Stability and Structure-Formation Problems for Dynamic and Chemical Systems

Thesis submitted in accordance with the requirements of the University of Liverpool for the degree of Doctor in Philosophy by

Michail Theofilatos

June 2022

Contents

Abstract **vii**

Acknowledgements **ix**

1 Introduction **1**
1.1 Population Protocols . . . 2
1.1.1 Motivation . . . 2
1.1.2 Formal definition . . . 3
1.1.3 Examples . . . 4
1.2 Network Constructors . . . 6
1.2.1 Motivation . . . 6
1.2.2 Formal definition . . . 6
1.2.3 Self-stabilizing protocols . . . 7
1.2.4 Examples . . . 8
1.3 Crystal Structure Prediction . . . 9
1.3.1 Algorithmic approaches . . . 9
1.4 Mobile Agents on Dynamic Graphs . . . 10
1.4.1 Motivation and related work . . . 11
1.4.2 A formal model for dynamic networks . . . 12
1.5 Thesis Contribution . . . 12
1.5.1 Counting and leader election in population protocols . . . 12
1.5.2 Fault tolerant network constructors . . . 13
1.5.3 Crystal structure prediction via oblivious local search . . . 14
1.5.4 Gathering in dynamic graphs . . . 14
1.6 Author's Publications Presented in this Thesis . . . 15

I Population Protocols **17**

2 Approximate Counting in Population Protocols **19**
2.1 Introduction . . . 19
2.1.1 Related work . . . 20
2.1.2 Preliminaries . . . 21

Approximate counting problem . . . 22
2.1.3 Contribution and roadmap for the chapter . . . 22
2.2 Fast Approximate Counting with a Unique Leader . . . 22
2.2.1 Abstract description and protocol . . . 22
2.2.2 Analysis . . . 23
2.3 Experiments . . . 27

3 Leader Election in Population Protocols **29**
3.1 Introduction . . . 29
3.1.1 Related work . . . 30
3.1.2 Preliminaries . . . 31
Leader election problem . . . 31
3.1.3 Contribution and roadmap for the chapter . . . 31
3.2 Leader Election with Approximate Knowledge of n . . . 32
3.2.1 Abstract description . . . 32
3.2.2 The protocol . . . 33
3.2.3 Analysis . . . 33
3.2.4 Dropping the assumption of knowing $\log n$ . . . 39
3.3 Experiments . . . 41
3.4 Open Problems . . . 42

II Structure Formation Problems **45**

4 Fault Tolerant Network Constructors **47**
4.1 Introduction . . . 48
4.1.1 Related work . . . 48
4.1.2 Contribution and roadmap for the chapter . . . 50
4.2 Model and Definitions . . . 51
4.2.1 Network constructors with crash failures . . . 51
4.2.2 Definition of terms . . . 53
4.3 Network Constructors without Fault Notifications . . . 54
4.3.1 Unbounded number of faults . . . 55
4.3.2 Bounded number of faults . . . 57
4.4 Notified Network Constructors . . . 63
4.4.1 Fault-tolerant protocols . . . 63
4.4.2 Universal fault-tolerant constructors . . . 65
4.4.3 Designing fault-tolerant protocols without waste . . . 77
4.5 Conclusions and Further Research . . . 83

5 Crystal Structure Prediction via Oblivious Local Search **85**
5.1 Introduction . . . 85
5.1.1 Contribution and roadmap for the chapter . . . 87
5.2 Preliminaries . . . 89
Composition . . . 89

Unit cell parameters. 90
Arrangement 90
5.2.1 Energy 91
Relaxation 92
5.2.2 Crystal structure prediction problems 92
5.3 Local Search 94
5.4 Algorithms 95
5.5 Experiments 96
5.6 Conclusions 104

III Mobile Agents on Dynamic Graphs 105

6 Gathering in 1-Interval Connected Graphs 107
6.1 Introduction 107
6.1.1 Previous work 107
6.1.2 Contribution and roadmap for the chapter 109
6.2 Model and Definitions 110
6.2.1 Dynamic network model 111
6.2.2 Definition of terms and the problem 112
6.3 Impossibility Results 113
6.3.1 Symmetric initial configurations in unicyclic graphs 115
Branch classes 115
Symmetric configurations 116
Agent position symmetries 116
6.3.2 Additional limitations on the solvability of *weak gathering* . . 119
6.4 An Algorithm for *Weak Gathering* in Dynamic Unicyclic Graphs . . . 121
6.4.1 *Weak gathering* algorithm 121
6.4.2 Analysis 130
First phase of the algorithm 131
Second phase of the algorithm 133
6.5 Open Problems 136

7 Conclusions 139

Bibliography 143

Abstract

In distributed systems and algorithms, a stable property is a global property which once achieved, continues to hold forever. Such a property might refer to a system's behavior under representative perturbations to their operating environments, where the ability to recover from such perturbations is termed in the literature as self-stabilization. In particular, self-stabilization is a concept of distributed algorithms and protocols that characterizes their ability to automatically recover from an arbitrary configuration and converge within finite time to a configuration that satisfies a given specification. For instance, in the population protocol (PP) model, we say that a configuration is stable when the system gets to a configuration that is correct and no subsequent sequence of transitions can take the PP to an incorrect configuration, while self-stabilization refers to the system's ability to start in any global configuration and always stabilize to a configuration that meets the task specification. Such systems can tolerate worst-case transient faults. Similarly, termination in distributed systems is a stronger form of stability which is persistent, meaning that once such a state has been reached, it will never change. However, termination cannot always be achieved.

In this thesis we examine some well-known problems that require stable solutions, and we provide new algorithms for them that achieve a correct stable state. To demonstrate their correctness and efficiency, we formally analyze them and for some of them we also provide experimental results. The thesis is broken down into three parts; in the first part we consider systems of interacting entities for the stable computation of functions, in the second part we work on structure formation problems where the goal is to construct stable structures that satisfy a given set of criteria, and in the third part we consider more general dynamic networks where mobile agents are able to move on a dynamically changing graph

In particular, Chapter 2 deals with the problem of approximate counting, while Chapter 3 with the problem of electing a leader, considering for both the Population Protocol model. For approximate counting, we provide a protocol for the stable computation of an approximation of the population size, and for leader election

a protocol that can trade off time and space by adjusting a parameter which is embedded into the algorithm.

Chapter 4 deals with the stable construction of graphs in the Network Constructors model where crash failures on the nodes may occur during the execution of the protocol and can result to an incorrect configuration of states in the population. We provide self-stabilizing protocols for several basic network constructions and universal constructors that can construct a large class of graphs. In Chapter 5 we work on optimization algorithms for the crystal structure prediction (CSP) problem, where the goal is to find the most stable configuration of atoms by minimizing the free energy function. We formalize the problem from a theoretical computer science perspective, we propose algorithms for it, and we provide experimental results and comparisons with existing algorithms for CSP.

Finally, Chapter 6 deals with the problem of gathering a set of mobile agents on some node of the dynamic graph they operate in. We first provide some impossibility results, and in light of them, we focus on a relaxed version of the problem where the agents have to stabilize to a configuration such that all agents remain in distance at most one from each other, and eventually terminate. This problem is termed in the literature as *weak gathering*. We provide a deterministic algorithm for unicyclic graphs that runs in $O(n^2 + nk)$ number of rounds, where n is the size of the graph, and k the number of mobile agents.

Acknowledgements

Undertaking this PhD has been a truly life-changing experience for me and it would not have been possible to do without the support and guidance that I received from many people throughout the past years.

First and foremost I am extremely grateful to my supervisors Prof. Paul Spirakis and Dr. Othon Michail for their invaluable insights, support, and patience. Their insightful feedback pushed me to sharpen my thinking and improve my academic skills. Paul and Othon, thank you for your guidance and everything you taught me, not only regarding research, but also in a personal level.

Many thanks also to my PhD advisors Prof. Leszek Gąsieniec and Prof. Igor Potapov for assessing my progress and helping me improve my presentation and research skills. I am also grateful to the members of my thesis committee, Prof. Leszek Gąsieniec and Prof. Tomasz Radzik, for their feedback during the viva examination and for their constructive comments on the thesis. A huge thanks goes to the people that I had the privilege to work with: Paul Spirakis, Othon Michail, Argyrios Deligkas, Dmytro Antypov, Matthew Rosseinsky, David Doty, George Skretas, George Mertzios, Vladimir Gusev, and Mahsa Eftekhari. Furthermore I would like to thank the members of the Networks and Distributed Computing Group for inspiring me, and the Leverhulme Research Centre for providing financial support to undertake this PhD.

A very special thank you goes to all my friends, without whom my life in Liverpool would not have been the same. I was very fortunate to meet most of them right from the beginning of my journey in Liverpool, which made my transition in the UK way smoother than what I expected. Thank you Manos, George, Argy, Katerina, Ari, Elektra, Eleni, Themi, Nico, Eleni, Fonta, Matina, Alkmini, Dimitri, Katerina, Peter, and Yanni. I would also like to thank my friends in Greece, and most importantly, my family: Nikolas, Mairi, and Angelos for encouraging me to follow my dreams.

And finally, I would like to say a heartfelt thank you to Katerina, who has been by my side throughout this PhD, living every single minute of it, and without whom, I would not have had the courage to embark on this difficult journey in the first place. Thank you for your support and your patience.

List of Figures

1.1 An example of three execution steps of the Spanning line protocol (Protocol 2). Node-states in red represent state updates and lines in red represent new edge activations. 8

2.1 Convergence time of *Approximate Counting (APC)* with a unique leader protocol. Both axes are logarithmic. The dots represent the results of individual experiments and the line represents the average values for each network size. 27
2.2 *Approximate Counting (APC)* with a unique leader. Estimations and actual sizes of the population. 28
2.3 *Approximate Counting (APC)* with a unique leader. Counters c_q and c_a when half of the population has been infected by the epidemic. 28

3.1 *Leader Election* with approximate knowledge of n. Both axes are logarithmic. The dots represent the results of individual experiments and the line represents the average values for each network size. 41
3.2 Convergence time of the composition of *Leader Election* and *Approximate Counting* protocols. 42
3.3 Composition of *Leader Election* and *Approximate Counting* protocols. Upper bounds and actual sizes of $\log n$. 42

4.1 In 4.1a, D defines a ring of size k. In 4.1b, each node of D corresponds to a set of nodes (or supernode), while for each edge of D between two nodes u_i and u_j, all nodes of V_i are connected to all nodes of V_j and vice versa. 59
4.2 An illustration of the fault notification mechanism. In the first example (Figures 4.2a and 4.2b), the gray node crashed, and the nodes that were adjacent to it at step i were notified and updated their state. At step $j > i + 1$, this node along with its adjacent edges are not present. In the second example (Figures 4.2c and 4.2d), the crashed node was isolated, thus an arbitrary node was notified and updated its state. 64
4.3 The left line is the result of one connection between two isolated nodes, and one expansion (rules 1 and 2 of Protocol 9). The second line is the result of a line merging (rule 3 of Protocol 9). 68
4.4 In Figure 4.4a, the walking state and the left endpoints are in states w_2 and e_2. Then, the walking state eventually reaches the second endpoint which is in state e_1, resulting to state l_0 (Figure 4.4c). 68

4.5 In Figure 4.5a there is one walking state, and one endpoint in state l_1. l_1 state is the result of a fault on an adjacent node of it. This state introduces an additional walking state, and in Figure 4.5c the two walking states interact and only one survives. The unique walking state w is then guaranteed to first traverse the whole line at least once before an endpoint becomes l_0. 69
4.6 The population is partitioned into two groups U and D that form a perfect matching. The nodes of U eventually form a spanning line, and simulate a TM of linear space. The TM repeatedly constructs random graphs in the nodes of D, until the graph belongs to the given graph language. 72
4.7 The population is partitioned into three groups M, U, and D of equal size. The nodes of U form a perfect matching with both U and M. The nodes of U eventually form a spanning line, and simulate a TM of linear space that uses the edges in M as an $O(n^2)$ binary memory. The TM repeatedly constructs random graphs in the nodes of D, until the graph belongs to the given graph language. 75

5.1 Most stable configuration of $SrTiO_3$ 89
5.2 Time to reach specific energy levels for $SrTiO_3$ (15 atoms). Figures (a) and (b) correspond to the algorithm of Section 5.4. Figures (c) and (d) correspond to basin hopping. The median times needed to reach every energy level are depicted in red on the top of each plot. 100
5.3 Time to reach specific energy levels for $SrTiO_3$ (20 atoms). Figures (a) and (b) correspond to the algorithm of Section 5.4. Figures (c) and (d) correspond to basin hopping. The median times needed to reach every energy level are depicted in red on the top of each plot. 102
5.4 Performance of the Axes algorithm for $Y_2Ti_2O_7$. The black points correspond to the energy found after the relaxation of a point computed by the Axes neighborhood at each step. The red line is the lower envelope of the energy found by our algorithm while the blue line corresponds to the lower envelope of basin hopping. 103
5.5 Performance of the basin hopping algorithm for $Y_2Ti_2O_7$. The black points correspond to the energy found after the relaxation of a random feasible configuration of atoms at each step. The red line is the lower envelope of the energy found by basin hopping, while the blue line corresponds to the lower envelope of our algorithm. 103

6.1 Example of a graph G with a unique cycle C. Each B_{w_i} is a tree (or branch) of G rooted at $w_i \in C$. 113

6.2 Example of a graph $G \in \mathcal{F}$. Dashed lines do not belong to the spanning bicyclic subgraph. e_1 and e_2 are called *blocking* edges. 114
6.3 Examples of symmetric configurations with periodic branches and port labels on the cycle. In the top left figure all unique branches have the same periodic appearance (i.e., $P_i = \{3\}, \forall i \in \{0,1,2\}$). In the top right and bottom figures, the branches have different periodic appearances (i.e., in the top right figure the sets of periods are $P_0 = \{4\}, P_1 = \{2\}$, and $P_2 = \{4\}$, while in the bottom figure the sets of periods are $P_0 = \{2,4\}$ and $P_1 = \{4\}$). 117
6.4 Indistinguishable unicyclic graphs 120
6.5 Locally constructed graph $\mathcal{G}$ by the end of the exploration process . . . 132

List of Tables

1.1 List of author's publications presented in this thesis 15

4.1 Summary of our results. 52

5.1 Comparison between depth approach and GULP for $SrTiO_3$. Energy is in electronvolts (eV). The energy difference shows the average difference in energy between the depth approach and the energy calculated by GULP (absolute value). Results averaged over 2000 random feasible structures. 96

5.2 Comparison of local neighbourhoods for reaching a combinatorial minimum for $SrTiO_3$ with 15 atoms per unit cell and $\delta = 1$Å (375 grid points). Time is in seconds and energy in electronvolts (eV). Results averaged over 1000 arrangements. 97

5.3 Evaluation of relaxation procedure after using a combinatorial neighborhood for $SrTiO_3$ with 15 atoms per unit cell. Time is in seconds and energy in electronvolts (eV). Results averaged over 1000 arragnements. . 97

5.4 Comparison of local neighbourhoods for reaching a combinatorial minimum for $Y_2Ti_2O_7$ and $\delta = 1$Å (343 grid points). Time is in seconds and energy in electronvolts (eV). Results averaged over 1000 arrangements. . 98

5.5 Evaluation of relaxation procedure after using a combinatorial neighborhood for $Y_2Ti_2O_7$. Time is in seconds and energy in electronvolts (eV). Results averaged over 1000 arragnements. 98

5.6 Statistics from the experiments depicted in Figures 5.2 and 5.3 ($SrTiO_3$ with 15 and 20 atoms per unit cell). 98

5.7 The values of Buckingham potential parameters we used in our experiments as they were found in [36]. All the missing parameters are set to zero. 102

6.1 Summary of our results for the *weak gathering* problem. Assumptions include the existence of *homebases*, knowledge of the graph size n, and knowledge of the number of agents k 111

Chapter 1

Introduction

Stability in algorithms is a notion that has been defined in several ways, and captures certain algorithm-specific and problem-specific properties. In some cases, the goal of an algorithm is to stably compute a function, or reach a correct, stable configuration. In numerical algorithms for differential equations, numerical stability is a desirable property which is concerned with the growth of round-off errors and/or small fluctuations in initial data which might cause a large deviation of the final answer from the exact solution. The stability of a learning algorithm pertains to how changes to the training data influence the result of the algorithm.

In this thesis, we have selected four areas where stability achievement is critical and in each of them we examine a fundamental problem and provide new algorithmic solutions. These areas are (a) systems of interacting entities for the stable computation of functions, (b) actively and distributedly constructing a network spanning a set of cooperating entities, (c) computation of stable atomic structures in materials, and (d) dynamic networks in which a set of mobile agents move and cooperate.

In particular, we first consider the Population Protocol model [7] and we study the problems of approximate counting and leader election. Our approximate counting protocol stabilizes in $\Theta(\log n)$ parallel time, where n is the size of the population, and provides a constant factor approximation of $\log n$. Parallel time is defined as the total number of interactions, divided by n. Our leader election protocol terminates in $O(\frac{\log^2 n}{\log m})$ parallel time, using $O(max\{\log m, \log\log n\})$ bits. By adjusting the parameter m between 1 and n we obtain a leader election protocol whose time and space can be smoothly traded off between $O(\log^2 n)$ to $O(\log n)$ time and $O(\log\log n)$ to $O(\log n)$ bits. We then work on structure formation problems, and we first consider the Network Constructors model [113]. Network Constructors resemble Population Protocols where the agents can additionally activate edges between them. We address the issue of dynamic formation of graphs under faults, and we answer the question of what graph languages can be stably constructed and under what assumptions. We

then work on optimization algorithms for the crystal structure prediction problem, where the goal is to find the most stable arrangement of atoms by minimizing the free energy function. Finally, we consider dynamic graphs where a set of k agents are allowed to move between neighboring nodes. We study the weak gathering problem and we provide an algorithm where the agents terminate in $O(n^2+nk)$ rounds either in the same node, or in two neighboring nodes of the graph.

1.1 Population Protocols

Population protocols [7] is a model of distributed computation, represented as networks that consist of very weak *mobile* computational entities (also called *nodes* or *agents*), regarding their individual capabilities. The kind of their mobility is known as *passive mobility* in the literature, hence they are classified as *passively dynamic networks*. These networks consist of a large population of indistinguishable (i.e., anonymous) entities that interact with each other and execute an identical protocol, stored in the local memory of each agent. Computation in an agent occurs upon interacting with some other agent in the network. The input to a population protocol is distributed across the initial state of the entire population, and the goal is for the population to cooperatively compute a required function on that input. Two examples of simple protocols are given in Section 1.1.3

The interaction pattern between the agents is unpredictable and controlled by an adversary. In the generic case, there is an underlying interaction graph specifying the permissible interactions between the agents, typically representing distance constraints. The most common choice of the interaction graph is the complete graph. In a complete interaction graph, all agents with the same state are indistinguishable, and only the counts of agents in each state affect the outcome of the protocol. A strong global fairness condition is imposed on the adversary to ensure that the protocol makes progress. Finally, population protocols cannot detect when they have finished, therefore terminate; instead, the agents' outputs are required to converge in finite time to an output state and remain to that state indefinitely. A formal definition of this model is given in Section 1.1.2

1.1.1 Motivation

The population protocol model was designed to model Wireless Sensor Networks (WSN) and formally study their capabilities and limitations. Wireless Sensor Networks consist of low-cost and low-power sensor nodes, which are equipped with very limited computational capacity and memory. These nodes are able to communicate in short distances and collaborate as a group in order to complete a task. They obtain data from the environment that they are deployed at, they locally carry out

computations, and transmit only the required and partially processed data. Finally, the topology of the sensor networks changes arbitrarily and very frequently.

Sensor networks have a wide range of applications that spans from continues sensing of some environmental variable, to event detection and location sensing. They may consist of many different types of sensors such as seismic, thermal, visual, acoustic, and many more [134], thus allowing them to be used in many application areas, such as for health monitoring, space exploration, environment monitoring, etc.

1.1.2 Formal definition

A population protocol is formally specified by a tuple (X, Y, Q, I, O, δ), where

1. X is a finite input alphabet
2. Y is a finite output alphabet
3. Q is a finite set of possible states for an agent
4. $I : X \to Q$ is an *input map function* from X to Q
5. $O : Q \to Y$ is an *output map function* from Q to Y
6. $\delta : Q \times Q \to Q \times Q$ is the *transition function* that describes how pairs of agents interact and update their states

The computation takes place among a population of $n \geq 2$ agents, and proceeds in rounds. During each round, a single pair of agents is chosen from the scheduler to interact, and based on δ they update their states. Initially, each agent is given as input a value from X, and then I determines its initial state from Q. At any point, each agent's state $q \in Q$ determines its output $O(q)$ at that time.

Let two agents u and w be in states q_1 and q_2, respectively, and are chosen by the scheduler for interaction. If (q_1, q_2, q_1', q_2') is in the transition function δ, u and w update their states to q_1' and q_2', respectively. To describe δ, we list all possible interactions that update the state of u and/or w using the notation $(q_1, q_2) \to (q_1', q_2')$. For all transitions that are not specified in δ, we assume null transitions (e.g., $(q_1, q_2) \to (q_1, q_2)$). Additionally, a population protocol may be *symmetric* or *asymmetric*. In symmetric protocols, two agents in an interaction (and thus in the corresponding transition) are indistinguishable if their states are identical. Thus, their states are identical also after the transition. In asymmetric protocols, the interacting nodes can always be distinguished, that is, they are assumed to communicate in ordered pairs (u, w), where u is called the *initiator* and w the *responder*.

The multiset of initial agent states determines the initial *configuration* for an execution. In particular, a configuration of the system is a mapping $C : V \to Q$

specifying the state of each node in the population V. Alternatively, a configuration can be described by an unordered multiset of all agents' states (i.e., agents with the same state are indistinguishable).

We write $C \to C'$ if C' can be obtained from C by a single interaction of two agents. An *execution* of a protocol on input I is an infinite sequence of configurations $C_0, C_1, C_2, \ldots$, each of which is a set of states drawn from Q. C_i is the configuration at round i, C_0 is the initial configuration, and $C_i \to C_{i+1}$, $\forall i \geq 0$. In general, interactions can occur simultaneously, but when writing down an execution, we can order those simultaneous interactions arbitrarily.

The interaction between the agents is unpredictable; an adversarial scheduler chooses in each round a pair of nodes for interaction. To allow meaningful computations, we must impose some restrictions to the scheduler; otherwise, the scheduler could initially choose a single pair of nodes that will interact in every round, leaving the rest of the population in the same state indefinitely. The *fairness* condition that we impose to the scheduler is the following: If C is a configuration that appears infinitely often in an execution, and $C \to C'$, then C' must also appear infinitely often in the execution. In other words, any configuration that is reachable by a sequence of interactions is eventually reached.

Finally, an agent's output may change during the execution, however, correctness is a property that must be satisfied eventually. This means that we require that the agents produce the correct output at some time in the execution and continue to do so forever after that time.

Definition 1.1. We say that a population protocol Π computes a function f that maps multisets of elements of I to Y if for every such multiset $\mathcal{I}$ and every fair execution of Π that starts from the initial configuration corresponding to $\mathcal{I}$, the output value of every agent eventually stabilizes to $f(\mathcal{I})$.

1.1.3 Examples

As an introduction, we here provide two simple population protocols. The first one is a protocol that elects a unique leader in the population. The set of states is $Q = \{\ell, f\}$, where ℓ indicates that the agent is a leader, and f indicates a follower. Initially, all nodes start as potential leaders (in state ℓ), and the goal is that eventually only one node should remain in state ℓ, the rest of them being followers (in state f).

The transition function of the simple leader election protocol consists of the following transition

$$(\ell,\ \ell) \to (\ell,\ f)$$

Whenever two potential leaders interact with each other, one of them becomes a follower. This protocol will eventually stabilize to a configuration with only one node in state ℓ.

For the next example, consider a scenario where we need to monitor a wild animal population. This might be necessary for a variety of reasons, such as for disease detection. Monitoring can be achieved by attaching or implanting monitoring devices that can collect data with a predetermined frequency and can additionally communicate with a base station that collects these data for further analysis. Depending on the method of data collection, the impact on the welfare of animals may be greater or less, both for the individual being monitored, and the population as a whole. It is then imperative that such devices are as small as possible, hence with limited capabilities. A single, more powerful device in the population may be allowed to communicate with the base station in order to send the result, or output, of the protocol.

Consider a population of penguins where each individual has an implanted sensor device that can recognize a particular infectious disease. Let the set of states be $Q = \{d, n, u\}$, where d indicates that the animal is infected with the disease, n indicates that it is not infected, while u indicates that the state of the animal is unknown. This may happen because of insufficient or inconclusive data; not all devices can conclude that the animal has the infection at the same time. We are interested in identifying whether the majority of penguins are either infected, or not. A very simple to state *approximate majority* protocol is given in [8]; its transition function is shown in Protocol 1.

Protocol 1 Approximate majority

$Q = \{d,\ n,\ u\}$
$\delta :$

1: $(d,\ n) \rightarrow (d,\ u)$
2: $(n,\ d) \rightarrow (n,\ u)$

3: $(d,\ u) \rightarrow (d,\ d)$
4: $(n,\ u) \rightarrow (n,\ n)$

This protocol makes use of two competing epidemic processes; $d's$ and $n's$ try to propagate their state throughout the population. Epidemic processes in population protocols permit us to design much more efficient protocols with respect to their convergence time. As shown in [8], under a uniform random scheduler, the above protocol converges in $O(n \log n)$ interactions, and its output is the majority, provided that its initial margin is at least $\omega(\sqrt{n} \log n)$.

Our protocols in Chapters 2 and 3 make use of epidemic processes in order to achieve optimal and almost optimal time for the problems of *Approximate Counting* and *Leader Election*, respectively.

1.2 Network Constructors

Network Constructors is a model where a population of agents running *Population Protocols* can additionally activate/deactivate links when they meet. This model was introduced in [113], and is based on the *Population Protocol* (PP) model [7, 11] and the *Mediated Population Protocol* (MPP) model [107]. In particular, initially all agents are isolated and in the same initial state. When two agents interact, a protocol, which is stored in the local memory of each agent, takes as input the states of the agents and the state of the connection between them, and updates them. The main difference between PPs and Network Constructors is that in the PP (and the MPP) model, the focus is on computation of functions of some input values, while Network Constructors are mostly concerned with the stable formation of graphs that belong to some graph language.

In [113], Michail and Spirakis gave protocols for several basic network construction problems, and they proved several universality results by presenting generic protocols that are capable of simulating a Turing Machine and exploiting it in order to stably construct a large class of networks.

1.2.1 Motivation

Network Constructors (and its geometric variant [104]) is a theoretical model that may be viewed as a minimal model for programmable matter operating in a dynamic environment [112]. Programmable matter refers to any type of matter that can *algorithmically* transform its physical properties, for example shape and connectivity. The transformation is the result of executing an underlying program, which can be either a centralized algorithm or a distributed protocol stored in the material itself. There is a wide range of applications, spanning from distributed robotic systems [81], to smart materials, and many theoretical models (see, e.g., [48, 53, 56, 110] and references therein) try to capture some aspects of them.

1.2.2 Formal definition

A Network Constructor (NET) is a distributed protocol defined by a 4-tuple $(Q, q_0, Q_{out}, \delta)$, where Q is a finite set of node-states, $q_0 \in Q$ is the initial node-state, $Q_{out} \subseteq Q$ is the set of output node-states, and $\delta : Q \times Q \times \{0,1\} \to Q \times Q \times \{0,1\}$ is the transition function, where $\{0,1\}$ is the set of edge states.

In the generic case, there is an *underlying interaction graph* $G_U = (V_U, E_U)$ specifying the permissible interactions between the nodes, and on top of G_U, there is a dynamic overlay graph $G_O = (V_O, E_O)$, meaning that G_O is always a subgraph of G_U. A mapping function F maps every node in the overlay graph to a distinct underlay node. In most cases, G_U is a *complete undirected interaction graph*, i.e., $E_U = \{uv : u, v \in V_U \text{ and } u \neq v\}$, while the overlay graph consists of a population of n initially *isolated* nodes (also called *agents*).

The NET protocol is stored in each node of the overlay network, thus, each node $u \in G_O$ is defined by a state $q \in Q$. Additionally, each edge $e \in E_O$ is defined by a binary state (*active/connected* or *inactive/disconnected*). Initially, all nodes are in the same state q_0 and all edges are inactive. The goal is for the nodes, after interacting and activating/deactivating edges for a while, to end up with a desired stable overlay graph, which belongs to some graph language L.

During a (pairwise) interaction, the nodes are allowed to access the state of their joining edge and either activate it (state = 1) or deactivate it (state = 0). When the edge state between two nodes $u, v \in G_O$ is activated, we say that u and v are *connected*, or *adjacent* at that time t, and we write $u \underset{t}{\sim} v$.

1.2.3 Self-stabilizing protocols

Self-stabilization is a concept of distributed algorithms and protocols that characterizes their ability to automatically recover from an arbitrary configuration and converge within finite time to a configuration that satisfies a given specification. The arbitrary configuration is usually the result of a finite number of *transient* faults, however stronger forms of self-stabilization have been introduced to cope with permanent failures, such as *process crashes* (e.g., *fault-tolerant self-stabilization* [22, 51]) and *Byzantine* failures (e.g., *strict stabilization* [68, 122]). Self-stabilization is considered as a versatile fault tolerance approach since it recovers from such faults in a unified manner, while state-of-the-art self-stabilizing algorithms have usually small overhead with respect to time and space complexity.

Self-stabilization is an attractive technique for distributed systems equipped of processes with low computational and memory capabilities, therefore, their study in models such as Population Protocols and Network Constructors is very natural. In Chapter 4, we examine self-stabilizing protocols that can cope with crash failures, that is, nodes, along with their adjacent edges, are removed from the configuration and do not participate in the execution of the protocol again.

1.2.4 Examples

As an introduction to the Network Constructors model, we first consider the case where no crash failures occur, and we present a simple protocol for the spanning line problem, given in [113]. Let n be the number of the distributed processes, or nodes. The goal is for the output which is induced by the active edges between the nodes, to stabilize to a graph isomorphic to a line of size n.

Protocol 2 Spanning line

$Q = \{q_0,\ q_1,\ q_2,\ l,\ w\}$
$\delta:$

1: $(q_0,\ q_0,\ 0) \rightarrow (q_1,\ l,\ 1)$
2: $(l,\ q_0,\ 0) \rightarrow (q_2,\ l,\ 1)$
3: $(l,\ l,\ 0) \rightarrow (q_2,\ w,\ 1)$
4: $(w,\ q_2,\ 1) \rightarrow (q_2,\ w,\ 1)$
5: $(w,\ q_1,\ 1) \rightarrow (q_2,\ l,\ 1)$

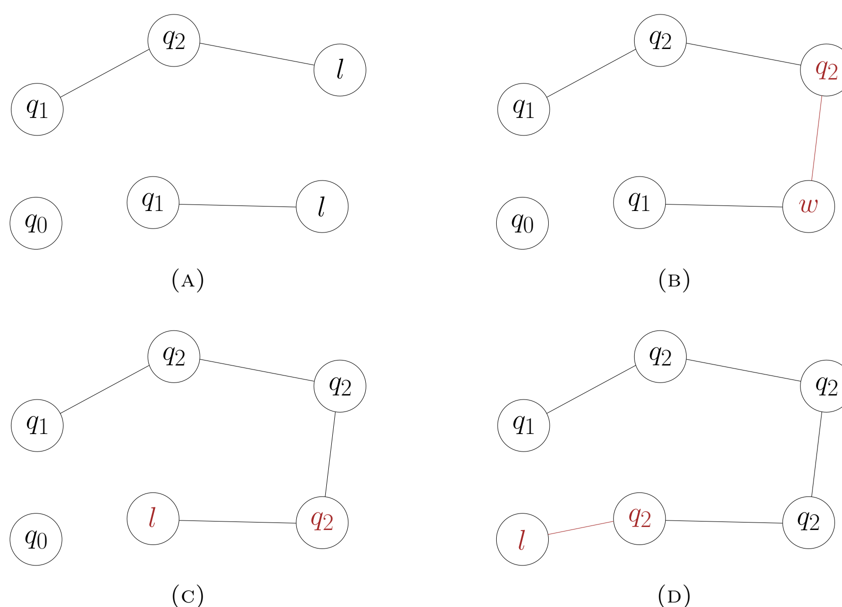


FIGURE 1.1: An example of three execution steps of the Spanning line protocol (Protocol 2). Node-states in red represent state updates and lines in red represent new edge activations.

In Figure 1.1a, there are two lines of size three and two, and a single isolated node. Each line has an endpoint in state q_1 and one in state l, while all intermediate nodes are in state q_2. Nodes in state l can only be on an endpoint of a line. Nodes in state l are used in order to absorb isolated nodes (i.e., expand the line), and connect with other lines in the population. Then, as shown in Figure 1.1b, the two endpoints in state l eventually interact with each other. They become connected and one of

them changes its state to q_2, while the other node becomes w. w is a walking state, meaning that when it interacts with a connected to it node in state q_2, they exchange their states. The walking state eventually reaches the endpoint in state q_1, resulting to state l (Figures 1.1c and 1.1d). This protocol, as shown in [113], constructs a spanning line in $\Omega(n^4)$ and $O(n^5)$ expected time under a uniform random scheduler.

In Section 4 we use the model of Network Constructors and we consider adversarial crash faults of nodes. This means that in each round, the scheduler can pick a node u and remove it from the configuration. We say that u crashes, and it never participates in the execution of the protocol again.

1.3 Crystal Structure Prediction

Crystal structure prediction (CSP) is one of the major problems in computational chemistry, that comprises of a more applied version of stability. The goal is to calculate the most stable structure, or ground state, of solids from first principles, which corresponds to the structure with the lowest free energy. CSP gained popularity in the 1980s, following statements from John Maddox on how scientists still struggle to predict crystal formation [101].

"*One of the continuing scandals in the physical sciences is that it remains in general impossible to predict the structure of even the simplest crystalline solids from a knowledge of their chemical composition.*"

Understanding the behaviors of materials at the atomic scale is fundamental to modern science and technology. The crystal structure of a material bears information about its properties, therefore making it one of the greatest problems of computational chemistry.

1.3.1 Algorithmic approaches

Traditional approaches to crystal structure prediction involve trial and error, rendering experimental solutions difficult, or in some cases impossible. An alternative approach to this problem which is now possible due to the significant progress in computational power, is by utilizing computational techniques in order to predict the lowest free energy structure. Given that this problem is highly-dimensional and the free energy surface is extremely rugged with high peaks and saddle points, it becomes almost impossible to classify that huge number of energy minima on the energy surface. Therefore, optimization methods that have been employed include evolutionary algorithms [97, 146, 152], data mining [40, 72, 89], simulated annealing [71, 149], and minima hopping [4].

The first step to crystal structure prediction is the formation of an appropriate evaluation function. Several methods have been developed, with ab initio calculations of the free energy being the most accurate methods but computationally expensive, while methods that approximate the free energy are computationally cheaper but can lead to a misguided search. Density functional theory (DFT) is a computational quantum mechanical modeling method, which usually indicates the ground truth when it comes to the calculation of the free energy of a structure. Other computational methods that are used to estimate the forces between atoms within molecules, also known as *force fields*, include the Lennard-Jones potential and the Buckingham potential. Lennard-Jones potential is the potential that has been studied most extensively and was not derived analytically but fitted numerically in contrast with theoretical Buckingham potential. Although Buckingham usually performs better than Lennard-Jones, its performance degrades in short range interactions [26, 92]. Finally, Buckingham-Coulomb potential adds the Coulomb interaction to the Pauli repulsion and van der Waals interaction which guarantees satisfactory reliable results for the representation of ionic interactions [98].

The next step of every crystal structure prediction algorithm is to define a way to sample new structures. Brute-force approaches are characterized by combinatorial explosion, rendering them highly inefficient. Hence, heuristic algorithms provide a natural way to efficiently find approximate solutions to crystal structure prediction. In Chapter 5 we provide an algorithm which is based on local search; an optimization method which moves from solution to solution by applying local changes until a solution deemed optimal is found or a time bound is elapsed. In particular, given a solution x, it computes the neighborhood of x, $N(x)$, and updates the current solution with an improved one from $N(x)$ using a local rule r (i.e., $x' = r(N(x))$). This is repeated until an optimal, with respect to N, solution is found, or until a time bound is reached. In Section 5.3 we define three such local neighborhoods, and in Section 5.4 we describe our local search algorithms in detail.

1.4 Mobile Agents on Dynamic Graphs

During the past two decades, there has been a rapid development in the field of distributed computing by mobile agents. A set of computational entities operating in a discrete universe, modeled as a graph, are able to move from node to node, while their movement is constrained by the nature of the graph they operate in. The main concern in the related literature is the study of the computational and complexity issues arising in such systems.

Regarding the nature of the graph, two different settings are identified; static and dynamic graphs. For the static setting, there is a rich variety of results, with many

model variations and problems examined so far. Examples of such problems include: the exploration problem, where a set of $k \geq 1$ agent(s) is required to navigate in the graph, visiting all nodes in a systematic manner; the rendezvous problem, where two mobile agents must move along the n nodes of the graph so as to minimize the time required to meet; black hole search, where a malicious node destroys any agent passing by it, and the goal is for at least one agent to survive and have the entire map of the graph with the location of the black hole; and the gathering problem, where a set of agents, initially located in arbitrary locations of the graph, must gather in the same node, not fixed in advance, and terminate. See [75] for a comprehensive survey on several variations and problems in the field of distributed computing by mobile agents.

The growing interest in problems that are inherently dynamic in nature have recently given rise in studying computational entities moving on top of dynamic graphs. Examples of problems that have been studied so far include the exploration problem [17, 70, 74] and the gathering problem [59, 114]. The main challenge in the design of algorithms in this setting is the uncertainty of the changes of the graph that the agents have to overcome. Problems that are simple to solve in the static case are far from trivial in a dynamic setting.

1.4.1 Motivation and related work

Social networks [120, 126, 136, 151], wireless networks [121], transportation networks [93] are a few examples of networks that are continuously changing and evolving with time. In social networks, the topology usually represents social relationships between a group of people, and it is evolving with time as the group and its relationships are updated (e.g., new individuals join or leave the group, new relationships are made or existing relationships are broken). Communication systems with a large number of mobile communicating devices is another example of a system where nodes join the network, leave, and move around, while communication links appear and disappear.

The recent interest in studying such systems is motivated by the plethora of their real-world applications, requiring us to develop models that capture their dynamics. These networks are best represented by dynamic graph models. Dynamic graphs, also known as temporal graphs [94, 103], temporal networks [90, 102], time-varying graphs [30], or evolving graphs [5, 135], are graphs that change with time. Motivated by the widespread impact of such networks, several models that capture their dynamic nature have been proposed in the literature. For a comprehensive survey on dynamic graph models see [153]. A dynamic graph can be thought of as a special case of labeled graphs, where labels capture some measure of time. For example, discrete sets of labels attached to each edge of the graph may correspond to the discrete time steps in which the corresponding edges appear. This notion of time gives rise to a

multitude of new challenging problems in graphs, whose solution will enable several important modern applications.

1.4.2 A formal model for dynamic networks

A network is most commonly modeled as an undirected connected graph $G_U = (V, E)$, referred to hereafter as an *underlying graph*. Each topological event in a dynamic graph can be viewed as the transformation from one static graph to another. Hence, the evolution of a dynamic graph $\mathcal{G}$ can be described as a sequence of static graphs. In particular, given an underlying graph $G_U = (V, E)$ on n vertices, a *dynamic graph* on G_U is a sequence $G_D = \{G_t = (V, E_t) : t \in \mathbb{N}\}$ of graphs such that $E_t \subseteq E$ for all $t \in \mathbb{N}$. Every G_t is the snapshot of G_D at time-step t.

A very common assumption in the literature of dynamic graphs is that the sequence G_D is controlled by an adversarial scheduler, subject to some constraints. In [96], the authors defined the T-interval connectivity model, in which throughout every block of T consecutive rounds there must exist a connected spanning subgraph. For $T = 1$ this means that the graph is connected in every round, but changes arbitrarily between rounds. The definition of 1-interval connectivity of [96] considers the case where the underlying graph is a complete clique. In Chapter 6, we generalize this to any underlying graph G_U, meaning that $G'_D = (V, \bigcup_t E_t) \subseteq G_U$.

1.5 Thesis Contribution

1.5.1 Counting and leader election in population protocols

In Chapter 2 and Chapter 3 we consider the Population Protocol model and we study the problems of counting the population size and electing a leader agent. As in Protocol 1, we employ the use of simple epidemics in order to provide efficient solutions to counting the size of a population of agents, and for electing a leader. Note that an epidemic process requires $\Theta(\log n)$ parallel time to infect the whole population, under a uniform random scheduler.

The problem of exact population counting is that of designing a protocol so that the agents eventually converge to a state where each agent encodes in its state the exact population size n. Relaxations of this problem have also been considered in the literature, and require the population to calculate an approximate population size (e.g., $2^{\lceil \log n \rceil}$). This freedom allows us to design protocols with exponentially smaller space complexity than the problem of exact counting. In Chapter 2 we give a protocol which provides a constant factor approximation of $\log n$, or an upper bound n_e which is at most n^a for some constant $a > 1$ (if the agents are provided with enough memory to store n^a). Our protocol assumes the existence of a unique leader in the population,

which initiates an epidemic process and then observes its progress in the population. The runtime of the protocol until stabilization is $\Theta(\log n)$ parallel time. Each node except from the unique leader uses only a constant number of bits. However, the leader is required to use $\Theta(\log\log n)$ bits. If we require the whole population to converge to a state that encodes the approximation of $\log n$, then we can utilize an additional epidemic process containing that information, which asymptotically doesn't affect the convergence time of the protocol.

In Chapter 3 we work on the leader election problem. We provide an algorithm that terminates in $O(\frac{\log^2 n}{\log m})$ parallel time, where $1 \leq m \leq n$ is a parameter, using $O(\max\{\log m, \log\log n\})$ bits. By adjusting the parameter m between a constant and n, we obtain a leader election protocol whose time and space can be smoothly traded off between $O(\log^2 n)$ to $O(\log n)$ time and $O(\log\log n)$ to $O(\log n)$ bits. Finally, our approximate counting protocol uses a leader, while our leader election protocol uses an approximate knowledge of n. We then compose our protocols in order to obtain a uniform protocol for both problems. We do not provide a formal analysis of this protocol, but simulation results suggest that it works as expected.

1.5.2 Fault tolerant network constructors

In Chapter 4 we consider adversarial crash faults of nodes in the network constructors model [113]. We first show that, without further assumptions, the class of graph languages that can be (stably) constructed under crash faults is non-empty but small. In particular, if an unbounded number of crash faults may occur, we prove that (i) the only constructible graph language is that of spanning cliques and (ii) a strong impossibility result holds even if the size of the graphs that the protocol outputs in populations of size n needs to grow with n (the remaining nodes being *waste*). When there is a finite upper bound f on the number of faults, we show that it is impossible to construct any *non-hereditary* graph language and leave as an interesting open problem the *hereditary case*. On the positive side, by relaxing our requirements we prove that: (i) permitting linear waste enables to construct on $n/(2f)-f$ nodes, any graph language that is constructible in the fault-free case, (ii) *partial constructibility* (i.e., not having to generate all graphs in the language) allows the construction of a large class of graph languages. We then extend the original model with a minimal form of *fault notifications*. Our main result here is a *fault-tolerant universal constructor*: We develop a fault-tolerant protocol for *spanning line* and use it to simulate a linear-space Turing Machine M. This allows a fault-tolerant construction of any graph accepted by M in linear space, with waste $min\{n/2+f(n),\ n\}$, where $f(n)$ is the number of faults in the execution. We then prove that increasing the permissible waste to $min\{2n/3+f(n),\ n\}$ allows the construction of graphs accepted by an $O(n^2)$-space Turing Machine, which is asymptotically the maximum simulation space

that we can hope for in this model. Finally, we show that logarithmic local memories can be exploited for a no-waste fault-tolerant simulation of any such protocol.

1.5.3 Crystal structure prediction via oblivious local search

In Chapter 5 we study Crystal Structure Prediction, one of the major problems in computational chemistry. This is essentially a continuous optimization problem, where many different, simple and sophisticated, methods have been proposed and applied. The simple searching techniques are easy to understand, usually easy to implement, but they can be slow in practice. On the other hand, the more sophisticated approaches perform well in general, however almost all of them have a large number of parameters that require fine tuning and, in the majority of the cases, chemical expertise is needed in order to properly set them up. In addition, due to the chemical expertise involved in the parameter-tuning, these approaches can be *biased* towards previously-known crystal structures.

Our contribution is twofold. Firstly, we formalize the Crystal Structure Prediction problem, alongside several other intermediate problems, from a theoretical computer science perspective. Secondly, we propose an oblivious algorithm for Crystal Structure Prediction that is based on local search. Oblivious means that our algorithm requires minimal knowledge about the composition we are trying to compute a crystal structure for. In addition, our algorithm can be used as an intermediate step by *any* method. Our experiments show that our algorithms outperform the standard basin hopping, a well studied algorithm for the problem.

