Department of Computer Science

Designing and finding very fast distributed algorithms for bounded-degree graphs

Jan Studený

Aalto University

DOCTORAL THESES

Aalto University publication series
DOCTORAL THESES 219/2024

Designing and finding very fast distributed algorithms for bounded-degree graphs

Jan Studený

A doctoral thesis completed for the degree of Doctor of Science (Technology) to be defended, with the permission of the Aalto University School of Science, at a public examination held at the lecture hall F239a at Otakaari 3 on 25th of October 2024 at 12:00.

Aalto University
School of Science
Department of Computer Science
Distributed Algorithms

Supervising professor
Associate Professor Jukka Suomela, Aalto University, Finland

Thesis advisor
Associate Professor Jukka Suomela, Aalto University, Finland

Preliminary examiners
Associate Professor Peter Robinson, Augusta University, United States of America
Assistant Professor Peter Davies-Peck, Durham University, United Kingdom

Opponent
Associate Professor Seth Gilbert, National University of Singapore, Singapore

Aalto University publication series
DOCTORAL THESES 219/2024

ISBN 978-952-64-2076-9 (printed)
ISBN 978-952-64-2077-6 (pdf)
ISSN 1799-4934 (printed)
ISSN 1799-4942 (pdf)
http://urn.fi/URN:ISBN:978-952-64-2077-6

Unigrafia Oy
Helsinki 2024

Finland

Printed matter
4041-0619

Author
Jan Studený

Name of the doctoral thesis
Designing and finding very fast distributed algorithms for bounded-degree graphs

Publisher School of Science

Unit Department of Computer Science

Series Aalto University publication series DOCTORAL THESES 219/2024

Field of research Distributed Algorithms

Manuscript submitted 14 May 2024 **Date of the defence** 25 October 2024

Permission for public defence granted (date) 27 September 2024 **Language** English

☐ **Monograph** ☒ **Article thesis** ☐ **Essay thesis**

Abstract

Distributed computing studies models where computation is split between multiple interconnected entities which can communicate with each other to achieve a common goal. In this thesis we look at the setting where the communication network is sparse, i.e., it has a bounded degree.

We first study a family of problems that can be characterized by a fact that their solutions can be locally checked for correctness. This rich family is called locally checkable labeling (LCL) problems and contains, for example, the problems of computing proper colorings, maximal independent set, maximal matching and orientation problems. The aim is to completely characterize these LCL problems in a given network topology and be able to automatically synthesize an asymptotically optimal algorithm given any LCL problem. We have achieved a complete characterization and synthesis for the following graph families: paths, cycles and regular rooted trees and a partial characterization for regular undirected trees. The results are complemented by unified implementations of all classifiers for asymptotic round complexity. Some of the results are using a newly discovered connection to automata theory while others use a novel concept of certificates of solvability.

In the second part of the thesis we focus on a specific problem of sparse matrix multiplication in a setting of low-bandwidth communication. We show that in the case of uniformly sparse matrices where each row and column has at most d non-zeroes we can break the quadratic barrier with respect to d and obtain a faster algorithm. One of the techniques used is to iteratively find regions that are locally dense and use more efficient dense multiplication algorithms for computing those parts.

Keywords distributed algorithms; locally checkable labeling; matrix multiplication; locality, distributed computational complexity; algorithm synthesis; congested clique; low-bandwidth model

ISBN (printed) 978-952-64-2076-9 **ISBN (pdf)** 978-952-64-2077-6

ISSN (printed) 1799-4934 **ISSN (pdf)** 1799-4942

Location of publisher Helsinki **Location of printing** Helsinki **Year** 2024

Pages 150 **urn** http://urn.fi/URN:ISBN:978-952-64-2077-6

Preface

First I want to thank my supervisor Jukka Suomela for guiding me throughout the whole PhD. Thank you for suggesting the right problems to study, giving me multiple perspectives on how to view them, and actively promoting collaborative research which I really liked. Collaboration was one of the reasons why I chose to do a PhD in the first place. I think that an environment where people from different universities all over the globe can work together towards a common goal is something special.

Next, I want to thank all of my collaborators: Alkida Balliu, Sebastian Brandt, Yi-Jun Chang, Chetan Gupta, Juho Hirvonen, Janne H Korhonen, Dennis Olivetti and Jukka Suomela. From you I have learned how to do research and helped me to shape myself as a scientist. You have played an indispensable role in my PhD.

Many thanks to the whole Distributed Algorithms group for a supportive environment, openness in sharing research ideas, and having a large collaborative network.

I would like to sincerely thank my pre-examiners, Peter Robinson and Peter Davies-Peck, for their thoughtful insights and suggestions that improved my thesis.

I am very much thankful to Seth Gilbert for accepting the role of the opponent and for dedicating his time and effort to this important academic process.

I also want to thank Aalto University for providing me with all the resources I needed throughout my stay. I also thank the Research Council of Finland, Grant 333837 and Grant 321901 for supporting my work.

My belief in stronger PhD community led me to Aallonhuiput (Aalto Doctoral Researches Association). There I have met an amazing group of people and together we really strengthened our PhD community. Thank you Nidia, Tejas, Arttu, Saynaz, Marius, Heta, Dipesh, Fereshteh, Elham, Aanika, Rina, Felix, Saani and Christoph.

During my stay at Aalto I have formed an incredible circle of friends with whom I have enjoyed various activities, have had casual and deep discussions and who have helped me when I really needed them. Thank you Nidia, Tejas, Amirreza, Victoria, Mehdi, Huijia, Abbas, Sergei, Kunal, Suhas, Ali, Ronja, Kunal, Feri, Stella, Ioanna, Sorrachai, Saynaz, Sahel, Dipesh, Luis, Elham, Endrit, Ankita, Arttu, Halvar, Saghar, Farzin, Nazaal, Juan and more of you.

My final thank you belongs to my family that always supported me and encouraged me in pursuing my studies abroad even though it meant I will see them much less. Děkuji za všechno mamko, tatko, Marunko, dědo Slávku, dědo Jirko, babičko Aničko, babičko Miluško.

Espoo, 7th of October 2024,

Jan Studený

Contents

List of Publications

This thesis consists of an overview and of the following publications which are referred to in the text by their Roman numerals.

I Yi-Jun Chang, Jan Studený, Jukka Suomela. Distributed graph problems through an automata-theoretic lens. *Theoretical Computer Science*, 951 113710, January 2023.

II Alkida Balliu, Sebastian Brandt, Yi-Jun Chang, Dennis Olivetti, Jan Studený, Jukka Suomela, Aleksandr Tereshchenko. Locally checkable problems in rooted trees. *Distributed Computing*, 36(3) 277–311, September 2023.

III Alkida Balliu, Sebastian Brandt, Yi-Jun Chang, Dennis Olivetti, Jan Studený, Jukka Suomela. Efficient classification of locally checkable problems in regular trees. *36th International Symposium on Distributed Computing, DISC 2022, October 25–27, 2022, Augusta, Georgia, USA*, 246 8:1–8:19, October 2022.

IV Chetan Gupta, Juho Hirvonen, Janne H Korhonen, Jan Studený, Jukka Suomela. Sparse Matrix Multiplication in the Low-Bandwidth Model. *SPAA '22: 34th ACM Symposium on Parallelism in Algorithms and Architectures, Philadelphia, PA, USA, July 11–14, 2022*, 435–444, July 2022.

Author's Contributions

Publication I: "Distributed graph problems through an automata-theoretic lens"

This project started when the author and Yi-Jun Chang were visiting Aalto in 2019. Many of the results were discovered during the stay. The results were a product of collaborative effort; key elements are Jukka Suomela's discovery of the concept of mirror-flexibility and Yi-Jun Chang's hardness result for solvability. The author was responsible for writing mainly the technical part of the paper.

Publication II: "Locally checkable problems in rooted trees"

This publication was a natural continuation of publication I. The author came up with the concept of certificates that provide the main results. The observation about the constant case was done jointly by Alkida Balliu, Sebastian Brandt, and Dennis Olivetti. Yi-Jun Chang made the discovery of the polynomial region. The author wrote the majority of the paper together with Jukka Suomela and also wrote the implementation of the classifier together with Aleksandr Tereshchenko.

