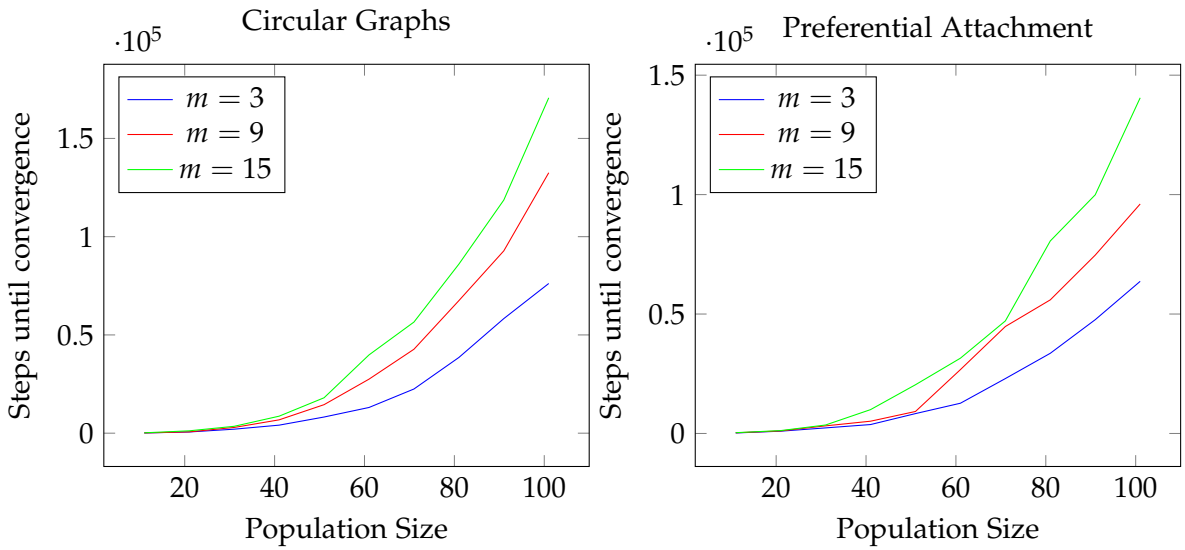


Figure 5.11: Varying m (continued)

In Figure 5.8 we present the results for *Star Graphs*. Here, there is almost no difference between the two swap values. Again, it seems that the complexity is cubic. The given values can also be well fitted by a quadratic polynomial, but it does not extrapolate as good to higher values.

Lastly, in Figure 5.9 we look at the convergence behavior for *Preferential Attachment* graphs, which were presented in Section 5.2.1. For this example, we selected the parameters $k_0 = 2$ and $k = 1$, which lead to quite sparse graphs, and to what seems to be cubic speed. The difference between the two swap values is quite big here; the convergence for $p = 1$ appears to be twice as fast. Changing k_0 does not appear to modify the speed. Increasing k leads to significantly faster convergence times, especially for small n, but the trend always remains cubic. Surprisingly, we obtained cubic speed for all considered topologies, but some of them were more suitable than others: it appears that complete graphs and star graphs are the best option, and that the bound found by Voinović et al. in [2] remains valid even when the topology is changed.

Recall that all simulations above hold for $m = 3$ and $d = 1$. In the following, we will see what happens when we change the parameter m. We do the simulations in the same way as

before, one time for every value of m. We have seen that changing p is not very significant, so we use what appears to be the best value, i.e. $p = 1$. The result for Star Graphs is given in Figure 5.10. Interestingly, there seems to be no clear effect when changing m. For Circular Graphs and Preferential Attachment Graphs, the situation is a bit different (see Figure 5.11). It becomes apparent that m has a constant influence on the performance. Similar results are obtained for Complete Graphs and Line Graphs. We could not find any examples where m appears to have more than a constant influence.

6 Conclusion and Future Work

We have proven (Chapter 3) that correct protocols on complete graphs still work on arbitrary, connected graphs, provided that the scheduler satisfies certain natural properties. For this to work, we introduced the schedulers M_2 and M_3, which use state swapping (agent repositioning) with probability p. This idea could be extended: instead of using the predefined constant p, we could design modified protocols that allow the population to self-regulate the parameter p for optimum performance. p would no longer be a characteristic of the scheduler, but would become part of the protocol. A built-in heuristic like making an agent swap positions more often, if its state remains constant for a certain number of interactions, could work. It would be exciting to see whether such modified protocols can lead to drastically different performance.

In Chapter 4, we provided two upper bounds on the expected convergence speed of canonical protocols running on graphs, by first noticing that the performance is greatly influenced by the expected time until two fixed agents interact, and then by bounding this value using two methods. The most powerful bound we found is $\mathcal{O}(d|V|^3|E|)$ (Theorem 3), where d is the diameter of the interaction graph $G := (V, E)$. Our bounds are sometimes tight (e.g. the bound from Theorem 2 for Star Graphs), but other examples show that this is not always the case (e.g. none of the two bounds is tight for Line Graphs). Further thoughts about improving these bounds, or about finding lower bounds, could help better understand what the true performance of our schedulers/protocols is. A possible starting point could be trying to use *stage graphs* (see [14]) instead of stage vectors to refine the upper bounds. It is also likely that a more careful analysis of the Markov chains from Theorem 3 improves the bound.