1.5.4 Gathering in dynamic graphs

In Chapter 6 we examine the problem of gathering $k \geq 2$ agents (or multi-agent rendezvous) in dynamic graphs which may change in every round. We consider a variant of the 1-interval connectivity model [96] described in Section 1.4.2, in which all instances (snapshots) are always connected spanning subgraphs of an underlying graph, not necessarily a clique. The agents are identical and not equipped with explicit communication capabilities, and are initially arbitrarily positioned on the graph. The problem is for the agents to gather at the same node, not fixed in advance. We first show that the problem becomes impossible to solve if the underlying graph has a cycle. In light of this, we study a relaxed version of this problem, called *weak gathering*, where the agents are allowed to gather either at the same node, or at two adjacent nodes. Our goal is to characterize the class of 1-interval connected graphs and initial configurations in which the problem is solvable, both with and without *homebases* (the nodes that the agents are initially placed are identified by an identical mark, visible to any agent passing by it). On the negative side we show that when the

underlying graph contains a spanning bicyclic subgraph and satisfies an additional connectivity property, *weak gathering* is unsolvable, thus we concentrate mainly on unicyclic graphs. As we show, in most instances of initial agent configurations, the agents must meet on the cycle. This adds an additional difficulty to the problem, as they need to explore the graph and recognize the nodes that form the cycle. We provide a deterministic algorithm for the solvable cases of this problem that runs in $O(n^2 + nk)$ number of rounds.

1.6 Author's Publications Presented in this Thesis

<table>
<tr><th colspan="5">Publications</th></tr>
<tr><th>Chapter</th><th>Title</th><th>Authors</th><th>Conference</th><th>Journal</th></tr>
<tr><td>2, 3</td><td>Simple and Fast Approximate Counting and Leader Election in Populations</td><td rowspan="3">Othon Michail,
Paul G. Spirakis,
Michail Theofilatos</td><td>SSS 2018 [1][117]
Brief in SIROCCO 2018 [2] [115]</td><td>Information and Computation (2021) [118]</td></tr>
<tr><td>4</td><td>Fault Tolerant Network Constructors</td><td>SSS 2019 [3][116]</td><td>Information and Computation (to appear)</td></tr>
<tr><td>6</td><td>Beyond Rings: Gathering in 1-Interval Connected Graphs</td><td>-</td><td>Parallel Processing Letters (2021) [114]</td></tr>
<tr><td>5</td><td>Crystal Structure Prediction via Oblivious Local Search</td><td>Dmytro Antypov,
Argyrios Deligkas,
Vladimir Gusev,
Matthew J. Rosseinsky,
Paul G. Spirakis,
Michail Theofilatos</td><td>SEA 2020 [4][13]</td><td>-</td></tr>
</table>

Table 1.1: List of author's publications presented in this thesis

[1]20th International Symposium on Stabilization, Safety, and Security of Distributed Systems
[2]25th International Colloquium on Structural Information and Communication Complexity
[3]21st International Symposium on Stabilization, Safety, and Security of Distributed Systems
[4]18th Symposium on Experimental Algorithms

Part I

Population Protocols

Chapter 2

Approximate Counting in Population Protocols

In this chapter, we study the problem of approximate counting for *population protocols*: networks of finite-state anonymous agents that interact randomly under a uniform random scheduler. We give a protocol which provides a constant factor approximation of $\log n$ of the population size n, or an upper bound n_e which is at most n^a for some constant $a > 1$ (if the nodes are provided with enough memory to store n^a). This protocol assumes the existence of a unique leader in the population and stabilizes in $\Theta(\log n)$ parallel time, using constant number of bits in every node, except from the unique leader which is required to use $\Theta(\log \log n)$ bits.

2.1 Introduction

Many distributed tasks require the existence of a leader prior to the execution of the protocol and, furthermore, some knowledge about the system (for instance the size of the population) can also help to solve these tasks more efficiently with respect both to time and space. *Counting* is a fundamental problem in distributed computing, where the nodes must determine the size n of the population. Similarly, *approximate counting* is the problem in which the nodes must determine an estimation k of the population size n. Counting can be then considered as a special case of approximate counting, where $k = n$.

One fundamental measure of convergence is the total number of pairwise interactions until all agents are in a correct output state. We also consider models in which reactions occur in parallel according to a Poisson process. This gives an equivalent distribution over sequences of reactions but suggests a measure of parallel time, assuming that each agent participates in an expected $\Theta(1)$ interactions per time unit.

This time measure is asymptotically equal to the number of interactions divided by the size n of the population.

Consider the setting in which an agent is in an initial state a, the rest $n-1$ agents are in state b and the only existing transition is $(a, b) \rightarrow (a, a)$. This is the *one-way epidemic* process and it can be shown that the expected time to convergence until all nodes change their state to a and under the uniform random scheduler is $\Theta(n \log n)$ (e.g., [9]), thus $\Theta(\log n)$ *parallel time*. In this chapter, we make an extensive use of epidemics, which means that information is being spread throughout the population, thus all nodes will obtain this information in $O(\log n)$ expected parallel time. By observing the rate of the epidemic spreading under the uniform random scheduler, we can extract valuable information about the population. This is the key idea of our *Approximate Counting* algorithm.

2.1.1 Related work

The framework of population protocols was first introduced by Angluin et al. [7] in order to model the interactions in networks between small resource-limited mobile agents. When operating under a uniform random scheduler, population protocols are formally equivalent to a restricted version of stochastic Chemical Reaction Networks (CRNs), which model chemistry in a well-mixed solution [142]. "CRNs are widely used to describe information processing occurring in natural cellular regulatory networks, and with upcoming advances in synthetic biology, CRNs are a promising programming language for the design of artificial molecular control circuitry" [35, 62]. Results in both population protocols and CRNs can be transfered to each other, owing to a formal equivalence between these models.

Angluin et al. [11] showed that all predicates stably computable in population protocols (and certain generalizations of it) are semilinear. Semilinearity persists up to $o(\log \log n)$ local space but not more than this [34]. Moreover, the computational power of population protocols can be increased to the commutative subclass of $\mathbf{NSPACE}(n^2)$, if we allow the processes to form connections between each other that can hold a state from a finite domain [109], or by equipping them with unique identifiers, as in [87]. For introductory texts to population protocols the interested reader is encouraged to consult [15, 109] and [111] (the latter discusses population protocols and related developments as part of a more general overview of the emerging theory of dynamic networks).

For the counting problem, the most studied case is that of *self-stabilization*, which makes the strong adversarial assumption that arbitrary corruption of memory is possible in any agent at any time, and promises only that eventually it will stop. Thus, the protocol must be designed to work from any possible configuration of the memory of each agent. It can be shown that counting is *impossible* without having

one agent (the "base station") that is protected from corruption [21]. In this scenario $\Theta(n \log n)$ time is sufficient [20] and necessary [14] for self-stabilizing counting.

In the less restrictive setting in which all nodes start from the same state (apart possibly from a unique leader and/or unique ids), there has been a growing interest recently. In [105], the authors proposed a terminating protocol in which a pre-elected leader equipped with two n-counters computes an approximate count between $n/2$ and n in $O(n \log n)$ parallel time with high probability. The idea is to have the leader implement two competing processes, running in parallel. The first process counts the number of nodes that have been encountered once, the second process counts the number of nodes that have been encountered twice, and the leader terminates when the second counter catches up the first. In the same paper, also a version assuming unique ids instead of a leader was given. Several leaderless solutions for approximate counting have been proposed recently [2, 25, 64, 65, 78, 143].

Regarding the exact counting problem, a uniform protocol is provided by our team and other co-authors in [66]. In this work, the authors give the first protocol that solves this problem in sublinear time. The protocol converges in $O(\log n \log \log n)$ time and uses $O(n^{60})$ states ($O(1) + 60 \log n$ bits of memory per agent) with probability $1 - O(\frac{\log \log n}{n})$. The time to converge is also $O(\log n \log \log n)$ in expectation. Crucially, unlike most published protocols with $\omega(1)$ states, this protocol is uniform: it uses the same transition algorithm for any population size, so does not need an estimate of the population size to be embedded into the algorithm. A subprotocol is the first uniform sublinear-time leader election population protocol, taking $O(\log n \log \log n)$ time and $O(n^{18})$ states. The state complexity of both the counting and leader election protocols can be reduced to $O(n^{30})$ and $O(n^9)$ respectively, while increasing the time to $O(\log^2 n)$. In a very recent work [25], the authors provide a protocol that always achieves (i.e., with probability 1) exact counting in $O(\log n)$ parallel time, using $O(n \log n \log \log n)$ states, which is improved to $O(n \log n)$ states if we allow the protocol to output the exact count with probability $1 - \frac{O(1)}{n}$. See [63] for a comprehensive survey on approximate and exact counting.

Finally, the task of counting has also been studied in the related context of worst-case dynamic networks [30, 91, 96, 100, 108].

2.1.2 Preliminaries

In this chapter, the system consists of a population V of n distributed and anonymous (i.e., do not have unique IDs) *processes*, also called *nodes* or *agents*, that are capable to perform local computations. Each of them is executing as a deterministic state machine from a finite set of states Q according to a transition function $\delta : Q \times Q \to Q \times Q$. Their interaction is based on the probabilistic (uniform random) scheduler, which picks in every discrete step a random edge from the complete graph G on n

vertices. When two agents interact, they mutually access their local states, updating them according to the transition function δ. The transition function is a part of the population protocol which all nodes store and execute locally. For a formal description of the Population Protocol model, see Section 1.1.2.

The time is measured as the number of steps until stabilization, divided by n (*parallel time*). The protocol that we propose does not enable or disable connections between nodes, in contrast with [113], where Michail and Spirakis considered a model where a (virtual or physical) connection between two processes can be in one of a finite number of possible states. The transition function that we present throughout this paper, follows the notation $(x, y) \to (z, w)$, which refers to the process states before *(x and y)* and after *(z and w)* the interaction, that is, the transition function maps pairs of states to pairs of states.

Approximate counting problem We define as *approximate counting* the problem in which a leader must determine an estimation n_e of the population size, where $n^\alpha \leq n_e \leq n^\beta$, $\alpha < \beta$. We call the constants α and β the estimation parameters.

2.1.3 Contribution and roadmap for the chapter

In this chapter we employ the use of simple epidemics in order to provide an efficient solution to approximate counting the size of a population of agents. Our goal is to get polylogarithmic parallel time and to also use small memory per agent.

We start by providing a protocol which finds an upper bound n_e of the size n of the population, where n_e is at most n^a for some $a > 1$ (if the nodes are provided with enough memory to store that value). If the unique leader is provided with $O(\log\log n)$ bits of memory, the protocol gives a constant factor approximation of $\log n$. This protocol assumes the existence of a unique leader in the population. The runtime of the protocol until stabilization is $\Theta(\log n)$ parallel time. Each node except from the unique leader uses only a constant number of bits. However, the leader is required to use $\Theta(\log\log n)$ bits. In Section 2.2.1 we give our Approximate Counting protocol, in Section 2.2.2 we prove its correctness, and finally, in Section 2.3, experiments that support our analysis can be found.

2.2 Fast Approximate Counting with a Unique Leader

2.2.1 Abstract description and protocol

In this section, we construct a protocol which solves the problem of approximate counting. Our probabilistic algorithm for solving the approximate counting problem requires a unique leader who is responsible to give an estimation on the number of

nodes. It uses the epidemic spreading technique and it stabilizes in $O(\log n)$ parallel time. There is initially a unique leader in state l and all other nodes are in state q. The leader l stores two counters in its local memory, initially both set to 0. The counters are part of the leader's state and can be updated only during an interaction. We use the notation $l_{(c_q,c_a)}$ to represent the leader states, where c_q is the value of the first counter and c_a is the value of the second one. The leader, after the first interaction starts an epidemic by turning a q node into an a node. Whenever a q node interacts with an a node, its state becomes a $((a,q) \rightarrow (a,a))$. The first counter c_q is being used for counting the q nodes and the second counter c_a for the a nodes, that is, whenever the leader l interacts with a q node, the value of the counter c_q is increased by one and whenever l interacts with an a node, c_a is increased by one. The termination condition is $c_q = c_a$ and then the leader holds a constant-factor approximation of $\log n$, which we prove that with high probability is $2^{c_q+1} = 2^{c_a+1}$. We first describe a simple terminating protocol that guarantee with high probability $n^\alpha \leq n_e \leq n^\beta$, for two constants $\alpha < \beta$, i.e., the population size estimation is polynomially close to the actual size. Chernoff bounds then imply that repeating this protocol a constant number of times suffices to obtain $n/2 \leq n'_e \leq 2n$ with high probability.

Protocol 3 Approximate Counting (APC)

$Q = \{q,\ a,\ l_{(c_q,c_a)},\ halt\}$: $c_q \geq 0,\ c_a \geq 0$
δ :

1: $(l_{(0,0)},\ q) \rightarrow (l_{(1,0)},\ a)$
2: $(a,\ q) \rightarrow (a,\ a)$
3: $(l_{(c_q,c_a)},\ q) \rightarrow (l_{(c_q+1,c_a)},\ q),\ if\ c_q > c_a$
4: $(l_{(c_q,c_a)},\ a) \rightarrow (l_{(c_q,c_a+1)},\ a),\ if\ c_q > c_a$
5: $(l_{(c_q,c_a)},\ \cdot) \rightarrow (halt,\ \cdot),\ if\ c_q = c_a$

2.2.2 Analysis

Lemma 2.1. *When half or less of the population has been infected, with high probability $c_q > c_a$. In fact, when the number of infected nodes is equal to the number of non-infected nodes, $c_q - c_a \approx \ln(n/2) - \sqrt{\ln n} > 0$ w.h.p.*

Proof. We divide the process of the epidemic elimination into rounds i, where round i means that there exist i infected nodes in the population (nodes in state a). Call an interaction a success if an effective rule applies and a new node in state a appears on some node (the number of infected nodes is increased by one). Let the random variable (r.v.) X be the total number of interactions between the leader l and non-infected nodes (state q), the random variable Y be the total number of interactions

between l and infected nodes and the r.v. I be the total number of interactions in the population until all nodes become infected. We also define the r.v. X_i, Y_i and I_i to be the corresponding numbers in round i. Then, it holds that $X = \sum_{i=1}^{r} X_i$, $Y = \sum_{i=1}^{r} Y_i$ and $I = \sum_{i=1}^{r} I_i$, for any $r > 0$. Finally, let the r.v. X_{ij} and Y_{ij} be independent Bernoulli trials such that for $1 \leq j \leq I_i$, $\Pr[X_{ij} = 1] = p_{Xi}$, $\Pr[X_{ij} = 0] = 1 - p_{Xi}$, $\Pr[Y_{ij} = 1] = p_{Yi}$ and $\Pr[Y_{ij} = 0] = 1 - p_{Yi}$. This means that in every interaction in round i, the leader, if chosen, interacts with a non-infected node with probability p_{Xi} and with an infected node with probability p_{Yi}. Then, it holds that $X_r = \sum_{i=1}^{I_r} X_{ij}$ and $Y_r = \sum_{i=1}^{I_r} Y_{ij}$, where I_r is the number of interactions until a success in round r.

$$p_{Xi} = \frac{2(n-i)}{n(n-1)},\ p_{Yi} = \frac{2i}{n(n-1)} \text{ and } p_{Ii} = \frac{2i(n-i)}{n(n-1)}$$

We also divide the whole process into two phases; the first phase ends when half of the population has been infected, that is $1 \leq i \leq \frac{n}{2}$ and for the second phase it holds that $\frac{n}{2} + 1 \leq i \leq n$. We shall argue that if the counter c_q reaches a value which is a function of n, before the second counter c_a reaches c_q, the leader gives a good estimation. We use X^a and Y^a to indicate the r.v. X and Y during the first phase and X^b, Y^b for the second phase.

For $1 \leq i \leq \frac{n}{2}$ (first phase) and by linearity of expectation we have:

$$E[X^a] = E[\sum_{i=1}^{n/2} X_i] = E[\sum_{i=1}^{n/2}\sum_{j=1}^{I_i} X_{ij}] = \sum_{i=1}^{n/2}\sum_{j=1}^{I_i} E[X_{ij}]$$

and by Wald's equation, we have that $E[\sum_{i=1}^{I_i} X_{ij}] = E[I_i]E[X_{ij}]$.

$$E[X^a] = \sum_{i=1}^{n/2} \frac{n(n-1)}{2i(n-i)} \frac{2(n-i)}{n(n-1)} = \sum_{i=1}^{n/2} \frac{1}{i} = H_{n/2} = \ln\frac{n}{2} + a_{n/2} \geq \ln\frac{n}{2}$$

where $H_{n/2}$ denotes the $(\frac{n}{2})$th Harmonic number and $0 < a_n < 1$ for all $n \in \mathbb{N}$ (Euler-Mascheroni constant).

$$E[Y^a] = E[\sum_{i=1}^{n/2} Y_i] = E[\sum_{i=1}^{n/2}\sum_{j=1}^{I_i} Y_{ij}] = \sum_{i=1}^{n/2}\sum_{j=1}^{I_i} E[Y_{ij}]$$

and by Wald's equation, we have that $E[\sum_{i=1}^{I_i} Y_{ij}] = E[I_i]E[Y_{ij}]$.

$$E[Y^a] = \sum_{i=1}^{n/2} \frac{n(n-1)}{2i(n-i)} \frac{2i}{n(n-1)} = \sum_{i=1}^{n/2} \frac{1}{n-i} = \sum_{i=1}^{n-1} \frac{1}{i} - \sum_{i=1}^{n/2-1} \frac{1}{i} = H_{n-1} - H_{n/2-1}$$
$$\approx \ln 2$$

We now make use of the following Chernoff Bounds:

$$\Pr[X \leq (1-\delta)\mu] \leq e^{-\frac{\mu\delta^2}{2}}, \delta \geq 0$$

$$\Pr[X \geq (1+\delta)\mu] \leq e^{-\frac{\mu\delta^2}{2+\delta}}, \delta \geq 0$$

The probabilities that the r.v. X^a is less than $(1-\delta)E(X^a)$ and more than $(1+\delta)E(X^a)$ are

$$\Pr[X^a \leq (1-\delta)E(X^a)] \leq e^{-\frac{\ln(n/2)\delta^2}{2}} = \frac{1}{(\frac{n}{2})^{\delta^2/2}}$$

$$\Pr[X^a \geq (1+\delta)E(X^a)] \leq e^{-\frac{\ln(n/2)\delta^2}{2+\delta}} = \frac{1}{(\frac{n}{2})^{\delta^2/(2+\delta)}}$$

that is, X^a does not deviate far from its expectation. The probability that the r.v. Y^a is more than $(1+\delta)E(Y^a)$, for $\delta = \frac{\ln n}{\ln 2} + 2$ is

$$\Pr[Y^a \geq (1+\delta)E(Y^a)] \leq e^{-\frac{\mu\delta^2}{2+\delta}} < e^{-\frac{\mu(\delta+2)(\delta-2)}{2+\delta}} = e^{-\mu(\delta-2)} = e^{-\ln 2(\frac{\ln n}{\ln 2}+2-2)}$$
$$= e^{-\ln n} = \frac{1}{n}$$

Thus, the leader interacts a constant number of times and w.h.p. less than $(1+\delta)E[Y^a]$ times with infected nodes during the first phase (half of the population is infected). In addition, it interacts $O(\log n)$ times with non-infected nodes w.h.p.. In Section 2.3, we have tested our results and the Figure 2.3 confirms this behavior. During the second phase, the infected nodes are more than the non-infected nodes, thus, eventually, the second counter c_a will reach c_q and the leader terminates. By the end of the first phase, the difference between the two counters is w.h.p. $c_q - c_a \leq \ln(n/2) - \sqrt{\log n} > 0$.

Corollary 2.2. APC *(Protocol 3) does not terminate w.h.p. until more than half of the population has been infected.*

It now suffices to show that the first counter c_q does not continue to rise significantly. During the second phase, where $\frac{n}{2}+1 \leq i \leq n$, we have

$$E[X^b] = E[\sum_{i=n/2+1}^{n} X_i] = H_n - H_{n/2} \approx \ln 2$$

By Chernoff Bound, the probability that the r.v. X^b is more than $(1+\delta)E(X^b)$, for $\delta = \frac{\ln n}{\ln 2} + 2$ is

$$\Pr[X^b \geq (1+\delta)E(X^b)] < e^{-\frac{\mu(\delta+2)(\delta-2)}{2+\delta}} = e^{-\ln n} = \frac{1}{n}$$

□

Lemma 2.3. APC *terminates after* $\Theta(\log n)$ *parallel time w.h.p., using* $\Theta(\log\log n)$ *bits of memory in the leader node.*

Proof. After half of the population has been infected, it holds that $|c_a - c_q| = \Theta(\log n)$. When this difference reaches zero, the unique leader terminates. We focus only on the effective interactions, which are always interactions between the leader l and nodes in states a or q. The probability that an interaction is (l,a) is $p_i = i/n > 1/2$, as more than half of the population is infected. Thus, the probability that an interaction is (l,q) is $q_i = 1 - p_i = (n-i)/n < 1/2$. In fact, the probability p_i is constantly decreasing as the epidemic spreads throughout the population. This process may be viewed as a random walk on a line with positions $[0,\infty)$. The particle starts from position $a\log n$ and there is an absorbing barrier at 0. The position of the particle corresponds to the difference $|c_a - c_q|$ of the two counters and it moves towards zero with probability $p_i > 1/2$. By the basic properties of random walks, after $\Theta(\log n)$ steps, the particle will be absorbed at 0. Thus, the total parallel time to termination is $\Theta(\log n)$.

The state space of the leader is the product of the counters' values, the size of which is bounded by $O(\log n)$. Thus, the number of states is $O(\log^2 n)$, which translates to $O(\log\log n)$ bits of memory. The rest of the nodes can only be in two states (infected or non-infected). □

Theorem 2.4. *When* $c_q = c_a$, $E[2^{c_q}] = n$, *and* $n \leq 2^{c_q} < n^2$ *w.h.p..*

Proof. The first counter c_q, during the first phase (half or less of the population is infected) is in expectation $ln(n/2)$ and w.h.p. $c_q = \Theta(\ln n)$, while $c_a = O(\sqrt{\ln n})$ w.h.p. During the second phase the first counter is increased by a small constant number on expectation, and w.h.p. less than $\ln n$, while the second counter c_a eventually catches up c_q. This means that the unique leader of Protocol 3, when it terminates it holds

with high probability the value $c_q = [\ln n, 2\ln n]$ ($c_q = \Theta(\ln n)$). Thus, $n < e^{c_q} < n^2$ is an upper bound on the population size, where e is the Euler's number. $\square$

2.3 Experiments

We have also measured the stabilization time of our *Approximate Counting with a unique leader* protocol for different network sizes. We have executed it 100 times for each population size n, where $n = 2^i$ and $i = [4, 14]$. The stabilization time is shown in Figure 2.1. The algorithm always gives a constant factor approximation of $\log n$, as shown in Figure 2.2. Moreover, in Figure 2.3, we show the values of the counters c_q and c_a, when half of the population has been infected by the epidemic. These experiments support our analysis, while the counter of infected nodes reaches a constant number and the counter of non-infected nodes reaches a value close to $\log n$.

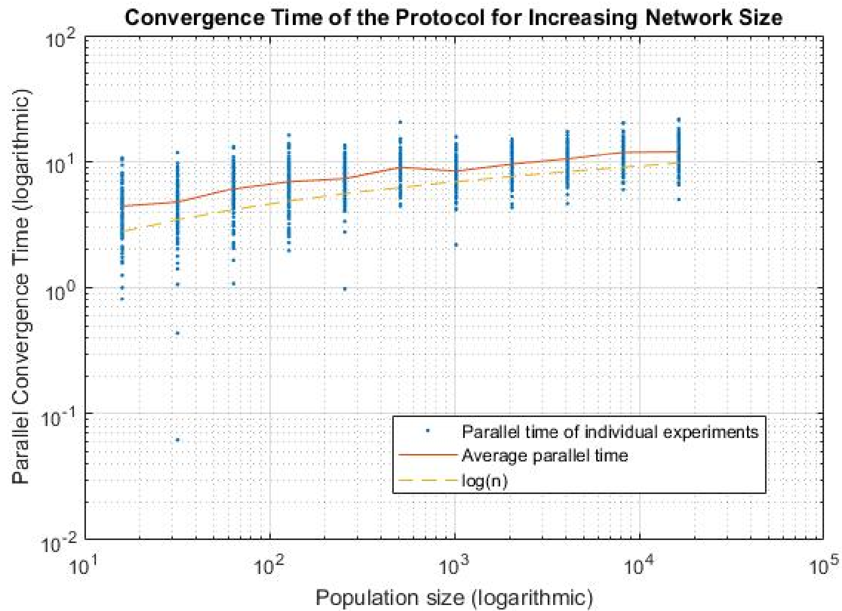


FIGURE 2.1: Convergence time of *Approximate Counting (APC)* with a unique leader protocol. Both axes are logarithmic. The dots represent the results of individual experiments and the line represents the average values for each network size.

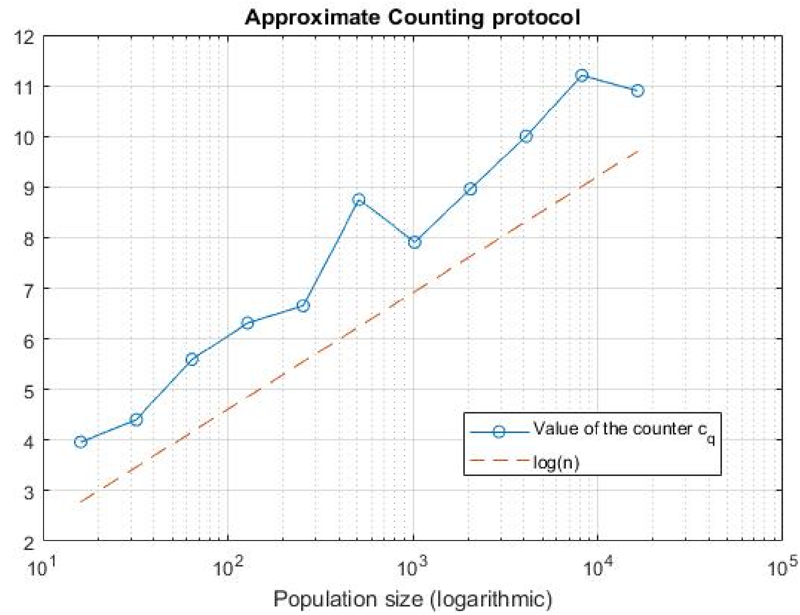


FIGURE 2.2: *Approximate Counting (APC)* with a unique leader. Estimations and actual sizes of the population.

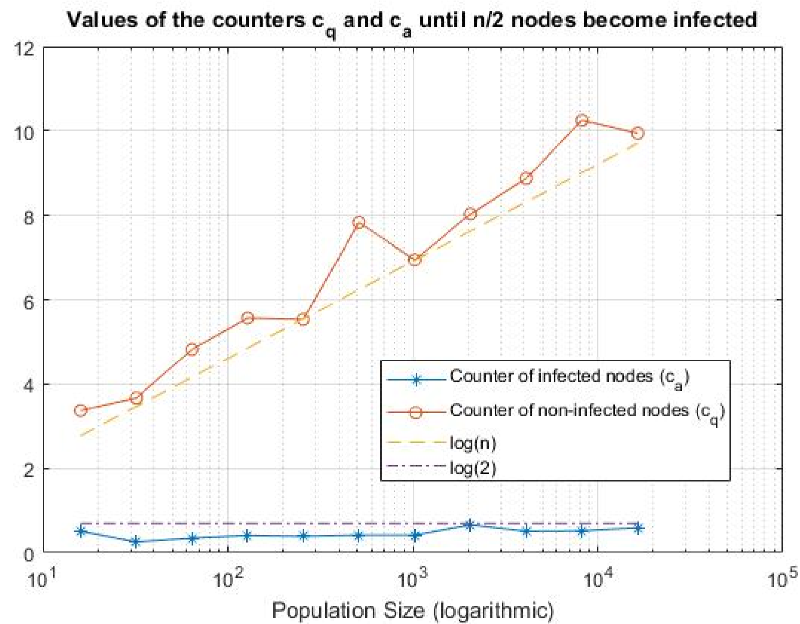


FIGURE 2.3: *Approximate Counting (APC)* with a unique leader. Counters c_q and c_a when half of the population has been infected by the epidemic.

Chapter 3

Leader Election in Population Protocols

In this chapter we study the problem of leader election for *population protocols*: networks of finite-state anonymous agents that interact randomly under a uniform random scheduler. We provide a protocol that terminates in $O(\frac{\log^2 n}{\log m})$ parallel time, where $1 \leq m \leq n$ is a parameter, using $O(\max\{\log m, \log\log n\})$ bits. By adjusting the parameter m between a constant and n, we obtain a leader election protocol whose time and space can be smoothly traded off between $O(\log^2 n)$ to $O(\log n)$ time and $O(\log\log n)$ to $O(\log n)$ bits.

3.1 Introduction

Leader Election, which is a fundamental problem in distributed computing, is the process of designating a single agent as the coordinator of some task distributed among several nodes. The nodes communicate among themselves in order to decide which of them will get into the *leader* state. Similarly to the previous chapter, we make an extensive use of epidemics, and we construct an algorithm that solves the *Leader Election* problem.

Optimal algorithms regarding the time complexity of fundamental tasks in distributed networks, for example leader election and majority, is the key for many distributed problems. The existence of a unique leader agent is a key requirement for many population protocols [9] and generally in distributed computing. Therefore, having a fast protocol that elects a unique leader is of high significance as it can lead to simpler and more efficient protocols. There are many solutions to the problem of leader election, such as in networks with nodes having distinct labels or anonymous networks [3, 6, 16, 73, 79].

Although the availability of an initial leader does not increase the computational power of standard population protocols (in contrast, it does in some settings where faults can occur [58]), still it may allow faster computation. Specifically, the fastest known population protocols for semilinear predicates without a leader take as long as linear parallel time to converge ($\Theta(n)$). On the other hand, when the process is coordinated by a unique leader, it is known that any semilinear predicate can be stably computed with polylogarithmic expected convergence time ($O(\log^5 n)$) [10].

3.1.1 Related work

For several years, the best known algorithm for leader election in population protocols was the pairwise-elimination protocol of Angluin et al. [7], in which all nodes are leaders in state l initially and the only effective transition is $(l, l) \rightarrow (l, f)$. This protocol always stabilizes to a configuration with unique leader, but this takes on average linear time. Recently, Doty and Soloveichik [67] proved that not only this, but any standard population protocol (constant number of states) requires linear time to solve leader election. This immediately led the research community to look into ways of strengthening the population protocol model in order to enable the development of sub-linear time protocols for leader election and other problems (note that Belleville, Doty, and Soloveichik [23] recently showed that such linear time lower bounds hold for a larger family of problems and not just for leader election). Fortunately, in the same way that increasing the local space of agents led to a substantial increase of the class of computable predicates [34], it has started to become evident that it can also be exploited to substantially speed-up computations. Alistarh and Gelashvili [3] proposed the first sub-linear leader election protocol, which stabilizes in $O(\log^3 n)$ parallel time, assuming $O(\log^3 n)$ states at each agent. In a very nice work, Gasieniec and Stachowiak [79] designed a space optimal ($O(\log \log n)$ states) leader election protocol, which stabilizes in $O(\log^2 n)$ parallel time. They use the concept of phase clocks (introduced in [9] for population protocols), which is a synchronization and coordination tool in distributed computing. General characterizations, including upper and lower bounds, of the trade-offs between time and space in population protocols were achieved in [2]. In a very recent work, Berenbrink et al. [24] designed the first time and space optimal leader election protocol which uses $\Theta(\log \log n)$ states per agent, and elects a leader in $O(n \log n)$ interactions in expectation. Finally, in [28] the authors study the problem of leader election, starting from any possible initial configuration, and provide a *silent* self-stabilizing protocol that uses optimal $O(n)$ expected parallel time and space. Silent leader election requires $\Omega(n)$ expected parallel time, and it means that the agents' states eventually stop changing. Moreover, some papers [44, 119] have studied leader election in the mediated population protocol model.

3.1.2 Preliminaries

Similarly to Chapter 2, we here consider the Population Protocol model, as described in Section 1.1.2, and we also assume that an approximate population size is hard-coded into the agents memory and that the agents can access a random number of a predefined number of bits. However, our protocol can be simulated in the standard population protocol model without increasing the stabilization time and the number of states asymptotically. This can be achieved by utilizing the random scheduler as a source of randomness. In particular, in [2] the authors introduced a synthetic coin technique that allows the agents to extract randomness from the scheduler and simulate almost fair coin flips.

Leader election problem The problem of leader election in distributed computing is for each node eventually to decide whether it is a leader or not subject to only one node decides that it is the leader. An algorithm A solves the leader election problem if eventually the states of agents are divided into *leader* and *follower*, a leader remains elected and a follower can never become a leader. In every execution, exactly one agent becomes leader and the rest determine that they are not leaders. All agents start in the same initial state q and the output is $O = \{leader, follower\}$. A randomized algorithm R, where the agents have access to random bits, solves the leader election problem if eventually only one leader remains in the system w.h.p.

3.1.3 Contribution and roadmap for the chapter

In this chapter we employ the use of simple epidemics in order to provide an efficient solution to leader election in populations. Our goal is to get polylogarithmic parallel time and to also use small memory per agent. We assume an approximate knowledge of the size of the population (i.e., an estimate $n^\alpha \leq n_e \leq n^\beta$, where n is the population size and $\alpha < \beta$ are two constant numbers) and provide a protocol (parameterized by the size m of a counter for drawing local random numbers) that elects a unique leader w.h.p. in $O(\frac{\log^2 n}{\log m})$ parallel time, with number of bits $O(\max\{\log m, \log\log n\})$ per node.

In Section 3.2 we present our *Leader Election* protocol, giving in Section 3.2.1 an abstract description, the algorithm in Section 3.2.2, and in Section 3.2.3 we present its analysis. Finally, in Section 3.2.4 we combine our *Approximate Counting* protocol of Chapter 2 and our *Leader Election* protocol in order to provide a *size oblivious* protocol which elects a leader in $O(\frac{\log^2 n}{\log m})$ parallel time. We do not provide a formal analysis of this protocol, but simulation results suggest that it works as expected.

We have measured the stabilization time of this protocol for different population sizes and the results can be found in Section 3.3.

3.2 Leader Election with Approximate Knowledge of n

3.2.1 Abstract description

We assume that the nodes know *a constant factor upper bound on* $\log n$, $b \log n$, where n is the number of nodes and b is any big constant number.
In addition, all nodes store three variables; the round e, a random number r and a counter c and they are able to compute random numbers within a predefined range $[1, m]$. We define two types of states; the leaders (l) and the followers (f). Initially, all nodes are in state l, indicating that they are all potential leaders. The protocol operates in rounds and in every round, the leaders compete with each other trying to survive (i.e., do not become followers). The followers just copy the *tuple* (r, e) from the leaders and try to spread it throughout the population. During the first interaction of two l nodes, one of them becomes follower, a random number between 1 and m is being generated, the leader enters the first round and the follower copies the round e and the random number r from the leader to its local memory. The followers are only being used for information spreading purposes among the potential leaders and they cannot become leaders again. Throughout this chapter, n denotes the *population size* and m *the maximum number that nodes can generate.*

Information spreading. It has been shown that the epidemic spreading of information can accelerate the convergence time of a population protocol. In this chapter, we adopt this notion and we use the followers as the means of competition and communication among the potential leaders. All leaders try to spread their information (i.e., their round and random number) throughout the population, but w.h.p. all of them except one eventually become followers. We say that a node x wins during an interaction with another node if one of the following holds:

- Node x is in a bigger round e.
- Both nodes are in the same round, but node x has bigger random number r.

One or more leaders L are in the *dominant state*, if their tuple (r_1, e_1) wins every other tuple in the population. Such a tuple (r_1, e_1) is being spread as an epidemic throughout the population, independently of the other leaders' tuples (all leaders or followers with the tuple (r_1, e_1) always win their competitors). We also call leaders L the *dominant leaders.*

Transition to next round. After the first interaction, a leader l enters the first round. We can group all the other nodes that l can interact with into three disjoint sets.

- The first group contains the nodes that are in a bigger round or have a bigger random number, being in the same round as l. If the leader l interacts with such a node, it becomes follower.
- The second group contains the nodes that are in a smaller round or have a smaller random number, being in the same round as l. After an interaction with a node in this group, the other node becomes a follower (if not already a follower) and the leader increases its counter c by one.
- The third group contains the followers that have the same tuple (r, e) as l. After an interaction with a node in this group, l increases its counter c by one.

As long as the leader l survives (i.e., does not become a follower), it increases or resets its counter c, according to the transition function δ. When the counter c reaches $b \log n$ (recall that n^b is the upper bound on the population size), it resets it and round e is increased by one. The followers can never increase their round or generate random numbers.

Stabilization. The protocol that we present stabilizes, as the whole population will eventually reach in a final configuration of states. To achieve this, when the round of a leader l reaches $\lceil \frac{2b \log n - \log(b \log^2 n)}{\log m} \rceil$, l stops increasing its round e, unless it interacts with another leader. This rule guarantees the stabilization of our protocol.

3.2.2 The protocol

In this section, we present our *Leader Election* protocol. We use the notation $p_{r,e}$ to indicate that node p has the random number r and is in the round e. Also, we say that $(r_1, e_1) > (r_2, e_2)$, if the tuple (r_1, e_1) wins with the tuple (r_2, e_2). A tuple (r_1, e_1) wins with the tuple (r_2, e_2) if $e_1 > e_2$ or if they are in the same round ($e_1 = e_2$), it holds that $r_1 > r_2$. The transition function of the protocol is given in Protocol 4

3.2.3 Analysis

The leader election algorithm that we propose, elects a unique leader after $O(\frac{\log^2 n}{\log m})$ parallel time w.h.p.. To achieve this, the algorithm works in stages, called *epochs* throughout this section and the number of potential leaders decreases exponentially between the epochs. An epoch i starts when any leader enters the ith round ($e = i$) and ends when any leader enters the $(i + 1)$th round ($e = i + 1$). Here we do the exact analysis for $m = \log n$. This can be generalized to any m between a constant and n.

Protocol 4 Leader Election

$Q = \{l, f_{r,e}, l_{r,e,c}\} : r \in [1, m]$, $e \in [0, \lceil \frac{2b \log n - \log(b \log^2 n)}{\log m} \rceil]$, $c \in [0, c \log n]$
δ :

#First interaction between two nodes. One of them becomes follower and the other remains leader. The leader generates a random number r and enters the first round ($e = 1$).

1: $(l, l) \to (l_{r,1,1}, f_{r,1})$

#A leader in round 0 (that is, a node in state l) always loses (i.e., becomes a follower) against a node in a higher round.

2: $(f_{r,e}, l) \to (f_{r,e}, f_{r,e})$
3: $(l_{r,e,c}, l) \to (l_{r,e,c+1}, f_{r,e})$

#The winning node propagates its tuple. If a leader loses, it becomes follower.

4: $(f_{r,i}, f_{s,j}) \to (f_{k,l}, f_{k,l})$, **if** $(r, i) > (s, j)$ **then** $(k, l) = (r, i)$ **else** $(k, l) = (s, j)$
5: $(l_{r,i,c}, l_{s,j,c'}) \to (l_{k,l,c+1}, f_{k,l})$, **if** $(r, i) \geq (s, j)$ **then** $(k, l) = (r, i)$
6: **else** $(k, l) = (s, j)$
7: $(l_{r,i,c}, f_{s,j}) \to (f_{s,j}, f_{s,j})$, **if** $(s, j) > (r, i)$
8: $(l_{r,i,c}, f_{s,j}) \to (l_{r,i,c+1}, f_{r,i})$, **if** $(r, i) > (s, j)$
9: $(l_{r,e,c}, f_{r,e}) \to (l_{r,e,c+1}, f_{r,e})$

#When a leader increases its counter, the following code is being executed. It checks whether it has reached $c \log n$. If yes, it moves to the next round, generates a new random number and checks if it has reached the final round in order to terminate.

10: **if** $c = b \log n$ **then**
11: Increase round e by one;
12: Generate a new random number r between 1 and m;
13: Reset counter c to zero;
14: **if** $e = \lceil \frac{2b \log n - \log(b \log^2 n)}{\log m} \rceil$ **Stop increasing the round, unless**
15: **you interact with a leader**

Lemma 3.1. *During the execution of the protocol, at least one leader will always exist in the population.*

Proof. Assume an epoch e, in which only one leader l_1 with the tuple (r_1, e_1) exists in the population and the rest of the nodes have become followers. In order for l_1 to become follower, there should be a follower with a tuple (r_2, e_2), where $(r_2, e_2) > (r_1, e_1)$. But, while the followers can never increase their epoch or generate a new random number, that would imply that there exists another leader l_2 with the tuple (r_2, e_2), which is a contradiction. □

Lemma 3.2. *Assume an epoch e and k leaders with the dominant tuple (r, e) in this epoch. The expected parallel time to convergence of their epidemic in epoch e is $\Theta(\log n)$.*

Proof. Let the random variable X be the total number of interactions until all nodes have the dominant tuple (r, e). We divide the interactions of the protocol into rounds, where round i means that the epidemic has been spread to i nodes. Initially, $i = k$, that is, the k leaders are already infected by the epidemic, but we study the worst case where $k = 1$. Call an interaction a success if the epidemic spreads to a new node. Let also the random variables $X_i, 1 \leq i \leq n-1$, be the number of interactions in the ith round. Then, $X = \sum_{i=1}^{n-1} X_i$. The probability p_i of success at any interaction during the ith round is:

$$p_i = \frac{2i(n-i)}{n(n-1)}$$

where $i(n-i)$ are the effective interactions and $\frac{n(n-1)}{2}$ are all the possible interactions. By linearity of expectation we have:

$$\begin{aligned}
E[X] = E[\sum_{i=1}^{n-1} X_i] &= \sum_{i=1}^{n-1} E[X_i] = \sum_{i=1}^{n-1} \frac{1}{p_i} = \sum_{i=1}^{n-1} \frac{n(n-1)}{2i(n-i)} \\
&= \frac{n(n-1)}{2} \sum_{i=1}^{n-1} \frac{1}{i(n-i)} \\
&= \frac{n(n-1)}{2} \sum_{i=1}^{n-1} \frac{1}{n}(\frac{1}{i} + \frac{1}{n-i}) \\
&= \frac{(n-1)}{2} [\sum_{i=1}^{n-1} \frac{1}{i} + \sum_{i=1}^{n-1} \frac{1}{n-i}] \\
&= \frac{(n-1)}{2} 2H_{n-1} \\
&= (n-1)[\ln(n-1) + a_{n-1}] = \Theta(n \log n)
\end{aligned}$$

where H_n denotes the nth Harmonic number and $a_n := H_n - \log n, (n \in \mathbb{N})$ is a decreasing sequence and $0 < a_n < 1$ for all $n \in \mathbb{N}$ (*Euler-Mascheroni constant*). In terms of parallel time, it holds that $E[\frac{X}{n}] = \frac{E[X]}{n} = \Theta(\log n)$. □

Lemma 3.3. *When a leader's counter $c \geq b \log n$, its epidemic (r, e) is spread throughout the whole population w.h.p..*

Proof. Let the r.v. X be the total number of interactions until all nodes have been infected by the dominant tuple. By Lemma 3.2, the expected interactions until the epidemic spreads throughout the whole population is $\mu = (n-1)\ln(n-1) + \Theta(1)$.