Publication III: "Efficient classification of locally checkable problems in regular trees"

This work is capturing an open question of publication II. Yi-Jun Chang is responsible for the key observation and a significant portion of the write-up. Precise formulations, algorithms and exposition was a joint effort by the author, Alkida Balliu, Sebastian Brandt, Dennis Olivetti and Jukka Suomela. The

author implemented the classifier and used the implementation to correct the write-up.

Publication IV: "Sparse Matrix Multiplication in the Low-Bandwidth Model"

This project started as a follow-up to a discussion with Chetan Gupta about parallel and distributed computing for Bayesian graphical models. The graph-theoretic results was a joint effort by the author, Juho Hirvonen and Jukka Suomela, and here a highlight is the author's work on the proof for finding a dense cluster that is a crucial element for the whole procedure. The optimal parameters and the algorithm for iterative clustering was developed by Chetan Gupta. The load balancing for the second phase of the algorithm was done by Janne H. Korhonen. Everyone contributed to writing the results.

1. Introduction

Distributed computing is an active area of study that encompasses any form of interconnected computational entities. Examples include drone colonies, data centers, supercomputers, sensor networks, and even swarms of bees or groups of people. In contrast with the study of centralized computing, which focuses on the computational perspective, the distributed computing focuses on communication. In this work we focus on synchronous message-passing setting in order to abstract away the complexities of asynchrony and faults which are separate areas of distributed computing in their own rights.

Two key metrics in distributed computing are locality and bandwidth, both of which are to be minimized.

Locality captures how far away a node needs to see to obtain enough information about its surrounding. An ideal model of computing when studying exclusively this metric is the LOCAL model of computing that was developed by Linial [Lin92]. In the LOCAL model, in each round we can send an arbitrarily large message to each direct neighbor. Canonical problems for the locality perspective are locally checkable labeling problems. These are problems whose solution can be locally checked for correctness. One of the problems that is locally checkable is graph coloring. Graph coloring is a problem where each node should output a color that has to be different from its neighbors. We can observe that the problem of 2-coloring is a global problem as a single node determines the colors for the rest of the graph (the colors are alternating in layers with respect to the distance from the first node). So each node needs to see the whole graph to correctly output its color with the furthest node potentially linear number of steps away (linear in the number of nodes in the graph). This can be contrasted with the sinkless orientation problem in trees where all nodes of degree at least 3 need to have at least one outgoing edge. To solve this problem, each node can search in its neighborhood for a closest node of degree 2 or 1 and point to it which is bound to happen in at most

logarithmic distance with respect to the number of nodes in the graph. Hence the problem is much more local.

Bandwidth captures how large and how many messages need to be sent. A model that captures both the locality and the bandwidth perspective is called the CONGEST model [Pel00]. In the CONGEST model, in each round a node can send only one $O(\log n)$ size message to each neighbor, where n is the total number of nodes in the network. Apart from the CONGEST model, there exists also the CONGESTED CLIQUE model [Lot+03], which drops the locality aspect as each node can now send an $O(\log n)$ message to each other node. This brings various advantages, for example for the above-mentioned problem of graph coloring, and more specifically $(\Delta + 1)$-coloring where Δ is the largest degree of any node. We have a lower bound of $\Omega(\log^* n)$ for the LOCAL model (hence also for CONGEST) [Lin92] but it has been shown that it takes only constant rounds in CONGESTED CLIQUE [CDP21]. One of the canonical problems where bandwidth is important is matrix multiplication. For matrix multiplication, we have the elements of both input matrices spread across multiple computers and we have to compute their product while respecting the bandwidth of each communication link.

1.1 Main contributions

In this thesis, we show multiple advancements in the characterization of LCL problems in the distributed setting and we uncover an area of subquadratic algorithms for very sparse matrix multiplication in a low-bandwidth communication setting.

To characterize LCL problems we will create a meta-algorithm that receives the description of an LCL problem and it determines whether the problem is solvable, what is its asymptotic locality and synthesizes an asymptotically optimal distributed algorithm. In Publication I we extend the work of [Bra+17] that considered only the case of directed cycles and show how to create such an algorithm for the case of cycles and paths both oriented and undirected and for a subset of problems on rooted regular trees. Our characterization uses a novel connection to automata-theory. In Publication II we create an almost complete characterization of LCL problems on rooted regular trees with the missing pieces in the polynomial regions of complexities. We use a novel concept of *certificates of solvability*. Finally in Publication III we solve the polynomial regions and provide a complete characterization of LCL problems in rooted regular trees. Moreover we extend the characterization of the LCL

problems in the polynomial and logarithmic region to undirected trees.

In the case of sparse matrix multiplication we consider the setting of uniformly sparse matrices where each row and column of each matrix has at most d non-zeroes and we are interested in at most d entries in each row and column of the product matrix. In the low-bandwidth communication setting we show in Publication IV that we can break the quadratic barrier with respect to d and obtain a faster algorithm. One of the techniques used is to iteratively find regions that are locally dense and use more efficient dense multiplication algorithms for computing those parts.

1.2 Roadmap

The thesis contains two main parts. The first one called LCL classification (Chapter 2) is about automatically characterizing LCL problems on specific graph families, namely paths, cycles, rooted and undirected regular trees. The chapter considers Publications I, II and III. We first establish important graph-theoretic concepts and models of computation in Section 2.1, followed by the original definition of the LCL problem and an alternative simplified definition in Section 2.2. After discussing the prior work in Section 2.3 we start presenting the key findings from Publication I, which uses automata-theoretic view to classify LCL problems on paths and cycles (Sections 2.4–2.8). Afterwards we refocus on classifying LCL problems on regular trees that required novel techniques which are described in Section 2.9: Certificates of solvability. Section 2.9.1 discusses key building blocks of Publications II and III in the area of classification of the polynomial region and the logarithmic round complexity. Sections 2.9.2 and 2.9.3 finalize the complete classification of rooted regular trees as they discuss the last two remaining regions which contain results from Publication II. Lastly, in Section 2.10 we describe a tool that incorporates all classifiers from Publications I, II and III to automatically classify any LCL problem of interest.

The second part focuses on sparse matrix multiplication in low-bandwidth setting (Chapter 3). The chapter considers Publication IV. In the first section we define the computational problem and the precise model of computing which we are using. Next, in Section 3.2 we review related work in centralized, parallel and distributed setting. After outlining applications of sparse matrix multiplication in Section 3.3 we describe the baseline solution for sparse matrix multiplication. Subsequently in Section 3.5 we express matrix multiplication in terms of counting triangles and give one of the key findings needed for a

faster algorithm. Then we outline the missing parts and construct a complete algorithm in Section 3.6. We close the matrix multiplication part by discussing future directions in Section 3.7.

2. LCL classification

In what follows we aim to create a meta-algorithm, an algorithm that would take as an input the description of an LCL problem and output both asymptotically optimal round complexity and create an asymptotically optimal distributed algorithm that solves the given LCL problem. Having such a meta-algorithm in full generality is impossible as for example it is undecidable if an LCL problem is constant-time solvable in non-toroidal grid graphs [NS95].

What the following sections show is a complete classification of LCL problems on paths, cycles and regular rooted trees and a partial classification for undirected regular trees.

2.1 Preliminaries

Definition 2.1.1 (Centered graph)**.** A pair (G, v) where G is a graph and v is vertex of G is called a centered graph.

Definition 2.1.2 (r radius ball around v)**.** Given a graph G and vertex v, the r-radius ball around v is the subgraph of G that contains v, all vertices that are at most distance r from v and all edges between those vertices.

Definition 2.1.3 (Set closed under permutations)**.** Let S be a set of tuples. S is *closed under permutations* if for all $(a_1, a_2, ..., a_k) \in S$ all permutations of $(a_1, a_2, ..., a_k)$ are also in S.

Next, we define a couple of distributed models. A distributed model consists of computers (nodes) that are connected via communication links (edges), and hence it can be represented as a graph. Usually we assume that the graph is simple. At the beginning of the computation, each node only knows how many communication links are attached to it (its degree) and how many computers are there in total. Each computer runs the same algorithm. Throughout the computing phase, computers communicate with each other via the links, can

perform arbitrarily complicated local computation or finish computing.

