DISS. ETH NO. 22941

Anonymous Distributed Computing: Computability, Randomization, and Checkability

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

JOCHEN SEIDEL

Dipl.-Inform., Karlsruhe Institute of Technology, Germany

born on 18.5.1984

citizen of
Germany

accepted on the recommendation of

Prof. Dr. Roger Wattenhofer, examiner
Prof. Dr. Yuval Emek, co-examiner
Prof. Dr. Jukka Suomela, co-examiner

2015

Abstract

This dissertation studies various aspects of computing in anonymous networks, where nodes are not equipped with unique identifiers. Nodes in the network exchange messages and each node computes some local output; the global output of the network is the combination of all local outputs. We focus mainly on randomized algorithms, beginning with the question "What can be computed in an anonymous network?".

Two classes of problems solvable in anonymous networks are defined, depending on whether nodes are allowed to revoke their outputs or not. We introduce and study the concept of a distributed oracle, which in yields a hierarchy of hard and complete problems for the classes. Several classic and/or characteristic problems in distributed computing are classified in terms of computability and hardness.

Access to random bits arguably has a huge impact on the computability in anonymous networks. In an effort to exactly characterize this impact, we prove that every problem that can be solved (and verified) by a *randomized* anonymous algorithm can also be solved by a *deterministic* anonymous algorithm provided that the latter is equipped with a 2-*hop coloring* of the input graph.

It is natural to ask how many random bits are required to solve any such problem. We find that the answer depends on the desired runtime of the algorithm. More precisely, we devise a randomized 2-hop coloring scheme that allows to trade an increase in runtime for a decrease in the random bit complexity. A lower bound we show yields that the trade-off achieved by our scheme is asymptotically optimal for any reasonable runtime, i.e., reducing the runtime must lead to an increase in the random bit complexity.

Lastly, we study local checkability of network properties like s-t reachability, or whether the network is acyclic or contains a cycle. A *prover* assigns a label to each node so that a *verifier* can check in constant time whether the property holds or not. We obtain asymptotically tight bounds for the label size of the latter two problems. For s-t reachability, we obtain a new asymptotically tight label size lower bound in one of our models,

and devise an emulation technique that allows us to transfer a previously known upper bound without asymptotic loss in the bit complexity in another model.

Zusammenfassung

Diese Dissertation betrachtet verschiedene Aspekte des Rechnens in anonymen Netzwerken, in denen die (Rechen-)Knoten nicht mit einem eindeutigen Namen ausgestattet sind. Die Knoten im Netzwerk tauschen Nachrichten untereinander aus und berechnen jeder eine lokale Ausgabe; die globale Ausgabe ergibt sich aus der Gesamtheit der lokalen Ausgaben. Wir betrachten hauptsächlich randomisierte Algorithmen, und stellen zuerst die Frage „Was kann in einem anonymen Netzwerk berechnet werden?“

Zwei Klassen von Problemen, die in anonymen Netzwerken lösbar sind, werden in Abhängigkeit davon, ob die Knoten ihre Ausgabe zurücknehmen können oder nicht, definiert. Das Konzept eines verteilten Orakels wird eingeführt und untersucht, und es ergibt sich eine Hierarchie mit schwierigen und vollständigen Problemen für die Klassen. Eingie klassische und/oder charakteristische Probleme aus dem Umfeld des verteilten Rechnens werden in Bezug auf Berechenbarkeit und Schwierigkeit klassifiziert.

Der Zugriff auf Zufallsbits hat bekanntermassen einen grossen Einfluss auf die Berechenbarkeit in anonymen Netzwerken. In dem Bemühen, diesen Einfluss genau zu charakterisieren, zeigen wir, dass jedes Problem, welches von einem *randomisierten* anonymen Algorithmus gelöst (und verifiziert) werden kann, auch von einem *deterministischen* anonymen Algorithmus gelöst werden kann, falls dieser eine *Abstand*-2-*Färbung* des Eingabegraphen zur Verfügung hat.

Es drängt sich die Frage auf, wie viele Zufallsbits zum Lösen solcher Probleme benötigt werden. Wir finden heraus, dass die Antwort darauf von der gewünschten Laufzeit des Algorithmus abhängt. Genauer gesagt entwickeln wir ein randomisiertes Schema für Abstand-2-Färbungen, welches erlaubt, eine höhere Laufzeit gegen eine geringere Anzahl benötigter Zufallsbits einzutauschen. Wir zeigen eine untere Schranke, die belegt, dass der Trade-off unseres Schemas asymptotisch optimal ist, d.h. eine Reduktion der Laufzeit führt zwangsweise zu einer höheren Anzahl benötigter Zufallsbits.

Schliesslich untersuchen wir die lokale Überprüfbarkeit von Netzwerkeigenschaften wie s-t-Erreichbarkeit, Kreisfreiheit, oder ob das Netzwerk

einen Kreis enthält. Dabei weist ein *Prover* (Beweiser) den Knoten *Label* zu, sodass ein *Verifier* (Prüfer) in konstanter Zeit prüfen kann, ob die Eigenschaft erfüllt ist, oder nicht. Wir erhalten hierbei asymptotisch scharfe Schranken für die Grösse der Labels für die letzten beiden Problemstellungen. Für s-t-Erreichbarkeit erhalten wir eine neue, asymptotisch scharfe untere Schranke in einem der betrachteten Modelle, und für ein anderes Modell entwickeln wir eine Emulationstechnik, die es uns erlaubt, eine zuvor bekannte obere Schranke ohne Einbussen in der asymptotischen Labelgrösse in unser Modell zu übertragen.

Acknowledgements

I would like to thank Prof. Roger Wattenhofer for the opportunity to write my thesis in his group. In particular I enjoyed that I had the liberty to explore different topics alongside my anonymous computing endeavors. The positive effect Roger had on my presentation skills deserves a special mention.

Also, I want to thank my two co-referees Prof. Jukka Suomela and Prof. Yuval Emek for the effort they put into reviewing my thesis. Special thanks go to Yuval for babysitting me during Roger's academic leave, and for the many fruitful discussions we had. Without him, this thesis would be even less understandable.

During my time at DISCO I have met many wonderful people, to whom I want to express my gratitude in no particular order. I thank Philipp Sommer for the warm welcome to Zurich, Johannes Schneider for taking me to places in Austria, Raphael Eidenbenz for rolling with me, Jasmin Smula for collecting clues, making deserts, and sharing laughs, Silvio Frischknecht for announcing his leave, Stephan Holzer for advocating healthy diets, Samuel Welten for talking Swiss-German to me and taking me stargazing in Gstaad, Tobias Langner for his fine wine taste, and for teaching me how to do a left turn, Jara Uitto for keeping me company in unexpected places, and for Finnish memories, David Stolz for explaining the basically two options, Laura Peer for promoting anarchy, Michael König for playing—with words, and random, Barbara Keller for making me a sandwich, and for being patient in Loèche-les-Bains, Klaus-Tycho Förster for explaining the world to me, and for his bread roll deliveries, Christian Decker for sharing his tomato-tuna sauce recipe in times of need, Sebastian Brandt for keeping his rusk box filled, Philipp Brandes for having nuts, Pascal Bissig for his baby stories, Yuezhou Lv for never being afraid to ask, Georg Bachmeier for his compatible sense of humor, Harald and Doris Schiöberg for awesome barbecues, Benny Gächter for fulfilling my hardware needs, Tanja Lantz for making my move to Switzerland a breeze, Friederike Bruetsch for teaching me how to print, and Beat Futterknecht for solving all problems at a moment's notice.

I would like to thank my friends Daniel, David, Hannes, Henning, Lucas, Manuel, and Marvin for making me feel at home whenever and

wherever we meet. From Karlsruhe, Erhard, Holger, Paul, Thomas, and Ulrich have a special place in my heart. I thank Hans-Jürgen for long and thoughtful discussions, and Christa for having a sane sleeping schedule, especially in Italy. I am indebted to Mariana for enduring me, also in stressful times. My parents Ute and Ralf have always supported me in my endeavors, and I am grateful for that.

Contents

1

Introduction

The umbrella term *distributed computing* encompasses the study of computing in networks. In this setting, the computation is performed by a network of processors, as opposed to a centralized Turing Machine. One way to study these kinds of computations is by modeling them in the form of *message passing algorithms*, where the processors are represented as nodes in a graph and the task is to produce an output at every node. Notions studied extensively for the centralized setting, e.g., computability or the effect of randomization, need to be re-evaluated in this model. Moreover, the decentralized setting raises new questions. For instance, the necessity to communicate motivates the question how many and how large messages need to be exchanged, or how many random bits nodes need to generate when executing a randomized algorithm.

Computability in networks (a.k.a. *distributed computability*) is Turing machine-equivalent if the nodes are equipped with unique IDs. This fact

remains intact even when the attention is restricted to *deterministic* distributed algorithms. However, as Angluin noticed in her seminal work [7], distributed computability becomes fascinating in *anonymous* networks, where nodes do not have unique IDs. On the one hand, the distributed problems that can be solved deterministically in anonymous networks are of a rather limited nature [84]. On the other hand, the question of distributed computability is more intricate when the nodes in an anonymous network gain access to random bits. For example, the extensively studied *maximal independent set (MIS)* problem [5, 78] is solvable in an anonymous network only if random bits are available. One contribution of this thesis is to study the role and cost randomization has for anonymous distributed computing.

Apart from its theoretical interest, the study of anonymous networks is motivated by various real-world scenarios. For example, the nodes may be indistinguishable due to their fabrication in a large-scale industrial process [9], in which equipping every node with a unique identifier (serial number) is not economically feasible. In other cases nodes may not wish to reveal their unique identity out of privacy and security concerns [62].

1.1 Overview

In Chapter 2 we study the distributed computability of anonymous message passing algorithms (referred to hereafter as *anonymous algorithms*). More precisely, we compare computability of algorithms that may change their output possibly multiple times during execution (*rewrite* algorithms) to conventional algorithms in which nodes decide on their output once and for all (*write-once* algorithms). As it turns out the effect output revocability has on the distributed computability of anonymous networks is remarkable: The two respective classes of solvable problems induced by rewrite and write-once algorithms, correspondingly, and the class of centrally solvable network problems form a strict linear hierarchy. For 21 classic and/or characteristic problems in distributed computing, we determine the exact class to which they belong.

Moreover, the hierarchy we find exhibits hard and complete problems. We introduce the concept of hardness through the notion of accessing an oracle in a distributed setting. Each of our 21 problems is then classi-

fied according to its *hardness* or *completeness* for the three classes (Section 2.5.2), thus obtaining a deeper understanding of the intrinsic properties of these problems. Notably, the three classes turn out to capture exactly the three *pillars* of distributed computing, namely, local symmetry breaking, coordination, and leader election, respectively.

In Chapter 3 we investigate more closely the role that (Las-Vegas type) randomness plays in the computational power of anonymous algorithms, regardless of round and message complexity considerations. Based on the observations in Chapter 2, we take care to rule out distributed problems in which unique IDs are (perhaps implicitly) encoded in the input, as those mock cases obviously do not faithfully represent the properties of distributed computability in anonymous networks. To that end, we restrict our focus to the class GRAN (standing for Genuinely solvable by Randomized algorithms in Anonymous Networks, see Chapter 3). What exactly characterizes the computational power of a randomized anonymous algorithm as opposed to a deterministic one? Surprisingly, randomization is only required to establish a 2-*hop coloring* of the network: Once a 2-hop coloring is known, every problem in GRAN can be solved by a deterministic anonymous algorithm.

But what is the amount of random bits, i.e., the *random bit complexity*, required to solve any such task? We strive to answer this question in Chapter 4, where alongside the random bit complexity, as a second efficiency measure, we consider the *runtime* required to solve such tasks. As it turns out, there is an *efficiency trade-off* between the runtime and the random bit complexity required to solve any task. We establish asymptotically tight lower and upper bounds on the achievable trade-off.

Lastly, in Chapter 5 we investigate the complexity of *nondeterministic* distributed algorithms. Nondeterministic distributed algorithms can be expressed as a deterministic algorithm with access to a *proof labeling* [60], where the proof labeling corresponds to an oracle in the sequential setting. The complexity of such nondeterministic algorithms is measured in terms of the maximum *proof label size* used by the deterministic representation.

In particular, we consider the proof label size in directed networks (whereas the network is thought to be undirected in the previous chapters). There are two ways to view communication in directed graphs: Nodes can communicate only in the direction of the edge (*directed one-*

way communication), or the edge direction imposes no restrictions for communication but only for the network property itself (*directed two-way* communication). We investigate both cases, as well as the undirected case, where nodes communicate with all their neighbors.

1.2 Preliminaries

Labeled Graphs. We denote the node- and edge-set of a graph G by V and E, respectively. For a node $v \in V$, we denote the set containing all neighbors of v by $\Gamma(v)$. We only consider finite connected simple[1] graphs G, with the exception of Chapter 5 where directed edges are allowed. A *labeling function* for V is a function $\ell : V \to L$ that assigns a *label* to every node in V. For the sake of simplicity, unless stated otherwise, we assume hereafter that all labels are finite bitstrings. Tuples $G = (V, E, \ell)$, where (V, E) is a graph and ℓ is a labeling function for V, are called *labeled graphs*. We often label vertices by more than one labeling function $\ell_1, \ldots, \ell_k$; in that case, we treat $G = (V, E, \ell_1, \ldots, \ell_k)$ as being labeled by a single labeling function ℓ that assigns $\ell(v) = \langle \ell_1(v), \ldots, \ell_k(v) \rangle$ to each node $v \in V$. When $G = (V, E)$ is an unlabeled graph and ℓ is a labeling function for V, we also write (G, ℓ) as a short-hand for the labeled graph (V, E, ℓ).

Distributed Problems. A distributed problem Π is specified by a set of *input instances* and for every input instance I of Π, a set $\Pi(I)$ of *valid outputs* for I. The input instances of Π are labeled graphs $I = (V, E, i)$ and the labeling function i, called the *input labeling* of I, assigns an *input label* to every node in V. For the input instance $I = (V, E, i)$, the set of valid outputs $\Pi(I)$ for I consists of labeling functions o for V called *valid output labelings* for I. At the risk of abusing the notation, we use Π to denote the set of input instances as well as the problem itself. For the sake of simplicity we assume that in every input instance $I = (V, E, i)$, the input label $i(v)$ of every node v includes v's degree. A typical example for a distributed problem is *graph coloring*, where the input is an arbitrary

[1] A graph is simple if it is undirected and does not contain any loops or parallel edges.

graph and the output must obey the rule $o(u) \neq o(v)$ if (u, v) is an edge in the input instance.

2-Hop Colorings. For a graph $G = (V, E)$, the labeling ℓ is said to be a *k-hop coloring* if $\ell(u) \neq \ell(v)$ for every $u, v \in V$, $u \neq v$, that are at most k hops away, i.e., G admits a path between u and v that consists of at most k edges. A labeling function that plays a central role in this thesis is the 2-hop coloring, i.e., a coloring that assigns a different label to each node in $\{u\} \cup \Gamma(u)$ for every $u \in V$. We say that a labeled graph $G = (V, E, \ell)$ is 2-*hop colored* if ℓ is a 2-hop coloring. Note that in the context of this thesis, we do not pay attention to the number of distinct colors used by the nodes under ℓ.

Randomized Anonymous Algorithms. Our definition of how distributed algorithms work follows the convention of [79] for synchronized network systems (message passing) with simultaneous starting times. In a labeled graph $I = (V, E, i)$, all nodes v execute the same message passing algorithm $\mathcal{A}$ with input $i(v)$. The input to a node v is fully specified by $i(v)$ — in particular, nodes are not equipped with a *(unique) identifier* nor do they possess an apriori knowledge of any global network parameter (unless specified as part of $i(\cdot)$). The execution of $\mathcal{A}$ on I is performed in synchronous rounds. In every round, each node v sends/receives messages of unbounded, yet finite, size to and from each individual neighbor, where v distinguishes between the *ports* corresponding to its incident edges. We consider randomized algorithms, where in every round, node v has access to one random bit. (Note that this is equivalent to accessing finitely many random bits per round as multiple rounds can be grouped together.)

Algorithm $\mathcal{A}$ is said to *solve* the distributed problem Π if the following two requirements hold for every $I \in \Pi$: (1) if $\mathcal{A}$ is executed on I, then $\mathcal{A}$ produces an irrevocable local output $\mathcal{A}(v)$ for every node v within finite time with probability 1; and (2) each output labeling o obtained with positive probability by setting $o(v) = \mathcal{A}(v)$ for every $v \in V$ satisfies $o \in \Pi(I)$ (i.e., we only consider Las-Vegas algorithms). This notion of solving (with irrevocable local outputs) is commonly used in the literature, and we will use it throughout this thesis. In Chapter 2 we will also study the notion of solving problems with *revocable* outputs.

In essence, an algorithm is a computable function[2] taking as input at node v the node's current state (initially v's input $i(v)$), the messages received by v (if any), and a bit chosen uniformly at random (independent of other nodes' random bits); the function returns a new state for v, the messages v sends in the next round, and possibly an output value for v. Algorithm $\mathcal{A}$ is called *deterministic* if it does not access any random bits. Note that in Chapters 2 and 3 the round and message complexities of $\mathcal{A}$ are not taken into consideration (as long as they are finite).

[2]With the exception of Section 3.3, where we consider a different kind of "algorithm".

2

The Impact of Output Revocability on Computability

What can be computed in an anonymous network, where nodes are not equipped with unique identifiers? The computability of deterministic algorithms merely depends on the topology of the network, and it is well known that problems like maximal independent set can be solved in an anonymous network only if the nodes are allowed to toss coins. In this chapter we therefore study the distributed computability of *randomized* algorithms running in anonymous networks. Notice that in this computability context, we do not impose any limitations on the complexity resources (time, message/memory size, ...), however, like in classic sequential computability theory, we do require a correct result after a finite amount of time.

It turns out that the answer to the computability question depends

on the commitment of the nodes to their first computed output value. In the following, we define two classes of problems solvable in anonymous networks, where in the first class nodes are allowed to revoke their outputs and in the second class they are not. These two classes are then related to the class of all centrally solvable network problems, observing that the three classes form a strict linear hierarchy, and for several classic and/or characteristic problems in distributed computing, we determine the exact class to which they belong.

Does this hierarchy exhibit complete problems? We answer this question in the affirmative by introducing the concept of a distributed oracle, thus establishing a more fine grained classification for distributed computability which we apply to the classic/characteristic problems. Among our findings is the observation that the three classes are characterized by the three pillars of distributed computing, namely, local symmetry breaking, coordination, and leader election.

2.1 Output Revocability

For this chapter, we consider every node v as being equipped with one *input register* holding some problem-dependent input value and one *output register*. The output register initially contains a special symbol ε indicating v *is not ready* to return an output. Any value $x \neq \varepsilon$ contained in v's output register is interpreted as v being *ready* to return its output and we say that v *has output* x. A global *configuration* in which all nodes are ready is called a *ready configuration*. When algorithm $\mathcal{A}$ is in a ready configuration, we define $\mathcal{A}$*'s output* $o_{\mathcal{A}} : V(G) \to O$ by setting $o_{\mathcal{A}}(v)$ to be the content of node v's output register. We consider two different notions of *output revocability*.

Definition (Output Revocability)**.** An algorithm is referred to as a *write-once* algorithm if every node is restricted to write to its output register at most once. If this restriction is lifted, then we call it a *rewrite* algorithm.

In other words, in a rewrite algorithm a node may revoke its output, e.g., by writing ε to its output register. While every execution of a write-once algorithm reaches at most one ready configuration, during the execution of a rewrite algorithm many ready configurations can occur.

Note that the converse does not hold: an algorithm that is guaranteed to reach at most one ready configuration is not necessarily a write-once algorithm. In the existing literature, and outside of this chapter, algorithms are typically considered to be write-once algorithms. In light of the output revocability notion, for this chapter, we need to revisit the definition of correctness of an algorithm.

Definition (Correctness)**.** Fix some problem Π and an algorithm $\mathcal{A}$. A ready configuration of $\mathcal{A}$ when invoked on an input instance $(G, i) \in \Pi$ is said to be *valid* if the output $o_{\mathcal{A}}$ of $\mathcal{A}$ in this configuration is a valid output for (G, i). Algorithm $\mathcal{A}$ is said to *solve* Π if it satisfies the following two conditions for every input instance $(G, i) \in \Pi$:

1. A ready configuration is reached within finite time with probability 1.
2. Every ready configuration that can occur with a positive probability is valid.

The aforementioned definition of correctness requires that all occurring ready configurations will be correct (i.e., correspond to a valid output). In Section 2.3 we show that our definition of correctness is robust to certain changes. Notice that in the scope of this chapter, we do not require that an algorithm *terminates* in order to be correct. However, the algorithms designed throughout the chapter do terminate, and the general transformation techniques we present (i.e., compilers/simulations) can be designed to ensure termination if the algorithms to which the transformation is applied terminate.

The choice of output revocability has a significant impact on the problems that an algorithm can solve. In the following the terms WO-algorithms and RW-algorithms will thus be used to denominate write-once and rewrite algorithms running in an anonymous network, respectively; RW and WO refer to the *classes* of distributed problems solvable by these two types of algorithms. Lastly, we denote by CF the class of distributed problems that are solvable in a centralized setting (by a Turing machine), bearing in mind that this class essentially includes every *computable function* on graphs. The distinction of these classes is justified by the following observation.

Observation 2.1. *The classes of distributed problems satisfy* WO $\subset$ RW $\subset$ CF *(in the strict sense).*

Proof. RW $\subset$ CF: A Turing machine can simulate an algorithm running in an anonymous network. On the other hand, the techniques from [7] can be used to show that leader election is not in RW, but it is clearly possible in the centralized setting.

WO $\subset$ RW: It is clear from the definition that every WO-algorithm is also a RW-algorithm. In Section 2.5.1, we will show that for example Consensus is in RW, but not in WO.

□

What can be computed in anonymous networks? As it turns out the effect output revocability has on the distributed computability of anonymous networks is remarkable. A total of 21 problems, including some of the most fundamental problems in distributed computing, are classified according to the exact class to which they belong (Section 2.5).

Does the hierarchy we present exhibit complete problems? To answer this question we introduce the notion of accessing an oracle in a distributed setting and show that this notion is sound (Section 2.4). As the first stepping stone in this effort we show that the classes WO and RW are robust against two modifications to the aforementioned correctness condition (Section 2.3). Each of our 21 problems is then classified according to its *hardness* or *completeness* for the three classes (Section 2.5.2), thus obtaining a deeper understanding of the intrinsic properties of these problems. Surprisingly, the WO, RW, and CF classes turn out to capture exactly the three *pillars* of distributed computing, namely, local symmetry breaking, coordination, and leader election, respectively.

2.2 Related Work

The history of distributed computability starts with the work of Angluin [7] proving that randomization does not help to elect a leader in anonymous networks. Later, it was shown that electing a leader in an anonymous ring network is possible if the size n of the ring is known [66], in fact, a $(2-\varepsilon)$-approximation of n is enough [1], not only in the special case of a ring but in general networks [88]. It turns out that all these

results (and many similar ones) come almost for free once our graph-theoretic characterization for the class RW is established. The connection between computation in anonymous networks and products of graphs (graph coverings) which was first observed in Angluin's seminal work plays an important role in this characterization.

There is a line of work that concentrates on *deterministic* distributed algorithms for problems in CF, in particular if some parameters of the topology of the graph (for instance, its size) are known, e.g. [33, 95]. Deterministic algorithms are interesting to investigate even if the graph is restricted to a ring [37,48], and also assignments of not necessarily unique identifiers were studied in this context [80].

Another line of research studies computability in anonymous (directed) networks in connection with termination. Not unlike us it is argued that termination in distributed systems is an issue that is not directly evident, since one may be interested in systems where nodes terminate independently of others. Different forms of termination and prior knowledge are studied in this line of work, where the strongest anonymous model considered is equivalent to deterministic write-once algorithms with knowledge of an upper bound to the network size [34]. When no prior knowledge is assumed the class of solvable problems can be fully characterized using *local views*[1] (quasi-coverings) and recursive functions [38]. Extending their approach, in the context of the current chapter an individual node executing a RW-algorithm can never be entirely sure about termination. We show that the class RW lies *between* the two classes WO (local termination) and CF (global termination).

Output revocability should not be confused with the concept of *eventual correctness*, where the network eventually converges to a correct output. For example, *self-stabilizing algorithms* [41] allow the system to return an incorrect output for a finite amount of time, thus allowing a fault-tolerant algorithm to recover from errors. With randomization, self-stabilizing leader election is possible on general graphs [42], hence with randomization every CF-problem is eventually solvable in an anonymous network. In our terminology eventual correctness could be viewed as requiring that some ready configuration, not necessarily the first one, is

[1] See Section 2.5.1 for a definition.

stable[2] and valid. We require though that an output is returned after finite time and that every output returned by the network is correct, but we do allow the network to revoke partial outputs. The problems solvable by self-stabilizing algorithms in directed graphs can be characterized by *fibrations* [36], the directed analog to factors[1] of graphs.

The self-stabilization concept is also used in the scope of population protocols, introduced by Angluin et al. [9]. Population protocols are an example for asynchronous distributed automata with restricted computational power. In this model, nodes cannot do arbitrary computations, as they are modeled by finite state machines, see [16] for an overview. Regarding computability in a clique network, [9–11] conclude the predicates solvable to be exactly those expressible in first-order Presburger arithmetic. On graphs with bounded degree a Turing machine with linearly bounded space can be simulated [8]. It was also studied how the correctness condition for population protocols affects solvability of the Consensus problem [12].

Apart from these results, not much is known about distributed computability, as a large fraction of research deals with complexity rather than computability. However, there are surprising connections between complexity and computability, which go beyond us borrowing the terms hardness and completeness. Regarding network algorithms, in the last thirty years, a lot of research went into the question how fast a particular problem can be computed by the network. Literally hundreds of new upper and lower bounds have been found. The fastest algorithms deliver a result within constant time, independent of the size of the network, see [94] for a recent survey. It is intriguing that our research which is about computability has most connections to this "fastest" class of distributed algorithms.

Naor and Stockmeyer [84] introduced the notion of locally checkable labelings (essentially an apply-once oracle, i.e., an oracle that can be invoked only once at the beginning of an algorithm; see Section 2.4) in identified networks and ask the question how a constant-time deterministic algorithm can decide whether the labeling represents a correct solution to a given problem. Follow-up work looked at the bit complexity required

[2] A configuration is said to be *stable* if the nodes no longer revoke their outputs, see Section 2.3.

to solve decision problems [68] and a problem hierarchy depending on the size of checkable labelings was suggested [60], also for anonymous networks. Our work also yields a characterization of decision problems in RW. How apply-once oracles can be used to make broadcast and wake-up schemes more efficient was studied in [52]. However, we do not restrict the run-time to be constant and allow randomization for symmetry breaking. Pruning algorithms [70] that build a solution gradually in a write-once fashion were inspired by the same line of research, in an effort to remove the necessity of global knowledge about the graph. While our algorithms are required to give a correct output in every execution, [53,54] study the notion of (p, q)-decidable decision problems (an anonymous randomized algorithm is allowed to return a wrong output with constant probability) and find a strict hierarchy among the classes of solvable problems depending on the success probabilities. If a randomized algorithm is allowed to fail (Monte-Carlo algorithm), then a leader can be elected [82] with high probability (w.h.p, i.e., with probability $1-n^{-c}$ for any c). Hence any CF-problem can be solved in an anonymous network with high probability, whereas we require a correct output with probability 1.

Non-deterministic algorithms running in an anonymous setting can fully determine the structure of the radius t-ball around itself in [51], and thus solve exactly the decision problems that are closed under so-called *t-homomorphisms*, that is, homomorphisms that preserve the structure t hops around every node, regardless of access to unique identifiers. In our model only the local view can be retrieved. It may thus be surprising that RW-algorithms can solve exactly the problems that such non-deterministic constant-time algorithms can solve in a single round.

Lastly, it is worth mentioning that in the context of *shared memory* systems a notion of distributed oracles in an asynchronous environment is studied. Usually such an oracle is applied once to implement a protocol (algorithm), and the tasks (e.g., consensus) also form hierarchies by their ability to implement each other [55, 63, 77]. Unlike in our model, computability in shared memory systems is hindered by asynchronous execution rather than the network structure and has surprising connections to topology [64]. Nonetheless variants of the consensus tasks turn out to be complete for the class RW.

2.3 Notions of Correctness

Our definition of a correct algorithm requires every ready configuration that occurs throughout an execution to be valid. For WO-algorithms this requirement is superfluous since its execution will reach at most one ready configuration. However, RW-algorithms may invalidate or change a ready configuration after it occurred. One may therefore wonder if strengthening the definition by allowing only one durable ready configuration makes the class of solvable problems strictly smaller. On the other hand one may be tempted to weaken this definition, in hope to capture a larger class of problems by requiring only the first occurring ready configuration to be correct. Perhaps surprisingly we show that these two variants have no effect and are equivalent to the current definition of correctness. This equivalence will play a key role when we reason about RW-algorithms in the next section which covers distributed oracles.

Definition (Sustainable Correctness)**.** A ready configuration is said to be *stable*, if the nodes no longer revoke their outputs. Algorithm $\mathcal{A}$ is said to *sustainably solve* a problem Π if it satisfies the following two conditions for every input instance $(G, i) \in \Pi$:

1. A ready configuration is reached within finite time with probability 1.
2. The first ready configuration that occurs is valid and stable.

Definition (Loose Correctness)**.** Algorithm $\mathcal{A}$ is said to *loosely solve* a problem Π if it satisfies the following two conditions for every input instance $(G, i) \in \Pi$:

1. A ready configuration is reached within finite time with probability 1.
2. The first ready configuration that occurs is valid.

The class *Sustainable-RW* (respectively, *Loose-RW*) consists of every distributed problem that can be sustainably solved (resp., loosely solved) by a RW-algorithm. Since sustainable correctness (resp., loose correctness) is a restriction (resp., a relaxation) of correctness as defined in Section 2.1, we conclude that Sustainable-RW $\subseteq$ RW $\subseteq$ Loose-RW. Note that the corresponding classes *Sustainable-WO* and *Loose-WO* for WO-algorithms are equal to the class WO due to the write-once restriction of these algorithms. The following theorem states that also for RW-algorithms the three classes are, in fact, equal.

Theorem 2.1. *The classes of problems solvable by* RW*-algorithms under the three different correctness notions satisfy Sustainable-*RW = RW = *Loose-*RW*.*

The proof of Theorem 2.1 is based on a simple concept referred to as *safe broadcast* in which information is broadcast throughout the whole network and no ready configuration is reached before all nodes have received the information. When a node v receives a previously unseen message M that should be safely broadcast, it writes ε to its output register for at least one round and forwards M to all its neighbors. This ensures that M propagates through the network together with a front of non-ready nodes, so that no ready configuration can be reached during the dissemination of M.

Based on the safe broadcast concept, we develop a generic technique called *inhibiting messages* which will also be useful when designing algorithms in Section 2.5. For every node v, this programming technique employs a register ρ, usually chosen to be v's output register, and a list L containing pairs (i, x) where i is an integer, typically a round or phase number, and x is an arbitrary value. Two methods are provided for every node v, where the invocation of these methods is determined by user defined conditions: A node v can (1) append a new pair (i, x) to L; and (2) broadcast an inhibiting message M_i for i. The operation is as follows. If v sends or receives an inhibiting message M_i, then for all x the pairs (i, x) are removed from L. Whenever L is empty, node v sets $\rho \leftarrow \varepsilon$. Assuming that L is non-empty, denote by $(i_{\min}, x_{\min})$ a pair in Q that satisfies $i_{\min} \leq i$ for all pairs (i, x) in Q. In that case, the default value stored in ρ is the value $x_{\min}$. The one exception to this rule occurs when v receives an inhibiting message $M_{i_{\min}}$, in which case v sets $\rho \leftarrow \varepsilon$ in the current round, which means that ε is written to ρ between any two consecutive non-ε values. Notice that the front of non-ready nodes propagates through the network with the inhibiting message M_i only as long as M_i *invalidates* the output currently contained in the output registers.

We employ inhibiting messages to show that the class RW is robust against the stated modifications to the definition of a correct algorithm. The proof of Theorem 2.1 relies on a *sustainability compiler* that takes a RW-algorithm $\mathcal{A}$ that loosely solves problem Π and transforms it into a

RW-algorithm $\hat{\mathcal{A}}$ that sustainably solves this problem. Specifically, under algorithm $\hat{\mathcal{A}}$, every node v simulates $\mathcal{A}$; to avoid confusion, let $\hat{\rho}$ be v's output register under $\hat{\mathcal{A}}$ and let ρ be v's register simulating the output register of $\mathcal{A}$. The compiler is based on sending inhibiting messages, where the register upon which the inhibiting message technique operates is $\hat{\rho}$ and the integers i of the technique are identified with the round numbers. In every round r, if v is not ready in round r under $\mathcal{A}$, then node v broadcasts an inhibiting message M_r, that is, v broadcasts an inhibiting message for r if $\rho = \varepsilon$. If on the other hand v's register ρ contains the value $x \neq \varepsilon$ in round r, then v appends the pair (r, x). Theorem 2.1 is established by proving the following lemma.

Lemma 2.2. *Let $\mathcal{A}$ be a* RW*-algorithm loosely solving a problem Π and let $\hat{\mathcal{A}}$ be the* RW*-algorithm obtained by applying the sustainability compiler to $\mathcal{A}$. Then $\hat{\mathcal{A}}$ sustainably solves Π.*

Proof. Consider some input instance $(G, i) \in \Pi$ and denote by η the execution of $\mathcal{A}$ on (G, i) that $\hat{\mathcal{A}}$ simulates. Algorithm $\hat{\mathcal{A}}$ employs inhibiting messages. For the sake of the analysis let $i_v(r)$ denote the value $i_{\min}$ of node v in round r, or NIL if v's queue is empty. In particular, if $i_v(r) \neq$ NIL, then the value stored in v's output register $\hat{\rho}$ is the output of v in round r of η. By definition, η must reach a ready configuration and the first ready configuration reached by η is valid; let r_0 denote the round in which this valid ready configuration is reached and let o_0 be the valid output returned by η in that round. Notice that under algorithm $\hat{\mathcal{A}}$, no node broadcasts an inhibiting message for round r_0, whereas at least one node broadcasts an inhibiting message M_r for every round $r < r_0$. This implies that under $\hat{\mathcal{A}}$, eventually $i_v(r) = r_0$ for every node v; let $r_1 \geq r_0$ be the first round in which this happens. Starting from round r_1, algorithm $\hat{\mathcal{A}}$ outputs o_0 and the design of the inhibiting message technique guarantees that $\hat{\mathcal{A}}$ will not revoke this output. Therefore, we only have to ensure that under $\hat{\mathcal{A}}$, in all rounds $r < r_1$ at least one node is not ready.

To that end, assume for the sake of contradiction that there exists a round $r < r_1$ in which all nodes are ready under $\hat{\mathcal{A}}$. In that case $i_v(r) \neq$ NIL for every node v. If $i_v(r) = i_u(r)$ for all $u, v \in V(G)$ then $\mathcal{A}$ was in a ready configuration in round r and thus $r = r_0 = r_1$. Therefore in round r under $\hat{\mathcal{A}}$, there must be nodes having outputs from two different

rounds of η. Moreover, since G is connected there must exist two such nodes u and v, $\{u, v\} \in E(G)$. Since the sustainability compiler employs the inhibiting message technique, we conclude that $i_u(r) \neq i_v(r)$ and without loss of generality assume that $i_u(r) < i_v(r)$. But this means, that in some round $r' < r$ node v sent an inhibiting message for round $i_u(r)$ and this message reaches u in round $r' + 1 \leq r$, in contradiction to the assumption that round $i_u(r)$ is non-inhibited for u in round r. It follows that $\hat{\mathcal{A}}$ does not reach a ready configuration prior to round r_1 which completes the proof. □

2.4 Distributed Oracles

In this section, we introduce the concepts of hardness and completeness, which are central to this work and allow us to gain a deeper understanding how the computability classes relate to each other. To that end, we introduce the notion of an oracle working in a distributed setting.

Definition (Algorithm with access to a Π-oracle)**.** Consider some problem Π. A C-algorithm, C ∈ {WO, RW}, with *access to a* Π*-oracle* is a distributed C-algorithm in which every node v is equipped with a designated *oracle input register* and a designated *oracle output register*. Given some $r \geq 1$, let $\tilde{i}(v)$ be the content of v's oracle input register in round r and let $\tilde{o}(v)$ be the content of v's oracle output register in round $r + 1$. If $(G, \tilde{i})$ is an input instance of Π, then it is guaranteed that $\tilde{o}$ is a valid output for $(G, \tilde{i})$. No assumptions are made on the operation of the algorithm if $(G, \tilde{i}) \notin \Pi$.