We now make use of the following Chernoff Bounds:

$$\Pr[X \leq (1-\delta)\mu] \leq e^{-\frac{\mu\delta^2}{2}}, \delta \geq 0$$

$$\Pr[X \geq (1+\delta)\mu] \leq e^{-\frac{\mu\delta^2}{2+\delta}}, \delta \geq 0$$

For $\delta = 1/2$, it holds that

$$\Pr[X \leq (1-\delta)\mu] \leq e^{\frac{-\delta^2\mu}{2}} \leq e^{-\frac{(n-1)\ln(n-1)}{8}} \leq \left(\frac{1}{n-1}\right)^{(n-1)/8}$$

And for the upper bound, for $\delta \geq 0$

$$\Pr[X \geq (1+\delta)\mu] \leq e^{-\frac{\mu\delta^2}{2+\delta}} < e^{-\frac{\mu(\delta+2)(\delta-2)}{2+\delta}} = e^{-\mu(\delta-2)} = \left(\frac{1}{n-1}\right)^{(n-1)(\delta-2)}$$

Thus, the interactions per node under the uniform random scheduler until all nodes become infected are w.h.p. $\frac{(n-1)\ln(n-1)}{n} < \frac{n\ln n}{n} = \ln n$. Thus, after $b\log n$ interactions, where n^b is the population size estimation and b a large constant, there are no non-infected nodes w.h.p.. □

Theorem 3.4. *After $O(\frac{\log n}{\log m})$ epochs, there is a unique leader in the population w.h.p..*

Proof. Assume an epoch e, in which there are k leaders with the dominant tuple (r, e) and m is the biggest number that the leaders can generate. We shall argue that by the end of the next epoch $e+1$, approximately $\frac{k(m-1)}{m}$ leaders will have become followers and approximately $\frac{k}{m}$ leaders will have a new dominant tuple (r_2, e_2). Whenever the k leaders enter to the next epoch $e+1$, they generate a new random number between 1 and m. Let the random variable X_e be the number of leaders that have randomly generated the biggest number in epoch e. We view the possible values of the random choices as m bins and we investigate how many leaders shall go to each bin. Assume the sequence of the random numbers $C_i^e, 1 \leq i \leq k$ that the leaders generate in epoch e. Let the random variables X_i^e be independent Bernoulli trials such that, for $1 \leq i \leq k$, $\Pr[X_i^e = 1] = p_i$ and $\Pr[X_i^e = 0] = 1 - p_i$ and $X_e = \sum_{i=1}^{k} X_i^e$. The probability that a leader chooses randomly a number is

$$p_i = \frac{1}{m}$$

Then, the expected number of balls in each bin, thus in the bin with the highest index also (X_e) is

$$\mu = E(X_e) = E(\sum_{i=1}^{k} X_i^e) = \sum_{i=1}^{k} E(X_i^e) = \sum_{i=1}^{k} p_i = \sum_{i=1}^{k} \frac{1}{m} = \frac{k}{m}$$

Assume now inductively that $X_e \geq a\log^2 n$, where $a > 0$ and $m = \log n$. By the Chernoff bound and observing that $k \geq ma\log n \Rightarrow \frac{k}{m} \geq a\log n \Rightarrow \mu \geq a\log n$, we prove that the number of the new dominant leaders will be more than or equal to $\frac{k}{m}(1+\delta)$ with a negligible probability.

$$\Pr[X_e \geq (1+\delta)\mu] \leq e^{-\frac{\mu\delta^2}{3}} \leq e^{-\frac{a\log n\delta^2}{3}} = n^{-\frac{a\delta^2}{3}} = n^{-\phi}$$

For $a \geq \frac{9}{\delta^2}$ it holds that $\Pr[X_e \geq (1+\delta)\mu] \leq n^{-3}$. Consequently, if we had X_e leaders in epoch e, we now shall have no more than $X_{e+1} \leq (1+\delta)\frac{X_e}{m}$ leaders in epoch $e+1$ with probability $\Pr[X_{e+1} \leq (1+\delta)\frac{X_e}{m}] \geq 1 - \frac{1}{n^3}$.
We can now assume that the expected number of leaders between the epochs can be described by the following recursive function.

$$G_e = \begin{cases} \frac{G_{e-1}}{m}, & i \geq 1 \\ n, & i = 0 \end{cases} \tag{3.1}$$

where $G_e = (1+\delta)X_e$. Then,

$$G_e = \frac{G_{e-1}}{m} = \frac{G_{e-2}}{m^2} = ... = \frac{n}{m^e}$$

The number of the expected epochs until at most $a\log^2 n$ leaders remain in the population is

$$G_t = a\log^2 n \Rightarrow \frac{G_{t-1}}{m} = a\log^2 n \Rightarrow \frac{G_{t-2}}{m^2} = a\log^2 n \Rightarrow ... \Rightarrow \frac{n}{m^t} = a\log^2 n \Rightarrow$$
$$m^t = \frac{n}{a\log^2 n} \Rightarrow \log_m(m^t) = \log_m(\frac{n}{a\log^2 n}) \Rightarrow t = \log_m n - \log_m(a\log^2 n) \Rightarrow$$
$$t = \frac{\log n - \log(a\log^2 n)}{\log m} \Rightarrow t = \frac{\log n - \log(a\log^2 n)}{\log\log n}$$

Let $E(e)$, be the event that in epoch e, there are at most G_e dominant leaders. We consider a success when $(E(e) \mid E(1) \cap E(2) \cap \ldots \cap E(e-1))$ occurs until we have at most $\log n$ leaders. By taking the union bound, the probability to fail after

$t = \frac{\log n - \log(a\log^2 n)}{\log\log n}$ epochs is given by

$$\Pr(\text{fail after } t \text{ epochs}) \leq \sum_{i=0}^{t} \Pr[\text{fail in epoch i} \,|\, \text{success until (i-1)th epoch}]$$

$$\leq \sum_{i=0}^{t} \frac{1}{n^{\frac{a\delta^2}{3}}} = \sum_{i=0}^{t} \frac{1}{n^{\phi}} = \frac{\frac{\log n - \log(a\log^2 n)}{\log\log n}}{n^{\phi}} \leq \frac{1}{n^{\phi-1}} \leq \frac{1}{n^2}$$

Corollary 3.5. *After* $t = \frac{\log n - \log(a\log^2 n)}{\log\log n}$ *epochs, the are at most* $a\log^2 n$ *leaders w.h.p..*

We argue that the number of leaders can be reduced from $a\log^2 n$ to $a\log n$ in one round w.h.p.. The expected value of dominant leaders is now $E[X_{t+1}] = a\log n$, thus, by the Chernoff Bound it holds that $\Pr[X_{t+1} \geq (1+\delta)\mu] \leq e^{-\frac{a\log n\delta^2}{3}}$, and for $a \geq \frac{9}{\delta^2}$, $\Pr[X_{t+1} \geq (1+\delta)\mu] \leq n^{-3}$.

Assume w.l.o.g. that $m = a\log n$ and according to the previous analysis, there exist $k = a\log n$ leaders after $t' = \frac{\log n - \log(a\log^2 n)}{\log\log n} + 1$ epochs. The expected value of $X_{t'+1}$ is now $\mu = E[X_{t'+1}] = 1$. Thus, by the Markov Inequality, the probability that the number of the dominant leaders in the next epoch are at least 2 is

$$P(X_{t'+1} \geq 2) \leq \frac{E[X_{t'+1}]}{2} = \frac{1}{2}$$

The probability that after $\log_m n$ epochs, there is no unique leader in the population is

$$P[\text{at least 2 leaders exist after } \log_m n \text{ epochs}] \leq (\frac{1}{2})^{\log_m n} = \frac{1}{2^{\log_m n}}$$

The total number of epochs until there exists a unique leader in the population is w.h.p. $\frac{2\log n - \log(a\log^2 n)}{\log m} + 1 = O(\frac{\log n}{\log m})$. □

Theorem 3.6. *Our* Leader Election *protocol elects a unique leader in* $O(\frac{\log^2 n}{\log\log n})$ *parallel time w.h.p., and each agent uses at most* $O(\max\{\log m, \log\log n\})$ *bits of memory.*

Proof. There are initially n leaders in the population. During an epoch e, by Lemma 3.2 the dominant tuple spreads throughout the population in $\Theta(\log n)$ parallel time, by Lemma 3.3, w.h.p. no (dominant) leader can enter to the next epoch if their epidemic has not been spread throughout the whole population before and by Theorem 3.4, there will exist a unique leader after $O(\frac{\log n}{\log m})$ epochs w.h.p., thus, for $m = b\log n$

the overall parallel time is $O(\frac{\log^2 n}{\log\log n})$. Finally, by Lemma 3.1, this unique leader can never become follower and according to the transition function in Protocol 4, a follower can never become leader again.
The rule which says the leaders stop increasing their rounds if $e >= \frac{2b\log n - \log(b\log^2 n)}{\log m}$, unless they interact with another leader, implies that the population stabilizes in $O(\frac{\log^2 n}{\log\log n})$ parallel time w.h.p. and when e exceeds this value, there will exist only one leader in the population and eventually, our protocol always elects a unique leader.

Each agent stores four values; $b\log n$, the round e, the random number r, and the counter c. The round is bounded by the maximum epoch that can be reached, and is $O(\frac{\log n}{\log m})$. The random number takes values in $[1, m]$, and the counter in $[0, b\log n]$. In addition, each agent stores an additional bit which indicates whether it is a leader or a follower. Thus, the total number of states is $\frac{2mb^2\log^3 n}{\log m}$, or $3\log\log n + \log m - \log\log m + 2\log 2b$ bits. If m is a constant number, the space complexity is $O(\log\log n)$ bits, while for $m = \Omega(\log n)$, it is $O(\log m)$ bits. □

Remark 3.7. By adjusting m to be any number between a constant and n and conducting a very similar analysis we may obtain a single leader election protocol whose time and space can be smoothly traded off between $O(\log^2 n)$ to $O(\log n)$ time and $O(\log\log n)$ to $O(\log n)$ bits.

3.2.4 Dropping the assumption of knowing $\log n$

Call a population protocol *size-oblivious* if its transition function does not depend on the population size. Our leader election protocol requires a rough estimate on the size of the population in order to elect a leader in polylogarithmic time, while our approximate counting protocol requires a unique leader who initiates the epidemic process and then gives an upper bound on the population size. In this section, we combine our Approximate Counting and Leader Election protocols in order to construct a size-oblivious protocol that can be executed in any uniform model of population protocols. We conjecture that it elects a unique leader in $O(\frac{\log^2 n}{\log m})$ parallel time, however we do not provide a proof of correctness.

To combine our protocols, in our new *Leader Election* algorithm, the nodes instead of using the c counter, as described in Section 3.2.1, they use two counters c_q and c_a. The only difference between this protocol and our leader election protocol of Section 3.2, is that we do not use an upper bound on $\log n$, but instead, the nodes perform the same experiment as in our *Approximate Counting* protocol in order to approximate it. Again, we call followers of a leader l the nodes that have the same tuple (r, e) as l, where r is a random number, and e the round of l. As in our *Approximate Counting*

protocol, the first counter is used in order to count the non-followers and the latter to count the followers. All nodes start as potential leaders in state l, and compete each other in order to survive (i.e., remain leaders).

Initially, $c_q = 1$ and $c_a = 0$. Let l be a leader with the tuple (r_1, e_1). As in Section 3.2, a tuple (r_1, e_1) is bigger that the tuple (r_2, e_2) if $r_1 > r_2$ or if $r_1 = r_2$ and $e_1 > e_2$. We can group all the other nodes that l can interact with into three independent sets.

- $(r_1, e_1) > (r_2, e_2)$. l increases its c_q counter by one.
- $(r_1, e_1) = (r_2, e_2)$. l increases its c_a counter by one.
- $(r_1, e_1) < (r_2, e_2)$. l becomes follower and resets its counters to zero.

When $c_q = c_a$ holds, l increases its round e_1 by one and resets c_q to one and c_a to zero. This process simulates the behavior of our *Approximate Counting* protocol, meaning that when $c_q = c_a$ holds, the epidemic of the dominant leaders is spread throughout the whole population w.h.p.. Regarding the termination condition, where the value $\log n$ is needed, the nodes store a variable s which contains the average value of c_q, when $c_q = c_a$. To this end, whenever a leader enters from round e_1 to $e_1 + 1$, it updates the value of s as follows:

$$s = \frac{s(e_1 - 1) + c_q}{e_1} \tag{3.2}$$

where s is initially zero. When $e_1 = \lceil \frac{as}{\log m} \rceil$ holds ($a \geq 1$ is a small constant number), the leader stops increasing its round and the population w.h.p. stabilizes in a configuration with a unique leader. Even though we do not provide a formal analysis of this protocol, in Section 3.3 we provide experiments that confirm this behavior, that is, after $O(\frac{\log^2 n}{\log m})$ parallel time there exists a single leader u in the population, and the value s of u is polynomially close to $\log n$.

Regarding the space complexity, as in Protocol 4, the nodes store the round $e = O(\frac{\log n}{\log m})$, and the random number $r = O(m)$. In addition, instead of one counter c, it stores two counters c_q and c_a that with high probability are $c_q = c_a = O(\log n)$ (Theorem 2.4). Finally, the variable s holds the average of c_q when reaching the end of a round ($c_q = c_a$), thus $s = O(\log n)$. In total, the number of states is $O(\frac{m \log^4 n}{\log m})$, meaning that if m is a constant number, the space complexity is $O(\log \log n)$ bits, while for $m = \Omega(\log n)$, it is $O(\log m)$ bits.

3.3 Experiments

We have also measured the stabilization time of our *Leader Election* protocol for different network sizes. We have executed it 100 times for each population size n, where $n = 2^i$ and $i = [4, 14]$. The results of our experiments, as shown in Figure 3.1, support our analysis and confirm its logarithmic behavior. In these experiments, the maximum number that the nodes could generate was $m = 100$. Finally, all executions elected a unique leader in $a\frac{\log^2 n}{\log m}$ parallel time, for some small constant a.

Regarding our protocol for leader election with no knowledge of $\log n$, the results are shown in Figures 3.2 and 3.3. All executions elected a unique leader after $a\frac{\log^2 n}{\log m}$ parallel time, as shown in Figure 3.2. Finally, as shown in Figure 3.3, the unique leader holds a constant factor upper bound on $\log n$ after $a\frac{\log^2 n}{\log m}$ parallel time.

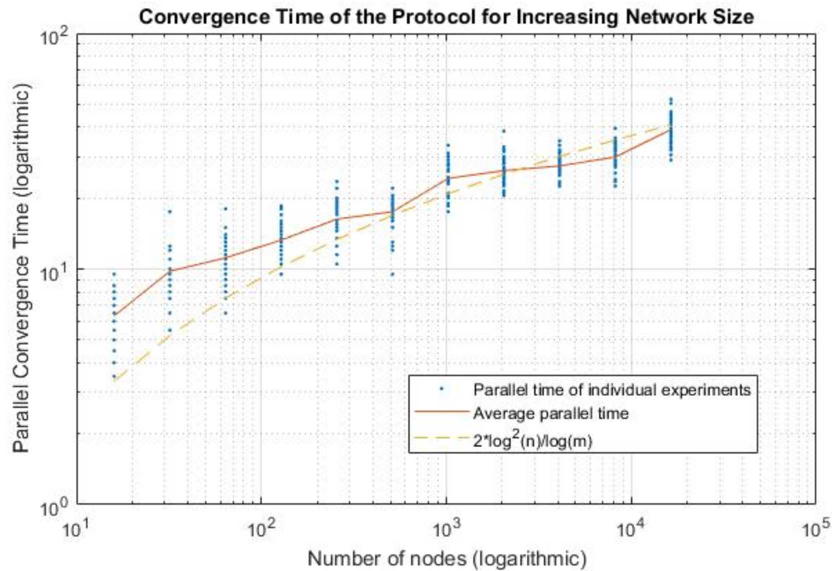


FIGURE 3.1: *Leader Election* with approximate knowledge of n. Both axes are logarithmic. The dots represent the results of individual experiments and the line represents the average values for each network size.

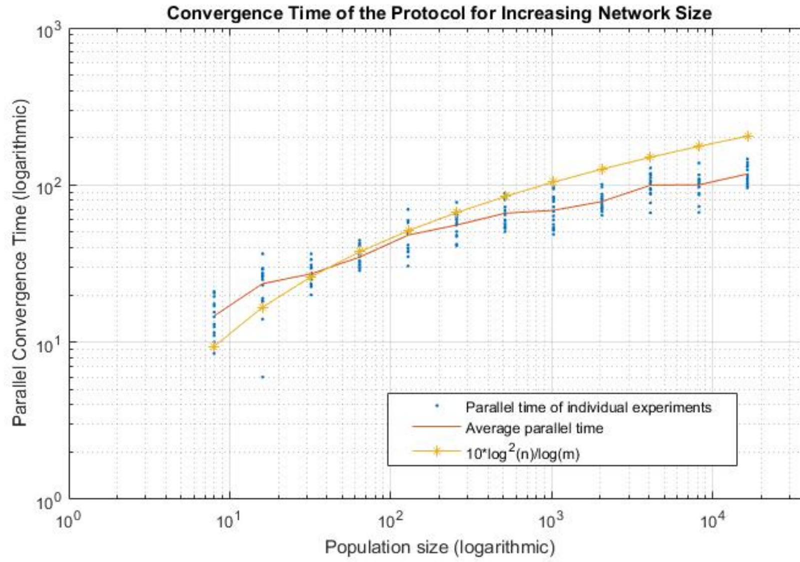


FIGURE 3.2: Convergence time of the composition of *Leader Election* and *Approximate Counting* protocols.

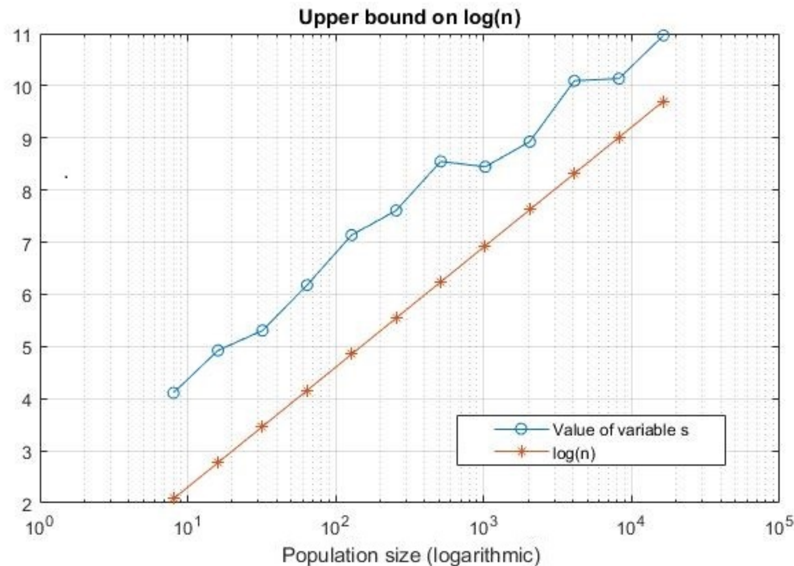


FIGURE 3.3: Composition of *Leader Election* and *Approximate Counting* protocols. Upper bounds and actual sizes of $\log n$.

3.4 Open Problems

In our leader election protocol of Section 3, when two nodes interact with each other, the amount of data which is transferred is $\Omega(\max\{\log\log n, \log m\})$ bits. In certain applications of population protocols, the processes are not able to transfer arbitrarily large amounts of data during an interaction. Can we design a polylogarithmic time

population protocol for the problem of leader election that satisfies this requirement? In addition, in Section 3.2.4 we describe how our Approximate Counting, presented in Section 2, and Leader Election protocols can be combined into a single protocol that can be executed in any uniform model of population protocols, and elects a unique leader in $\Omega(\frac{\log^2 n}{\log m})$ parallel time. We provide some evidence that this is indeed the case in Section 3.3, and we leave the proof of its correctness as an open problem.

Part II

Structure Formation Problems

Chapter 4

Fault Tolerant Network Constructors

In this chapter, we consider adversarial crash faults of nodes in the network constructors model [113]. We first show that, without further assumptions, the class of graph languages that can be (stably) constructed under crash faults is non-empty but small. In particular, if an unbounded number of crash faults may occur, we prove that (i) the only constructible graph language is that of spanning cliques and (ii) a strong impossibility result holds even if the size of the graphs that the protocol outputs in populations of size n needs to grow with n (the remaining nodes being *waste*). When there is a finite upper bound f on the number of faults, we show that it is impossible to construct any *non-hereditary* graph language and leave as an interesting open problem the *hereditary case*. On the positive side, by relaxing our requirements we prove that: (i) permitting linear waste enables to construct on $n/(2f) - f$ nodes, any graph language that is constructible in the fault-free case, (ii) *partial constructibility* (i.e., not having to generate all graphs in the language) allows the construction of a large class of graph languages. We then extend the original model with a minimal form of *fault notifications*. Our main result here is a *fault-tolerant universal constructor*: We develop a fault-tolerant protocol for *spanning line* and use it to simulate a linear-space Turing Machine M. This allows a fault-tolerant construction of any graph accepted by M in linear space, with waste $min\{n/2 + f(n),\ n\}$, where $f(n)$ is the number of faults in the execution. We then prove that increasing the permissible waste to $min\{2n/3 + f(n),\ n\}$ allows the construction of graphs accepted by an $O(n^2)$-space Turing Machine, which is asymptotically the maximum simulation space that we can hope for in this model. Finally, we show that logarithmic local memories can be exploited for a no-waste fault-tolerant simulation of any such protocol.

4.1 Introduction

4.1.1 Related work

In this chapter, we address the issue of the dynamic formation of graphs under faults. We do this in a minimal setting, that is, a population of agents running *Population Protocols* that can additionally activate/deactivate links when they meet. This model, called *Network Constructors*, was introduced in [113], and is based on the *Population Protocol* (PP) model [7, 11] and the *Mediated Population Protocol* (MPP) model [107]. We are interested in answering questions like the following: If one or more faults can affect the formation process, can we always re-stabilize to a correct graph, and if not, what is the class of graph languages for which there exists a fault-tolerant protocol? What are the additional minimal assumptions that we need to make in order to find fault-tolerant protocols for a bigger class of languages?

Population Protocols run on networks that consist of computational entities called *agents* (or *nodes*). One of the challenging characteristics is that the agents have no control over the schedule of interactions with each other. In a population of n agents, repeatedly a pair of agents is chosen to interact. During an interaction their states are updated based on their previous states. In general, the interactions are scheduled by a *fair scheduler*. When the execution time of a protocol needs to be examined, a typical fair scheduler is the one that selects interactions uniformly at random.

In this chapter, we examine the setting where *adversarial crash faults* may occur, and we address the question of which families of graph languages can be stably formed. Fault tolerance must deal with the graph topology, thus, previous results on self-stabilizing PPs [12, 19, 38] and MPPs [119] do not apply here. In particular, if Π is a protocol that constructs a graph language L, then a crash fault may result to a configuration C such that no execution of Π starting from C stabilizes to a graph in L. This means that when faults occur, the population must perform some computations so as to reach a configuration where all executions of Π will again stabilize to a graph in L. The problem is then to study self-stabilizing protocols under crash faults. Here, adversarial crash faults mean that an adversary knows the rules of the protocol and can select some node to be removed from the population at any time. For simplicity, we assume that the faults can only happen sequentially. This means that in every step at most one fault may occur, as opposed to the case where many faults can occur during each step. The cases of sequential and parallel occurrence of faults are equivalent to each other in the Network Constructors model w.l.o.g., but not in the extended version of this model (which allows fault notifications) that we consider later.

A main difference between our work and traditional self-stabilization approaches is that the nodes are supplied with constant local memory, while in principle they can

form linear (in the population size) number of connections per node. Existing self-stabilization approaches that are based on restarting techniques cannot be directly applied here [60, 61], as the nodes cannot distinguish whether they still have some activated connections with the remaining nodes, after a fault has occurred. This difficulty is the reason why it is not sufficient to just reset the state of a node in case of a fault. In addition, in contrast to previous self-stabilizing approaches [69, 86] that are based on *shared memory* models, two adjacent nodes can only store 1 bit of memory in the edge joining them, which denotes the existence or not of a connection between them.

Angluin *et al.* [12] incorporated the notion of self-stabilization into the population protocol model, giving self-stabilizing protocols for some fundamental tasks such as token passing and leader election. They focused on the goal of stably maintaining some property such as having a unique leader or a legal coloring of the communication graph.

Delporte-Gallet *et al.* [52] studied the issue of correctly computing functions on the node inputs in the Population Protocol model [7], in the presence of crash faults and transient faults that can corrupt the states of the nodes. They construct a transformation which makes any protocol that works in the failure-free setting, tolerant in the presence of such failures, as long as modifying a small number of inputs does not change the output. In the context of fault tolerance, [58] uses a leader to make any protocol tolerant to omission failures (i.e., failure by an agent to read its partner's state during an interaction). In [73], Fischer and Jiang introduced the $\Omega?$ detectors in order to solve leader election under crash or transient faults. An eventual leader detector $\Omega?$ is an oracle that eventually detects the presence or absence of a leader in the population.

Guerraoui and Ruppert [87] introduced an interesting model, called *Community Protocol*, which extends the Population Protocol model with unique identifiers and enough memory to store a constant number of other agents' identifiers. They show that this model can solve any decision problem in NSPACE($n \log n$) while tolerating a constant number of Byzantine failures.

Peleg [130] studies logical structures, constructed over static graphs, that need to satisfy the same property on the resulting structure after node or edge failures. He distinguishes between the stronger type of fault-tolerance obtained for geometric graphs (termed *rigid fault-tolerance*) and the more flexible type required for handling general graphs (termed *competitive fault-tolerance*). It differs from our work, as we address the problem of constructing such structures over dynamic graphs.

4.1.2 Contribution and roadmap for the chapter

The goal of any Network Constructor (NET) protocol is to stabilize to a graph that belongs to (or satisfies) some graph language L, starting from an initial configuration where all nodes are in the same state and all connections are disabled. In [113], only the fault-free case was considered. In this chapter, we formally define the model that extends NETs allowing crash failures, and we examine protocols in the presence of such faults. Whenever a node crashes, it is removed from the population, along with all its activated edges. This leaves the remaining population in a state where some actions may need to be taken by the protocol in order to eventually stabilize to a correct network.

We first study the constructive power of the original NET model in the presence of crash faults. We show that the class of graph languages that is in principle constructible is non-empty but very small: for a potentially unbounded number of faults, we show that the only stably constructible language is the *Spanning Clique*. We also prove a strong impossibility result, which holds even if the size of graphs that the protocol outputs in populations of size n needs to grow with n (i.e., $\omega(1)$), and the remaining nodes being *waste*. For a bounded number of faults, we show that any non-hereditary graph language is impossible to be constructed. However, we show that by relaxing our requirements we can extend the class of constructible graph languages. In particular, permitting linear waste enables to construct on $n/(2f)-1$ nodes where f is a finite upper bound on the number of faults, any graph language that is constructible in a failure-free setting. Alternatively, by allowing our protocols to generate only a subset of all graphs in the language (called *partial constructibility*), a large class of graph languages becomes constructible (see Section 4.3).

In light of the impossibilities in the Network Constructors model, we introduce the minimal additional assumption of *fault notifications*. This is essentially a failure detector ([32, 33]) that provides information about crash fault events in some nodes of the network. In [33] the failure detector $\Diamond S$ eventually outputs a set of crashed process identities at each process of the network. In our work, after a fault on some node u occurs, all nodes that maintain an active edge with u at that time (if any) are notified. If there are no such nodes, an arbitrary node in the population is notified. In that way, we guarantee that at least one node in the population will sense the removal of u. Nevertheless, some of our constructions work without notifications in the case of a crash fault on an isolated node (Section 4.4).

We obtain two fault-tolerant universal constructors. One of the main technical tools that we use in them, is a fault-tolerant construction of a stable path topology (i.e., a line). We show that this topology is capable of simulating a Turing Machine (abbreviated “TM” throughout this chapter), and, in the event of a fault, is capable

of always reinitializing its state correctly (Section 4.4.2). Our protocols use a subset of the population (called *waste*) in order to construct there a TM, while the graph which belongs to the required language L is constructed in the rest of the population (called *useful space*). Throughout this chapter, we call waste all nodes that do not belong to the constructed graph $G \in L$ after stabilization, and remain either isolated nodes or part of a component such as the TM. The idea is based on [113], where they show several universality results by constructing on k nodes of the population a network G_1 capable of simulating a TM, and then repeatedly drawing a random network G_2 on the remaining $n - k$ nodes. The idea is to execute on G_1 the TM which decides the language L with the input network G_2. If the TM accepts, it outputs G_2, otherwise the TM constructs a new random graph.

This allows a fault-tolerant construction of any graph accepted by a TM in linear space, with waste $min\{n/2+f(n), n\}$, where $f(n)$ is the number of faults in the execution. We finally prove that increasing the permissible waste to $min\{2n/3 + f(n),\ n\}$ allows the construction of graphs accepted by an $O(n^2)$-space Turing Machine, which is asymptotically the maximum simulation space that we can hope for in this model.

In order to give fault-tolerant protocols without waste, we design a protocol that can be composed in parallel with any protocol in order to make it fault-tolerant. The idea is to restart the protocol whenever a crash failure occurs. We provide a protocol Π' such that, given any network constructor Π with notifications, Π' is a fault-tolerant version of Π without waste. We show that restarting is impossible with constant local memory, if the nodes may form a linear (in the population size) number of connections; hence, the required memory per node in this protocol is $O(\log n)$ bits. Table 4.1 summarizes all results proved in this chapter.

Finally, in Section 4.5 we conclude and discuss further research directions opened by our work.

4.2 Model and Definitions

4.2.1 Network constructors with crash failures

In Section 1.2.2 we formally define the Network Constructor model. In this chapter, we present a version of this model that allows *adversarial crash failures.* A crash (or *halting*) failure causes an agent to cease functioning and play no further role in the execution. This means that all the adjacent edges of $F(u) \in G_U$, where $F(u)$ is the node of the underlying graph that u is mapped to, are removed from E_U, and, at the same time, all the adjacent edges of $u \in G_O$ become inactive (i.e., removed from the configuration). Finally, we consider G_U to be the *complete undirected interaction graph.*

Constructible languages		
Without notifications		With notifications
Unbounded faults (Section 4.3.1)	Bounded faults (Section 4.3.2)	Unbounded faults
Only Spanning Clique	Non-hereditary impossibility	Fault-tolerant protocols: Spanning Star, Cycle Cover, Spanning Line (Section 4.4.1)
Strong impossibility even with linear waste	A representation of any finite graph (partial constructibility)	Universal Fault-tolerant Constructors (with waste) (Section 4.4.2)
	Any constructible graph language with linear waste	Universal Fault-tolerant Restart (without waste) (Section 4.4.3)

Table 4.1: Summary of our results.

The execution of a protocol proceeds in discrete steps. In every step, an edge $e \in E_U$ between two nodes $F(u)$ and $F(v)$ is selected by an *adversary scheduler*, subject to some *fairness* guarantee. The corresponding nodes u and v interact with each other and update their states and the state of the edge $uv \in G_O$ between them, according to a joint transition function δ. If two nodes in states q_u and q_v with the edge joining them in state q_{uv} encounter each other, they can change into states q'_u, q'_v and q'_{uv}, where $(q'_u, q'_v, q'_{uv}) \in \delta(q_u, q_v, q_{uv})$. In the original model, G_U is the complete directed graph, which means that during an interaction, the interacting nodes have distinct roles. In our protocols, we consider the following constraint that is imposed by the fact that the edges of the interaction graph are undirected. In particular, $\delta(q_u, q_v, q_{uv}) = (a, b, c)$ implies $\delta(q_v, q_u, q_{vu}) = (b, a, c)$, for any $q_u, q_v \in Q$.

A configuration is a mapping $C : V_I \cup E_I \to Q \cup \{0, 1\}$ specifying the state of each node and each edge of the interaction graph. An execution of the protocol on input I is a finite or infinite sequence of configurations, $C_0, C_1, C_2, \ldots$, each of which is a set of states drawn from $Q \cup \{0, 1\}$. In the initial configuration C_0, all nodes are in state q_0 and all edges are inactive. Let q_u and q_v be the states of the nodes u and v, and q_{uv} denote the state of the edge joining them. A configuration C_k is obtained from C_{k-1} by one of the following types of transitions:

1. **Ordinary transition:** $C_k = (C_{k-1} - \{q_u, q_v, q_{uv}\}) \cup \{q'_u, q'_v, q'_{uv}\}$ where $\{q_u, q_v, q_{uv}\} \subseteq C_{k-1}$ and $(q'_u, q'_v, q'_{uv}) \in \delta(q_u, q_v, q_{uv})$.

2. **Crash failure:** $C_k = C_{k-1} - \{q_u\} - \{q_{uv} : uv \in E_I\}$ where $\{q_u, q_{uv}\} \subseteq C_{k-1}$.

We say that C' is *reachable from* C and write $C \rightsquigarrow C'$, if there is a sequence of configurations $C = C_0, C_1, \ldots, C_t = C'$, such that $C_i \to C_{i+1}$ for all i, $0 \leqslant i < t$. The fairness condition that we impose on the scheduler is quite simple to state. Essentially, we do not allow the scheduler to avoid a possible step forever. More formally, if C is a configuration that appears infinitely often in an execution, and $C \to C'$, then C' must also appear infinitely often in the execution. Equivalently, we require that any configuration that is always reachable is eventually reached.

We define the output of a configuration C as the graph $G(C) = (V, E)$ where $V = \{u \in V_O : C(u) \in Q_{out}\}$ and $E = \{uv : u, v \in V,\ u \neq v, \text{ and } C(uv) = 1\}$. If there exists some step $t \geq 0$ such that $G_O(C_i) = G$ for all $i \geq t$, we say that the output of an execution $C_0, C_1, \ldots$ *stabilizes* (or *converges*) to graph G, every configuration C_i, for $i \geq t$, is called *output-stable*, and t is called the *running time* under our scheduler. We say that a protocol Π stabilizes eventually to a graph G of *type* L if and only if after a finite number of pairwise interactions, the graph defined by 'on' edges does not change and belongs to the graph language L.

4.2.2 Definition of terms

In this chapter, unless otherwise stated, a graph language L is an infinite set of graphs satisfying the following properties:

1. (*No gaps*): For all $n \geq$ c, where $c \geq 2$ is a finite integer, $\exists G \in L$ of order n.

2. (*No Isolated Nodes*): $\forall G \in L$ and $\forall u \in V(G)$, it holds that $d(u) \geq 1$ (where $d(u)$ is the degree of u).

Even though graph languages are not allowed to contain isolated nodes, there are cases in which a protocol might be allowed to output one or more isolated nodes. In particular, if a protocol Π constructing L is allowed a waste of at most w, then whenever Π is executed on n nodes, it must output a graph $G \in L$ of order $|V(G)| \geq n - w$, leaving at most w nodes in one or more separate components (could be all isolated).

Definition 4.1. We say that a protocol Π *constructs* a graph language L if: (i) every execution of Π on n nodes stabilizes to a graph $G \in L$ s.t. $|V(G)| = n$ and (ii) $\forall G \in L$ there is an execution of Π on $|V(G)|$ nodes that stabilizes to G.

Definition 4.2. We say that a protocol Π *partially constructs* a graph language L, if: (i) from Definition 4.1 holds, and $\exists G \in L$ s.t. no execution of Π on $|V(G)|$ nodes stabilizes to G.

Definition 4.3 (Fault-tolerant protocol)**.** Let Π be a NET protocol that, in a failure-free setting, constructs a graph $G \in L$. Π is called *f-fault-tolerant* if for any population size n any execution of Π constructs a graph $G \in L$, where $|V(G)| \geq n - f$, and $f < n$ is an upper bound on the number of faults. We also call Π *fault-tolerant* if the same holds for any number $f \leq n - 2$ of faults.

Definition 4.4 (Waste and useful space)**.** We say that a protocol Π *constructs* a graph language L with *waste w* if: (i) every execution of Π on n nodes stabilizes to a graph $G \in L$ s.t. $n - w \leq |V(G)| \leq n$ and (ii) $\forall G \in L$ there is an execution of Π on n nodes s.t. $|V(G)| \leq n \leq |V(G)| + w$ that stabilizes to G. This implies that the waste includes all crashed nodes and any auxiliary nodes required by Π to construct G. Finally, we call $|V(G)|$ the *useful space.*

Definition 4.5 (Constructible language)**.** A graph language L is called *constructible (partially constructible)* if there is a protocol that constructs (partially constructs) it. Similarly, we call L *constructible under f faults*, if there is an f-fault-tolerant protocol that constructs L, where f is an upper bound on the maximum number of faults during an execution.

Definition 4.6 (Critical node)**.** Let G be a graph that belongs to a graph language L. Call u a *critical node* of G if by removing u and all its edges, the resulting graph $G' = G - \{u\} - \{uv : v \sim u\}$, does not belong to L. In other words, if there are no critical nodes in G, then any (induced) subgraph G' of G that can be obtained by removing nodes and all their edges, also belongs to L.

Definition 4.7 (Hereditary Language)**.** A graph language L is called *hereditary* if for any graph $G \in L$, every induced subgraph of G also belongs to L. In other words, there is no graph $G \in L$ with critical nodes.

This notion is known in the literature as *hereditary property* of a graph w.r.t. (with respect to) some graph language L. Observe that if there exists a graph G s.t. for any induced subgraph G' of G, $G' \in L$, does not imply that the same holds for any graph in L. Some examples of hereditary languages are "Bipartite graph", "Planar graph", "Forest of trees", "Clique", "Set of cliques", and "Maximum node degree $\leq \Delta$".

4.3 Network Constructors without Fault Notifications

In this section, we study the constructive power of the original NET model in the presence of bounded and unbounded crash faults when no form of notification is

available to the nodes. We start from the case in which the number of nodes that crash during an execution can be anything from 0 up to $n-2$ nodes. We are interested in characterizing the class of constructible graph languages. Observe that we cannot trivially conclude that the adversary can always leave us with just 2 nodes, only allowing our protocols to form a line of length 1. This is because our definition of constructible languages under faults takes into account all possible executions with f faults, for all values of $f \in \{0, 1, \ldots, n-2\}$. We show that in the case where the number of faults cannot be bounded by a constant number, the only language that is constructible is the $L_c = \{G : G \text{ is a spanning clique}\}$.

We then consider the setting where only a constant number of faults are allowed, and we show that no language L is constructible under a single fault, if L is not Hereditary. However, if we allow linear waste in the population, any language that is constructible without faults, becomes constructible under a constant number of faults.

Finally, we show a family of graph languages that is partially constructible (without waste in the population). The exact characterization of the class of partially constructible languages remains as an open problem.

4.3.1 Unbounded number of faults

We consider here the setting where the number of faults can be any number up to $n-2$. We prove that the only constructible graph language is *Spanning Clique* = $\{G : G \text{ is a spanning clique}\}$, i.e., a clique containing all nodes.

We first present a very simple protocol which constructs the language *Spanning Clique* and we show that it can tolerate any number of faults. Let *Clique* be the following protocol.

Protocol 5 Clique

$Q = \{b\}$
Initial state: b

δ :
1: $(b,\ b,\ 0) \rightarrow (b,\ b,\ 1)$

\# All transitions that do not appear have no effect.

Lemma 4.8. Clique *(Protocol 5) is a fault-tolerant protocol for* Spanning Clique.

Clearly, for any number $f < n$ of faults, where n is the population size, Protocol 5 constructs the language *Spanning Clique*.

By Lemma 4.8, we know that the language *Spanning Clique* is constructible under $n-2$ faults. To clarify, this means that for any execution of Protocol 5 on n nodes,

f of which crash ($f \in \{0, 1, \ldots, n-2\}$), Protocol 5 is guaranteed to stabilize to a clique of order $n - f$.

We will now prove that (due to the power of the adversary), no other graph language is constructible under unbounded faults.

Lemma 4.9. *Let* Π *be a protocol constructing a language* L *and* $G \in L$ *be a graph that* Π *outputs on* $|V(G)|$ *nodes. If* G *has an independent set* $S \subseteq V$*, s.t.* $|S| \geq 2$*, then there is an execution of* Π *on* n *nodes which stabilizes on* $|S|$ *isolated nodes (where* $|S| = n - f$ *and* f *is the number of faults in that execution).*

Proof. Consider an execution of Π that outputs G. By definition, there is a point in this execution after which no further edge updates can occur (no matter what the infinite execution suffix will be). Take any configuration C_{stable} after that point and consider its sub-configuration C_S induced by the independent set S. Observe that C_S encodes the state of each node $u \in S$ in that particular stable configuration C_{stable}. Denote also by Q_S the multiset of all states assigned by C_S to the nodes in S.

Every state in Q_S is reachable (in the sense that there exists an execution that produces it). For each $q \in Q_S$ consider the smallest population V_q in which there is some execution a_q of protocol Π that produces state q. Consider the population $V = \bigcup_{q \in Q_S} V_q$ (or equivalently of size $n = \sum_{q \in Q_S} |V_q|$).

For each V_q in population V we execute a_q until q is produced on some node u_q. After this, every $q \in Q_S$ is present in the population V. Then, the adversary crashes all nodes in $V_q \setminus \{u_q\}$ (i.e., only u_q remains alive in each V_q). This leaves the execution with a set of alive nodes equivalent in cardinality and configurations to the independent set S under C_S.

The above construction is a finite prefix of fair executions. For the sake of contradiction, assume that in any fair continuation of the above prefix, Π eventually stabilizes to a graph with no isolated nodes (as required by the fact that Π constructs a graph language L). Take one such continuation γ. As γ starts from a configuration in all respects equivalent to that of S under C_{stable}, it follows that γ can also be applied to C_{stable} and in particular on the independent set S starting from C_S. It follows that γ must have exactly the same effect as before, that is, eventually it will cause the activation of at least one edge between the nodes in S. But this violates the fact that C_{stable} is a stable configuration, therefore no edge could have been activated by Π in the continuation, implying that the continuation must have been an execution stabilizing on $|S|$ isolated nodes. □

Theorem 4.10. *Let* L *be any graph language such that* $L \neq$ *Spanning Clique. Then, there is no protocol that constructs* L *if an unbounded number of crash failures may occur.*

Proof. As $L \neq$ *Spanning Clique*, there exists $G \in L$ such that G is not complete (and by definition no $G' \in L$ has isolated nodes). Therefore G has an independent set S of size at least 2. If there exists a protocol Π that constructs L, then by Lemma 4.9 there must be an execution of Π which stabilizes on at least 2 isolated nodes. The latter is a stable output not in L, therefore a contradiction. □

Theorem 4.11. *If an unbounded number of faults may occur, the Spanning Clique is the only constructible language.*

Proof. Directly from Lemma 4.8 and Theorem 4.10. □

Theorem 4.12. *Let L be any graph language such that the graphs $G \in L$ have maximum independent sets whose size grows with $|V(G)|$ (i.e., $\omega(1)$). If the useful space of protocols is required to grow with n, then there is no protocol that constructs L in the unbounded-faults case.*

Proof. The proof is a direct application of Lemma 4.9. As the size of the maximum independent set of G grows with $|V(G)|$ in L, and the useful space is a non-constant function of n, it follows that, as n grows, the stable output-graph (on the useful space) has an independent set of size that grows with n (consider, for example, the leaves of binary trees of growing size as such a growing independent set). As any such stable independent set of size $g(n)$ implies that another execution has to stabilize to $g(n)$ isolated nodes, it follows that any protocol for L would produce infinitely many stable outputs of isolated nodes. The latter is contradicting the fact that the protocol constructs L. □

4.3.2 Bounded number of faults

The exact characterization established above, shows that under unbounded failures and without further assumptions, we cannot hope for non-trivial constructions. We now relax the power of the faults adversary, so that there is a *finite upper bound f* on the number of faults. In particular, for any $n \geq 1$, and fixing any such $0 \leq f \leq n$ in advance, it is guaranteed that for all executions of a protocol on n nodes, at most f nodes may fail during the execution. Then the class of constructible graph languages is naturally parameterized in f. We first show that non-hereditary languages are not constructible under a single fault.

Theorem 4.13. *If there exists a critical node in G, there is no 1-fault-tolerant NET protocol that stabilizes to it.*

Proof. Let Π be a NET protocol that constructs a graph language L, tolerating one crash failure. Consider an execution $\mathcal{E}$ and a sequence of configurations $C_0, C_1, \ldots$ of $\mathcal{E}$. Assume a time t that the output of $\mathcal{E}$ has stabilized to a graph $G \in L$ (i.e.,

$G(C_i) = G$, $\forall i \geq t$). Let u be a critical node in G. Assume that the scheduler removes u and all its edges (crash failure) at time $t' > t$, resulting to a graph $G' \notin L$. In order to fix the graph (i.e., re-stabilize to a graph $G'' \in L$), the protocol must change at some point t'' the configuration. This can only be the result of a state update on some node v. Now, call $\mathcal{E}'$ the execution that node u does not crash and, besides that, is the same as $\mathcal{E}$. Then, between t' and t'' the node v has the same interactions as in the previous case where node u crashed. This results to the same state update in v, since it cannot distinguish $\mathcal{E}$ from $\mathcal{E}'$. The fact that u either crashes or not, leads to the same result (i.e., v tries to fix the graph thinking that u has crashed). This means that if we are constantly trying to detect faults in order to deal with them, this would happen indefinitely and the protocol would never be stabilizing. Consider that the network has stabilized to G. At some point, because of the infinite execution, a node will surely but wrongly detect a crash failure. Thus, G has not really stabilized. □

By Definition 4.7 and Theorem 4.13 it follows that.