Definition 2.1.4 (PN model)**.** The Port Numbering model is a model of distributed computing. The communication network corresponds to the input graph. Each edge is connected to a vertex via a port. Vertex v has ports numbered $1, 2, \ldots, \deg(v)$, where $\deg(v)$ is the degree of vertex v. Each vertex represents a computer and runs the same algorithm. Communication proceeds in synchronous, error-free rounds, where each node can send an arbitrarily-sized message to each of their neighbors.

Definition 2.1.5 (LOCAL model [Lin92])**.** The LOCAL model is a model of distributed computing. Communication network corresponds to an input graph. Each node has a polynomially-sized unique identifier (polynomial in the number of nodes in the network). Each node also knows the number of nodes in the network (denoted by n) and a maximal degree of the input graph (denoted by Δ). Each node represents a computer and runs the same algorithm. Communication proceeds in synchronous, error-free rounds, where each node can send an arbitrarily-sized message to each of their neighbors.

Definition 2.1.6 (RandLOCAL)**.** Randomized LOCAL model (RandLOCAL) is a modification of a LOCAL model, where instead of unique identifiers, each node has an unbounded stream of random bits.

Definition 2.1.7 (CONGEST [Pel00])**.** The CONGEST model is a variation of the LOCAL model, where each node, instead of arbitrarily sized message can send a message of size $O(\log n)$ bits to each neighbor.

Definition 2.1.8 (RandCONGEST)**.** Randomized CONGEST model (RandCONGEST) is a modification of a CONGEST model, where instead of unique identifiers, each node has an unbounded stream of random bits.

2.2 LCL definition

Locally checkable labeling problems (LCLs) are problems whose solution can be checked locally for correctness. This means that if the local neighborhood of each node is correct, it is also correct globally. The formal definition of LCL problems was introduced by Naor and Stockmeyer in 1995 [NS95]. LCLs represent a rich class of problems; examples are maximal matching, maximal independent set, vertex coloring, sinkless orientation and more.

Definition 2.2.1 (LCL problem)**.** LCL problem Π consists of $(r, \Sigma, \Gamma, \mathcal{C})$ where

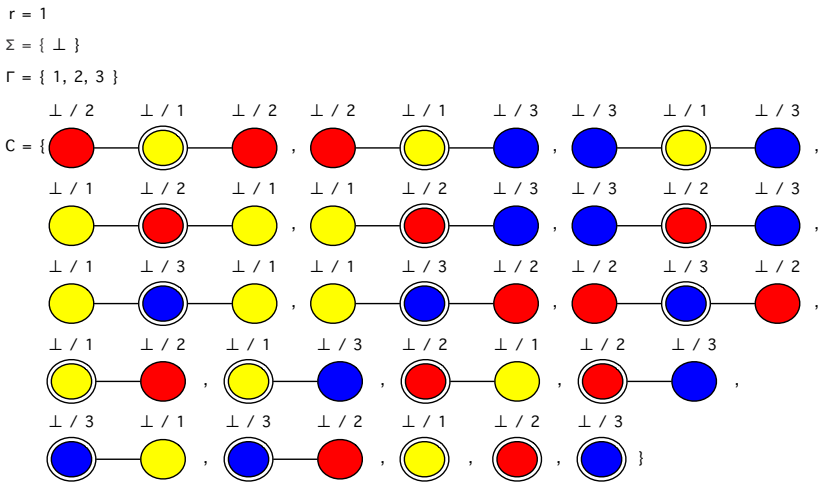


Figure 2.1. Example of how to encode an LCL problem that represents 3-coloring of a path graph. Nodes of the centered graphs are labeled according to the output color only as the input label is the same for all nodes. Center of the graph is denoted by the double outline. Text above the nodes is textual representation of input color / output color.

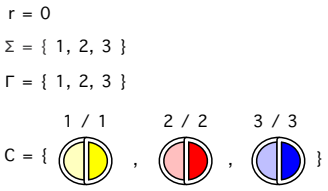


Figure 2.2. Example of how to encode an LCL problem that represents a copying task.

- r is a checkability radius
- Σ is a finite set of input labels
- Γ is a finite set of output labels
- $\mathcal{C}$ is a list of labeled centered graphs of radius r which represents allowed neighborhoods.

For a better understanding of the definition, let us consider two examples of LCL problems. The first problem (Figure 2.1) represents a proper coloring of path graphs. As coloring does not have any input, the input is just a dummy symbol $\perp$. The second problem (Figure 2.2) is a simple copying task from input labeling to output labeling.

Definition 2.2.2 (Solution of an LCL problem)**.** Let $\Pi = (r, \Sigma, \Gamma, \mathcal{C})$ be an LCL problem. For a graph $G = (V, E)$ with input labels $\lambda_i : V \to \Sigma$ a labeling $\lambda_o : V \to \Gamma$ is a solution if for each $v \in V$ the labeled r radius ball around v is isomorphic to a graph in $\mathcal{C}$.

Another way to define LCL problems is by using the LOCAL model.

Definition 2.2.3 (LCL problem via LOCAL model)**.** Problem Π is an LCL problem if there exists a LOCAL algorithm $\mathcal{A}$ that can verify the solution such that

1. $\mathcal{A}$ runs in $O(1)$ rounds.
2. $\mathcal{A}$ is independent of unique identifiers (change of identifiers should not affect verification output).

We obtain a simpler class of problems if we use the PN model instead of the LOCAL model.

Definition 2.2.4 (PN-checkable labeling via PN model)**.** Problem Π is a PN-checkable labeling problem if there exists a PN algorithm $\mathcal{A}$ that can verify the solution such that

1. $\mathcal{A}$ runs in $O(1)$ rounds.
2. $\mathcal{A}$ is independent of port numbers (change of port numbers should not affect verification output).

Many interesting LCL problems are also PN-checkable labelings: coloring problems, matching problems, orientation problems and more. An example of a problem that is *not* PN-checkable but is an LCL problem is to label a node black if it is part of a triangle and white otherwise.

If we wanted a graph-theoretic definition of PN-checkable problems, the configurations would consists of centered trees, namely:

Definition 2.2.5 (PN-checkable labeling)**.** PN-checkable labeling problem Π consists of $(r, \Sigma, \Gamma, \mathcal{C})$ where

- r is a checkability radius.
- Σ is a finite set of input labels.
- Γ is a finite set of output labels.
- $\mathcal{C}$ is a list of labeled centered trees of radius r which represents allowed neighborhoods.

Now we need to define how to map general graphs into trees for verification. We will use universal covering graphs [Ang80].

Definition 2.2.6 (Universal covering graph [Ang80])**.** The universal covering graph T of G is constructed as follows. Let v be a node of G. Each vertex of T is a non-backtracking walk (cannot visit vertex again along the edge it has just traversed) in G starting at node v. Vertices u, w are connected by an edge if one is one-step extension of the other (as walks).

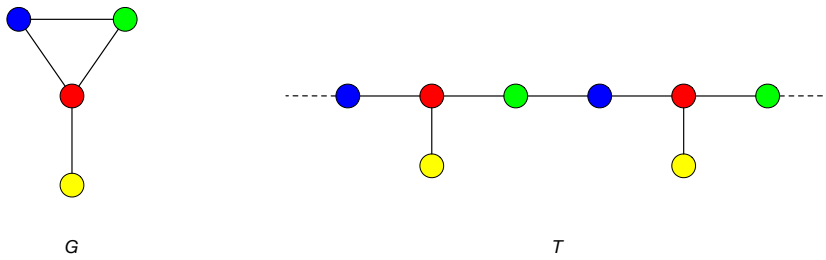


Figure 2.3. Example of a graph G and its universal covering graph T. Nodes of T are colored using the universal covering map.

The universal covering map maps vertex v from T (which is a walk) to its last visited vertex in G. An example of an universal covering graph together with an universal covering map can be seen in Figure 2.3.

The solution to a PN-checkable labeling is defined as follows.