While applying the oracle in every round of the algorithm may seem powerful, allowing the distributed algorithm to arbitrarily choose the rounds in which the oracle is applied may require some sort of global coordination, which is not necessarily possible. In comparison, a weaker definition of "accessing an oracle" would be to allow application of the oracle only once in round 1. This distinction does not make a difference for problems Π without inputs ($|I(\Pi)| = 1$), e.g., for graph theoretic problems like coloring, maximal independent set, or determining the diameter, because the oracle is always applied on the same input instance. It does

however affect problems that do receive inputs ($|I(\Pi)| \geq 2$), e.g., CONSENSUS or logical AND and OR. It will be convenient to refer to this weaker manner of accessing an oracle as accessing an *apply-once* oracle.

As stated above, based on the oracle concept, we will soon introduce the notion of hard and complete problems for the hierarchy of problem classes. This notion would be ill-defined if accessing an oracle to a problem $\Pi_C \in C$ could enhance the computational power of a C-algorithm. We ensure that the notion of an algorithm with access to an oracle is sound in the following theorem. Note that the statement of the theorem does not mention the case C = CF, since the soundness of oracles for centralized models is well understood and in any case, beyond the scope of this thesis.

Theorem 2.2 (Soundness)**.** *If a problem* Π *is solvable by a* C*-algorithm,* $C \in \{RW, WO\}$*, accessing an oracle to a problem* $\Pi_C \in C$*, then* Π *can also be solved by a* C*-algorithm that does not access any oracle.*

The key to proving this theorem is to show that in a C-algorithm $\mathcal{A}^a$ that solves a problem Π with *access* to a Π_C-oracle, $\Pi_C \in C$, one can *replace* the oracle access by simulating a C-algorithm $\mathcal{A}^r$ that solves Π_C without any oracle access. We will first prove that accessing apply-once oracles does not enhance the computational power of RW- and WO-algorithms, since the two algorithms $\mathcal{A}^a$ and $\mathcal{A}^r$ can be executed consecutively one after the other, or in other words, that algorithm $\mathcal{A}^a$ accessing an apply-once oracle can be simulated without accessing an oracle by executing $\mathcal{A}^r$ first. This turns out to be a non-trivial task especially for RW-algorithms since a node v simulating $\mathcal{A}^r$ cannot know for sure that the output returned by $\mathcal{A}^r$ will not be revoked later on, i.e., whether it can be safely used for the execution of $\mathcal{A}^a$. It therefore does not know when such a result is valid so that a simulation of $\mathcal{A}^a$ can be invoked based on this result. The technique we present to resolve this issue for RW-algorithms is based on Theorem 2.1. Actually, we will need an extension of Lemma 2.2 (the key to the proof of Theorem 2.1) to RW-algorithms accessing a Π'-oracle for some problem Π'. To that end, we observe that the construction of the sustainability compiler and the arguments used in the proof of Lemma 2.2 can be repeated with no changes to yield the following.

Lemma 2.3. *Fix some problem* Π'*. Let* $\mathcal{A}$ *be a* RW*-algorithm with access to a* Π'*-oracle loosely solving a problem* Π *and let* $\hat{\mathcal{A}}$ *be the* RW*-algorithm with access to a* Π'*-oracle obtained by applying the sustainability compiler to* $\mathcal{A}$*. Then* $\hat{\mathcal{A}}$ *sustainably solves* Π *with an access to a* Π'*-oracle.*

In other words, Lemma 2.3 states that the three notions of correctness for RW-algorithms are equivalent even when the algorithm has an access to a Π'-oracle for some (arbitrary) problem Π'. This enables us to establish the following lemma that states the soundness of apply-once oracles for RW-algorithms.

Lemma 2.4 (Consecutive RW Execution)**.** *Let* Π_{C} *be a problem in* RW *and let* $\mathcal{A}^a$ *be a* RW*-algorithm solving an arbitrary problem* Π *with access to an apply-once* Π_{C}*-oracle. Then* Π *is solvable by a* RW*-algorithm without access to any oracle.*

Proof. Let $\mathcal{A}^{\mathrm{r}}$ be a RW-algorithm solving Π_{C}. Employing Lemmas 2.2 and 2.3, we assume that $\mathcal{A}^{\mathrm{r}}$ and $\mathcal{A}^{\mathrm{a}}$ sustainably solve Π_{C} and Π, respectively. We would like to show that $\Pi \in \mathrm{RW}$ by designing a RW-algorithm $\mathcal{A}$ that solves Π without access to any oracle. This will be accomplished by letting $\mathcal{A}$ simulate the execution of $\mathcal{A}^{\mathrm{a}}$, using $\mathcal{A}^{\mathrm{r}}$ to replace $\mathcal{A}^{\mathrm{a}}$'s access to the apply-once Π_{C}-oracle. Algorithm $\mathcal{A}$ faces the issue that an output returned to a node v by $\mathcal{A}^{\mathrm{r}}$ may not be part of a ready configuration and thus it is not clear whether v should use this value as an output of the Π_{C}-oracle that $\mathcal{A}^{\mathrm{a}}$ invoked. To cope with that, algorithm $\mathcal{A}$ performs a systematic search for some round in which $\mathcal{A}^{\mathrm{r}}$ reaches a ready configuration.

Algorithm $\mathcal{A}$ simulates algorithms $\mathcal{A}^{\mathrm{r}}$ and $\mathcal{A}^{\mathrm{a}}$; to avoid confusion, let ρ, ρ^{r}, and ρ^{a} denote the output registers of v under $\mathcal{A}, \mathcal{A}^{\mathrm{r}}$, and $\mathcal{A}^{\mathrm{a}}$, respectively. Algorithm $\mathcal{A}$ works in phases, where phase p consists of $2p$ rounds as follows. In each phase p, every node v first simulates p rounds of $\mathcal{A}^{\mathrm{r}}$; the role of this simulation is to replace the access to the (apply-once) Π_{C}-oracle. While this simulation takes place, node v sets $\rho \leftarrow \varepsilon$ ensuring that a ready configuration can only be reached in the second half of phase p. Node v is referred to as *sad* if $\rho^{\mathrm{r}} = \varepsilon$ at the end of round p of phase p; otherwise, node v is referred to as *happy*. If node v is sad, then it does not participate in the next p rounds of phase p and sets its output register

$\rho \leftarrow \varepsilon$ in round $p+1$. If node v is happy, then in the next p rounds of phase p, it simulates p rounds of $\mathcal{A}^{\mathrm{a}}$ using the value stored in ρ^{r} as the output of the Π_{C}-oracle (accessed by $\mathcal{A}^{\mathrm{a}}$) and sets $\rho = \rho^{\mathrm{a}}$ in every round of the simulation. For convenience, let σ_p^{a} denote the sequence of rounds (of $\mathcal{A}$'s execution) that are dedicated to simulating algorithm $\mathcal{A}^{\mathrm{a}}$ in phase p, i.e., $\sigma_1^{\mathrm{a}} = [2], \sigma_2^{\mathrm{a}} = [5, 6]$ and so on. It will be important for the analysis that when simulating algorithms $\mathcal{A}^{\mathrm{r}}$ and $\mathcal{A}^{\mathrm{a}}$ in phase $p+1$, node v reuses the same random bits that were used in phase p to which v only adds the random bits required for the simulation of round $p+1$ in both algorithms.

For the sake of the analysis, let η^{r} be the execution of algorithm $\mathcal{A}^{\mathrm{r}}$ that corresponds to the simulation performed by algorithm $\mathcal{A}$. Notice that η^{r} is well-defined since under $\mathcal{A}$, the simulation of $\mathcal{A}^{\mathrm{r}}$ reuses the same random bits in every phase, so that in all phases p, the first p rounds of $\mathcal{A}^{\mathrm{r}}$ correspond to the first p rounds of η^{r}. Denote by o^{r} the output obtained from the stable ready configuration reached by η^{r}. Based on that, let η^{a} be the execution of algorithm $\mathcal{A}^{\mathrm{a}}$ that corresponds to the simulation performed by algorithm $\mathcal{A}$ in which the oracle access is replaced by o^{r}, and let o^{a} denote the output obtained from the stable ready configuration reached by η^{a}. The execution η^{a} and its output o^{a} are well defined since $\mathcal{A}^{\mathrm{a}}$ sustainably solves Π and $\mathcal{A}$ reuses random bits to simulate $\mathcal{A}^{\mathrm{a}}$ as well. Denote by t^{r} and t^{a} the rounds in which the stable ready configurations of η^{r} and η^{a} are reached for the first time, respectively, and let $t = \max\{t^{\mathrm{r}}, t^{\mathrm{a}}\}$. We argue that algorithm $\mathcal{A}$ reaches the first ready configuration in phase t, namely in round $\sigma_t^{\mathrm{a}}(t^{\mathrm{r}})$ of $\mathcal{A}$'s execution, and that the output of this ready configuration is o^{a}, which together with Theorem 2.1 establishes the assertion since $\mathcal{A}$ (at least) loosely solves Π.

To see that this is indeed true recall that under algorithm $\mathcal{A}$, a node v may only set ρ to a non-ε value in the second half of a phase that is dedicated to simulating $\mathcal{A}^{\mathrm{a}}$. In phases $p < t^{\mathrm{r}}$ at least one node is sad, i.e., not ready in round p of η^{r}, and therefore not ready during the second half of phase p. On the other hand, in phases $p \geq t^{\mathrm{r}}$ all nodes are happy and the simulation of $\mathcal{A}^{\mathrm{a}}$ performed by $\mathcal{A}$ corresponds to the first p rounds of η^{a}. The correctness of $\mathcal{A}$ now follows from the sustainable correctness of $\mathcal{A}^{\mathrm{a}}$. □

The crux in the proof of Lemma 2.4 was to show how two RW-algorithms can be executed consecutively in a correct manner. At first, it seems that the same technique is inapplicable to a WO-algorithm (accessing an apply-once oracle), since under algorithm $\mathcal{A}$ described in the proof of Lemma 2.4, a node will revoke any output it returned in the last round of a phase, i.e., algorithm $\mathcal{A}$ is not a WO-algorithm due to our construction. However the technique can be slightly modified so that it is applicable to WO-algorithms as well.

To address the aforementioned issue, we make three adjustments to the construction of algorithm $\mathcal{A}$ when it is applied to a WO-algorithm $\mathcal{A}^{\mathrm{a}}$ with access to a Π_{C}-oracle, $\Pi_{\mathrm{C}} \in \mathrm{WO}$. To describe the adjustments we use the same terminology as in the proof of Lemma 2.4: (1) Node v is not allowed to change the value stored in its output register ρ after the first value $x \neq \varepsilon$ was written to it. (2) If v is sad at the end of round p of phase p, then v broadcasts a *sadness message* for phase p. (3) If a happy node v in phase p receives a sadness message for that phase (in one of the rounds $\sigma_p^{\mathrm{a}}(1), \ldots, \sigma_p^{\mathrm{a}}(p)$), then v stops to participate in the simulation of $\mathcal{A}^{\mathrm{a}}$, and in particular does not write to its output register ρ in the remainder of phase p.

The first adjustment immediately ensures that the resulting algorithm $\mathcal{A}$ is indeed a WO-algorithm. We argue that $\mathcal{A}$ reaches a ready configuration in phase t, and that the output of $\mathcal{A}$ is o^{a} (and therefore correct). In phases $p < t$ there is at least one node v that is sad or did not produce an output under algorithm $\mathcal{A}^{\mathrm{a}}$, and therefore v does not become ready in the second half of phase p. In phases $p \geq t$ on the other hand, all nodes are happy and the simulation of $\mathcal{A}^{\mathrm{a}}$ corresponds to η^{a}.

Since $\mathcal{A}$ is a WO-algorithm we need to ensure that the output $o_{\mathcal{A}}$ of $\mathcal{A}$ satisfies $o_{\mathcal{A}}(v) = o^{\mathrm{a}}(v)$ for all nodes v since a node that wrote to its output register in some phase $p < t$ cannot revoke its output in later phases. Consider some node v and denote by p the phase in which v writes to its output register. This occurs in round $s = \sigma_p^{\mathrm{a}}(s^{\mathrm{a}})$ dedicated to simulating round s^{a} of $\mathcal{A}^{\mathrm{a}}$. All nodes u in the inclusive s^{a}-hop neighborhood $\Gamma^+_{r^{\mathrm{a}}}(v)$ must be happy in phase p (otherwise v would have received a sadness message). Moreover, the simulation that a node u at distance $d < s^{\mathrm{a}}$ from v performs of $\mathcal{A}^{\mathrm{a}}$ agrees with η^{a} for the first $s^{\mathrm{a}} - d$ rounds. Therefore for node v, the first s^{a} rounds of $\mathcal{A}$'s simulation of $\mathcal{A}^{\mathrm{a}}$ correspond to the first

s^{a} rounds of η^{a}. Since $\mathcal{A}^{\mathrm{a}}$ and $\mathcal{A}^{\mathrm{r}}$ are both correct WO-algorithms, this implies that $o_{\mathcal{A}}(v) = o^{\mathrm{a}}(v)$, which concludes our argument. Lemma 2.5 follows.

Lemma 2.5 (Consecutive WO Execution)**.** *Let* Π_{C} *be a problem in* WO *and let* $\mathcal{A}^{a}$ *be a* WO*-algorithm solving an arbitrary problem* Π *with access to an apply-once* Π_{C}*-oracle. Then* Π *is solvable by a* WO*-algorithm without access to any oracle.*

When trying to extend the proof of Lemma 2.4 in attempt to establish the RW case of Theorem 2.2, the issue we needed to solve for RW-algorithms with an access to an apply-once oracle multiplies: Between every two simulated rounds of $\mathcal{A}^{\mathrm{a}}$, one invocation of $\mathcal{A}^{\mathrm{r}}$ is required to replace the oracle access, and a simulating node cannot know for sure that an output obtained from $\mathcal{A}^{\mathrm{r}}$ is part of a ready configuration for any such simulation of $\mathcal{A}^{\mathrm{r}}$. However, the ideas used to prove Lemma 2.4 can be extended to cope with this difficulty. We will show how to interleave single rounds in the simulation of an algorithm $\mathcal{A}^{\mathrm{a}}$ accessing an oracle with executions of an algorithm $\mathcal{A}^{\mathrm{r}}$ that replaces the oracle.

Proof of Theorem 2.2. Let C be either WO or RW, let Π_{C} be a problem in the class C, and let $\mathcal{A}^{\mathrm{r}}$ be a C-algorithm solving Π_{C}. Let $\mathcal{A}^{\mathrm{a}}$ be a C-algorithm solving an arbitrary problem Π with access to a Π_{C}-oracle (applied in every round of $\mathcal{A}^{\mathrm{a}}$). If C = RW, then by Theorem 2.1 and Lemma 2.3, we assume that $\mathcal{A}^{\mathrm{r}}$ and $\mathcal{A}^{\mathrm{a}}$ sustainably solve Π_{C} and Π, respectively. We wish to simulate $\mathcal{A}^{\mathrm{a}}$ and multiple invocations of $\mathcal{A}^{\mathrm{r}}$ using a C-algorithm $\mathcal{A}$ without access to any oracle. Denote by ρ the output register of node v. The construction of algorithm $\mathcal{A}$ is similar to the construction we used in the proofs of Lemmas 2.4 and 2.5; the difference is that in phase p, algorithm $\mathcal{A}$ should now simulate p invocations of algorithm $\mathcal{A}^{\mathrm{r}}$, one before each round of the simulated execution of $\mathcal{A}^{\mathrm{a}}$, instead of just a single invocation. That is, we precede each round $1 \leq i \leq p$ of $\mathcal{A}^{\mathrm{a}}$'s simulated execution under $\mathcal{A}$ with a simulation of an invocation of $\mathcal{A}^{\mathrm{r}}$ that runs for p rounds and replaces $\mathcal{A}^{\mathrm{a}}$'s access to the Π_{C}-oracle between rounds $i-1$ and i. Specifically, phase p now consists of p^2+p rounds, where each round $r \equiv 0 \pmod{p+1}$ of phase p is dedicated to simulating round $r/(p+1)$ of $\mathcal{A}^{\mathrm{a}}$, whereas each round $r \not\equiv 0 \pmod{p+1}$ is

dedicated to simulating round $r \pmod{p+1}$ in invocation $i = \lceil r/(p+1) \rceil$ of $\mathcal{A}^{\mathrm{r}}$, occurring between rounds $i-1$ and i of $\mathcal{A}^{\mathrm{a}}$. For convenience, let σ_p^{a} denote the sequence of rounds (of $\mathcal{A}$'s execution) that are dedicated to simulating algorithm $\mathcal{A}^{\mathrm{a}}$ in phase p, i.e., $\sigma_1^{\mathrm{a}} = \langle 2 \rangle, \sigma_2^{\mathrm{a}} = \langle 5, 8 \rangle$ and so on.

During phase p, it may happen that the simulation of invocation $1 \leq i \leq p$ of $\mathcal{A}^{\mathrm{r}}$ in node v outputs ε, which means that v cannot simulate round i of $\mathcal{A}^{\mathrm{a}}$; when this happens, node v becomes sad for the current phase p. Recall that this means that v stops participating in the remainder of phase p and sets $\rho \leftarrow \varepsilon$. Moreover, if C = WO, then in addition to that, v broadcasts a sadness message. As before, node v sets $\rho \leftarrow \varepsilon$ during simulations of $\mathcal{A}^{\mathrm{r}}$, and when v is happy ρ is used to simulate the output register of $\mathcal{A}^{\mathrm{a}}$.

For the sake of the analysis we inductively define executions η_i^{r} of algorithm $\mathcal{A}^{\mathrm{r}}$ and an execution η^{a} of algorithm $\mathcal{A}^{\mathrm{a}}$. Let η_1^{r} be the execution of algorithm $\mathcal{A}^{\mathrm{r}}$ that corresponds to the simulation that $\mathcal{A}$ performs to replace $\mathcal{A}^{\mathrm{a}}$'s first oracle access, and denote by o_1^{r} the output obtained from the stable ready configuration of η_1^{r}. Both η_1^{r} and o_1^{r} are well-defined since under $\mathcal{A}$, the simulation of $\mathcal{A}^{\mathrm{r}}$ reuses the same random bits in every phase and due to the sustainable correctness of $\mathcal{A}^{\mathrm{r}}$. Let $\eta_{(1)}^{\mathrm{a}}$ be the first round of $\mathcal{A}^{\mathrm{a}}$'s execution η^{a} that algorithm $\mathcal{A}$ simulates in which the first oracle access of $\mathcal{A}^{\mathrm{a}}$ is replaced with o_1^{r}. Based on η_1^{r}, the first round $\eta_{(1)}^{\mathrm{a}}$ in η^{a} is well-defined. We define the executions η_i^{r} and the remaining rounds of η^{a} inductively: (1) Let η_i^{r} be the execution of algorithm $\mathcal{A}^{\mathrm{r}}$ that corresponds to the simulation that $\mathcal{A}$ performs to replace $\mathcal{A}^{\mathrm{a}}$'s oracle access after round $i-1$ of η^{a}, and denote by o_i^{r} the output obtained from the stable ready configuration of η_i^{r}. (2) Let $\eta_{(i)}^{\mathrm{a}}$ be the ith round in the execution of $\mathcal{A}^{\mathrm{a}}$ that corresponds to the simulation performed by algorithm $\mathcal{A}$ in which $\mathcal{A}^{\mathrm{a}}$'s oracle access is replaced by o_i^{r}. Note that (1) and (2) together are well-defined, since the induction is based on $\eta_{(1)}^{\mathrm{a}}$ and $\eta_{(1)}^{\mathrm{r}}$, and the simulations of $\mathcal{A}^{\mathrm{r}}$ and $\mathcal{A}^{\mathrm{a}}$ reuse the same random bits in every phase. Thanks to the sustainable correctness of $\mathcal{A}^{\mathrm{a}}$ we denote by o^{a} the output obtained from the stable ready configuration η^{a} reaches.

With these definitions in mind, denote by t^{a} the first round in which η^{a} is in a ready configuration. Denote by t_i^{r} the first round in which η_i^{r} is in a ready configuration and let $t^{\mathrm{r}} = \max_{i<t^{\mathrm{a}}}\{t_i^{\mathrm{r}}\}$. Lastly, denote by o^{a} the output obtained from the stable ready configuration reached by η^{a} in

round t^{a}. We argue that algorithm $\mathcal{A}$ reaches the first ready configuration in phase $t = \max\{t^{\mathrm{a}}, t^{\mathrm{r}}\}$, specifically in round $\sigma_t^{\mathrm{a}}(t^{\mathrm{a}})$, and that the output of $\mathcal{A}$ in that phase is o^{a}.

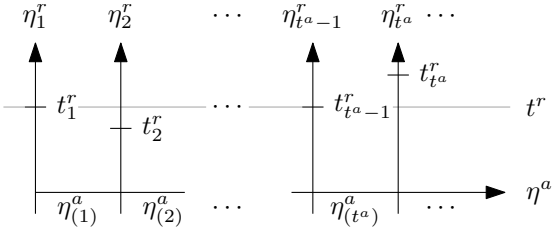

Figure 2.1: Executions η^{a} and η_i^{r} of $\mathcal{A}^{\mathrm{a}}$ and $\mathcal{A}^{\mathrm{r}}$, respectively.

Assume for the sake of contradiction that in some phase $p < t$ algorithm $\mathcal{A}$ reaches a ready configuration. Due to our construction this can only occur in a round $s = \sigma_p^{\mathrm{a}}(s^{\mathrm{a}})$ dedicated to simulating some round s^{a} of algorithm $\mathcal{A}^{\mathrm{a}}$. In that case all nodes are happy in round s, which can only occur if $t_i^{\mathrm{r}} \leq p$ for all $i < s^{\mathrm{a}}$. This implies that the first s^{a} rounds that $\mathcal{A}$ simulated of algorithm $\mathcal{A}^{\mathrm{a}}$ correspond to the first s^{a} rounds of η^{a}, i.e., η^{a} reaches a ready configuration in round $s^{\mathrm{a}} = t^{\mathrm{a}}$ in contradiction with the choice of t. In phase $p = t$ on the other hand, for all $i \leq t^{\mathrm{a}}$ execution η_i^{r} reaches a ready configuration within p rounds. Therefore the first t^{a} rounds that $\mathcal{A}$ simulates of $\mathcal{A}^{\mathrm{a}}$ correspond to the first t^{a} rounds of η^{a} and $\mathcal{A}$ reaches a ready configuration in round t^{a}. In the case C = RW, the (at least loose) correctness of $\mathcal{A}$ now follows from the correctness of $\mathcal{A}^{\mathrm{a}}$ and the proof is concluded by applying Theorem 2.1.

For the case C = WO however, we need to ensure that the output $o_{\mathcal{A}}$ of algorithm $\mathcal{A}$ satisfies $o_{\mathcal{A}}(v) = o^{\mathrm{a}}(v)$ for all nodes v, since in algorithm $\mathcal{A}$ a node v may write to its output register ρ prior to phase t. Let v be a node that irrevocably sets $\rho \leftarrow o_{\mathcal{A}}(v)$ in phase p. This can only occur in round $s = \sigma_p^{\mathrm{a}}(s^{\mathrm{a}})$ for some s^{a}. Since v did not receive a sadness message for phase p all nodes u in the inclusive s^{a}-hop neighborhood $\Gamma_{s^{\mathrm{a}}}^+(v)$ of v are happy in round s. In other words, all nodes u are ready in the first s^{a} simulations of $\mathcal{A}^{\mathrm{r}}$ performed by $\mathcal{A}$ in phase p. It follows that for node v the first s^{a} rounds of the s^{a} simulations of $\mathcal{A}^{\mathrm{a}}$ correspond to the first

s^{a} rounds of η^{a}. Since $\mathcal{A}^{\mathrm{r}}$ and $\mathcal{A}^{\mathrm{a}}$ are both WO-algorithms we conclude that indeed $o_{\mathcal{A}}(v) = o^{\mathrm{a}}(v)$. □

Now that Theorem 2.2 is established we introduce the concept of hard problems by borrowing the terminology from sequential complexity theory.

Definition (Hardness)**.** For two classes $\mathrm{B} \supseteq \mathrm{C}$, a problem Π is said to be B-*hard* with respect to C, denoted by $\Pi \in B\text{-}hard_{\mathrm{C}}$, if for every problem $\Pi_{\mathrm{B}} \in \mathrm{B}$, there exists a C-algorithm that solves Π_{B} with access to a Π-oracle. We say that Π is *complete* in B with respect to C, denoted by $\Pi \in B\text{-}complete_{\mathrm{C}}$, if additionally Π itself is contained in B.

Following our notational convention, we would refer to an $\mathcal{NP}$-hard problem as being $\mathcal{NP}\text{-}hard_{\mathcal{P}}$. For example, the problem of electing a leader is well known to be CF-$hard_{\mathrm{WO}}$ since once a leader is available, this leader can assign unique identifiers to all other nodes and solve the problem centrally. Our definition yields the three hardness classes CF-$hard_{\mathrm{RW}}$, CF-$hard_{\mathrm{WO}}$ and RW-$hard_{\mathrm{WO}}$, allowing us to study how algorithms running in anonymous networks relate to centralized algorithms as well as how the two output revocability notions relate among each other. By definition, every CF-$hard_{\mathrm{WO}}$ problem is both CF-$hard_{\mathrm{RW}}$ and RW-$hard_{\mathrm{WO}}$; it turns out that the converse direction is also true. In Section 2.6 we will have the necessary tools to prove this statement, as cast in the following theorem.

Theorem 2.3. *The hardness classes satisfy*

$$\text{CF-}hard_{\mathrm{WO}} = \text{CF-}hard_{\mathrm{RW}} \cap \text{RW-}hard_{\mathrm{WO}}.$$

2.5 Problem Zoo

In this section, we study the computability and hardness of various problems in our setting. A total of 21 problems are investigated as depicted in Figure 2.2, including variations of approximation guarantees or output specification. First, we will focus on the computability of each problem, i.e., whether it is in WO, in RW $\setminus$ WO, or in CF $\setminus$ RW. Later in Section 2.5.2, we will investigate the hardness of each of the problems. Based on that, we establish Theorem 2.3 in Section 2.6.

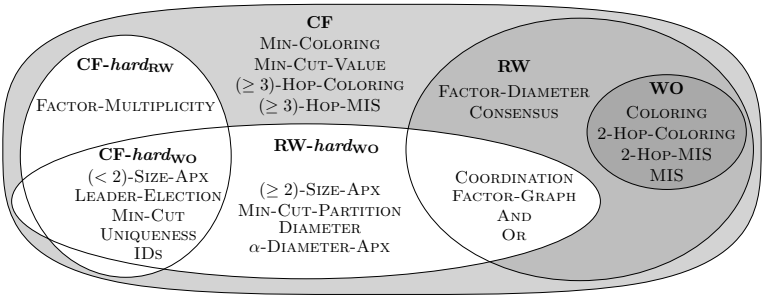

Figure 2.2: Classes CF, RW and WO, and the respective hardness classes.

2.5.1 Computability

Almost all results regarding (non-)computability of problems derived in this section are obtained using one of two general proof frameworks. To characterize problems that can be solved by RW-algorithms, we find a necessary and sufficient condition. For the class WO, we use a necessary condition that allows us to rule out the inclusion of problems in this class. All but one result on non-computability can then be derived using the two characterizations. For computability of problems in RW, the same characterization can be used, while for problems in WO we refer to known algorithms.

Graph Factors, Products and Local Views

The key to our characterization of problems in RW is the notion of *graph factors*.[3]

Definition (Graph Factors)**.** Let G and H be two simple undirected graphs and ℓ_G and ℓ_H two labelings of G and H, respectively, such that ℓ_G and ℓ_H share the same co-domain. A surjective function $f : V(G) \to V(H)$ is called a *factorizing map* of G inducing H if it has the following properties:

[3] In the distributed computing literature, the concept of graph factors was also referred to as covering graphs and graph lifts.

(i) if $(u, v) \in E(G)$, then $(f(u), f(v)) \in E(H)$ for every $u, v \in V(G)$, that is, f is a graph homomorphism;
(ii) for every node $v \in V(G)$, the restriction $f|_{\Gamma(v)}$ of f to v's neighborhood is a bijection onto the neighborhood $\Gamma(f(v))$ of v's image $f(v)$, that is, f is *locally* one-to-one and onto; and
(iii) the labeling functions satisfy $\ell_G(v) = \ell_H(f(v))$ for every node $v \in V(G)$, that is, f preserves the labels.

If there exists such a factorizing map f, then we say that (G, ℓ_G) is a *product* of (H, ℓ_H) or equivalently, that (H, ℓ_H) is a *factor* of (G, ℓ_G). A labeled graph (G, ℓ_G) is *prime* if all factors of (G, ℓ_G) are isomorphic, i.e., if the only factor of (G, ℓ_G) is the graph itself.

The above definition essentially corresponds to the definition given in [58] for covering graphs extended to respect node labels. It is a known fact that $|V(G)|$ must be an integer multiple m of $|V(H)|$ (see, e.g., [58]). We say that (G, ℓ_G) is an *m-product* of (H, ℓ_H) or equivalently that (H, ℓ_H) is an *m-factor* of (G, ℓ_G), denoted $(G, \ell_G) \cong m \cdot (H, \ell_H)$ and $m \cdot (H, \ell_H) \cong (G, \ell_G)$, respectively, when we want to emphasize the specific value of m. It will be convenient to use the notation $(G, \ell_G) \cong m \cdot (H, \ell_H)$ without explicitly specifying m as well, in which case the exact value of m is typically not important. Note however that an m-product of a graph is not necessarily unique (not even for $m = 1$). For two unlabeled graphs G and H, we assume that ℓ_G and ℓ_H both assign the same label to all nodes, and we omit the labeling functions in our notation.

We will use factors of graphs to derive a characterization for problems with input and output labelings i and o of a graph G, respectively. Note that the combined labeling (i, o) is also a labeling of G in which every node v is labeled by the pair $(i(v), o(v))$. If (G, i) is an m-product of (G, i') by a factorizing map f and o is a valid output labeling of (G, i), then we denote the labeling $o'(\cdot) = o(f(\cdot))$ as the *natural extension* of o to (G', i'). Observe, that in this case $(G', i', o') \cong m \cdot (G, i, o)$.

Product graphs are used in the existing literature to derive negative results for computability of problems by anonymous distributed algorithms, dating back to the seminal work of Angluin [7]. Those proofs are based on *lifting* a computation that occurs in a graph (G, i) to some product $(G', i') \cong m \cdot (G, i)$ and forcing node $v' \in V(G')$ to copy the execution of its

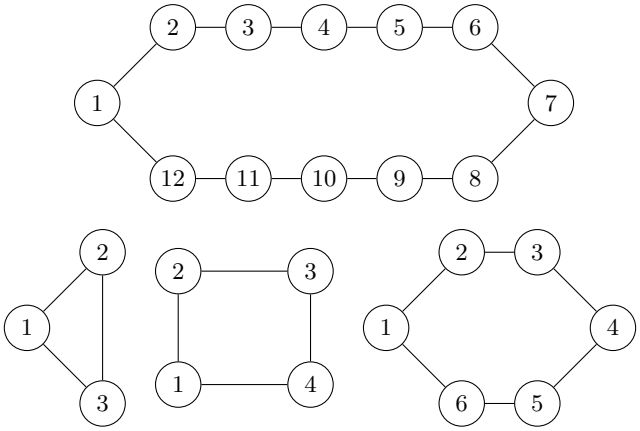

Figure 2.3: The cycles C_3, C_4 and C_6 on 3, 4 and 6 nodes are factors of the 12-cycle C_{12} by mapping node i in C_{12} to the node $i \pmod{3, 4 \text{ or } 6}$ in the respective cycle. The prime factors of C_{12} are C_3 and C_4.

image under the factorizing map f. This technique was used, for example, to prove the impossibility of electing a leader in anonymous networks [7], and the same technique can be used to show that Leader-Election is not in RW. As it turns out, graph products actually lead to a complete characterization of problems in RW.

Theorem 2.4 (Characterization of RW)**.** *Problem* Π *is in* RW *if and only if*

$$\begin{aligned}
&\forall (G,i) \in \Pi,\ \exists o : (G,i,o) \in \Pi \ \textit{s.t.} \\
&\forall (G',i') \in \Pi,\ \exists o' : (G',i',o') \in \Pi \ \textit{s.t.} \\
&(G',i') \cong m \cdot (G,i) \implies (G',i',o') \cong m \cdot (G,i,o)\,. \qquad (2.1)
\end{aligned}$$

Consider a problem Π whose input instances are arbitrary labeled graphs with $O(\Pi) = \{\text{YES}, \text{NO}\}$, and fix some subset Y of the input instances. The problem Π is is called a *(distributed) decision problem* (cf. [60, 68]), if for every $(G,i) \in Y$, all nodes must output YES and for every

input instance $(G, i) \notin Y$, at least one node outputs NO. The instances in the set Y are referred to as the *YES-instances* of Π. Theorem 2.4 implies that the class of decision problems in RW is exactly the class of decision problems that are *closed* under taking products of the solved problem instances, namely if $(G, i, o) \in \Pi$ and $(G', i', o') \cong m \cdot (G, i, o)$, then $(G', i', o') \in \Pi$.

The proof of Theorem 2.4 relies in part on the aforementioned lifting technique [7]. More specifically, fix some instance (G, i) and let (G', i') be a product of that instance by the factorizing map f. For every node $v \in V(G)$, let $\eta(v)$ denote the execution of an algorithm $\mathcal{A}$ that is invoked on (G, i) from the perspective of v. Note that η is fully determined by the random bits used by each node in the course of $\mathcal{A}$'s execution. Denote by $\eta'(\cdot) := \eta(f(\cdot))$ the natural extension of η to (G', i'). In η' every node v will perform exactly the same execution as its image $f(v)$, and if an output o is reached in execution η of $\mathcal{A}$, then the output $o'(\cdot) = o(f(\cdot))$, i.e., the natural extension of o to (G', i'), is reached in execution η. We shall refer to execution η' as *lifting* η from (G, i) to (G', i') and conclude with the following lemma.

Lemma 2.6 (Lifting an Execution [7])**.** *Consider some* RW*-algorithm $\mathcal{A}$ and let (G, i) and (G', i') be two labeled graphs satisfying $(G', i') \cong m \cdot (G, i)$ with factorizing map $f : V(G') \to V(G)$. For every finite execution η of $\mathcal{A}$ on (G, i) ending in a ready configuration with output o, there exists a finite execution η' of $\mathcal{A}$ on (G', i') ending in a ready configuration with output o' such that $o'(v) = o(f(v))$ for every $v \in V(G')$.*

In particular, if no valid output labeling for (G, i) can be naturally extended to a valid output labeling for (G', i'), then it is also not possible for an algorithm to (always) return a correct output in both graphs. Theorem 2.4 is also closely related to the Factor-Graph problem introduced later in the result statements, and therefore deferred until then. A necessary condition for problems in WO can be defined using *local views*.

Definition (Local View)**.** Consider some randomized algorithm $\mathcal{A}$. Let (G, ℓ) be a labeled graph and let v be a node in $V(G)$. Fix some assignment β of random bits to the nodes and denote by $\beta_t(v)$ the (finitely many) random bits used by v in all rounds $r \leq t$. The *depth-t local view* of v under β is the rooted tree $L_t^\beta(v)$ of depth t with a labeling ℓ_t defined as

follows. For every node v, the local view $L_0^\beta(v)$ contains only a single vertex[4] $\mathfrak{r}$ and the labeling $\ell_0(\mathfrak{r})$ is $(\ell(v), \deg(v), \beta_0(v))$. From the labeled forest $F_t(v) := \{L_t^\beta(u) \mid u \in \Gamma(v)\}$, the depth-$(t+1)$ local view $L_{t+1}^\beta(v)$ is constructed in two steps: (1) Prune the sub-tree corresponding to node v from the root vertex $\mathfrak{r}_u$ of every $L_t^\beta(u)$, $u \in \Gamma(v)$, to obtain the pruned local view $L_t'^\beta(u)$; let $F_t'(v) = \{L_t'^\beta(u) \mid u \in \Gamma(v)\}$ be the forest containing the pruned local views of v's neighbors. (2) Construct $L_{t+1}^\beta(v)$ from the pruned local views in $F_t'(v)$ by introducing a new root $\mathfrak{r}$ as the parent of $\mathfrak{r}_u$ for all $u \in \Gamma(v)$. The labeling $\ell_{t+1}(\mathfrak{r}) := (\ell(v), \deg(v), \beta_{t+1}(v))$, whereas for all nodes in the pruned sub-trees $L_t'^\beta(u)$ of $\mathfrak{r}$, the labeling remains unchanged. In cases where no assignment of random bits is assumed the *(deterministic) depth-t local view* $L_t(v)$ is obtained in the same way by excluding $\beta_t(v)$ in the vertex labels.