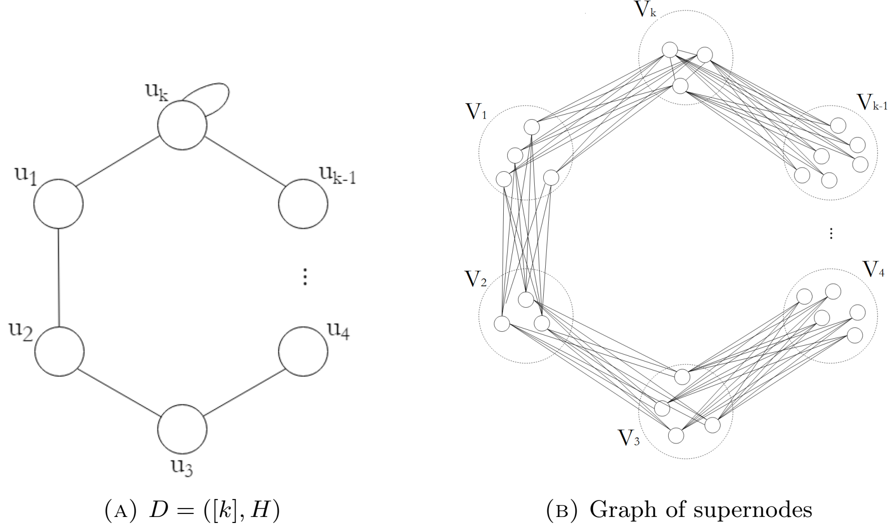
Corollary 4.14. *If a graph language L is non-hereditary, it is impossible to be constructed under a single fault.*

Note that this does not imply that any hereditary language is constructible under a constant number of faults. We leave this as an interesting open problem.

On the positive side we show that in the case of bounded number of faults, there is a non-trivial class of languages that is partially constructible. Consider the class of graph languages defined as follows. Any such language $L_{D,f}$ in the family is uniquely specified by a graph $D = ([k], H)$ and the finite upper bound $f < k$ on the number of faults. A graph $G = (V, E)$ belongs to $L_{D,f}$ iff there are k partitions $V_1, V_2, \ldots, V_k$ of V s.t. for all $1 \leq i, j \leq k$, $||V_i| - |V_j|| \leq f + 1$. In addition, E is constructed as follows. The graph $D = ([k], H)$, possibly containing self-loops, defines a neighboring relation between the k partitions. For every $(i, j) \in H$ (where possibly $i = j$), E contains all edges between partitions V_i and V_j, i.e., a complete bipartite graph between them (or a clique in case $i = j$). As no isolated nodes are allowed, every V_i must be fully connected to at least one V_j (possibly itself). In Figure 4.1, we present an example of a graph that belongs to $L_{D,f}$, where D defines a ring graph.

We first consider the case where $k = 2^{\epsilon}$, for some constant $\epsilon \in \mathbb{N}_0$, and we provide a protocol that divides the population into k partitions. The protocol works as follows: initially, all nodes are in state c_0 (we call this the partition 0). When two nodes in states c_i, where $i \geq 0$ interact with each other, they update their states to c_{2i+1} and c_{2i+2}, moving to partitions $2i + 1$ and $2i + 2$ respectively. Interactions between nodes in different c-states (c_i, c_j, where $i \neq j$) do not affect the configuration.

When $j = 2i + 1 \geq k - 1$ (or $j = 2i + 2 \geq k - 1$) for the first time, it means that the node has reached its final partition. It updates its state to P_m, where $m = j - k + 1$, thus, the final partitions are $\{P_0, P_1, \ldots, P_{k-1}\}$.



(A) $D = ([k], H)$ (B) Graph of supernodes

FIGURE 4.1: In 4.1a, D defines a ring of size k. In 4.1b, each node of D corresponds to a set of nodes (or supernode), while for each edge of D between two nodes u_i and u_j, all nodes of V_i are connected to all nodes of V_j and vice versa.

This process divides each partition into two partitions of equal size. However, in the case where the number of nodes is odd, a single node remains unmatched. For this reason, all nodes participate to the final formation of H regardless of whether they have reached their final partitions or not. There is a straightforward mapping of each internal partition to a distinct leaf of the binary tree, that is, each partition c_i behaves as if it were in partition P_i. In order to avoid false connections between the partitions, we also allow the nodes to disconnect from each other if they move to a different partition. This process guarantees that eventually all nodes end up in a single partition, and their connections are strictly described by H.

Lemma 4.15. *In the absence of faults, Protocol 6, divides the population into k partitions of at least $n/k - 1$ nodes each.*

Proof. Initially all nodes are in state c_0. When two c_0 nodes interact with each other, one of them becomes c_1 and the other one c_2. This means that all n nodes split into two partitions of equal size. No node can become c_0 again at any time during the execution. In addition, there is only one partition c_j that produces nodes of some other partition c_i, where i is either $2j + 1$ or $2j + 2$, and the size of them are half the size of c_j. This process can be viewed as traversing a labeled binary tree, until all nodes reach to their final partition. A node in state c_i has reached its final partition when $i \geq k - 1$. This process describes a subdivision of the nodes,

Protocol 6 Graph of Supernodes

$Q = \{c_i, P_j\} \times \{0,1\}$, $0 \le i \le 2(k-1)$, $0 \le j \le k-1$
Initial state: c_0

δ :
\# Partitioning
1: $(c_i,\ c_i,\ 0) \to (c_{2i+1},\ c_{2i+2},\ 0)$, if $(i+1) < k$
2: $(c_i,\ \cdot,\ \cdot) \to (P_j,\ \cdot,\ \cdot)$, if $(i \ge k-1)$, $j = i-k+1$

\# Formation of graph H
3: $(P_i, P_j, 0) \to (P_i, P_j, 1)$, if $(i,j) \in H$
4: $(P_i, P_j, 1) \to (P_i, P_j, 0)$, if $(i,j) \notin H$
5: $(c_i, P_j, 0) \to (c_i, P_j, 1)$, if $(i,j) \in H$
6: $(c_i, P_j, 1) \to (c_i, P_j, 0)$, if $(i,j) \notin H$
7: $(c_i, c_j, 0) \to (c_i, c_j, 1)$, if $(i,j) \in H$
8: $(c_i, c_j, 1) \to (c_i, c_j, 0)$, if $(i,j) \notin H$

\# All transitions that do not appear have no effect.

where each partition splits into two partitions of equal size. The final partitions are $\{c_{k-1}, c_k, \dots, c_{2k-2}\}$.

Assume now that the initial population size is n_0 (level 0 of the binary tree). If n_0 is even, the size of the following two partitions c_1 and c_2 will be $n_0/2$. If n_0 is odd, one node remains unmatched, thus, the size of c_1 and c_2 will be $n_1 = \frac{n_0-1}{2}$. In the next level of the binary tree, at most one node will remain unmatched in each partition, thus $n_2 = \frac{n_1-1}{2}$. Consequently, the size of a partition in level p can be calculated recursively, and (in the worst case) it is $n_p = \frac{n_{p-1}-1}{2}$.

$$\begin{aligned} n_p &= \frac{n_{p-1}-1}{2} = \frac{n_{p-1}}{2} - \frac{1}{2} = \frac{\frac{n_{p-2}}{2} - \frac{1}{2}}{2} - \frac{1}{2} \\ &= \frac{n_{p-2}}{4} - \frac{1}{4} - \frac{1}{2} = \cdots = \frac{n_0}{2^p} - \sum_{i=1}^{p} \frac{1}{2^i} \\ &= \frac{n_0}{2^p} - (1 - 2^{-p}) > \frac{n_0}{2^p} - 1 \end{aligned} \tag{4.1}$$

For $p = \log k$ levels, each partition has either $\frac{n_0}{k}$ or $\frac{n_0}{k} - 1$ nodes. □

Lemma 4.16. *Protocol 6, stabilizes after $\Theta(kn^2)$ expected time.*

Proof. Protocol 6 operates in phases, where each phase doubles the number of partitions. After $\log k$ phases, there exist k groups in the population and the nodes terminate.

We now study the time that each group c_i needs in order to split into two partitions. Here, for simplicity, i indicates the level of a partition c in the binary tree and m_i the number of nodes of partition c_i.

Let X be a random variable (r.v.) defined to be the number of steps until all m_i nodes move to their next partitions. Call a step a success if two nodes in c_i interact, thus, moving to their next partitions. We divide the steps of the protocol into *epochs*, where epoch j begins with the step following the jth success and ends with the step at which the $(j+1)$st success occurs. Let also the r.v. X_j, $1 \leq j \leq m_i$ be the number of steps in the jth epoch.

The probability of success during the jth epoch, for $0 \leq j \leq m_i$, is $p_j = \frac{(m_i-j)(m_i-j-1)}{n(n-1)}$ and $E[X_j] = 1/p_j$. By linearity of expectation we have

$$\begin{aligned} E[X] = E[\sum_{j=0}^{m_i-2} X_j] = \sum_{j=0}^{m_i-2} E[X_j] = n(n-1)\sum_{j=0}^{m_i-2} \frac{1}{(m_i-j)(m_i-j-1)} \\ = n(n-1)\sum_{j=2}^{m_i} \frac{1}{j(j-1)} < n(n-1)\sum_{j=2}^{m_i} \frac{1}{(j-1)^2} \\ = n(n-1)\sum_{j=1}^{m_i-1} \frac{1}{j^2} < n(n-1)\frac{\pi^2}{6} = O(n^2) \end{aligned} \tag{4.2}$$

The above uses the fact that $m_i \leq n$ for any $i \geq 0$.

For the lower bound, observe that the last two remaining nodes in c_i need on average $n(n-1)/2$ steps to meet each other. Thus, we conclude that $E[X] = \Theta(n^2)$.

In total, $\sum_0^{\log(k)-1} 2^i = 2^{\log k} - 1 = k-1$ partitions split, thus, the total expected time to stabilization is $\Theta(kn^2)$ steps. □

Lemma 4.17. *In the case where up to f faults occur during the execution of Protocol 6, each final partition has at least $n/k-f-1$ nodes, where k is the number of partitions and $f < k$.*

Proof. Call $\mathcal{P}_i^p$ the set of partitions that are in the binary tree starting from a partition c_i in distance p from c_i. We now study the relation between the number of faults on some partition c_i with the size of the partitions in $\mathcal{P}_i^p$.

Consider the case where f_1 crash faults occur in some partition c_i. The nodes of each partition c_i operate independently from the rest of the population, that is, they never update their states and/or connections when they interact with nodes from a different partition. Thus, if no more faults occur, we can assume that we have a failure-free execution on $|c_i| - f_1$ nodes. By Lemma 4.15, after p subdivisions, each partition in $\mathcal{P}_i^p$ will have $\left\lfloor \frac{|c_i|-f_1}{2^p} \right\rfloor$ nodes. Consequently, any number of faults in a partition c_i are equally split into the partitions following c_i.

Now, consider a partition $c_j \in \mathcal{P}_i^{p_1}$, where f_2 faults occur. The number of nodes of c_j is then

$$|c_j| - f_2 = \left\lfloor \frac{|c_i| - f_1}{2^{p_1}} \right\rfloor - f_2 \tag{4.3}$$

Then, all the partitions in $\mathcal{P}_j^{p_2}$, by Lemma 4.15 will have at least

$$\left\lfloor \frac{|c_j| - f_2}{2^{p_2}} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{|c_i|-f_1}{2^{p_1}} \right\rfloor - f_2}{2^{p_2}} \right\rfloor \geq \frac{\frac{|c_i|-f_1}{2^{p_1}} - 1 - f_2}{2^{p_2}} - 1 > \frac{\frac{|c_i|}{2^{p_1}} - f_1 - f_2 - 1}{2^{p_2}} - 1 \tag{4.4}$$

nodes. This means that if $f_1 + f_2$ faults were occurred only in c_j (and not f_1 faults in c_i), then the subsequent partitions in $\mathcal{P}_j^p$ would have less nodes. This argument can be generalized for faults in any partition.

It is then obvious that in the worst case, where up to f faults can occur, a final partition (leaf of the binary tree) will have $\frac{n}{k} - f - 1$ nodes, and this is the result of f faults in that final partition. □

By Lemma 4.17:

Corollary 4.18. $||V_i| - |V_j|| \leq f + 1$, $\forall 1 \leq i, j \leq k$.

By Lemma 4.17 and the definition of partial constructibility (Definition 4.5):

Theorem 4.19. *The language $L_{D,f}$, where k is a constant number, is partially constructible under f faults.*

We now show that if we permit a waste linear in n, any graph language that is constructible in the fault-free NET model, becomes constructible under a bounded number of faults.

Theorem 4.20. *Take any NET protocol Π of the original fault-free model. There is a NET Π' such that when at most f faults may occur on any population of size n, Π' successfully simulates an execution of Π on at least $\frac{n}{2f} - 1$ nodes.*

Proof. Consider any constructible language L and a protocol Π that constructs it. For any bounded number of faults f, set $k = 2^\epsilon$, where $2^{\epsilon-1} < f < 2^\epsilon$. Consider a protocol Π', which consists of the rules 1 and 2 of Protocol 6. These rules partition the population into k groups, where k is an input parameter of Π'. By Lemma 4.17, each group has at least $n/k - f - 1$ nodes. For 2^ϵ partitions, the number of nodes in each partition is at least $\frac{n}{2^\epsilon} - f - 1$. However, as the number of partitions is strictly more than the upper bound on the number of faults ($2^\epsilon > f$), there exists at least

one partition that no fault has occurred. In the worst case where $f = 2^\delta$ for some $\delta \in \mathbb{N}$, there exists at least one partition with $\frac{n}{2f} - 1$ nodes. □

4.4 Notified Network Constructors

In light of the impossibility results of Section 4.3, we allow fault notifications when nodes crash, aiming at constructing a larger class of graph languages. In particular, we introduce a *fault flag* in each node, which is initially zero. When a node u crashes at time t, every node v which was adjacent to u at time t is notified, that is, the fault flag of all v becomes 1 (see Figures 4.2a and 4.2b). In the case where u is an isolated node (i.e., it has no active edges), an arbitrary node w in the graph is notified, and its fault flag becomes 2 (see Figures 4.2c and 4.2d). Then, the fault flag becomes immediately zero after applying a corresponding rule from the transition function.

More formally, the set of node-states is $Q \times \{0, 1, 2\}$, and for clarity in our descriptions and protocols, we define two types of transition functions. The first one determines the node and connection state updates of pairwise interactions ($\delta_1 : Q \times Q \times \{0, 1\} \to Q \times Q \times \{0, 1\}$), while the second transition function determines the node state updates due to fault notifications ($\delta_2 : Q \times \{1, 2\} \to Q \times \{0\}$). This means that during a step t that a node u crashes, all its adjacent nodes are allowed to update their states based on δ_2 at that same step. If there are no adjacent nodes to u, an arbitrary node is notified, thus, updating its state based on δ_2 at step t.

We have assumed that the faults can only occur sequentially (at most one fault per step). This assumption was equivalent to the case where many faults can occur in each step in the original NET model. However, when fault notifications are allowed, this does not hold, unless the fault flag could be used as a counter of faults in each step. We want to keep the model as minimal as possible, thus, we only allow the adversary to choose one node at most in each step to crash.

In this section, we investigate whether the additional information in each agent (the fault flag) is sufficient in order to design fault-tolerant or $f-$fault-tolerant protocols, overcoming the impossibility of certain graph languages in the NET model. Such a minimal fault notification mechanism can be exploited to construct a larger class of graph languages than in the original Network Constructors model where no form of notifications was available.

4.4.1 Fault-tolerant protocols

In this section, we give protocols for some basic network construction problems, such as *spanning star* (all $u \in G$ form a single star), *cycle cover* (set of cycles which are

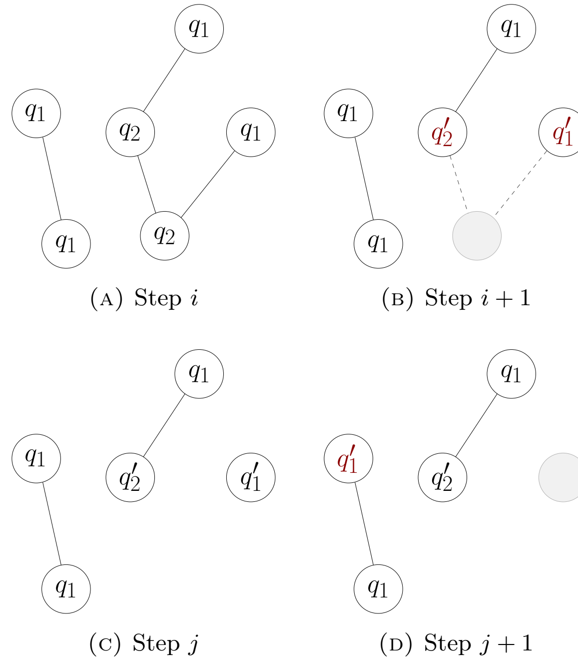


FIGURE 4.2: An illustration of the fault notification mechanism. In the first example (Figures 4.2a and 4.2b), the gray node crashed, and the nodes that were adjacent to it at step i were notified and updated their state. At step $j > i+1$, this node along with its adjacent edges are not present. In the second example (Figures 4.2c and 4.2d), the crashed node was isolated, thus an arbitrary node was notified and updated its state.

subgraphs of G and contain all vertices of G), and in Section 4.4.2 we give a fault-tolerant spanning line protocol which is part of our generic constructor capable of constructing a large class of networks.

Protocol 7 constructs a *spanning star*. Initially all nodes are in the same state b (or *black*) and they eliminate each other, becoming r (or *red*). Eventually, only one node will remain in state b which will be the center of the star. In order to handle crash faults, when a *red* node is notified about a fault, it becomes *black*. In this way and because of the fact that a *red* node cannot be isolated, we guarantee that a black node will always exist in the population.

Proposition 4.21. FT Spanning Star *(Protocol 7) is a fault-tolerant protocol that constructs a* spanning star*.*

Proof. Assume that any number of faults $f < n$ occur during an execution. Initially, all nodes are in state b (*black*). Two nodes connect with each other, if either one of them is black, or both of them are black, in which case one of them becomes r (*red*). A black node can become red only by interaction with another black node, in which case they also become connected. Thus, with no crash faults, a connected component

Protocol 7 FT Spanning Star

$Q = \{b,\ r\} \times \{0, 1\}$
Initial state: b

δ_1 :
Formation of spanning star. Eventually, only one node in state b remains.
1: $(b,\ b,\ 0) \rightarrow (b,\ r,\ 1)$
2: $(b,\ r,\ 0) \rightarrow (b,\ r,\ 1)$
3: $(r,\ r,\ 1) \rightarrow (b,\ b,\ 0)$
4: $(b,\ b,\ 1) \rightarrow (b,\ r,\ 1)$

δ_2 :
5: $(r, 1) \rightarrow (b, 0)$ # A leaf becomes the initial state b after a fault notification.

All transitions that do not appear have no effect.

always includes at least one black node. In addition, all isolated nodes are always in state b. This is because, if a red node removes an edge it becomes black.

Then, if a (connected) node crashes, the adjacent nodes are notified and the red nodes become black, thus, any connected component should again include at least one black node. Now, consider the case where only one black node remains in the population. Then the rest of the population (in state r) should be in the same connected component as the unique b node. Then, if b crashes, at least one black node will appear, thus, this protocol maintains the invariant, as there is always at least one black node in the population. Finally, all connected nodes in state r will eventually disconnect from each other. *FT Spanning Star* then stabilizes to a star with a unique black node in the center. □

Similarly, we can show the following.

Proposition 4.22. FT Cycle-Cover *(Protocol 8) is a fault-tolerant protocol that forms a* cycle cover.

In this protocol, the state of a node indicates its degree. In particular, all nodes are initially in state q_0, indicating that they are isolated. Whenever a node in state q_i, forms a connection, it moves to state q_{i+1}. At the same time, whenever a node is notified about a fault in a neighboring node, it decreases i by one. In this protocol $0 \leq i \leq 2$, which guarantees that eventually all nodes will have degree 2, except maybe for a single node or a pair of nodes which will form a line.

4.4.2 Universal fault-tolerant constructors

In this section, we ask whether there is a generic fault-tolerant constructor capable of constructing a large class of graphs. We first give a fault-tolerant protocol that

Protocol 8 FT Cycle-Cover

$Q = \{q_0,\ q_1,\ q_2\} \times \{0, 1\}$
Initial state: q_0

δ_1 :
1: $(q_0,\ q_0,\ 0) \to (q_1,\ q_1,\ 1)$
2: $(q_1,\ q_0,\ 0) \to (q_2,\ q_1,\ 1)$
3: $(q_1,\ q_1,\ 0) \to (q_2,\ q_2,\ 1)$

δ_2 :
The state of a node indicates its degree. A fault notification implies that the degree was decreased by one.
4: $(q_1, 1) \to (q_0, 0)$
5: $(q_2, 1) \to (q_1, 0)$

All transitions that do not appear have no effect.

constructs a spanning line (i.e., a graph of size n that forms a line), and then we show that we can simulate a given TM on that line, tolerating any number of crash faults. Finally, we exploit that in order to construct any graph language that can be decided by an $O(n^2)-$space TM, paying at most linear waste.

Lemma 4.23. FT Spanning Line *(Protocol 9) is a fault-tolerant protocol that constructs a* spanning line*.*

Proof. Initially, all nodes are in state q_0 and they start connecting with each other in order to form lines that eventually merge into one.

When two q_0 nodes become connected, one of them becomes a leader (state l_0) and starts connecting with q_0 nodes (expands). A leader state l_0 is always an endpoint. The other endpoint is in state e_i (initially e_1), while the inner nodes are in state q_2. Our goal is to have only one leader l_0 on one endpoint, because l_0 are also used in order to merge lines. Otherwise, if there are two l_0 endpoints, the line could form a cycle. When two l_0 leaders meet, they connect (line merge) and a w node appears. This process corresponds to the rules 1, 2, and 3 of Protocol 9 (depicted also in Figure 4.3).

The w state performs a random walk on the line and its purpose is to meet both endpoints (at least once) before becoming an l_0 leader. After interacting with the first endpoint, it becomes w_1 and changes the endpoint to e_1. Whenever it interacts with the same endpoint they just swap their states from e_1, w_1 to e_2, w_2 and vice versa. In this way, we guarantee that w_i will eventually meet the other endpoint in state e_j, $j \neq i$, or l_0. In the first case, the w_i node becomes a leader (l_0), after having walked the whole line at least once. This process is described by rules $4 - 10$ of Protocol 9 (depicted also in Figure 4.4).

Protocol 9 FT Spanning Line

$Q = \{q_0,\ q_2,\ e_1,\ e_2,\ l_0,\ l_1,\ w,\ w_1,\ w_2\} \times \{0,1\}$
Initial state: q_0

δ_1 :
Formation of lines, and merging between them.
1: $(q_0,\ q_0,\ 0) \to (e_1,\ l_0,\ 1)$
2: $(l_0,\ q_0,\ 0) \to (q_2,\ l_0,\ 1)$
3: $(l_0,\ l_0,\ 0) \to (q_2,\ w,\ 1)$

w nodes perform a random walk on their line.
4: $(w_i,\ q_2,\ 1) \to (q_2,\ w_i,\ 1)$
5: $(w,\ q_2,\ 1) \to (q_2,\ w,\ 1)$

Nodes in state w introduce a unique endpoint in state l_0 on their line.
6: $(w,\ e_i,\ 1) \to (w_i,\ e_i,\ 1)$
7: $(w_i,\ e_i,\ 1) \to (w_j,\ e_j,\ 1),\ i \neq j$
8: $(w_i,\ e_j,\ 1) \to (q_2,\ l_0,\ 1),\ i \neq j$
9: $(w,\ l_i,\ 1) \to (w_1,\ e_1,\ 1)$
10: $(w_i,\ l_i,\ 1) \to (q_2,\ l_0,\ 1)$

w nodes eliminate each other, until only one survives.
11: $(w_i,\ w_j,\ 1) \to (w,\ q_2,\ 1)$
12: $(w,\ w_j,\ 1) \to (w,\ q_2,\ 1)$

Fault notifications on internal nodes (states q_2, w, and w_i), become l_1, which then introduce a new walking state.
13: $(l_1,\ q_2,\ 1) \to (e_1,\ w_1,\ 1)$

δ_2 :
14: $(e_i, 1) \to (q_0, 0)$
15: $(l_i, 1) \to (q_0, 0)$
16: $(q_2, 1) \to (l_1, 0)$
17: $(w, 1) \to (l_1, 0)$
18: $(w_i, 1) \to (l_1, 0)$

All transitions that do not appear have no effect.

Now, consider the case where a fault may happen on some node on the line. If the fault flag of an endpoint state becomes 1, it updates its state to q_0. Otherwise, the line splits into two disjoint lines and the new endpoints become l_1. An l_1 becomes a walking state w_1, changes the endpoint into e_1 and performs a random walk (rule 13 of Protocol 9).

If there are more than one walking states on a line, then all of them are w, or w_i and they perform a random walk. None of them can ever satisfy the criterion

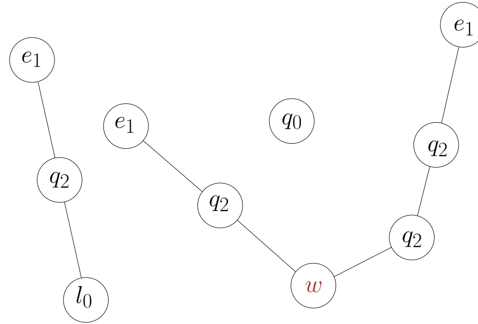


FIGURE 4.3: The left line is the result of one connection between two isolated nodes, and one expansion (rules 1 and 2 of Protocol 9). The second line is the result of a line merging (rule 3 of Protocol 9).

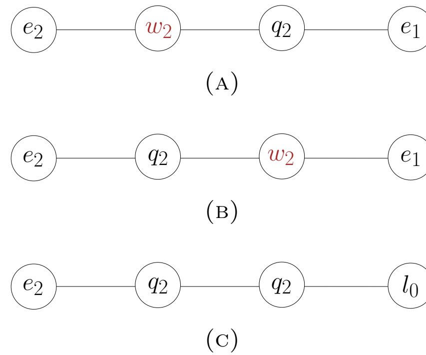


FIGURE 4.4: In Figure 4.4a, the walking state and the left endpoints are in states w_2 and e_2. Then, the walking state eventually reaches the second endpoint which is in state e_1, resulting to state l_0 (Figure 4.4c).

to become l_0 before first eliminating all the other walking states and/or the unique leader l_0 (when two walking states meet, only one survives and becomes w), simply because they form natural obstacles between itself and the other endpoint (rules 11 and 12 of Protocol 9). This process is depicted in Figure 4.5. If a new fault occurs, then this can only introduce another w_i state which cannot interfere with what existing w_i's are doing on the rest of the line (can meet them eventually but cannot lead them into an incorrect decision).

If an l_0 leader is merging while there are w_i's and/or w's on its line (without being aware of that), the merging results in a new w state, which is safe because a w cannot make any further progress without first succeeding to beat everybody on the line. A w can become l_0 only after walking the whole line at least once (i.e., interact with both endpoints) and to do that it must have managed to eliminate all other walking states of the line on its way.

We have shown that despite the presence of faults, any expansion or merging eventually succeeds, meaning that the population eventually forms a line with a single leader in one endpoint. $\square$

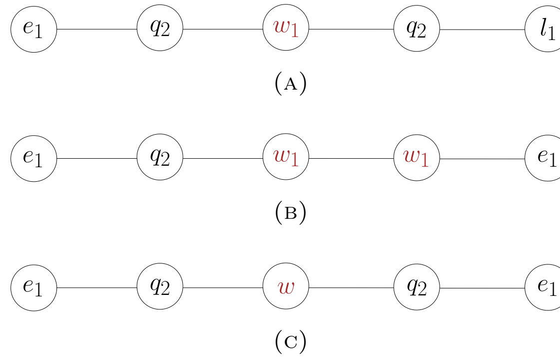


FIGURE 4.5: In Figure 4.5a there is one walking state, and one endpoint in state l_1. l_1 state is the result of a fault on an adjacent node of it. This state introduces an additional walking state, and in Figure 4.5c the two walking states interact and only one survives. The unique walking state w is then guaranteed to first traverse the whole line at least once before an endpoint becomes l_0.

Lemma 4.24. *There is a NET* Π *(with notifications) such that when* Π *is executed on* n *nodes and at most* f *faults can occur, where* $0 \leq f < n$, Π *will eventually simulate a given TM* M *of space* $O(n - f)$ *in a fault-tolerant way.*

Proof. The state of Π has two components (P, S), where P is executing a spanning line formation procedure, while S handles the simulation of the TM M. Our goal is to eventually construct a spanning line, where initially the state of the second component of each node is in an initial state s_0 except from one node which is in state *head* and indicates the head of the TM.

In general, the states P and S are updated in parallel and independently from each other, apart from some cases where we may need to reset either P, S or both.

In order to form a spanning line under crash failures, the P component will be executing our *FT Spanning Line* protocol which is guaranteed to construct a line, spanning eventually the non-faulty nodes.

It is sufficient to show that the protocol can successfully reinitialize the state of all nodes on the line after a final *event* has happened and the line is stable and spanning. Such an *event* can be a line merging, a line expansion, a fault on an endpoint or an intermediate fault. The latter though can only be a final event if one of the two resulting lines is completely eliminated due to faults before merging again. In order to re-initialize the TM when the line expands to an isolated node q_0, we alter a rule of the *FT Spanning Line* protocol. Whenever, a leader l_0 expands to an isolated node q_0, the leader becomes q_2 while the node in q_0 becomes l_1, thus introducing a new walking state.

We now exploit the fact that in all these cases, *FT Spanning Line* will generate a w or a w_i state in each affected component.

Whenever a w_1 or w_2 state has just appeared or interacted with an endpoint e_1 or e_2 respectively, it starts resetting the simulation component S of every node that it encounters. If it ever manages to become a leader l_0, then it finally restarts the simulation on the S component by reintroducing to it the *tape head*.

When the last event occurs, the final spanning line has a w or w_i leader in it, and we can guarantee a successful restart due to the following invariant. Whenever a line has at least one w/w_i state and no further events can happen, *FT Spanning Line* guarantees that there is one w or w_i that will dominate every other w/w_i state on the line and become an l_0, while having traversed the line from endpoint to endpoint at least once.

In its final departure from one endpoint to the other, it will dominate all w and w_i states that it will encounter (if any) and reach the other endpoint. Therefore, no other w/w_i states can affect the simulation components that it has reset on its way, and upon reaching the other endpoint it will successfully introduce a new *head* of the TM while all simulation components are in an initial state s_0. □

Lemma 4.25. *There is a fault-tolerant NET* Π *(with notifications) which partitions the nodes into two groups* U *and* D *with waste at most* $2f(n)$*, where* $f(n)$ *is an upper bound on the number of faults that can occur.* U *is a spanning line with a unique leader in one endpoint and can eventually simulate a TM* M*. In addition, there is a perfect matching between* U *and* D*.*

Proof. Initially all nodes are in state q_0. Protocol Π partitions the nodes into two equal sets U and D and every node maintains its type forever. This is done by a perfect matching between q_0's where one becomes q_u and the other becomes q_d. Then, the nodes of U execute the *FT Spanning Line* protocol, which guarantees the construction of a spanning line, capable of simulating a TM (Lemma 4.24). The rest of the nodes (D), which are connected to exactly one node of U each, are used to construct on them random graphs. Whenever a line merges with another line or expands towards an isolated node, the simulation component S in the states of the line nodes, as described in Lemma 4.24, is reinitialized sequentially.

Assume that a fault occurs on some node of the perfect matching before that pair has been attached to a line. In this case, its pair will become isolated therefore it is sufficient to switch that back to q_0.

If a fault occurs on a D node u after its pair z has been attached to a line, z goes into a detaching state which disconnects it from its line neighbors, turning them into l_1 and itself becoming a q_0 upon release. An l_1 state on one endpoint is guaranteed to walk the whole line at least once (as w_i) in order to ensure that a unique leader l_0 will be created. If u fails before completing this process, its neighbors on the line shall be notified becoming again l_1, and if one of its neighbors fails we shall treat

this as part of the next type of faults. This procedure shall disconnect the line but may leave the component connected through active connections within D. But this is fine as long as the *FT-Spanning Line* guarantees a correct restart of the simulation after any event on a line. This is because eventually the line in U will be spanning and the last event will cause a final restart of the simulation on that line.

Assume that a fault occurs on a node $u \in U$ that is part of the line. In this case the neighbors of u on the line shall instantly become l_1. Now, its D pair v, which may have an unbounded number of D neighbors at that point, becomes a special *deactivating state* that eventually deactivates all connections and never participates again in the protocol, thus, it stays forever as waste. This is because the fault partially destroys the data of the simulation, thus, we cannot safely assume that we can retrieve the degree of v and successfully deactivate all edges. As there can be at most $f(n)$ such faults we have an additional waste of $f(n)$. Now, consider the case where u is one neighbor of a node z which is trying to release itself after its v neighbor in D failed. Then, z implements a 2-counter in order to remember how many of its alive neighbors have been deactivated by itself or due to faults in order to know when it should become q_0. □

Theorem 4.26. *For any graph language L that can be decided by a linear space TM, there is a fault-tolerant NET* Π *(with notifications) that constructs a graph in L with waste at most* $min\{n/2 + f(n),\ n\}$*, where* $f(n)$ *is an upper bound on the number of faults that can occur.* [1]

Proof. By Lemma 4.25, there is a protocol that constructs two groups U and D of equal size, where each node of U is matched with exactly one node of D, and vice versa. In addition, the nodes of U form a spanning line, and by Lemma 4.24 it can simulate a TM M. After the last fault occurs, M is correctly initialized and the head of the TM is on one of the endpoints of the line. The two endpoints are in different states, and assume that the endpoint that the head ends up is in state q_l (*left* endpoint), and the other is in state q_r (*right* endpoint). This construction is depicted in Figure 4.6.

We now provide the protocol that performs the simulation of the TM M, which we separate into several subroutines. The first subroutine is responsible for simulating the direction on the tape and is executed once the head reaches the endpoint q_l. The simulation component S (as in Lemma 4.24) of each node has three sub-components (h, c, d). h is used to store the head of the TM, i.e., the actual state of the control of the TM, c is used to store the symbol written on each cell of the TM, and d is either

[1] Given a target graph of size $|V(G)|$, the size of the initial population required to construct G depends on the number of faults that occur and on the state of the nodes during the crash failures. In particular, the minimum size required to construct G is $2n$ (no faults occur), while the maximum number of nodes is $2(n + f(n))$.

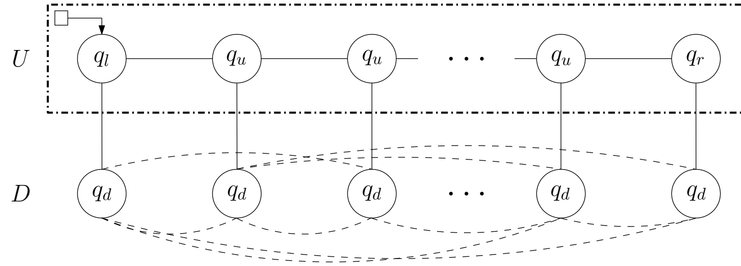


FIGURE 4.6: The population is partitioned into two groups U and D that form a perfect matching. The nodes of U eventually form a spanning line, and simulate a TM of linear space. The TM repeatedly constructs random graphs in the nodes of D, until the graph belongs to the given graph language.

l, r or $\sqcup$, indicating whether that node is on the left or on the right of the head (or unknown). Assume that after the initialization of the TM, $d = \sqcup$ for all nodes of the line. Finally, whenever the head of the TM needs to move from a node u to a node z, $h_z \leftarrow h_u$, and $h_u \leftarrow \sqcup$.

Direction. Once the head of the TM is introduced in the endpoint q_l by the lines' leader, it moves on the line, leaving l marks on the d component of each node. It moves on the nodes which are not marked, until it eventually reaches the q_r endpoint. At that point, it starts moving on the marked nodes, leaving r marks on its way back. Eventually, it reaches again the q_l endpoint. At that time, for each node on its right it holds that $d = r$. Now, every time it wants to move to the right it moves onto the neighbor that is marked by r while leaving an l mark on its previous position, and vice versa. Once the head completes this procedure, it is ready to begin working as a TM.

Construction of random graphs in D. This subroutine of the protocol constructs a random graph in the nodes of D. Here, the nodes are allowed to toss a fair coin during an interaction. This means that we allow transitions that with probability 1/2 give one outcome and with 1/2 another. To achieve the construction of a random graph, the TM implements a binary counter C ($\log n$ bits) in its memory and uses it in order to uniquely identify the nodes of set D according to their distance from q_l. Whenever it wants to modify the state of edge (i, j) of the network in D, the head assigns special marks to the nodes in D at distances i and j from the left of the endpoint q_l. Note that the TM uses its (distributed) binary counter in order to count these distances. If the TM wants to access the $i-$th node in D, it sets the counter C to i, places a mark on the left endpoint q_l and repeatedly moves the mark one position to the right, decreasing the counter by one in each step, until $C = 0$. Then, the mark has been moved exactly i positions to the right. In order to construct a random graph in D, it first assigns a mark r_1 to the first node q_l, which indicates that this node should perform random coin tosses in its next interactions with the

other marked nodes, in order to decide whether to form connections with them, or not. Then, the leader moves to the next node on its line and waits to interact with the connected node in D. It assigns a mark r_2, and waits until this mark is deleted. The two nodes that have been marked (r_1 and r_2), will eventually interact with each other, and they will perform the (random) experiment. Finally the second node deletes its mark (r_2). The head then, moves to the next node and it performs the same procedure, until it reaches the other endpoint q_r. Finally, it moves back to the first node (marked as r_1), deletes the mark and moves one step right. This procedure is repeated until the node that should be marked as r_1 is the right endpoint q_r. It does not mark it and it moves back to q_l. The result is an equiprobable construction of a random graph. In particular, all possible graphs over $|D|$ nodes have the same probability to occur. Now, the input to the TM M is the random graph that has been drawn on D, which provides an encoding equivalent to an adjacency matrix. Once this procedure is completed, the protocol starts the simulation of the TM M. There are $m = k(k-1)/2$ edges, where $k = |D|$ and M has available $\frac{k}{2} = \sqrt{m}$ space, which is sufficient for the simulation on a $\sqrt{m}-$space TM.

Read edges of D. We now present a mechanism, which can be used by the TM in order to read the state of an edge joining two nodes in D. Note that a node in D can be uniquely identified by its distance from the endpoint q_l. Whenever the TM needs to read the edge joining the nodes i and j, it sets the counter C to i. Assume w.l.o.g. that $i < j$. It performs the same procedure as described in the subroutine which draws the random graph in D. It moves a special mark to the right, decreasing C by one in each step, until it becomes zero. Then, it assigns a mark r_3 on the $i-$th node of D, and then performs the same for $C = j$, where it also assigns a mark r_4 (to the $j-$th node). When the two marked nodes (r_3 and r_4) interact with each other, the node which is marked as r_4 copies the state of the edge joining them to a flag $\mathcal{F}$ (either 0 or 1), and they both delete their marks. The head waits until it interacts again with the second node, and if the mark has been deleted, it reads the value of the flag $\mathcal{F}$.

After a simulation, the TM either accepts or rejects. In the first case, the constructed graph belongs to L and the Turing Machine halts. Otherwise, the random graph does not belong to L, thus the protocol repeats the random experiment. It constructs again a random graph, and starts over the simulation on the new input.

A final point that we should make clear is that if during the simulation of the TM an event occurs (crash fault, line expansion, or line merging), by Lemma 4.24 and Lemma 4.25, the protocol reconstructs a valid partition between U and D, the TM is re-initialized correctly, and a unique head is introduced in one endpoint. At that time, edges in D may exist, but this fact does not interfere with the (new) simulation

of the TM, as a new random experiment takes place for each pair of nodes in D prior to each simulation. □

We now show that if the constructed network is required to occupy 1/3 instead of half of the nodes, then the available space of the TM-constructor dramatically increases from $O(n)$ to $O(n^2)$. We provide a protocol which partitions the population into three sets U, D and M of equal size $k = n/3$. The idea is to use the set M as a $\Theta(n^2)$ binary memory for the TM, where the information is stored in the $k(k-1)/2$ edges of M.

Lemma 4.27. *Protocol 10 (3-Partition) partitions the nodes into three groups U, D and M, with waste $3f(n)$, where $f(n)$ is an upper bound on the number of faults that can occur. U is a spanning line with a unique leader in one endpoint and can eventually simulate a TM, each node in $D \cup M$ is connected with exactly one node of U, and each node of U is connected to exactly one node in D and one node in M.*

Proof. Protocol 3−Partition constructs lines of three nodes each, where one endpoint is in state q_d, the other endpoint in state q_m, and the center is in state q_u. The nodes of U operate as in Lemma 4.25 (i.e., they execute the *FT Spanning Line* protocol). A (connected) pair of nodes waits until a third node is attached to it, and then the center becomes q_u and starts executing the *FT Spanning Line* protocol. Note that at some point, it is possible that the population may only consist of pairs in states q_d and q'_u. For this reason, we allow q'_u nodes to connect with each other, forming lines of four nodes. One of the q'_u nodes becomes q_u and the other becomes q'_m. A node in q'_m becomes q_m only after deactivating its connection with a q_d node (its previous pair). This results in lines of three nodes each with nodes in states q_d, q_u and q_m. Then, the q_u nodes start forming a line, spanning all nodes of U. In a failure-free setting, the correctness of this protocol follows from Lemma 4.25. In addition, by Lemma 4.24, the TM of the line is initialized correctly after the last occurring event (line expansion, line merging, or crash fault).

If we consider crash failures, it is sufficient to show that eventually U is a spanning line and M and D are disjoint. If a node ever becomes q_d or q_m, it might form connections with other nodes in D or M respectively, because of a TM simulation. A node in M never forms connections with nodes in D. After they receive a fault notification, they become the *deactivating state s*. A node in state s is disconnected from any other node, thus, it eventually becomes isolated and never participates in the execution again. We do this because nodes in M and D can form unbounded number of connections. The data of the TM have been partially destroyed (because of the crash failure), therefore it is not safe to assume that we can retrieve the degree of them and successfully re-initialize them.

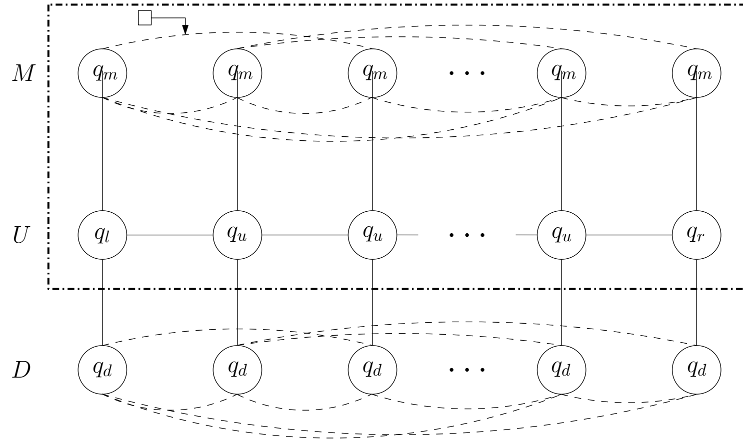


FIGURE 4.7: The population is partitioned into three groups M, U, and D of equal size. The nodes of U form a perfect matching with both U and M. The nodes of U eventually form a spanning line, and simulate a TM of linear space that uses the edges in M as an $O(n^2)$ binary memory. The TM repeatedly constructs random graphs in the nodes of D, until the graph belongs to the given graph language.

A node u in state q'_m (inner node of a line of four nodes), after a fault notification it becomes q_w. A node in q_w waits until its next interaction with a connected node v. If v is in state q_u, this means that now a triple has been formed, thus u becomes q_m. If v is in state q_d, they delete the edge joining them, u becomes q_0 and v becomes s (v might have formed connections with other nodes in D).

A node u in q_u, after a fault notification it becomes q'_w and waits until its next interaction with a connected node v. At that point, v can be either q_d, q'_m, or q_m. In all cases they disconnect from each other and u becomes q'_0. The state q'_0 indicates that the node should release itself from the spanning line in U. This procedure works as described in Lemma 4.25, thus, after releasing itself from the line, it becomes q_0. If v is in state q_d or q_m, it becomes s. If v is in state q'_m, it becomes q'_u, as its (unique) adjacent node can only be in state q_d.

A node in q'_u or q_w, after a fault notification it becomes q_0 and continues participating in the execution again. Finally, a node in state q'_w, after receiving a fault notification, it becomes q'_0 (a q'_w is the result of a fault notification in a $U-$ node).

Note that a node in any state except from q_d and q_m can be re-initialized correctly, thus they may participate in the execution again. It is apparent that no node that might have formed unbounded number of connections can participate in the execution again after a crash fault. This guarantees that the connections in D and M can be correctly initialized after the final event, and that no node in $D \cup M$ can be connected with more than one node in U. In addition, if a $U-$node receives a fault notification, it releases itself from the line, thus introducing new walking states in the resulting line(s). By Lemma 4.24, this guarantees the correct re-initialization of the TM.

Finally, a crash failure can lead in deactivating two more nodes, in the worst case. These nodes never participate in the execution again, thus they remain forever as waste. This means that after $f(n)$ crash failures, the partitioning will be constructed in $n - 3f(n)$ nodes. □

Protocol 10 3-Partition

$Q = \{q_0,\ q_0',\ q_d,\ q_u,\ q_u',\ q_m,\ q_m',\ q_w,\ q_w',\ s\} \times \{0, 1\}$

Initial state: q_0

δ_1 :

Formation of independent lines of three nodes.