Definition 2.2.7 (Solution of an PN-checkable labeling)**.** Given a PN-checkable labeling $\Pi = (r, \Sigma, \Gamma, \mathcal{C})$, a graph $G = (V, E)$ with input labels $\lambda_i : V \to \Sigma$, universal covering graph T of G and covering map f, a labeling $\lambda_o : V \to \Gamma$ is a solution if for each $v \in T$, the labeled r radius ball around v with labels mapped using covering map f is isomorphic to a graph in $\mathcal{C}$.

A useful characteristic of the PN-checkable labeling is that we can reduce the checkability radius by requiring nodes to output labels for its whole neighborhood. Therefore, we can conveniently define a problem using the node-edge-checkable formalism. In this formalism, each edge is split into two halves and the task is to label those half-edges. Then to define a problem, we need to describe all allowed configurations around each node, and for each edge. For simplicity, the definition will not consider input labeling as the input labels are not used in the context where we use such formalism.

Definition 2.2.8 (Node-edge-checkable formalism)**.** Problem Π in node-edge-checkable formalism for graphs of maximum degree Δ consists of:

1. Σ, which is a finite set of output labels.
2. $C_V \subseteq (\Sigma \cup \Sigma^2 \cdots \cup \Sigma^\Delta)$ closed under permutation, which is a set of allowed configurations over a node.
3. $C_E \subseteq \Sigma^2$ closed under permutation, which is a set of allowed configurations over an edge.

In the case of trees, cycles or paths, all definitions (locally checkable labeling, PN-checkable labeling and node-edge-checkable formalism) turn out to be equivalent to each other up to behavior on constant-size graphs, and in this case we can use the node-edge-checkable formalism for any LCL problem.

Moreover when working with oriented graphs, LCL problems can refer to the orientation. In the accompanying work, we focus on consistently oriented paths, consistently oriented cycles and rooted regular trees without boundary conditions. We do not allow boundary conditions as they could be used to emulate an oriented path with input labels whose decidability is provably hard [Bal+19].

For those cases, we define the node-edge-checkable formalism as following.

Definition 2.2.9 (Node-edge-checkable formalism for a consistently oriented path)**.** Problem Π in node-edge-checkable formalism for a consistently oriented path consists of:

1. Σ, which is a finite set of output labels.
2. $C_V \subseteq \Sigma^2$, which is a set of allowed configurations over an inner node.
3. $C_S \subseteq \Sigma$, which is a set of allowed configurations for a node that has only one outgoing edge.
4. $C_F \subseteq \Sigma$, which is a set of allowed configurations for a node that has only one incoming edge.
5. $C_E \subseteq \Sigma^2$, which is a set of allowed configurations over an edge.

Definition 2.2.10 (Node-edge-checkable formalism for a consistently oriented cycle)**.** Problem Π in node-edge-checkable formalism for a consistently oriented cycle consists of:

1. Σ, which is a finite set of output labels.
2. $C_V \subseteq \Sigma^2$, which is a set of allowed configurations over an inner node.
3. $C_E \subseteq \Sigma^2$, which is a set of allowed configurations over an edge.

Definition 2.2.11 (Node-edge-checkable formalism for a rooted d-ary tree without boundary conditions)**.** Problem Π in node-edge-checkable formalism for a rooted d-ary tree without boundary conditions consists of:

1. Σ, which is a finite set of output labels.
2. $C_V \subseteq \Sigma \times \Sigma^d$ where the last d elements are closed under permutation, which is a set of allowed configurations from a parent label to children labels.
3. $C_E \subseteq \Sigma^2$, which is a set of allowed configurations over an edge.

For the case of rooted trees we can show that having allowed configuration over an edge and splitting them is redundant, hence we can refer to LCL problems on rooted trees without the edge configurations. For configurations over the nodes, we can write them with a colon that signifies the parent label, i.e. $(a : b_1, \ldots, b_d)$.

2.3 Prior work

Prior work can be divided based on the type of the underlying graphs. The simplest graph families are paths and cycles. There, possible complexities of LCL problems are $\Theta(1)$, $\Theta(\log^* n)$ and $\Theta(n)$. These complexities are a result of the two speedup theorems: One speeds up any $o(\log^* n)$ problem to $O(1)$ (implication of [NS95] mentioned by [CP19] in Appendix A). The other speeds up $o(n)$ problems to $O(\log^* n)$ by [CKP19, Theorem 7].

For decidability, [Bra+17, Claim 1] studied the case of oriented cycles without input labels. They showed a polynomial-time algorithm (polynomial in the problem description) that classifies the complexity of a given problem. Then [Bal+19, Theorems 8, 9] proved that classifying the complexity of problems on paths with input labels is decidable, but PSPACE-hard (Theorem 5) for both undirected and oriented paths. Therefore efficient decidability of undirected paths or cycles or oriented paths was an open question which was closed in our work by Publication I and is described in the sections directly following the prior work.

Next, we have 2-dimensional grids. The possible deterministic complexities are the same as in paths and cycles [Bra+17]. But it is now undecidable to determine the right complexity of a problem [Bra+17, Theorem 3; NS95].

Another graph family is trees. There are problems of complexities $O(1)$, $\Theta(\log^* n)$, $\Theta(\log \log n)$, $\Theta(\log n)$, $\Theta(n^{1/k})$ for any integer $k \geq 1$ but it is not known if this list is exhaustive. For the constituting gaps, there are gaps between $\omega(n^{1/k})$ and $o(n^{1/(k-1)})$ by [Cha20; Bal+21], between $n^{o(1)}$ and $\omega(\log n)$ by [CP19], between $o(\log n)$ and $\omega(\log^* n)$ for deterministic algorithms by [CKP19], between $o(\log n)$ and $\omega(\text{poly}(\log \log n))$ by combining results of [CP19], [FG17] and [RG20], between $o(\log \log n)$ and $\omega(\log^* n)$ for randomized algorithms [CKP19] and $o(\log \log^* n)$ and $\omega(1)$ by [CP19, Appendix A]. Therefore, open regions in the complexity landscape are between $o(\log^* n)$ and $\omega(\log \log^* n)$ and between $o(\text{poly}(\log \log n))$ and $\omega(\log \log n)$ for randomized algorithms.

For decidability, we can distinguish between $O(\log n)$ and $n^{\Omega(1)}$ using the algorithm from [CP19] and then between each $O(n^{1/k})$ and $\Omega(n^{1/(k-1)})$ using the algorithm from [Cha20]. Note that the deciding algorithm in [CP19] only proves that the runtime of the algorithm is bounded and the algorithm in [Cha20] is doubly exponential.

What we show is that the runtime for an algorithm for distinguishing between the above-mentioned regions in regular trees is polynomial. Moreover, for the class of rooted regular trees we show that there are only problems of complexities

$O(1)$, $\Theta(\log^* n)$, $\Theta(\log n)$, $\Theta(n^{1/k})$ for any integer $k \geq 1$, neither randomness nor large messages help and we have an algorithm that decides between any two classes in at most exponential time. These results come from Publications II and III.

2.4 Finite automata

This part will show the connection between LCL problems and automata theory. The high-level goal is to use this connection to be able to automatically classify LCL problems on paths and cycles.

First, let us define what a non-deterministic finite automaton is.

Definition 2.4.1. A non-deterministic finite automaton is a tuple $\mathcal{M} = (\Sigma, S, S_0, \delta, F)$ where:

- Σ is a finite input alphabet of symbols,
- S is a set of states that the automaton has,
- S_0 is a set of initial states,
- δ is a transition funcion ($\delta : S \times \Sigma \to 2^S$),
- F is a set of final states.

In our case, we would be dealing with a special kind of automaton that has an alphabet of size one.

Definition 2.4.2 (Unary automaton)**.** A non-deterministic finite automaton is called unary if the size of the alphabet is one.

When dealing with unary automata, we use the symbol o to denote the only symbol of the alphabet.

Intuitively, with unary automata, we are only capturing the length of the word. Moreover, as there is only one word of each length, existential and universal quantifiers that talk about a word of specific length collapse. This makes checking a lot of the properties much easier.