Informally, the depth-t local view of node v captures the network from v's point of view in round t. Local views without random bits were used before, e.g., to discuss solvability of leader election in the context of deterministic anonymous algorithms [95]. Theorem 2.5 relies on the possibility that nodes whose executions are indistinguishable from v's perspective under deterministic algorithms may remain indistinguishable from v's perspective for a finite amount of time also under randomized algorithms.

Theorem 2.5. *Problem* Π *is not in* WO *if*

$$\exists (G,i) \in \Pi \text{ s.t. } \forall o : (G,i,o) \in \Pi, \forall t \in \mathbb{N}, \exists (G',i') \in \Pi \text{ s.t.}$$
$$\forall o' : (G',i',o') \in \Pi, \exists v \in G, \exists v' \in G' \text{ s.t.}$$
$$L_t(v) = L_t(v'), \text{ and} \tag{2.2}$$
$$o(v) \neq o'(v'). \tag{2.3}$$

Proof. Let $W(\Pi)$ denote the characterization for a problem Π stated in the theorem. Assume for the sake of contradiction that there exists a problem $\Pi \in$ WO for which $W(\Pi)$ holds and let $\mathcal{A}$ be a WO-algorithm solving Π. Invoke $\mathcal{A}$ on the input instance (G,i) promised by $W(\Pi)$ to obtain $\mathcal{A}$'s output o after t steps and denote the random bits used by node v up to round t in this execution of $\mathcal{A}$ by $\beta_t(v)$. Let $(G',i') \in \Pi$ be the

[4] To avoid the confusion between the basic elements in the graph G and those in the rooted tree $L_t^\beta(v)$, we refer to the former as nodes and to the latter as vertices.

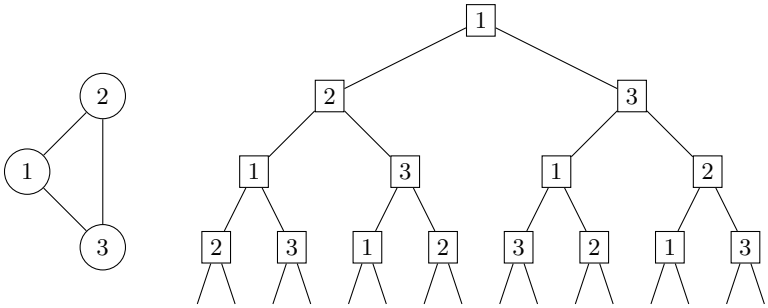

Figure 2.4: Cycle on 3 nodes and the corresponding local view of depth 4 as seen by node 1.

labeled graph promised by $W(\Pi)$ for (G, i, o) and t. For every valid output o' to (G', i'), the property $W(\Pi)$ guarantees the existence of two nodes $v \in V(G)$ and $v' \in V(G')$ satisfying both (2.2) and (2.3). Constraint (2.2) implies that with positive probability nodes v and v' observe the same execution up to (and including) round t, namely if $L_t^{\beta}(v) = L_t^{\gamma}(v')$ for some assignment of random bits γ to nodes in G'. Therefore, with positive probability, v' will return an output $o'(v') = o(v)$. But (2.3) implies that o' cannot be a valid output for (G', i'), in contradiction to the assumption that algorithm $\mathcal{A}$ solves Π. □

Results

We start by briefly stating the proof techniques derived from Theorems 2.4 and 2.5 that we use to establish computability results.

$\Pi \notin$ WO: The inclusion of Π in WO will be disproved by finding an input instance $(G, i) \in \Pi$ and for all valid outputs to (G, i) and arbitrary t, a construction of an input instance $(G', i') \in \Pi$ in which the depth-t local view of some node $v' \in V(G')$ is the same as that of some node $v \in V(G)$, but the output of v' must differ from that of v.

$\Pi \notin$ RW: The inclusion of Π in RW will be disproved by finding an input instance $(G, i) \in \Pi$ and for all valid outputs o to (G, i), an input instance $(G', i') \in \Pi$ satisfying $(G', i') \cong m \cdot (G, i)$ such that no natural

extension of o to (G', i') is a valid output for that instance.

$\Pi \in$ RW: The inclusion of Π in RW will be established by showing that for every input instance $(G, i) \in \Pi$, there is a valid output o such that for every input instance $(G', i') \in \Pi$ satisfying $(G', i') \cong m \cdot (G, i)$, the natural extension of o to (G', i') is a valid output for that instance.

The two techniques for RW rely on Theorem 2.4, which we did not prove yet. Therefore, after giving a brief overview of problems known to be in WO, we will focus on proving the theorem first.

MIS and other Local Symmetry Breaking. The well studied symmetry breaking tasks Maximal-Independent-Set (MIS), $(\Delta + 1)$-Co-lo-ring and Maximal-Matching are indeed in WO: The famous Luby-algorithm [5,78] satisfies the WO condition already. Similarly, there are algorithms to solve $(\Delta+1)$-Co-lo-ring [76][5] and Maximal-Matching [65] that are WO-algorithms. Two other problems studied before are 2-Hop-MIS and 2-Hop-Coloring in which two nodes in the independent set or two nodes having the same color, respectively, must not have a common neighbor. In [47] both problems were found solvable by WO-algorithms using an even weaker computational model. The algorithm from [47] that solves 2-Hop-Coloring uses up to $\Delta^2 - \Delta + 1$ colors, which is a simple upper bound on the number of required colors.

Factor-Graph. In the Factor-Graph problem, nodes in the network (G, i) are required to agree on a factor (H, j) of (G, i). That is, every node $v \in G$ should output the same factor (H, j) of (G, i) (with inputs and uniquely named nodes), and its own name $f(v)$ in H, where f is the factorizing map inducing H. Had we proven Theorem 2.4 already, it would follow from the definition that Factor-Graph it is in RW. Instead we use this problem to establish the theorem, starting with the following observation which is essential for the first half of the proof.

Lemma 2.7. *There is a* RW*-algorithm solving* Factor-Graph.

[5]The algorithm for $(\Delta + 1)$-Co-lo-ring described in the cited work also works if no upper bound on Δ is known by replacing a node of degree d in the overlay graph with a complete graph on $d + 1$ nodes.

Proof. We present a RW-algorithm $\mathcal{A}$ that solves FACTOR-GRAPH on arbitrary input instances (G, i). Algorithm $\mathcal{A}$ progresses in phases where during each phase p, every node v constructs a *candidate factor* (G_p, i_p). Nodes in $V(G_p)$ are identified by a randomly chosen (candidate) *identifier* $\beta_p(v)$, and an edge $\{\beta_p(u), \beta_p(v)\}$ is added to $E(G_p)$ if the edge $\{u, v\}$ is present in $E(G)$. All nodes $v \in V(G)$ start in phase 1, and advance from phase p to $p + 1$ if v sends or receives an *inhibiting message* for phase p.

In the beginning of a phase p, all nodes v first choose a random bit string $\beta_p(v)$ containing p random bits. Node v then exchanges $\beta_p(v)$, its input $i(v)$, and its degree $\deg(v)$ with every neighbor. After v received a message containing the corresponding values of every neighbor, it broadcasts a *my-neighborhood* message $M_p(v)$ containing $(\beta_p(v), \deg(v), i(v))$, and the corresponding values of all its neighbors. While v receives my-neighborhood messages $M_p(u)$ from other nodes u, node v gradually constructs its candidate factor (G_p, i_p) by inserting the node $\beta_p(u)$ with the label $i(u)$ contained in $M_p(u)$, and edges to all of u's neighbors. Note that some edges may point to nodes that were not yet inserted into the graph. We say that v detects an *inconsistency*, if either two messages $M_p(u) \neq M_p(u')$ are received for which $\beta_p(u) = \beta_p(u')$, or if a message from a node u with degree $\deg(u)$ was received that did not contain $\deg(u) + 1$ different identifiers for u and its neighbors. When v detects an inconsistency it broadcasts an inhibiting message for phase p. A node v sending an inhibiting message for the current phase p sets its output register to ε and starts phase $p + 1$. If v did not receive an inhibiting message for a phase p and all endpoints of edges in (G_p, i_p) were inserted, then v returns the output $((G_p, i_p), \beta_p(v))$.

We start the analysis of algorithm $\mathcal{A}$ by showing that $\mathcal{A}$'s output is correct if a ready configuration is reached. For this, observe that if two neighboring nodes u and v are in different phases p_u and p_v respectively, then u or v is currently broadcasting an inhibiting message and is therefore not ready. When on the other hand all nodes are in the same phase p and all nodes are ready, then no node detected an inconsistency in (G_p, i_p). Therefore the returned graph (G_p, i_p) is the same graph for every node, and we have to show that β_p is a factorizing map inducing (G_p, i_p). The function β_p is surjective, because every node in $V(G_p)$ has a preimage in G. Further β_p is a graph homomorphism since for every edge $\{u, v\}$ in G

the edge $\{\beta_p(u), \beta_p(v)\}$ is inserted into G_p. The inconsistency detection ensures that the restriction $\beta_p|_{\Gamma(v)}$ is an injection on $\Gamma(\beta_p(v))$ for every node v. Because the input labeling $i_p(\beta_p(v))$ is defined by the input value assigned to v, the function β_p respects the graph labeling, and we conclude that $m \cdot (G_p, i_p) \cong (G, i)$ for some m. It is left to show that $\mathcal{A}$ reaches a ready configuration with probability 1. But this will happen at latest in a phase p_0 in which every node chooses a unique random identifier, because this ensures that every my-neighborhood message is unique. In this case the algorithm will return a graph G_{p_0} that is isomorphic to G. □

Having established that FACTOR-GRAPH is a problem in RW, we now present the proof of Theorem 2.4.

Proof of Theorem 2.4. For the proof, let $R(\Pi)$ denote the graph theoretic characterization (2.1) stated in the theorem, that is

$$\begin{aligned} R(\Pi) = &\forall (G,i) \in \Pi, \exists o : (G,i,o) \in \Pi \text{ s.t.} \\ &\forall (G',i') \in \Pi, \exists o' : (G',i',o') \in \Pi \text{ s.t.} \\ &(G',i') \cong m \cdot (G,i) \implies (G',i',o') \cong m \cdot (G,i,o). \end{aligned}$$

We wish to prove that $\Pi \in \text{RW} \Leftrightarrow R(\Pi)$, and we prove both directions of the if and only if separately.

if: Let Π be a distributed problem that satisfies $R(\Pi)$; we prove that then Π must be in RW. To accomplish that, we describe a RW-algorithm $\mathcal{A}$ solving Π with access to a FACTOR-GRAPH-oracle. Since FACTOR-GRAPH is solvable by a RW-algorithm without access to any oracle (Lemma 2.7) and oracles are sound (Theorem 2.2), this is sufficient to conclude that $\Pi \in \text{RW}$. The key to algorithm $\mathcal{A}$ is to invoke the FACTOR-GRAPH-oracle until it returns a valid input instance (G, i) of Π. For every such instance, the characterization $R(\Pi)$ promises the existence of a valid output o to (G, i) satisfying that for every product $(G', i') \cong m \cdot (G, i)$, with $(G', i') \in \Pi$, the natural extension o' of o to (G', i') is a valid output for (G', i'). Algorithm $\mathcal{A}$ exploits that as follows.

Fix some instance $(G, i) \in \Pi$. At the beginning of round r, node v appends a random bit to the (initially empty) string $\beta_{r-1}(v)$ to obtain $\beta_r(v)$. Then, node v invokes the oracle with input $(i(v), \beta_r(v))$. In all rounds $r > 1$ the oracle output register of every node v contains a labeled

graph $(H_r, (j_r, \gamma_r))$ satisfying $(H_r, (j_r, \gamma_r)) \cong m \cdot (G, (i, \beta_{r-1}))$, and every node receives a name $f_r(v) \in V(H_r)$ assigned to v by the factorizing map inducing H_r. Node v now checks whether (H_r, j_r) is an input instance of Π. If it is, then v chooses the lexicographically smallest o_r that satisfies $R(\Pi)$ for (H_r, j_r) and writes $o_r(f_r(v))$ to its output register.

When in round r every node v returns some output $o_r(v)$, the output of algorithm $\mathcal{A}$ is valid for the instance (G, i) on which the algorithm is executed, because $(G, i) \cong m \cdot (H_r, j_r)$. Algorithm $\mathcal{A}$ will reach a stable ready configuration with probability 1 within finite time, since the output from the oracle will satisfy $(H_r, j_r) \cong 1 \cdot (G, i)$ in round r if every node tossed a unique random string $\beta_{r-1}(v)$ in round $r-1$. Notice that $\mathcal{A}$ does not need to change its output register once it wrote to it, which allows us to conclude that in fact Factor-Graph is in fact RW-*complete*$_{\text{WO}}$.

only if: For the sake of contradiction, assume that $\neg R(\Pi)$ holds for some problem $\Pi \in \text{RW}$, that is

$$\begin{aligned}\neg R(\Pi) = {} & \exists (G, i) \in \Pi \text{ s.t. } \forall o : (G, i, o) \in \Pi, \\ & \exists (G', i') \in \Pi \text{ s.t. } \forall o' : (G', i', o') \in \Pi : \\ & (G', i') \cong m \cdot (G, i) \wedge \neg\big((G', i', o') \cong m \cdot (G, i, o)\big).\end{aligned}$$

Let $\mathcal{A}$ be a RW-algorithm solving Π, let η be an execution of $\mathcal{A}$ on the instance (G, i) promised by $\neg R(\Pi)$, and let o be the output of $\mathcal{A}$ obtained in η. Note that o satisfies $(G, i, o) \in \Pi$, and therefore the property $\neg R(\Pi)$ guarantees the existence of some $(G', i') \in \Pi$ with $(G', i') \cong m \cdot (G, i)$ such that $(G', i', o') \cong m \cdot (G, i, o)$ does not hold for any o', i.e., the natural extension of o to (G', i') is not a valid output to (G', i'). Lift the execution η of $\mathcal{A}$ on (G, i) to obtain the execution η' of $\mathcal{A}$ on (G', i'). By Lemma 2.6 we see that $\mathcal{A}$'s output o' in execution η' is the natural extension of o to (G', i'), contradicting the assumption that $\mathcal{A}$ solves Π. □

As stated in the *if*-part of the proof, Factor-Graph is RW-*complete*$_{\text{WO}}$, and as such cannot be solved by a WO-algorithm.

Corollary 2.8. *Finding a factor of the input graph is* RW-*complete*$_{\text{WO}}$.

Coordination. In coordination problems nodes in the network keep track of some shared state and wish to determine whether their shared

state is in unison. This kind problem arises for example in atomic commit protocols and in the two generals problem, where all participants in the network need to agree on the same value before they can proceed. More formally, we consider the problem Coordination where the input instances are all labeled graphs, and the solved instances satisfy the following. If all nodes are labeled with the same input label, then all nodes output "UNISON", otherwise there is at least one pair of nodes with different labels and all nodes output "DISCORD".

Coordination is not contained in WO. Let (G, i) be the 3-cycle with input 0, so the output to this instance is "UNISON" for all nodes in G, and let v be any node in $V(G)$. For arbitrary t, let further (H, j) be the cycle on $2t + 1$ nodes, in which exactly one node w gets input 1, and let v' be the node in $V(H)$ furthest away from w. The depth-t local views of v and v' are equal, but while node $v \in V(G)$ must return "UNISON" the only correct output of $v' \in V(G')$ is "DISCORD".

We stated that the class RW is essentially the class of coordination problems, and indeed we use the characterization of Theorem 2.4 to show that Coordination is in RW. For this, let (G, i) be a labeled graph in which all nodes get the same input x. In all products of (G, i) every node has input x, so returning "UNISON" leads to a correct output in all products of (G, i). On the other hand, if (G, i) contains two nodes $u \neq v$ with different inputs $x \neq y$ respectively, then all its products also contain nodes with different inputs x and y, and therefore the output "DISCORD" for all nodes can be extended to all products of (G, i). In Section 2.5.2 we show that Coordination is complete in RW.

Logical And & Or. The definition for the problems And and Or are straigt-forward: All nodes are provided with a binary input value and have to compute the logical conjunction resp. disjunction of all those inputs.

The problem And (Or) is not contained in WO for essentially the same reason as Coordination: Let (G, i) be the 3-cycle with input 1 (0), so the only admissible output to this instance is 1 (0), and let v be any node in $V(G)$. For arbitrary t, let further (H, j) be the cycle on $2t + 1$ nodes, in which exactly one node w gets input 0 (1), and let v' be the node in $V(H)$ furthest away from w. The depth-t local views of v and v' are equal, but v and v' are not allowed to return the same output.

The two problems are, however, both in RW (again for similar reasons as COORDINATION is). If an input instance (G, i) contains a node with input 0 (1 for OR), then all products of (G, i) also contain a node with that input, and the only correct output for every node is 0 (1) in these instances. If on the other hand no node in (G, i) has input 0 (1), then the only correct output in (G, i) is 1 (0) for every node, which is also true for all products of such an input instance.

Consensus. In the well-known binary CONSENSUS problem nodes can have either 0 or 1 as possible input. All nodes are required to agree on the same output, which must also appear as input to at least one node.

Like the COORDINATION problem CONSENSUS is also not solvable by a WO-algorithm, but Theorem 2.5 cannot be used to disprove that. Instead, assume for the sake of contradiction that there is a WO-algorithm $\mathcal{A}$ solving CONSENSUS. Let (G_0, i_0) and (G_1, i_1) be 3-cycles, where in G_0 every node gets input 0 and in G_1 every node gets input 1. Execute $\mathcal{A}$ on both instances to obtain correct output labelings o_0 and o_1 after t_0 and t_1 rounds, respectively, and denote the random bits used in the execution of $\mathcal{A}$ up to round t on each respective instance by β_t and γ_t. Let u_0 and u_1 denote two arbitrary nodes in G_0 and G_1. Let further G' be the cycle consisting of $2 \cdot (t_0 + t_1 + 1)$ nodes, and denote by u'_0 and u'_1 two nodes in G' with maximal distance. It is possible to assign inputs and random bits δ_t to all nodes so that $L^{\beta}_{t_0}(u_0) = L^{\delta}_{t_0}(u'_0)$ and $L^{\beta}_{t_1}(u_1) = L^{\delta}_{t_1}(u'_1)$, i.e., the two nodes u'_0 and u'_1 in G' observe the same depth-t_0 and depth-t_1 local view under δ_t as the corresponding nodes u_0 and u_1 did in G_0 and G_1, respectively. Thus, there is an execution of $\mathcal{A}$ (by choosing the random bits as determined by δ_t) which leads to a configuration where the output returned by u'_0 will be 0, while that of u'_1 will be 1, contradicting our assumption.

However, we can show that CONSENSUS is in RW by applying Theorem 2.4. For this, let (G, i) be a labeled graph in which all nodes get input 0. In all products of (G, i) every node has input 0, so agreeing to output 0 is valid in all products of (G, i). On the other hand, if (G, i) contains a node with input 1, then all its products also contain a node with input 1, and therefore the output in which all nodes agree on 1 can be extended to all products of (G, i).

Factor-Multiplicity. Another problem related to graph factors is FACTOR-MULTIPLICITY: In an *unlabeled* network G, every node should output the multiplicity m of a graph H such that $m \cdot H \cong G$, and the number of nodes in H is minimal among all possible factors of G. The last constraint prohibits the nodes from answering 1 in every graph (each graph is of course a factor of itself).

The problem is not solvable for RW-algorithms: Let G be any prime graph, for example a 3-cycle, such that the smallest factor of G has multiplicity 1. In any non-trivial product of G, this answer is however not correct. Using this problem we will establish in Section 2.5.2 that the two hardness classes CF-$hard_{\mathrm{RW}}$ and CF-$hard_{\mathrm{WO}}$ are distinct.

Factor-Diameter. Agreeing on the diameter of some factor of an input instance is certainly possible in the RW model, as nodes able to agree on a factor, and may just output its diameter. To see that the problem is not solvable by a WO-algorithm let G be the 3-cycle, so that the only admissible outputs for G will be those in which the agreed upon factor H is a 3-cycle and all three nodes choose a different name. For every t, construct the cycle G' on p nodes where $p > t$ is prime so that in particular, G' is prime. Any arbitrarily chosen $v \in V(G)$ and $v' \in V(G')$ satisfy $L_t(v) = L_t(v')$, but the only admissible output $o'(v')$ is $\lfloor p/2 \rfloor$ for every $v' \in V(G')$. However, the two problems differ in their hardness, as we will see in Section 2.5.2.

k-Hop-MIS, k-Hop-Coloring and Min-Coloring. In the k-HOP-MIS problem, nodes shall output a maximal set in which any two nodes in the set have at least distance k (measured in hops), i.e., a shortest path between them uses $k + 1$ edges. Similarly, in a solution to the k-HOP-COLORING problem, nodes having the same color must be at least k hops apart. As we saw earlier, both problems are in WO for $k \leq 2$.

For $k > 2$ they are not in RW, and neither is coloring with the minimum amount of colors. To see this for k-HOP-COLORING let G be the triangle so that every solution to G will use exactly three different colors. Now, let $G' \cong 2 \cdot G$ be the 6-cycle. Any valid k-HOP-COLORING of G' with $k > 2$ needs to use six colors, thus the natural extension of a valid output o to G cannot be a valid output for G'. On the other hand, a

MIN-COLORING of the 6-cycle only needs two colors, therefore a natural extension of o to G' will not be a minimum coloring on the 6-cycle. Similarly, for k-HOP-MIS, a 2-product of a valid output on the 3-cycle violates the distance requirement on the 6-cycle.

Diameter and approximating it. We consider the problem of finding the DIAMETER of the network as well as the approximation problem DIAMETER-APX. The problem is not in RW, which can be seen by—again—taking G to be the triangle (with diameter one), and G' to be the cycle on six nodes with diameter three. The only valid output labeling for G cannot be extended to a valid output labeling on G'. As for the approximation problem with approximation ratio α, let G' be the $3\lfloor\alpha + 1\rfloor$-cycle. We will however see in Section 2.5.2 that both problems are prime examples for the class of problems that are RW-$hard_{\mathrm{WO}}$.

Min-Cut. A surprisingly interesting problem to study is the MIN-CUT problem. A *cut* of a graph $G = (V, E)$ is a subset $C \subseteq E$ so that the graph becomes disconnected when the edges from C are removed from G. A cut C is called *minimum cut* if C has the smallest possible cardinality, and we refer to this cardinality as the *value* of the minimum cut. Any cut can also be described by a partition of the nodes into two subsets, and in that case C is the set of edges with one endpoint in each of the two partitions. There are basically three ways to define the MIN-CUT problem in our setting: We can require nodes to output the *value* of the minimum cut, a *partition* of the nodes (say, into black and white nodes) inducing the minimum cut, or thirdly, we can ask for the combination of *both*. We denote those output specification variants by MIN-CUT-VALUE, MIN-CUT-PARTITION and MIN-CUT, respectively. This does not change their computability (no variant can be solved in an anonymous network), but we will in Section 2.5.2 find that the exact specification *does* make a difference for the hardness of each single variant.

To prove that none of the problem variants is in RW, first observe that the graph G in Figure 2.5 has a unique minimum cut. Therefore, every valid output to the min-cut problem in G must output the cut-value and/or partition indicated in the figure. Because the graph G' in

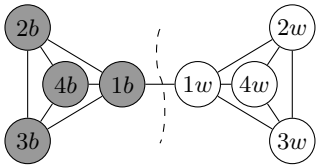

Figure 2.5: Graph G, with unique minimum cut 1. Black and white nodes indicate the two partitions in the output of the minimum cut.

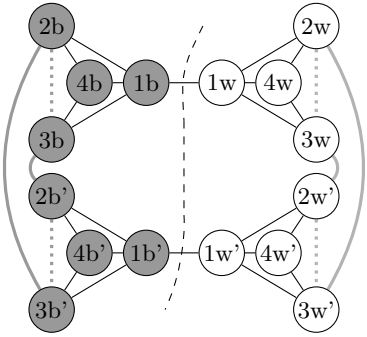

Figure 2.6: Graph G', which is a 2-product of G from Figure 2.5, with increased cut value and double the number of nodes. The dashed edges are replaced by the corresponding thick edges without violating the graph factor property.

Figure 2.6 is a product of G, but has a different cut value, MIN-CUT-VALUE is not solvable in RW. From this we can immediately follow that MIN-CUT can also not be solved. To see that MIN-CUT-PARTITION is also not solvable, we alter G' slightly to obtain G'' (depicted in Figure 2.7) in

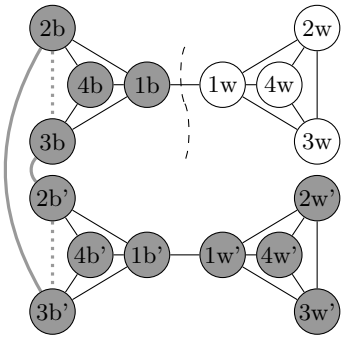

Figure 2.7: Graph G'', which is a 2-product of G from Figure 2.5, with the same cut value and double the number of nodes. The dashed edges are replaced by the corresponding thick edges without violating the graph factor property.

which only the edges {2b, 3b} and {2b', 3b'} are replaced to connect {2b, 3b'} and {2b', 3b} (as indicated in the figure), while the edges connecting $\{2w, 3w\}$ and $\{2w', 3w'\}$ are left unchanged. Graph G'' has a cut of size 1 and is also a product of G. Thus the only valid output o to G cannot be naturally extended to obtain a valid output in every product of G.

Computing with unique identifiers: Leader-Election et cetera. In the Leader-Election problem, we demand of a valid solution that there is exactly one node with output "leader", and all other nodes return "not leader". As mentioned above, [7] showed that Leader-Election is not even solvable in the RW model. It is well understood (see, e.g., [88]) that accessing a Spanning-Tree-oracle, or an oracle that equips every node with a unique identifier (the IDs problem) is equivalent to having a single leader already for WO-algorithms. The Uniqueness problem, in which nodes have to test whether every node is supplied with a unique input and output "ALL UNIQUE" or "NOT ALL UNIQUE" depending on the outcome of this test, is CF-$complete_{\mathrm{WO}}$ as well. (Nodes can find unique identifiers by invoking the Uniqueness-oracle with random strings of increasing length until it replies with "ALL UNIQUE", but it is not in RW because no solution for the 3-cycle can be extended to a solution on

the 6-cycle). Similarly, knowing the network SIZE n can be considered equivalent in our model, because nodes can broadcast random identifiers of increasing length until they observe exactly n different identifiers. On the other hand, an approximation SIZE-APX of the network size with approximation guarantee α for the triangle G is not a valid approximation on a ring of size $3\lfloor\alpha + 1\rfloor$, convincing us that not even on cycles one can find a α-SIZE-APX with a RW-algorithm. In [88] the authors present a Monte Carlo algorithm to construct a spanning tree that can be turned into a Las Vegas algorithm if a $(2 - \varepsilon)$-SIZE-APX is known. Using our terminology, the same can be seen by giving a RW-algorithm access to an apply-once oracle to α-SIZE-APX and computing a FACTOR-GRAPH. Denote the true network size by n, the approximation provided by the oracle by $\bar{n}$ and the number of nodes in a computed factor by n_{F}. If it is guaranteed that $\bar{n} < 2 \cdot n$, then a found factor H of G is indeed G itself if and only if $n_{\text{F}} \leq 2 \cdot \bar{n}$, since $|V(G)|$ must be an integer multiple of $|V(H)|$. On the other hand, when the approximation factor $\alpha \geq 2$, the answer 6 can be supplied from the oracle in both the triangle graph as well as in the ring of six nodes, and a RW-algorithm with access to such an oracle has no means of distinguishing the two.

2.5.2 Hardness of Problems

The last discussion already gave us an understanding of problems that are known to be CF-$hard_{\text{WO}}$, and while investigating FACTOR-GRAPH we already found the problem to be RW-$complete_{\text{WO}}$. In determining the exact containment of each problem introduced in the last section, we find all three hardness classes CF-$hard_{\text{WO}}$, CF-$hard_{\text{RW}}$ and RW-$hard_{\text{WO}}$ and also the three corresponding classes of complete problems to be non-empty and distinct.

Results

To show a problem $\Pi \in \text{B}$ is B-$hard_{\text{C}}$ it will be sufficient to describe a C-algorithm that solves a complete problem for C with access to a Π-oracle. In order to fully classify each problem we are also interested in negative results regarding completeness to completely characterize each

of the studied problems. The following techniques derived from Theorems 2.4 and 2.5 will be used to show that a problem to not be in one of the hardness classes.

$\Pi \notin$ CF-*hard*$_{\mathrm{RW}}$: To prove that a problem Π does not empower RW-algorithm to solve problems in CF we start with a graph (G, i). From this, we construct an m-product $(G', i') \cong m \cdot (G, i)$ with $m > 1$. We will have to ensure that there is a sequence of oracle answers to Π supplied to nodes in (G, i) that is also a valid sequence of oracle answers to the corresponding nodes in (G', i'). This is less of a problem, if Π does not take any input (other than the topology of the graph itself), because the same oracle can be used in every round. Treating the answers supplied by the oracle as additional input labels to each node, this disproves the existence of a RW-algorithm accessing an oracle as an implication of Theorem 2.4.

$\Pi \notin$ RW-*hard*$_{\mathrm{WO}}$: One needs to show that there is a problem Π' in RW that cannot be solved in the WO model, even if an oracle supplies each node with a solution to Π. We find an input instance $(G, i) \in \Pi$, and for all valid outputs $(G, i, o) \in Pi$ and finite t, we describe the construction of a graph (G', i'). In the construction we will specify two nodes $v' \in V(G')$ and $v \in V(G)$, and a sequence of t oracles for each graph, such that depth-t local view of v (including the answers supplied by the oracles) is the same as that of v', but the output of v' must differ from that of v. Treating the answers supplied by the oracle as additional input labels to each node, Theorem 2.5 then implies that Π cannot be in WO.

We omit the previously discussed results on problems that are CF-*hard*$_{\mathrm{WO}}$ and the completeness of Factor-Graph, and start by presenting another problem that is complete in RW with respect to WO.

Coordination. Indeed, Coordination is RW-*complete*$_{\mathrm{WO}}$. We show how to turn a RW-algorithm $\mathcal{A}_{\mathrm{RW}}$ solving Π without access to an oracle into a WO-algorithm $\mathcal{A}_{\mathrm{WO}}$ that solves Π with access to an Coordination-oracle. In algorithm $\mathcal{A}_{\mathrm{WO}}$ every node v will simulate one round of $\mathcal{A}_{\mathrm{RW}}$ in every round; we denote v's simulated output register of $\mathcal{A}_{\mathrm{RW}}$ by ρ_{RW}, and the actual output register of $\mathcal{A}_{\mathrm{WO}}$ by ρ_{WO}. If in round r the register $\rho_{\mathrm{RW}} = \varepsilon$, then v writes "NOT READY" to the input register of the oracle, otherwise it invokes the oracle with input "READY". When the oracle answers "UNISON" in round $r + 1$ and node v was ready in round r, the

network was in a ready configuration in round r, and v sets ρ_{WO} to the value contained in ρ_{RW} in round r.

Logical And & Or. The problem AND as well as OR can be used in a similar way to determine whether the network is in a ready configuration at the end of every simulated round. In the previous construction we used to show hardness of COORDINATION, one only needs to replace the value "NOT READY" with 0 if the simulating algorithm is accessing a AND-oracle (with 1 for an OR-oracle), and the value "READY" with 1 (0) respectively.

Consensus. CONSENSUS is not RW-$hard_{\text{WO}}$. We prove this by showing that an oracle to CONSENSUS cannot be used to solve OR with a WO-algorithm. Once more, let (G, i) be the 3-cycle with input 0 so the only admissible output to OR for this instance is 0. For any t, let (H, j) be the cycle on $2t + 1$ nodes, in which exactly one node w gets input 1 while all other nodes get input 0. Let v be any node in $V(G)$ and denote by $\nu_r(v)$ the content stored in v's oracle input register in round $r - 1$ so that the value $\nu_r(v)$ is a valid answer from the CONSENSUS-oracle in round r for all nodes in G. Let further v' be the node in $V(H)$ with maximum distance to w. Since $L_t(v) = L_t(v')$ the value $\nu_r(v)$ is the same as $\nu_r(v')$, the value that v' provides to the oracle in round r, for all $r \leq t$. Regardless of the input of w to the oracle, the answers $\nu_r(v) = \nu_r(v')$ are also valid in (H, j) for $r \leq t$. Thus we have $L_t(v) = L_t(v')$, but $o(v) = 0$, while in a valid output labeling o', node v' must output 1. Note that the same reasoning can also be used to disprove hardness of other problems such as k-SET-AGREEMENT.

Factor-Multiplicity. We show that the problem of finding the multiplicity of a smallest (by number of nodes) factor of an unlabeled graph is CF-$hard_{\text{RW}}$. To establish that, we define the helper problem Π_M. The input instances $(G, i) \in \Pi_M$ are all graphs G in which the label assigned to every node by i is the multiplicity m of the smallest factor H s.t. $G \cong m{\cdot}H$. An output labeling o is valid, if in o, exactly one node is labeled "leader" while all others are labeled "not leader". Observe that $\Pi_M \in$ RW if and

only if there exists a RW-algorithm $\mathcal{A}$ that solves Leader-Election with an access to an (apply-once) Factor-Multiplicity-oracle.

We argue that Π_M is indeed in RW. To that end, we prove that Π_M fulfills the characterization of Theorem 2.4. Let G be some graph and let H be its smallest (prime) factor, where $G \cong m \cdot H$. By definition, the input labeling i satisfies $i(v) = m$ for every node $v \in V(G)$. Consider some graph $(G', i') \cong c \cdot (G, i)$. If $c > 1$, then $(G', i') \cong (c \cdot m) \cdot H$ and therefore, (G', i') is not an input instance of Π_M for the input labeling i' that assigns m to all nodes. Otherwise, if $c = 1$, then the factorizing map $f : V(G) \to V(H)$ is in fact an isomorphism and for every valid output o to (G, i), the natural extension $o(f(\cdot))$ is a valid output to (G', i'). The argument follows since every graph is either prime or a product of another prime graph.

Next, we show that Factor-Multiplicity is not RW-$hard_{\text{WO}}$. To see this, we argue that a WO-algorithm with access to a Factor-Multiplicity-oracle cannot solve Or. Again, let (G, i) be the triangle in which all nodes get input 0, so in a correct output to this instance all nodes will agree on 0; the only valid answer from the oracle is multiplicity 1 since the triangle is a prime graph. Now, for arbitrary t let (G', i') be the cycle on $p > 2t$ nodes where p is prime in which all but one node have input 0, and exactly one node w' gets input 1. Let v be any node in (G, i) and let v' be a node in (G', i') furthest away from w'; in particular, the distance between v' and w' is least t. Then both v and v' observe the same depth-t local view, but in a valid output to (G', i') all nodes must agree to return 1.

Factor-Diameter. As we have established the Factor-Graph problem as being RW-$hard_{\text{WO}}$, one might think that a WO-algorithm with access to a Factor-Diam-oracle is able to solve problems in RW. This is not the case, and thus Factor-Diam is another example of a problem in RW that is not RW-$hard_{\text{WO}}$. The sufficient condition stated in Theorem 2.5 is not strong enough to disprove this problems hardness. We will however use a very similar technique that relies on *multiple* nodes in G to "reappear" in G'.