1: $(q_0,\ q_0,\ 0) \to (q_u',\ q_d,\ 1)$
2: $(q_u',\ q_0,\ 0) \to (q_u,\ q_m,\ 1)$

Two connected pairs of nodes can form a line of four nodes. In this case, one of the endpoints is disconnected from the line.

3: $(q_u',\ q_u',\ 0) \to (q_u,\ q_m',\ 1)$
4: $(q_m',\ q_d,\ 1) \to (q_m,\ q_0,\ 0)$

q_w is the result of a fault on either a node in q_d or q_u state. The nodes in this state wait until they interact with (the unique) adjacent node, and update their state accordingly.

5: $(q_w,\ q_d,\ 1) \to (q_0,\ s,\ 0)$
6: $(q_w,\ q_u,\ 1) \to (q_m,\ q_u,\ 1)$

q_w' is the result of a fault on a node in q_d, q_m, or q_m' state. The nodes in this state wait until they interact with (the unique) adjacent node, and update their state accordingly. q_0' eventually becomes q_0 after releasing itself from the spanning line.

7: $(q_w',\ q_d,\ 1) \to (q_0',\ s,\ 0)$
8: $(q_w',\ q_m,\ 1) \to (q_0',\ s,\ 0)$
9: $(q_w',\ q_m',\ 1) \to (q_0',\ q_u',\ 0)$

Nodes in state s are disconnected from all nodes, and are left as waste.

10: $(s, \cdot,\ 1) \to (s, \cdot,\ 0)$

δ_2 :

11: $(q_u', 1) \to (q_0, 0)$

\# States q'_m and q_u indicate intermediate nodes of the line. After a fault notification, they enter to a temporary state, and wait until their first interaction with the remaining (unique) adjacent node.

12: $(q'_m, 1) \rightarrow (q_w, 0)$
13: $(q_u, 1) \rightarrow (q'_w, 0)$

\# A node in state q_w or q'_w, is guaranteed to have exactly one neighbor in the (q_m, q_u, q_d) line. Thus, after a fault notification, it becomes q_0 (or q'_0 if it belongs to the set U)

14: $(q_w, 1) \rightarrow (q_0, 0)$
15: $(q'_w, 1) \rightarrow (q'_0, 0)$

\# q_d and q_m nodes remain as waste (in state s) after a fault notification.

16: $(q_d, 1) \rightarrow (s, 0)$
17: $(q_m, 1) \rightarrow (s, 0)$

\# All transitions that do not appear have no effect.

Theorem 4.28. *For any graph language L that can be decided by an $(n^2/18 + O(n))-$space TM, there is a protocol that constructs L equiprobably with waste at most min$\{2n/3 + f(n),\ n\}$, where $f(n)$ is an upper bound on the number of faults.*

Proof. Protocol 10 partitions the population in three groups U, D and M and by Lemma 4.27, it tolerates any number of crash failures, while initializing correctly the TM after the final event (line expansion, line merging, or crash fault). Reading and writing on the edges of M is performed in precisely the same way as reading/writing the edges of D (described in Theorem 4.26). Thus, the Turing Machine has now a $n^2/18-$space binary memory (the edges of M) and $O(n)-$space on the nodes of the spanning line U. The random graph is constructed on the k nodes of D (useful space), where by Lemma 4.27, $k = (n - 3f(n))/3 = n/3 - f(n)$ in the worst case. □

4.4.3 Designing fault-tolerant protocols without waste

A very simple, (yet impractical) idea that could tolerate any number $f < n$ of faults is to restart the protocol each time a node crashes. The implementation of this idea requires the ability of some nodes to *detect* the removal of a node.

Definition 4.29 (Global restart)**.** Let Π be a protocol that constructs a graph language L and $\mathcal{C}$ be a set of configurations that all executions of Π starting from any $C \in \mathcal{C}$ stabilize to a graph $G \in L$. We call *global restart* the process which reaches Π to a configuration $C \in \mathcal{C}$ in finite time.

Our goal is to come up with a protocol A that can be composed with any NET protocol Π (with notifications), so that their composition is a fault-tolerant version

of Π. Essentially, whenever a fault occurs, A will restart all nodes in a way equivalent to as if a new execution of Π had started on the whole remaining population. Parallel execution of protocols is easily achieved in the Population Protocol model, by taking the Cartesian product of their state sets and updating the states for each protocol independently when a transition occurs ([12]). We denote the parallel composition of two protocols Π_1 and Π_2 as $\Pi_1 \circ \Pi_2$. However, in the Network Constructors model, the connections between the nodes are binary and the Cartesian product of their states would imply that each protocol Π_1 and Π_2 maintains its own connection state between each pair of nodes. To overcome this problem, we only consider parallel composition between two protocols where only one of them is allowed to activate/deactivate edges between the nodes. In particular, assume that Π_1 is a protocol with transition function $\delta_1 : Q_1 \times Q_1 \rightarrow Q_1 \times Q_1$ and Π_2 is a protocol with transition function $\delta_2 : Q_2 \times Q_2 \times \{0, 1\} \rightarrow Q_2 \times Q_2 \times \{0, 1\}$. Then, the composition of these protocols $\Pi_1 \circ \Pi_2$ is a Network Constructor with transition function $\delta_{1,2} : (Q_1 \times Q_2) \times (Q_1 \times Q_2) \times \{0, 1\} \rightarrow (Q_1 \times Q_2) \times (Q_1 \times Q_2) \times \{0, 1\}$.

Definition 4.30. Consider any execution $\mathcal{E}_i$ of a protocol Π. There exists a finite number of different executions, and for each execution a step t_i that Π stabilizes. Call $C_{i,j}$ the $j-$th configuration of execution $\mathcal{E}_i$, where $j \leq t_i$. Then, we call *maximum reachable degree* of Π the value $d = max\{\text{Degree}(G(C_{i,j}))\}, \forall i, j$.

We first show that even in the case where the whole population is notified about a crash failure, global restart is *impossible for protocols with* $d = \omega(1)$, *if the nodes have constant memory.* However, we provide a protocol that restarts the population, but we supply the nodes with $O(\log n)$ bits of memory. In our approach, we use fault notifications, and if a node z crashes, the set N_z of the nodes that are notified, has the task to restart the protocol.

Theorem 4.31. *Consider a protocol* Π *with unbounded maximum reachable degree. Then, global restart of* Π *is impossible for nodes with constant memory, even if every node u in the population is notified about the crash failure.*

Proof. Consider a protocol Π with constant number of states k and unbounded *maximum reachable degree*, which constructs a graph language L. Assume also that at time t a crash failure occurs and that there are some edges in the graph (call them *spurious edges*). Protocol Π is allowed to have rules that are triggered by the fault and try to erase those edges (*erasing process*). We assume that all nodes in the population are notified about the crash failure.

Observe that any degree more than k cannot be remembered by a node, that is, a state q cannot indicate its degree. This means that a node cannot detect the termination of the *erasing process* and eventually reset its state to an initial one to

allow the restart. To stop the erasing process is equivalent to counting the remaining edges and wait until the degree reaches zero, but this would require logarithmic to *maximum reachable degree* number of bits.

The above observation means that the agents must enter to a new initial state and start forming new connections prior to the termination of the *erasing process*. But the only way to distinguish connections made before and after a fault is to enter to a different set of states after each fault occurs. Otherwise the *erasing process* will fail. This can be achieved by having parallel executions of Π. In particular, given Π, the agents execute $\Pi' = \Pi^{(1)} \circ \Pi^{(2)} \circ \cdots \circ \Pi^{(\delta)}$, where $\Pi^{(i)}$ is obtained from Π by adding a constant i to it. Initially, the agents execute $\Pi^{(i)}$ for $i = 1$, and whenever a fault notification is received, the agents start executing $\Pi^{(i+1)}$.

As an example, consider the following protocol Π with initial state s and a single rule $(s, s) \to (r, r)$. Then $\Pi' = \Pi^{(1)} \circ \Pi^{(2)}$ has the following rules: $(s_1, s_1) \to (r_1, r_1)$ and $(s_2, s_2) \to (r_2, r_2)$, and the agents are initially in state s_1.

Assume that there exists an *erasing process* that can distinguish between edges made by different $\Pi^{(i)}$. Let $E^t_{u,i}$ be the set of activated edges of a node u at time t that were formed during the execution of $\Pi^{(i)}$. As the memory of each node is constant, then δ is also a constant number. In addition, the number of faults that can occur is unbounded, thus after δ faults at time t a node must execute a $\Pi^{(i)}$ that was executed in a previous step. However, $E^t_{u,i}$ might not be empty and then the *erasing process* will fail. $\square$

In light of the impossibility result of Theorem 4.31, we allow the nodes to use non-constant local memory in order to develop a fault tolerating procedure based on restart.

We give a protocol that restarts any protocol Π as follows. All nodes are initially leaders. Through a standard pairwise leader elimination procedure, a unique leader would be guaranteed to remain in the absence of failures. But because a fault can remove the last remaining leader, the protocol handles this by generating a new leader upon getting a fault notification. This guarantees the existence of at least one leader in the population and eventually (after the last fault) of a unique one. There are two main events that trigger a new restarting phase: a fault and a leader elimination. As any new event must trigger a new restarting phase that will not interfere with an outdated one, eventually overriding the latter and restarting all nodes once more, we use phase counters to distinguish among phases. In the presence of a new event it is always guaranteed that a leader at maximum phase will eventually increase its phase, therefore a restart is guaranteed after any event. The restarts essentially cause gradual deactivation of edges (by having nodes remember their degree throughout) and restoration of nodes' states to q_0, thus executing Π

on a fresh initial configuration. For the sake of clarity, we first present a simplified version of the restart protocol that guarantees resetting the state of every node to a uniform initial state q_0. So, for the time being we may assume that the protocol to be restarted through composition is any Population Protocol Π that always starts from the uniform q_0 initial configuration (all $u \in V$ in q_0 initially). Later on we shall extend this to handle with protocols that are Network Constructors instead.

Description of the PP Restarting Protocol. The state of every node consists of two components S_1 and S_2. S_1 runs the restarting protocol A while S_2 runs the given PP Π. In general, they run in parallel with the only exception when A restarts Π. The S_1 component of every node stores a *leader* variable, taking values from $\{l, f\}$, and is initially l, a *phase* variable, taking values from $\mathbb{N}_{\geq 0}$, initially 0, and a *fault* binary flag, initially 0.

The transition function is as follows. We denote by $x(u)$ the value of variable x of node u and $x'(u)$ the value of it after the transition under consideration.

If a leaders' fault flag becomes 1 or 2, it sets it to 0, increases its phase by one, and restarts Π. If a followers' flag becomes 1 or 2, it sets it to 0, increases its phase by one, becomes a leader, and restarts Π. We now distinguish three types of interactions.

When a leader u interacts with a leader v, one of them remains leader (state l) and the other becomes a follower (state f), both set their phase variable to $max\{\text{phase}(u), \text{phase}(v)\} + 1$ and both reset their S_2 component (protocol Π) to q_0 (i.e., restart Π).

When a leader u interacts with a follower v, if $\text{phase}(u) = \text{phase}(v)$, do nothing in S_1 but execute a transition of Π (both u and v involved). If $\text{phase}(u) < \text{phase}(v)$, then both set their phase variable to $max\{\text{phase}(u), \text{phase}(v)\} + 1$ and both restart Π, and finally, if $\text{phase}(u) > \text{phase}(v)$, then $\text{phase}'(v) = \text{phase}(u)$ and v restarts Π.

When a follower u interacts with a follower v, if $\text{phase}(u) = \text{phase}(v)$ do nothing in S_1 but execute transition of Π. If $\text{phase}(u) > \text{phase}(v)$, then v sets $\text{phase}'(v) = \text{phase}(u)$ and v restarts Π, and finally, if $\text{phase}(u) < \text{phase}(v)$, then u sets $\text{phase}'(u) = \text{phase}(v)$ and u restarts Π.

We now show that given any such PP Π, the above restart protocol A when composed as described with Π, gives a fault-tolerant version of Π (tolerating any number of crash faults).

Lemma 4.32 (Leader Election)**.** *In every execution of A, a configuration C with a unique leader is reached, such that no subsequent configuration violates this property.*

Proof. If after the last fault there is still at least one leader, then from that point on at least one more leader appears (due to the fault flags) and only pairwise eliminations

can decrease the number of leaders. But pairwise elimination guarantees eventual stabilization to a unique leader. It remains to show that there must be at least one leader after the last fault. The leader state becomes absent from the population only when a unique leader crashes. This generates a notification, raising at least one follower's fault flag, thus introducing at least one leader. □

Call a *leader-event* any interaction that changes the number of leaders. Observe that after the last leader-event in an execution there is a stable unique leader u_l.

Lemma 4.33 (Final Restart)**.** *On or after the last leader-event, u_l will go to a phase such that phase(u_l) > phase(u), $\forall u \in V' \setminus \{u_l\}$, where V' denotes the remaining nodes after the crash faults. As soon as this happens for the first time, let S denote the set of nodes that have restarted Π exactly once on or after that event. Then $\forall u \in V' \setminus S$, $v \in S$, an interaction between u and v results in $S \leftarrow S \cup \{u\}$. Thus, S will eventually be $S = V'$.*

Proof. We first show that on or after the last leader-event there will be a configuration in which $\text{phase}(u_l) > \text{phase}(u)$, $\forall u \in V' \setminus \{u_l\}$ and it is stable. As there is a unique leader u_l and follower-to-follower interactions do not increase the maximum phase within the followers population, u_l will eventually interact with a node that is in the maximum phase. At that point it will set its phase to that maximum plus one and we can agree that before that follower also sets its own phase during that interaction to the new max, it has been satisfied that $\text{phase}(u_l) > \text{phase}(u)$, $\forall u \in V' \setminus \{u_l\}$.

When the above is first satisfied, $S = \{u_l, u\}$ and $\text{phase}(u_l) = \text{phase}(u) > \text{phase}(v)$, $\forall v \in V' \setminus S$. Any interaction within S, only executes a normal transition of Π, as in S they are all in the same phase. Any interaction between a $u \in V' \setminus S$ and a $v \in S$, results in $S \leftarrow S \cup \{u\}$, because interactions between followers in $V' \setminus S$ cannot increase the maximum phase within $V' \setminus S$, thus $\text{phase}(v) > \text{phase}(u)$ holds and the transition is: $\text{phase}'(u) = \text{phase}(v)$ and u restarts Π, thus enters S. It follows that S cannot decrease and any interaction between the two sets increases S, thus S eventually becomes equal to V'. □

Putting Lemma 4.32 and Lemma 4.33 together gives the aforementioned result.

Theorem 4.34. *For any such PP Π, it holds that $A \circ \Pi$ is a fault-tolerant version of Π.*

Lemma 4.35. *The required memory in each agent for executing protocol A is $O(\log n)$ bits.*

Proof. Initially all nodes are potential leaders, and they eliminate each other, moving to next phases at the same time. In the worst case, a single leader u will eliminate

every other leader, turning them into followers, thus in a failure-free setting the phase of u becomes at most $n-1$. If we consider the case where crash faults may occur, each fault can result in notifying the whole population. This will happen if u was adjacent to every other node by the time it crashed. Thus, all nodes increase their phase by one and become leaders again. In the worst case, a single leader eliminates all the other leaders, thus, after the first fault, the maximum phase will be increased by $n-2$. The maximum phase than can be reached is $\sum_{i=0}^{k}(n-i) = O(kn)$, where k is the maximum number of faults that may occur ($k < n$). Thus, each node is required to have $O(\log n)$ bits of memory. □

NET Restarting Protocol (with Notifications). We are now extending the *PP Restarting Protocol* in order to handle any NET protocol Π (with notifications). Call this new protocol B. We store in the S_1 component of each node $u \in V$ a *degree* variable, that is, whenever a connection is formed or deleted, u increases or decreases the value of *degree* by one respectively. In addition, whenever the *fault flag* of a node u becomes one, it means that an adjacent node of it has crashed, thus it decreases *degree* by one. In the case of Network Constructors, the nodes cannot instantly restart the protocol Π by setting their state to the initial one q_0. By Theorem 4.31, it is evident that we first need to remove all the edges in order to have a successful restart and eventually stabilize to a correct network.

We now define an intermediate phase, called *Restarting Phase R*, where the nodes that need to be restarted enter by setting the value of a variable *restart* to 1 (stored in the S_1 component). As long as their degree is more than zero, they do not apply the rules of the protocol Π in their second component S_2, but instead they deactivate their edges one by one. Eventually their degree reaches zero, and then they set *restart* to 0 and continue executing protocol Π. We can say that a node u, which is in phase i (phase$(u) = i$), becomes available for interactions of Π (in S_2) only after a successful restart. This guarantees that a node u will not start executing the protocol Π again, unless its degree firstly reaches zero.

The additional Restarting Phase does not interfere with the execution of the *PP Restarting Protocol*, but it only adds a delay on the stabilization time.

Lemma 4.36. *The variable degree of a node u always stores its correct degree.*

Proof. In a failure-free setting, whenever a node u forms a new connection, it increases its *degree* variable by one, and whenever it deactivates a connection, it decreases it by one. In case of a fault, all the adjacent nodes are notified, as their *fault flag* becomes one. Thus, they decrease their *degree* by one. In case of a fault with no adjacent nodes, a random node is notified, and its *fault flag* becomes two. In that case, it leaves the value of *degree* the same. □

Theorem 4.37. *For any NET protocol* Π *(with notifications), it holds that* $B \circ \Pi$ *is a fault-tolerant version of* Π.

Proof. Consider the case where a node u (either leader or follower) needs to be restarted. It enters to the restarting phase in order to deactivate all of its enabled connections, and it will start executing Π only after its degree becomes zero (by Lemma 4.36 this will happen correctly), thus, Π always runs in nodes with no spurious edges (edges that are the result of previous executions). Whenever two connected nodes $u \in R$ and $v \notin R$, where R is the *Restarting Phase*, interact with each other, they both decrease their *degree* variable by one, and they delete the edge joining them. Obviously, this fact interferes with the execution of Π in node v (which is not in the restarting phase), but v is surely in a previous phase than u and will eventually also enter in R. This follows from the fact that a node in some phase i can never start forming new edges before it has successfully deleted all of its edges before. New edges are only formed with nodes in the same phase i.

The new *Restarting Phase* does not interfere with the states of the *PP Restarting Protocol*, thus the correctness of B follows by Lemma 4.32 and Lemma 4.33. □

Lemma 4.38. *The required memory in each agent for executing protocol* B *is* $O(\log n)$ *bits.*

Proof. The maximum value that the variable *degree* can reach is the *maximum reachable degree* (d) of protocol Π. Thus, by Lemma 4.35, the states that each node is required to have is $O(dkn)$. Both d and k are less that $n - 1$, thus, $O(n^3)$ states $= O(\log n)$ bits. □

4.5 Conclusions and Further Research

A number of interesting problems are left open for future work. Our only exact characterization was achieved in the case of unbounded faults and no notifications. If faults are bounded, non-hereditary languages were proved impossible to construct without notifications but we do not know whether all hereditary languages are constructible. Relaxations, such as permitting waste or partial constructibility were shown to enable otherwise impossible transformations, but there is still work to be done to completely characterize these cases. In case of notifications, we managed to obtain fault-tolerant universal constructors, but it is not yet clear whether the assumptions of waste and local coin tossing that we employed are necessary and how they could be dropped. Finally, in Section 4.4.3 we showed a protocol that restarts the population whenever a fault occurs, and to achieve it we empowered the agents with $O(\log n)$ memory. An immediate question here is whether we can achieve the same results by simulating the algorithm of [87] to handle crash failures. Apart from

these immediate technical open problems, some more general related directions are the examination of different types of faults such as random, Byzantine, and communication/edge faults. Finally, a major open front is the examination of fault-tolerant protocols for stable dynamic networks in models stronger than NETs.

Chapter 5

Crystal Structure Prediction via Oblivious Local Search

We study Crystal Structure Prediction, one of the major problems in computational chemistry. This is essentially a continuous optimization problem, where many different, simple and sophisticated, methods have been proposed and applied. The simple searching techniques are easy to understand, usually easy to implement, but they can be slow in practice. On the other hand, the more sophisticated approaches perform well in general, however almost all of them have a large number of parameters that require fine tuning and, in the majority of the cases, chemical expertise is needed in order to properly set them up. In addition, due to the chemical expertise involved in the parameter-tuning, these approaches can be *biased* towards previously-known crystal structures. Our contribution is twofold. Firstly, we formalize the Crystal Structure Prediction problem, alongside several other intermediate problems, from a theoretical computer science perspective. Secondly, we propose an oblivious algorithm for Crystal Structure Prediction that is based on local search. Oblivious means that our algorithm requires minimal knowledge about the composition we are trying to compute a crystal structure for. In addition, our algorithm can be used as an intermediate step by *any* method. Our experiments show that our algorithms outperform the standard basin hopping, a well studied algorithm for the problem.

5.1 Introduction

The discovery of new materials has historically been made by experimental investigation guided by chemical understanding. This approach can be both time consuming and challenging because of the large space to be explored. For example, a "traditional" method for discovering inorganic solid structures relies on knowledge of crystal chemistry coupled with repeating synthesis experiments and systematically varying

elemental ratios, each of which can take lots of time [141, 144]. As a result there is a very large unexplored space of chemical systems: only 72% of binary systems, 16% of ternary, and just 0.6% of quaternary systems have been studied experimentally [145]. These inefficiencies forced physical scientists to develop computational approaches in order to tackle the problem of finding new materials. The first approach is based on data mining where *only* pre-existing knowledge is used [39, 80, 88, 123, 138]. Although this approach has proven to be successful, there is the underlying risk of missing best-in-class materials by being biased towards *known* crystal structures. Hence, the second approach tries to fill this gap and aims at finding new materials with *little, or no,* pre-existing knowledge, by *predicting the crystal structure* of the material. This approach has led to the discovery of several new, counterintuitive, materials whose existence could not be deduced by the structures of previously-known materials [37].

Several heuristic methods have been suggested for crystal structure prediction. All these methods are based on the same fundamental principle. Every arrangement of ions in the 3-dimensional Euclidean space corresponds to an energy value and it defines a point on the *potential energy surface.* Then, the crystal structure prediction problem is formulated as a *mathematical optimization* problem where the goal is to compute the structure that corresponds to the global minimum of the potential energy surface, since this is the most likely structure that corresponds to a stable material. The difficulties in solving this optimization problem is that the potential energy surface is *highly non convex*, with *exponentially many*, with respect to the number of ions, local minima [125]. For this reason, several different algorithmic techniques were proposed ranging from simple techniques, like *quasi-random sampling* [76, 132, 133, 137], *basin hopping* [82, 147], and *simulated annealing* techniques [128, 139], to more sophisticated techniques, like *evolutionary and genetic algorithms* [29, 50, 99, 124, 150], and *tiling* approaches [36, 37]. A recent comprehensive review on these techniques can be found in [125].

The simple searching techniques are easy to understand, usually easy to implement, and they are *unbiased*, but they can be slow in practice. On the other hand, the more sophisticated approaches perform well in general, however almost all of them have a large number of parameters that require fine tuning and, in the majority of the cases, chemical expertise is needed in order to properly set them up. In addition, due to the chemical expertise involved in the parameter-tuning, these approaches can be *biased* towards previously-known structures.

The majority of the aforementioned heuristic techniques work, at a very high level, in a similar way. Given a current solution x for the crystal structure prediction problem, i.e., a location for every ion in the 3-dimensional space, they iteratively perform the following three steps.

1. Choose a new potential solution x'. This can be done by taking into account, or modifying, x.
2. Perform gradient descent on the potential energy surface starting from x', until a local minimum is found. This process is called *relaxation* of x'.
3. Decide whether to keep x as the candidate solution or to update it to the solution found after relaxing x'.

For example, basin hopping algorithms randomly choose x', they relax x' and if the energy of the relaxed structure is lower than the x, or a Metropolis criterion is satisfied, they accept this as a current solution; else they keep x and they randomly choose x''. The procedure usually stops when the algorithm fails to find a structure with lower energy within a predefined number of iterations. The more sophisticated algorithms take into account knowledge harvested from chemists and put constraints on the way x' is selected. For example, the MC-EMMA [37] and the FUSE [36] algorithms use a set of building blocks to construct x'. These building blocks are local configurations of ions that are present in, or similar to, known crystal structures. These approaches restrict the search space ,which accelerate search, but reduce the number of possible solutions.

This general algorithm is easy to understand, however there are some hidden difficulties that make the problem more challenging. Firstly, it is not trivial even how to *evaluate* the potential energy of a structure. There are several different methods for calculating the energy of a structure, ranging from *quantum mechanical* methods, like *density functional theory* [1], to *force fields* methods [2], like the *Buckingham-Coulomb* potential function. All of which though, are hard to compute (see Section 5.2.1) from the point of view of (theoretical) computer science and thus only numerical methods are known and used in practice for them [77]; still there are cases where some methods need considerable time to calculate the energy of a structure. This yields another, more important, difficulty, the relaxation of a structure. Since it is hard to compute the energy of a structure, it is even harder to apply gradient descent on the potential energy surface. For these reasons, the majority of the heuristic algorithms depend on *external*, well established, codes [77] for computing the aforementioned quantities. Put differently, both energy computations and relaxations of structures are treated as *oracles* or *black boxes*.

5.1.1 Contribution and roadmap for the chapter

Our contribution is twofold. Firstly, in Section 5.2 we formalize the Crystal Structure Prediction problem from the theoretical computer science perspective; to the best of

[1] `https://en.wikipedia.org/wiki/Density_functional_theory`
[2] `https://en.wikipedia.org/wiki/Force_field_(chemistry)`

our knowledge, this is among the few papers that attempt to connect computational chemistry and computer science. En route to this, in Section 5.2.2 we introduce several intermediate open problems from computational chemistry in CS terms. Any (partial) positive solution to these questions can significantly help computational chemists to identify new materials. On the other hand, any negative result can formally explain why the discovery of new materials is a notoriously difficult task.

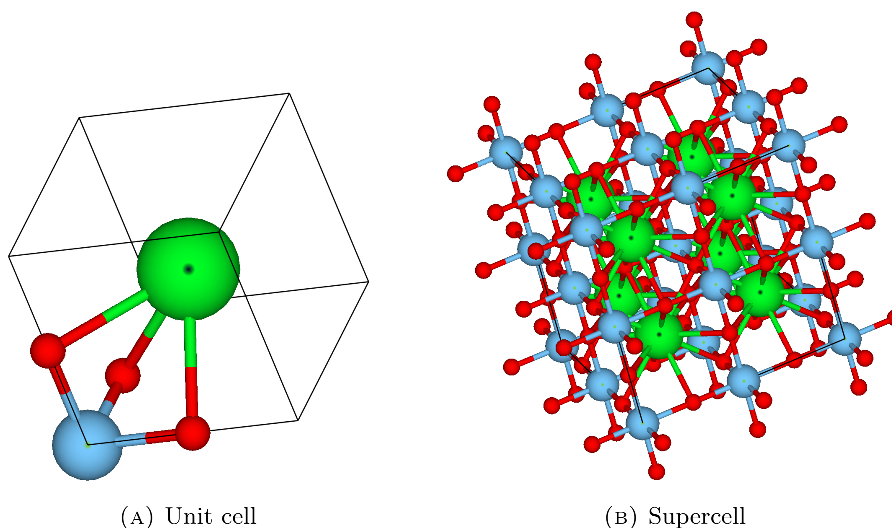
Our second contribution is the partial answer for some of the questions we cast. In general, our goal is to create *oblivious* algorithms that are easy to implement, they are fast, and they work well in practice. With oblivious we mean that we are seeking for *general procedures* that require *minimal input* and they have zero, or just a few, parameters chosen by the user.

- In Section 5.4 we propose a purely combinatorial method for estimating the energy of a structure, which we term *depth energy computation.* We choose to compare our method against GULP [77], which is considered to be the state of the art for computing the energy of a structure and for performing relaxations when the Buckingham-Coulomb energy is used. Our method requires only the charges of the atoms and their corresponding Buckingham coefficients to work; see Eq. 5.2 in Section 5.2.1. In addition, it needs only one parameter, the depth k. We experimentally demonstrate that our method monotonically approximates with respect to k the energy computed by GULP and that it achieves an error of 0.0032 for $k = 6$. Our experiments (Section 5.5) show that the structure that achieves the minimum energy in depth 1 is likely to be the structure with the minimum energy overall. In fact we show something much stronger. If the energy of x is lower than the energy of x' when it is computed via the depth energy computation for $k = 1$, then, almost always, the energy of x will be lower than the energy of x' when it is computed via GULP.

- We derive oblivious algorithms for choosing which structure to relax next. All of our algorithms are based on local search. More formally, starting with x and using only local changes we select x'. In Section 5.3 we define several "combinatorial neighborhoods" and we evaluate their efficiency. Our neighborhoods are oblivious since they only need access to an oracle that calculates the energy of a structure. We show that our method outperforms basin hopping. Moreover, we view our algorithms as an intermediate step before relaxation that can be applied to *any* existing algorithm.

5.2 Preliminaries

A *crystal* is a solid material whose atoms are arranged in a highly ordered configuration, forming a *crystal structure* that extends in all directions. A crystal structure is characterized by its *unit cell*; a parallelepiped that contains atoms in a specific arrangement. The unit cell is the *period* of the crystal; unit cells are stacked in the three dimensional space to form the crystal. In this paper we focus on *ionically bonded crystals*, which we describe next; what follows is relevant only on crystals of this type. In order to fully define the unit cell of a ionically bonded crystal structure, we have to specify a *composition*, *unit cell parameters*, and an *arrangement* of the ions.



(A) Unit cell (B) Supercell

FIGURE 5.1: Most stable configuration of $SrTiO_3$

Composition A composition is the chemical formula that describes the ratio of ions that belong to the unit cell. The chemical formula contains *anions*, negatively charged ions, and *cations*, positively charged ions. The chemical formula is a way of presenting information about the *chemical proportions* of ions that constitute a particular chemical compound, and it does not provide any information about the exact number of atoms in the unit cell.

More formally, the composition is defined by a set of distinct chemical elements $\{e_1, e_2, \ldots, e_m\}$, their multiplicity n_i, and a non-zero integer charge q_i for each element i. The number m denotes the total number of distinct chemical elements, and $n_i / \sum_{j=1}^{m} n_j$ is the proportion of the atoms of type e_i in the unit cell. It is required that the sum of the charges adds up to zero, i.e. , $\sum_{i=1}^{m} q_i n_i = 0$, so that the unit cell is charge neutral. For example, the composition for Strontium Titanate, $SrTiO_3$,

denotes that the following hold. For every ion of strontium (Sr) in the unit cell, there exists one ion of titanium (Ti) and three ions of oxygen (O). Furthermore, the charge of every ion of Sr is +2, of Ti is +4, and of O is −2. Hence, when the ratios of the ions are according to the composition, the charge of the unit cell is zero.

Another parameter for every atom is the *atomic radius*. This usually corresponds to the distance from the center of the nucleus to the boundary of the surrounding shells of electrons. Since the boundary is not a well-defined physical entity, there are various non-equivalent definitions of atomic radius. In crystal structures though, the *ionic radius* is used and usually is treated as a hard sphere. Thus, we will use ρ_i to denote the ionic radius of the element e_i.

Unit cell parameters. Unit cell parameters provide a formal description of the parallelepiped that represents the unit cell. These include the lengths y_1, y_2, y_3 of the parallelepiped in every dimension and the angles θ_{12}, θ_{13}, and θ_{23} between the corresponding facets. For brevity, we denote $y = (y_1, y_2, y_3)$ and $\theta = (\theta_{12}, \theta_{13}, \theta_{23})$, and we use (y, θ) to denote the unit cell parameters.

Arrangement An arrangement describes the position of each atom of the composition in the unit cell. The position of ion i is specified by a point $x_i = (x_{i1}, x_{i2}, x_{i3})$ in the parallelepiped defined by the unit cell parameters; fractional coordinates x_i denote the location of the nucleus of the ion i in the unit cell. A unit cell parameters-arrangement combination (y, θ, x) in a unit cell with n ions is a point in the $3n + 6$-dimensional space. For any two points x_i and x_j we will use $\mathtt{d}(x_i, x_j)$ to denote their Euclidean distance.

As we have already said, a unit cell parameters-arrangement configuration (y, θ, x) defines the period of an infinite structure that covers the whole 3d space. To get some intuition, assume that we have an orthogonal unit cell, i.e., all the angles are 90 degrees. Then for every ion with position (x_{i1}, x_{i2}, x_{i3}) in the unit cell, there exist "copies" of the ion in the positions $(k_1 \cdot y_1 + x_{i1}, k_2 \cdot y_2 + x_{i2}, k_3 \cdot y_3 + x_{i3})$ for every possible combination of integers k_1, k_2, and k_3. A unit cell parameters-arrangement configuration is *feasible* if the hard spheres of any two ions of the crystal structure do not overlap; formally, it is feasible if for every two ions i and j it holds that $\mathtt{d}(x_i, x_j) \geq \rho_i + \rho_j$.

Definition 5.1 (Crystal Structure)**.** A crystal structure is characterized by a composition and a unit cell parameters-arrangement configuration (y, θ, x), as follows:

1. *Composition*: a set of m distinct chemical elements $\{e_1, e_2, \ldots, e_m\}$, their multiplicity n_i, and a non-zero integer charge q_i for each element i, such that $\sum_{i=1}^{m} q_i n_i = 0$.

2. *Unit cell parameters*: lengths of the parallelepiped $y = (y_1, y_2, y_3)$, where y_j is the length of the unit cell along the j-axis, and the angles $\theta = (\theta_{12}, \theta_{13}, \theta_{23})$ between them.

3. *Arrangement of ions*: for each ion i defined by the composition, (x_{i1}, x_{i2}, x_{i3}) are its fractional coordinates, such that $0 \leq x_{ij} \leq 1$. In other words, each x_{ij} defines the position of ion i along the j-axis of the unit cell.

5.2.1 Energy

Any unit cell parameters-arrangement configuration of a composition corresponds to a *potential energy*. When the number of ions in the unit cell is fixed, the set of configurations define the *potential energy surface*.

Buckingham-Coulomb potential is among the most well adopted methods for computing energy [27, 148] and it is the sum of the Buckingham potential and the Coulomb potential. The Coulomb potential is *long-range* and depends only on the charges and the distance between the ions; for a pair of ions i and j, the Coulomb energy is defined by

$$CE(i, j) := \frac{q_i q_j}{\mathtt{d}(x_i, x_j)}. \tag{5.1}$$

Note, ions i and j can be in *different* unit cells.

The Buckingham potential is *short-range* and depends on the species of the ions and their distance. More formally, it depends on positive composition-dependent constants A_{e_i,e_j}, B_{e_i,e_j}, and C_{e_i,e_j} for every pair of species e_i and e_j; here i can be equal to j [3]. So, for the pair of ions i, of specie e_i, and j, of specie e_j, the Buckingham energy is

$$BE(i, j) := A_{e_i,e_j} \cdot \exp(-B_{e_i,e_j} \cdot \mathtt{d}(x_i, x_j)) - \frac{C_{e_i,e_j}}{\mathtt{d}(x_i, x_j)^6}. \tag{5.2}$$

Again, ions i and j can be in different unit cells.

Let $S(x_i, \rho)$ denote the sphere with center x_i and radius ρ. The total energy of a crystal structure whose unit cell is characterized by n ions with arrangement

[3] The Buckingham constants are composition-depended since they can have small discrepancies in different compositions. For example the constants $A_{\mathrm{Ti,O}}, B_{\mathrm{Ti,O}}$, and $C_{\mathrm{Ti,O}}$ for $SrTiO_3$ can be different than those for $MgTiO_3$. There is a long line of research in computational chemistry that tries to learn/estimate the Buckingham constants for various compositions. In addition, more than one set of Buckingham constants can be available for a given composition.

$x = (x_1, \dots, x_n)$ is then defined

$$E(y, \theta, x) = \lim_{\rho \to \infty} \sum_{i=1}^{n} \sum_{j \neq i, j \in S(x_i, \rho)} \left(BE(i,j) + CE(i,j) \right).$$

$E(y, \theta, x)$ *conditionally converges* to a certain value [131] and usually numerical approaches are used to compute it. For this reason, and since we aim for an oblivious algorithm, we view the computation of the energy of a structure as a *black box*. More specifically, we assume that we have an oracle that given any structure (y, θ, x), it returns its corresponding energy.

Open Question 1. Given a composition and Buckingham parameters for it, find a simple, purely combinatorial way that approximates the energy for every crystal structure.

Open Question 2. Given a composition $\{e_1, e_2, \dots, e_m\}$ and an oracle that computes the energy of every structure for this composition, learn efficiently (with respect to the number of oracle calls) the Buckingham parameters A_{e_i,e_j}, B_{e_i,e_j}, and C_{e_i,e_j} for every $i, j \in [m]$.

Relaxation The *relaxation* of a crystal structure (y, θ, x) computes a stationary point on the potential energy surface by applying gradient descent starting from (y, θ, x). The relaxation of a structure can change *both* the arrangement x of the ions in the unit cell *and* the unit cell parameters (y, θ) of the unit cell. We follow a similar approach as we did with the energy and we assume that there is an oracle that given a crystal structure (y, θ, x) it returns the relaxed structure.

Open Question 3. Find an alternative, quicker, way to compute an approximate local minimum when:

a) the unit cell parameters (y, θ) of the unit cell are fixed;

b) the arrangement x of the ions is fixed;

c) both unit cell parameters and arrangement are free.

5.2.2 Crystal structure prediction problems

In crystal structure prediction problems the general goal is to minimize the energy in the unit cell. There are two kinds of problems we are concerned. The first cares only about the value of the energy and the second one cares for the arrangement and the unit cell parameters that achieve the minimum energy. From the computational chemistry point of view, both questions are interesting in their own right. The

existence of a unit cell parameters-arrangement that achieves lower-than-currently-known energy usually suffices for constructing a new material. On the other hand, identifying the arrangement and the unit cell parameters of a crystal structure that achieves the lowest possible energy can help physical scientists to predict the properties of the material.

MinEnergy

Input: A composition (as defined in Definition 5.1) with its corresponding Buckingham constants, a positive integer n, and a rational $\hat{E}$.
Question: Is there a crystal structure (y, θ, x) for the composition with n ions that is neutrally charged and achieves Buckingham-Coulomb energy $E(y, \theta, x) < \hat{E}$?

MinStructure

Input: A composition and a positive integer n, such that $\sum_{i=1}^{m} n_i = n$, and the corresponding Buckingham constants.
Task: Find a crystal structure (y, θ, x) for the composition with n ions that is neutrally charged and the Buckingham-Coulomb energy $E(y, \theta, x)$ is minimized.

The second class of problems, the ones that ultimately computational chemists would like to solve, take as input only the composition and the goal is to construct a unit cell, with any number of atoms, such that the *average energy per ion* is minimized.

AvgEnergy

Input: A composition with its corresponding Buckingham constants and a rational $\hat{E}$.
Question: Is there a crystal structure for the composition that is neutrally charged and $\frac{E(y,\theta,x)}{n} < \hat{E}$?

AvgStructure

Input: A composition with its corresponding Buckingham constants.
Task: Find a crystal structure for the composition that is neutrally charged and the average Buckingham-Coulomb energy per ion in the unit cell, $\frac{E(y,\theta,x)}{n}$, is minimized.

Although the problems are considered to be intractable [125], only recently the first *correct* NP-hardness result was proven for a variant of CSP [1]. However, for the problems presented, there are no correct NP-hardness results in the literature.

Open Question 4. Provide provable lower bounds and upper bounds for the four problems defined above.

Open Question 5. Construct a heuristic algorithm that works well in practice.

5.3 Local Search

Local search algorithms start from a feasible solution and iteratively obtain better solutions. The key concept for the success of such algorithms, is given a feasible solution, to be able to *efficiently find* an improved one. Put formally, a local search algorithm is defined by a *neighborhood function* N and a *local rule* r. In every iteration, the algorithm does the following.

- Has the current best solution x.
- Computes the neighborhood $N(x)$.
- If there is an improved solution x' in $N(x)$, then it updates x according to the rule r, i.e. $x' = r(N(x))$; else it terminates and outputs x.

The neighborhood $N(x)$ of a solution x consists of all feasible solutions that are "close" in some sense to x. The size of the neighborhood can be constant or a function of the input. In principle, the larger the size of the neighborhood, the better the quality of the locally optimal solutions. However, the downside of choosing large neighborhoods is that, in general, it makes each iteration computationally more expensive. Running time and quality of solutions are competing considerations, and the trade off between them can be determined through experimentation.

We study the following *combinatorial* neighborhoods for Crystal Structure Prediction. All of them keep the unit cell parameters fixed and change only the arrangement x of the n ions. Thus, for notation brevity, we define the neighborhoods only with respect to the arrangement x.

1. **k-ion swap.** This neighborhood consists of all feasible arrangements that are produced by swapping the locations of k ions. The size of this neighborhood is $O(n^k k!)$.

2. **k-swap.** This neighborhood is parameterized by a discretization step δ. Using δ we discretize the unit cell and then we perform swaps of k ions with the content of every point of the discretization. So, an ion can swap positions with

another ion, or simply move to another vacant position. Again, we take into account only the feasible arrangements of the ions. The size of this neighborhood is $O(n^k k!/\delta^{3k})$.

3. **Axes.** This neighborhood has a parameter δ and computes the following for every ion i. Firstly, for every dimension it computes a plane parallel to the corresponding facet of the unit cell and contains the ion i. The intersection of any pair of these planes defines an "axis". Then, this axis is discretized according to δ. The neighborhood locates the ion to every point on the discretization on the three axes and we keep only the arrangements that are feasible. The size of this neighborhood is $O(n/\delta)$.

In all of our neighborhoods, we are using a *greedy* rule to choose x'; x' is an arrangement that achieves the minimum energy in $N(x)$.

5.4 Algorithms

We propose two algorithms. The first one is a step towards answering Open Question 1 while the second is a heuristic for MinStructure problem.

For Open Question 1, we propose the *depth energy computation* for estimating the energy of a structure. Our algorithm has a single parameter, the depth parameter k, and works as follows. Given a crystal structure, it creates k layers around the unit cell with copies of the structure. So, for the unit cell parameters (y, θ) and the arrangement x of n atoms the energy is $E(y, \theta, x) = \sum_{i=1}^{n} \sum_{j \neq i, j \in D(k)} (BE(i,j) + CE(i,j))$, where $D(k)$ denotes the set of ions in the k layers of unit cells, and $BE(i,j)$ and $CE(i,j)$ are computed as in Equations 5.2 and 5.1 respectively.

For MinStructure problem, we slightly modify basin hopping. In a step of basing hopping, a structure is randomly chosen and it is followed by a relaxation. Our algorithm applies a combinatorial local search using the Axes neighborhood, since this turned out to be the best among our heuristics, before the relaxation. So, we will perform a relaxation, *only after* combinatorial local search cannot further improve the solution. Our algorithm can be used as a standalone one and it can also be integrated into *any other* heuristic algorithm for the Crystal Structure Prediction problem since it is oblivious. In addition, it provides a very fast criterion that when it succeeds it guarantees finding a lower energy crystal structure.

5.5 Experiments

In this section we evaluate our algorithms via experimental simulations. We first focus on $SrTiO_3$ which we use as a benchmark. We do this because it is a well studied composition for which the Crystal Structure Prediction problem is solved. We have implemented the algorithms in Python 2.7 and we use the Atomic Simulation Environment (`https://wiki.fysik.dtu.dk/ase/`) package for setting up, manipulating, running, visualizing and analyzing atomistic simulations. All experiments were performed on a 4-core Intel i7-4710MQ with 8GB of RAM.

	Energy difference					
k	1	2	3	4	5	6
15 atoms	0.0639	0.0226	0.0114	0.0068	0.0045	0.0032
20 atoms	0.0670	0.0238	0.0120	0.0072	0.0047	0.0033

TABLE 5.1: Comparison between depth approach and GULP for $SrTiO_3$. Energy is in electronvolts (eV). The energy difference shows the average difference in energy between the depth approach and the energy calculated by GULP (absolute value). Results averaged over 2000 random feasible structures.

We evaluate the depth energy computation in several different dimensions. For all the experiments we performed for energy computation, we fixed the unit cell to be cubic. Firstly, we evaluate how depth energy computation behaves with respect to k. We see that the method converges very fast and $k = 6$ already achieves accuracy of three decimal points. Then, we compare our depth approach against GULP; see Table 5.1. Our goal is to provide an intuitively simpler to interpret and work with method for computing the energy. Even though the energy calculated by the depth approach differs from the one calculated by GULP, we observe that the relative energies between two random arrangements remain usually the same even for $k = 1$. To be more precise, let $E_1(x)$ denote the energy of a feasible arrangement x when $k = 1$ and let $E_G(x)$ denote the energy of this arrangement as it is computed by GULP. Our experiments show that if for two random feasible arrangements x_1 and x_2 it holds that $E_1(x_1) < E_1(x_2)$, then $E_G(x_1) < E_G(x_2)$ for 99.8% of 1000 pairs of arrangements. This percentage reaches 100% for $k = 6$. For the "special" arrangement of ions x^* that minimizes the energy computed by GULP, that is $x^* = \text{argmin} E_G(x)$, our experiments show that it is *always true* that $E_k(x^*) < E_k(x)$, for every $k = 1, \ldots, 6$, where x is a random feasible structure over 10000 of them. So, this is a good indication that the arrangement that minimizes the energy for $k = 1$, also minimizes the energy overall. We view this as a striking result; it significantly simplifies the problem thus new, analytical, methods can be derived for the problem.