A property of automata that will have a direct connection to LCL complexity is directability, namely D3-directability. It was first mentioned in [IS99] and is defined as follows.

Definition 2.4.3 (D3-directability)**.** An automaton $\mathcal{M}$ is D3-directable if there exists a word w such that there exists a state c that can be reached from any state of $\mathcal{M}$ after reading w.

A few examples of automata together with the description of why they are (or they are not) D3-directable can be found in Figure 2.4.

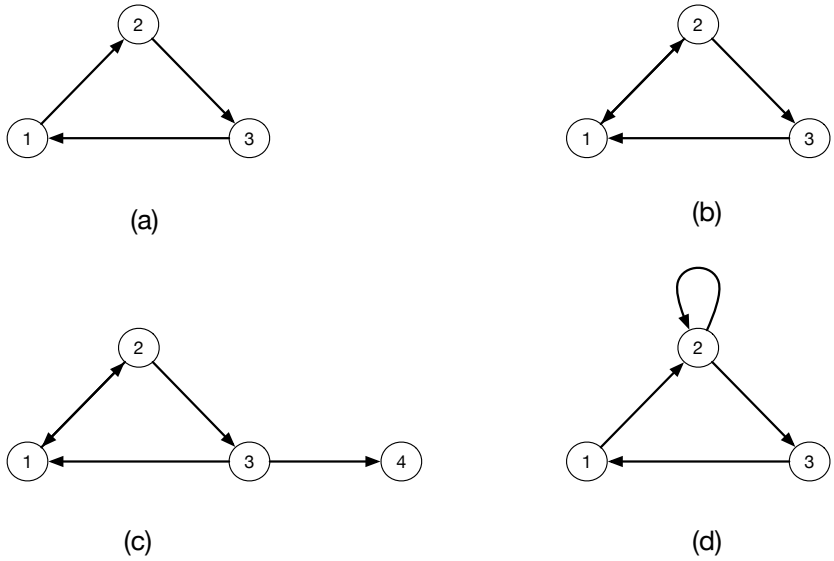


Figure 2.4. Examples of unary automata and their D3-directability. (a) Not D3-directable as for any two different states, after reading a symbol, we move clockwise in the diagram to two different states, never the same ones. (b) Is D3-directable as we have a word $w = ooooo$ and state $c = 1$ such that for all states we can go in five steps back to state 1 ($1 \to 2 \to 1 \to 2 \to 3 \to 1$; $2 \to 3 \to 1 \to 2 \to 3 \to 1$; $3 \to 1 \to 2 \to 1 \to 2 \to 1$). The reason is that the graph is strongly connected and there are two cycles of coprime lengths. (c) Is not D3-directable. Even though when starting from any of the three vertices 1, 2 or 3 we can get with sufficiently long word anywhere, when starting from vertex 4 we have nowhere to go. (d) Is D3-directable for similar reason as the case (b), here we have a word $w = oo$ and state $c = 2$ such that from all states we can go in two steps to state 2.

2.5 Transformation of LCL problems to automata

Now we show how to transform an LCL problem into an automaton whose analysis will determine solvability and round complexity.

We first define an automaton from LCL problem on cycles.

Definition 2.5.1 (Automaton from an LCL problem on cycles)**.** Let $\Pi = (\Sigma, C_V, C_E)$ be an LCL problem on cycles. Automaton $\mathcal{M}_\Pi$ is defined as follows.

1. The set of symbols is $\{o\}$.
2. The set of states is C_E.
3. There is a transition from a state (a, b) to a state (c, d) if $(b, c) \in C_V$.

Note that for cycles we do not have any starting state or final state, hence the automaton will transform to a non-deterministic semiautomaton (automaton having inputs but no output). For paths we have explicit constraints for the endpoints, hence the automaton will be defined as follows.

Definition 2.5.2 (Automaton from an LCL problem on paths)**.** Let $\Pi = (\Sigma, C_V, C_S, C_F, C_E)$ be an LCL problem on oriented paths or let $\Pi = (\Sigma, C_V, C_E)$ be an LCL problem for undirected paths. Automaton $\mathcal{M}_\Pi$ is defined as follows.

1. The set of symbols is $\{o\}$.
2. The set of states is C_E.
3. There is a transition from a state (a, b) to a state (c, d) if $(b, c) \in C_V$.
4. State (a, b) is a starting state if $a \in C_S$ or $a \in C_V$.
5. State (a, b) is a final state if $b \in C_F$ or $b \in C_V$.

To see how the automaton operates, we define how to generate a labeled path or cycle.

Definition 2.5.3 (Generating labeled path or cycle)**.** Let Π be an LCL problem and $\mathcal{M}_\Pi$ the corresponding automaton. A cycle $(a_1, a_2), (a_3, a_4), \ldots, (a_{2m-1}, a_{2m})$ is generated by $\mathcal{M}_\Pi$ if all (a_{2i-1}, a_{2i}) are states of $\mathcal{M}_\Pi$ for $i \leq m$, there is a transitions from (a_{2i-3}, a_{2i-2}) to (a_{2i-1}, a_{2i}) for all $i < m$ and there is a state transition from (a_{2m-1}, a_{2m}) to (a_1, a_2).

A path $(a_1, a_2), (a_3, a_4), \ldots, (a_{2m-1}, a_{2m})$ is generated by $\mathcal{M}_\Pi$ if all (a_{2i-1}, a_{2i}) are states of $\mathcal{M}_\Pi$ for $i \leq m$, (a_1, a_2) is a starting state, (a_{2m-1}, a_{2m}) is a final state and there is a transitions from (a_{2i-3}, a_{2i-2}) to (a_{2i-1}, a_{2i}) for all $i < m$.

We allow setting $m = 1$ in the definition.

2.6 Properties of automata

Now we define some properties of automata that will be useful when deciding the complexity of corresponding LCL problem. In the definitions, by a walk from state (a, b) to state (c, d) (denoted as $(a, b) \rightsquigarrow (c, d)$), we mean a sequence of transitions starting at (a, b) and ending at (c, d). Let Π be an LCL problem and $\mathcal{M}_\Pi$ the corresponding automaton. For problems that use the orientation of edges we define:

Definition 2.6.1 (Repeatable state)**.** A state $(a, b) \in S$ is repeatable if there exists a walk from (a, b) to (a, b).

Definition 2.6.2 (Flexible state)**.** A state $(a, b) \in S$ is flexible with flexibility k if k is the smallest number such that for all $l \geq k$ there is a walk from (a, b) to (a, b) of length l.

Definition 2.6.3 (Loop)**.** A state $(a, b) \in S$ is a loop if there is a transition from (a, b) to (a, b).

For problems that do not refer to orientation we define:

Definition 2.6.4 (Mirror-flexible state)**.** A state $(a, b) \in S$ is mirror-flexible with flexibility k if k is the smallest number such that for all $l \geq k$ there are walks $(a, b) \rightsquigarrow (a, b)$, $(a, b) \rightsquigarrow (b, a)$, $(b, a) \rightsquigarrow (a, b)$ and $(b, a) \rightsquigarrow (b, a)$ of lengths exactly l.

Definition 2.6.5 (Mirror-flexible loop)**.** A state $(a, b) \in S$ is a mirror-flexible loop if (a, b) is a mirror-flexible state and there is a transition from (a, b) to (a, b).

Deciding if a state has a given property can be computed in polynomial time with respect to the size of the automaton. The non-trivial case of flexibility and mirror-flexibility consists of finding two returning walks of coprime lengths. For an automaton with q states, we showed that it is enough to consider only walks whose length is at most $2q - 1$. More details can be found in Section 5 of Publication I.

2.7 Flexible states vs directability

Now let us connect the property of the automata with the concept of directability.

Theorem 2.7.1. *Let Π be an LCL problem and $\mathcal{M}_\Pi$ the corresponding automaton. There exists a flexible state in $\mathcal{M}_\Pi$ if and only if there is a D3-directing word in $\mathcal{M}_\Pi$ restricted to a strongly connected component.*

Proof. Let us start with the forward implication. Let C be the strongly connected component where the flexible state s is. For any $v \in C$, we can build a walk such that we go s in l_v steps. Note that because v is in a strongly connected component, there is always a walk to s in at most $|C|$ steps. After arriving to s which is a flexible state with flexibility k, we can use the flexibility and concatenate the walk with a returning walk to s in $k + (|C| - l_v)$ steps which brings the total length of the walk to $k + |C|$ that is independent of the starting vertex. Hence a word $o^{k+|C|}$ is a D3-directing word for $\mathcal{M}_\Pi$ restricted to C as for every state in C, after reading $o^{k+|C|}$ we can end up in a state s.