As usual, let G be the triangle, so the only valid answer from the oracle is 1, and denote by v_1, v_2 and v_3 the three nodes in G. For any t, let G' be

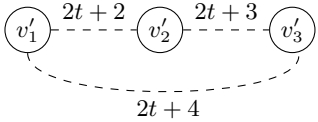

Figure 2.8: Counterexample showing that FACTOR-DIAM is not RW-*hard*$_{\text{WO}}$. Dashed edges indicate a path of the denoted length, node labels indicate the mapping of nodes to the 3-cycle factor.

the ring of size $3(2t+4)$ as depicted in Figure 2.8. Then G is a factor of G', and therefore 1 is a valid answer of the FACTOR-DIAM-oracle to any node in G'. The paths between the three nodes depicted in Figure 2.8 contain at least $2t+2$ nodes. For every finite t, fix an assignment β_t of random bits to nodes. Because the t-hop neighborhoods of the three nodes in G' do not overlap, it is possible to find γ_t that satisfies $L_t^{\beta}(v_k) = L_t^{\gamma}(v'_k)$ for $k \in \{1,2,3\}$. This assignment of random bits γ will occur in G' with positive probability. Because the local views of v_1, v_2 and v_3 in G' will then be the same as that in G, they will also return the same output. In particular, all will agree on the triangle as the factor, and each one will map itself do a different node in the triangle. But this means each path between v'_1, v'_2 and v'_3 must be of length 1 (mod 3). This cannot be the case, because the three paths have different lengths even modulo 3, contradicting our assumption.

k-Hop-MIS, k-Hop-Coloring and Min-Coloring. None of these problems is CF-*hard*$_{\text{RW}}$ nor RW-*hard*$_{\text{WO}}$ for any constant k. Let G be the $(2k+1)$-cycle, and fix a solution (G, s) to k-HOP-MIS, k-HOP-COLORING or MIN-COLORING to be the oracle supplied to nodes in G. To see that none of the problems is CF-*hard*$_{\text{RW}}$, let $G' \cong 2 \cdot G$ be the $4k+2$-cycle and s' a natural extension of s to G', so s' is also a valid oracle to the chosen problem. But since G' is a 2-product of G a RW-algorithm cannot elect a leader in both G and G', not even with access to an oracle to one of the problems k-HOP-MIS, k-HOP-COLORING or MIN-COLORING. We use the same graph G to show that they are not RW-*hard*$_{\text{WO}}$ by arguing why OR cannot be solved using a WO-algorithm with access to an oracle to any of the three problems. Let (G, i) be an input instance to OR on

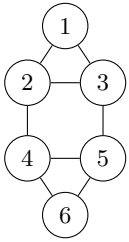

Figure 2.9: Graph G with diameter 3.

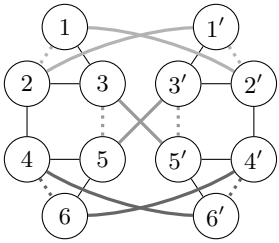

Figure 2.10: Graph G', which is a 2-product of G from Figure 2.9, with diameter 3. The six dotted edges were changed as indicated by the thick edges.

the $(2k+1)$-cycle in which every node v in G receives input $i(v) = 0$ so that for all nodes the only valid output is $o(v) = 0$. For arbitrary t, let $(H_t, j_t) \cong t \cdot (G, i)$ be the cycle on $t \cdot (2k+1)$ nodes in which exactly one node w receives input 1, while all other nodes v' get input 0. Because the problems do not depend on any input, we may safely assume an oracle to one of the problems supplies the same answer in every round. Denote by s_t the natural extension of s from (G, i) to (H_t, j_t) such that s_t is a valid oracle in (H_t, j_t). The node v' in (H_t, j_t) which is furthest away from w and its corresponding node v in G satisfy $L_t(v) = L_t(v')$, even when taking s_t into account, and while the only valid output of v is 0, node v' must output 1.

Diameter and approximating it. Approximating the diameter to an arbitrary factor of α (including 1) is not CF-$hard_{\mathrm{RW}}$. To see this, observe that the graph G in Figure 2.9 is a factor of G' in Figure 2.10. Since both have diameter 3, the answers supplied by an oracle in G are also valid for corresponding nodes in G'. However, a valid output for Leader-Election in G cannot be naturally extended to obtain a valid output in G'. In contrast to that, the approximation problem α-Diameter-Apx is RW-$hard_{\mathrm{WO}}$ for arbitrary approximation guarantees α. This is the case because a WO-algorithm can solve Or by broadcasting all input values

for D rounds if the answer supplied by the oracle to α-DIAMETER-APX is D. If after D rounds of broadcasting node v received a message containing input 1, then v returns 1, otherwise v returns 0. It follows that any upper bound on the network diameter (or its size) is RW-$hard_{\mathrm{WO}}$. This is true even if not all nodes are equipped with the same upper bound, which simplifies the proof regarding the hardness result of MIN-CUT-PARTITION for RW with respect to WO in the following section.

Min-Cut. Neither MIN-CUT-VALUE nor MIN-CUT-PARTITION are in the class CF-$hard_{\mathrm{RW}}$, because the two graphs G and G' from Figures 2.5 and 2.6 have the same minimum cut value (share the "same" partition inducing the minimum cut). Thus the same answer supplied by the oracle to G is also valid in G' for MIN-CUT-VALUE or MIN-CUT-PARTITION, respectively, but G' has twice the number of nodes than G. MIN-CUT-VALUE is also not RW-$hard_{\mathrm{WO}}$. To see that, observe that all cycles share a minimum cut size of 2. With the same argument as for the FACTOR-MULTIPLICITY problem, a WO algorithm cannot use this information to solve the OR problem. On the other hand, we find that MIN-CUT-PARTITION is RW-$hard_{\mathrm{WO}}$.

Claim 2.9. MIN-CUT-PARTITION *is* RW-$hard_{\mathrm{WO}}$.

Proof. We show that using a WO-algorithm $\mathcal{A}$ with access to an (apply-once) oracle for the MIN-CUT-PARTITION problem, every node v can determine an upper bound on the diameter. Since α-DIAMETER-APX is RW-$hard_{\mathrm{WO}}$ for any α, this is sufficient to prove our claim. Given a *partition* of the network into *black* and *white* nodes, we refer to a node which has a neighbor in the opposite partition as a *border node*. By the term *depth* of a node v we denote the minimum distance of v to a border node within the same partition, i.e, border nodes have depth zero. Observing that the degree of each node is an upper bound on the cut size, the main idea is now to bound the diameter of the network in terms of the maximum depth of a node in each partition.

To accomplish that, algorithm $\mathcal{A}$ proceeds in three stages: The only purpose of the first stage is to locally gather necessary information from the oracle prior to the second stage. The main stage of $\mathcal{A}$ is the second one, in which nodes compute an upper bound on the diameter of each

partition individually. Lastly, the third stage's role is to combine the two individual upper bounds to compute an upper bound on the diameter of the whole network, and disseminate the bound throughout the network.

In the beginning of the first stage of algorithm $\mathcal{A}$ each node v invokes the Min-Cut-Partition-oracle to determine whether it is in the black or in the white partition. Thereafter, node v sends a message containing the name of its partition to all of its neighbors, so that every node can determine whether it is a border node. Every non-border node v initializes its depth value $d(v) \leftarrow \infty$, while border nodes b set $d(b) \leftarrow 0$. Additionally, border nodes b initialize a value $D(b) = 0$ to keep track of the maximum observed depth inside b's partition. Following that, all nodes enter the second stage of $\mathcal{A}$.

The first round of the second stage starts with all border nodes b sending the message $(D \geq 0)$ to all neighbors within the same partition. This initiates parallel breadth-first searches inside each partition to determine the depth of every node, and to report back the maximum depth of a node inside each partition. More specifically, when a node v with $d(v) < \infty$ receives a message $(D \geq i)$ for some i, then v forwards this message to all members within the same partition. If on the other hand a node v with $d(v) = \infty$ receives a message $(D \geq i)$, then v does not forward this message, but instead it sets $d(v) \leftarrow i + 1$ and broadcasts the message $(D \geq i + 1)$ among all members of the same partition. A border node b that receives a message $(D \geq i)$ additionally updates $D(b) \leftarrow i$ accordingly, thus keeping track of the maximum observed depth of a node inside its partition. The crucial point is that a border node b enters stage three, if it does not receive an update to $D(b)$ in round $2D(b) \cdot (\deg(b) + 1) + 2$ of stage two. In other words, a border node b adjusts the time at which it will enter stage three of algorithm $\mathcal{A}$ with each update to $D(b)$.

When both nodes b and w of a cut edge $\{b, w\}$ have entered stage three of algorithm $\mathcal{A}$, both b and w exchange their corresponding values $D(w)$ and $D(b)$. If a border node b has exchanged $D(b)$ with its neighbor w, broadcasts the message $(\text{diam} \leq (2 \cdot \deg(b) + 1) \cdot (D(b) + D(w)))$. Nodes v receiving a message $(\text{diam} \leq i)$ for the first time enter stage three and set their output register to i after forwarding the message to all their neighbors.

We have to prove two things, namely that the upper bound broadcast

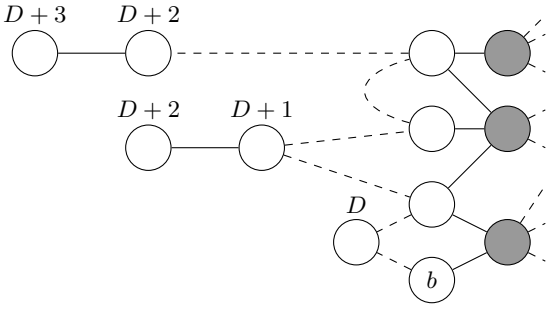

Figure 2.11: Illustration of the proof that MIN-CUT-PARTITION is RW-$hard_{\mathrm{WO}}$. A border node b that has observed a node at depth $D, D+1$ or $D+2$ will wait long enough to receive a message from the closest node next in depth.

by a border node is indeed an upper bound for the diameter, and that no border node broadcasts an incorrect value prematurely. To see the latter, it suffices to show that any border node b as illustrated in Figure 2.11 will wait long enough before starting with the third stage of the algorithm. Our proof will be by induction on the maximum depth $d(v)$ of a node v, where we show that if b has received a message ($D \geq D(b)$), then it will also receive a message indicating a maximum depth of $D(b)+1$ from a node of that depth that is closest to b, if there is one. The initial step is for $d(v) = 0$, i.e., the partition of b contains only border nodes, in which case there is nothing to show. For the induction step, let v be a node with depth $d(v)$, and assume the border node b has learned the maximum depth of a node within the partition is at least $D(b) = d(v) - 1$ by the induction hypothesis. Let b' denote a border node closest to b having a node v' at distance $d(v)$ with $d(v') = \mathit{depth}(v)$. The distance from b to b' is bounded by $2 \cdot \deg(b) \cdot D(b)$, since the shortest path between b and b' that remains inside the same partition contains at most $\deg(b)$ border nodes, and the distance between any two border nodes on that path is at most $2 \cdot D(b)$. Any node u with depth i will broadcast its distance in round i of stage two, and this message will arrive at the closest border nodes (in distance i) after $2i$ rounds. Thus, node b' will start forwarding the message ($D \geq d(v')$) received from v' in round $t_{v',b'} = 2d(v')$ of stage two,

and this message has to travel at most additional $t_{b',b} = 2 \cdot \deg(b) \cdot D(b)$ steps to reach b. But $t_{v,b'} + t_{b',b} = 2 \cdot (D(b) + 1) + 2 \cdot \deg(b) \cdot D(b)$ is exactly the round until which b waits for a message indicating a possibly larger depth. Thus, no border node prematurely transmits an incorrect partition depth to its neighboring nodes in the opposite partition.

Lastly, the found value is indeed an upper bound on the diameter by a similar argument. The degree $\deg(b)$ of b in is an upper bound for the size of the cut, and therefore a shortest path between any two nodes u and v can cross the cut at most $\deg(b)$ times and contains at most $\deg(b)$ border nodes. □

It follows that MIN-CUT is also RW-$hard_{\text{WO}}$, since the output includes the value of a minimum cut and, in particular, a valid output to MIN-CUT-PARTITION. In fact, MIN-CUT is CF-$hard_{\text{RW}}$ as is established in the following proof.

Claim 2.10. MIN-CUT *is* CF-$hard_{\text{RW}}$.

Proof. We describe an algorithm $\mathcal{A}$ that elects a leader among all border nodes of the white partition. This will be accomplished by choosing a new random identifier for every white border node in each round. The main insight is that it can be checked whether every node tossed a unique identifier by counting the number of cut edges incident to different identifiers, and comparing this number to the size of the cut provided by an oracle. To that end, in a preliminary step of $\mathcal{A}$ every node v once invokes the oracle to MIN-CUT so that v is supplied with the value of the minimum cut k and its partition denoted by *black* or *white*. After obtaining an answer from the oracle every node sends a message indicating the partition it is in to all neighbors, enabling every node v to determine the number $c(v)$ of cut edges incident to v.

For the remainder of algorithm $\mathcal{A}$ all white border nodes w, i.e., nodes inside the white partition with $c(w) > 0$, are referred to as *candidate leaders*. Nodes inside the black partition and white nodes with $c(v) = 0$ set their output register $\rho \leftarrow$ "not leader". In every round r, every candidate leader w chooses an identifier $\beta_r(w)$ by appending one random bit to the previous identifier $\beta_{r-1}(w)$, and broadcasts the message $M = (r, \beta_r(w), c(w))$ among all white nodes. If some node v receives two

messages M, M' containing the same round numbers r and identifiers $\beta_r(w)$, but a different number of cut edges $c(w)$, then v broadcasts an *inhibiting message* for round r. For all candidate leaders w denote by $M_w(s)$ the set of different messages M that were sent in round s and received by w so far. Denote further by $c_w(s) := \sum_{(s,\beta,c)\in M_w(s)} c$ the total number of cut edges from different nodes received by w for round s, and by s_w the smallest non-inhibited round s for w that satisfies $c_w(s) = k$, i.e., the first non-inhibited round in which every candidate leader tossed a different identifier. Whenever s_w is not defined in round r, node w sets its output register $\rho \leftarrow \varepsilon$. When on the other hand s_w is defined in round r, node w checks whether the identifier it chose in round s_w was the smallest among those appearing in $M_w(s_w)$. If this is the case, node w sets $\rho \leftarrow$ "leader", otherwise it sets $\rho \leftarrow$ "not leader".

To see that the algorithm is correct, assume for the sake of contradiction that in round r all nodes are ready and two different candidate leaders u, w output "leader". Since all nodes are ready no node is currently sending an inhibiting message, and because both u and w output "leader" the round numbers s_u and s_w are both defined and not inhibited for any node; assume w.l.o.g. that $s_u \leq s_w$. On the other hand, the value $c_u(s_u)$ must be the same as $c_w(s_w)$, namely they must both be k. This can only be the case if both u and w have received a message from all candidate leaders for round s_u and s_w respectively . In round r node w received all messages sent by other candidate leaders in round s_w, and therefore w must also have received all the messages sent by candidate leaders in round s_u. Because round s_u is not inhibited for any node we conclude that $s_u = s_w$ and since no inhibiting message is being sent in round r also $M_u(s_u) = M_w(s_w)$ must hold, contradicting that both u and w output "leader". Lastly, algorithm $\mathcal{A}$ will reach a ready configuration after every candidate leader tossed a unique identifier. □

One can also give a WO-algorithm solving Leader-Election with access to a Min-Cut oracle by using a more careful construction and analysis. It is however easier to see that Min-Cut is CF-$hard_{\mathrm{WO}}$ by applying Theorem 2.3, i.e., because Min-Cut is both CF-$hard_{\mathrm{RW}}$ and RW-$hard_{\mathrm{WO}}$ it must also be CF-$hard_{\mathrm{WO}}$. This completes our effort to identify in which classes each of the presented problems lie, and we turn ourselves

to proving the missing link in the last argument, namely Theorem 2.3.

2.6 Proof of Theorem 2.3

The techniques introduced in Section 2.3 together with the completeness result for OR found in Section 2.5.2 allow us to present a proof for Theorem 2.3. A key ingredient in the proof is the notion of a *fork*, which is a sub-process of the execution dedicated to simulating some algorithm $\mathcal{A}$. The fork's name $[r]$ will indicate the round number in which the simulation was started. A fork $[r]$ dedicated to $\mathcal{A}$ encapsulates the complete state required to simulate $\mathcal{A}$, and messages sent and received by $[r]$ are identified by the fork's name.

The theorem states that if a problem Π is both CF-$hard_{\mathrm{RW}}$ and RW-$hard_{\mathrm{WO}}$, then it is also CF-$hard_{\mathrm{WO}}$. Let $\Pi \in \text{CF-}hard_{\mathrm{RW}} \cap \text{RW-}hard_{\mathrm{WO}}$ be a problem satisfying the premise. Denote by $\mathcal{A}_{\mathrm{LE}}$ a RW-algorithm solving LEADER-ELECTION with an access to a Π-oracle, and by $\mathcal{A}_{\mathrm{OR}}$ a WO-algorithm solving OR with an access to a Π-oracle respectively. Employing Lemma 2.3, we assume that $\mathcal{A}_{\mathrm{LE}}$ in fact sustainably solves LEADER-ELECTION. We wish to establish the assertion by presenting a WO-algorithm $\mathcal{A}$ solving LEADER-ELECTION with access to a Π-oracle.

Of course, algorithm $\mathcal{A}$ cannot directly simulate $\mathcal{A}_{\mathrm{LE}}$ because it is a RW-algorithm. We would therefore like to perform multiple simulations of $\mathcal{A}_{\mathrm{OR}}$ in order to detect a ready configuration of $\mathcal{A}_{\mathrm{LE}}$. Unfortunately, these multiple simulations cannot be carried out concurrently since each one of them requires its own independent access to the Π-oracle, whereas $\mathcal{A}$ accesses the Π-oracle only once per round. Instead, we will use a careful forking mechanism to schedule disjoint accesses to this scarce resource.

Algorithm $\mathcal{A}$ simulates $\mathcal{A}_{\mathrm{LE}}$ in phases, starting from phase 1, where each phase p is responsible for executing round p of the simulation of $\mathcal{A}_{\mathrm{LE}}$. Indeed, in round 1 of phase p, node v executes round p of this simulation accessing the Π-oracle. Following that, node v initiates a fork called $[p]$ dedicated to the simulation of $\mathcal{A}_{\mathrm{OR}}$. The input to fork $[p]$ is 0 if v was ready in round p under $\mathcal{A}_{\mathrm{LE}}$ (v observes that from the outcome of round 1 of phase p); the input is 1 otherwise. In the next p rounds of the phase, forks $[1], [2], \ldots, [p]$ (all dedicated to $\mathcal{A}_{\mathrm{OR}}$) are executed, one fork per round (say, in lexicographic order), so in total phase p consists

of $p + 1$ rounds. The output of $\mathcal{A}$ is determined as follows: if fork $[r]$ for some $r \leq p$ has output 0 during phase p, then v writes the output value of $\mathcal{A}_{\text{LE}}$'s round r (which was obtained during phase r) to $\mathcal{A}$'s output register.

The fixed execution order of the forks simulating $\mathcal{A}_{\text{OR}}$ guarantees that every fork $[p]$ is executed in a synchronized manner, that is, all nodes execute round r of this fork in the same round under $\mathcal{A}$. The logic of OR guarantees that fork $[r]$ of $\mathcal{A}_{\text{OR}}$ has output 0 if and only if round r under $\mathcal{A}_{\text{LE}}$'s simulation is in a ready configuration. Since $\mathcal{A}_{\text{OR}}$ is a WO-algorithm, node v can immediately rely on a returned 0 value to conclude that this indeed happened. Moreover, as $\mathcal{A}_{\text{LE}}$ is sustainably solving the leader election problem, the output returned by v under $\mathcal{A}$ must lead to a correct output for LEADER-ELECTION, thus establishing Theorem 2.3.

3

The Role of Randomness

Our goal in this chapter is to investigate the role that (Las-Vegas type) randomness plays in the computational power of anonymous message passing algorithms (referred to hereafter as *anonymous algorithms*), regardless of round and message complexity considerations. However, before we can do so, care must be taken to rule out distributed problems in which unique IDs are (perhaps implicitly) encoded in the input instances as those mock cases obviously do not faithfully represent the properties of distributed computability in anonymous networks. To that end, we restrict our focus to the class GRAN (standing for Genuinely solvable by Randomized algorithms in Anonymous Networks) of distributed problems Π that satisfy (1) there exists a randomized anonymous algorithm that solves Π; and (2) there exists a randomized anonymous algorithm that decides whether a given graph is a legal input instance of Π; we refer to Section 3.1 for a formal definition. Notice that with the exception of some artificial cases

generated for the purpose of investigating the leader election problem, essentially all interesting distributed problems studied in the existing literature in the context of anonymous networks belong, in fact, to GRAN.

What exactly characterizes the computational power of a randomized anonymous algorithm as opposed to a deterministic one? Surprisingly, randomization is only required to establish a 2-*hop coloring* of the network: Once a 2-hop coloring is known, every problem in GRAN can be solved by a deterministic anonymous algorithm. More precisely, we prove that every problem that can be solved (and verified) by a *randomized* anonymous algorithm can also be solved by a *deterministic* anonymous algorithm provided that the latter is equipped with a 2-*hop coloring* of the input graph. Since the problem of 2-hop coloring a given graph (i.e., ensuring that two nodes with distance at most 2 have different colors) can by itself be solved by a randomized anonymous algorithm, it follows that with the exception of a few mock cases (those not in GRAN), the execution of every randomized anonymous algorithm can be decoupled into a generic preprocessing randomized stage that computes a 2-hop coloring, followed by a problem-specific deterministic stage. The main ingredient of our proof is a novel simulation method that relies on some surprising connections between 2-hop colorings and an extensively used graph lifting technique.

3.1 Preliminaries and Genuine Solvability

2-Hop Colored Problems. Consider some distributed problem Π. The 2-*hop colored variant* Π^c of Π is the problem defined as follows: the input instance set Π^c is

$$\Pi^c = \{(V, E, i, c) : (V, E, i) \in \Pi \text{ and } c \text{ is a 2-hop coloring of } (V, E)\}$$

and given an input instance $I = (V, E, i) \in \Pi$, the valid output labeling set for every corresponding input instance $I^c = (V, E, i, c) \in \Pi^c$ is $\Pi^c(I^c) = \Pi(I)$.

Local Views. Given a node v in the labeled graph $G = (V, E, \ell)$, we denote by $L_d(v, G)$ a rooted tree called the *depth-d local view* of v in

G. (When G is clear from the context we may write $L_d(v)$ instead.) To avoid confusion, we distinguish between nodes and labels in G and *vertices* and *marks* in $L_d(v)$. The local view of node v is defined inductively as follows: $L_1(v)$ consists of a single vertex x marked with $\ell(v)$; $L_{d+1}(v)$ is the tree obtained by connecting the root of $L_d(u)$ as a child of $L_1(v)$'s root for every $u \in \Gamma(v)$. Refer to Figure 3.1 for an illustration. Notice that the the local view $L_d(v)$ in G essentially captures all information that a deterministic algorithm $\mathcal{A}$ executed by node v in G could possibly gather in d rounds of execution. The *depth-infinity local view* of a node v is the infinite tree $L_\infty(v)$ obtained from the inductive construction of $L_d(v)$ by taking d to infinity.

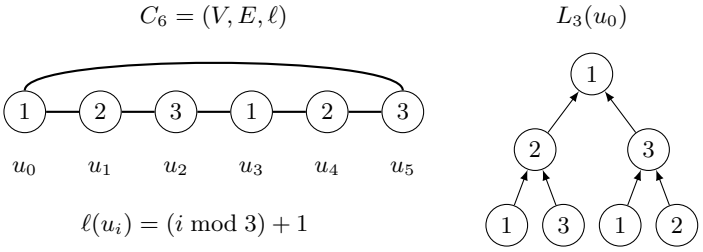

Figure 3.1: Depth-3 local view of node u_0 in the labeled graph C_6.

Genuine Solvability. Let Y be a set of labeled graphs called *yes-instances*. The *distributed decision problem* Δ_Y obtained from Y (see, e.g., [54]) is the problem whose input instances I are all labeled graphs, and whose valid output labelings $o \in \Delta_Y(I)$ are such that all nodes output "YES" if $I \in Y$ and at least one node outputs "NO" if $I \notin Y$. We say that problem Π is *genuinely* solvable by randomized algorithms in anonymous networks if (1) there exists a randomized anonymous algorithm that solves Π; and (2) there exists a randomized anonymous algorithm that solves the distributed decision problem Δ_Π, namely, the problem of deciding whether a given labeled graph is an input instance of Π. Denote the class of such problems Π by GRAN (standing for Genuinely solvable by Randomized algorithms in Anonymous Networks).

Classic distributed symmetry breaking problems such as maximal independent set and graph (1-hop) coloring are known to be in GRAN. Common to these two problems is the local nature of their symmetry breaking challenges.[1] While the 2-hop variant of graph coloring is still solvable by randomized anonymous algorithms and thus, belongs to GRAN, it is not difficult to show that this no longer holds for its k-hop variant for any $k > 2$. (In fact, the same can be said for maximal independent set under a natural extension to k-hop variants that are not discussed in this chapter, cf. [83].) Is this a coincidence? Does GRAN contain problems that require (systematic) symmetry breaking between nodes which are more than two hops apart? This chapter's main result provides a negative answer to these questions.

Theorem 3.1. *If $\Pi \in$ GRAN, then Π^c is solvable by a deterministic anonymous algorithm.*

3.2 Related Work

The seminal work by Angluin [7] established the connection between computation in networks and *factors/products* of graphs (see Section 3.3 for a definition) and marks the beginning of history for distributed computability theory. Her work employs a lifting technique from a graph to its products to establish impossibility of leader election (and equivalent problems, e.g., assigning IDs), even under the assumption of Las-Vegas algorithms. As stated in [57] graph products also characterize recognizable cases for graph rewriting systems, a localized model for distributed computation. Fibrations, i.e, a related generalization for directed graphs (see [35] for an extensive overview), were found to characterize problems solvable by self stabilization (a possibly incorrect output stabilizes to a correct one) in [36]. The lifting technique also plays a key role in our proof of Theorem 3.1.

Product graphs are also studied in their own right (see, e.g., [13, 74]), also products obtained from a random process [6]. For a graph G the

[1] Despite the inherent locality of the notions of maximal independent set and coloring, the results of [73, 76] show that the corresponding distributed computational tasks cannot be solved in constant time.

universal cover $U(G)$ (cf. [7]) is a (possibly infinite) product of G closely related to the depth-infinity local view. (The un-rooted tree $U(G)$ can be obtained from $L_\infty(v)$ of any node v in G by (1) for every vertex x in $L_\infty(v)$ pruning x's child corresponding to x's parent; and (2) making every edge in the resulting tree undirected.) In Section 3.4 we apply the result of Norris [86] that isomorphism in $U(G)$ up to depth $|V|-1$ implies isomorphism to all depths to obtain a finite representation of a specific factor of G.

There is a line of research investigating (mock-anonymous) problems where the input instances permit leader election. For example, electing a leader in a ring network is possible if the size n of the ring is known [66,67]. Later it was found that a $(2-\varepsilon)$-approximation of n is enough [1], even in general networks [88]. The impact of prior knowledge (e.g., the network size) on the solvability of various problems was studied in [33, 95]. We restrict ourselves to problems in GRAN, which rules out cases that permit leader election.

Electing a leader with a Monte-Carlo algorithm (a randomized algorithm that is allowed to fail) was studied in rings [66, 67] and in general graphs [3, 87]. Recently, the problem was found to be solvable with high probability (i.e., with probability $1 - n^{-c}$ for any $c \geq 1$, w.h.p. for short) in [82]. Since electing a leader and assigning IDs are equivalent, any distributed problem solvable with IDs is w.h.p. solvable in an anonymous network. On the other hand it is known that some symmetry-breaking problems, e.g., MIS [5,78] or coloring [76], are in GRAN. It is thus a natural question to ask what exactly distinguishes Las-Vegas algorithms from deterministic ones. We find that a 2-hop coloring suffices to completely characterize the capabilities of a Las-Vegas algorithm solving a GRAN problem.

Anonymous networks and electing a leader therein also plays a major role in self-stabilization research, e.g. [43]. Self-stabilizing leader election is possible with population protocols (nodes are controlled by asynchronous finite state machines, cf. [9]) if the network is a ring [29]. In the presence of an oracle the problem becomes solvable also in general networks, however, the required oracle is impossible to implement as a population protocol [28]. We hope that our contribution may also play a role towards a better understanding of randomization in self-stabilization

(cf. [44, 75]).

Naor and Stockmeyer [84] introduced the notion of locally checkable labelings (LCL), a labeling that can be checked by a deterministic constant-time algorithm in a network with IDs. With IDs a randomized constant-time algorithm cannot solve more LCLs than a deterministic one. This is in contrast to the anonymous model, where the run-time is unbounded but finite and a deterministic algorithm requires a 2-hop coloring to replace randomization. The impact of having/not having identifiers on verifying a proof (solution) to a decision problem was studied in [68], and the authors observe that the existence of a uniquely determined leader cannot be verified without identifiers. A hierarchy of decision problems in terms of bit complexity required for a proof is established in [60]. In [53, 54] Monte-Carlo algorithms for decision problems are studied in networks with unidentified but distinguishable nodes and a strict hierarchy depending on the success probability is found. We utilize the notion of decision problems to characterize the problem class GRAN.

The notion of a 2-hop coloring (also referred to as distance-2 coloring) has been used to assign frequencies in radio networks [71], to solve optimization problems in parallel on shared memory computers [56], and to emulate Turing machines in population protocols [8]. The related k-local election problem, where a local leader needs to be unique only up to distance k, was studied in an asynchronous model in [83]. A solution to the 2-hop coloring problem can already be found in the weak model of [47], where nodes are controlled by finite state machines. Minimizing the number of colors in a k-hop coloring is, however, $\mathcal{NP}$-complete for any $k \geq 1$ due to a result in [81]. We would like to note that port numbers are not necessary under the assumption of randomized algorithms. Since each node v knows its degree a 2-hop coloring can be found even without port numbers and by including the sender's color in every message missing port numbers can be emulated. We show that a 2-hop coloring uniquely determines a graph's prime factor.

3.3 The Case for Infinity

Before presenting the proof of Theorem 3.1, we state and prove a slightly easier variant of it that captures some of its main ideas. To that end,

for the moment, assume the following rather strong *infinity model* for anonymous computation. Fix some problem Π, let Π^c be its 2-hop colored variant, and let $I^c = (V, E, i, c) \in \Pi^c$ be some input instance. An algorithm under the infinity model is fully specified by a function $\mathcal{A}_\infty$ from the set of depth-infinity local views to the possible output labels. The algorithm sets the output of node $v \in V$ to be $o(v) = \mathcal{A}_\infty(L_\infty(v))$, namely, it applies the function $\mathcal{A}_\infty$ to $L_\infty(v)$ and uses the returned image as the output of v. We shall use $\mathcal{A}_\infty$ to denote the algorithm under the infinity model as well as the function that lies at its heart. Disregarding computability issues of $\mathcal{A}_\infty$ for the moment, we say that $\mathcal{A}_\infty$ *solves* Π^c if the labeling o satisfies $o \in \Pi^c(I^c)$. Note that the infinity model involves neither communication nor randomization. In other words, node v's output is completely determined by $L_\infty(v)$ in which the vertices are only marked with input labels and a 2-hop coloring. The remainder of this section is devoted to proving the following theorem.

Theorem 3.2. *If $\Pi \in$ GRAN, then Π^c is solvable in the infinity model.*

A key ingredient of our proof for Theorem 3.2 is the notion of an *infinite view graph* G_∞ of a 2-hop colored graph G.

Definition 3.1. Let $G = (V, E, \ell)$ be a 2-hop colored graph. We define the *infinite view graph* $G_\infty = (V_\infty, E_\infty, \ell_\infty)$ of G by identifying $L_\infty(v)$ with $\overset{\infty}{v}$ and setting

$$
\begin{aligned}
V_\infty &:= \{\overset{\infty}{v} : v \in V\} \qquad \text{(the different depth-infinity local views in } G\text{)},\\
E_\infty &:= \{(\overset{\infty}{u}, \overset{\infty}{v}) : (u, v) \in E\},\\
\ell_\infty(\overset{\infty}{v}) &:= \ell(v)
\end{aligned}
$$

Note that $|V_\infty| \leq |V|$, where the inequality is strict when different nodes in G have the same depth-infinity local view For example in the graph C_6 from Figure 3.1 the local views of nodes with the same color are equal.

For the remainder of this section, fix some problem $\Pi \in$ GRAN, let $\mathcal{A}_\mathrm{R}$ be a randomized anonymous algorithm solving Π, and let $I^c = (V, E, i, c)$ be some input instance of Π^c. We would like to construct an algorithm $\mathcal{A}_\infty$ that solves Π^c under the infinity model. The idea behind $\mathcal{A}_\infty$ is to perform the following three steps for each node $v \in V$:

(i) construct the infinite view graph $I^c_\infty = (V_\infty, E_\infty, i_\infty, c_\infty)$ from $L_\infty(v)$;
(ii) simulate a specific terminating execution of $\mathcal{A}_{\mathrm{R}}$ on $J = (V_\infty, E_\infty, i_\infty)$; and
(iii) use the output of node $\tilde{v}$ in that simulation as output for node v.

We now turn to explaining these three steps in detail.

3.3.1 Constructing I^c_∞

Consider $L_\infty(v, I^c)$ for some node $v \in V$. For every node $u \in V$, the local view $L_\infty(u)$ appears as a sub-tree in $L_\infty(v)$. Conversely, every depth-infinity sub-tree of $L_\infty(v)$ is the depth-infinity local view of some node (or nodes) in V. Therefore, the set of all depth-infinity sub-trees of $L_\infty(v)$ is exactly the node set of I^c's infinite view graph I^c_∞. Moreover, (u, u') is an edge in I^c if and only if $L_\infty(u')$ appears as a sub-tree of $L_\infty(u)$ rooted at a child of $L_\infty(u)$'s root. In other words, I^c_∞ can be uniquely constructed from $L_\infty(v)$.

Algorithm $\mathcal{A}_\infty$ and its analysis relies on a canonical representation of the depth-infinity trees $L_\infty(u)$, namely, fixing the order of the vertices in each depth-level. For that purpose, it suffices to fix a total order among the children of each vertex. Such a total order follows immediately by noticing that since I^c is 2-hop colored, every two siblings must have distinct marks. Using these canonical representations, two depth-infinity local views can now be compared level by level, thus implying a total order on V_∞; let $\tilde{u}_1, \ldots, \tilde{u}_k$ be the nodes in V_∞ indexed according to this total order.

3.3.2 Simulating $\mathcal{A}_{\mathrm{R}}$

The second step in $\mathcal{A}_\infty$ is to simulate an execution of algorithm $\mathcal{A}_{\mathrm{R}}$, the randomized anonymous algorithm solving Π. More precisely, multiple executions of $\mathcal{A}_{\mathrm{R}}$ will be simulated on the input $J = (V_\infty, E_\infty, i_\infty)$. A t-round simulation σ of $\mathcal{A}_{\mathrm{R}}$ on J (corresponding to executing $\mathcal{A}_{\mathrm{R}}$ on J for t rounds) is fully determined by an assignment $b : V_\infty \to \{0,1\}^t$ of t random bits to every node in V_∞. We refer to this simulation σ as the simulation *induced by* b. The simulation σ is said to be *successful* if every node $\tilde{v} \in V_\infty$ produces an output under σ.

It will be essential for $\mathcal{A}_\infty$'s correctness that all nodes in I^c choose the same simulation of $\mathcal{A}_\mathrm{R}$ on J — i.e., the same assignment b — to determine their output. This will be accomplished by using the total order on V_∞ to fix a total order among the possible assignments $b : V_\infty \to \{0,1\}^t$. To that end, given two assignments $b_1, b_2 : V_\infty \to \{0,1\}^t$, $b_1 \neq b_2$, the relation $b_1 < b_2$ holds if and only if $(b_1(\tilde{u}_1), \ldots, b_1(\tilde{u}_k)) < (b_2(\tilde{u}_1), \ldots, b_2(\tilde{u}_k))$, where the latter comparison is done lexicographically. For convenience, we extend the total order on the assignments b so that it also covers assignments $b_1 : V_\infty \to \{0,1\}^{t_1}$ and $b_2 : V_\infty \to \{0,1\}^{t_2}$, $t_1 \neq t_2$, by defining that $b_1 < b_2$ holds if and only if $t_1 < t_2$.

Assuming that there exists a successful simulation of $\mathcal{A}_\mathrm{R}$ on J, algorithm $\mathcal{A}_\infty$ selects the successful simulation induced by the smallest assignment $b : V_\infty \to \{0,1\}^t$. We denote this simulation by σ_∞ and summarize in the following lemma.