The next set of experiments compares the three neighborhoods described in Section 5.3 for $SrTiO_3$[4]. We compare them in several different dimensions: the average CPU time they need in order to find a local optimum with respect to their combinatorial neighborhood and the average drop in energy until they reach such a local optimum (Tables 5.2 and 5.4); the average CPU time the relaxation needs starting from such local minimum and the average drop in energy from relaxation (Tables 5.3 and 5.5). In addition, for the case of $SrTiO_3$, we compare how often we can find the optimal arrangement from a single structure. We observe that the Axes neighborhood has the best tradeoff between energy drop and CPU time. The 2-ion-swap neighborhood outperforms the other two in terms of running time, however it seems to decrease the probability of finding the best arrangement when performing a relaxation on the resulting structures. This renders the use of 2-ion-swap neighborhood inappropriate. Axes neighborhood is significantly faster and performs smoother in terms of running time than the 2-swap neighborhood. However the latter one performs better with respect to the energy drop, which is expected since axes is a subset of the 2-swap neighborhood. In addition, the relaxation from the local minimum found by 2-swap significantly improves the probability of finding the best arrangement with only one relaxation. We should highlight that there exist structures where the relaxation cannot improve their energy, but the neighborhoods do; hence using Axes neighborhood we can escape from some local minima of the continuous space.

Neighborhood	Running time	Time stdev	Energy drop	Energy drop stdev
Axes	5.36	1.54	13.46	10.60
2-ion swap	0.96	0.33	7.75	8.45
2-swap	34.66	14.06	16.21	10.94

Table 5.2: Comparison of local neighbourhoods for reaching a combinatorial minimum for $SrTiO_3$ with 15 atoms per unit cell and $\delta = 1$Å (375 grid points). Time is in seconds and energy in electronvolts (eV). Results averaged over 1000 arrangements.

Neighborhood	Running time	Time stdev	Energy drop	Energy drop stdev	Global minimum
Random structures-GULP	8.80	6.35	18.60	10.26	6.6%
Axes-GULP	7.92	6.16	5.53	2.28	10.0%
2-ion swap-GULP	8.82	6.25	11.09	5.64	4.7%
2-swap-GULP	5.14	5.08	2.79	1.05	14.8%

Table 5.3: Evaluation of relaxation procedure after using a combinatorial neighborhood for $SrTiO_3$ with 15 atoms per unit cell. Time is in seconds and energy in electronvolts (eV). Results averaged over 1000 arragnements.

[4]The values of the Buckingham parameters can be found in Table 5.7

Neighborhood	Running time	Time stdev	Energy drop	Energy drop stdev
Axes	7.56	2.08	8.23	3.71
2-ion swap	0.88	0.43	1.16	1.80
2-swap	27.93	9.98	10.26	4.05

Table 5.4: Comparison of local neighbourhoods for reaching a combinatorial minimum for $Y_2Ti_2O_7$ and $\delta = 1$Å (343 grid points). Time is in seconds and energy in electronvolts (eV). Results averaged over 1000 arrangements.

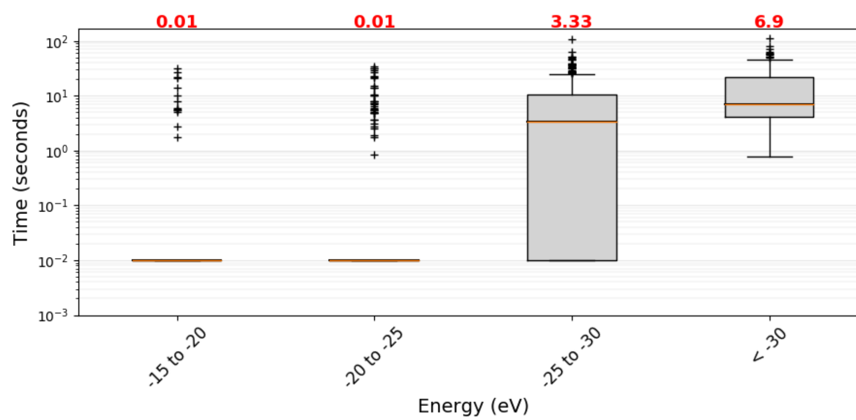
Neighborhood	Running time	Time stdev	Energy drop	Energy drop stdev
Random structures-GULP	2.81	1.45	12.84	4.88
Axes-GULP	2.30	1.20	5.09	1.81
2-ion swap-GULP	2.47	2.25	12.29	4.38
2-swap-GULP	1.99	1.09	3.11	0.93

Table 5.5: Evaluation of relaxation procedure after using a combinatorial neighborhood for $Y_2Ti_2O_7$. Time is in seconds and energy in electronvolts (eV). Results averaged over 1000 arragnements.

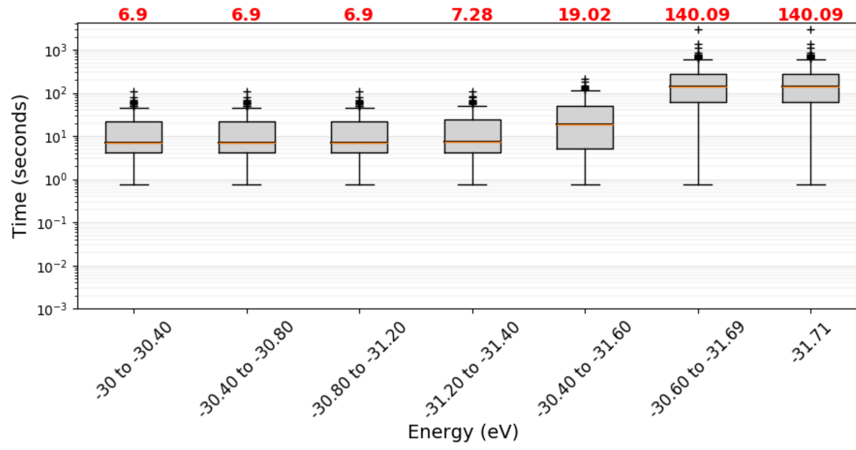
Next, we compare our algorithm for MinStructure against basin hopping where the next structure to relax is chosen at random. Based on the results of our previous experiments, we have chosen the Axes neighborhood as an intermediate step before the relaxation. We have run these algorithms 200 times for $SrTiO_3$ with 15 atoms per unit cell, and 25 times for $SrTiO_3$ with 20 atoms per unit cell. We report how the energy varies with respect to time until the best arrangement is found (Fig. 5.2) and we report other statistics that further validate our approach (Table 5.6). As we can see, it is relatively easy to reach low levels of energy and the majority of time is needed to find the absolute minimum. In addition, the overhead posed by the use of the neighborhood search divided by the time needed by the basin hopping to find the global minimum decreases as the number of the atoms in the unit cell increases.

Algorithm	Number of atoms	Total time mean	Total time stdev	Relaxations	Time for relaxations	Time for local search
Axes-GULP	15	227.89	287.21	13.24	126.26	101.63
	20	2280.57	781.66	104.33	1016.72	1049.13
Basin hopping	15	167.89	114.89	18.14	160.79	–
	20	5766.20	4748.33	450.66	4895.60	–

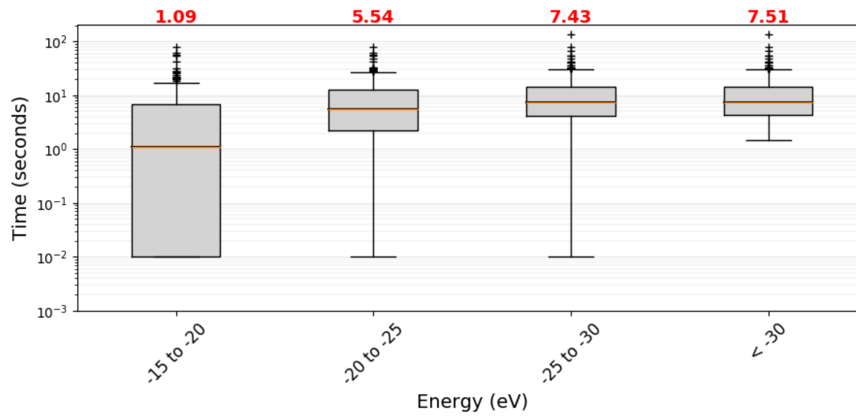
Table 5.6: Statistics from the experiments depicted in Figures 5.2 and 5.3 ($SrTiO_3$ with 15 and 20 atoms per unit cell).



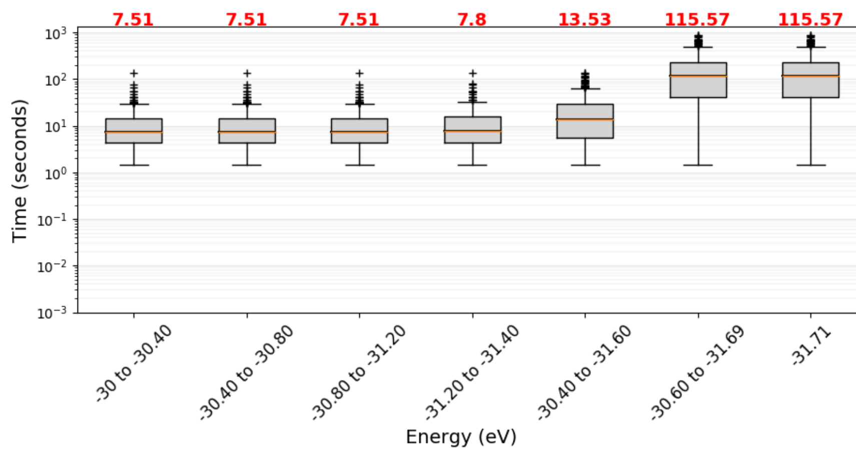
(A) Axes - GULP coarse



(B) Axes - GULP fine

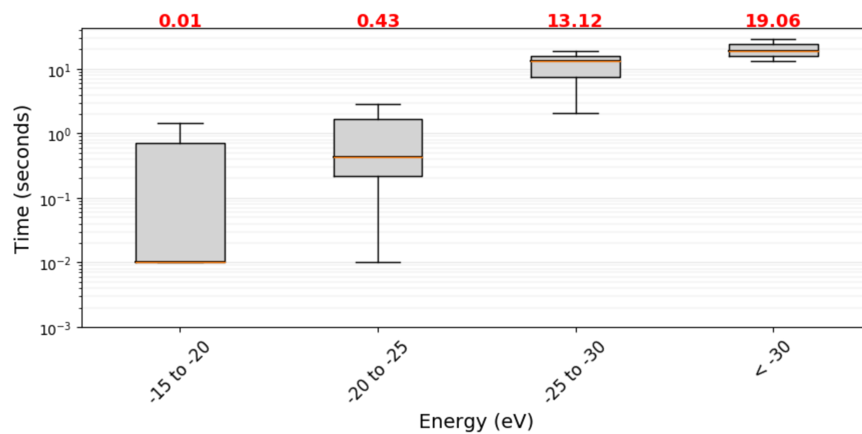


(C) GULP coarse

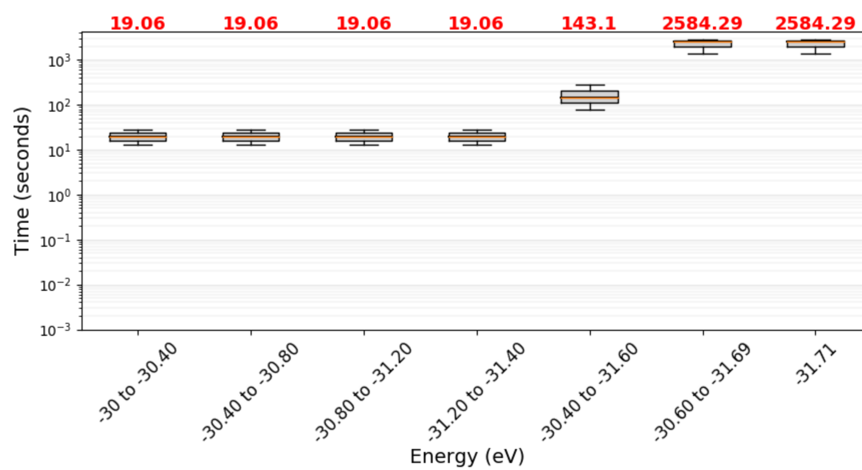


(D) GULP fine

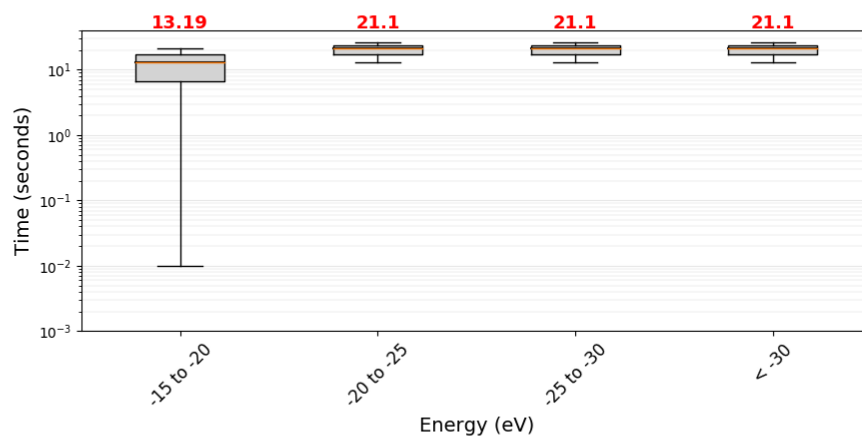
FIGURE 5.2: Time to reach specific energy levels for $SrTiO_3$ (15 atoms). Figures (a) and (b) correspond to the algorithm of Section 5.4. Figures (c) and (d) correspond to basin hopping. The median times needed to reach every energy level are depicted in red on the top of each plot.



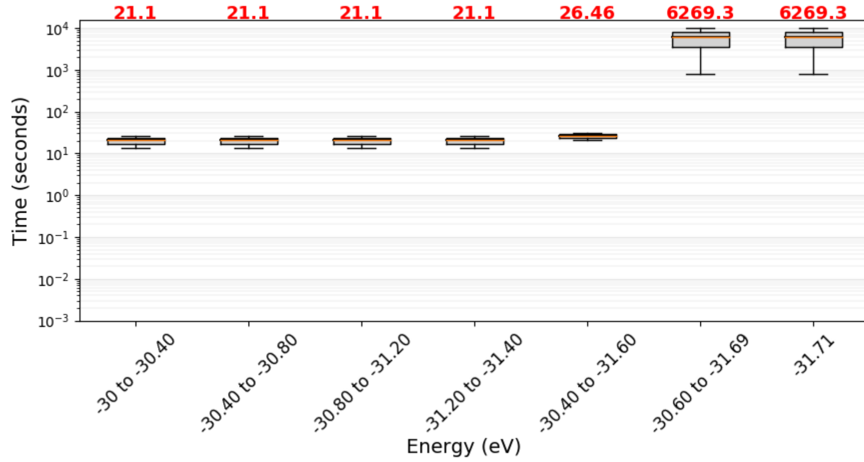
(A) Axes - GULP coarse



(B) Axes - GULP fine



(C) GULP coarse



(D) GULP fine

FIGURE 5.3: Time to reach specific energy levels for $SrTiO_3$ (20 atoms). Figures (a) and (b) correspond to the algorithm of Section 5.4. Figures (c) and (d) correspond to basin hopping. The median times needed to reach every energy level are depicted in red on the top of each plot.

In our last set of experiments, we compare our algorithm against basin hopping algorithm for $Y_2Ti_2O_7$ which contains 22 atoms in its primitive unit cell. In this set of experiments, we produced 3500 random structures. We simulated basin hopping by sequentially relaxing the constructed structures. However, none of the relaxations managed to find the optimal configuration. Our algorithm, using the same order of structures as before, first used the Axes neighborhood as an intermediate step followed by a relaxation; it managed to find the optimal configuration after visiting only 720 structures. The execution of these algorithms is depicted in Figure 5.4 and 5.5.

The Buckingham potential parameters that were used for both $Y_2Ti_2O_7$ and $SrTiO_3$ are presented in Table 5.7.

Interaction	A (eV)	ρ (Å)	C (eV Å^{-6})
$O^{2-} - O^{2-}$	1388.77	0.36262	175
$Y^{3+} - O^{2-}$	23000	0.24203	0
$Sr^{2+} - O^{2-}$	1952.39	0.33685	19.22
$Ti^{4+} - O^{2-}$	4590.7279	0.261	0

TABLE 5.7: The values of Buckingham potential parameters we used in our experiments as they were found in [36]. All the missing parameters are set to zero.

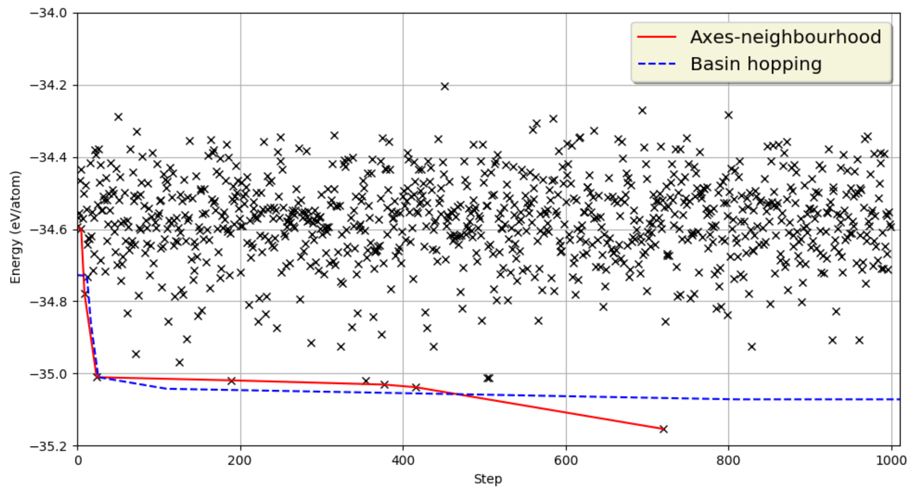


FIGURE 5.4: Performance of the Axes algorithm for $Y_2Ti_2O_7$. The black points correspond to the energy found after the relaxation of a point computed by the Axes neighborhood at each step. The red line is the lower envelope of the energy found by our algorithm while the blue line corresponds to the lower envelope of basin hopping.

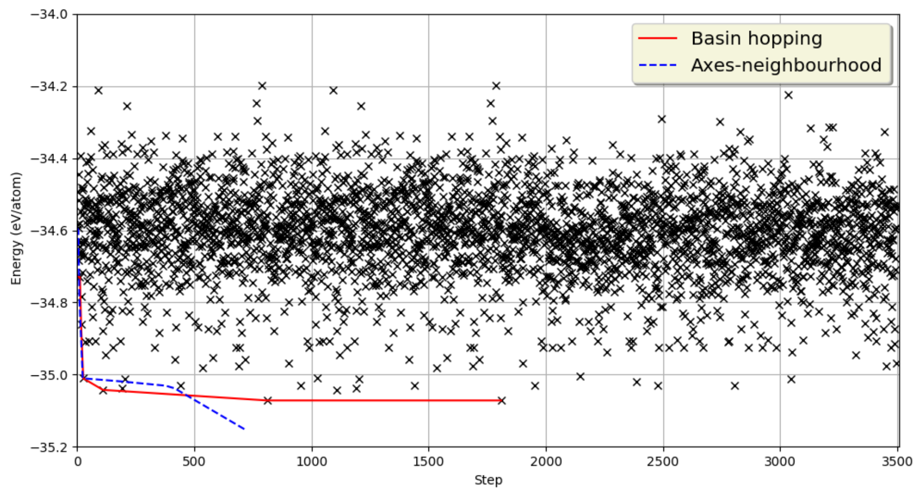


FIGURE 5.5: Performance of the basin hopping algorithm for $Y_2Ti_2O_7$. The black points correspond to the energy found after the relaxation of a random feasible configuration of atoms at each step. The red line is the lower envelope of the energy found by basin hopping, while the blue line corresponds to the lower envelope of our algorithm.

5.6 Conclusions

In this chapter we introduced and studied the Crystal Structure Prediction problem through the lens of computer science. This is an important and very exciting problem in computational chemistry, which computer scientists are not actively studying yet. We have identified several open questions whose solution would have significant impact to the discovery of new materials. These problems are challenging and several different techniques and machineries from computer science could be applied for solving them. Our simple-to-understand algorithms are a first step towards their solution. We hope that our algorithms will be used as benchmarks in the future, since more sophisticated techniques for basin hopping can be invented. For the energy computation via the depth approach, we conjecture that the arrangement that minimizes the energy for $k = 1$ or $k = 2$, matches the arrangement that minimizes the energy when it is computed via GULP. Our numerical simulations provide significant evidence towards this. A formal result of this would greatly simplify the objective function of the optimization problem and it would give more hope to faster methods for relaxation. In addition, it could provide the foundations for new techniques for crystal structure prediction. Our algorithm that utilizes the Axes neighborhood as an intermediate step before relaxation, seems to speed up the time the standard basin hopping needs to find the global minimum. Are there any other neighborhoods that outperform the Axes one? Can local search, or Axes neighborhood in particular, improve existing methods for crystal structure prediction by a simple integration as an intermediate step? We believe that this is indeed the case.

Part III

Mobile Agents on Dynamic Graphs

Chapter 6

Gathering in 1-Interval Connected Graphs

In this chapter we examine the problem of gathering $k \geq 2$ agents (or multi-agent rendezvous) in dynamic graphs which may change in every round. We consider a variant of the 1-interval connectivity model [96] in which all instances (snapshots) are always connected spanning subgraphs of an underlying graph, not necessarily a clique. The agents are identical and not equipped with explicit communication capabilities, and are initially arbitrarily positioned on the graph. The problem is for the agents to gather at the same node, not fixed in advance. We first show that the problem becomes impossible to solve if the underlying graph has a cycle. In light of this, we study a relaxed version of this problem, called *weak gathering*, where the agents are allowed to gather either at the same node, or at two adjacent nodes. Our goal is to characterize the class of 1-interval connected graphs and initial configurations in which the problem is solvable, both with and without *homebases*. On the negative side we show that when the underlying graph contains a spanning bicyclic subgraph and satisfies an additional connectivity property, *weak gathering* is unsolvable, thus we concentrate mainly on unicyclic graphs. As we show, in most instances of initial agent configurations, the agents must meet on the cycle. This adds an additional difficulty to the problem, as they need to explore the graph and recognize the nodes that form the cycle. We provide a deterministic algorithm for the solvable cases of this problem that runs in $O(n^2 + nk)$ number of rounds.

6.1 Introduction

6.1.1 Previous work

The problem of *gathering* on graphs requires a set of k identical mobile agents that operate in Look-Compute-Move cycles, to end up in the same node. In each cycle,

an agent takes a snapshot of its immediate neighborhood (Look), performs some computations in order to decide whether to move to one of its adjacent nodes, or remain idle (Compute), and in the former case makes an instantaneous move to that neighbor (Move).

The feasibility of *gathering* has been extensively studied in the static setting, and under various assumptions. A very common assumption that makes the problem solvable in ring graphs is for the agents to have distinct identities [42, 49, 55]. Alternatively, another assumption which pertains to the communication capabilities of the agents, is either to supply each node with a *whiteboard* where the agents can leave notes as they travel [140], mark the nodes that the agents are initially placed, or provide the agents with a constant number of movable tokens that can be placed on nodes, picked up, and carried while moving [41]. Under the first communication assumption the problem becomes solvable even in the presence of some faults [18, 31].

In [59], the authors study the feasibility of gathering a set of identical and without explicit communication capabilities agents in dynamic rings (1-interval connectivity, [96]). As *strict gathering* becomes infeasible in this setting, they focus on a variation of gathering, called *weak gathering*, where the agents are allowed to gather either at the same node, or at two adjacent nodes. They investigate the impact that *chirality* (i.e., common sense of orientation on the cycle) and *cross detection* (i.e., the ability to detect whether some other agent is traversing the same edge in the same round) have on the solvability of the problem. In order to drop the latter assumption, they later construct a mechanism which avoids agents crossing each other (i.e., no agents traverse the same edge at the same round and in opposite directions), called *Logic Ring*. To enable feasibility of *weak gathering*, they empower the agents with some minimal form of implicit communication, called *homebases* (the nodes that the agents are initially placed are identified by an identical mark, visible to any agent passing by it). They provide a complete characterization of the classes of initial configurations from which *weak gathering* is solvable in the presence or absence of cross detection and chirality, by providing polynomial time distributed algorithms. They prove that without chirality, knowledge of the ring size is strictly more powerful than knowledge of the number of agents. Finally, with chirality, they show that knowledge of the ring size can be substituted by knowledge of the number of agents, yielding the same classes of feasible initial configurations.

In [57] the authors investigate the feasibility of the decentralized (or *live*) exploration problem in 1-interval connected rings by a set of mobile agents. They consider both the fully synchronous and semi-synchronous cases and study the impact that *anonymity* and knowledge of the ring size has on the solvability of the problem.

Other recent work [47] has considered the broadcasting problem in dynamic, constantly connected, networks, where the agents can communicate when they meet at a node, and they have global visibility allowing them to see the location of other agents in the graph. Finally, exploration by $O(n)$ agents of dynamic tori graphs has been investigated in [85], and, in a very recent work, exploration in time-varying graphs (including 1-interval connectivity) of arbitrary topology in [84].

6.1.2 Contribution and roadmap for the chapter

The existing literature on the gathering and rendezvous problems is extensive and has been examined under various assumptions for both the environment that the agents navigate and the capabilities of the agents (for surveys see [95, 129]). Despite their differences, they investigate these problems in the case where the topological structure does not change over time (i.e., they only consider static graphs). Recently, there has been a growing interest in studying these problems in dynamic settings, with [59] being the first work that examines the problem of *weak gathering* in 1-interval connected ring graphs. Embarking from this work, we investigate if and under what assumptions we can go beyond rings and how far, in the presence or absence of *homebases*. Our main result is a distributed algorithm that solves *weak gathering* in unicyclic graphs. A *unicyclic* graph is a connected graph containing exactly one cycle. Observe that ring graphs are special cases of unicyclic graphs.

We use a traditional model in the literature of autonomous mobile agents on graphs (see, e.g., [54]), and we consider it in a dynamic synchronous setting. In particular, in this model the edges are locally labeled in each node, and at any round some edges can be missing (i.e., being disabled), provided that the resulting graph is connected. This means that the snapshots of the dynamic graph are always spanning and connected subgraphs of some given underlying graph. This notion of dynamicity includes the classic 1-interval connectivity as a sub-case when the underlying graph is a clique.

We start a characterization of the class of solvable graphs in the aforementioned generalized 1-interval connectivity setting, and we study the effect that port labels have on the solvability of *weak gathering*. We show that *weak gathering* is unsolvable when the underlying graph contains a spanning bicyclic subgraph and satisfies an additional connectivity property, regardless of any other additional assumptions (i.e., communication or knowledge of graph properties). In light of this, we then focus on unicyclic graphs, and we study the classes of initial configurations on which *weak gathering* is feasible in the presence or absence of *homebases*. In particular, we characterize the classes of unicyclic graphs in which certain symmetries occur and would render impossible the problem of symmetry breaking. We show that if neither the size of the graph nor the number of agents is known, then the agents are not able

to distinguish between symmetric and asymmetric configurations. The additional difficulty in unicyclic graphs comes from the fact that in most instances of initial agent configurations, the agents must necessarily gather on the unique cycle. This requires them to perform some sort of exploration in order to reach the cycle, while the scheduler can choose some agent, or agents, and delay them.

In [59] the authors utilized *homebases* in order to break the symmetry on the ring, however, in our setting we can exploit the topological asymmetries of the graph and the port labeling to solve the problem. We then show that *homebases* can be used in order to expand the class of feasible configurations, that is, the initial placement of the agents might create some additional topological asymmetries. We also assume that the agents have cross-detection. In Section 6.5 we discuss how the mechanism of Di Luna et al. [59], that avoids agents crossing each other, could be used in order to drop this assumption, and we conjecture that this approach gives a correct algorithm that solves *weak gathering*.

We then provide a deterministic algorithm that solves *weak gathering* in all asymmetric unicyclic graphs, and runs in a polynomial number of synchronous rounds. For the cases of symmetric unicyclic graphs with symmetric initial agent placement, we show that the problem becomes impossible to solve, and we leave as an open problem the case of symmetric unicyclic graphs and asymmetric initial agent placement. We carefully design a non-trivial mechanism that utilizes the graph topology and after $O(n^2 + nk)$ rounds all agents reach and forever stay on the cycle. Given this, the second part of the algorithm guarantees that the agents (weakly) gather on the cycle, in $O(n)$ rounds.

In summary, we show that in a large class of graphs F *weak gathering* is unsolvable. Our paper establishes that *weak gathering* is solvable in unicyclic 1-interval connected graphs in $O(n^2 + nk)$ time, and we leave a small gap for graphs $G \notin (F \cup \mathit{Unicyclic})$. For a summary of our results see Table 6.1.

In Section 6.2, we formally describe the model and we provide all necessary definitions. In Section 6.3 we provide impossibility results for (strict) *gathering*, and we describe the class of graphs where *weak gathering* is impossible to solve. In Sections 6.3.1 and 6.3.2 we provide a characterization of the feasible initial configurations in unicyclic graphs and the basic limitations of *weak gathering*. Finally, in Section 6.4 we provide our deterministic *weak gathering* algorithm and its analysis.

6.2 Model and Definitions

Graph class	Assumptions	Feasibility of *weak gathering*
Graphs in $\mathcal{F}$ Definition 6.7	Any	Infeasible Proposition 6.8
Symmetric unicyclic graphs in $S_s \cap S_a$	Any	Infeasible Lemma 6.11
Unicyclic graphs	No knowledge of n and k	Infeasible Proposition 6.13
Unicyclic graphs not in S_a	*Homebases*, knowledge of n and k	Feasible Theorem 6.19
Unicyclic graphs not in S_s	Knowledge of n and k	Feasible Theorem 6.19

TABLE 6.1: Summary of our results for the *weak gathering* problem. Assumptions include the existence of *homebases*, knowledge of the graph size n, and knowledge of the number of agents k

6.2.1 Dynamic network model

A network is modeled as an undirected connected graph $G_U = (V, E)$, referred to hereafter as an *underlying graph.* The number of nodes $n = |V|$ of the graph is called its *size.* Every node $u \in G_U$ has $\delta(u)$ incident edges, where $\delta(u)$ is its degree. For each of them, it associates a port and the ports are arbitrarily labeled with unique labels from the set $\{0, \ldots, \delta(u) - 1\}$. We call these labels the *port numbers.*

Given an underlying graph $G_U = (V, E)$ on n vertices, a *dynamic graph* on G_U is a sequence $G_D = \{G_t = (V, E_t) : t \in \mathbb{N}\}$ of graphs such that $E_t \subseteq E$ for all $t \in \mathbb{N}$. Every G_t is the snapshot of G_D at time-step t. We assume that the sequence G_D is controlled by an adversarial scheduler, subject to the constraint that the resulting dynamic graph should be 1*-interval connected*, that is, each G_t should be connected. The definition of 1-interval connectivity of [96] considers the case where the underlying graph is a complete clique. In our work, we generalize this to any underlying graph, meaning that $G'_D = (V, \bigcup_t E_t) \subseteq G_U$.

Definition 6.1 (Generalized 1-interval connectivity)**.** A dynamic graph G_D is generalized 1-interval connected if for every integer $t \geq 0$, the snapshot $G_t = (V, E_t)$ is a connected and spanning subgraph of a given underlying graph G_U.

Agents. There is a set $A = \{\alpha_1, \ldots, \alpha_k\}$ of k *anonymous* computational entities (also called *agents*), each provided with memory and computational capabilities, that execute the same protocol and can move on the graph. During the execution of the protocol, an agent learns the local port number by which it enters a node and the degree of the node. The agents are initially arbitrarily placed on some nodes of the graph, not necessarily distinct, and they are not aware of the other agents' positions.

More than one agent can be in the same node and may move through the same port number (i.e., the same edge) in the same round. We say that an agent α is *blocked* if the edge that α decided to cross in the current round is disabled by the scheduler. We consider the *strong multiplicity detection* model, in which each agent can count the number of agents in its current node. Based on that information, the port labeling, and the contents of its memory, it determines whether or not to move, and through which port number. In addition, the agents do not have any visibility around them, meaning that we do not allow them to see agents on their adjacent nodes.

We say that the system has *cross detection* when the agents have the ability to detect whether some other agent is traversing the same edge in opposite direction during the same round. We assume that the system has *cross detection*, and in Section 6.5 we discuss about how this assumption could be dropped.

We assume that the nodes of G do not have unique identifiers, and the agents do not have *explicit* communication capabilities. We do this in order to capture the limitations and the basic assumptions that make gathering in dynamic networks feasible. Finally, we assume that the agents do not have knowledge of any graph properties, other that its size.

6.2.2 Definition of terms and the problem

Definition 6.2 (Homebases)**.** We call *homebases* the nodes that the agents are initially placed. Each node u is specially marked by a bit b_u, such that if $b_u = 0$ no agent was initially placed on u, while $b_u = 1$ means then at least one agent was initially placed on u (i.e., its a homebase). In addition, each agent can determine whether its current node is a homebase or not.

Definition 6.3 (Gathering problem)**.** The *gathering* problem requires a set of k agents, initially arbitrarily placed on the graph, to gather within finite time at the same node of the underlying graph, not known to them in advance, and terminate.

Definition 6.4 (Weak gathering problem)**.** The relaxed version of the gathering problem, called *weak gathering*, requires all agents to gather within finite time at the same node, or on two neighboring nodes of the underlying graph, and terminate.

The above definition means that all agents must terminate in at most two nodes of the graph that are adjacent in the underlying graph. Finally, throughout the paper, we call *unicyclic* graphs the connected graphs that have exactly one cycle, and *bicyclic* graphs the connected graphs that have exactly two cycles, with possibly a single common vertex.

Definition 6.5 (Branch)**.** Let G be a unicyclic graph. Then G consists of a unique cycle C of n_c nodes and b trees, where $0 \leq b \leq n_c$. Each tree B_{w_i} is rooted at a node $w_i \in C$, such that the only node of the intersection of B_{w_i} with C is w_i. We call these trees the *branches* of G. See Figure 6.1 for an example.

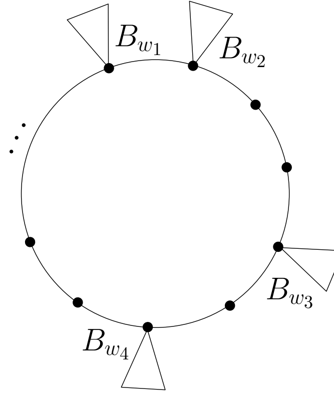


FIGURE 6.1: Example of a graph G with a unique cycle C. Each B_{w_i} is a tree (or branch) of G rooted at $w_i \in C$.

6.3 Impossibility Results

In this section we start by showing that strict *gathering* is unsolvable in 1-interval connected graphs that have at least one cycle. In the model that we consider, the case of connected acyclic graphs is equivalent to static trees. We then focus on the *weak gathering* problem, and we show that in a large class of graphs *weak gathering* is unsolvable.

Proposition 6.6. *For any generalized* 1*-interval connected graph with at least one cycle, there exists an initial agent placement such that* gathering *is unsolvable, regardless of any communication assumptions and knowledge of graph properties (e.g., its size).*

Proof. Consider an underlying graph $G_U = (V, E)$, where there exists at least one cycle of size $c > 3$. Let C be an arbitrary such cycle of G_U. Consider an execution such that for each cycle $C' \neq C$, the scheduler disables an arbitrary edge in C' that does not belong to C. Call the resulting graph G'_U. G'_U can now be represented as a unicyclic graph, where each node $w \in C$ is the root of a connected tree, or branch, G_w.

Consider an initial agent placement where there are at least two agents α and α' that are placed on branches G_w and G_u, such that $w \neq u$. Then, all paths between α and α' contain at least one edge $e \in C$. Since our graph is 1-interval connected,

the scheduler can additionally remove an edge of the cycle C in order to block the agents from reaching the same node, without violating the connectivity constraints. Therefore, they cannot achieve *gathering*. □

Definition 6.7. [Class F of graphs with blocking edges] Let G be a graph that has a spanning bicyclic subgraph with cycles C_1 and C_2. For any node u that belongs to at least one cycle, let G_u be the maximal connected subgraph, such that the intersection of the node set of G_u with C_1 and C_2 contains only u. We say that G belongs to class $\mathcal{F}$ if there are two edges $e_1 \in C_1$ and $e_2 \in C_2$, with endpoints u_1, w_1 and u_2, w_2 respectively, such that no node of G_{u_1} and G_{w_1} is adjacent with any node of G_{u_2} and G_{w_2} in G. Call these edges *blocking*.

In other words, we say that a graph G belongs to $\mathcal{F}$ if G has a spanning bicyclic subgraph G_B with two cycles C_1 and C_2, such that all paths that contain two edges $e_1 \in C_1$ and $e_2 \in C_2$ also contain at least one more edge from each cycle. The fact that no node of G_{u_1} and G_{w_1} is adjacent with any node of G_{u_2} and G_{w_2} in G, and because each G_v contains all nodes of the corresponding connected component (i.e., it is maximal), it means that there is no path from any node of G_{u_1} and G_{w_1} to any node of G_{u_2} and G_{w_2} that does not contain at least one more edge from each cycle of G_B. An example is shown in Figure 6.2.

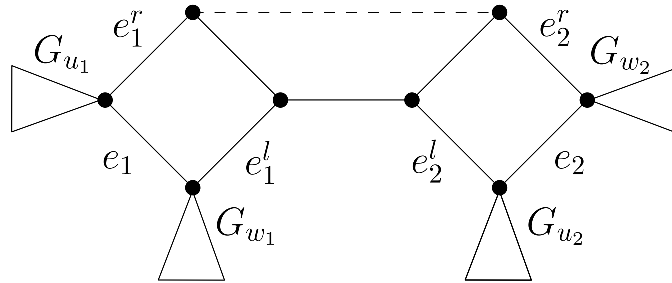


FIGURE 6.2: Example of a graph $G \in \mathcal{F}$. Dashed lines do not belong to the spanning bicyclic subgraph. e_1 and e_2 are called *blocking* edges.

Proposition 6.8. *For any generalized* 1*-interval connected graph that belongs to* $\mathcal{F}$*, there exists an initial agent placement such that* weak gathering *is unsolvable, regardless of any communication assumptions and knowledge of graph properties.*

Proof. Take any underlying graph $G_U \in \mathcal{F}$, and let α and α' be two agents that are initially placed on some endpoint of two *blocking* edges $e_1 \in C_1$ and $e_2 \in C_2$, respectively. Let p_t and p'_t be the positions of these agents at round t.

Consider an execution such that the scheduler disables all edges that are not contained in the (spanning) bicyclic subgraph, and also disables the neighboring edges $e_t \neq e_1$ and $e'_t \neq e_2$ of p_t and p'_t on the corresponding cycles, when $p_t \in C_1$ and

$p'_t \in C_2$, respectively. Observe that this does not violate the connectivity constraints, as these edges belong to different cycles. Then, in each round, an agent can either decide to wait at its current node, move on the other endpoint of e_1 (and e_2 for α'), or move towards a tree rooted at p_t (p'_t, respectively). Observe that in the last case, the distance in G_U between the agents is increased. This is because, by the definition of $\mathcal{F}$, the nodes of the trees starting from the endpoints of e_1 are not adjacent with any node of the corresponding trees of e_2 in G. Let w be a node that α moves to. All paths between w and α' pass through some endpoint of e_1. However, when α is on C_1, the scheduler blocks it from making any progress towards α', thus remain in distance at least two from α'. Symmetrically, the same holds also for α'. Therefore, the agents will never reach the same or two neighboring nodes, thus fail to solve *weak gathering*.

As an example, consider that graph of Figure 6.2 which belongs to $\mathcal{F}$, and two agents α and α' that are initially placed on u_1 and u_2, respectively. When α is on u_1, e_1^r is disabled and when it is on w_1, e_1^l is disabled. Similarly for α'. Observe that in any of these cases, the connectivity of the graph is maintained, while α is always blocked on some node of G_{u_1} or G_{w_1} and α' on some node of G_{u_2} or G_{w_2}. □

6.3.1 Symmetric initial configurations in unicyclic graphs

The main difficulty in solving gathering is symmetry which occurs in several ways, such as the topology of the graph, the port labeling, and the initial positions of the agents. Given that the agents are identical and there is no means of explicit communication, in case that the configuration is highly symmetric, the problem is clearly impossible to solve by deterministic means. The problem of deterministically breaking the symmetry is translated into the problem of distinguishing a node, or an edge for the *weak gathering* problem, where the agents should meet.

In light of the above impossibilities, we hereafter consider unicyclic graphs and we describe the class of symmetric initial configurations in such graphs, with and without *homebases*. We show that *weak gathering* is unsolvable in symmetric unicyclic graphs. Note that in this section we do not consider any graph dynamics, as we are only interested in identifying the graph classes that even in a static setting, the problem of symmetry breaking is impossible.

An *initial configuration* is defined by the graph, the port labeling, and the (initial) positions of the agents.

Branch classes

Call C the nodes of the unique cycle and B_i the branch rooted on $u_i \in C$, $\forall u_i \in C$, possibly consisting only of the root node u_i. We define a class I of indistinguishable

branches, the class where all of the following hold:

1) Any two branches B and B' in I, rooted at u and u', respectively, are isomorphic.

2) For each pair of branches B and B' in I, the branches are label-preserving, meaning that vertices with equivalent port labels (i.e., the same) are mapped onto the vertices with equivalent port labels and vice versa. This means that for each pair of connected nodes $(u, w) \in B$ which is mapped onto the (connected) pair of nodes $(u', w') \in B'$ in that order, the port label of u leading to w is the same as the port label of u' leading to w' and vice versa.

3) For any two branches B and B' in I with root nodes u and u', respectively, the port labels of u and u' leading to their clockwise neighbor of the ring are the same. Similarly, the port labels of u and u' that lead to their counter-clockwise neighbors of the ring are the same.

Symmetric configurations

Let $\mathcal{I} = \{I_0, I_1, \dots, I_l\}$ be the set of all distinct branch classes of a given unicyclic graph G. Let u_i denote the i-th node in the clockwise direction of the cycle, starting from an arbitrary node $u_0 \in C$, and $s_i \in \mathcal{I}$ be the class that the branch starting from u_i belongs to. We call a graph *symmetric* if the following holds: for each $I_j \in \mathcal{I}$ let $P_j = \{p^j_0, p^j_1, \dots, p^j_{m_j}\}$ be the set of all periods, where $p^j_z < |C|$ and $0 \leq z \leq m_j$, such that for each node u_i with $s_i = I_j$, there exists p^j_z such that $s_i = s_{(i+p^j_z) \mod |C|}$, $\forall i < |C|$ (see examples in Fig. 6.3).

We call $\mathcal{S}_s$ the set of all symmetric configurations defined by the graph topology and port labeling.

Agent position symmetries

In a similar way we define the symmetries that are induced by the initial placement of the agents on the graph. These symmetries can only be defined in configurations of $\mathcal{S}_s$. Assume that each vertex is initially labeled with a bit b, indicating whether an agent is initially placed on that vertex, or not; call them agent labels. Then, we define the set $\mathcal{S}_a$ of such configurations as follows. For any pair of branches B and B' that belong to the same branch class I and have the same periodic appearance, the branches are label-preserving, meaning that vertices with the same agent labels are mapped onto the vertices with the same agent labels and vice versa.

Definition 6.9. We say that a configuration S is symmetric if $S \in \mathcal{S}_s$. If, additionally, the communication model allows *homebases*, we call S symmetric if $S \in (\mathcal{S}_s \cap \mathcal{S}_a)$.

FIGURE 6.3: Examples of symmetric configurations with periodic branches and port labels on the cycle. In the top left figure all unique branches have the same periodic appearance (i.e., $P_i = \{3\}$, $\forall i \in \{0,1,2\}$). In the top right and bottom figures, the branches have different periodic appearances (i.e., in the top right figure the sets of periods are $P_0 = \{4\}, P_1 = \{2\}$, and $P_2 = \{4\}$, while in the bottom figure the sets of periods are $P_0 = \{2,4\}$ and $P_1 = \{4\}$).

Lemma 6.10. *A unique (leader) node can be elected in all non-symmetric configurations. We call these configurations* feasible. *This holds regardless of any communication assumptions and knowledge of graph properties.*

Proof. We prove this lemma by construction: we provide a deterministic algorithm that given a configuration S (the graph topology, the port labeling, and, if available, the *homebases*), such that $S \in \mathcal{S} \backslash \mathcal{S}_s$ or $S \in \mathcal{S} \backslash (\mathcal{S}_s \cap \mathcal{S}_a)$, elects a unique node as a leader.