We discussed the existence of an optimal parameter p of our schedulers (Section 4.3). We have seen that, when fixing an initial configuration, an optimal p exists, but as soon as we allow the initial configuration to vary, we can no longer expect this. It is not clear, in general, how to find the best p for a given initial configuration, nor how to select a p if we do not

know the initial configuration in advance. Another open question is whether M_3 is expectedly *always* faster than M_2 under the same p, if they both start in the same initial configuration. Our examples suggest that this is the case, but we could not find a proof. However, further work in this direction would be rather theoretical, because empirical evidence suggests that M_3 is almost always significantly faster. Examples where running a protocol on a specific graph is much faster than running it on a complete graph would also be worth looking into.

In Chapter 5 we presented Peregrine, and gave a tour of the new features we implemented. Peregrine can be effectively used to gather data about runs of specific protocols on specific graphs, and it also helps building an intuition about the behavior of the protocols. There are many ways our implementation could be improved. Most of the simulations are done in the frontend. A more clever implementation in a language like C++ could make it possible to run simulations on very big graphs (> 10.000 nodes), whereas our implementation sometimes struggles with a few hundredths nodes. Including more graph choices, more interesting statistics, schedulers and protocols is of course also a good idea. We empirically analyzed the AVC protocol (Section 5.3) and our results suggest cubic performance, no matter what structure the interaction graph has. A logical next step would be to theoretically analyze the performance of this protocol on specific interaction graphs, using the results obtained in [2] as a starting point.

In summary, the thesis has shown that running protocols on arbitrary interaction graphs is doable and leads to new research questions. Despite the fact that we could partially address the concerns regarding correctness and performance, we feel that this topic is still mostly populated by open and captivating questions.

List of Figures

Bibliography

[1] D. Angluin, J. Aspnes, Z. Diamadi, M. Fischer, and R. Peralta. *Computation in networks of passively mobile finite-state sensors*. In: *Distributed Computing* (2006), pp. 235–253.

[2] R. D. Alistarh and M. Vojnović. *Fast and exact majority in population protocols*. In: Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, 2015, pp. 47–56.

[3] D. Angluin, J. Aspnes, and D. Eisenstat. *Stably computable predicates are semilinear*. In: *ACM Symposium on Principles of Distributed Computing (PODC)* (2006), pp. 292–299.

[4] M. Blondin, J. Esparza, S. Jaax, and A. Kučera. *Black Ninjas in the Dark: Formal Analysis of Population Protocols*. In: LICS '18: Proceedings of the 33rd IEEE Symposium of Computer Science, 2018.

[5] J. Esparza, P. Ganty, J. Leroux, and R. Majumdar. *Verification of population protocols*. In: *Acta Informatica 54, 2* (2017), pp. 191–215.

[6] M. Draief and M. Vojnović. *Convergence Speed of Binary Interval Consensus*. In: *SIAM J. Control Optim., 50(3)* (2012), 1087–1109 (23 pages).

[7] M. Blondin, S. Jaax, J. Esparza, and P. Meyer. *Towards Efficient Verification of Population Protocols*. In: (Mar. 2017).

[8] I. Chatzigiannakis, O. Michail, and P. Spirakis. *Algorithmic Verification of Population Protocols*. In: *Stabilization, Safety, and Security of Distributed Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 221–235.

[9] D. Angluin, J. Aspnes, M. Chan, M. Fischer, H. Jiang, and R. Peralta. *Stably Computable Properties of Network Graphs*. In: *Distributed Computing in Sensor Systems*. Springer Berlin Heidelberg, 2005, pp. 63–74.

[10] M. Blondin, J. Esparza, and S. Jaax. *Peregrine: A Tool for the Analysis of Population Protocols*. In: Proc. 30th International Conference on Computer Aided Verification (CAV), July 2018.

[11] P. Offtermatt. *A Tool for Verification and Simulation of Population Protocols*. Bachelor's Thesis. Technical University of Munich, 2017.

[12] C. Baier, J.-P. Katoen, and K. G. Larsen. *Principles of Model Checking*. MIT Press, 2008.

[13] R. G. Gallager. *Stochastic Processes: Theory for Applications*. Cambridge University Press, 2014.

[14] M. Blondin, J. Esparza, and A. Kučera. *Automatic Analysis of Expected Termination Time for Population Protocols*. In: CONCUR 2018, July 2018.

[15] L. de Moura and N. Bjørner. *Z3: An Efficient SMT Solver*. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2008, pp. 337–340.

[16] Schmidt and Karsten. *LoLA: A low level analyser*. In: vol. 1825. June 2000, pp. 465–474.

[17] D. Flanagan and P. Ferguson. *JavaScript: The Definitive Guide*. 3rd. USA: O'Reilly & Associates, Inc., 1998.

[18] A. J. T. Davie. *An Introduction to Functional Programming Systems Using Haskell*. USA: Cambridge University Press, 1992.