Lemma 3.2. *If algorithm $\mathcal{A}_R$ returns an output when executed on J, then in $\mathcal{A}_\infty$, all nodes select the same successful simulation σ_∞.*

3.3.3 The Output of $\mathcal{A}_\infty$

The output value $o(v) = \mathcal{A}_\infty(L_\infty(v))$ of node $v \in V$ is set to the output produced by node $\tilde{v}$ in simulation σ_∞ of $\mathcal{A}_\mathrm{R}$ on J. For that to be well defined, there must exist an execution of $\mathcal{A}_\mathrm{R}$ on J in which every node $\tilde{v}$ produces an output (leading to a successful simulation). We establish the existence of such an execution by using the well-known lifting lemma [7,34] to assert that $J \in \Pi$ and thus, guarantee that a terminating execution of $\mathcal{A}_\mathrm{R}$ on J exists. This requires developing a better understanding of the infinite view graph's fundamental properties based on the notion of factor graphs.

Factor Graphs and 2-Hop Colorings

The central concept of our analysis is that of *factor/product graphs*.[2] For two labeled graphs $G = (V, E, \ell)$ and $G' = (V', E', \ell')$, we say that G'

[2] Our notion of product graphs should not be confused with binary operations on two graphs referred to as *graph products*. The unlabeled counterpart of the concept of product graphs is often called graph lifts, or covering graphs in the existing literature. We extend the definition presented in [58] to incorporate node labels. Note that changes are required when considering non-simple or undirected graphs.

is a *factor* of G and that G is a *product* of G' if there exists a function $f : V \to V'$, referred to as a *factorizing map*, that satisfies:

(i) the mapping f is surjective (onto);

(ii) f respects the labeling functions, that is, $\ell(v) = \ell'(f(v))$ for every node $v \in V$; and

(iii) f is a *local isomorphism*, that is, for every node $v \in V$, the restriction $f|_{\Gamma(v)}$ is a bijection onto $\Gamma(f(v))$.

We shall use the notations $G' \preceq_f G$ (and $G \succeq_f G'$) to denote that G' is a factor of G (and G is a product of G'). The role of the factorizing map f is sometimes emphasized by saying that the factor/product is *induced* by f. Refer to Figure 3.2 for an illustration.

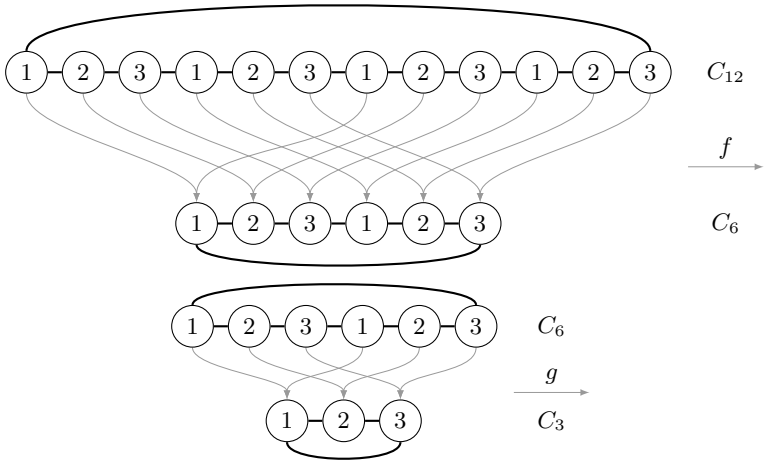

Figure 3.2: The labeled graph C_6 is a factor of C_{12} induced by the factorizing map f (and C_{12} is a product of C_6). Similarly, the labeled graph C_3 is a factor of C_6 induced by the factorizing map g.

It is known that $|V| = m \cdot |V'|$ for some positive integer m (see, e.g., [58]). If $m = 1$, then the factorizing map is bijective and both $G' \preceq_f G$ and $G \preceq_{f^{-1}} G'$ hold. In that case, we refer to the two labeled

graphs G and G' as being *isomorphic* (since f is a graph isomorphism that respects the node labels) and write $G \cong_f G'$ or $G \cong G'$ if the specific bijection is not relevant. Moreover, it is well-established that if $G \succeq_f G'$, then the graphs G and G' are indistinguishable from the perspective of a node v in G or G' (see, e.g., [86]) as cast in the following fact.

Fact 1. Let G, G' be two labeled graphs. If $G \succeq_f G'$, then $L_\infty(v) = L_\infty(f(v))$ for every node v in G.

It follows from Fact 1 that the factor of a 2-hop colored graph is also 2-hop colored. Let $G = (V, E, \ell)$ be a 2-hop colored graph and let $G_\infty = (V_\infty, E_\infty, \ell_\infty)$ be its infinite view graph. Lemmas 3.3 to 3.5 establish some important properties of G and G_∞. These three lemmas can be derived from the results on graph fibrations presented in [35] using an intricate construction. (we briefly sketch this connection in Section 3.5); for completeness, we also present stand-alone proofs. Our first Lemma 3.3 establishes that G_∞ is a factor of G induced by the factorizing map $f_\infty : V \to V_\infty$ that maps v to $\tilde{v}$; we subsequently refer to f_∞ as the *infinite view (factorizing) map* of G.

Lemma 3.3. *Let $G = (V, E, \ell)$ be a 2-hop colored graph, and denote by $G_\infty = (V_\infty, E_\infty, \ell_\infty)$ its infinite view graph, and $f_\infty : V \to V_\infty$ the infinite view map. Then, G_∞ is a factor of G induced by f_∞, i.e., $G_\infty \preceq_{f_\infty} G$.*

Proof. We show that f_∞ is a factorizing map inducing the factor G_∞ by verifying the properties required in the definition of factor graphs. (1) The function $f_\infty : V \to V_\infty$ is surjective by the definition of V_∞. (2) For every $v \in V$, the labeling functions satisfy $\ell(v) = \ell_\infty(f_\infty(v))$ by the definition of ℓ_∞. (3) Bijectivity of $f_\infty|_{\Gamma(v)}$ for every $v \in V$ is established by showing that $f_\infty|_{\Gamma(v)}$ is injective and surjective separately.

To see that $f_\infty|_{\Gamma(v)}$ is injective, observe that two nodes $u_1, u_2 \in \Gamma(v)$, $u_1 \neq u_2$, have different labels $\ell(u_1) \neq \ell(u_2)$ since ℓ is a 2-hop coloring. Property (2) thus ensures that $f_\infty(u_1) \neq f_\infty(u_2)$, i.e., $f_\infty|_{\Gamma(v)}$ is injective. Denote by $\tilde{v}$ the node $f_\infty(v)$ and let $\tilde{u}$ be some neighbor of $\tilde{v}$ in G_∞. It follows from the definitions of E_∞ and local views that the tree $\tilde{u}$ is a sub-tree rooted at a child of $\tilde{v}$'s root vertex. Thus, G admits some node $u \in \Gamma(v)$ with $L_\infty(u) = \tilde{u}$ and the function f_∞ satisfies $f_\infty|_{\Gamma(v)}(u) = \tilde{u}$. Hence, $f_\infty|_{\Gamma(v)}$ is also surjective. □

A labeled graph G is called *prime* (cf. [35]) if all factors of G are isomorphic to it. For example, the labeled 3-cycle C_3 in Figure 3.2 is prime, whereas C_{12} and C_6 are not. Both C_{12} and C_6 have C_3 as a prime factor. Lemma 3.4 states that the only prime factor of a 2-hop colored graph G is its infinite view graph (indeed, C_3 is isomorphic to the infinite view graph of C_{12} and C_6).

Lemma 3.4. *If G is a 2-hop colored graph, then the infinite view graph G_∞ is the unique prime factor of G (up to isomorphism).*

Proof. Let G and G' be 2-hop colored graphs with $G' \preceq_f G$. To establish the statement we show that G' is either isomorphic to G_∞ or not prime. The key to our proof is to show that G and G' have the same infinite view graph; this establishes the assertion due to Lemma 3.3.

To that end, let $G_\infty = (V_\infty, E_\infty, \ell_\infty)$ and $G'_\infty = (V'_\infty, E'_\infty, \ell'_\infty)$ be the infinite view graphs of G and G', respectively, i.e., $G_\infty \preceq_{f_\infty} G$ and $G'_\infty \preceq_{f'_\infty} G'$. Fact 1 implies that $V'_\infty = V_\infty$. The construction of E_∞ guarantees that an edge $(\widetilde{u}, \widetilde{v})$ is in E_∞ if and only if $\widetilde{v}$ is a sub-tree rooted at one of the children of $\widetilde{u}$'s root. Since $V'_\infty = V_\infty$, the same edge is also in E'_∞, and vice versa, every edge in E'_∞ is also in E_∞. Therefore, it also holds that $E'_\infty = E_\infty$. Finally, observe that $\ell_\infty(\widetilde{w})$ and $\ell'_\infty(\widetilde{w}')$ for any nodes $\widetilde{w} \in V_\infty$ and $\widetilde{w}' \in V'_\infty$, respectively, are completely determined by the marks attributed to the corresponding root vertices of $\widetilde{w}$ and $\widetilde{w}'$. We conclude that $(V_\infty, E_\infty, \ell_\infty) = (V'_\infty, E'_\infty, \ell'_\infty)$, i.e., $G_\infty = G'_\infty$. □

Note that the previous Lemma 3.4 does not hold for arbitrary graphs G, since, e.g., the uncolored 12-cycle has two distinct prime factors, namely, the 3-cycle and the 4-cycle. In prime 2-hop colored graphs however, based on the following key lemma, we can use $L_\infty(v)$ of node v in a prime 2-hop colored graph G as the *alias* of v in G.

Lemma 3.5. *Let $G = (V, E, \ell)$ be a prime 2-hop colored graph and consider some $u, v \in V$. Then, $u = v$ if and only if $L_\infty(u) = L_\infty(v)$.*

Proof. Let $G = (V, E, \ell)$ be a prime 2-hop colored graph and let $u, v \in V$ be two nodes in G. Since the "only-if" direction is true by the assumption $u = v$, we only need to show that $L_\infty(u) = L_\infty(v)$ implies $u = v$. To that end, consider the infinite view graph $G_\infty = (V_\infty, E_\infty, \ell_\infty)$ of G and assume

for the sake of contradiction that $L_\infty(u) = L_\infty(v)$ in G but $u \neq v$. By the definition of G_∞, this implies that $|V_\infty| < |V|$ and thus, Lemma 3.3 guarantees that G admits a non-trivial factor, in contradiction to the assumption that G is prime. □

Establishing the Output's Validity

Getting back to $\mathcal{A}_\infty$, we have shown that the graph I^c_∞ constructed by $\mathcal{A}_\infty$ (independently, at every node v) satisfies $I^c_\infty \preceq_{f_\infty} I^c$. (Recall that $I^c = (V, E, i, c)$ is an input instance of Π's 2-hop colored variant Π^c and that $I = (V, E, i)$ is the corresponding input instance of Π.) Algorithm $\mathcal{A}_\infty$ simulates algorithm $\mathcal{A}_{\mathrm{R}}$ on the input $J = (V_\infty, E_\infty, i_\infty)$. Since $I^c_\infty \preceq_{f_\infty} I^c$, the input instance I satisfies $J \preceq_{f_\infty} I$ with the same factorizing map f_∞.

We argue that J is an input instance of Π. Indeed, as Π is genuinely solvable, there exists a randomized anonymous algorithm $\mathcal{B}$ that decides whether a given labeled graph is an input instance of Π. By the lifting lemma, $\mathcal{B}$ cannot distinguish J from I (cf. [7, 34]), hence the fact that $I \in \Pi$ implies that $J \in \Pi$, as required.

It follows that $\mathcal{A}_{\mathrm{R}}$ returns a correct output when executed on J, thus Lemma 3.2 ensures that the same successful simulation σ_∞ is selected by every node. Employing the lifting lemma once more, we conclude that the simulation obtained by lifting σ_∞ from nodes in V_∞ to nodes in V corresponds to a possible execution η of $\mathcal{A}_{\mathrm{R}}$ on I (cf. [7, 34]). The output labeling that $\mathcal{A}_\infty$ produces for the input instance I^c is exactly the output labeling produced by η in I and since the latter is valid, the former must also be valid by the definition of problem Π^c, thus establishing Theorem 3.2.

Since the description of $\mathcal{A}_\infty$ involves trees of infinite depth, one may wonder whether it can be replaced by a "real" algorithm. This issue is addressed in the next section, where we also establish Theorem 3.1.

3.4 Dealing with (In)finity

Recall that under the infinity model introduced in Section 3.3, each node v in the graph essentially receives $L_\infty(v)$. This luxury, of course, cannot

be realized in a standard anonymous algorithm, where node v can only obtain $L_d(v)$ for finite values of d. Nevertheless, in this section we show how the algorithm presented in Section 3.3 can be adapted to finite depth local views. A key ingredient in this adaptation is the following theorem established by Norris [86].[3]

Theorem 3.3 (Norris [86])**.** *Let G be a labeled graph with n nodes. The local view $L_n(v)$ fully determines $L_\infty(v)$ for every node $v \in V$.*

Employing Lemma 3.5, we obtain the following corollary that facilitates the usage of depth n local views as aliases for the nodes in an n-node prime 2-hop colored graph (instead of the depth-infinity local views that were used in Section 3.3).

Corollary 3.6. *Let $G = (V, E, \ell)$ be an n-node prime 2-hop colored graph and consider some $u, v \in V$. Then, $u = v$ if and only if $L_n(u) = L_n(v)$.*

Consider a 2-hop colored graph $G = (V, E, \ell)$ and denote by $n = |V_\infty|$ the number of different depth-infinity local views in G. For a node $v \in V$, denote by $\overset{*}{v} = L_n(v)$ the depth-n local view in G. The graph $G_* = (V_*, E_*, \ell_*)$, where $V_* = \{\overset{*}{v} : v \in V\}$, $E_* = \{(\overset{*}{u}, \overset{*}{v}) : (u, v) \in E\}$, and $\ell_*(\overset{*}{v}) = \ell(v)$ is called the *finite view graph* of G. The following corollary is established due to Theorem 3.3, where f_n is the *depth-n truncating function* that truncates every depth-k local view, $k \geq n$, to depth n, i.e., $f_n(\overset{\infty}{u}) = \overset{*}{v}$.

Corollary 3.7. *For a 2-hop colored graph G, it holds that $G_* \cong_{f_n} G_\infty$.*

The graph G_* can thus serve as a canonical representative for its equivalence class under the equivalence relation $\cong$. This is crucial because in contrast to G_∞, the graph G_* has a finite bitstring representation. Furthermore, each node v in a 2-hop colored graph G can identify its corresponding node in G_*, within $n = |V_*|$ rounds of the execution.

[3] The result in [86] is described in terms of the depth $n-1$ sub-trees of a graph's universal cover. To prove the corresponding statement for depth-n local views the same refinement argument can be made.

3.4.1 Algorithm $\mathcal{A}_*$

Fix some problem $\Pi \in$ GRAN and randomized anonymous algorithm $\mathcal{A}_{\mathrm{R}}$ that solves Π. Let Π^c be the 2-hop colored variant of Π and let $I^c = (V, E, i, c) \in \Pi^c$ be an arbitrary input instance of Π^c. To establish Theorem 3.1, we present a deterministic algorithm $\mathcal{A}_*$ that solves Π^c. Algorithm $\mathcal{A}_*$ resembles algorithm $\mathcal{A}_\infty$ presented in Section 3.3, however, the graph I^c_∞ is replaced by its finite representation I^c_* utilizing Corollary 3.7.

Algorithm $\mathcal{A}_*$, described from the perspective of an arbitrary node $v \in V$, proceeds in phases indexed by the positive integers. Throughout the execution of $\mathcal{A}_*$, node v keeps track of an initially empty bitstring $b(v)$, where the value of $b(v)$ for phase $p+1$ is determined during phase p. It will be convenient to denote by b^p the labeling function derived from the values $b(v)$ in phase p by setting $b^p(v) = b(v)$ for phase p. Correspondingly, we denote by $I^p = (V, E, i, c, b^p)$ the graph obtained by augmenting I^c with the labeling b^p. In each phase p, every node v invokes the three sub-procedures `Update-Graph`, `Update-Output`, and `Update-Bits` in this order. We now describe each sub-procedure individually. For convenience, we also include a pseudo-code style description of $\mathcal{A}_*$ in Figure 3.3.

`Update-Graph`. We say that a labeled graph $\hat{G} = (\hat{V}, \hat{E}, \hat{i}, \hat{c}, \hat{b})$ is a *candidate* for phase p if it satisfies the following three conditions:

C1. $|\hat{V}| \leq p$;
C2. there exists a node $\hat{v} \in \hat{V}$ such that $L_p(\hat{v}, \hat{G}) = L_p(v, I^p)$; and
C3. $(\hat{V}, \hat{E}, \hat{i}, \hat{c})$ is an input instance of Π^c.

Denote by $\mathcal{F}$ the set containing the finite view graphs of all candidates for phase p. In phase p, node v computes $L_p(v, I^p)$ (requires p rounds) and based on that, constructs the set $\mathcal{F}$.

Note that the set $\mathcal{F}$ can be totally ordered in a predetermined way. To see this, observe that for any finite view graph $G_* = (V_*, E_*, \ell_*)$, the set V_* can be totally ordered in a predetermined way similarly to the order used in Section 3.3.1. This total order on V_* fully determines a representation of G_* as a finite bitstring $s = s(G_*)$ (encoding the ordinal number and label of every node as well as every edge in G_*). Given two finite view graphs $G_* = (V_*, E_*, \ell_*)$ and $G'_* = (V'_*, E'_*, \ell'_*)$, we write $G_* < G'_*$ if

either $|V_*| < |V'_*|$ or $|V_*| = |V'_*|$ and $s(G_*) < s(G'_*)$ lexicographically.

If the set $\mathcal{F}$ is empty in phase p, then node v skips the remainder of this phase. Otherwise, `Update-Graph` selects the smallest finite view graph $\hat{G}_* = (\hat{V}_*, \hat{E}_*, \hat{i}_*, \hat{c}_*, \hat{b}_*) \in \mathcal{F}$. Let $\hat{G}$ be a candidate that corresponds to $\hat{G}_*$ and recall that condition C2. guarantees the existence of a node $\hat{v} \in \hat{V}$ such that $L_p(\hat{v}, \hat{G}) = L_p(v, I^p)$. Let $\overset{*}{v} = L_q(\hat{v}, \hat{G})$, where $q = |\hat{V}_*|$, be the node in $\hat{V}_*$ that corresponds to v.

`Update-Output`. Node v simulates $\mathcal{A}_{\mathrm{R}}$ on the instance $J = (\hat{V}_*, \hat{E}_*, \hat{i}_*)$ using the bitstrings provided by $\hat{b}_*$ as a replacement for $\mathcal{A}_{\mathrm{R}}$'s random bits. Recall that $\hat{b}_*$ corresponds to the bitstring assignment $\hat{b}$ in some candidate $\hat{G}$ and as such, reflects the bitstring assignment b^p of I^p. Since b^p may assign bitstrings of varying lengths to the nodes in V, $\hat{b}_*$ may also assign bitstrings of varying lengths to the nodes in $\hat{V}_*$. Therefore, the simulation of $\mathcal{A}_{\mathrm{R}}$ on J, denoted by σ, lasts for l rounds, where $l = \min\{\mathrm{length}(\hat{b}_*(\overset{*}{u})) : \overset{*}{u} \in \hat{V}_*\}$ is the length of the shortest bitstring assigned under $\hat{b}_*$ to the nodes in $\hat{V}_*$. If the simulation σ is successful (recall the definition of a successful simulation in Section 3.3.2), then `Update-Output` sets v's output to the value returned by node $\overset{*}{v}$ in σ.

`Update-Bits`. The task of `Update-Bits` is to update the value of $b(v)$, extending it to a bitstring of length p. An assignment $\hat{b}'_* : \hat{V}_* \to \{0,1\}^p$ is said to be a *p-extension* of $\hat{b}_*$ if the bitstring $\hat{b}_*(\overset{*}{u})$ is a prefix of $\hat{b}'_*(\overset{*}{u})$ for every node $\overset{*}{u} \in \hat{V}_*$. Let $\mathcal{B}$ be the set of p-extensions of $\hat{b}_*$ that induce successful simulations of $\mathcal{A}_{\mathrm{R}}$ on $J = (\hat{V}_*, \hat{E}_*, \hat{i}_*)$.

If $\mathcal{B}$ is empty, then $b(v)$ remains unchanged. Otherwise, the predetermined total order on $\hat{V}_*$ implies a predetermined total order on $\mathcal{B}$ — let $b_{\min}$ be the smallest bitstring assignment in $\mathcal{B}$ according to this total order and update the bitsring $b(v)$ so that $b(v) \leftarrow b_{\min}(\overset{*}{v})$.

3.4.2 Analysis

In our effort to prove Theorem 3.1, we need to show two things: (1) algorithm $\mathcal{A}_*$ terminates; and (2) the output produced by $\mathcal{A}_*$ is valid. We use the same notation as in Section 3.4.1 to denote the various graphs involved with $\mathcal{A}_*$. Recall that the description of $\mathcal{A}_*$ in Section 3.4.1 is provided

from the perspective of node v in phase p and when necessary, we explicitly mention p and/or v by adding them as a superscript. Specifically, let

- $I_*^p = (V_*^p, E_*^p, i_*^p, c_*^p, b_*^p)$ be the finite view graph of I^p;
- $\mathcal{F}^{p,v}$ be the set of finite view graphs from `Update-Graph`;
- $\hat{G}_*^{p,v} = (\hat{V}_*^{p,v}, \hat{E}_*^{p,v}, \hat{i}_*^{p,v}, \hat{c}_*^{p,v}, \hat{b}_*^{p,v}) \in \mathcal{F}^{p,v}$ be the finite view graph selected by `Update-Graph`;
- $\overset{*}{v}^p$ be the node in $\hat{V}_*^{p,v}$ corresponding to v;
- $J^{p,v} = (\hat{V}_*^{p,v}, \hat{E}_*^{p,v}, \hat{i}_*^{p,v})$ be the graph used to simulate $\mathcal{A}_{\mathrm{R}}$ in `Update-Output`; and
- $\sigma^{p,v}$ be the corresponding simulation of $\mathcal{A}_{\mathrm{R}}$ on $J^{p,v}$ induced by $\hat{b}_*^{p,v}$.

We further denote by $I_*^c = (V_*, E_*, i_*, c_*)$ the finite view graph of I^c and set $n = |V_*|$.

Termination. Recall the node $\hat{v}$ in a candidate $\hat{G}$ promised by condition C2.; we henceforth also refer to the node $\overset{*}{v}$ in the finite view graph of $\hat{G}$ that corresponds to $\hat{v}$ as being *promised* by condition C2.. Our analysis of $\mathcal{A}_*$ begins with the following insight regarding the graphs in $\mathcal{F}^{p,v}$, which follows from Corollary 3.6 as $|\hat{V}_*^{p,v}| \leq p$.

Corollary 3.8. *Consider some $\hat{G}'_* = (\hat{V}'_*, \hat{E}'_*, \hat{i}'_*, \hat{c}'_*, \hat{b}'_*) \in \mathcal{F}^{p,v}$ and let $\overset{*}{\hat{v}} \in \hat{V}'_*$ be the node promised by condition C2.. Then, $L_p(\overset{*}{\hat{v}}, \hat{G}_*) = L_p(v, I^p)$.*

Denote by $\hat{H}_*^{p,v} = (\hat{V}_*^{p,v}, \hat{E}_*^{p,v}, \hat{i}_*^{p,v}, \hat{c}_*^{p,v})$ the graph obtained from $\hat{G}_*^{p,v}$ by ignoring the labeling $\hat{b}_*^{p,v}$. To establish that $\mathcal{A}_*$ terminates, we show that the graph $\hat{H}_*^{p,v}$ "converges" towards I_*^c as cast in the following lemma.

Lemma 3.9. *There exists some q such that for every phase $p \geq q$, the graph $\hat{H}_*^{p,v}$ satisfies $\hat{H}_*^{p,v} \cong I_*^c$ for all nodes $v \in V$.*

The difficulty in proving Lemma 3.9 is that the finite view graphs $\hat{G}_*^{p,v}$ are constructed based on the local views in (V, E, i, c, b^p) rather than (V, E, i, c) — in particular, the labels $b^p(v)$ are constantly changing. Observe however that in $\mathcal{A}_*$, the value $b^{p+1}(v)$ depends solely on $L_p(v, I^p)$. This means that for every two nodes $u, v \in V$ and phase p, if $L_p(u, I^p) = L_p(v, I^p)$, then $b^{p+1}(u) = b^{p+1}(v)$. By induction on p, we conclude that $L_\infty(u, I^p) = L_\infty(v, I^p)$ if and only if $L_\infty(u, I^c) = L_\infty(v, I^c)$.

In other words, nodes indistinguishable without the labeling b^p are also indistinguishable when the labeling b^p is included. This immediately implies that in every phase p, the graph obtained from I^p_∞ by ignoring the labeling b^p is isomorphic to I^c_∞, which derives the following observation due to Corollary 3.7.

Observation 3.10. *For every phase p under algorithm $\mathcal{A}_*$, it holds that*

$$(V^p_*, E^p_*, i^p_*, c^p_*) \cong I^c_* \,.$$

Utilizing Observation 3.10, Lemma 3.9 can be established by showing that $\hat{G}^{p,v}_* = I^p_*$. The following Lemma 3.11 assures that I^p_* is among the candidates in all phases $p \geq n$.

Lemma 3.11. *If $p \geq n$, then the set $\mathcal{F}^{p,v}$ contains I^p_* for every node $v \in V$.*

Proof. We establish the assertion by showing that I^p_* (or a graph isomorphic to it) is a candidate for phase p, noticing that $(I^p_*)_* = I^p_*$. By Observation 3.10, we conclude that $|V^p_*| = |V^c_*| = n \leq p$, thus condition C1. holds. Since I^p_* is a factor of I^p, Fact 1 guarantees that node $\overset{*}{v} = L_n(v, I^p) \in V^p_*$ satisfies $L_p(\overset{*}{v}, I^p_*) = L_p(v, I^p)$, thus condition C2. holds as well. As already argued in Section 3.3.3, the lifting lemma guarantees that I^c_∞ is an instance of Π^c, therefore by Corollary 3.7, so is I^p_*, implying that condition C3. holds which completes the proof. □

Lemma 3.11 confirms that I^p_* *may* be selected by `Update-Graph` if $p \geq n$. It remains to show that there is a phase p in which I^p_* *will* be selected by `Update-Graph`. The following Lemma 3.12 confirms that the latter occurs if $p \geq 2n$, thus establishing Lemma 3.9.

Lemma 3.12. *If $p \geq 2n$, then $\hat{G}^{p,v}_* = I^p_*$ for all nodes $v \in V$.*

Proof. Lemma 3.11 guarantees that $I^p_* \in \mathcal{F}^{p,v}$ for every node $v \in V$. The assertion is established by showing that I^p_* is the smallest graph in $\mathcal{F}^{p,v}$ according to the total order used in `Update-Graph`.

Let $\hat{G}'_* = (\hat{V}'_*, \hat{E}'_*, \hat{i}'_*, \hat{c}'_*, \hat{b}'_*) \in \mathcal{F}^{p,v}$ be the finite view graph of some candidate $\hat{G}'$ and set $n' = |\hat{V}_*|$. Assume for the sake of contradiction that $\hat{G}_* < I^p_*$. This implies that either (1) $n' < n$; or (2) $n' = n$ and

$s(\hat{G}'_*) < s(I^p_*)$. We will show that neither (1) nor (2) hold and thus, contradict $\hat{G}'_* < I^p_*$.

To that end, let $\overset{*}{v}' \in \hat{V}'_*$ and $\overset{*}{v} \in V^p_*$ be the nodes in $\hat{G}'_*$ and I^p_*, respectively, promised by property C2.. Since $|V^p_*| = n$ and $|V'_*| = n' \leq n$, it follows that the diameter of both I^p_* and $\hat{G}'_*$ is at most $n-1$. Since $p \geq 2n$, the local view $L_p(\overset{*}{v}', \hat{G}'_*)$ contains the sub-tree $\overset{*}{u}'$ for every $u' \in \hat{V}'_*$ and each distinct depth-n' sub-tree of $L_p(\overset{*}{v}', \hat{G}'_*)$ corresponds to a different node $\overset{*}{u}' \in \hat{V}'_*$. Similarly, the local view $L_p(\overset{*}{v}, I^p_*)$ contains the sub-tree $\overset{*}{u}$ for every $u \in V^p_*$ and each distinct depth-n sub-tree of $L_p(\overset{*}{v}, I^p_*)$ corresponds to a node $\overset{*}{u} \in V^p_*$.

Corollary 3.8 guarantees that $L_p(\overset{*}{v}', \hat{G}'_*) = L_p(v, I^p) = L_p(\overset{*}{v}, I^p_*)$. Therefore, the depth-$n'$ truncating function $f_{n'}$ maps every node $\overset{*}{u} \in V^p_*$ to a node in $\hat{V}'_*$. It follows by the definition of local view that $\hat{G}'_* \preceq_{f_{n'}} I^p_*$.

If $n' = n$, then $f_{n'}$ is the identity function, hence $\hat{G}'_* = I^p_*$, in contradiction to the assumption that $\hat{G}'_* < I^p_*$. If on the other hand $n' < n$, then $\hat{G}'_*$ is a non-trivial factor of I^p_*, contradicting the fact that the finite view graph I^p_* is prime. The assertion follows. □

Consider some node $v \in V$ and phase $p \geq 2n$. Lemma 3.12 guarantees that the graph $J^{p,v}$ on top of which the simulation $\sigma^{p,v}$ is carried out is in fact (V_*, E_*, i_*) — denote this graph by $\widetilde{J}$. The design of `Update-Graph` ensures that $\widetilde{J} \in I$ and the reasoning from Section 3.3.2 can be applied to show that all nodes $u \in V$ perform the same simulation $\sigma^{p,u}$ on $\widetilde{J}$ — denote this simulation by σ^p and let $b^p : V_* \to \{0,1\}^*$ be the bitstring assignment that induces σ^p.

Let z be the smallest integer $z \geq 2n$ so that there exists a z-extension of b^{2n} that induces a successful simulation on $\widetilde{J}$ and let b' be the smallest such z-extension according to the predetermined total order on the bitstring assignments. Notice that the integer z is well defined since $\mathcal{A}_{\mathrm{R}}$ is guaranteed to produce a correct output with probability 1. The design of `Update-Bits` ensures that in phase z, every node $u \in V$ updates $b(u) \leftarrow b'(\overset{*}{u})$, where $\overset{*}{u}$ is the node in V_* that corresponds to u. The design of `Update-Output` then ensures that in phase $z+1$, all nodes set their outputs according to the successful simulation σ^{z+1}, thus establishing the termination of $\mathcal{A}_*$ as cast in the following lemma.

Lemma 3.13. *In phase $z+1$, all nodes $v \in V$ set their outputs $\mathcal{A}_*(v)$.*

Correctness. It remains to show that the output produced by $\mathcal{A}_*$ is correct. Denote by $o^p(v)$ the output of node $v \in V$ in phase p of $\mathcal{A}_*$, where we use the designated symbol ε to indicate that v does not return any output in phase p, writing $o^p(v) = \varepsilon$. Intuitively, we establish the correctness of $\mathcal{A}_*$ by arguing that there exists an execution η of $\mathcal{A}_{\mathrm{R}}$ on $I = (V, E, i)$ such that for every node $v \in V$ and integer $1 \leq p \leq z+1$, if $o^p(v) \neq \varepsilon$, then the output of node v in *round* p under η, denoted by $o^p_\eta(v)$, is $o^p_\eta(v) = o^p(v)$. (In fact, the execution η is obtained by lifting σ^{z+1} from V_* to V.) The correctness of $\mathcal{A}_*$ then follows from the correctness of $\mathcal{A}_{\mathrm{R}}$.

Lemma 3.14. *For any phase p, if node v returns an output $o^p(v) \neq \varepsilon$ in phase p, then $o^p(v) = o^p_\eta(v)$.*

Proof. Let p be a phase, and suppose that node $v \in V$ sets its output in phase p to $o^p(v) \neq \varepsilon$. Since v sets $o^p(v)$ in phase p, the graph $\hat{G}^{p,v}_*$ is defined and the simulation $\sigma^{p,v}$ of the randomized algorithm $\mathcal{A}_{\mathrm{R}}$ is successful. With that in mind, let t be the length of $\sigma^{p,v}$, and let η be the execution of $\mathcal{A}_{\mathrm{R}}$ on $I = (V, E, i)$ obtained by lifting σ^{z+1} from V_* to V. The goal now is to show that $o^p(v) = o^p_\eta(v)$.

Note that for any $k > 0$, the first k rounds in η are fully determined by the first k random bits of each node $u \in V$ and their respective input values. More specifically, the first k rounds in η for a *single* node v are fully determined by $L_k(v, I)$ and the first $k-i$ bits replacing the random bits of each node $u \in \mathcal{H}^i(v)$, $0 \leq i \leq k$, where $\mathcal{H}^i(v)$ the set of all nodes at most i hops away from v, i.e, $\mathcal{H}^0(v) = \{v\}$ and $\mathcal{H}^{i+1}(v) = \mathcal{H}^i(v) \cup \Gamma(\mathcal{H}^i(v))$ for every $i \geq 0$.

The construction of $J^{p,v}$ ensures that $L_p(\overset{*}{v}{}^p, J^{p,v}) = L_p(v, I)$. Moreover, for every $u \in \mathcal{H}^p(v)$, there exists a node $\overset{*}{u} \in \hat{V}^{p,v}_*$ such that the bitstring assigned to $\overset{*}{u}$ satisfies $\hat{b}^{p,v}_*(\overset{*}{u}) = b^p(u)$. Since v sets $o^p(v)$ in `Update-Output`, the simulation $\sigma^{p,v}$ is successful and thus, the length of $\hat{b}^{p,v}_*(\overset{*}{u})$ is at least t. The design of `Update-Bits` ensures that $b^p(u)$ is a prefix of $b^{p+1}(u)$ for every phase p and node $u \in V$. Therefore, the first $t-i$ bits assigned to node $u \in \mathcal{H}^i(v)$, $0 \leq i \leq t$ are the same $t-i$ bits that are used in the first t rounds of η. We conclude that in η node v returns the output $o^t_\eta(v) = o^p(v)$ at node v in round p. □

Lemma 3.14 is sufficient to establish the correctness of $\mathcal{A}_*$ as well: Us-

ing the same line of arguments as in Section 3.3.3, once more invoking the lifting lemma, one can show that the output obtained by lifting the output of simulation σ^{z+1} is valid for I (and I^c). By combining Lemmas 3.13 and 3.14, Theorem 3.1 now follows.

3.5 Fibrations and 2-Hop Colorings

Boldi and Vigna [35] extensively study the notion of fibrations, roughly speaking a generalization of factorizing maps to edge-colored directed graphs (refer to [35] for an exact definition). A special case the two authors study are *deterministic (edge) colorings* which require that for every node, all out-edges must be colored differently. In this section we wish to highlight a connection between our observations regarding 2-hop colored graphs in Section 3.3 and deterministically edge colored directed graphs. In the following, we write *undirected* as well as *directed* edges as tuples (u, v). It will be clear from the context whether we are referring to a directed or an undirected edge.

Let $G = (V, E, c)$ be a 2-hop colored undirected graph, and consider the edge colored directed graph $H = (V', E', c')$ obtained by (1) choosing $V' = V$; (2) adding two directed edges (u, v) and (v, u) to E' for every undirected edge $(u, v) \in E$; and (3) setting $c'(e) = \langle c(u), c(v) \rangle$ for every directed edge $e = (u, v) \in E'$. In the terminology of [35], the graph H is symmetric, since for every edge (u, v) a symmetric edge (v, u) is present. Moreover the edge coloring c' is deterministic, and c' respects the edge symmetries since for every edge $e = (u, v)$, colored $\langle c_1, c_2 \rangle$, the symmetric edge $\bar{e} = (v, u)$ is colored $\langle c_2, c_1 \rangle$. We call the graph H obtained from G in this manner as being G's *directed (edge colored) representation.* Note that reversing the construction, in hope to obtain a 2-hop colored graph, is not possible for general deterministically edge-colored symmetric directed graphs.

Observe that a fibration $\varphi : H \to H'$, where H and H' are directed representations of two graphs G and G', translates to a factorizing map $f : G \to G'$ as defined in Section 3.3, and vice versa. One can use this connection to derive the statements from Section 3.3 from the results presented in [35].