Consider a rooted tree B with port labels. Starting from the root node u, a sequence that uniquely represents B can be constructed as follows. Consider a sequence $\mathcal{T}_B =< T_1, T_2 \ldots, T_\ell >$, $T_i = (t_0^i, t_1^i, \ldots, t_{d_i-2}^i)$, where $\ell = |B|$, each tuple T_i corresponds to a node of the tree, and d_i is its degree. Each t_j^i is the tree size rooted at each child node of T_i (i.e., we exclude T_i's parent node from T_i). Then, the size of the subtree rooted at T_i is $1 + \sum_{j=0}^{d_i-2} t_j^i$. Their order in T_i is defined by selecting port labels in ascending order. If we consider the case where identical labels exist on

the initial positions of the agents (i.e., *homebases*), then the above sequences can be modified by having an H symbol in the beginning of the tuple T of each node where a *homebase* exists. Finally, the tuples $T_i \in \mathcal{T}_B$ are ordered in a DFS way, where we visit the children of each node by selecting the port labels in an ascending order. Observe that there is a simple algorithmic strategy that can be used by an agent to traverse the tree in a DFS way: when the agent arrives at node w through a port p, it leaves w through port $(p+1) \bmod \delta(w)$ in the next step. Initially, the agent starts by leaving the port 0 of the root node u.

Given an arbitrary orientation on the cycle, same for all branches, construct a set $\mathcal{C}_B = (\mathcal{T}_B, p_0, p_1)$ for each branch of the tree B, where p_0 is the port label of the root node of B that leads to its neighboring node on the arbitrarily chosen orientation and p_1 the port label that leads to its second neighbor of the cycle. For each distinct set $\mathcal{C}_B$, assign a unique label $a \in \mathbb{Z}$; call them *branch labels*. Observe that this yields an assignment of unique labels on branches that do not belong to the same branch class. The above construction can be then used to distinguish a node on C as follows. Let P_c^u and P_{ccl}^u be the sequences of branch labels as constructed above by following the clockwise and counter-clockwise directions respectively, starting from node $u \in C$. Let δ_{w_1} and δ'_{w_2} denote the two lexicographically minimum sequences of P_c^u and P_{ccl}^u of size $|C|$, $\forall u \in C$.

If there is a unique lexicographically minimum sequence, let that be δ_w, we elect w as the leader node. In case that the lexicographically minimum sequences δ_{w_1} and δ'_{w_2} are identical, and $w_1 \neq w_2$, it means that there is a unique axis of symmetry, equidistant from w_1 and w_2 (otherwise, the configuration would be symmetric). If that axis passes through one node w and one edge, we elect w as the leader node. If the axis passes through two nodes u_1 and u_2, there exist two lexicographically minimum sequences of size $|C|/2$ starting from w_1 and w_2 that both contain either u_1 or u_2. Then, we elect u_1 or u_2 as the leader node, respectively. Observe that it is not possible that one sequence contains u_1 and the other u_2, as in that case the configuration would clearly be symmetric (i.e., the configuration would have two axes of symmetry). Similarly, if the unique axis of symmetry passes through two edges e_1 and e_2, the two lexicographically minimum sequences contain both either e_1 or e_2; let that edge be e_1. Observe that in this case the trees starting from the endpoints of e_1 have both been assigned the same label α. This means that the ports of e_1's endpoints are different (if they were the same, the labels would be different). Then, we elect as leader the endpoint of e_1 with the minimum label. □

Lemma 6.11. *If the graph and the initial agent placement are symmetric,* weak gathering *cannot be solved. This holds regardless of* homebases *and knowledge of graph properties.*

Proof. Let S be a symmetric configuration with k agents, and B_u the branch starting from a node u of the cycle (containing u). Consider an execution in which no edge of the underlying graph ever becomes disabled.

Consider a symmetric initial agent placement. Then, call a group the agents that are mapped onto vertices of the same branch class with the same periodic appearance. The agents of each group will then perform exactly the same actions, based on the same observations. This means that for each group of agents, they will always be on branches that belong to the same branch class with the same periodic appearance, and their current positions will be always mapped onto the same vertices. If they move on the cycle, they will again perform the same actions (i.e, they will move either clockwise, or counter-clockwise), thus, the distance between consecutive agents of the same group will never change. Observe that this holds regardless of the existence of *homebases*. □

The only case that we haven't examined is the case without *homebases*, where the graph is symmetric, and the initial agent placement is asymmetric. Consider a symmetric ring graph, where all port labels that lead to the clockwise neighbors are the same, and the port labels that lead to the counter-clockwise neighbors are also the same, and no edge is ever missing. Independently of the initial agent placement, all agents operate with the same observations in each round, therefore they all move either clockwise, or counter-clockwise (i.e., the distance between consecutive agents of the ring remain always the same). Therefore, a more precise characterization of the feasible graph configurations is necessary for this case. We believe that without a way to elect a leader, same for all agents, the problem of *weak gathering* becomes impossible, and we leave this as an open problem.

6.3.2 Additional limitations on the solvability of *weak gathering*

In this section we examine the impact that some additional limitations have on the solvability of *weak gathering*. First, observe that in generalized 1-interval connected unicyclic graphs, the scheduler can completely block an agent from reaching some part of the graph. This implies that if some other agent moves only on that part, then these agents would never meet or end up in neighboring nodes. As we show in the following lemma, this problem can be overcome only if all agents explore the graph, identify the cycle, and solve the problem there.

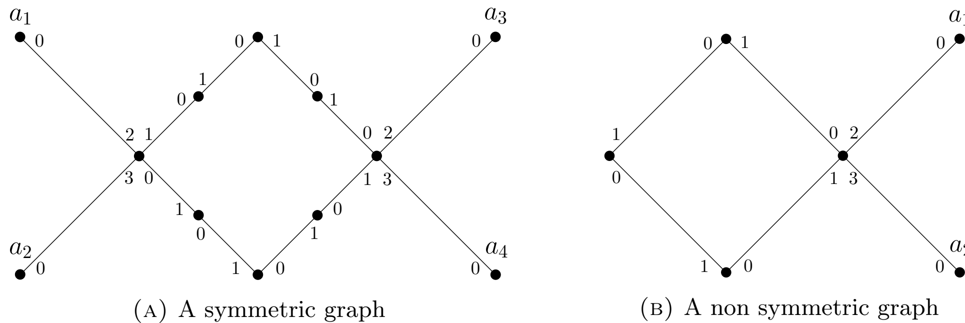
Lemma 6.12. *For any generalized* 1*-interval connected unicyclic graph with cycle* C *of size* $|C| > 3$*, there exists an initial agent placement such that* weak gathering *can only be achieved on the cycle. This holds regardless of any communication assumptions and knowledge of graph properties.*

Proof. We call a branch empty if it only consists of the root node. Consider any unicyclic graph with a cycle C, $|C| > 3$, and at least one non-empty branch (if all nodes have empty branches, then the lemma trivially follows). Let B_w be the branch rooted on node $w \in C$. Let α be an agent that is initially placed on some node of $B_w \backslash w$, and an agent α' that is initially placed on a (possibly empty) branch B_u, such that u is in distance at least two from w. Then, consider an execution in which the scheduler selects α' and blocks it in distance two from B_w (i.e., whenever it is at distance two from w, it disables the corresponding edge). Therefore, in order to solve *weak gathering*, agent α must first reach the cycle. Additionally, the scheduler can always block them from reaching the same branch, as it can keep them at distance at least one. If some agent decides to move towards a branch, then the scheduler can still block the other agent from reaching that branch. □

The above lemma means that the agents must first explore the graph in order to identify the nodes of the cycle C and gather on some node $v \in C$, otherwise, the scheduler can always block some agent from reaching the rest of them.

Proposition 6.13. *If neither n nor k are known, then the agents cannot distinguish periodic from aperiodic graphs. This holds regardless of* homebases.

Proof. Let G_1 be the graph of Figure 6.4a and G_2 be the graph of Figure 6.4b. The numbers represent the local port labels of each node, and a_i are the initial positions of the agents.



(A) A symmetric graph (B) A non symmetric graph

Figure 6.4: Indistinguishable unicyclic graphs

Consider an execution where no edge is missing at any round. If neither k nor n are known to the agents, these two graphs are indistinguishable between each other, even when the initial positions of the agents are identically marked (*homebases*). If the agents are able to recognize infeasibility in G_1 (because of being symmetric), then this would also wrongly happen in G_2. Otherwise, if the agents solved the problem in G_2 and terminated, then the agents in G_1 would also terminate in two nodes that are not neighbors.

Given any unicyclic graph G_u with k agents, we can construct periodic unicyclic graphs that the agents cannot distinguish from G_u. Let $C_u = \{u_1, u_2, \ldots, u_c\}$ be the set of nodes of the cycle in G_u. Then, to construct a periodic graph G_p, construct a cycle $C_p = \{u_1^1, u_2^1, \ldots, u_c^1, u_1^2, u_2^2, \ldots, u_c^2, \ldots, u_1^p, u_2^p, \ldots, u_c^p\}$, and for each i, all nodes $u_i^j, \forall j$, have the same port numbers between them, and with the corresponding nodes of C_u. In addition, for each i, the trees starting from nodes $u_i^j \in C_p, \forall j$ are exact copies of the corresponding trees of nodes $u_i \in C_u$. Finally, all agents of G_u are also mapped onto all copies of the corresponding nodes of G_p (i.e., we have pk agents in G_p). Then, similarly to Figure 6.4, the agents cannot distinguish G_u from any G_p. □

6.4 An Algorithm for *Weak Gathering* in Dynamic Unicyclic Graphs

In light of the impossibilities of Section 6.3, we hereafter consider generalized 1-interval connected unicyclic graphs, and we provide a deterministic algorithm that solves *weak gathering* for all non-symmetric configurations. We assume that the agents have knowledge of n, knowledge of the number of agents k, and the system has *cross detection*. Finally, our algorithm solves *weak gathering* for both cases, with and without *homebases*, provided that the configuration is not symmetric (as defined in Definition 6.9). If the configuration is symmetric, then the agents agree on unsolvability and terminate. A very significant aspect of mobile agent systems is the memory requirements of the agents. In order to achieve symmetry breaking by exploiting the topological asymmetries of the graph itself and the port labeling, our algorithm constructs a map of the graph in the local memory of each agent. Therefore, we assume that the agents have non-constant memory.

6.4.1 *Weak gathering* algorithm

Our deterministic algorithm is divided into two phases, and the overall idea is the following: During the first phase all agents explore the graph using a DFS approach, try to identify the nodes that belong to the cycle, and at the same time independently build a map of the graph. The latter is necessary in order to break the symmetry on the cycle and agree on a unique target node.

The problem of graph mapping becomes impossible if neither of n or k are known and without *whiteboards* on the nodes [43]. Most of the algorithms rely on either the usage of *whiteboards* [45, 46], or assume that the agents can observe the memory contents of each other when they meet at the same node [83]. In the latter approach, the agents maintain multiple hypotheses when ambiguity about the graph topology

occurs, and they resolve that ambiguity when they meet. Interestingly, in the special case of unicyclic graphs we show that knowledge of n alone (i.e., knowledge of k and *whiteboards* are not necessary) can lead to the construction of graph maps that are consistent (i.e., the same) among all agents.

When all agents have completed the first phase, the executed process (second phase) ensures that they will eventually gather on the cycle. Note that the agents may move to the second phase asynchronously. In this case, gathering might be unsuccessful; the agents recognize it and they start the second phase again. In order to make the description of the algorithm more clear, we first introduce a number of variables that are stored in the local memory of each agent.

- *rounds*: A counter that is used in order to count the number of rounds in several cases of the second phase. It is increased by one in each round.
- *inPort*, *outPort*: Port labels that the agent enters and leaves a node, respectively.
- *Graph (or $\mathcal{G}$)*: Contains lists that represent the nodes visited by an agent. A specific node of the underlying graph might correspond to multiple nodes in $\mathcal{G}$. We refer to the *Graph* of an agent α as $\mathcal{G}_\alpha$.
- *currendNode*: A pointer to the current node in $\mathcal{G}_\alpha$.
- *depth*: The distance between the agent and its initial position in $\mathcal{G}_\alpha$.
- *roundsBlocked*: The number of consecutive rounds that the agent remains blocked.
- *numAgents*: The number of agents in the current node of the agent. The considered model allows the agents to count the number of agents in their current node.
- *numAgentsPrev*: The number of agents in the node of the agent during the previous round (at the end of each round, the value of *numAgents* is copied to *numAgentsPrev*).
- *numAgentsTemp*: A temporary variable that is used to store the number of agents in several cases of the second phase.
- *orientation*: This variable is used during the second phase of the algorithm and indicates the direction in which the agent traverses the cycle (i.e., clockwise or counter-clockwise).
- *orientationTemp*: A temporary variable that is used to store the orientation.

- *crossed*: A bit which becomes one when the agent crosses some other(s) agent(s). At the end of each round it is reset to zero.

Phase 1. This phase is responsible for traversing the graph (exploration) and identifying the nodes that form the cycle. We now present all procedures that take place during this phase.

Graph exploration. Each agent α stores in its local memory (in $\mathcal{G}$) a list of the neighbors of each vertex visited and the port numbers that led to those nodes. Let u be the initial node of an agent α. Then, α constructs a list $L(u)$ which represents u. Assume that it traverses an edge through port number i and arrives at a node w at port number j. It then constructs a new list $L(w)$, and in $L(u)$ it stores a tuple which consists of the port number i and a pointer to $L(w)$. At the same time, it stores in $L(w)$ a tuple with the port number j and a pointer to $L(u)$. If α is in a node v of $\mathcal{G}$ and moves through a port that is contained in a tuple of $L(v)$, it does not update $\mathcal{G}$. Finally, for each node visited (or for each list of $\mathcal{G}$), it also stores its degree. We call these lists the *Graph* of α or $\mathcal{G}_\alpha$. The initial node of each agent in $\mathcal{G}_\alpha$ (i.e., the first node that was added in $\mathcal{G}_\alpha$) is marked with a special character $\mathcal{I}$. In each round, the agent α stores in *currentNode* a pointer to the node (or list) of $\mathcal{G}_\alpha$ that it is at. In addition, in each round it calculates the (shortest) distance in $\mathcal{G}_\alpha$ between its current node and its initial node and stores it to its *depth* variable.

We use a traditional technique which makes each agent traverse a tree in a DFS way. In a round, when the agent arrives at node u through a port i, it leaves u through port $(i+1) \bmod \delta(u)$ in the next round (if the edge is available). Initially, each agent starts by leaving the port 0, and when $depth = n$, the agent moves through the port that it arrived from. The exploration ends when the agent has explored all paths of length n from its initial position. This can be achieved by checking if there is a node in $\mathcal{G}_\alpha$ in distance less than n that have at least one unexplored neighbor. In particular, if the number of tuples stored in a node list is less than its degree, then the exploration is not complete. At any point, if the edge that an agent tries to traverse is missing, it waits until it becomes enabled. As we show later, all agents either make progress towards the exploration of the graph and eventually complete the first phase of the algorithm, or if the scheduler blocks an agent indefinitely, our algorithm guarantees that the rest of the agents will reach some endpoint of the missing edge, and terminate. This means that if some agent(s) fail to explore the graph, all agents will still solve *weak gathering*.

Cycle detection. During this step, the agents check a number of predicates that help them to detect the nodes that belong to the cycle. In particular, whenever an agent α reaches a node u with degree $\delta(u) = 1$, it marks the corresponding node in $\mathcal{G}_\alpha$

with a special character $\not\emptyset$, indicating that it does not belong to the cycle, and never moves to that node (of $\mathcal{G}_\alpha$) again. In addition, if a node $u \in \mathcal{G}_\alpha$ with degree $\delta(u)$ has $\delta(u) - 1$ marked neighbors in $\mathcal{G}_\alpha$, the agent also marks u.

In addition, each agent marks in $\mathcal{G}_\alpha$ its initial node with a different special character $\mathcal{T}$. Let u be the $\mathcal{T}$-marked node in $\mathcal{G}_\alpha$. If at any point the agent α marks u in $\mathcal{G}_\alpha$ with $\not\emptyset$, it moves the $\mathcal{T}$ mark to the unique neighbor of u that is not marked with $\not\emptyset$ in $\mathcal{G}_\alpha$.

As we show in Lemma 6.14, this procedure guarantees that by the end of the *graph exploration* step, the $\mathcal{T}$ marks of all agents will correspond to nodes of the cycle. Each agent moves on that node by following the shortest path of $\mathcal{G}$, and performs the following computations.

Let u_0 be a node that an agent α is located after the end of the *graph exploration* procedure. Let B_{u_0} be the branch that starts from node u_0. The agent α needs to resolve the ambiguity that occurs in $\mathcal{G}_\alpha$ in order to identify the nodes that belong to the cycle. At this point, α arbitrarily chooses one of the two directions of the cycle (e.g., the one with the lowest port number p_{u_0}), and deletes all nodes of $\mathcal{G}_\alpha$ on the opposite direction. To distinguish B_{u_0} from the nodes of the cycle, observe that at that point all nodes of B_{u_0} are $\not\emptyset$-marked in $\mathcal{G}$, while the neighbors of u on the cycle are not marked in $\mathcal{G}$. This holds even if the initial position of an agent α is on the cycle. Then, because of the fact that the agent explored all paths up to distance n from its initial node, it means that it has traversed at least once the whole cycle. In addition, as we show in Lemma 6.15, during the first cycle traversal, all branches have been explored and marked with $\not\emptyset$. Then, observe that $\mathcal{G}$ is a tree that has a line path $L = (u_0, u_1, u_2, \ldots, u_c, u_0, u_1, u_2 \ldots)$ that all nodes are unmarked. Then, starting from the first node of L, it counts and keeps all nodes of the corresponding branches, until the total number of nodes becomes n. It removes the rest of the nodes, and constructs the cycle by setting the corresponding port of the last node of L that it kept to lead to the first node of L (i.e., u_0), and vice versa.

$\mathcal{G}_\alpha$ is now a correct map of the graph that can be used to break the symmetry on the cycle. Note that α can now traverse the cycle both clockwise and counter-clockwise, though, the orientation between two agents might be different. In particular, each agent has a private orientation o_i, $0 \le i \le k$, where *clockwise* is initially the orientation defined by traversing the nodes in the order $u_0, u_1, \ldots, u_c$ of L. We say that there is *chirality* if there are no agents α_i and α_j such that $o_i \neq o_j$, $i \neq j$. We later explain how to obtain chirality in our model.

An overview of the steps of the first phase of the algorithm is the following.

1. Initialization of variables.

2. Add the initial node in the local graph map $\mathcal{G}$, and mark it with $\mathcal{T}$.

3. Explore the graph up to distance n, and construct the map of the graph. Initially, leave through port 0.

4. Mark all nodes with $\not\mathcal{O}$ in $\mathcal{G}$ that have exactly one unmarked neighbor (i.e., initially the leaves). When you mark the node where the $\mathcal{T}$ mark is, move $\mathcal{T}$ on the unique unmarked neighbor.

5. Upon exploring all paths of length n from the initial position, move on the node which is marked with $\mathcal{T}$, and construct the cycle C.

6. If the map $\mathcal{G}$ is symmetric, terminate. Otherwise, elect a leader and move to the *Second phase*.

Finally, the pseudocode of the first phase can be found in Algorithm 11.

Phase 2. When an agent α enters this phase, it means that is has constructed a correct map $\mathcal{G}_\alpha$ of the graph in its local memory. Then, a unique node can be elected as a leader, as described in Lemma 6.10. If the configuration is symmetric, then the agents recognize it and terminate. In any other case a unique node will be elected and will be the same for all agents. Let that leader be a node ℓ. At this point, they can also obtain *chirality* by utilizing the port numbers of ℓ. Let p_1 and p_2 be the ports that lead to its neighboring nodes in the cycle. Assume, without loss of generality, that $p_1 < p_2$. Then, α sets as *clockwise* the orientation that is defined by traversing p_1 and *counter-clockwise* the one defined by traversing p_2.

In this phase, an agent can either be in state *walking* or *gathering*, and initially it is in state *gathering*. Each agent α in this phase assumes that all agents have entered the second phase, and it performs some actions that would solve weak gathering in case that this assumption is true. Otherwise (i.e., there exists some other agent that has not entered the second phase), *weak gathering* will (temporarily) fail, and updates its state to *walking*. In *grouping* we explain how the agents form groups when certain predicates are satisfied. We call a set of agents a *group* if they are on the same node and move in the same direction.

Walking state. An agent in this state traverses the cycle counter-clockwise, and after $|C|$ rounds, where C is the cycle, it changes its state to *gathering*. To achieve this, when it enters to this state, it resets the value of its *rounds* variable to zero and in each round it increases its value by one.

Gathering state. The agents in this state perform the following actions, and if they fail to solve *weak gathering* they change their state to *walking*. We divide this process into two steps. During the *first step*, each agent initially resets its *rounds* variable to zero, and moves for $2|C|$ rounds towards the elected node ℓ, by following the

Algorithm 11 First phase of *weak gathering* algorithm

Result: Identifies the nodes that form the cycle.

1: *state* $\leftarrow$ Phase 1
2: *statePrev* $\leftarrow \emptyset$
3: *rounds*, *depth*, *roundsBlocked*, *outPort* $\leftarrow 0$
4: $\mathcal{G} \leftarrow \emptyset$ # create a new empty list

1: **procedure** FirstPhase
2: $L \leftarrow \emptyset$
3: mark(L, $\mathcal{T}$)
4: mark(L, $\mathcal{I}$)
5: append($\mathcal{G}$, (L, *degree*)) # add L in $\mathcal{G}$.
6: *currentNode* $\leftarrow \mathcal{G}(L)$ # the current node is a pointer to the L list of $\mathcal{G}$
7: **while** $\exists u \in \mathcal{G}$: dist$(u, \mathcal{I}) < n$ and $|L(u)| <$ degree(u) AND *state* $\neq$ terminate **do**
8: **if** *state* = terminating **then**
9: TerminationCondition() # See Algorithm 2
10: Continue to next iteration of the loop (next round).
11: **end if**
12: **if** *degree* = 1 OR markedNeighbors($\mathcal{G}$, *currentNode*, $\varnothing$) = *degree* - 1 **then**
13: mark(*currentNode*, $\varnothing$)
14: **if** *currentNode* is marked with $\mathcal{T}$ **then**
15: unmark(*currentNode*, $\mathcal{T}$)
16: neighbor $\leftarrow$ unmarkedNeighbor(*currentNode*, $\varnothing$)
17: mark(*neighbor*, $\mathcal{T}$)
18: **end if**
19: **end if**
20: *nextNode* $\leftarrow$ getNode(*currentNode*, *outPort*)
21: **if** nextNode = $\emptyset$ **then** *depth* $\leftarrow$ *depth* + 1
22: **if** *depth* $= n + 1$ **then**
23: *inPort* $\leftarrow$ *outPort*
24: **else**
25: Move(*outPort*) # See Algorithm 3
26: $L \leftarrow$ [(*inPort*, pointer(*currentNode*))]
27: append(*currentNode*, (*outPort*, L))
28: append($\mathcal{G}$, (L, *degree*))
29: *currentNode* $\leftarrow \mathcal{G}(L)$
30: **end if**
31: **else**
32: *depth* $\leftarrow$ dist($\mathcal{G}$, *currentNode*)
33: **if** *nextNode* is marked with $\varnothing$ OR *depth* $= n + 1$ **then**
34: *inPort* $\leftarrow$ *outPort*

```
35:             else
36:                 Move(outPort)
37:                 currentNode ← nextNode
38:             end if
39:         end if
40:         if numAgents = k then
41:             state = terminate
42:         end if
43:         outPort ← (inPort + 1) mod degree
44:     end while
45:     if state ≠ terminate then
46:         state ← Phase 2
47:         cycle ← detectCycle(𝒢) # As described in Phase 1 of Section 6.4.1
48:         leader ← electLeader(𝒢, cycle) # Algorithm of Lemma 6.10
49:     end if
50: end procedure
```

Algorithm 12 Termination condition of *weak gathering* algorithm

Result: Achieves *weak gathering* in case that the agent is blocked long enough for the rest of the agents to reach some endpoint of the missing edge.

```
1: procedure TerminationCondition
2:     if state ≠ terminating AND roundsBlocked ≥ 4n + k then
3:         statePrev ← state
4:         state ← terminating
5:     else if state = terminating AND numAgents ≠ numAgentsPrev then
6:         state ← statePrev
7:         roundsBlocked ← 0
8:     else if roundsBlocked ≥ 8n + 2k then
9:         state ← terminate
10:    end if
11: end procedure
```

Algorithm 13 *Move* step of an agent

```
1: procedure Move(outPort)
2:     while edge through port outPort is disabled do
3:         roundsBlocked ← roundsBlocked +1
4:         TerminationCondition() # See Algorithm 2
5:     end while
6:     roundsBlocked ← 0
7:     Move through port outPort
8: end procedure
```

shortest path. The orientation that is followed is stored in both *orientation* and *orientationTemp* variables. After $2|C|$ rounds, the agents enter the *second step*. We distinguish the following cases for an agent α, depending on whether α reached ℓ, or not, by the end of the *first step*. If α arrived at node ℓ, it checks whether all agents are there (i.e., $numAgents = k$). If yes, it terminates. Otherwise, it resets *rounds* to zero, sets *orientation* and *orientationTemp* to *clockwise*, and starts moving clockwise on the cycle for $|C|$ rounds. The agents that due to missing edges did not reach the elected node ℓ, reset *rounds* to zero, set *orientation* and *orientationTemp* to *counter-clockwise* and start moving counter-clockwise for $|C|$ rounds. As we show in Lemma 6.18, by the end of round $2|C|$ all agents that entered state *gathering* during a time window of length $|C|$ are divided into at most two groups.

After the end of the *first step*, we want the agents of each group to start the *second step* at the same time (the two different groups may start at different rounds). However, observe that the agents might not enter into state *gathering* at the same time, thus start the *second step* asynchronously. In *grouping*, we explain how the agents start walking on the cycle as groups during the *second step*. At this point, there are two groups of agents moving towards each other. In any case, the two groups of agents will either end up on the same node, or they will cross each other, or they will become blocked on the endpoints of the same edge. In *grouping*, we explain how these groups of agents merge after at most $|C|$ rounds, or terminate in neighboring nodes. In the first two cases, if there exists some agent that has not entered the second phase, *weak gathering* will temporarily fail and they all update their states to *walking*. In the later case, we show that all agents will be gathered at the endpoints of the same edge.

Grouping. This subroutine of the algorithm forms groups of agents in the following cases.

(1) *First predicate.* During the *first step* of state *gathering*, the agents reset their *rounds* variable to zero and move towards the elected node for $2|C|$ rounds. However, not all agents start this step at the same time. The first predicate of *grouping* is responsible to synchronize the agents so as to begin the *second step* at the same time, and then continue moving as groups. In particular, when an agent α enters the *second step*, it resets *rounds* to zero and starts moving either clockwise or counter-clockwise depending on whether it reached the elected node or not. Let u be the node where α was at the end of the *first step*. Then, for the next $|C|$ rounds it tries to move to the neighboring node (and wait there). The rest of the agents in u detect that the number of agents in their current node was decreased; this is achieved by calculating the difference between *numAgents* and *numAgentsPrev* in each round. They enter into the *second step* and they try to move towards α. If they successfully

reach α, they continue moving as a group until $rounds = n$. Otherwise, the *second step* is completed (after $|C|$ rounds), therefore they enter into state *walking*.

(2) *Second predicate.* If an agent in state *walking* visits the elected node ℓ and there are some other agents there, it assumes that they are in state *gathering*. In this case, it enters into state *gathering*, resets *rounds* to zero, and waits there at most $2|C|$ rounds, or until the first predicate of *grouping* is satisfied. In other words, the elected node absorbs the agents passing by it.

(3) *Third predicate.* When two agents or groups of agents cross each other or visit the same node, they merge into a single group. To achieve this, when they cross each other, the agents of the group which is closer to the elected node ℓ by following the clockwise path (say G_1), reverse direction and update the value of their *orientation* variable. The agents of the other group G_2 wait until G_1 catches them. The distance to ℓ can be easily calculated using the graph map which is stored in the local memory of each agent. Therefore, each agent knows the distance to ℓ from the nodes of both G_1 and G_2. If the edge between the two groups is missing, they wait until it becomes available again, or until the *termination condition* is satisfied. After a successful merging, the agents that are in state *gathering* continue walking the cycle in their initial direction defined by *orientationTemp*, while the agents in state *walking* reverse their initial direction (i.e., they change the value of their *orientationTemp* variable and set *orientation* = *orientationTemp*). Similarly, if the groups of agents visit the same node (i.e., they do not cross each other), the agents in state *walking* reverse direction, while the group of agents in state *gathering* does not do anything (i.e., they continue walking in their initial direction). Finally, after a successful edge traversal of G_1, if G_2 is missing, it reverses direction again, otherwise the agents in state *walking* enter into the *second step* of *gathering*.

Termination condition. The overall idea is that if an agent is blocked long enough for the rest of the agents to reach some endpoint of the missing edge, then weak gathering is achieved and the agents terminate. To achieve this, in each round, if an agent α is blocked at a node u, it increases *roundsBlocked* by one and waits there until either the edge becomes available again (in which case it resets *roundsBlocked* to zero), or until the termination condition is satisfied. In particular, if $roundsBlocked_\alpha \geq 4n + k$, it enters into state *terminating*. In this state, if during the next $4n + k$ rounds the number of agents remains the same on u, it terminates. Otherwise it moves back to its previous state and resets *roundsBlocked* to zero. Finally, at any time during the execution of the algorithm, if the number of agents on a node is k, they all terminate.

An overview of the steps of the second phase of the algorithm can be found in Algorithm 15.

Algorithm 14 Grouping subroutine of *weak gathering* algorithm

Result: Group formation of agents.

Each agent α performs the following during the second phase of the algorithm.

1. First predicate:
   (a) At the end of the *first step* of state *gathering*, store to *numAgentsTemp* the value of numAgents. After the first edge traversal during the *second step* of state *gathering*, wait until *numAgents* = *numAgentsTemp*.
   (b) If α is in the *first step* of *gathering* and has either reached the elected node, or it is blocked, if *numAgents* < *numAgentsPrev*, enter into the *second step* of *gathering*.

2. Second predicate:
   If α is in state *walking*, its current node is the elected node ℓ, and the number of agents on ℓ are more than one, enter into state *gathering*.

3. Third predicate:
   During the *walking* state and during the second step of *gathering*, in each round that this predicate is not satisfied, store to *numAgentsTemp* the value of *numAgents*. If α crossed some agent(s), go to (3a). If *numAgents* > *numAgentsPrev*, go to (3b):
   (a) Calculate the distance between the current node and the elected node ℓ in the clockwise path of the cycle. If α is closer to ℓ than the agents that were crossed, reverse direction (i.e., change the value of *orientation*), and after a successful edge traversal (merging) go to (3b). Otherwise wait until the number of agents in the current node is increased (i.e., *numAgents* > *numAgentsPrev*) and then go to (3b).
   (b) If α is in state *waking*, reverse the initial direction (i.e., the direction before the crossing/merging which is stored in *orientationTemp*), enter into the *second step* of *gathering*, and go to (3c). If α is in state *gathering*, move towards the initial direction as defined in *orientationTemp*.
   (c) After a successful edge traversal, if the number of agents remains the same as before the crossing/merging (i.e., *numAgents* = *numAgentsTemp*), go back to the previous state.

6.4.2 Analysis

We first show that after the end of the first phase of the algorithm, all agents correctly identify the nodes that form the cycle C, and then they only move on C. In addition, because of the fact that an agent can be blocked on a node of C indefinitely, we show that during the first phase all agents reach some endpoint of the missing edge after at most $2n$ rounds. We then continue and show that in the second phase of the algorithm all agents eventually enter into state *gathering* and they correctly solve *weak gathering*.

Algorithm 15 Second phase of *weak gathering* algorithm

Result: Solves *weak gathering* on some node of the cycle.

1. State *walking*:
   (a) Reset *rounds* to zero and move counter-clockwise.
   (b) When *rounds* $= |C|$, change to state *gathering*.
2. State *gathering*:
   (a) *First step.* Reset *rounds* to zero and follow the shortest path towards the elected node ℓ until *rounds* $= 2|C|$.
   (b) *Second step.* If reached ℓ, reset *rounds* to zero and move clockwise for $|C|$ rounds. Otherwise, move counter-clockwise for $|C|$ rounds.
3. *Grouping* and *termination*:
   (a) In each round, depending on the state and step of the agent, check the appropriate predicates of *grouping* and perform the corresponding actions.
   (b) If at any time *numAgents* $= k$, terminate.
   (c) If *roundsBlocked* $= 4n + k$, change to state *terminating*.
   (d) If during the next $4n+k$ rounds the number of agents in the current node remains the same (i.e., *numAgents* $=$ *numAgentsPrev* in each round), terminate. Otherwise, move back to the previous state.

First phase of the algorithm

Lemma 6.14. *Let $d_t(\mathcal{T}_\alpha, \mathcal{C})$ denote the (shortest) distance between the $\mathcal{T}$ mark of an agent α and the closest node u of the cycle C at round t. Then, $\{d_t(\mathcal{T}_\alpha, \mathcal{C})\}$, $t \geq 0$ is a decreasing sequence (i.e., $d_t \geq d_{t+1}$), and the $\mathcal{T}$ mark will eventually correspond to u.*

Proof. Initially, the agents are arbitrarily placed on some nodes of the graph. During the first phase, each agent α constructs in its local memory the graph $\mathcal{G}_\alpha$, and marks with $\mathcal{T}$ its initial node (in $\mathcal{G}_\alpha$). We refer to the $\mathcal{T}$ mark of an agent α as $\mathcal{T}_\alpha$. Then, it starts the exploration of the graph in a DFS way, and up to a maximum depth which depends on the size of the graph ($depth = n$). Call C the unique cycle and B_u the branch rooted on $u \in C$ where an agent α is initially placed. In order to mark a node $w \in \mathcal{G}_\alpha$ with $\not\varnothing$, all its neighbors except one must already be marked in $\mathcal{G}_\alpha$. This can only happen initially on the leaf nodes, then their neighbors, and so on. Now observe that all nodes of the cycle (including u) have two neighbors that belong to the cycle C, thus, α cannot mark any of them. This means that all nodes in the shortest path between the current position of the $\mathcal{T}_\alpha$ mark and u are not marked in $\mathcal{G}_\alpha$, while the rest of the nodes $v \in B_u$ will eventually be marked with $\not\varnothing$. When α marks with $\not\varnothing$ the node that its $\mathcal{T}_\alpha$ mark is, it removes it, and marks the unique neighbor that is not marked with $\not\varnothing$. Similarly, the above argument will be satisfied for the new position of $\mathcal{T}_\alpha$. Because of this, $\mathcal{T}_\alpha$ can only move closer to the cycle

every time the corresponding agent moves it. The exploration of all paths of length n from its initial position guarantees that α will visit all nodes of the branch B_u, thus, mark with $\not\emptyset$ all of its nodes, except u. Consequently, the $\mathcal{T}$ mark will correspond u. □

In contrast to the literature on exploration of graphs and graph map construction, in our model the agents cannot assign distinct labels on the nodes, thus recognize them when encountered again (cf., e.g., [127]). For this reason, when an agent enters the cycle and completes a tour, the whole graph is again considered as unexplored. However, in the special case of unicyclic graphs, the problem of map construction becomes solvable and our algorithm guarantees that after $O(n^2 + nk)$ rounds all agents construct a correct map of the graph.

Lemma 6.15 (Cycle detection)**.** Cycle detection *correctly identifies the nodes that form the cycle, and $\mathcal{G}_\alpha$ of each agent α is a correct map of the graph.*

Proof. By Lemma 6.14 the $\mathcal{T}$ marks of all agents will eventually correspond to nodes of the cycle. Let α be an agent that after the exploration process is on a node $u_1 \in C$.

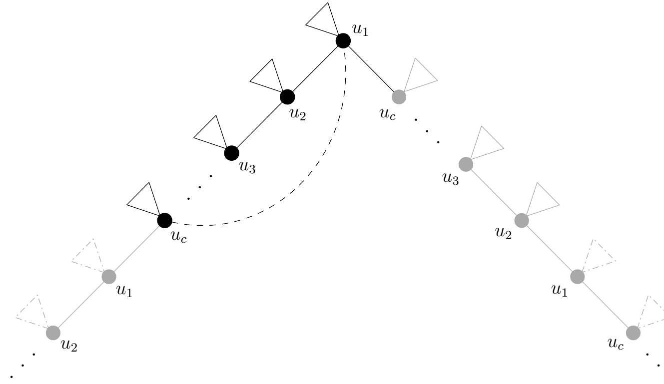


FIGURE 6.5: Locally constructed graph $\mathcal{G}$ by the end of the exploration process

Observe that $\mathcal{G}_\alpha$ is a tree (rooted at u_1), which has two line paths that the nodes are not marked with $\not\emptyset$, that correspond to the clockwise and counter-clockwise directions of the cycle. Figure 6.5 represents the locally constructed graph $\mathcal{G}_\alpha$. Then, α deletes one of the two paths and their corresponding branches (e.g, the one with the lowest port number of u_1). Observe that the distance between u_1 and all nodes of the branches until node u_c, where c is the size of the cycle, is less than n. This means that α has explored them and marked all nodes of the branches with $\not\emptyset$. Then, by counting the nodes of the cycle and their branches, it can construct a correct map of the graph. □

Lemma 6.16. *The number of rounds until all agents complete phase* 1 *is bounded by $O(n^2 + nk)$.*

Proof. Call C the unique cycle and B_u the branch rooted on node $u \in C$, where an agent α is initially placed. The number of rounds until an agent explores all paths of length n depends on the topology of the graph. In particular, when an agent enters the cycle and completes a tour, the whole graph is again seen as unexplored, thus, the agent continues exploring nodes that it has already visited in previous rounds. The number of complete tours of the cycle that can occur are $n/|C|$, and $n - |C|$ nodes are visited during each tour, provided that the depth that the agent has reached is less than n. The number of rounds R needed for the DFS exploration is then:

$$R = 4 \sum_{i=0}^{n/|C|} (n - i|C|) = \frac{4n(n + |C|)}{2|C|} = O(n^2) \tag{6.1}$$

The worst case is when the cycle has size 3, while the best case is when the cycle has size n. Due to the 1-interval connectivity, the scheduler can block α when it wants to traverse an edge of the cycle. During the DFS exploration, the number of edge traversals on the cycle is $2|C|$ for every complete tour of it. Now observe that if α is blocked for more than $8n + 2k$ rounds, it terminates, and as we show later, *weak gathering* is achieved. This means, that the worst case which does not lead to gathering, the scheduler blocks the agents for $8n + 2k - 1$ rounds for each edge traversal in C. In addition, for each cycle tour, $n - |C|$ nodes (in the worst case) can be explored without being blocked by the scheduler. Therefore, the total number of rounds that an agent can remain blocked during the first phase, considering the worst case choices of the scheduler, can be bounded by:

$$S = 2|C| \frac{n}{|C|}(8n + 2k - 1) = O(n^2 + nk) \tag{6.2}$$

The total number of rounds until all agents complete the first phase of the algorithm is then $O(n^2 + nk)$. □

Observation 1. Each agent in the first phase of the algorithm visits all nodes of the cycle every $O(n)$ rounds (at most $2n$ rounds), if not blocked by the scheduler.

The above observation holds because the number of nodes that are explored during each cycle tour is $n - |C|$ nodes on the branches and $|C|$ nodes on the cycle. The DFS exploration then guarantees that the number of rounds needed are $2(n - |C| + |C|) = 2n$.

Second phase of the algorithm

We now show that phase 2 of the algorithm successfully gathers all agents either at the same node, or at the endpoints of the same edge.

Lemma 6.17. *Let the variable roundsBlocked*$_\alpha$ *of an agent* α *be* $8n + 2k$*. Then, all agents are gathered on the endpoints of the missing edge and terminate.*

Proof. Let α be an agent that is blocked on some node u of the cycle. By Observation 1 and because of the fact that the scheduler can only remove at most one edge in each round, the rest of the agents in phase 1 perform a block-free execution, thus, after at most $2n$ rounds, they traverse the cycle and they reach u. An agent in phase 2 of the algorithm can either be in state *walking* or *gathering*. In the first case, after at most $4|C|$ rounds, it reaches u. This is because it either reaches u after at most $|C|$ rounds, or it enters into state *gathering* after at most $|C|$ rounds (second predicate of *grouping*) and remains at the elected node for $2|C|$ rounds. Then, after $|C|$ more rounds it reaches u. In the second case, an agent needs $2|C|$ rounds to move towards the elected node and then it walks the cycle (either clockwise or counter-clockwise) for $|C|$ more rounds. In all these cases observe that *grouping* can only delay the agents from reaching some endpoint of the missing edge for at most $k - 2$ rounds. This holds because they perform a block-free execution of the algorithm, and each merging of agent groups takes in the worst case one additional round. Therefore, after at most $4|C| + k - 2 \leq 4n + k$ rounds, all agents reach some endpoint of the missing edge.

After $4n + k$ rounds α enters into state *terminating*. In this state, it remains idle and observes the number of agents on its node. In case that the number of agents changes, as a result of the missing edge being enabled again, it moves back to its previous state and continues the execution of the algorithm. If after $4n + k$ more rounds the number remains the same, it terminates. Observe that when it terminates, it is guaranteed that all agents will be in state *terminating* and (weakly) gathered on the endpoints of the missing edge. This means that independently of the choices of the scheduler, after that round all agents will remain idle and will eventually terminate. □

Lemma 6.18. *Consider a set of agents* S *moving towards a node* u *in cycle* C*, following the shortest path. After* $|C|$ *rounds the agents of* S *are in at most two nodes of* C*, and one of them is in* u*.*

Proof. Consider a set of agents $S_1 \in S$ moving clockwise and a set of agents $S_2 \in S$ moving counter-clockwise. Consider two agents $\alpha_1,\ \alpha_2 \in S_1$ moving towards u. Assume that in the shortest path to u, the distance between α_1 and u is d_1 and the distance between α_2 and u is d_2.

The number of successful edge traversals until they reach u is at most $|C|/2$. Assume that α_1 didn't reach u after $|C|$ rounds. This means that it was blocked for at least $|C|/2 + 1$ rounds. Since 1-interval connectivity in this setting allows only one edge to be missing in each round, α_2 can be blocked for at most $|C|/2 - 1$ rounds

(when not in the same node with α_1). Thus, if $d_1 < d_2$, α_2 reaches α_1 by round $|C|$, and if $d_1 > d_2$, it reaches u by round $|C|$. Now consider an agent $\alpha_3 \in S_2$ moving towards u (different orientation from α_1 and α_2). Since α_1 was blocked for at least $|C|/2+1$ rounds and the agents follow the shortest path to u (they cannot be blocked on the endpoints of the same edge), α_3 can be blocked for at most $|C|/2-1$ rounds. Thus it reaches u by round $|C|$.

Overall, if an agent α is blocked for more than $|C|/2+1$ rounds, then all agents that move in the same orientation towards α reach α by round $|C|$, while the rest of the agents reach u. Otherwise, all agents reach u by round $|C|$. □

Theorem 6.19. *Our algorithm solves* weak gathering *in unicyclic graphs in* $O(n^2 + nk)$ *rounds.*

Proof. By Lemma 6.16, in $O(n^2+nk)$ rounds all agents complete the first phase of the algorithm and by Lemma 6.15 they construct a correct map of the graph. By Lemma 6.10, if the configuration is not symmetric, the agents elect a unique node on the cycle as a leader. If the configuration is symmetric, it means that there are several candidate leaders, thus, by Lemma 6.11, there are agent configurations where *weak gathering* is unsolvable and the agents terminate. Let r' be the round that the last agent enters into the second phase of the algorithm. Let also $R = \{r_1, r_2, \ldots, r_k\}$ be the rounds that the k agents enter into state *gathering* for the first time after r' (i.e., $r_i \geq r'$, $1 \leq i \leq k$).

In the second phase of the algorithm the agents can either be in state *walking* or *gathering*. Consider a set of agents S_1 that are in state *gathering* and $|r_i - r_j| < |C|$, $\forall i,j \in S_1$, and a second set of agents S_2 contains the rest of them. At this point, by Lemma 6.18 all agents that enter phase 2 at the same time, after at most $|C|$ rounds are divided into at most two groups G_1 and G_2, and one of them (say G_1) is on the elected node u. After $|C|$ rounds all agents in S_1 are in state *gathering*, thus, after $2|C|$ rounds all agents of S_1 are divided into two groups. In addition, the first predicate of *grouping* subroutine guarantees that the agents of G_1 and G_2 will continue moving as groups during the second step of phase 2. We now consider two cases.