Now we prove the backward implication. If we have a D3-directing word o^m that leads to state s, for every $k \geq m$ we can start in s, arbitrarily walk somewhere for $k - m$ steps while staying in the strongly connected component and then from the resulting vertex use the directability to return to s in m steps. Hence s will be a flexible state. □

2.8 Results on paths and cycles

The classification of LCL problems using the automata properties is described by a Table 2.1. The results come from Publication I and are applicable for all 4 models (LOCAL, RandLOCAL, CONGEST, RandCONGEST). The table characterizes 11 classes of problems and for each class describes its solvability and round complexity. The classification of a problem and its respective class is solvable in polynomial time with respect to its description size.

To get an intuition on how certain automata properties enable faster round complexity, we will look at (mirror-)flexible states and mirror-flexible loops.

With (mirror-)flexible state s we can label a variable length path between two fixed nodes labeled by s. This enables the solution in the following manner. We compute in $O(\log^* n)$ rounds a (k, k)-ruling set in the line graph of the input graph[1] where k is the flexibility of s (flexibility is defined such that k is unique) using e.g. maximal independent set algorithm to the kth power of the line graph of the input graph. We label the delimiters by s and then we just use the property of (mirror-)flexible state to fill the constant size parts.

[1] (α, β)-ruling set is a set I of vertices such that every two vertices of I are at least α distance apart and each vertex is at most β distance far away from a closest vertex in I.

Type	A	B	C	D	E	F	G	H	I	J	K
closed under permutations	yes	yes	yes	no	yes	yes	no	yes	no	yes	no
repeatable state	yes	yes	yes	yes	yes	yes	yes	yes	yes	no	no
flexible state [Bra+17]	yes	yes	yes	yes	yes	yes	yes	no	no	no	no
loop [Bra+17]	yes	yes	yes	yes	no	no	no	no	no	no	no
mirror-flexible state	yes	yes	no	—	yes	no	—	no	—	no	—
mirror-flexible loop	yes	no	no	—	no	no	—	no	—	no	—

Number of instances:

0 = zero $<$ = finite ∞ = infinite ? = NP-complete to decide

· solvable cycles	∞	∞	∞	∞	∞	∞	∞	∞	∞	0	0
· solvable paths	∞	∞	∞	∞	∞	∞	∞	∞	∞	$<$	$<$
· unsolvable cycles	0	0	0	0	$<$	$<$	$<$	∞	∞	∞	∞
· unsolvable paths	$<$	$<$	$<$	$<$	$<$	$<$	$<$	?	?	∞	∞

Distributed round complexity:

$\square = O(1)$ $\boxplus = \Theta(\log^* n)$ $\blacksquare = \Theta(n)$ $\times$ = N/A

· directed cycles [Bra+17]	$\square$	$\square$	$\square$	$\square$	$\boxplus$	$\boxplus$	$\boxplus$	$\blacksquare$	$\blacksquare$	$\times$	$\times$
· directed paths	$\square$	$\square$	$\square$	$\square$	$\boxplus$	$\boxplus$	$\boxplus$	$\blacksquare$	$\blacksquare$	$\square$	$\square$
· undirected cycles	$\square$	$\boxplus$	$\blacksquare$	$\times$	$\boxplus$	$\blacksquare$	$\times$	$\blacksquare$	$\times$	$\times$	$\times$
· undirected paths	$\square$	$\boxplus$	$\blacksquare$	$\times$	$\boxplus$	$\blacksquare$	$\times$	$\blacksquare$	$\times$	$\square$	$\times$

Table 2.1. Classification of LCL problems in cycles and paths as seen in publication I. This table defines 11 types, labeled with A–K, based on six properties of the automaton representing the LCL problem. For each problem type, we show what is the number of solvable instances, the number of unsolvable instances, and the distributed round complexity for both directed and undirected paths and cycles. The cases marked with "×" refer to problems that are not well-defined or that are never solvable. For the cases labeled with "?" deciding the number of unsolvable instances is NP-complete (or co-NP-complete depending on the way one defines the decision problem). However, for all other cases the type directly determines both solvability, and all these cases are also decidable in polynomial time.

For the mirror-flexible loop s we use a slightly different strategy. We create in $O(1)$ time fragments of consistent orientation of length at least k where k is the flexibility of s. Then we label each fragment by s and where two fragments meet, we substitute the neighboring parts with a correct oriented transition.

2.9 Certificates of solvability

When moving onto LCL classification for richer class of problems, namely problems on rooted and undirected d-regular trees, we need new techniques to capture the right complexities. In the following subsections we define the concepts of certificates of solvability which certify that the problem can be solved in at most $O(n^{1/k})/O(\log n)/O(\log^* n)/O(1)$ rounds where k is any positive integer with the last two certificates restricted to only the rooted case. The results come from Publications II and III and are applicable for all 4 models (LOCAL, RandLOCAL, CONGEST, RandCONGEST) implying that in case of undirected tress neither randomness nor large messages help in the polynomial region and in the case of rooted tress randomness nor large messages help at all.

2.9.1 Certificate of $O(\log n)$ and $O(n^{1/k})$ solvability

The first from the series of certificates builds on top of the classification for paths and cycles. The certificates in this region exists for both rooted and undirected regular trees but we start by describing the rooted regular case. It uses the automata view to find and iteratively remove labels that would require long-distance coordination along a long path and labels that cannot be used on large enough instances. Then, if we are left with some labels that do not require such coordination we can show the problem is $O(\log n)$-round solvable. Otherwise, depending on how many iterations we need to eliminate all labels we categorize it in one of the $O(n^{1/k})$-round solvable problems. To formally define the certificate of $O(\log n)$ and $O(n^{1/k})$ solvability we need the following definitions.

Definition 2.9.1 (Path-form of an LCL problem)**.** Let $\Pi = (\Sigma, C_V)$ be an LCL problem on d-ary rooted trees. The *path-form* of Π, denoted by Π^{path} is a problem on oriented paths defined as follows. We transform each configuration $c \in C_V$ of form $c = (a : b_1, b_2, .., b_d)$ to d configurations of Π^{path}, $(a : b_i)$, for all $1 \leq i \leq d$.

Definition 2.9.2 (Path-flexibility)**.** Let Π be an LCL problem and Π^{path} its path-flexible form. A label $\sigma \in \Sigma_\Pi$ is *path-flexible* if σ is a flexible state in automaton $\mathcal{M}(\Pi^{\text{path}})$, and *path-inflexible* otherwise.

Definition 2.9.3 (Minimal absorbing subgraph)**.** Let G be a directed graph. A subgraph $G' \subseteq G$, is called a minimal absorbing subgraph if G' is a strongly connected component of G and G' does not have any outgoing edges.

Definition 2.9.4 (Trimming)**.** Given a subset $\tilde{\Sigma} \subseteq \Sigma$ of labels, we define $\mathsf{trim}(\tilde{\Sigma})$ as a set of all labels $\sigma \in \tilde{\Sigma}$ such that for each $i \geq 1$ it is possible to find a correct labeling of perfect d-ary tree of depth i such that the label of the root is σ and the labels of the remaining nodes are in $\tilde{\Sigma}$.

Definition 2.9.5 (Restriction)**.** Given an LCL problem $\Pi = (\Sigma, C_V)$, a restriction of Π to labels $\Sigma' \subseteq \Sigma$ is a new LCL problem $\Pi' = (\Sigma', C'_V)$ where C'_V is obtained by keeping all and only the configurations in C_V that only use labels in Σ'.