Algorithm: $\mathcal{A}_*$, a deterministic algorithm solving Π^c at node v
  ▷ *Initialization of variables for node v:*
  $(\hat{V}_*, \hat{E}_*, \hat{\imath}_*, \hat{c}_*, \hat{b}_*) \leftarrow$ the empty labeled graph.
  $\overset{*}{v} \leftarrow$ NULL
  $b(v) \leftarrow$ the empty bitstring
  for *phase* $p \leftarrow 1, \ldots$ **do**
    `Update-Graph`(p)
    `Update-Output`(p)
    `Update-Bits`(p)
  end
end
Procedure `Update-Graph`(p)**:**
  $L \leftarrow L_p(v, I^p)$ ▷ $b(v)$ *is treated as a labeling function*
  construct the set $\mathcal{F}$ of candidates for phase p
  if $\mathcal{F} = \emptyset$ **then**
    skip phase p
  end
  else
    $\hat{G}_* = (\hat{V}_*, \hat{E}_*, \hat{\imath}_*, \hat{c}_*, \hat{b}_*) \leftarrow$ the smallest graph in $\mathcal{F}$
    $\hat{G} = (\hat{V}, \hat{E}, \hat{\imath}, \hat{c}, \hat{b}) \leftarrow$ a candidate that corresponds to $\hat{G}_*$
    $\hat{v} \leftarrow$ the node $\hat{v} \in \hat{V}$ such that $L_p(\hat{v}, \hat{G}) = L_p(v, I^p)$ as assured by C2.
    $q \leftarrow |\hat{V}_*|$
    $\overset{*}{v} \leftarrow$ the node $\overset{*}{v} = L_q(\hat{v}, \hat{G}) \in \hat{V}_*$ with $L_p(\hat{v}, \hat{G}) = L$
  end
end
Procedure `Update-Output` (p)**:**
  $\sigma \leftarrow$ the simulation of $\mathcal{A}_{\mathrm{R}}$ on $(\hat{V}_*, \hat{E}_*, \hat{\imath}_*)$ induced by $\hat{b}_*$
  if σ *is successful* **then set** $o(v)$ to be the output of $\overset{*}{v}$ in σ
end
Procedure `Update-Bits` (p)**:**
  $\mathcal{B} \leftarrow \{b : b$ is a p-extension of $\hat{b}_*$, and the simulation of $\mathcal{A}_{\mathrm{R}}$ on $(\hat{V}_*, \hat{E}_*, \hat{\imath}_*)$ induced by b is successful$\}$
  if $\mathcal{B} \neq \emptyset$ **then**
    $b_{\min} \leftarrow$ the smallest $b \in \mathcal{B}$
    $b(v) \leftarrow b_{\min}(\overset{*}{v})$
  end
end

Figure 3.3: The deterministic algorithm $\mathcal{A}_*$ that solves Π^c.

4

The Cost of Randomness

We have seen that in an anonymous network, symmetry breaking tasks can only be solved if randomization is available. But how many random bits are required to solve any such task? As it turns out, the answer to this question depends on the desired runtime of the algorithm.

Consider, for example, the fundamental symmetry breaking problem of graph coloring, where the goal is to assign colors to nodes so that every two neighbors get a different color. In a complete network, i.e., when every node is connected to all other nodes, a unique color must be used for every node. Therefore, for complete networks the answer is at least $\log n$ random bits. One result of our work is that in expectation $\Omega(\log n)$ random bits are required even if every node in the network has at most 3 neighbors. Moreover, we establish that $\mathcal{O}(\log n)$ random bits in expectation are also sufficient to solve all tasks in any network.

Alongside this *random bit complexity*, we consider the *runtime* required

to solve such tasks. Increasing the runtime allows one to draw the random bits more carefully, thus reducing the number of unnecessarily drawn random bits. Conversely, drawing random bits more generously enables faster runtime. We study how exactly the random bit complexity relates to the runtime.

More precisely, we show that there is an *efficiency trade-off* between the runtime and the random bit complexity required to solve any task. Our contribution is to establish asymptotically tight lower and upper bounds on the achievable trade-off. Those bounds imply that using more than $\mathcal{O}(\log\log n)$ rounds to solve a task does not result in a better random bit complexity. Since Linial showed that local symmetry breaking requires roughly $\log^* n$ rounds [76], we obtain that the interesting cases occur when the asymptotic runtime is between $\log^* n$ and $\log\log n$. In the respective extreme cases, i.e., when the runtime is $\log^* n$ or $\log\log n$, our lower bound states that the random bit complexity is $\Omega(\sqrt[d]{n})$ and $\Omega(\log n)$, correspondingly, where d is a constant that depends on the runtime.

For the upper bound we devise a randomized scheme that produces sufficiently many random bits for any anonymous network algorithm. To this end we introduce the notion of a *target function* f which specifies the desired runtime of our scheme, and consider the cases where $f(n)$ is asymptotically between $\log^* n$ and $\log\log n$. The trade-off achieved by our scheme asymptotically matches the lower bound with high probability[1] and in expectation, also for all runtimes f that lie between the two extremes.

Our scheme is *uniform*: The algorithm does not require any knowledge about the network topology, such as its size or diameter. As such, it can be used to devise new uniform algorithms for classic symmetry breaking problems by utilizing existing deterministic algorithms. This is due to the fact that those algorithms often assume IDs, but function correctly even if those IDs are only *locally* unique. As one example, consider the deterministic coloring algorithm from [91] which runs in $\mathcal{O}(\log^* n)$ time on graphs with bounded growth. By applying our scheme, we obtain a uniform coloring algorithm for anonymous networks with the same runtime. In light of Linial's lower bound [76] the $\mathcal{O}(\log^* n)$ runtime is asymptoti-

[1] We say an event occurs *with high probability* (w.h.p.) if it occurs with probability $1 - n^{-c}$ for any constant c.

cally optimal. This speed comes at the cost of a relatively high random bit complexity, which is $\Theta(\sqrt[d]{n})$. Note, however, that d is a freely selectable parameter of our scheme that turns into a constant factor of the runtime hidden in the big-$\mathcal{O}$ notation. If one is willing to sacrifice the asymptotic runtime, on the other end of the spectrum, our approach allows to solve the same task in $\mathcal{O}(\log \log n)$ time using as little as $\mathcal{O}(\log n)$ random bits. By tuning the f parameter, any trade-off between the two extremes can be achieved.

So how can we possibly bound the random bit complexity for any computable task? The answer to this *complexity* question can be based on the *computability* result presented in Chapter 3, where we saw that a 2-hop coloringis necessary and sufficient to replace access to random bits in any anonymous network algorithm. We therefore establish our upper bound by devising a 2-hop coloring algorithm whose runtime and random bit complexity are tuneable by a target function f and a number d.

4.1 Broadcast Model and Target Functions

Broadcast Algorithms. We consider randomized algorithms that always return a correct output and have finite expected runtime (Las Vegas algorithms). In contrast to before, our algorithms run under the synchronous broadcast model, i.e., in every round, each node u sends the same finite length (broadcast) message to all nodes in $\Gamma(u)$. Moreover, we assume that the source of random bits for node u is independent from the source of random bits for any other node $v \in V$. Recall that $\mathcal{A}$ is called *deterministic* if $\mathcal{A}$ does not draw any random bits. Like before, we restrict ourselves to *uniform* algorithms, i.e., the nodes are unaware of any network parameter, e.g., the network size n, nor do they have unique identifiers (the network is *anonymous*).

We consider two complexity measures of an algorithm $\mathcal{A}$. (1) The *runtime* of $\mathcal{A}$ in some graph G is the number of rounds that are executed until all nodes terminate, and (2) the *random bit complexity* of $\mathcal{A}$ is the maximum number of random bits drawn by any node during the execution of $\mathcal{A}$.

The Target Function $f(n)$. A function f is called a *target function* if f is positive, strictly increasing, and continuous. Note that the properties of a target function f ensure that the *inverse target function* $f^{-1}(n)$ of $f(n)$ is well-defined. For easier readability, we denote the inverse function by $g_f(n) = f^{-1}(n)$, or $g(n)$ if f is clear from the context.

The purpose of a target function is to capture the runtime of some deterministic algorithm $\mathcal{A}$. The runtime $f_*(n)$ of $\mathcal{A}$ is positive, but not necessarily strictly increasing in the input size n, nor continuous. However, for any $\varepsilon > 0$, there is a target function f such that $f_*(n) \leq f(n) \leq f_*(n)+\varepsilon$, i.e., f "captures" f_* at all integer values $n \geq 1$.

4.2 Related Work

The theory of distributed computability began with Angluin's insight that leader election is impossible in anonymous rings [7]. A similar impossibility argument can be made for deterministic algorithms that solve local symmetry breaking tasks, e.g., coloring or MIS, and literally hundreds of more impossibilities are known [19]. In short, the computational power of deterministic anonymous network algorithms is limited [84].

Under the assumption of uniform algorithms, the leader election impossibility result from [7] extends to the case where randomization is available. In contrast to that, when randomization is available, there are well known algorithms that solve the local symmetry breaking problems coloring [76] and MIS [5, 78] also in anonymous networks. It is interesting to note that both randomized MIS algorithms are used to construct completely derandomized (deterministic) variants under the assumption that unique identifiers are available. How much randomization an anonymous network will ever need from a *computability* perspective can be characterized in terms of a 2-hop coloring [46] (presented in Chapter 3). In this chapter, based on that observation, we tackle the *complexity* question, i.e., the random bits and runtime necessary to obtain a 2-hop coloring. Also outside of anonymous algorithms, randomization has many applications in distributed computing (cf. [20]), e.g., in agreement [17, 18], self stabilization [45], and non-uniform leader election [5].

Still, one of the most basic tasks to solve in a distributed setting remains coloring, and often coloring and MIS algorithms go hand in hand.

As such, they were studied thoroughly (please refer to [26] for an extensive overview), usually aiming to use at most $\Delta + 1$ (or at least some small function of Δ) many colors. Perhaps surprisingly, when identifiers are available, deterministic coloring algorithms are among the fastest. A recent series of results by Barenboim, Elkin, and Kuhn [24, 27, 72] yields a $\Delta+1$ coloring in $\mathcal{O}(\Delta+\log^* n)$ runtime by utilizing a new defective coloring technique. The picture is completed by the observation that colors can be traded for runtime [25], i.e., one can get $\mathcal{O}(\Delta^\varepsilon+\log^* n)$ for $\mathcal{O}(\Delta)$ colors or $\mathcal{O}(\log\Delta\cdot\log^* n)$ for $\mathcal{O}(\Delta^{1+\varepsilon})$ colors. These deterministic coloring algorithms have in common that they need to assume IDs. Also randomized algorithms (e.g. [90, 91]) often assume IDs and are not uniform, i.e., they assume knowledge about n or some other global network parameter. Relieving the algorithm from that knowledge, we focus on achieving a good random bit complexity instead of low number of used colors, and refer to standard methods (e.g., [59]) to reduce this number. On the other hand, the $\mathcal{O}(\log n)$ algorithms for MIS [5, 78] and coloring [76] are uniform, and can be formulated even in very restricted models [92]. We improve on the runtime at the lowest possible price one needs to pay for that in terms of random bit complexity.

It is worth mentioning that in the context of self-stabilization [41], uniform MIS and (2-hop) coloring protocols were studied also for anonymous networks. For instance, [93] considers deterministic and randomized protocols that color paths and rings, and later [61] obtain randomized protocols for MIS and coloring in arbitrary networks. The recent work [31] presents a 2-hop coloring protocol for graphs of bounded degree. In the self-stabilization context, the difficulty lies in dealing with faults. The random bit complexity is of no concern in the protocols mentioned above, and the runtime of [31] is necessarily much higher than in our non-faulty environment.

A concept related to that of randomization is non-determinism. The distributed notion of this concept, where often IDs are assumed, was initiated by Naor and Stockmeyer [84], who studied what could be *checked* by deterministic constant-time algorithms if some labeling (non-determinism) is known in advance. Subsequently, the number of non-deterministic choices required to solve decision problems in this distributed manner was investigated [68]. A hierarchy of decidable problems depending on

the necessary amount of non-determinism arises [60], also when the network is anonymous. Recently, it was found that in fact the *combination* of non-determinism with randomization allows distributed algorithms to decide any language in constant time [50].

4.3 Tailor-Made 2-Hop Coloring

Our technical contribution starts by presenting a 2-hop coloring algorithm, called Tailor-2-Hop-Coloring, with customizable runtime. Specifically, our algorithm is parametrized by a target function f and two integers $a > 2, d \geq 2$. As discussed before, we assume that $f(n)$ is between $\log^* n$ [76] and $\log \log n$ (see Section 4.4). Then, the algorithm finds a 2-hop coloring in $3d \cdot f(n)$ rounds in expectation and with probability $1 - n^{a-2}$.

The main difficulty is to choose how quickly random bits should be drawn, without knowledge of n. From the discussion above we know that in some round $3d \cdot f(n)$, we should have drawn at least $\Omega(\log n)$ bits. If we draw the bits too quickly, however, we might draw too many bits in the last round before the algorithm finishes. To deal with that, we design our *bit drawing function* $b(i)$ for the target function f and the integer parameters a and d as follows. Let i be some positive integer, and write $i = dp + s$ with $0 \leq s \leq d - 1$, i.e., $p = \lfloor i/d \rfloor$ and $s = i \pmod d$. The bit drawing function for i is defined as

$$b(i) = b(dp + s) = a \cdot \lceil \log g(p) \rceil^{(d-s)/d} \cdot \lceil \log g(p+1) \rceil^{s/d}.$$

We describe Tailor-2-Hop-Coloring from the perspective of node $u \in V$ (please refer to Algorithm 1 for a pseudo-code description). The algorithm progresses in phases p, starting from phase 1, and every phase consists of d sub-phases, which in turn consist of 3 rounds each.

Node u maintains a variable x storing all random bits drawn in the course of the execution. In the first sub-phase of each phase, u appends bits to x until the length of x is $b(dp)$. In the remaining $d - 1$ sub-phases $s = 1, \ldots, d - 1$ of phase p, by appending bits to x, the number of used random bits is increased to $b(dp + s)$. This process takes place in the first round of each sub-phase. After drawing bits in round 1 of sub-phase i, u sends its (preliminary) color x to all nodes $v \in \Gamma(u)$.

In the beginning of the second round of sub-phase i, node u receives the colors chosen by all nodes in $\Gamma(u)$. The list consisting of u's own color x and all the received colors is then sent to all neighbors of u. In the beginning of the third round of sub-phase i node u receives such a list from each neighbor. If x occurs only once in each list, then u selects color x and terminates. Otherwise, if x was used by multiple nodes, the process continues.

The idea behind TAILOR-2-HOP-COLORING is as follows. In the first sub-phase of each phase, every node u draws a random color x from the set of colors $\{1, \ldots, g(p)^a\}$. Our choice of b ensures that the remaining

Algorithm 1: TAILOR-2-HOP-COLORING(f, a, d) as executed by node u.

Initialization:
  $g(n) \leftarrow f^{-1}(n)$
  $x \leftarrow \varepsilon$ ▷ *the empty bit string*
Phase $p = 1, 2, \ldots$**:**
  For sub-phase $s = 0, 1, 2, \ldots, d-1$**:**
    ▷ *Round* 1 *of sub-phase* s*:*
    Append random bits to x until $|x| = b(pd + s)$
    Send x to all neighbors
    ▷ *Round* 2 *of sub-phase* s*:*
    Receive $x_1, \ldots, x_\delta$ from each non-terminated neighbor $v_1, \ldots, v_\delta \in \Gamma(u)$
    Send list $\langle x, x_1, \ldots, x_\delta \rangle$ to all neighbors
    ▷ *Round* 3 *of sub-phase* s*:*
    Receive lists $L_1, \ldots, L_\delta$ from each neighbor
    if *x appears exactly once in every list* **then**
      Choose color x and terminate
    end
  end
end

sub-phases of phase p are used to interpolate between $g(p)^a$ and $g(p+1)^a$ if the chosen colors are not a valid 2-hop coloring. The interpolation is performed so that within each phase p, the multiplicative increase in the number of random bits used in each sub-phase is fixed. If, for instance, Tailor-2-Hop-Coloring is in the first sub-phase of some phase $p = \lceil f(n) \rceil$, then the number of bits used by u is at least $a \log n$.

Please note that in round 3 of each sub-phase, a node chooses a color only if it does not violate the 2-hop coloring constraint. Thus, the output of Tailor-2-Hop-Coloring is always a valid 2-hop coloring. The remainder of this section is dedicated to establishing the following theorem.

Theorem 4.1. *The runtime of* Tailor-2-Hop-Coloring *with high probability and in expectation is* $\mathcal{O}(f(n))$ *rounds. The random bit complexity of* Tailor-2-Hop-Coloring *with high probability and in expectation is* $\mathcal{O}(h(f(n)) \cdot \log n)$ *bits, where*

$$h(i) = \sqrt[d]{\frac{\lceil \log g(i+1) \rceil}{\lceil \log g(i) \rceil}} .$$

It will sometimes be convenient to express the bit drawing function in terms of h:

$$b(pd+s) = b(dp) \cdot h(p)^s , \qquad \text{for } 0 \le s \le d, \text{ and} \tag{4.1}$$

$$b(pd+s+1) = b(dp+s) \cdot h(p) , \qquad \text{for } 0 \le s \le d . \tag{4.2}$$

Consider the last phase p and sub-phase s for which $b(pd+s) < a \log n$. In that case, $b(pd+s+1) \ge a \log n$ bits are drawn in the next step. Thus, due to the second expression, the essence of Theorem 4.1 is that Tailor-2-Hop-Coloring "overshoots" the necessary $a \log n$ bits by at most a factor of $h(p)$.

Recall that the target function f can be thought of as the runtime function of any deterministic algorithm that relies on a 2-hop coloring. Before getting into the details of the analysis, let us briefly put Theorem 4.1 into perspective by considering the corner cases where $f \in \Theta(\log \log n)$ or $f \in \Theta(\log^* n)$. In the former case $h(f(n))$ is in $\mathcal{O}(1)$, whereas in the latter case $h(f(n))$ is in $\mathcal{O}(\sqrt[d]{n})$. Thus, we obtain the following corollary from Theorem 4.1.

Corollary 4.1. *Consider a target function f, and let R denote the random bit complexity of* Tailor-2-Hop-Coloring.

(i) If $f(n) \in \Theta(\log^* n)$, then R is $\mathcal{O}(\sqrt[d]{n} \cdot \log n) \subseteq \mathcal{O}(\sqrt[d-1]{n})$ w.h.p. and in expectation.

(ii) If $f(n) \in \Theta(\log \log n)$, then R is $\mathcal{O}(\log n)$ w.h.p. and in expectation.

The analysis of Tailor-2-Hop-Coloring's runtime and random bit complexity are done separately. We first establish the high-probability results, beginning with the runtime.

Lemma 4.2. *Algorithm* Tailor-2-Hop-Coloring *terminates after at most $\mathcal{O}(f(n))$ rounds w.h.p.*

Proof. To establish Lemma 4.2 we will show that all nodes terminate in phase $p = \lceil f(n) \rceil$ with high probability. This is sufficient to establish the claim, since every phase consists of $3d$ rounds. In our proof, for any node z, we denote by x_z the random bit string stored in z's variable x after the first sub-phase of phase p.

First, consider some node $u \in V$ that did not terminate before phase p. By the definition of the bit drawing function b, it holds that $|x_u| \geq a \log g(p) \geq a \log n$. Node u terminates in round 3 of sub-phase 1 only if $x_u \neq x_v$ for all $v \in \Gamma^2(u)$. If v terminated in some phase $r < p$, then $|x_v| < |x_u|$ since v stopped drawing random bits in phase r, and in particular the probability that $x_u = x_v$ is 0. If on the other hand v still participates in phase p, then the probability that $x_u = x_v$ is at most $1/2^{a \log n} = 1/n^a$, since $|x_u| = |x_v| \geq a \log n$ in phase p.

Now consider the event T that all nodes terminate in phase p. The opposite event $\neg T$, i.e., the event that at least one node does not terminate in phase p, occurs if there are nodes $u, v \in V$ such that $x_u = x_v$ and $u \in \Gamma^2(v)$. By applying the union bound, we get

$$\Pr[\neg T] \leq \sum_{u,v \in V} \Pr[x_u = x_v] \leq \frac{1}{n^{a-2}} ,$$

so that T occurs with probability at least $(1 - 1/n^{a-2})$. For any constant c, we obtain that T occurs with probability $1 - n^{-c}$ by setting $a = c + 2$ in Tailor-2-Hop-Coloring. Recalling that each phase consists of $3d$ rounds, the claim follows. □

Lemma 4.3. *The random bit complexity of* Tailor-2-Hop-Coloring *is at most* $h(f(n)) \cdot a \log n$ *with high probability.*

Proof. We prove the statement by taking the exact sub-phase in which Tailor-2-Hop-Coloring terminates (w.h.p.) into account for our analysis. To that end, let p and s be a phase and a sub-phase, correspondingly, such that

$$(h(p))^s \cdot \log g(p) \leq \log n \leq (h(p))^{s+1} \cdot \log g(p)\,. \tag{4.3}$$

In sub-phase $s+1$, each non-terminated node draws at least $a(h(p))^{s+1} \cdot \log g(p) \geq a \log n$ bits and therefore, the probability that any two nodes draw the same bit string is bounded from above by $1/2^{a \log n} = 1/n^a$. Let $\neg T$ denote the event that at least one node u does not terminate in sub-phase $s+1$ of phase p. Event $\neg T$ occurs if there is at least one node in $v \in \Gamma^2(u)$ that has drawn the same bit sequence as u, and applying the union bound yields that

$$\Pr[\neg T] \leq \sum_{u,v \in V} \frac{1}{n^a} \leq \frac{1}{n^{a-2}}\,.$$

As the last step of the proof, we bound the number of bits $a \cdot g(p) \cdot (h(p))^{s+1}$ used in sub-phase $s+1$ of phase p. Since Equation (4.3) implies that $p \leq f(n)$, we obtain that with high probability the bit complexity is

$$a(h(p))^{s+1} \cdot \log g(p) \leq ah(p) \cdot \log n \leq ah(f(n)) \cdot \log n\,,$$

as desired. □

Next, we establish the results for the expected values.

Lemma 4.4. *The expected runtime of* Tailor-2-Hop-Coloring *is at most* $\mathcal{O}(f(n))$.

Proof. Let P be the random variable denoting the phase in which Tailor-2-Hop-Coloring terminates. From the definition of the expected value, for the expected runtime $\mathbf{E}[P]$ we obtain

$$\mathbf{E}[P] = \sum_{i=1}^{\infty} i \cdot \Pr[P = i] = \sum_{i=1}^{\infty} \Pr[P \geq i]\,,$$

where $\Pr[P \geq i]$ corresponds to the probability of proceeding to phase i. Furthermore, the algorithm only proceeds to phase $i+1$ in the case that there is a pair of nodes u and v within 2-hops distance that get assigned the same color in phase i. The number of bits used in sub-phase 0 of any phase p is $a\lceil \log g(p) \rceil$. Therefore, we can apply the union bound and get that

$$\Pr[P \geq i+1] \leq \sum_{u,v \in V} \frac{1}{2^{a\lceil \log g(i) \rceil}} \leq \sum_{u,v \in V} \frac{1}{2^{a \log g(i)}} \leq \frac{n^2}{(g(i))^a} \,.$$

Recalling that $g(f(n)) = n$ we get that

$$\Pr[P \geq f(n)+i] \leq \frac{n^2}{(g(f(n)+i))^c} \leq \frac{n^2}{n^a \cdot 2^i} \leq \frac{1}{2^i} \,,$$

given that $a > 2$. We set $\varphi = f(n)$ and divide the summation of the expected runtime into the part before and after φ. It follows that

$$\begin{aligned} \mathbf{E}[P] = \sum_{i=1}^{\infty} \Pr[P \geq i] &= \sum_{i=1}^{\varphi} \Pr[P \geq i] + \sum_{i=1}^{\infty} \Pr[P \geq \varphi + i] \\ &\leq \sum_{i=1}^{\varphi} 1 + \sum_{i=1}^{\infty} \frac{1}{2^i} \leq \varphi + 1 \end{aligned}$$

The claim follows since each phase consists of $3d$ rounds. □

Lemma 4.5. *If $f(n)$ is at least $\log^* n$, then the random bit complexity of* Tailor-2-Hop-Coloring *is $\mathcal{O}(h(f(n)) \cdot \log n)$ in expectation.*

The proof of Lemma 4.5, similar to that of Lemma 4.4, relies on carefully inspecting the round in which Tailor-2-Hop-Coloring terminates. However, due to the possibly large growth of g (which directly affects the growth of the bit drawing function), the analysis requires more attention. Instead of considering only the phase in which Tailor-2-Hop-Coloring terminates, we take the exact step in that phase into account. This yields a division of the expected value into 5 (instead of the previous 2) terms. Bounding each term individually leads to a rather lengthy proof, which is therefore deferred to the appendix. Theorem 4.1 is now established by combining Lemmas 4.2 to 4.5.

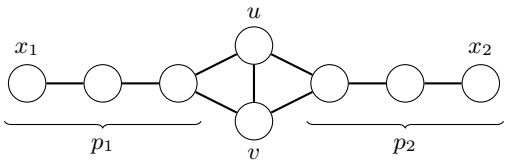

Figure 4.1: A (u, v)-gadget of length $i = 4$, consisting of $2i$ nodes: The two special nodes u and v, and the two paths p_1 and p_2 of length $i-1$ with endpoints x_1 and x_2, respectively. Since the gadget is symmetric, symmetry between u and v can only be broken by their individual random coin tosses.

4.4 Trade-off Lower Bound

Our goal in this section is to show that the trade-off achieved by Tailor-2-Hop-Coloring's bit drawing function is asymptotically optimal. For this effort, it is sufficient to study lower bounds for the 1-hop variant of the coloring problem, since every 2-hop coloring is also a 1-hop coloring. More precisely, we are going to establish the following:

Theorem 4.2. *Let $\mathcal{A}$ be any randomized uniform anonymous coloring algorithm. If the expected runtime of $\mathcal{A}$ is asymptotically smaller than that of* Tailor-2-Hop-Coloring*, then $\mathcal{A}$'s expected random bit complexity is asymptotically larger than that of* Tailor-2-Hop-Coloring*.*

The rough idea is that in order to break symmetry, the nodes have to draw random bits according to some (possibly randomized) scheme. We distinguish two cases: In the first case, $\mathcal{A}$ may try to break symmetry quickly by using many random bits. We show that then, the expected random bit complexity of $\mathcal{A}$ needs to be large. For the second case, where $\mathcal{A}$ prevents this behavior, we show that the expected runtime of $\mathcal{A}$ is asymptotically as large as that of Tailor-2-Hop-Coloring.

Our proof relies on a graph construction consisting of several so-called (u, v)-*gadgets*. A (u, v)-gadget of length i (depicted in Figure 4.1) consists of $2i$ nodes, namely two two paths p_1, p_2 of length $i - 1$ and two special nodes u and v, connected by an edge. Furthermore, nodes u and v are connected to one endpoint of both p_1 and p_2. The other endpoints of p_1 and p_2 are referred to as x_1 and x_2, respectively. We obtain the graph

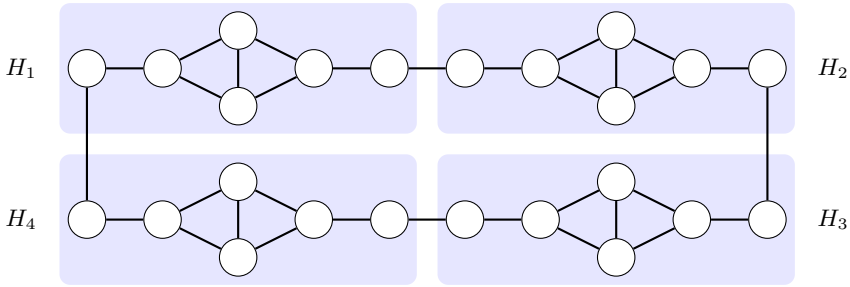

Figure 4.2: The graph $G(4, 3)$, consisting of 4 (u, v)-gadgets H_1, H_2, H_3, and H_4, each of length 3.

$G(m, i)$ utilized in our lower bound proofs by connecting m (u, v)-gadgets of length i in a ring-like topology. This is done by simply chaining the m gadgets together by their endpoint nodes x_1 and x_2—please refer to Figure 4.2 for an illustration. We note that $G(m, i)$ consists of $2im$ nodes.

Consider, for example, the graph $G = G(2^k, 3)$ for some arbitrarily large k. Since the graph G is symmetric from the perspective of each (u, v)-pair in any of the gadgets, every such pair can break symmetry only by their individual random coin tosses. Assume now for the sake of contradiction, that there is a coloring algorithm $\mathcal{A}$ with an expected bit complexity $\beta \in \omega(\log n)$. In that case, with arbitrarily large probability, at least one of the (u, v)-pairs tosses exactly the same sequence of random bits. This contradicts the claim that $\beta \in \omega(\log n)$, and thus we obtain the following result from our graph construction.

Corollary 4.6. *Any coloring algorithm must have an expected random bit complexity in* $\Omega(\log n)$.

In our effort to prove the trade-off lower bound we would like to have a better grip than that on the random coin tosses made by the nodes. Specifically, for any algorithm $\mathcal{A}$ and (u, v)-gadget H, we denote by $B_{\mathcal{A}}(i, H)$ the random variable taking on the maximum number of random bits drawn by nodes u and v in H until and including round i. Whenever $\mathcal{A}$ is clear

from the context, we omit it in the notation and write $B(i, H)$ instead. The following insight about those random variables in the graph $G(m, i)$ will be helpful in our proof of Theorem 4.2.

Lemma 4.7. *Consider any algorithm $\mathcal{A}$, and let H be a single (u, v)-gadget of length i. Let $m \geq 2$ be an integer, and denote by $H_1, \ldots, H_m$ the m (u, v)-gadgets in the graph $G(m, i)$. For any $j \leq i$, all the random variables $B(j, H_k)$, obtained from an execution of $\mathcal{A}$ in $G(m, i)$, are independent and distributed like $B(j, H)$.*

Proof. Observe that in every H_k, the nodes u and v are i hops away from each endpoint, and that $j \leq i$. Consider any (u, v)-gadget H_k in $G(m, i)$. Since $j \leq i$, the execution of $\mathcal{A}$ until round j for nodes u and v cannot depend on any node w from a different gadget $H_l \neq H_k$. It thus holds that all $B(j, H_k)$ are independent and distributed like $B(j, H)$. □

As noted before, the proof for Theorem 4.2 is divided into two parts, depending on how $\mathcal{A}$ chooses to draw random bits (in expectation). For that, based on the bit drawing function b used by Tailor-2-Hop-Coloring (for fixed parameters f, a, and d), we introduce a threshold for the number of random bits drawn by some algorithm. Specifically, we say that algorithm $\mathcal{A}$ draws *a lot of random bits* if

$$\exists i_0 \forall i \geq i_0 \quad \mathbf{E}[B(i, H)] \geq b(3i)/4\,.$$

Note that here H is a (u, v)-gadget of length i. If $\mathcal{A}$ does not draw a lot of random bits, then we say that $\mathcal{A}$ *draws few random bits*. Due to Lemma 4.7, properties of single (u, v)-gadgets can be lifted to instances of $G(m, i)$. One such property we will use is encapsulated in the following technical lemma:

Lemma 4.8. *Let $\mathcal{A}$ be any coloring algorithm. If $\mathcal{A}$ draws a lot of random bits, then*

$$\forall i \geq i_0 \exists j \leq i \quad \mathbf{E}[B(j, H)] \leq b(i)/4, \text{ and } \mathbf{E}[B(j+1, H)] \geq b(i+2)/4\,,$$

where H is a (u, v)-gadget of length i.

Proof. Assume the statement is false. By induction,

$$\begin{aligned}
\mathbf{E}[B(0,H)] &= 0 = b(0) < b(i)/4 \\
\mathbf{E}[B(1,H)] &< b(i+2)/4 \\
\mathbf{E}[B(2,H)] &< b(i+4)/4 \\
&\vdots \\
\mathbf{E}[B(i,H)] &< b(i+2i)/4
\end{aligned}$$

This contradicts the premise that $\mathcal{A}$ draws a lot of random bits, i.e., that $\mathbf{E}[B(i,H)] \geq b(3i)/4$ for the (u,v)-gadget H of length i. □

We now have the essential tools to prove Theorem 4.2, and first consider the case where $\mathcal{A}$ draws a lot of random bits. In that case, for sure, the runtime of $\mathcal{A}$ can be better than that of Tailor-2-Hop-Coloring. Imagine for example a process that draws infinitely many random bits in the first round—one would immediately obtain a 2-hop coloring within a single round with probability 1. The essential insight of the following Lemma 4.9 is that no matter how "smartly" one tries to draw a lot of random bits in hopes to get a better runtime, the expected bit complexity will be asymptotically worse than that of Tailor-2-Hop-Coloring.

Lemma 4.9. *Let $\mathcal{A}$ be any coloring algorithm. If $\mathcal{A}$ draws a lot of random bits, then $\mathcal{A}$'s expected random bit complexity is $\Omega(h(f(n))^2 \cdot \log n)$.*

Proof. Consider the i_0 promised by the fact that $\mathcal{A}$ draws a lot of random bits. Let k be such that $i = dk > i_0$, and let j be the integer obtained from Lemma 4.8 for this i. By Markov's inequality it holds that $\Pr[B(j,H) \leq b(i)/8] \leq 1/2$, where H denotes the (u,v)-gadget of length i. Now consider the graph $G = G(m,i)$, where $m = 2^{b(i)}/(2i)$, and denote by $H_1, \ldots, H_m$ the m copies of H in G. Then, G consists of $n = 2^{b(i)}$ many nodes, i.e., $\log n = b(i)$.

Let N denote the random variable taking on the number of gadgets H_k for which $B(j,H_k) \leq \log(n)/8$. Due to Lemma 4.7, we obtain that all $B(j,H_k)$ are independent and distributed like $B(j,H)$. Therefore, $\mathbf{E}[N]$ is at least $m/2$, and once again applying Markov's inequality, we

get $\Pr[N \geq m/4] \leq 1/2$. The probability p to terminate in round j can now be bounded as

$$p \leq \frac{1}{2} + \frac{1}{2}\left(1 - \frac{1}{2^{\log(n)/8}}\right)^{m/4} \leq \frac{1}{2} + \frac{1}{2}\left(1 - \frac{1}{\sqrt[8]{n}}\right)^{n/(8i)}.$$

Note that since $b(di) = \log g(i)$ and $b(d(i+1)) = \log g(i+1)$, and $g(n) \geq 2^n$, it must hold that $b(d(i+1)) \geq b(di) + 1$. Therefore,

$$b(i) \geq \sum_{\alpha=1}^{\lfloor i/d \rfloor} \underbrace{b(\alpha d) - b((\alpha-1)d)}_{\geq 1} \geq \left\lfloor \frac{i}{d} \right\rfloor \geq \frac{i}{2d}.$$

We obtain that $i \leq 2db(i) \leq 2d \log n$. This means for p that

$$p \leq \frac{1}{2} + \frac{1}{2}\left(1 - \frac{1}{\sqrt[8]{n}}\right)^{n/(16d \log n)} \leq 2/3,$$

for large i. The probability of entering round $j+1$ is thus at least $1/3$. Recalling from Lemma 4.8 that $\mathbf{E}[B(j+1, H)] \geq b(i+2)/4$, we obtain that the expected bit complexity is at least $\frac{1}{3} \cdot \frac{1}{4} b(i+2)$. Now, since $f(n) = f(2^{b(i)}) = f(2^{b(dk)}) = f(2^{a \log g(k)}) \geq k$, we obtain that $df(n) \geq i$. We can therefore bound the expected bit complexity from below by

$$\frac{1}{12} b(i+2) \geq \frac{1}{12} b(df(n)+2) = \frac{1}{12} h(f(n))^2 \cdot \log g(f(n)) \in \Omega(h(f(n))^2 \cdot \log n),$$

as desired. □

Next, we consider the opposite case where $\mathcal{A}$ draws only few random bits.

Lemma 4.10. *Let $\mathcal{A}$ be any coloring algorithm. If $\mathcal{A}$ draws few random bits, then the expected runtime of $\mathcal{A}$ is $\Omega(df(n))$.*

Proof. Fix some i_0, and let $i \geq i_0$ be the constant guaranteed by the fact that $\mathcal{A}$ draws few random bits, i.e., $\mathbf{E}[B(i, H)] < b(3i)/4$. Therefore, due to the Markov inequality, we get that $\Pr[B(i, H) < b(3i)/2] > 1/2$. Now consider the graph G consisting of $m = 2^{b(3i)}/(2i)$ many copies

$H_1, \ldots, H_m$ of H, connected in a ring topology by their endpoints. Then, G consists of $n = 2^{b(3i)}$ many nodes, i.e., $\log n = b(3i)$.