(1) All agents of S_1 reached u. The agents of S_2 are in state *walking*, and because of the second predicate of *grouping* some of them reach u and enter into state *gathering*, while the rest of them, again by Lemma 6.18, become a group that did not reach u due to missing edges. Observe that this group walks the cycle counterclockwise, while the agents of S_1 walk the cycle clockwise. At this point there are two groups of agents moving towards each other. Therefore, the third predicate of *grouping* guarantees that after at most $|C|$ rounds the two groups will either merge

(in this case they terminate), or they will become blocked on the endpoints of the same edge until the *termination condition* will be satisfied.

(2) In this case, the agents of S_1 are divided into two groups at round $r_1 + 2|C|$. During the first $2|C|$ rounds, some of the agents of S_2 may reach u, thus enter into state *gathering* and continue moving as a group with G_1.

(a) If the agents of G_2 move clockwise towards u, then the rest of the agents of S_2 may cross the agents of G_2 or arrive on the same node. In both cases they will merge into a single group (third predicate of *grouping*).

(b) If the agents of G_2 move counter-clockwise towards u, then the rest of the agents of S_2 end up on the same node with the agents of G_2. This is because the agents of G_2 remain blocked long enough that at round $r_1 + 2|C|$ they did not reach u. Then, all of the agents in the clockwise path from G_2 to u, after $2|C|$ rounds reach G_2 (by Lemma 6.18). In this case, the agents of S_2 reverse direction to clockwise (due to the third predicate of *grouping*), however G_2 will continue moving counter-clockwise. Then, they reverse to their initial direction (counter-clockwise), and in the next round, if the edge is not missing, they again reach G_2. This procedure continues until some agent in G_2 enters into the *second step* of phase 2. Then, they will cross each other and *grouping* guarantees that they will merge into a single group.

Finally, in both cases (a) and (b), all agents in state *walking* (S_2) are absorbed by the agents in state *gathering* (S_1). Then, the two groups of agents move towards each other and *grouping* guarantees that after $|C|$ rounds they achieve *weak gathering*.

In all these cases, all agents reach either the same node and the *termination condition* is satisfied, or they become blocked at the endpoints of the same missing edge where, by Lemma 6.17, they solve *weak gathering*. □

6.5 Open Problems

In [59] the authors provide a mechanism which avoids agent crossing on the ring. In particular, each agent constructs an edge labeled bidirectional ring, such that the intersection of the labels assigned in the edges of the clockwise direction with the ones of the counter-clockwise direction is empty. Then, the agents move on the actual ring subject to the constraint that at round r they can traverse an edge only if the set of labels of that edge contains r. This guarantees that two agents moving in opposite directions will never cross each other on an edge of the actual ring. An immediate open problem is to examine whether that, or a similar, mechanism could be adapted and used in our algorithm. In our algorithm, *cross detection* is only required during the second phase. All agents after $O(n^2 + nk)$ rounds enter into that phase and elect the same node ℓ as leader, thus, obtain the same sense of orientation. By implementing a counter for the rounds, we could then allow the

agents to move clockwise on the cycle only at odd rounds, and counter-clockwise only at even rounds. This guarantees that no two agents could traverse the same edge in opposite directions during the same round. Then, by slightly modifying the second phase of our algorithm (e.g., allow $4|C|$ rounds during the *first step* in state *gathering*, and $2|C|$ during the *second step*), we conjecture that the agents will again solve *weak gathering*.

Even though we almost completely characterized the class of 1-interval connected graphs in which gathering can be solved, there is a number of interesting directions emanating from our existing knowledge on the problem. An immediate open problem is whether we can achieve the same results if the class of dynamics is the T-interval connectivity, for $T > 1$. Other dynamic models that can be considered are *periodic*, that is, each edge is periodically enabled/disabled, *recurrent* [30], meaning that an edge cannot remain disabled indefinitely, and other worst-case dynamic networks in which the topology may change arbitrarily from round to round subject to some constraints (cf., e.g., [106]). The problem of strict *gathering* becomes feasible in these cases, and the goal is to find efficient algorithms for the problem. For example, consider a ring graph and two groups of agents blocked on the endpoints of a missing edge. Then, our algorithm could eventually achieve strict *gathering* by just waiting for the disabled edge to become enabled, rather than terminate after $O(n)$ rounds. However, a more efficient algorithm could decide to change the orientation of the agents and meet on some other node of the cycle. This might be the case for many *strict gathering* algorithms for static graphs. By simulating any such algorithm, while the agents just wait for the missing edges to become enabled, it might be possible to solve *strict gathering* in all the solvable cases of static graphs. However, this technique may not be applied to algorithms that are based on synchronization of agents.

Other interesting related problems for the generalized 1-interval connectivity model are *partial gathering*, and *gathering with waste*. The *partial gathering* problem requires, for a given positive integer g, that each agent should move to a node and terminate so that at least g agents should meet at each of the nodes they terminate at. This is a generalization of the *strict gathering* problem, and for values of $g \leq k/2$ it enables feasibility in a larger class of graphs. It is not clear whether this requirement is weaker of stronger than that of *weak gathering*. For example, in ring graphs with k agents and $g = k/2$, the agents can terminate in any two nodes of the graph, provided that the number of agents in both nodes are $k/2$. However, observe that this problem enables feasibility in a larger class of graphs. Consider two cycles that are connected with a line. *Weak gathering* is unsolvable in this setting, while *partial gathering* might be possible. Consider the case that $k/2 + a$, $a < k/2$ agents are on

a cycle C_1 and the rest of them are on the cycle C_2. Then, a agents from C_1 is possible to escape from the cycle and reach C_2, thus achieve *partial gathering*. *Partial gathering with waste* g is the problem of gathering at least g agents on some node (the rest of them being the waste). Similarly to *partial gathering*, at most one agent might remain trapped on a cycle in this dynamic model. Finally, a generalization of *weak gathering*, where the agents are allowed to gather in at most g nodes (*grouping*), might also enable feasibility in a larger class of (dynamic) graphs.

Chapter 7

Conclusions

In this thesis we work on four areas where stability achievement is critical, and in each of them we examine fundamental problems. In particular, we work on three models of distributed computation, namely population protocols, network constructors, and mobile agents that are moving on dynamic graphs. Finally, we provide local search algorithms for the Crystal Structure Prediction problem. A number of interesting questions remain to be answered in each of them, as described in the corresponding chapters.

In Chapter 4 we worked on the Network Constructors model, and we examined the case where crash faults may occur on the nodes. We almost characterized the class of graphs that can be constructed under a bounded and unbounded number of faults. We introduced some minimal form of fault notifications and we examined two cases; partial constructibility, that is, the protocol is allowed to construct the graph in a subset of the population, and the case where the agents have non-constant memory. For the first case we presented a generic constructor that utilizes random bits in order to simulate a Turing Machine that constructs any constructible graph language, and for the second case we presented a protocol that restarts the population whenever a fault occurs. As part of that protocol, a standard pairwise elimination protocol for leader election is executed. The leader is then responsible to restart the population when necessary. This sub-protocol is slow, taking $O(n)$ parallel time to elect a unique leader, therefore an interesting question is whether we can utilize our approach on leader election in population protocols to speed up the execution time of our NET restarting protocol. This task is not trivial as we need to carefully handle cases where the unique leader crashes, and a new one has to be re-introduced in the population. Our leader election protocol of Chapter 3 is based on epidemic processes, which require $O(\log n)$ parallel time to reach the whole population. A possible application of this approach in our restarting protocol could be the following; in our restarting protocol, the agents' state consists of two components, S_1, which runs the restarting

protocol, and S_2 which runs the given protocol. We can additionally have a third component S_3 that runs our leader election protocol, and whenever a node is notified about a crash failure, becomes a leader again and increases its phase by one. This will guarantee that the new leaders' tuple will dominate the rest of them and eventually, in $O(\frac{\log^2 n}{\log m})$ parallel time, a single leader will remain in the population.

In Chapter 6 we worked on a model where a set of agents is moving on dynamic graphs, and in particular we presented a protocol that solves *weak gathering* in dynamic unicyclic graphs, in a fault-free setting. There is a growing interest in problems such as gathering, exploration, and black hole search in dynamic settings recently, and several model variations and assumptions have been considered. However, to the best of our knowledge, none of them has considered faulty agents moving on dynamic graphs. An interesting research direction would be to consider scenarios where the agents may crash, while the rest of them still have to solve the problem. This is not always trivial to do; for example, regarding the gathering problem, solutions that are based on counting the number of agents and terminate when they are all gathered would not work in that case, as the agents are unaware of the number of remaining agents. Would a failure detector, similar to the one that we presented in Chapter 4.4, or a variation of that, help to overcome this problem? Alternatively, we could assume that any agent passing by a node where some other agent has crashed is informed about the crash failure, or that the rest of the agents are completely unaware about crash failures and examine alternative solutions to the problem that are not based on this approach. Finally, a stronger form of faults may also be considered, such as Byzantine faults.

In Chapter 5 we examined the problem of Crystal Structure Prediction from a theoretical computer science perspective, and we provided local search algorithms that explore the search space in order to find lower energy crystal structures. All of the algorithms that have been proposed for this problem are centralized, meaning that the algorithm makes all decisions of where each atom will be placed in each step. A very interesting yet ambitious approach would be to model crystal structures in a similar way as Network Constructors. Each combination of unit cell parameters and arrangement of atoms corresponds to an energy, and each pair of atoms contributes to that overall energy based on their types and distance between them. A possible way to think of this problem would be to consider each atom as an agent, and that ionic bonds between the atoms are translated to connections between agents. We could additionally consider edge states that translate to distance between the atoms. Information about the types of the atoms can be embedded into the agents' states. For instance, a state o can indicate that the agent corresponds to an oxygen atom. The only constraint is to maintain the correct proportion of states in the initial population, which is defined by the composition of the material. Then, a protocol

that captures laws of crystal chemistry on how the atoms interact with each other and form ionic bonds could prove to help identify low energy crystal structures.

Bibliography

[1] Duncan Adamson, Argyrios Deligkas, Vladimir V. Gusev, and Igor Potapov. “On the Hardness of Energy Minimisation for Crystal Structure Prediction”. In: *SOFSEM 2020: Theory and Practice of Computer Science - 46th International Conference on Current Trends in Theory and Practice of Informatics, SOFSEM 2020, Limassol, Cyprus, January 20-24, 2020, Proceedings*. 2020, pp. 587–596. DOI: 10.1007/978-3-030-38919-2_48. URL: https://doi.org/10.1007/978-3-030-38919-2_48.

[2] Dan Alistarh, James Aspnes, David Eisenstat, Rati Gelashvili, and Ronald L Rivest. “Time-space trade-offs in population protocols”. In: *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM. 2017, pp. 2560–2579.

[3] Dan Alistarh and Rati Gelashvili. “Polylogarithmic-Time Leader Election in Population Protocols”. In: *42nd International Colloquium on Automata, Languages, and Programming (ICALP)*. Vol. 9135. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2015, pp. 479 –491.

[4] Maximilian Amsler and Stefan Goedecker. “Crystal structure prediction using the minima hopping method”. In: *The Journal of chemical physics* 133.22 (2010), p. 224104.

[5] Aris Anagnostopoulos, Ravi Kumar, Mohammad Mahdian, Eli Upfal, and Fabio Vandin. “Algorithms on evolving graphs”. In: *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. 2012, pp. 149–160.

[6] Dana Angluin. “Local and global properties in networks of processors”. In: *Proceedings of the 12th annual ACM symposium on Theory of computing (STOC)*. Los Angeles, California, USA: ACM, 1980, pp. 82–93. ISBN: 0-89791-017-6. DOI: 10.1145/800141.804655. URL: http://doi.acm.org/10.1145/800141.804655.

[7] Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. “Computation in networks of passively mobile finite-state sensors”. In: *Distributed Computing* 18.4 (2006), pp. 235–253.

[8] Dana Angluin, James Aspnes, and David Eisenstat. "A simple population protocol for fast robust approximate majority". In: *Distributed Computing* 21.2 (2008), pp. 87–102.

[9] Dana Angluin, James Aspnes, and David Eisenstat. "Fast computation by population protocols with a leader". In: *Distributed Computing* 21.3 (2008), pp. 183–199.

[10] Dana Angluin, James Aspnes, and David Eisenstat. "Stably computable predicates are semilinear". In: *25th annual ACM Symposium on Principles of Distributed Computing (PODC)*. Denver, Colorado, USA: ACM Press, 2006, pp. 292–299. ISBN: 1-59593-384-0. DOI: http://doi.acm.org/10.1145/1146381.1146425.

[11] Dana Angluin, James Aspnes, David Eisenstat, and Eric Ruppert. "The computational power of population protocols". In: *Distributed Computing* 20.4 (2007), pp. 279–304.

[12] Dana Angluin, James Aspnes, Michael J. Fischer, and Hong Jiang. "Self-stabilizing population protocols". In: *ACM Trans. Auton. Adapt. Syst.* 3.4 (2008), pp. 1–28. ISSN: 1556-4665. DOI: http://doi.acm.org/10.1145/1452001.1452003.

[13] Dmytro Antypov, Argyrios Deligkas, Vladimir Gusev, Matthew J Rosseinsky, Paul G Spirakis, and Michail Theofilatos. "Crystal Structure Prediction via Oblivious Local Search". In: *18th International Symposium on Experimental Algorithms (SEA 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2020.

[14] James Aspnes, Joffroy Beauquier, Janna Burman, and Devan Sohier. "Time and Space Optimal Counting in Population Protocols". In: *20th International Conference on Principles of Distributed Systems (OPODIS 2016)*. Vol. 70. 2017, 13:1–13:17.

[15] James Aspnes and Eric Ruppert. "An introduction to population protocols". In: *Middleware for Network Eccentric and Mobile Applications*. Ed. by Benoît Garbinato, Hugo Miranda, and Luís Rodrigues. Springer-Verlag, 2009, pp. 97–120.

[16] Chagit Attiya, Marc Snir, and Manfred Warmuth. "Computing on an anonymous ring". In: *Proceedings of the fourth annual ACM symposium on Principles of distributed computing*. PODC '85. Minaki, Ontario, Canada: ACM, 1985, pp. 196–203. ISBN: 0-89791-168-7. DOI: http://doi.acm.org/10.1145/323596.323614. URL: http://doi.acm.org/10.1145/323596.323614.

[17] Chen Avin, Michal Koucký, and Zvi Lotker. “Cover time and mixing time of random walks on dynamic graphs”. In: *Random Structures & Algorithms* 52.4 (2018), pp. 576–596.

[18] Lali Barriere, Paola Flocchini, Pierre Fraigniaud, and Nicola Santoro. “Rendezvous and election of mobile agents: impact of sense of direction”. In: *Theory of Computing Systems* 40.2 (2007), pp. 143–162.

[19] Joffroy Beauquier, Peva Blanchard, and Janna Burman. “Self-stabilizing leader election in population protocols over arbitrary communication graphs”. In: *International Conference on Principles of Distributed Systems*. Springer. 2013, pp. 38–52.

[20] Joffroy Beauquier, Janna Burman, Simon Claviere, and Devan Sohier. “Space-optimal counting in population protocols”. In: *DISC 2015: International Symposium on Distributed Computing*. Springer. 2015, pp. 631–646.

[21] Joffroy Beauquier, Julien Clement, Stephane Messika, Laurent Rosaz, and Brigitte Rozoy. “Self-stabilizing Counting in Mobile Sensor Networks with a Base Station”. In: *Distributed Computing*. Springer Berlin Heidelberg, 2007, pp. 63–76.

[22] Joffroy Beauquier and Synnöve Kekkonen-Moneta. “On ftss-solvable distributed problems”. In: *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*. 1997, p. 290.

[23] Amanda Belleville, David Doty, and David Soloveichik. “Hardness of Computing and Approximating Predicates and Functions with Leaderless Population Protocols”. In: *44th International Colloquium on Automata, Languages, and Programming (ICALP)*. Ed. by Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl. Vol. 80. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 141:1–141:14. ISBN: 978-3-95977-041-5. DOI: 10.4230/LIPIcs.ICALP.2017.141. URL: http://drops.dagstuhl.de/opus/volltexte/2017/7504.

[24] Petra Berenbrink, George Giakkoupis, and Peter Kling. “Optimal time and space leader election in population protocols”. In: *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*. 2020, pp. 119–129.

[25] Petra Berenbrink, Dominik Kaaser, and Tomasz Radzik. “On counting the population size”. In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 2019, pp. 43–52.

[26] Richard A Buckingham. “The classical equation of state of gaseous helium, neon and argon”. In: *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences* 168.933 (1938), pp. 264–283.

[27] Richard A Buckingham. “The classical equation of state of gaseous helium, neon and argon”. In: *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences* 168.933 (1938), pp. 264–283.

[28] Janna Burman, Ho-Lin Chen, Hsueh-Ping Chen, David Doty, Thomas Nowak, Eric Severson, and Chuan Xu. “Time-optimal self-stabilizing leader election in population protocols”. In: *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*. 2021, pp. 33–44.

[29] Seth T Call, Dmitry Yu Zubarev, and Alexander I Boldyrev. “Global minimum structure searches via particle swarm optimization”. In: *Journal of computational chemistry* 28.7 (2007), pp. 1177–1186.

[30] Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. “Time-varying graphs and dynamic networks”. In: *International Journal of Parallel, Emergent and Distributed Systems* 27.5 (2012), pp. 387–408.

[31] Jérémie Chalopin, Shantanu Das, and Nicola Santoro. “Rendezvous of mobile agents in unknown graphs with faulty links”. In: *International Symposium on Distributed Computing*. Springer. 2007, pp. 108–122.

[32] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. “The weakest failure detector for solving consensus”. In: *Journal of the ACM (JACM)* 43.4 (1996), pp. 685–722.

[33] Tushar Deepak Chandra and Sam Toueg. “Unreliable failure detectors for reliable distributed systems”. In: *Journal of the ACM (JACM)* 43.2 (1996), pp. 225–267.

[34] Ioannis Chatzigiannakis, Othon Michail, Stavros Nikolaou, Andreas Pavlogiannis, and Paul G. Spirakis. “Passively mobile communicating machines that use restricted space”. In: *Theoretical Computer Science* 412.46 (2011), pp. 6469–6483.

[35] Ho-Lin Chen, David Doty, and David Soloveichik. “Deterministic function computation with chemical reaction networks”. In: *Nat. Comput. 7* (2014), pp. 517 –534.

[36] C Collins, GR Darling, and MJ Rosseinsky. “The Flexible Unit Structure Engine (FUSE) for probe structure-based composition prediction”. In: *Faraday discussions* 211 (2018), pp. 117–131.

[37] C Collins, MS Dyer, MJ Pitcher, GFS Whitehead, M Zanella, P Mandal, JB Claridge, GR Darling, and MJ Rosseinsky. “Accelerated discovery of two crystal structure types in a complex inorganic phase field”. In: *Nature* 546.7657 (2017), p. 280.

[38] Colin Cooper, Anissa Lamani, Giovanni Viglietta, Masafumi Yamashita, and Yukiko Yamauchi. “Constructing self-stabilizing oscillators in population protocols”. In: *Information and Computation* 255 (2017), pp. 336–351.

[39] Stefano Curtarolo, Gus LW Hart, Marco Buongiorno Nardelli, Natalio Mingo, Stefano Sanvito, and Ohad Levy. “The high-throughput highway to computational materials design”. In: *Nature materials* 12.3 (2013), p. 191.

[40] Stefano Curtarolo, Dane Morgan, Kristin Persson, John Rodgers, and Gerbrand Ceder. “Predicting crystal structures with data mining of quantum calculations”. In: *Physical review letters* 91.13 (2003), p. 135503.

[41] Jurek Czyzowicz, Stefan Dobrev, Evangelos Kranakis, and Danny Krizanc. “The power of tokens: rendezvous and symmetry detection for two mobile agents in a ring”. In: *International Conference on Current Trends in Theory and Practice of Computer Science*. Springer. 2008, pp. 234–246.

[42] Jurek Czyzowicz, Andrzej Pelc, and Arnaud Labourel. “How to meet asynchronously (almost) everywhere”. In: *ACM Transactions on Algorithms (TALG)* 8.4 (2012), pp. 1–14.

[43] Shantanu Das. “Mobile agents in distributed computing: Network exploration”. In: *Bulletin of EATCS* 1.109 (2013).

[44] Shantanu Das, Giuseppe Antonio Di Luna, Paola Flocchini, Nicola Santoro, and Giovanni Viglietta. “Mediated Population Protocols: Leader Election and Applications”. In: *International Conference on Theory and Applications of Models of Computation*. Springer. 2017, pp. 172–186.

[45] Shantanu Das, Paola Flocchini, Shay Kutten, Amiya Nayak, and Nicola Santoro. “Map construction of unknown graphs by multiple agents”. In: *Theoretical Computer Science* 385.1-3 (2007), pp. 34–48.

[46] Shantanu Das, Paola Flocchini, Amiya Nayak, and Nicola Santoro. “Distributed exploration of an unknown graph”. In: *International Colloquium on Structural Information and Communication Complexity*. Springer. 2005, pp. 99–114.

[47] Shantanu Das, Nikos Giachoudis, Flaminia L. Luccio, and Euripides Markou. “Broadcasting with mobile agents in dynamic networks”. In: *24th International Conference on Principles of Distributed Systems (OPODIS)* (2020).

[48] Joshua J Daymude, Zahra Derakhshandeh, Robert Gmyr, Alexandra Porter, Andréa W Richa, Christian Scheideler, and Thim Strothmann. “On the runtime of universal coating for programmable matter”. In: *Natural Computing* 17.1 (2018), pp. 81–96.

[49] Gianluca De Marco, Luisa Gargano, Evangelos Kranakis, Danny Krizanc, Andrzej Pelc, and Ugo Vaccaro. “Asynchronous deterministic rendezvous in graphs.” In: *Theor. Comput. Sci.* 355.3 (2006), pp. 315–326.

[50] David M Deaven and Kai-Ming Ho. “Molecular geometry optimization with a genetic algorithm”. In: *Physical review letters* 75.2 (1995), p. 288.

[51] Carole Delporte-Gallet, Stéphane Devismes, and Hugues Fauconnier. “Stabilizing leader election in partial synchronous systems with crash failures”. In: *Journal of Parallel and Distributed Computing* 70.1 (2010), pp. 45–58.

[52] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Eric Ruppert. “When Birds Die: Making Population Protocols Fault-Tolerant”. In: *IEEE 2nd Intl Conference on Distributed Computing in Sensor Systems (DCOSS)*. Vol. 4026. Lecture Notes in Computer Science. Springer-Verlag, 2006, pp. 51–66.

[53] Zahra Derakhshandeh, Shlomi Dolev, Robert Gmyr, Andréa W Richa, Christian Scheideler, and Thim Strothmann. “Brief announcement: amoebot–a new model for programmable matter”. In: *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*. ACM. 2014, pp. 220–222.

[54] Anders Dessmark, Pierre Fraigniaud, Dariusz R Kowalski, and Andrzej Pelc. “Deterministic rendezvous in graphs”. In: *Algorithmica* 46.1 (2006), pp. 69–96.

[55] Anders Dessmark, Pierre Fraigniaud, and Andrzej Pelc. “Deterministic rendezvous in graphs”. In: *European Symposium on Algorithms*. Springer. 2003, pp. 184–195.

[56] Giuseppe A Di Luna, Paola Flocchini, Nicola Santoro, Giovanni Viglietta, and Yukiko Yamauchi. “Shape formation by programmable particles”. In: *Distributed Computing* (2019), pp. 1–33. DOI: `https://doi.org/10.1007/s00446-019-00350-6`.

[57] Giuseppe Antonio Di Luna, Stefan Dobrev, Paola Flocchini, and Nicola Santoro. “Live exploration of dynamic rings”. In: *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2016, pp. 570–579.

[58] Giuseppe Antonio Di Luna, Paola Flocchini, Taisuke Izumi, Tomoko Izumi, Nicola Santoro, and Giovanni Viglietta. "Population protocols with faulty interactions: the impact of a leader". In: *International Conference on Algorithms and Complexity (CIAC)*. Springer. 2017, pp. 454–466.

[59] Giuseppe Antonio Di Luna, Paola Flocchini, Linda Pagli, Giuseppe Prencipe, Nicola Santoro, and Giovanni Viglietta. "Gathering in dynamic rings". In: *Theoretical Computer Science* 811 (2020), pp. 79–98.

[60] Shlomi Dolev. *Self-stabilization*. Cambridge, MA, USA: MIT Press, 2000. ISBN: 0-262-04178-2.

[61] Shlomi Dolev, Amos Israeli, and Shlomo Moran. "Self-stabilization of dynamic systems assuming only read/write atomicity". In: *Distributed Computing* 7.1 (1993), pp. 3–16. ISSN: 1432-0452. DOI: `10.1007/BF02278851`. URL: `https://doi.org/10.1007/BF02278851`.

[62] David Doty. "Timing in chemical reaction networks". In: *Proc. of the 25th Annual ACM-SIAM Symp. on Discrete Algorithms (SODA)*. 2014, pp. 772–784.

[63] David Doty and Mahsa Eftekhari. "A survey of size counting in population protocols". In: *arXiv preprint arXiv:2105.05408* (2021).

[64] David Doty and Mahsa Eftekhari. "Efficient size estimation and impossibility of termination in uniform dense population protocols". In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 2019, pp. 34–42.

[65] David Doty, Mahsa Eftekhari, Leszek Gasieniec, Eric Severson, Grzegorz Stachowiak, and Przemysław Uznański. "A time and space optimal stable population protocol solving exact majority". In: *arXiv preprint arXiv:2106.10201* (2021).

[66] David Doty, Mahsa Eftekhari, Othon Michail, Paul G. Spirakis, and Michail Theofilatos. "Exact size counting in uniform population protocols in nearly logarithmic time". In: *CoRR* abs/1805.04832 (2018). arXiv: `1805.04832`. URL: `http://arxiv.org/abs/1805.04832`.

[67] David Doty and David Soloveichik. "Stable leader election in population protocols requires linear time". In: *Distributed Computing* 31.4 (2018), pp. 257–271.

[68] Swan Dubois, Toshimitsu Masuzawa, and Sébastien Tixeuil. "Maximum metric spanning tree made byzantine tolerant". In: *Algorithmica* 73.1 (2015), pp. 166–201.

[69] Bertrand Ducourthial and Sébastien Tixeuil. “Self-stabilization with r-operators”. In: *Distributed Computing* 14.3 (2001), pp. 147–162. ISSN: 1432-0452. DOI: 10.1007/PL00008934. URL: https://doi.org/10.1007/PL00008934.

[70] Thomas Erlebach, Michael Hoffmann, and Frank Kammer. “On temporal graph exploration”. In: *Journal of Computer and System Sciences* 119 (2021), pp. 1–18.

[71] Ji Feng, Wojciech Grochala, Tomasz Jaroń, Roald Hoffmann, Aitor Bergara, and NW Ashcroft. “Structures and potential superconductivity in SiH 4 at high pressure: En route to “metallic hydrogen””. In: *Physical Review Letters* 96.1 (2006), p. 017006.

[72] Christopher C Fischer, Kevin J Tibbetts, Dane Morgan, and Gerbrand Ceder. “Predicting crystal structure by merging data mining with quantum mechanics”. In: *Nature materials* 5.8 (2006), pp. 641–646.

[73] Michael Fischer and Hong Jiang. “Self-stabilizing Leader Election in Networks of Finite-State Anonymous Agents”. In: *OPODIS* vol 4305 (2006).

[74] Paola Flocchini, Bernard Mans, and Nicola Santoro. “On the exploration of time-varying networks”. In: *Theoretical Computer Science* 469 (2013), pp. 53–68.

[75] Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. “Distributed Computing by Mobile Entities”. In: *Current Research in Moving and Computing* 11340 (2019).

[76] CM Freeman, JM Newsam, SM Levine, and CRA Catlow. “Inorganic crystal structure prediction using simplified potentials and experimental unit cells: application to the polymorphs of titanium dioxide”. In: *Journal of Materials Chemistry* 3.5 (1993), pp. 531–535.

[77] Julian D Gale and Andrew L Rohl. “The general utility lattice program (GULP)”. In: *Molecular Simulation* 29.5 (2003), pp. 291–341.

[78] Leszek Gasieniec and Grzegorz Staehowiak. “Fast space optimal leader election in population protocols”. In: *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2018, pp. 2653–2667.

[79] Leszek Gasieniec and Grzegorz Staehowiak. “Fast space optimal leader election in population protocols”. In: *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2018, pp. 2653–2667.

[80] Romain Gautier, Xiuwen Zhang, Linhua Hu, Liping Yu, Yuyuan Lin, Tor OL Sunde, Danbee Chon, Kenneth R Poeppelmeier, and Alex Zunger. “Prediction and accelerated laboratory discovery of previously unknown 18-electron ABX compounds”. In: *Nature chemistry* 7.4 (2015), p. 308.

[81] Kyle Gilpin, Ara Knaian, and Daniela Rus. "Robot pebbles: One centimeter modules for programmable matter through self-disassembly". In: *Robotics and Automation (ICRA), 2010 IEEE International Conference on*. IEEE. 2010, pp. 2485–2492.

[82] Stefan Goedecker. "Minima hopping: An efficient search method for the global minimum of the potential energy surface of complex molecular systems". In: *The Journal of chemical physics* 120.21 (2004), pp. 9911–9917.

[83] Chaohui Gong, Stephen Tully, George Kantor, and Howie Choset. "Multi-agent deterministic graph mapping via robot rendezvous". In: *2012 IEEE International Conference on Robotics and Automation*. IEEE. 2012, pp. 1278–1283.

[84] Tsuyoshi Gotoh, Paola Flocchini, Toshimitsu Masuzawa, and Nicola Santoro. "Tight bounds on distributed exploration of temporal graphs". In: *23rd International Conference on Principles of Distributed Systems (OPODIS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2020.

[85] Tsuyoshi Gotoh, Yuichi Sudo, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. "Group exploration of dynamic tori". In: *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2018, pp. 775–785.

[86] Nabil Guellati and Hamamache Kheddouci. "A survey on self-stabilizing algorithms for independence, domination, coloring, and matching in graphs". In: *Journal of Parallel and Distributed Computing* 70.4 (2010), pp. 406 – 415. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2009.11.006. URL: http://www.sciencedirect.com/science/article/pii/S0743731509002299.

[87] Rachid Guerraoui and Eric Ruppert. "Names Trump Malice: Tiny Mobile Agents Can Tolerate Byzantine Failures". In: *36th International Colloquium on Automata, Languages and Programming (ICALP)*. Vol. 5556. LNCS Part 2. Springer-Verlag, 2009, pp. 484–495.

[88] Geoffroy Hautier, Chris Fischer, Virginie Ehrlacher, Anubhav Jain, and Gerbrand Ceder. "Data mined ionic substitutions for the discovery of new compounds". In: *Inorganic chemistry* 50.2 (2010), pp. 656–663.

[89] Detlef WM Hofmann and Joannis Apostolakis. "Crystal structure prediction by data mining". In: *Journal of Molecular Structure* 647.1-3 (2003), pp. 17–39.

[90] Petter Holme and Jari Saramäki. "Temporal networks". In: *Physics reports* 519.3 (2012), pp. 97–125.

[91] Tomoko Izumi, Keigo Kinpara, Taisuke Izumi, and Koichi Wada. "Space-efficient self-stabilizing counting population protocols on mobile sensor networks". In: *Theor. Comput. Sci.* 552 (2014), pp. 99–108.

[92] Frank Jensen. *Introduction to computational chemistry*. John wiley & sons, 2017.

[93] Ekkehard Köhler, Katharina Langkau, and Martin Skutella. "Time-expanded graphs for flow-dependent transit times". In: *European symposium on algorithms*. Springer. 2002, pp. 599–611.

[94] Vassilis Kostakos. "Temporal graphs". In: *Physica A: Statistical Mechanics and its Applications* 388.6 (2009), pp. 1007–1023.

[95] Evangelos Kranakis, Danny Krizanc, and Euripides Markou. "The mobile agent rendezvous problem in the ring". In: *Synthesis Lectures on Distributed Computing Theory* 1.1 (2010), pp. 1–122.

[96] Fabian Kuhn, Nancy Lynch, and Rotem Oshman. "Distributed computation in dynamic networks". In: *Proceedings of the 42nd ACM symposium on Theory of computing (STOC)*. Cambridge, Massachusetts, USA: ACM, 2010, pp. 513–522. ISBN: 978-1-4503-0050-6. DOI: http://doi.acm.org/10.1145/1806689.1806760. URL: http://doi.acm.org/10.1145/1806689.1806760.

[97] Alessandro Laio, Antonio Rodriguez-Fortea, Francesco Luigi Gervasio, Matteo Ceccarelli, and Michele Parrinello. "Assessing the accuracy of metadynamics". In: *The journal of physical chemistry B* 109.14 (2005), pp. 6714–6721.

[98] GV Lewis and CRA Catlow. "Potential models for ionic oxides". In: *Journal of Physics C: Solid State Physics* 18.6 (1985), p. 1149.

[99] David C Lonie and Eva Zurek. "XtalOpt: An open-source evolutionary algorithm for crystal structure prediction". In: *Computer Physics Communications* 182.2 (2011), pp. 372–387.

[100] Giuseppe Antonio Di Luna, Roberto Baldoni, Silvia Bonomi, and Ioannis Chatzigiannakis. "Counting in Anonymous Dynamic Networks under Worst-Case Adversary". In: *IEEE 34th International Conference on Distributed Computing Systems (ICDCS)* (2014).

[101] John Maddox. "Crystals from first principles". In: *Nature* 335.6187 (1988), pp. 201–201.

[102] Naoki Masuda and Renaud Lambiotte. *A guide to temporal networks*. World Scientific, 2016.

[103] Othon Michail. "An introduction to temporal graphs: An algorithmic perspective". In: *Internet Mathematics* 12.4 (2016), pp. 239–280.

[104] Othon Michail. “Terminating distributed construction of shapes and patterns in a fair solution of automata”. In: *Distributed Computing* 31.5 (2018), pp. 343–365. ISSN: 0178-2770. DOI: http://dx.doi.org/10.1007/s00446-017-0309-z. URL: http://link.springer.com/article/10.1007/s00446-017-0309-z.

[105] Othon Michail. “Terminating distributed construction of shapes and patterns in a fair solution of automata”. In: *Distributed Computing* 31.5 (2018), pp. 343–365.

[106] Othon Michail, Ioannis Chatzigiannakis, and Paul G Spirakis. “Causality, influence, and computation in possibly disconnected synchronous dynamic networks”. In: *Journal of Parallel and Distributed Computing* 74.1 (2014), pp. 2016–2026.

[107] Othon Michail, Ioannis Chatzigiannakis, and Paul G. Spirakis. “Mediated population protocols”. In: *Theoretical Computer Science* 412.22 (2011), pp. 2434–2450. ISSN: 0304-3975. DOI: http://dx.doi.org/10.1016/j.tcs.2011.02.003. URL: http://dx.doi.org/10.1016/j.tcs.2011.02.003.

[108] Othon Michail, Ioannis Chatzigiannakis, and Paul G Spirakis. “Naming and counting in anonymous unknown dynamic networks”. In: *15th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*. Springer, 2013, pp. 281–295.

[109] Othon Michail, Ioannis Chatzigiannakis, and Paul G. Spirakis. *New Models for Population Protocols*. Ed. by Nancy A. Lynch. N. A. Lynch (Ed), Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool, 2011.

[110] Othon Michail, George Skretas, and Paul G Spirakis. “On the Transformation Capability of Feasible Mechanisms for Programmable Matter”. In: *Journal of Computer and System Sciences* 102 (2019), pp. 18–39.

[111] Othon Michail and Paul G Spirakis. “Elements of the Theory of Dynamic Networks”. In: *Communications of the ACM* 61.2 (2018).

[112] Othon Michail and Paul G. Spirakis. “Network Constructors: A Model for Programmable Matter”. In: *SOFSEM 2017: Theory and Practice of Computer Science*. Ed. by Bernhard Steffen, Christel Baier, Mark van den Brand, Johann Eder, Mike Hinchey, and Tiziana Margaria. Cham: Springer International Publishing, 2017, pp. 15–34. ISBN: 978-3-319-51963-0.

[113] Othon Michail and Paul G. Spirakis. "Simple and efficient local codes for distributed stable network construction". In: *Distributed Computing* 29.3 (2016), pp. 207–237. ISSN: 0178-2770. DOI: `http://dx.doi.org/10.1007/s00446-015-0257-4`. URL: `https://link.springer.com/article/10.1007/s00446-015-0257-4`.

[114] Othon Michail, Paul G Spirakis, and Michail Theofilatos. "Beyond Rings: Gathering in 1-Interval Connected Graphs". In: *Parallel Processing Letters* (2021), p. 2150020.

[115] Othon Michail, Paul G Spirakis, and Michail Theofilatos. "Brief Announcement: Fast Approximate Counting and Leader Election in Populations". In: *International Colloquium on Structural Information and Communication Complexity.* Springer. 2018, pp. 38–42.

[116] Othon Michail, Paul G Spirakis, and Michail Theofilatos. "Fault tolerant network constructors". In: *International Symposium on Stabilizing, Safety, and Security of Distributed Systems.* Springer. 2019, pp. 243–255.

[117] Othon Michail, Paul G. Spirakis, and Michail Theofilatos. "Simple and Fast Approximate Counting and Leader Election in Populations". In: *Stabilization, Safety, and Security of Distributed Systems.* Ed. by Taisuke Izumi and Petr Kuznetsov. Also as a short paper in the *International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, 2018. Cham: Springer International Publishing, 2018, pp. 154–169. ISBN: 978-3-030-03232-6.

[118] Othon Michail, Paul G Spirakis, and Michail Theofilatos. "Simple and fast approximate counting and leader election in populations". In: *International Symposium on Stabilizing, Safety, and Security of Distributed Systems.* Springer. 2018, pp. 154–169.

[119] Ryu Mizoguchi, Hirotaka Ono, Shuji Kijima, and Masafumi Yamashita. "On space complexity of self-stabilizing leader election in mediated population protocol". In: *Distributed Computing* 25.6 (2012), pp. 451–460.

[120] Katarzyna Musiał and Przemysław Kazienko. "Social networks on the internet". In: *World Wide Web* 16.1 (2013), pp. 31–72.

[121] Michael J Neely, Eytan Modiano, and Charles E Rohrs. "Dynamic power allocation and routing for time varying wireless networks". In: *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No. 03CH37428).* Vol. 1. IEEE. 2003, pp. 745–755.

[122] Mikhail Nesterenko and Anish Arora. "Dining philosophers that tolerate malicious crashes". In: *Proceedings 22nd International Conference on Distributed Computing Systems*. IEEE. 2002, pp. 191–198.

[123] Nicola Nosengo. "Can artificial intelligence create the next wonder material?" In: *Nature News* 533.7601 (2016), p. 22.

[124] Artem R Oganov and Colin W Glass. "Crystal structure prediction using ab initio evolutionary techniques: Principles and applications". In: *The Journal of chemical physics* 124.24 (2006), p. 244704.

[125] Artem R Oganov, Chris J Pickard, Qiang Zhu, and Richard J Needs. "Structure prediction drives materials discovery". In: *Nature Reviews Materials* (2019), p. 1.

[126] A James O'malley and Peter V Marsden. "The analysis of social networks". In: *Health services and outcomes research methodology* 8.4 (2008), pp. 222–269.

[127] Petrişor Panaite and Andrzej Pelc. "Exploring unknown undirected graphs". In: *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 1998, pp. 316–322.

[128] J Pannetier, J Bassas-Alsina, J Rodriguez-Carvajal, and V Caignaert. "Prediction of crystal structures from crystal chemistry rules by simulated annealing". In: *Nature* 346.6282 (1990), p. 343.

[129] Andrzej Pelc. "Deterministic rendezvous in networks: A comprehensive survey". In: *Networks* 59.3 (2012), pp. 331–347.

[130] David Peleg. "As Good as It Gets: Competitive Fault Tolerance in Network Structures". In: *Stabilization, Safety, and Security of Distributed Systems*. Ed. by Rachid Guerraoui and Franck Petit. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[131] Chris J Pickard. "Real-space pairwise electrostatic summation in a uniform neutralizing background". In: *Physical Review Materials* 2.1 (2018), p. 013806.

[132] Chris J Pickard and RJ Needs. "Ab initio random structure searching". In: *Journal of Physics: Condensed Matter* 23.5 (2011), p. 053201.

[133] Chris J Pickard and RJ Needs. "High-pressure phases of silane". In: *Physical Review Letters* 97.4 (2006), p. 045504.

[134] Cauligi S Raghavendra, Krishna M Sivalingam, and Taieb Znati. *Wireless sensor networks*. Springer, 2006.

[135] Ryan A Rossi, Brian Gallagher, Jennifer Neville, and Keith Henderson. "Modeling dynamic behavior in large evolving graphs". In: *Proceedings of the sixth ACM international conference on Web search and data mining.* 2013, pp. 667–676.

[136] Nicola Santoro, Walter Quattrociocchi, Paola Flocchini, Arnaud Casteigts, and Frédéric Amblard. "Time-Varying Graphs and Social Network Analysis: Temporal Indicators and Metrics". In: *Workshop on Social Networks And MultiAgent Systems (SNAMAS).* Leicester, United Kingdom, Apr. 2011, pp. 32–38. URL: https://hal.archives-ouvertes.fr/hal-00854313.

[137] Martin U Schmidt and Ulli Englert. "Prediction of crystal structures". In: *Journal of the Chemical Society, Dalton Transactions* 10 (1996), pp. 2077–2082.

[138] J Christian Schön. "How can Databases assist with the Prediction of Chemical Compounds?" In: *Zeitschrift für anorganische und allgemeine Chemie* 640.14 (2014), pp. 2717–2726.

[139] J Christian Schön and Martin Jansen. "First step towards planning of syntheses in solid-state chemistry: determination of promising structure candidates by global optimization". In: *Angewandte Chemie International Edition in English* 35.12 (1996), pp. 1286–1304.

[140] Masahiro Shibata, Norikazu Kawata, Yuichi Sudo, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. "Move-optimal partial gathering of mobile agents without identifiers or global knowledge in asynchronous unidirectional rings". In: *Theoretical Computer Science* 822 (2020), pp. 92–109.

[141] Theo Siegrist and Terrell A Vanderah. "Combining Magnets and Dielectrics: Crystal Chemistry in the BaO- Fe2O3- TiO2 System". In: *European Journal of Inorganic Chemistry* 2003.8 (2003), pp. 1483–1501.

[142] David Soloveichik, Matthew Cook, Erik Winfree, and Jehoshua Bruck. "Computation with finite stochastic chemical reaction networks". In: *Nat. Comput. 7* (2008), pp. 615 –633.

[143] Yuichi Sudo, Fukuhito Ooshita, Taisuke Izumi, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. "Logarithmic expected-time leader election in population protocol model". In: *International Symposium on Stabilizing, Safety, and Security of Distributed Systems.* Springer. 2019, pp. 323–337.

[144] TA Vanderah, JM Loezos, and RS Roth. "Magnetic dielectric oxides: subsolidus phase relations in the BaO: Fe2O3: TiO2system". In: *Journal of Solid State Chemistry* 121.1 (1996), pp. 38–50.

[145] P Villars, K Brandenburg, M Berndt, S LeClair, A Jackson, Y-H Pao, B Igelnik, M Oxley, B Bakshi, P Chen, et al. "Binary, ternary and quaternary compound former/nonformer prediction via Mendeleev number". In: *Journal of alloys and compounds* 317 (2001), pp. 26–38.

[146] David J Wales and Jonathan PK Doye. "Global optimization by basin-hopping and the lowest energy structures of Lennard-Jones clusters containing up to 110 atoms". In: *The Journal of Physical Chemistry A* 101.28 (1997), pp. 5111–5116.

[147] David J Wales and Jonathan PK Doye. "Global optimization by basin-hopping and the lowest energy structures of Lennard-Jones clusters containing up to 110 atoms". In: *The Journal of Physical Chemistry A* 101.28 (1997), pp. 5111–5116.

[148] Jearl D Walker. *Fundamentals of physics extended.* Wiley, 2010.

[149] Yanchao Wang, Jian Lv, Li Zhu, and Yanming Ma. "Crystal structure prediction via particle-swarm optimization". In: *Physical Review B* 82.9 (2010), p. 094116.

[150] Yanchao Wang, Jian Lv, Li Zhu, and Yanming Ma. "Crystal structure prediction via particle-swarm optimization". In: *Physical Review B* 82.9 (2010), p. 094116.

[151] Stanley Wasserman, Katherine Faust, et al. "Social network analysis: Methods and applications". In: (1994).

[152] Scott M Woodley, Peter D Battle, Julian D Gale, and C Richard A Catlow. "Prediction of inorganic crystal framework structures Part 1: Using a genetic algorithm and an indirect approach to exclusion zones". In: *Physical Chemistry Chemical Physics* 6.8 (2004), pp. 1815–1822.

[153] Aya Zaki, Mahmoud Attia, Doaa Hegazy, and Safaa Amin. "Comprehensive survey on dynamic graph models". In: *International Journal of Advanced Computer Science and Applications* 7.2 (2016), pp. 573–582.