Definition 2.9.6 (Flexible-SCC)**.** Given any LCL problem $\Pi = (\Sigma, C_V)$ and any $\tilde{\Sigma} \subseteq \Sigma$, we define $\mathsf{flexible\text{-}SCC}(\tilde{\Sigma})$ as the set of all subsets $U \subseteq \tilde{\Sigma}$ that is a path-flexible strongly connected component of the automaton of the path-form of Π restricted to labels from $\tilde{\Sigma}$.

Currently there are two certificates of $O(\log n)$ and $O(n^{1/k})$ solvability. An older one coming from Publication II that has a loose bound in the polynomial region when no $O(\log n)$ certificate exists and can certify only $O(\log n)$ solvability, whereas the newer one coming from Publication III provides a tight bound that certifies also problems of the form $O(n^{1/k})$ where k is a positive integer.

Let us first introduce the older way of constructing the certificate.

1. Let Σ' be the set of path-flexible nodes of Π.
2. Let Π' be the restriction of Π to labels only from Σ'.
3. If $\Pi' \neq \Pi$ then replace Π by Π' and return to step 1.
4. If Π is empty then return ϵ as the problem does not have a certificate of $O(\log n)$ solvability.
5. Otherwise, let Σ' be the labels that induce a minimal absorbing subgraph of automaton $\mathcal{M}(\Pi)$.
6. Return Π' which is a restriction of Π to labels only from Σ'.

The intuition of why this is an $O(\log n)$ certificate is that we can use a rake-and-compress algorithm [MR85] to split the graph into long paths and their connections. The rake-and-compress algorithm works by alternating between a rake operation (which separates all leaf nodes) and a compress operation (which separates all long paths of only in-degree 1 nodes). This will take $O(\log n)$ rounds and is the crucial part of solving the problem. Then we iteratively use the flexibility of the labels to efficiently label the long paths regardless of the labels of their endpoints in the same way as when solving the problem on path in $O(\log^* n)$ rounds.

The algorithm for finding the certificate runs in polynomial time with respect to the LCL description length as we always remove at least one label and finding flexible labels takes only polynomial time.

The newer algorithm for finding the certificate coming from Publication III works as follows.

1. Let $\Sigma' = \mathsf{trim}(\Sigma)$. If Σ' is empty then the depth of the problem Π is 0 so return 0.
2. If $\mathsf{flexible\text{-}SCC}(\Sigma')$ contains only one strongly connected component that has every label from Σ' then return ∞.
3. Otherwise for each strongly connected component Σ'' in $\mathsf{flexible\text{-}SCC}(\Sigma')$ run the algorithm again with the problem Π restricted to labels from $\mathsf{trim}(\Sigma'')$ and note the achieved depths.
4. Return the largest depth (0 if no component) + 1.

The depth of the problem determines how fast the problem can be solved and the sequence of gradual restrictions of the labels acts as a certificate. That is, for a depth k, the problem is solvable in $O(n^{1/k})$ rounds and $O(\log n)$ rounds if $k = \infty$.

The solution for the polynomial complexities builds on top of modified rake-and-compress algorithm, where instead of alternating between rake and compress we do rake $n^{1/k}$ times followed by 1 compress operation. This brings down the number of layers from $O(\log n)$ to just k. To label the graph, we proceed in layers and label the nodes using the labels from appropriate depth.

The newer algorithm for finding the certificate also runs in polynomial time with respect to the LCL problem description.

Note on the undirected case

Algorithm for finding the certificate for undirected tress is almost the same as for the rooted case with the exception that instead of flexibility we use mirror-flexibility and instead of restricting the set of individual labels we alternate between restricting the set of allowed configurations over a node and the set of allowed configurations for the path-form of the problem. The certificate then consists of those restrictions.

Also the distributed algorithm is built in the same way as in the rooted case, using the (modified) rake and compress decomposition.

For more details see Section 5 of Publication III.

2.9.2 Certificate of $O(\log^* n)$ solvability

Whereas the previous certificates were building on top of the path case and used the automata to help with determining the complexity, the $O(\log^* n)$ certificate brings a novel view on how to see the problems that can be solved in $O(\log^* n)$ rounds. This certificate is applicable only for rooted regular trees.

The algorithm for solving any $O(\log^* n)$-round solvable problem can be described in the following way.

1. We split the input tree into a collection of perfect trees of height either α or β (except the trees touching the leaves of the original tree which can be smaller and root of each tree is a leaf of the tree above). The α and β are coprime constants that depend on the LCL description. This can be computed in $O(\log^* n)$ time.
2. Color the leaves of each tree in such a way that the rest of the tree has a valid labeling no matter the label of the root. This is done in parallel for each tree.
3. Fill the rest of each tree given the root label which comes from the tree above. This is also done in parallel.

As the trees are of constant depth, both steps 2. and 3. take only constant time so the whole procedure takes $O(\log^* n)$ time.

The crucial thing in the mentioned procedure is to find the leaf-labeled tree that can have any label on the root. This is exactly what the $O(\log^* n)$ certificate will provide.

Definition 2.9.7 (Certificate for $O(\log^* n)$ solvability)**.** Let Π be an LCL problem. A certificate of $O(\log^* n)$ solvability for $\Pi = (\Sigma, C_V)$ with labels $\Sigma_{\mathcal{T}} = \{\sigma_0, \ldots, \sigma_t\} \subseteq \Sigma$ and depth k is a sequence $\mathcal{T}$ of t labeled trees (denoted by $\mathcal{T}_i$) such that:

1. Each tree is a perfect d-ary tree of depth k (k has to be at least one).
2. Each tree $\mathcal{T}_i$ is labeled by labels from $\Sigma_{\mathcal{T}}$ and correct with respect to configurations C_V.
3. Let $\overline{\mathcal{T}}_i$ be the tree obtained by starting from $\mathcal{T}_i$ and removing the labels of all non-leaf nodes. It must hold that all trees $\overline{\mathcal{T}}_i$ are isomorphic, preserving the labeling.
4. Root of tree $\mathcal{T}_i$ is labeled with label σ_i.

To get an idea how the certificate for $O(\log^* n)$ solvability can be found we describe the high level view of the algorithm for finding it. For every possible

non-empty subset of labels $\Sigma' \subseteq \Sigma$, we take an LCL problem restricted to Σ'. For each restriction, we will be building trees where each node is labeled by a set of labels where leaf nodes are having the set size of one. We start with the constraints, which can be seen as trees of depth one that will have multiple root labels when some constraints have the same children labels but different root labels (e.g. if $A : B, B$ and $C : B, B$ are in the constraints then the root of one of the depth one tree will be labeled by A, C). Then for every choice of d trees (we can take one tree multiple times), we try to combine them to obtain a larger tree with a root label that has all labels that can be created by a combination of all root labels of d trees according to the configurations of LCL problem (hence if we have two trees with root labels A, B and B, C and configurations $A : A, C$, $C : A, B$ and then we can create a larger tree with a root label A, C). For each possible root label we keep at most one representative and finish when we cannot combine the trees that would give a new root label. Then if there exists a tree with root label Σ' we return such tree, otherwise there is no certificate for $O(\log^* n)$ solvability. More detailed description can be found in Publication II (Algorithm 4).

The mentioned procedure finishes in exponential time as we search for each subset of labels (at most $2^{|\Sigma|}$ subsets) and in each step we enlarge the number of trees by at least one (of which there can be at most $2^{|\Sigma|}$ as they have different root labels).

2.9.3 Certificate of $O(1)$ solvability

Certificate of $O(1)$ solvability is surprisingly similar to the certificate of $O(\log^* n)$ solvability. We additionally need a special configuration that consists of label that can follow itself and such label should be contained as one of the leaves for the certificate of $O(\log^* n)$ solvability. With just these two additions we can show that the problem can be solved in constant time. This certificate is also applicable only for rooted regular trees.

Definition 2.9.8 (Certificate for $O(1)$ solvability)**.** Let $\Pi = (\Sigma, C_V)$ be an LCL problem. A certificate for $O(1)$ solvability for problem Π is a pair $\mathcal{S}$ consisting of a certificate for $O(\log^* n)$ solvability $\mathcal{T}$ and a configuration $(a : b_1, \ldots, a, \ldots, b_d) \in C_V$ where $a, b_i \in \Sigma_{\mathcal{T}}$ and at least one leaf of the trees in $\mathcal{T}$ is labeled a.