Again, due to Lemma 4.7, all $B(j, H_k)$ are independent and distributed like $B(j, H)$. Let N denote the random variable taking on the number of gadgets H_k in G for which $B(j, H_k) < \log(n)/2$. Then it holds that $\mathbf{E}[N] > m/2$, and with Markov's inequality we get that $\Pr[N > m/4] > 1/2$. Let p be the probability that some gadget H_k does not terminate in round i. We get that

$$p \geq \frac{1}{2}\left(1 - \left(1/\sqrt{n}\right)^{m/4}\right) \geq \frac{1}{3}\,,$$

for sufficiently large i_0. Observe that $f(n) = f(2^{b(3i)}) \leq f(2^{\log g(\lceil 3i/d \rceil)}) = f(g(\lceil 3i/d \rceil)) \leq 3i/d + 1$. We can thus bound the expected runtime of $\mathcal{A}$ from below by $(i+1)/3 \geq (d-1)f(n)/9$, which is in $\Omega(df(n))$ as desired. □

We obtain the desired optimality of the Tailor-2-Hop-Coloring Algorithm from Lemma 4.9 only if $h(f(n))^2 \in \omega(h(f(n)))$. In the case where $f \in \mathcal{O}(\log \log n)$, however, $h(f(n))$ is bounded from above by a constant. It may thus appear that such an f is not covered by our lemmas.

To see that this is not an issue, observe that the constant 3 in the definition of drawing a lot of random bits was chosen arbitrarily. In other words, when $h(f(n))$ is bounded by some constant ρ, one may replace 3 in the above definition with $\rho + 3$. This way, we obtain that the coloring algorithm $\mathcal{A}$ draws "ρ-few" random bits. We can now apply the same reasoning as in the proof of Lemma 4.10 to obtain that the runtime of $\mathcal{A}$ is in the same order as that of Tailor-2-Hop-Coloring. This concludes our effort to establish Theorem 4.2.

5

Local Checking

Network administrators must know whether the network is correct, e.g. whether destination t is reachable from source s, or whether the forwarding rules present in the network imply that packets may potentially be sent in a cycle. Often such network properties are checked by constantly sending probe packets into the network, or, alternatively, by sending the state of all nodes in the network to a central location where all the data is then verified. Both methods take time, often too much time. It would be advantageous to perform these costly global operations only if needed – and otherwise rely on inexpensive local verification [89]. The chapter at hand studies local checkability of fundamental structural properties for directed as well as undirected networks: Nodes of a network can check whether a given *global* structural property of a network is guaranteed, just by *locally* comparing their state with the state of their neighbors.

The concept of local checkability was popularized already in the 1990s

by Naor and Stockmeyer [85]. In our context, this concept refers to the nodes' ability to decide (verify) whether the network has the desired property by exchanging labels with their neighbors. The notion of a *decision* in this distributed setting is inspired by the class co-$\mathcal{NP}$ from sequential complexity theory: The nodes decide Yes if all nodes agree, and No if at least one node disagrees. In practice the disagreement could subsequently be reported. With deterministic algorithms only few properties can be checked locally. If however nodes are allowed to use (a bounded amount of) nondeterminism, a rich complexity hierarchy arises [60]. We focus on the fastest possible case where nodes are only allowed to communicate a single round, cf. [69]. Furthermore, our model has *no strings attached*, i.e., we do not assume any identifiers or port numbers: All we allow is a single exchange of labels between neighbors.

To obtain a better understanding of nondeterminism in the context of distributed computing, let us quickly explain a toy example. Consider the set Bipartite containing all bipartite graphs. In the sequential setting, Bipartite would be called a *language*, and the Yes-instances (*words*) in Bipartite are exactly the graphs that allow a bipartition of the nodes. As in the sequential setting, one may now ask: Is there a (nondeterministic) distributed algorithm deciding whether a given graph G is in Bipartite, using only a single communication round? Indeed, such an algorithm exists [60]. First each node v nondeterministically chooses either the value 0 or 1 and sends it to all neighbors. Next, v checks if all its neighbors sent the value *not* chosen by v.

The proposed nondeterministic algorithm indeed decides Bipartite. A bipartition of the graph corresponds to a nondeterministic choice of 0 and 1 for every node v so that all neighbors of v choose the opposite value. Thus, when the graph G is bipartite, the nodes nondeterministically decide Yes. On the other hand, if G is not bipartite, then in all possible nondeterministic choices of the nodes, at least two nodes will have a neighbor that chose the same value. In that case, the nodes decide No.

Every nondeterministic distributed algorithm can be expressed as a deterministic algorithm with access to a *proof labeling* [60], where the proof labeling corresponds to an oracle in the sequential setting. More precisely, a nondeterministic algorithm is a pair $(\mathcal{P}, \mathcal{V})$, referred to as

prover-verifier pair (PVP). The task of the *prover* $\mathcal{P}$ is to assign labels to nodes (the *proof*) in a YES-instance. The *verifier* $\mathcal{V}$ gets as input at node v only the labels of v and its neighbors. Now $\mathcal{V}$ has to decide YES (at all nodes) in YES-instances labeled by $\mathcal{P}$; In NO-instances $\mathcal{V}$ has to decide NO (for at least one node) regardless of the node labels.

The complexity of such nondeterministic algorithms is measured in terms of the maximum *proof label size* used by $\mathcal{P}$. This corresponds to the number of nondeterministic choices made throughout the execution, and bears similarity with the notion of oracle size in sequential complexity theory. In our BIPARTITE example each node only needs a single bit[1] as its label.

There are two ways to view communication in directed graphs: Nodes can communicate only in the direction of the edge (*directed one-way* communication), or the edge direction imposes no restrictions for communication but only for the network property itself (*directed two-way* communication). We investigate both cases, as well as the undirected case, where nodes communicate with all their neighbors. One of our findings is that all three models are fundamentally different, not only in terms of proof label size, but also in terms of decidability. The results for each of our three network structure detection problems are summarized in Table 5.1.

Another result of our work is the first non-trivial asymptotically tight lower bound for the directed s-t reachability [4] problem that does not rely on descriptive complexity methods. In that problem, two nodes s and t are guaranteed by the problem setting, and the question is whether there is a directed path from s to t. Note that both the directed and the undirected variant are well understood in terms of descriptive complexity, and the directed variant is known to be more difficult [4, 30]. While the observations from [30] lead to a proof label size of 1-bit for the undirected variant, showing a non-trivial lower bound for the directed case remained an open question.

In light of our tight $\Theta(\log n)$ bound for the s-t reachability problem with directed one-way communication we revisit the $O(\log \Delta)$ bound from [60]. In particular, their upper bound relies on the fact that the underlying communication mechanism discloses port numbers to the verifier.

[1] Note that a standard covering argument (the 6-cycle is bipartite, while the 3-cycle is not) can be used to show that one nondeterministic choice is also necessary.

Decision Problem	Directed one-way	Directed two-way	Undirected	Section
s-t reachability[2]	$\Theta(\log n)$	$O(\log \Delta)$	1 [60]	5.4
Contains a Cycle	not possible	$\Theta(\log n)$	2	5.3.1
Acyclic	$\Theta(\log n)$	$\Theta(\log n)$	same as Tree	5.3.2
Tree[3]	$\Theta(\log n)$ [69]	$\Theta(\log n)$	$\Theta(\log n)$ [69]	5.3.3

Table 5.1: The proof label size (in bits) necessary and sufficient for a PVP with respect to different graph decision problems and communication primitives. Here n denotes the number of nodes in the network G, and Δ is the maximum degree of any node in G.

As we will detail in Section 5.4, this is unlikely to be necessary: When directed two-way communication is available, the label can be extended to include *checkable* port numbers using only $O(\log \Delta)$ additional bits. Since referring to a single port number requires $\log \Delta$ bits anyway, this does not change the asymptotic label size.

5.1 Local Checkability in (Un)directed Graphs

Directed and Undirected Graphs. In this chapter, a graph $G = (V, E)$ may be either directed or undirected, but we always assume G to be (weakly[4]) connected. For a node $v \in V$, we denote by $\deg_{\text{in}}(v)$ and $\deg_{\text{out}}(v)$ the number of incoming and outgoing edges of v in G, respectively. We set $\deg(v) = \deg_{\text{in}}(v) = \deg_{\text{out}}(v)$ if G is undirected, and $\deg(v) = \deg_{\text{in}}(v) + \deg_{\text{out}}(v)$ if G is directed. By $\Delta(G) = \max_{u \in V} \deg(u)$ (or simply Δ) we denote the maximum degree in G.

For two nodes $u, v \in V$, let $\text{dist}(u, v)$ denote the distance between both nodes in G (regarding the distance function in the underlying undirected graph in the directed case).

[2] The $O(\log \Delta)$ one-way upper bound with port numbers [60] translates to our two-way model, see Section 5.4.

[3] The $O(\log n)$ upper bound for directed one-way communication from [69] also applies in the two-way model.

[4] A directed graph is called weakly connected if the underlying undirected graph is connected.

Communication Means. Let G be a graph, let ℓ be a node labeling for G, and let v be a node in G. We now consider three means of communication in G, namely U, D_1, and D_2, corresponding to *undirected*, *one-way*, and *two-way* communication, respectively. If G is undirected, then $U(v)$ is the multiset $[\ell(u_1), \ldots, \ell(u_k)]$ containing $\deg(v)$ labels, where $u_1, \ldots, u_k$ are the neighbors of v. If G is directed, then we distinguish two cases. For directed one-way communication, $D_1(v)$ is the multiset $[\ell(u_1), \ldots, \ell(u_k)]$ containing $\deg_{\text{in}}(v)$ labels, where $u_1, \ldots, u_k$ are the in-neighbors of v. For directed two-way communication, $D_2(v)$ is a pair (I, O), where I is $D_1(v)$ and O is the multiset containing $\deg_{\text{out}}(v)$ many labels of v's out-neighbors. We denote the empty multiset by $[\,]$.

Observe that all multisets above are unordered, i.e., there are no unique identifiers and there is no notion of port labels on the edges. If such an order is necessary (for some verifier), then the means to order the multiset need to be included in the proof labels, since the communication mechanism itself does not attach any strings to the messages. In the directed two-way case, however, there is a clear distinction between messages transferred along the edge direction or opposite to it. Note that this distinction is necessary: If it was not made, the directed two-way mode would essentially be equivalent to the undirected case, since the edge direction becomes indistinguishable.

Local Checkability. An *(un)directed network property* is specified by a set Y of (un)directed graphs containing the YES-*instances*, and any (un)directed graph $G \notin Y$ is referred to as a NO-*instance*. A *prover-verifier pair* $(\mathcal{P}, \mathcal{V})$ *for* Y (PVP for short) works as follows.

The *prover* $\mathcal{P}$ gets as an input a graph $G \in Y$ and computes a (finite) node label $\ell(v)$ for every $v \in V$. This labeling ℓ obtained from $\mathcal{P}$ is referred to as *proof*. Let G be any graph, and let ℓ be any node labeling for G. The *verifier* $\mathcal{V}$ is a distributed algorithm that gets as an input at node v the label $\ell(v)$; and in addition either $U(v)$ if Y is an undirected property, or $D_1(v)$ respectively $D_2(v)$ depending on the communication means if Y is a directed property.

A PVP $(\mathcal{P}, \mathcal{V})$ is *correct for* Y if it satisfies

(1) if $G \in Y$ and ℓ was obtained from $\mathcal{P}$, then $\mathcal{V}$ returns YES at all nodes; and

(2) if $G \notin Y$, then $\mathcal{V}$ returns NO for at least one node, regardless of the node labels.

Whenever necessary, we specify the PVP by the communication means used for the verifier, and write *U-PVP*, *D_1-PVP*, and *D_2-PVP* correspondingly. When $X \in \{U, D_1, D_2\}$ is some means of communication, then a network property Y is *X-locally checkable* if there is a correct X-PVP for Y.

The quality of a PVP is measured in terms of the maximum label size in bits assigned by the prover. For a PVP $(\mathcal{P}, \mathcal{V})$, the *proof size of* $(\mathcal{P}, \mathcal{V})$ is $f(n)$ if the labels assigned by $\mathcal{P}$ use at most $f(n)$ bits in any YES-instance containing at most n nodes. For a network property Y, the *X-proof size for Y* is the smallest proof size for which there exists a correct X-PVP for Y. Since the communication means are clear for undirected properties, we omit them in that case. Throughout this chapter, all logarithms use base 2 and are rounded up to be of integer value.

5.2 Related Work

More than 20 years ago, Naor and Stockmeyer [85] raised the question of "*What can be computed locally*?" In their work, the notion of *Locally Checkable Labelings (LCL)* is investigated, where labels are checked in a local fashion, i.e., in a constant number of communication rounds.

This line of research is being followed in many directions, with the concepts of *Proof Labeling Schemes (PLS)*, *Nondeterministic Local Decisions (NLD)*, and *Locally Checkable Proofs (LCP)* being most related to our work. We note that all three approaches are strictly stronger than the model discussed here (by adding either identities, port numbers, or more potent communication models).

The term Locally Checkable Proofs was coined by Suomela and Göös in [60] as an extension to Locally Checkable Labelings, where LCP(f) allows for $f(n)$ bits of additional information per node. They study decision problems from the viewpoint of nondeterministic distributed local algorithms: Is there a proof of size $f(n)$ such that all nodes will output YES for YES-instances, with any (invalid) proof for a NO-instance being rejected by at least one node? The authors introduce a complexity hierarchy for various problems, with LCP(0) being equivalent to LCL. For most of the

results in [60], unique identifiers are assumed for each node, or at least port numbers – which can be used for verification purposes. Thus, their algorithms may use additional strings of information free of cost, which might not be relevant asymptotically for large proof sizes, but come into play for small labels: E.g., in the case of directed s-t reachability, they show that $O(\log \Delta)$ bits suffice by "pointing" at the successor node in the s-t path, a technique relying on port numbers.

The Proof Labeling Schemes of Korman et al. [69] differ from LCPs in the sense that they only use one round of communication to transfer the labels. Thus, upper bounds from PLS apply to LCP and lower bounds from LCP apply to PLS, as the LCP model is strictly more powerful than the PLS model. In [69], the authors also investigate the role of unique identities in PLS and show that there are cases where (given) unique identities are necessary, but also examples where the transition to identities is possible. Nonetheless, they assume the nodes to be aware of the port numbers of their edges.

Closely related to our work, they study (among other problems) the question of whether a connected subgraph is a tree and give asymptotically matching upper and lower bounds of $\Theta(\log n)$ bits for directed one-way communication and the undirected case. Their proofs and techniques for trees carry over to the model considered in this chapter and are thus referenced in Table 5.1. For spanning tree verification, the construction in [69] is also used in the context of Software Defined Networks [89]: Inconsistencies of a spanning tree for routing can be detected locally, triggering a (costly) global recomputation only if needed.

Nondeterministic Local Decisions [54] considers distributed nondeterminism for decision problems. Like LCP and unlike PLS, they allow more than one communication round. However, the proofs are not allowed to depend on the identifier of a node (see [50, 53] for the impact of (missing) identifiers on local decisions). In some sense, as described by [60], the class NLD for connected graphs can be understood as $\text{LCL} \subsetneq \text{NLD} \subsetneq \text{LCP}(\infty)$. Unlike LCP and PLS above, Fraigniaud et al. [54] also study the impact of randomization. Among many other results, they reveal surprising connections between randomization and oracles related to nondeterministic computing: As it turns out, an oracle providing the nodes with the size of the graph gives *"roughly* [...] *the same power to nondeterministic dis-*

tributed computing as randomization does" [54]. Additional recent results concerning the power of randomization for local distributed computing can be found in [49].

Furthermore, there exists a strong connection between proof labeling schemes and self-stabilization (we refer to [41] for an overview of the topic): As characterized by Blin et al. [32], *"any mechanism insuring silent self-stabilization is essentially equivalent to a proof-labeling scheme"*. Even more so, the proof size nearly corresponds to the number of registers for self-stabilization [32]. As such, there has been a long line of research connecting local checking with self-stabilization [2, 21–23].

We ask the question of how a global prover can convince a distributed verifier that it fulfills a certain property. One may also ask the converse question, i.e., how a distributed prover could convince a centralized verifier that knows only node labels, but not the graph structure. This inverted setting is studied in the works of Arfaoui et al. for trees [15] and cycle-freeness [14].

5.3 Checking Network Properties

5.3.1 Cycles

Let U-Cycle denote the set of all undirected connected graphs containing at least one cycle. Let correspondingly D-Cycle denote the set of all weakly connected directed graphs containing at least one directed cycle. Note that an undirected graph is in U-Cycle exactly if it is not an undirected tree, while a directed graph G is in D-Cycle exactly if G is not a directed acyclic graph (DAG). In the remainder of this section we establish the following:

Theorem 5.1. *For the cycle detection problem, it holds that*
(i) There is no D_1-PVP for D-Cycle.
(ii) The D_2-proof size for D-Cycle *is* $\Theta(\log n)$ *bits.*
(iii) The U-proof size for U-Cycle *is* 2 *bits.*

We prove each claim listed in Theorem 5.1 separately, starting with the directed cases. As the first step we show that there cannot be a D_1-PVP for D-Cycle.

Lemma 5.1. *There is no D_1-PVP for* D-Cycle.

Proof. Assume, for the sake of contradiction, that there exists a correct D_1-PVP $(\mathcal{P}, \mathcal{V})$ for D-Cycle. Our goal is to construct a No-instance H and node labels ℓ' for the nodes in H so that $\mathcal{V}$ returns Yes at all nodes. To that end, consider the Yes-instance G (depicted in Figure 5.1) consisting of a cycle with two nodes c_1, c_2, and two additional nodes a, b, where a has the two outgoing edges (a, c_1) and (a, b). Let ℓ denote the node labeling assigned to G by $\mathcal{P}$, and denote by A and B the values $\ell(a)$ and $\ell(b)$, respectively.

Our No-instance H, as shown in Figure 5.1, consists of the three nodes a, b, and b', and the two edges (a, b) and (a, b'). Note that indeed, H does not contain a cycle. By assigning the labels $\ell'(a) = \text{A}$ and $\ell'(b) = \ell'(b') = \text{B}$, we obtain that for all nodes u in H there is a corresponding node v in G for which $(\ell'(u), D_1(u)) = (\ell(v), D_1(v))$. The verifier $\mathcal{V}$ can therefore not differentiate between u and v and thus returns Yes for all nodes in H. This contradicts the assumption that $(\mathcal{P}, \mathcal{V})$ is correct for D-Cycle. □

Lemma 5.2. *There is a D_2-PVP for* D-Cycle *with a proof size of* $\log n$ *bits.*

Proof. We describe a D_2-prover-verifier pair $(\mathcal{P}, \mathcal{V})$ for D-Cycle as required. Let $G = (V, E) \in$ D-Cycle and let $C \subseteq V$ be the set of all nodes that are in a directed cycle. The prover $\mathcal{P}$ labels all nodes $v \in V$ as follows. First, all nodes $v_c \in C$ are labeled with $\ell(v_c) = 0$. All other nodes $v \in V$ are labeled regarding their distance to the closest cycle: The prover $\mathcal{P}$ sets $\ell(v) = \text{dist}_C(v)$, where $\text{dist}_C(v) = \min_{v_c \in C} \text{dist}(v_c, v)$. We

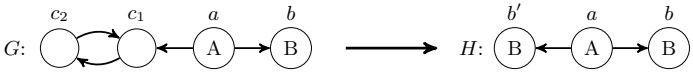

Figure 5.1: Yes-instance G and No-instance H of D-Cycle. A and B are the labels assigned to the nodes a and b in G by the prover $\mathcal{P}$, the node labels in the cycle are not shown. The construction of H yields that for each u in H there is a v in G with $(\ell'(u), D_1(u)) = (\ell(v), D_1(v))$.

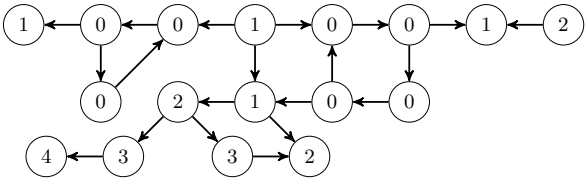

Figure 5.2: A YES-instance of D-CYCLE labeled for two-way communication. All nodes on cycles have the label 0, and all other nodes are labeled with the minimum distance to the nearest cycle using the distance function in the underlying undirected graph.

refer to Figure 5.2 for an example. As the distance is bounded from above by n, the maximum label size is $\log n$ bits.

The verifier $\mathcal{V}$ returns YES for nodes v_c with $\ell(v_c) = 0$ if for the received pair (I, O) of labels holds: There is a label of 0 in I and a label of 0 in O. For the other nodes $v \in V$, YES is returned by $\mathcal{V}$ if a) there is an edge (u, v) or (v, u) such that $\ell(v) = \ell(u) + 1$ and b) no edge (u', v) or (v, u') such that $\ell(v) > \ell(u') + 1$. In all other cases, $\mathcal{V}$ returns NO.

We now show that $\mathcal{V}$ returns YES for all nodes v in YES-instances that were labeled by the prover $\mathcal{P}$: The prover $\mathcal{P}$ labeled only (and all the) nodes on a directed cycle with a 0, i.e., if $\ell(v) = 0$, then $\mathcal{V}$ returns YES for v. The remaining case is $\ell(v) = j > 0$. If $\ell(v) = j$, then $\text{dist}_C(v) = j$, i.e., there exists a node $u \in V$ such that $\text{dist}_C(v) = \text{dist}_C(u) + 1$ and no node $u' \in V$ such that $\text{dist}_C(v) > \text{dist}_C(u') + 1$, as by the definition of $\mathcal{P}$. Thus, $\mathcal{V}$ returns YES as well.

For the D_2-PVP $(\mathcal{P}, \mathcal{V})$ to be correct, it is left to show that $\mathcal{V}$ returns NO for at least one node if the considered graph is not in D-CYCLE. Analogously to the undirected case, let G_{no} be a weakly connected directed graph containing no directed cycle.

For contradiction, assume there would be a node $v \in V(G_{no})$ with $\ell(v) = 0$. Then there has to be a node v_1 with $\ell(v)_1 = 0$ such that there exists an edge (v, v_1), else $\mathcal{V}$ would return NO. This concept of "following the zero" can be iterated, but as the graph is finite (and does not contain a directed cycle), there will be a node v_j for which no node v_{j+1} with $\ell(v_{j+1}) = 0$ exists such that there is an edge (v_j, v_{j+1}). Hence, $\mathcal{V}$ would

return NO and therefore no node can be labeled with 0 in G_{no}.

An idea similar to following the zero can now be applied again: W.l.o.g., let v be a node with the label k. There has to be an edge (v, v_1) with $\ell(v_1) = k - 1$, else $\mathcal{V}$ would return NO for v. Again, as the graph is finite and contains no cycle, following the outgoing edge to a decreasing label is no longer possible at some point. Thus $\mathcal{V}$ will return NO for any weakly connected directed graph not containing a cycle, meaning that the D_2-PVP $(\mathcal{P}, \mathcal{V})$ is correct. □

Lemma 5.3. *The D_2-PVP proof size for* D-CYCLE *is at least* $\log\left(\frac{n-5}{2}\right)/2$ *bits.*

We establish Lemma 5.3 by showing that any D_2-PVP $(\mathcal{P}, \mathcal{V})$ with a smaller proof size can be fooled. To that end, we apply $\mathcal{P}$ to a YES-instance G. We then use the labels applied by $\mathcal{P}$ to construct a NO-instance H for which $\mathcal{V}$ must return YES.

Our construction relies on a graph G, obtained from an undirected path by alternating the edge directions, and creating a cycle with the last two nodes (see Figure 5.3 for an illustration). If the proof size is at most $\log\left(\frac{n-5}{2}\right)/2 - 1$ bits, then less than $\sqrt{n-5}/2\sqrt{2}$ different labels are available. Thus, in G a pair of adjacent labels A, B on the path will appear twice. Moreover, the nodes labeled A have only outgoing edges, and conversely, the nodes labeled B have only incoming edges. We obtain the acyclic NO-instance H by copying the to pairs of nodes, and connecting them as depicted in Figure 5.3. This construction ensures that for all nodes u in H, there is a corresponding node v in G with $(\ell(u), D_2(u)) = (\ell(v), D_2(v))$. Therefore, the verifier $\mathcal{V}$ returns YES for all nodes in H.

Proof. Assume, for the sake of contradiction, there exists a D_2-PVP $(\mathcal{P}, \mathcal{V})$ for D-CYCLE using $\log\left(\frac{n-5}{2}\right)/2 - 1$ bits. Let G be the path $v_1, ..., v_{n-2}$ with $n-2$ nodes and alternating edge directions, connected at v_{n-2} to the cycle v_n, v_{n-1} (which consists of just two nodes). The graph G is a YES-instance of the problem, and thus the verifier $\mathcal{V}$ has to return YES for every node if the graph G was labeled by the prover $\mathcal{P}$.

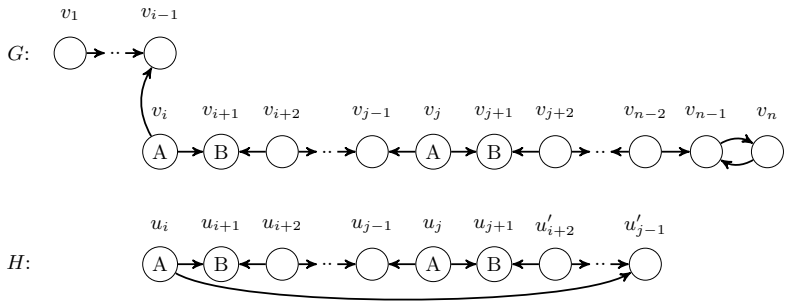

Figure 5.3: YES-instance G (with odd n) and NO-instance H of D-CYCLE. G consists of the n nodes $v_1, \ldots, v_n$. For even k, node v_k has two incoming edges from v_{k-1} and v_{k+1}, whereas all v_k with odd k have two outgoing edges to v_{k-1} and v_{k+1}, i.e., the edge directions alternate. The prover $\mathcal{P}$ assigned the labels $\ell(v_i) = \ell(v_{j+1}) = \text{A}$ and $\ell(v_{i+1}) = \ell(v_{j+1}) = \text{B}$ to the corresponding nodes in G. In H, the nodes $u_i, \ldots, u_{j+2}$ are copies of $v_i, \ldots, v_{j+2}$ from G, and the nodes $u'_{i+1}, \ldots, u'_{j-1}$ are obtained by copying (again) the nodes $v_{i+1}, \ldots, v_{j+2}$. Note that H does not contain a cycle, but due to our construction each $u \in V(H)$ has a corresponding node $v \in V(G)$ with $(\ell(u), D_2(u)) = (\ell(v), D_2(v))$.

We will now construct a NO-instance H (based on G and $\mathcal{P}$) such that for each $v_H \in V(H)$ there is a $v_G \in V(G)$ with $(\ell(v_H), D_2(v_H)) = (\ell(v_G), D_2(v_G))$, i.e., the verifier will output YES for every node in H.

First, we will prove that (due to the construction of G) there are $i \neq j$, with $2 \leq i, j \leq n-3$, such that *a)* $\ell(v_i) = \ell(v_j)$, *b)* $\ell(v_{i+1}) = \ell(v_{j+1})$, and *c)* $\text{dist}(v_i, v_j) = 2k, k \in \mathbb{N}$: There are at least $\lfloor \frac{n-5}{2} \rfloor$ pairs $(i, i+1)$ with $2 \leq i \leq n-3$ for each direction of the edge v_i, v_{i+1}. Labeling each pair differently requires at least $\sqrt{(n-5)/2}$ different labels, i.e., at least $\frac{1}{2}\log(\frac{n-5}{2})$ bits. Hence (using the pigeonhole principle), the claim holds.

The NO-instance H can now be constructed as follows: Let P be the (possibly empty) sub-path $v_{i+2}, ..., v_{j-1}$ in G. We construct the cycle like structure H using two copies of P to connect copies of the pairs v_i, v_{i+1} and v_j, v_{j+1}, see Figure 5.3. We obtain the graph H with the nodes $u_i, ..., u_{j+1}, u'_{i+2}, ..., u'_{j-1}$, where the underlying undirected graph forms a

ring.

It is left to show that we can assign labels to nodes in H such that $\mathcal{V}$ returns YES for all nodes in H. We assign the labels to the nodes in H by setting $\ell(u_x) = \ell(v_x)$ for all x and $\ell(u'_x) = \ell(v_x)$ for all x. It holds for each node $v_H \in H$ that there is a node v_G in G such that $D_2(v_H) = D_2(v_G)$ and $\ell(v_h) = \ell(v_G)$. Thus, as $\mathcal{V}$ returns YES for all nodes in G, $\mathcal{V}$ must return YES for all nodes in H, which contradicts that $(\mathcal{P}, \mathcal{V})$ is correct. □

The claims (i) and (ii) of Theorem 5.1 for directed graphs are now established by Lemmas 5.1 to 5.3. The next two lemmas cover the undirected case (iii).

Lemma 5.4. *There is a U-PVP for* U-CYCLE *with a proof size of* 2 *bits.*

The upper bound for the optimal proof size is established by providing a U-PVP $(\mathcal{P}, \mathcal{V})$ with the desired proof label size. The idea is similar to the directed case, but this time the prover $\mathcal{P}$ labels all cycles with a 3 instead of a 0. Since removing all cycles from G leaves a forest of undirected trees (instead of the collection of DAGs in the directed case), one can save quite a few bits in the labels for the remaining nodes. For each tree, $\mathcal{P}$ picks a root node r that was originally adjacent to a cycle. In each tree, all nodes are labeled with their distance to r modulo 3. An example of a graph G labeled by $\mathcal{P}$ is depicted in Figure 5.4.

The correctness of the PVP is established in a similar manner as in the directed case: The verifier $\mathcal{V}$ can then check if each node supposedly on a cycle (label 3) has at least two neighbors in a cycle, and if every other node (label $\neq 3$) has exactly one node "closer" to the root node of its tree. If G is acyclic, then there can be no node with label 3, as all nodes with a label of 3 would form a forest with at least one leaf. Assume for the sake of contradiction that the verifier returns YES for all nodes, and consider any node in some acyclic graph G. The path obtained by following the labels in descending order (modulo 3), i.e., going towards the root, must have infinite length, since there is no node adjacent to a cycle to break the succession.

Proof. We describe a U-prover-verifier pair $(\mathcal{P}, \mathcal{V})$ as required. Let $G = (V, E) \in$ U-CYCLE. The prover $\mathcal{P}$ labels all nodes $v \in V$ as follows: If v is

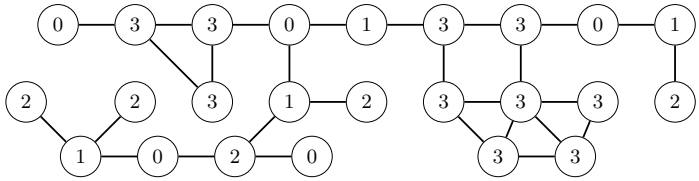

Figure 5.4: A labeled YES-instance of U-CYCLE. Nodes in cycles are labeled 3. The remaining nodes form a forest. After picking a root node adjacent to a cycle for each tree in the forest, all nodes in a tree are labeled with their distance (modulo 3) to the corresponding root.

part of a cycle, then $\ell(v) = 3$. By removing all nodes (and incident edges) that belong to a cycle, the graph decomposes into a set of Trees $\mathscr{T}$. Each tree $T \in \mathscr{T}$ is labeled by first picking a node $r \in T$ that was originally adjacent to a cycle and setting $\ell(r) = 0$. Then, for each other node $t \in T$ let $\text{dist}_T(r, t)$ be the distance from r to t in T and set $\ell(t) = \text{dist}_T(r, t) \mod 3$. An example for the labeling can be found in Figure 5.4. As only the labels $\{0, 1, 2, 3\}$ are used, 2 bits suffice.

The verifier $\mathcal{V}$ returns YES for nodes v with *a*) at least two neighbors have a label of 3 if $\ell(v) = 3$ or *b*) if $\ell(v) = j$, $j \in \{0, 1, 2\}$, then the following three conditions must be fulfilled: *i*) There is no neighbor with a label of j, *ii*) There is exactly one neighbor with a label of $j - 1$ if $j \in \{1, 2\}$ or at most one neighbor with a label of 2 if $j = 0$, and *iii*) all other neighbors must have a label of exactly $j + 1 \mod 3$ or 3. In all other cases, $\mathcal{V}$ returns NO. If a node v is part of an undirected cycle (hence, $\ell(v) = 3$), then it has at least two neighbors in the cycle with the label 3, meaning that $\mathcal{V}$ returns YES for v. Else, consider the tree $T \in \mathscr{T}$ from above with $v \in T$ with the corresponding "root" node r picked by the prover. If $v = r$, then all neighbors in T have the label 1 and all other neighbors (of whom at least one exists) are on cycles with a label of 3. Thus, $\mathcal{V}$ outputs YES for $v = r$. If $v \in T$ and $v \neq r$, then all neighbors v' of v in T are labeled according to $\ell(v') = \text{dist}_T(r, v') \mod 3$. All other neighbors (if any exist) of v in G must be on cycles with a label of 3. Hence, $\mathcal{V}$ returns also YES in this case.

For the U-prover-verifier pair $(\mathcal{P}, \mathcal{V})$ to be correct, it is left to show

that $\mathcal{V}$ returns NO for at least one node if the considered graph is not in U-CYCLE. Let G_{no} be a connected undirected graph containing no cycle.

Assume there would be a node $v \in V(G_{no})$ with $\ell(v) = 3$. Consider all nodes with a label of 3 in $V(G_{no})$: As there is no cycle, the subgraph(s) induced by these nodes form a forest $\mathscr{F}$. Let $T \in \mathscr{F}$ be the tree with $v \in T$. Pick a leaf of T: It has at most one neighbor with a label of 3, meaning that $\mathcal{V}$ will return NO for at least one node.

Thus, no node v with $\ell(v) = 3$ can exist. Now, pick any node $v \in V(G_{no}$ with $\ell v \in \{0, 1, 2\}$. v (and also any other node in $V(G_{no})$ must have exactly one neighbor v_1 with a label of $\ell(v_1) = \ell(v) - 1 \mod 3$, as else $\mathcal{V}$ would return NO for v. Consider the path starting from v that picks as its next node the unique neighbor with a label smaller by one modulo 3, i.e., $v, v_1, \ldots$ - until no such node exists any more. Since G_{no} is cycle-free, the path must be finite and end at some node v_j. As v_j has no neighbor with a label of 3 or a label of $\ell(v_j) - 1 \mod 3$, the verifier $\mathcal{V}$ returns NO for v_j. Thus $\mathcal{V}$ will return NO for any connected graph not containing a cycle, meaning that the U-PVP $(\mathcal{P}, \mathcal{V})$ is correct. □

Lemma 5.5. *The U-PVP proof size for* U-CYCLE *is at least* 2 *bits.*

The proof is by case distinction, cf. Figure 5.5. We construct a graph G consisting of a path P of three nodes attached to a cycle. If the proof size is restricted to just one bit, then there are only $2^3 = 8$ possible combinations to label the three nodes in P. For each of the eight cases, we can construct a NO-instance H such that for each $v_H \in V(H)$ there is a $v_G \in V(G)$ with $(\ell(v_H), U(v_H)) = (\ell(v_G), U(v_G))$, meaning that the verifier will output YES for each node in H.

Proof. Assume there exists a U-PVP $(\mathcal{P}, \mathcal{V})$ for U-CYCLE using 1 bit. We use a YES-instance $G = (V(G), E(G))$ of U-CYCLE consisting of a cycle with three nodes with a path P of three nodes attached to it. We will show that for any labeling ℓ assigned to the nodes on the path P, for which $\mathcal{V}$ returns YES for all nodes in G, there exists a NO-instance $H = (V(H), E(H))$ of U-CYCLE for which $\mathcal{V}$ must also return YES for all nodes in H. W.l.o.g. consider the four cases in Figure 5.5.

These four cases combined with their analogous inversions, where all labels are switched on the path P, present all combinations of how labels

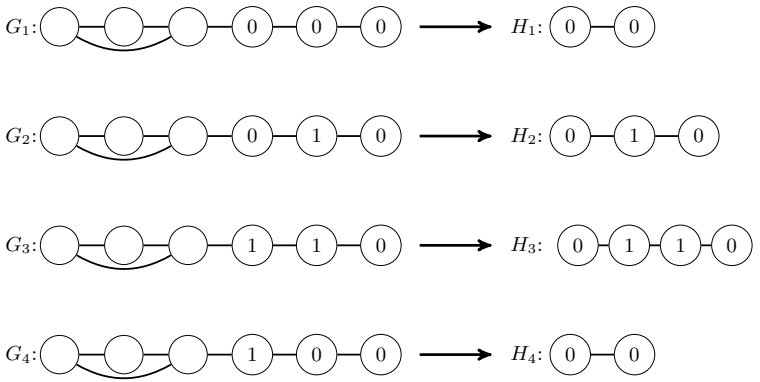

Figure 5.5: YES-instances G_1, G_2, G_3, G_4 and NO-instances H_1, H_2, H_3, H_4 of U-CYCLE. For any labeling assigned to the YES-instances, there exists a NO-instance for which $\mathcal{V}$ must return YES for all nodes. In these graphs, the labels for the nodes in the cycle can be chosen arbitrarily. The numbers in the remaining nodes are their labels. All labels in this figure can be inverted to get the remaining 4 possible combinations for a labeling.

can be assigned to the nodes on the path P. For every YES-instance G there exists a NO-instance H such that for each $v_H \in V(H)$ there is a $v_G \in V(G)$ with $(\ell(v_H), U(v_H)) = (\ell(v_G), U(v_G))$. Since $\mathcal{V}$ can not differentiate between v_H and V_G, it must also return YES for all nodes in the corresponding NO-instance, which contradicts that $(\mathcal{P}, \mathcal{V})$ is correct. It follows that there is no correct proof labeling scheme $(\mathcal{P}, \mathcal{V})$ using only 1 bit. □

5.3.2 Acyclicity

In the undirected case, an acyclic graph is nothing but an undirected tree. The question of detecting undirected trees was already answered in [69] (see Section 5.3.3). In the directed case, however, not every acyclic graph is necessarily a tree. Let D-ACYCLIC denote the set of all weakly connected directed acyclic graphs. In the remainder of this section we establish the following:

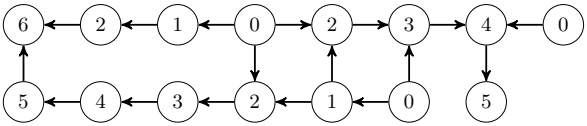

Figure 5.6: A labeled YES-instance of D-ACYCLIC. Nodes without incoming edges are labeled 0, all other nodes have a label that is equal to the highest incoming label plus 1.

Theorem 5.2. *For the acyclicity detection problem, it holds that*
(i) The D_1-proof size for D-ACYCLIC *is* $\Theta(\log n)$ *bits.*
(ii) The D_2-proof size for D-ACYCLIC *is* $\Theta(\log n)$ *bits.*

While not every directed acyclic graph is a directed tree, the converse holds, i.e., every directed tree is a directed acyclic graph. Techniques similar to those used by Korman et al. [69] can be used to obtain the claimed lower bound for tree detection in our model. Hence, we only need to establish the upper bounds in Theorem 5.2. Note that any D_1-PVP immediately yields a D_2-PVP with the same proof size by simply ignoring the information obtained via outgoing edges. It is therefore sufficient to find a D_1-PVP with the desired proof size.

Lemma 5.6. *There is a D_1-PVP for* D-ACYCLIC *with a proof size of* $\log n$ *bits.*

In the proof, the prover assigns each node with no incoming labels the label 0, and each other node the highest incoming label plus one. We refer to Figure 5.6 for illustration. Thus, each node with label $j > 0$ can check if there is an incoming label $j - 1$, or when $j = 0$, if the multiset of incoming labels is the empty set. As each NO-instance contains a cycle, a node with the highest label in the cycle would send its label to another node, causing this node to output NO.

Proof. We describe a D_1-prover-verifier pair $(\mathcal{P}, \mathcal{V})$ as required. Let $G = (V, E) \in$ D-ACYCLIC and let $V_0 \subseteq V$ be the set of all nodes $v_0 \in V$ with $D_1(v) = [\emptyset]$, i.e., v_0 has zero incoming edges. The prover $\mathcal{P}$ labels all nodes $v \in V$ as follows. *a*) All nodes $v_0 \in V_0$ have the label $\ell(v_0) = 0$, and *b*) for all other nodes $v_+ \in V$ holds: $\ell(v_+) = 1 + \max_{(u,v_+)\in E} \ell(u)$. We

refer to Figure 5.6 for an example. As a label i requires a label $i-1$ to exist, the highest label is bounded from above by n, inducing a maximum label size of $\log n$ bits.

The verifier $\mathcal{V}$ returns YES for nodes v with *a)* $D_1(v) = [\,]$ if $\ell(v) = 0$ or *b)* $\ell(v) = 1 + \max_{(u,v)\in E} \ell(u)$ if $D_1(v) \neq [\,]$. In all other cases, $\mathcal{V}$ returns NO. Thus, the verifier returns YES for all nodes in V if G was labeled by $\mathcal{P}$, as all incoming labels are available to the verifier.

For the D_1-prover-verifier pair $(\mathcal{P}, \mathcal{V})$ to be correct, it is left to show that $\mathcal{V}$ returns NO for at least one node if the considered graph is not in D-ACYCLIC. Let G_c be a weakly connected directed graph containing a directed cycle $C = v_1, v_2, \ldots, v_{|C|}, v_1$. W.l.o.g., let $v_i \in C$ be a node with the highest labeling in C. Consider the outgoing edge from v_i in C: The corresponding neighbor of v_i in C cannot have a higher label than v_i. Thus $\mathcal{V}$ will return NO, meaning that the D_1-PVP $(\mathcal{P}, \mathcal{V})$ is correct. □

5.3.3 Trees

Let U-TREE denote the set of all undirected trees. Let correspondingly D-TREE denote the set of all weakly connected directed trees in which all edges are directed away from some unique root node.

Theorem 5.3 ([69]). *For the tree detection problem, it holds that*

(i) The proof size for U-TREE *is* $\Theta(\log n)$ *bits.*
(ii) The D_1*-proof size for* D-TREE *is* $\Theta(\log n)$ *bits.*
(iii) The D_2*-proof size for* D-TREE *is* $\Theta(\log n)$ *bits.*

While the authors of [69] assumed port numbers to be available, the PLS used in the upper bound construction do not make use of them. Therefore, the upper bound claims (i) and (ii) carry over to our model. Since their port numbering model is strictly stronger than ours, the same is true for the lower bounds. Naturally, upper bounds for D_1-proof sizes carry over to the D_2-case, so the only thing that is left is to show that there exists no D_2-PVP with a proof size of $o(\log n)$ bits. Since the counter example construction to establish this claim are very similar to the construction used in [69], we omit the details here.

5.4 Port Numbers vs. s-t Reachability

As pointed out by Göös and Suomela in [60], "*To ask meaningful questions about connectivity [...] we have the promise that there is exactly one node with label s and exactly one node with label t.*" In this section, we thus assume that all graphs have at least two nodes, of which one node has the unique label s and another node has the unique label t. It is known that in the undirected case, the U-proof size for s-t reachability is 1 bit, as argued in the introduction to this chapter. In the directed case, on which we focus, a non-trivial lower bound remained an open question [60]. For that, let s-t REACHABILITY denote the set of all directed graphs containing a directed path from s to t.

We show a lower bound for s-t REACHABILITY with one-way communication by combining our previously used techniques. The upper bound for the two-way case requires a new insight: As it turns out, port numbers can be emulated in our model by implementing a 2-hop coloring with only $O(\log \Delta)$ bits. Then, whenever a port number is required for some proof, we only need to pay at most $O(\log \Delta)$ bits. While this seems like a high price to pay, we note that referring to a specific port number requires $O(\log \Delta)$ bits even if the port numbering itself is provided for free. We will later see how this applies in the case of two-way s-t REACHABILITY (cf. [60]). In the remainder of this section we establish the following theorem:

Theorem 5.4. *For the s-t REACHABILITY problem, it holds that*
(i) The D_1-proof size for s-t REACHABILITY is $\Theta(\log n)$ bits.
(ii) The D_2-proof size for s-t REACHABILITY is at most $O(\log \Delta)$ bits.

To see that s-t REACHABILITY permits a D_1-PVP with a proof size of $O(\log n)$ bits, observe that the nodes on the path can simply be enumerated, cf. Figure 5.7. Each node on the path can now check whether it has a predecessor on the path, i.e., every YES-instance is verified correctly. To see that NO-instances will be rejected, one can follow a similar line of arguments as in Lemma 5.6: Every path obtained by following descending incoming labels, starting from t, must end in a node without a predecessor, since the graph is finite and s and t are not connected.

Lemma 5.7. *There is D_1-PVP for s-t* REACHABILITY *with a proof size of $O(\log n)$ bits.*

Proof. We describe a D_1-prover-verifier pair $(\mathcal{P}, \mathcal{V})$ as required. Let the directed graph $G = (V, E) \in$ *s-t* REACHABILITY and let $P = s, v_1, \ldots, v_j, t$ be a shortest directed path from s to t. The prover $\mathcal{P}$ labels all nodes $v \notin P$ with $\ell(v) = 0$ and each node $v_i \in P = s, v_1, \ldots, v_j, t$ with $\ell(v_i) = i$, i.e., $\ell(v_i) = \text{dist}(s, v_i)$ by definition of P. We refer to Figure 5.7 for illustration. As $dist(s, t) \leq n$, $\ell(s) = s$, and $\ell(t) = t$, the proof size is in $O(\log n)$ bits.

The verifier $\mathcal{V}$ returns YES for all nodes v with a label of 0 and for the node s with unique label $\ell(s) = s$. For the node t with the unique label $\ell(t) = t$, YES is returned if *a*) the label s is received, or *b*) if a label greater than zero is received. $\mathcal{V}$ returns YES for all other nodes v with label $\ell(v) = i > 1$, if one of the received labels is $i - 1$. For the special case of $\ell(v) = 1$, one of the received labels has to be s.

Thus, the verifier will return YES at all nodes for YES-instances labeled by $\mathcal{P}$: The nodes v with $\ell(v) = 0$ and s return YES. Furthermore, as each other node is on the path $P = s, v_1, \ldots, v_j, t$ with $\ell(v_i) = i$, they have a predecessor on the path with the desired label, and hence return YES as well.

It is left to show that $\mathcal{V}$ returns NO for at least one node if the graph G is not in *s-t* REACHABILITY. Let H be a NO-instance of *s-t* REACHABILITY, i.e., there is no directed path from s to t. Consider the set Z of nodes that can be reached from t by traversing directed edges in the reverse direction to a node with a label lower by exactly one, or in the case of t, with any label greater than zero. Note that by definition of H, there is no node $v' \in Z$ such that there is an edge $(s, v') \in H(E)$. Let v^* be a node with the lowest label $\ell(v^*) = x$ in Z: As v^* cannot receive a label $x - 1$ or the label s, v^* will return NO. Hence, the desribed D_1-PVP $(\mathcal{P}, \mathcal{V})$ is correct. □

Lemma 5.8. *The D_1-PVP proof size for s-t* REACHABILITY *is at least* $\log\left(\frac{n}{4}\right) - 2$ *bits.*

Proof. Assume, for the sake of contradiction, that there is a D_1-PVP $(\mathcal{P}, \mathcal{V})$ for *s-t* REACHABILITY with a proof size of $\log(n/4) - 3$ bits. Let n

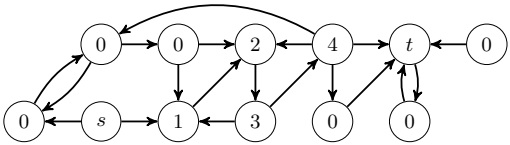

Figure 5.7: A labeled YES-instance of s-t REACHABILITY. All nodes v in G on the shortest s-t path P of length 5 are labeled $\ell(v) = \text{dist}(s, v)$. All other nodes are labeled 0.

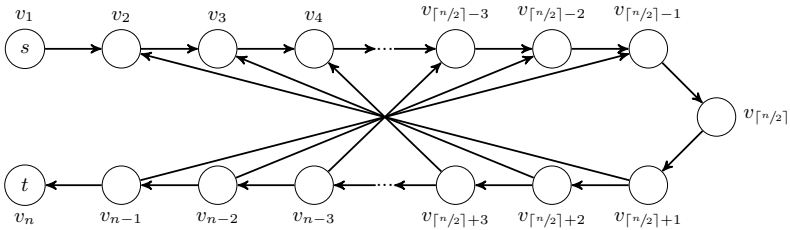

Figure 5.8: The YES-instance G of s-t REACHABILITY used in our proof of Lemma 5.8. Nodes v_j with $j > \lceil n/2 \rceil$ have an outgoing edge to $v_{j-\lceil n/2 \rceil+1}$. Note that G contains only one simple s-t path.

be odd and let G be the directed path $P = v_1, \ldots, v_n$ where $v_1 = s$, and $v_n = t$. We add to G directed edges so that all nodes v_k with $n > k > \lceil n/2 \rceil$ have an outgoing edge to $v_{k-\lceil n/2 \rceil+1}$, as depicted in Figure 5.8. We note that G is a YES-instance for s-t REACHABILITY, and that there is only one simple path from s to t in G. Like above, we now apply $\mathcal{P}$ to G and use the obtained labels ℓ to construct a NO-instance H with a labeling ℓ'. The construction ensures that for every node u in H, there is a node v in G with $(\ell'(u), D_1(u)) = (\ell(v), D_1(v))$.

With an argument analogous to that in the proof of Lemma 5.3, we will first show that there are $i \neq j$, with $\lceil n/2 \rceil + 2 \leq i \leq n - 4$ and $i + 2 \leq j \leq n - 2$, such that $\ell(v_i) = \ell(v_j)$. In other words, we are looking for two non-adjacent nodes v_i and v_j that are within the second half of

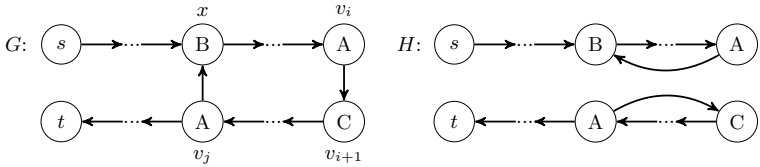

Figure 5.9: YES-instance G of s-t REACHABILITY labeled by $\mathcal{P}$, and the corresponding NO-instance H for s-t REACHABILITY. Some label A appears twice on the s-t path, namely at the nodes v_i and v_j. Since v_j is at least two steps after v_i and has an outgoing edge to a node x before v_i, the NO-instance H for which $\mathcal{V}$ fails can be constructed. For the sake of simplicity not all edges are shown.

the path P, and v_i comes before v_j on P. Suppose that there are no such v_i, v_j. Consequently, $\ell(v_{\lceil n/2 \rceil+2})$ must be different from the label of each node in $\{v_{\lceil n/2 \rceil+4}, ..., v_{n-2}\}$. By induction, if v_i, v_j with the desired properties do not exist, then there need to be at least $\lfloor \frac{n}{4} \rfloor - 2$ different labels on the sub-path $v_{\lceil n/2 \rceil+2}, ..., v_{n-2}$. This is a contradiction to the assumption that the proof size is limited to $\lfloor \frac{n}{4} \rfloor - 3$ bits, and we conclude that such nodes v_i, v_j must be present.

To complete our proof of Lemma 5.8, we now construct the NO-instance H and the labeling ℓ'. For that, let v_i and v_j be the two nodes in G with $\ell(v_i) = \ell(v_j)$ as above. We denote by x the node $v_{j-\lceil n/2 \rceil+1}$ in the first half of P with an incoming edge from v_j. To construct H and ℓ', we first copy the graph G including the labels assigned by ℓ. We then replace the edges (v_i, v_{i+1}) and (v_j, x) by (v_i, x) and (v_j, v_{i+1}) (see Figure 5.9). Note that in H, the two distinguished nodes s and t are no longer connected by a directed path.

It is left to show that $\mathcal{V}$ will return YES for all nodes in H when labeled with ℓ'. Note that in H, the only nodes that were changed in some way in comparison to G were v_i, v_{i+1}, v_j and x. However, all four nodes still have the same labels, and the incoming edges were changed only for x and v_{i+1}. As it holds that $\ell'(v_i) = \ell'(v_j)$, it follows that $D_1(v_{i+1})$ is the same in G and in H, equivalently for $D_1(x)$. Thus, since the verifier $\mathcal{V}$ returned YES for all nodes in G, $\mathcal{V}$ must also return YES for all nodes in H, contradicting that $(\mathcal{P}, \mathcal{V})$ is correct. □

We note that the construction in the proof of Lemma 5.8 has constant degree in every node. Therefore, there cannot be a D_1-PVP for which the proof size depends only on Δ. The missing part to establish Theorem 5.4 is a D_2-PVP for s-t REACHABILITY.

The PVP for this problem as proposed by [60] relies on the nodes' ability to "point to an edge" by using its port number. The authors suggest to mark an s-t path P by simply pointing to the edges used by P. We argue that one can point to an edge in our models U and D_2, even though port numbers are not available. To that end, we enrich the labels to include a 2-hop coloring of the graph G, i.e., a coloring (node labeling) such that the label of each node v is unique among all nodes with distance at most 2. We denote this coloring by $c(v)$.

Since a 2-hop coloring requires at most Δ^2+1 colors, each color can be encoded using $O(\log \Delta)$ bits. Moreover, the 2-hop coloring can be checked locally, since each node v only needs to verify if v and all its neighbors have different colors. A PVP can now rely on the fact that v's color is unique among u's neighbors.

To obtain a D_2-PVP for for s-t REACHABILITY, a node $u \in V$ can now point to an edge $(u, v) \in E$ by referencing $c(v)$ in its label. In this way, the pointed-to edge is uniquely specified for both u and v. Applying the same reasoning as in [60], we obtain the following lemma, which together with Lemmas 5.7 and 5.8 concludes our effort to establish Theorem 5.4.

Lemma 5.9. *There is a D_2-PVP for s-t* REACHABILITY *with a proof size of $O(\log \Delta)$ bits.*

Bibliography

[1] Abrahamson, K.R., Adler, A., Higham, L., Kirkpatrick, D.G.: Probabilistic solitude verification on a ring. In Halpern, J.Y., ed.: PODC, ACM (1986) 161–173

[2] Afek, Y., Kutten, S., Yung, M.: The local detection paradigm and its application to self-stabilization. Theor. Comput. Sci. **186**(1-2) (1997) 199–229

[3] Afek, Y., Matias, Y.: Elections in anonymous networks. Inf. Comput. **113**(2) (1994) 312–330

[4] Ajtai, M., Fagin, R.: Reachability is harder for directed than for undirected finite graphs. J. Symb. Log. **55**(1) (1990) 113–150

[5] Alon, N., Babai, L., Itai, A.: A fast and simple randomized parallel algorithm for the maximal independent set problem. Journal of Algorithms **7**(4) (1986) 567 – 583

[6] Amit, A., Linial, N., Matousek, J., Rozenman, E.: Random lifts of graphs. In Kosaraju, S.R., ed.: SODA, ACM/SIAM (2001) 883–894

[7] Angluin, D.: Local and global properties in networks of processors (extended abstract). In Miller, R.E., Ginsburg, S., Burkhard, W.A., Lipton, R.J., eds.: STOC, ACM (1980) 82–93

[8] Angluin, D., Aspnes, J., Chan, M., Fischer, M.J., Jiang, H., Peralta, R.: Stably computable properties of network graphs. In Prasanna, V.K., Iyengar, S.S., Spirakis, P.G., Welsh, M., eds.: DCOSS. Volume 3560 of Lecture Notes in Computer Science., Springer (2005) 63–74

[9] Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M.J., Peralta, R.: Computation in networks of passively mobile finite-state sensors. [39] 290–299

[10] Angluin, D., Aspnes, J., Eisenstat, D.: Stably computable predicates are semilinear. In: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing. PODC '06, New York, NY, USA, ACM (2006) 292–299

[11] Angluin, D., Aspnes, J., Eisenstat, D., Ruppert, E.: The computational power of population protocols. Distributed Computing **20** (2007) 279–304

[12] Angluin, D., Fischer, M., Jiang, H.: Stabilizing consensus in mobile networks. In Gibbons, P., Abdelzaher, T., Aspnes, J., Rao, R., eds.: Distributed Computing in Sensor Systems. Volume 4026 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2006) 37–50

[13] Angluin, D., Gardiner, A.: Finite common coverings of pairs of regular graphs. J. Comb. Theory, Ser. B **30**(2) (1981) 184–187

[14] Arfaoui, H., Fraigniaud, P., Ilcinkas, D., Mathieu, F.: Distributedly testing cycle-freeness. In: WG. (2014) 15–28

[15] Arfaoui, H., Fraigniaud, P., Pelc, A.: Local decision and verification with bounded-size outputs. In: SSS. (2013) 133–147

[16] Aspnes, J., Ruppert, E.: An introduction to population protocols. In Garbinato, B., Miranda, H., Rodrigues, L., eds.: Middleware for Network Eccentric and Mobile Applications. Springer-Verlag (2009) 97–120

[17] Aspnes, J., Waarts, O.: Randomized consensus in expected $o(n \log^2 n)$ operations per processor. SIAM J. Comput. **25** (1996) 1024–1044

[18] Attiya, H., Censor, K.: Tight bounds for asynchronous randomized consensus. J. ACM **55** (2008)

[19] Attiya, H., Ellen, F.: Impossibility Results for Distributed Computing. Morgan & Claypool Publishers (2014)

[20] Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations and Advanced Topics. John Wiley & Sons (2004)

[21] Awerbuch, B., Patt-Shamir, B., Varghese, G.: Self-stabilization by local checking and correction (extended abstract). In: FOCS. (1991) 268–277

[22] Awerbuch, B., Patt-Shamir, B., Varghese, G., Dolev, S.: Self-stabilization by local checking and global reset (extended abstract). In: WDAG. (1994) 326–339

[23] Awerbuch, B., Varghese, G.: Distributed program checking: a paradigm for building self-stabilizing distributed protocols (extended abstract). In: FOCS. (1991) 258–267

[24] Barenboim, L., Elkin, M.: Distributed $(\Delta + 1)$-coloring in linear (in Δ) time. In: STOC. (2009)

[25] Barenboim, L., Elkin, M.: Deterministic distributed vertex coloring in polylogarithmic time. J. ACM **58** (2011) 23

[26] Barenboim, L., Elkin, M.: Distributed Graph Coloring: Fundamentals and Recent Developments. Morgan & Claypool Publishers (2013)

[27] Barenboim, L., Elkin, M., Kuhn, F.: Distributed $(\Delta + 1)$-coloring in linear (in Δ) time. SIAM J. Comput. **43** (2014) 72–95

[28] Beauquier, J., Blanchard, P., Burman, J.: Self-stabilizing leader election in population protocols over arbitrary communication graphs. In

Baldoni, R., Nisse, N., van Steen, M., eds.: OPODIS. Volume 8304 of Lecture Notes in Computer Science., Springer (2013) 38–52

[29] Beauquier, J., Gradinariu, M., Johnen, C.: Memory space requirements for self-stabilizing leader election protocols. [40] 199–207

[30] Beeri, C., Kanellakis, P.C., Bancilhon, F., Ramakrishnan, R.: Bounds on the propagation of selection into logic programs. J. Comput. Syst. Sci. **41**(2) (1990) 157–180

[31] Blair, J.R.S., Manne, F.: An efficient self-stabilizing distance-2 coloring algorithm. Theor. Comput. Sci. **444** (2012) 28–39

[32] Blin, L., Fraigniaud, P., Patt-Shamir, B.: On proof-labeling schemes versus silent self-stabilizing algorithms. In: SSS. (2014) 18–32

[33] Boldi, P., Vigna, S.: Computing anonymously with arbitrary knowledge. [40] 181–188

[34] Boldi, P., Vigna, S.: An effective characterization of computability in anonymous networks. In Welch, J.L., ed.: DISC. Volume 2180 of Lecture Notes in Computer Science., Springer (2001) 33–47

[35] Boldi, P., Vigna, S.: Fibrations of graphs. Discrete Mathematics **243**(1–3) (2002) 21 – 66

[36] Boldi, P., Vigna, S.: Universal dynamic synchronous self–stabilization. Distributed Computing **15**(3) (2002) 137–153

[37] Chalopin, J., Das, S., Santoro, N.: Groupings and pairings in anonymous networks. In: Proceedings of the 20th international conference on Distributed Computing. DISC'06, Berlin, Heidelberg, Springer-Verlag (2006) 105–119

[38] Chalopin, J., Godard, E., Métivier, Y.: Local terminations and distributed computability in anonymous networks. In Taubenfeld, G., ed.: Distributed Computing. Volume 5218 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2008) 47–62

[39] Chaudhuri, S., Kutten, S., eds.: Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004, St. John's, Newfoundland, Canada, July 25-28, 2004. In Chaudhuri, S., Kutten, S., eds.: PODC, ACM (2004)

[40] Coan, B.A., Welch, J.L., eds.: Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing, PODC, '99Atlanta, Georgia, USA, May 3-6, 1999. In Coan, B.A., Welch, J.L., eds.: PODC, ACM (1999)

[41] Dolev, S.: Self-Stabilization. Mit Press (2000)

[42] Dolev, S., Israeli, A., Moran, S.: Uniform dynamic self-stabilizing leader election. In Toueg, S., Spirakis, P., Kirousis, L., eds.: Distributed Algorithms. Volume 579 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (1992) 167–180

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

[44] Dolev, S., Schiller, E., Welch, J.L.: Random walk for self-stabilizing group communication in ad-hoc networks. In: SRDS, IEEE Computer Society (2002) 70–79

[45] Dolev, S., Tzachar, N.: Randomization adaptive self-stabilization. Acta Inf. **47** (2010) 313–323

[46] Emek, Y., Pfister, C., Seidel, J., Wattenhofer, R.: Anonymous networks: randomization = 2-hop coloring. In: PODC. (2014)

[47] Emek, Y., Wattenhofer, R.: Stone age distributed computing. In Fatourou, P., Taubenfeld, G., eds.: PODC, ACM (2013) 137–146

[48] Flocchini, P., Kranakis, E., Krizanc, D., Luccio, F.L., Santoro, N.: Sorting and election in anonymous asynchronous rings. J. Parallel Distrib. Comput. **64**(2) (February 2004) 254–265

[49] Fraigniaud, P., Göös, M., Korman, A., Parter, M., Peleg, D.: Randomized distributed decision. Distributed Computing **27**(6) (2014) 419–434

[50] Fraigniaud, P., Göös, M., Korman, A., Suomela, J.: What can be decided locally without identifiers? In: PODC. (2013) 157–165

[51] Fraigniaud, P., Halldórsson, M., Korman, A.: On the impact of identifiers on local decision. In Baldoni, R., Flocchini, P., Binoy, R., eds.: Principles of Distributed Systems. Volume 7702 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 224–238

[52] Fraigniaud, P., Ilcinkas, D., Pelc, A.: Oracle size: A new measure of difficulty for communication tasks. In: Proceedings of the Twenty-fifth Annual ACM Symposium on Principles of Distributed Computing. PODC '06, New York, NY, USA, ACM (2006) 179–187

[53] Fraigniaud, P., Korman, A., Parter, M., Peleg, D.: Randomized distributed decision. In Aguilera, M.K., ed.: DISC. Volume 7611 of Lecture Notes in Computer Science., Springer (2012) 371–385

[54] Fraigniaud, P., Korman, A., Peleg, D.: Local distributed decision. In Ostrovsky, R., ed.: FOCS, IEEE Computer Society (2011) 708–717

[55] Fraigniaud, P., Rajsbaum, S., Travers, C.: Locality and checkability in wait-free computing. In Peleg, D., ed.: DISC. Volume 6950 of Lecture Notes in Computer Science., Springer (2011) 333–347

[56] Gebremedhin, A.H., Manne, F., Pothen, A.: Parallel distance-k coloring algorithms for numerical optimization. In Monien, B., Feldmann, R., eds.: Euro-Par. Volume 2400 of Lecture Notes in Computer Science., Springer (2002) 912–921

[57] Godard, E., Métivier, Y., Muscholl, A.: Characterizations of classes of graphs recognizable by local computations. Theory Comput. Syst. **37**(2) (2004) 249–293

[58] Godsil, C.D., Royle, G.: Algebraic Graph Theory. Graduate Texts in Mathematics. Springer (2001)

[59] Goldberg, A.V., Plotkin, S.A., Shannon, G.E.: Parallel symmetry-breaking in sparse graphs. SIAM J. Discrete Math. **1** (1988) 434–446

[60] Göös, M., Suomela, J.: Locally checkable proofs. In Gavoille, C., Fraigniaud, P., eds.: PODC, ACM (2011) 159–168

[61] Gradinariu, M., Tixeuil, S.: Self-stabilizing vertex coloration and arbitrary graphs. In: OPODIS. (2000)

[62] Guerraoui, R., Ruppert, E.: What can be implemented anonymously? In: Proceedings of the 19th international conference on Distributed Computing. DISC'05, Berlin, Heidelberg, Springer-Verlag (2005) 244–259

[63] Herlihy, M., Rajsbaum, S.: A classification of wait-free loop agreement tasks. Theoretical Computer Science **291**(1) (2003) 55 – 77 ¡ce:title¿Distributed Computing¡/ce:title¿.

[64] Herlihy, M., Shavit, N.: The topological structure of asynchronous computability. J. ACM **46**(6) (November 1999) 858–923

[65] Israeli, A., Itai, A.: A fast and simple randomized parallel algorithm for maximal matching. Information Processing Letters **22**(2) (1986) 77 – 80

[66] Itai, A., Rodeh, M.: Symmetry breaking in distributive networks. In: FOCS, IEEE Computer Society (1981) 150–158

[67] Itai, A., Rodeh, M.: Symmetry breaking in distributed networks. Inf. Comput. **88**(1) (1990) 60–87

[68] Korman, A., Kutten, S., Peleg, D.: Proof labeling schemes. In Aguilera, M.K., Aspnes, J., eds.: PODC, ACM (2005) 9–18

[69] Korman, A., Kutten, S., Peleg, D.: Proof labeling schemes. Distributed Computing **22**(4) (2010) 215–233

[70] Korman, A., Sereni, J.S., Viennot, L.: Toward more localized local algorithms: removing assumptions concerning global knowledge. In: Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing. PODC '11, New York, NY, USA, ACM (2011) 49–58

[71] Krumke, S.O., Marathe, M.V., Ravi, S.S.: Models and approximation algorithms for channel assignment in radio networks. Wireless Networks **7**(6) (2001) 575–584

[72] Kuhn, F.: Weak graph colorings: distributed algorithms and applications. In: SPAA. (2009)

[73] Kuhn, F., Moscibroda, T., Wattenhofer, R.: What cannot be computed locally! [39] 300–309

[74] Leighton, F.T.: Finite common coverings of graphs. J. Comb. Theory, Ser. B **33**(3) (1982) 231–238

[75] Lenzen, C., Suomela, J., Wattenhofer, R.: Local algorithms: Self-stabilization on speed. In Guerraoui, R., Petit, F., eds.: SSS. Volume 5873 of Lecture Notes in Computer Science., Springer (2009) 17–34

[76] Linial, N.: Locality in distributed graph algorithms. SIAM Journal on Computing **21**(1) (1992) 193–201

[77] Liu, X., Xu, Z., Pan, J.: Classifying rendezvous tasks of arbitrary dimension. Theoretical Computer Science **410**(21–23) (2009) 2162 – 2173

[78] Luby, M.: A simple parallel algorithm for the maximal independent set problem. In Sedgewick, R., ed.: STOC, ACM (1985) 1–10

[79] Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1996)

[80] Mavronicolas, M., Michael, L., Spirakis, P.: Computing on a partially eponymous ring. In: Proceedings of the 10th international conference on Principles of Distributed Systems. OPODIS'06, Berlin, Heidelberg, Springer-Verlag (2006) 380–394

[81] McCormick, S.T.: Optimal approximation of sparse hessians and its equivalence to a graph coloring problem. Mathematical Programming **26**(2) (1983) 153–171

[82] Métivier, Y., Robson, J.M., Zemmari, A.: Analysis of fully distributed splitting and naming probabilistic procedures and applications - (extended abstract). In Moscibroda, T., Rescigno, A.A., eds.: SIROCCO. Volume 8179 of Lecture Notes in Computer Science., Springer (2013) 153–164

[83] Métivier, Y., Saheb, N., Zemmari, A.: Randomized local elections. Inf. Process. Lett. **82**(6) (2002) 313–320

[84] Naor, M., Stockmeyer, L.: What can be computed locally? SIAM Journal on Computing **24**(6) (1995) 1259–1277

[85] Naor, M., Stockmeyer, L.J.: What can be computed locally? In: STOC. (1993) 184–193

[86] Norris, N.: Universal covers of graphs: Isomorphism to depth $n-1$ implies isomorphism to all depths. Discrete Applied Mathematics **56**(1) (1995) 61–74

[87] Ramanathan, M.K., Ferreira, R.A., Jagannathan, S., Grama, A., Szpankowski, W.: Randomized leader election. Distributed Computing **19**(5-6) (2007) 403–418

[88] Schieber, B.: Calling names in nameless networks. In Rudnicki, P., ed.: PODC, ACM (1989) 319–328

[89] Schmid, S., Suomela, J.: Exploiting locality in distributed SDN control. In: HotSDN. (2013) 121–126

[90] Schneider, J., Elkin, M., Wattenhofer, R.: Symmetry breaking depending on the chromatic number or the neighborhood growth. Theor. Comput. Sci. **509** (2013) 40–50

[91] Schneider, J., Wattenhofer, R.: A log-star distributed maximal independent set algorithm for growth-bounded graphs. In: PODC. (2008)

[92] Scott, A., Jeavons, P., Xu, L.: Feedback from nature: an optimal distributed algorithm for maximal independent set selection. In: PODC. (2013)

[93] Shukla, S.K., Rosenkrantz, D.J., Ravi, S.S.: Developing self-stabilizing coloring algorithms via systematic randomization. In: Proceedings of the International Workshop on Parallel Processing. (1994)

[94] Suomela, J.: Survey of local algorithms. ACM Comput. Surv. (to appear) Preliminary version.

[95] Yamashita, M., Kameda, T.: Computing on anonymous networks: Part i-characterizing the solvable cases. IEEE Trans. Parallel Distrib. Syst. **7**(1) (January 1996) 69–89

Curriculum Vitae

May 18$^{\text{th}}$, 1984	Born in Recklinghausen, Germany
1994–2003	High school "Gymnasium Petrinum" Recklinghausen, Germany
2003–2004	Civil service
2004–2011	Studies in computer science Karlsruhe Institute of Technology, Germany
2005–2008	Student assistant Karlsruhe Institute of Technology, Germany
May 2011	Dipl.-Inform. (roughly comparable to M.Sc.) Supervisor: Prof. Peter Sanders
2011–2015	Ph.D. student, research and teaching assistant Distributed Computing Group Prof. Roger Wattenhofer, ETH Zürich, Switzerland
August 2015	Ph.D. degree Supervisor: Prof. Roger Wattenhofer Co-Examiner: Prof. Yuval Emek, Technion, Israel Co-Examiner: Prof. Jukka Suomela, Aalto University, Finland

Publications

During my time at ETH, I co-authored the following publications. Note that the authors are ordered alphabetically. The order does not reflect the amount of contribution.

- Brandt, S., Seidel, J., Wattenhofer, R.: Toehold DNA Languages are Regular. In: ISAAC. (2015)
- Decker, C., Guthrie, J., Seidel, J., Wattenhofer, R.: Making Bitcoin Exchanges Transparent. In: ESORICS. (2015)
- Decker, C., Seidel, J., Wattenhofer, R.: Bitcoin Meets Strong Consistency. In: ICDCN. (2016)
- Emek, Y., Pfister, C., Seidel, J., Wattenhofer, R.: Anonymous Networks: Randomization = 2-Hop Coloring. In: PODC. (2014)
- Emek, Y., Seidel, J., Wattenhofer, R.: Computability in Anonymous Networks: Revocable vs. Irrevocable Outputs. In: ICALP. (2014)
- Förster, K.T., Luedi, T., Seidel, J., Wattenhofer, R.: Local Checkability, No Strings Attached. In: ICDCN. (2016)
- Förster, K.T., Seidel, J., Wattenhofer, R.: Deterministic Leader Election in Multi-Hop Beeping Networks. In: DISC. (2014)
- Seidel, J., Uitto, J., Wattenhofer, R.: Randomness vs. Time in Anonymous Networks. In: DISC. (2015)