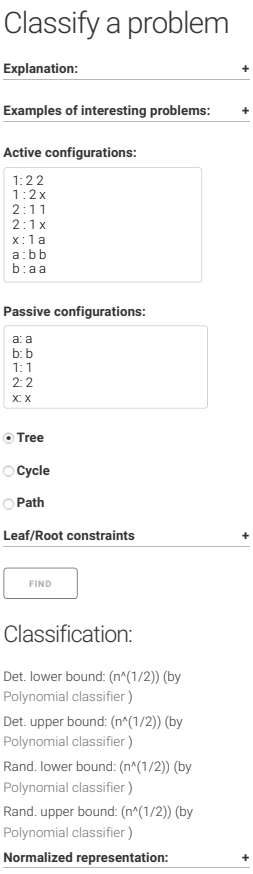
The high level idea for solving the problem given the certificate in constant time is as follows. We are creating a defective distance-k coloring, fixing the defects using the special configuration which will create regions of correct

distance-k coloring that will be used to speed up the solution from $O(\log^* n)$ time to constant time using the $O(\log^* n)$ certificate and using the correct leaf label to put the parts together. We first order children under every node by their identifiers. Then, we label each node by a sequence of child positions when going up towards to root of some constant but large enough length. Then labels that are periodic (such as 1212121... or 112112112...) will form paths that can be labeled by the special configuration. For labels that are not periodic, we can show that they form distance-k coloring and we can proceed as in the $O(\log^* n)$ case by splitting those regions to trees, filling the trees and for every start of a periodic path we need to order the leaves of the upper tree such that we connect the right certificate leaf to it.

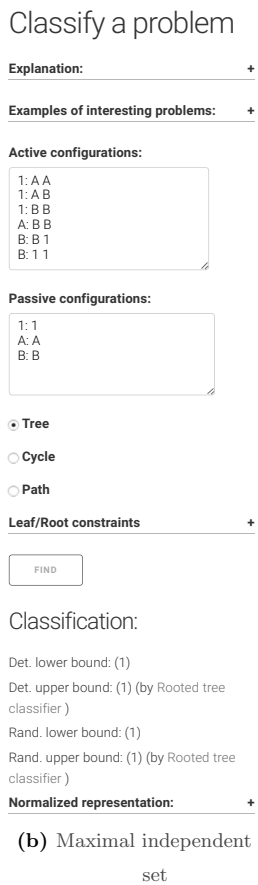
Finding the certificate is almost identical to finding the one for $O(\log^* n)$ solvability, with the only difference that for each label l that can follow itself we additionally keep in the construction of trees a note whether such tree has l as one of its leaves. If we succeed in finding the label l then we have a certificate of $O(1)$ solvability, otherwise there is none.

2.10 LCL classifier interface

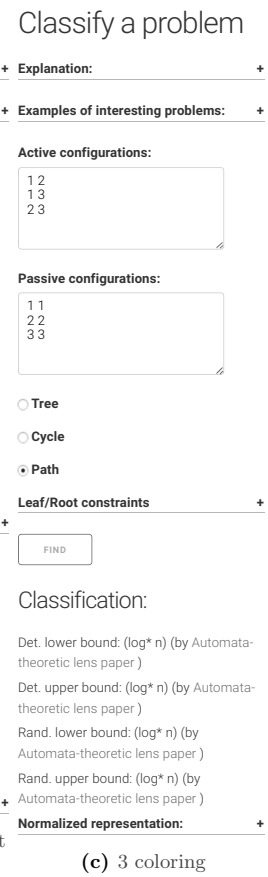
To facilitate identification of a concrete LCL problem, all classifiers have been incorporated into the LCL classifier tool that was created by Tereshchenko [Ter21] and can be accessed at `http://lcl-classifier.cs.aalto.fi`. The idea is to have a single place that enables researchers to easily determine round complexity of any given LCL problem in various graph families. Even though the classification takes exponential time in the case of rooted regular trees, it is nevertheless practical as the problems of interest can be usually described using a small number of labels. Examples of encoding problems for all three classifiers can be seen in Figure 2.5.



(a) Two and a half coloring



(b) Maximal independent set



(c) 3 coloring

Figure 2.5. Examples of outputs for various problems using LCL classifier tool.

3. Sparse matrix multiplication

Matrix multiplication is a useful primitive used for example in machine learning for deep neural networks. We study the setting where the matrices to be multiplied are sparse, namely they consist of mostly zeroes. More specifically, we consider the *uniformly sparse* setting, where each row and column of the input matrices has at most d non-zero values. For the output, we are interested in at most d elements for each row and column. One of the applications is triangle detection in bounded degree graphs; in this case we need only a uniformly sparse subset from the output matrix (see Section 3.3 for more information). Our algorithm is also directly applicable if the structure of the input matrices is such that the output will be guaranteed to be uniformly sparse. We are solving the sparse matrix multiplication in the *supported low-bandwidth model* where each computer holds one row of each of the matrices and is responsible for one row of the output matrix. In the low-bandwidth model, each computer can send only one $O(\log n)$-bit number per round. And in the supported low-bandwidth model, we have moreover a centralized view of the structure of the matrices, i.e., where the non-zero values are.

As a warm-up, we show that there is a straightforward algorithm that can solve the uniformly sparse matrix multiplication in $O(d^2)$ rounds. Then we show part of the techniques that are used to break the quadratic barrier and achieve runtime of $O(d^{1.907})$ rounds. All results come from Publication IV.

3.1 Preliminaries

Let us first define the problem we are going to be solving.

Definition 3.1.1 (Square matrix multiplication)**.** Let A and B be square n-by-n matrices with elements a_{ij} and b_{ij} respectively. Matrix X with elements

x_{ij} is the product of A and B if

$$x_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \text{ for all } 1 \leq i, j \leq n.$$

Definition 3.1.2 (Uniformly sparse matrix)**.** Let A be a matrix. Matrix A is said to be uniformly sparse with parameter d if each row and each column contains at most d non-zeros.

Next, we need to define the model of computing which we will be using.

Definition 3.1.3 (Low bandwidth model)**.** The low bandwidth model is a model of distributed computing, where we have all-to-all communication. Input is spread across nodes. In each round, a node can send and receive only one $O(\log n)$ sized message.

In our case of matrix multiplication, a node i is holding row i of each input matrices and has to output row i of the output matrix.

For the model of computing, we work with a low bandwidth model that is moreover equipped with a support. A model of distributed computing is called *supported* if we moreover have a centralized view of the structure of problem we are going to solve and we are just missing concrete instantiation. E.g. we are given a supergraph G (structure of the problem) with the concrete instance being a subgraph of G.

So in case of sparse matrix multiplication, in supported model, we are globally given the positions of non-zeroes (structure) with the concrete values being unknown. This information will be denoted by indicator matrices $\hat{A}, \hat{B}$ and $\hat{X}$.

To reiterate the setting, we will be solving sparse square matrix multiplication in the supported low bandwidth model. The reason for choosing the low bandwidth model is that the problem of computing matrix multiplication for very sparse matrices becomes trivial in high bandwidth setting and we want to provide a more fine-grained complexity for the very sparse region. Using the supported model enabled avoiding complex routing schemes and having a fast algorithm in the supported model is a prerequisite to having a fast algorithm in an unsupported model.

3.2 Related work

Understanding the complexity of matrix multiplication is an active area of research that can be divided into centralized matrix multiplication algorithms and matrix multiplication in parallel and distributed models.

3.2.1 Centralized matrix multiplication

In the centralized, sequential setting, matrix multiplication can be solved directly using the definition, which yields $O(n^3)$ running time. This was for a long time the best algorithm and some were trying to prove its optimality. This changed in 1969 when Volker Strassen [Str69] discovered a matrix multiplication algorithm that works in $O(n^{2.807})$ time. The algorithm is a divide-and-conquer algorithm. The idea is to first split each matrix in 4 submatrices. Then, the algorithm creates only 7 matrices (instead of 8 for the traditional matrix multiplication) that have one (recursive) multiplication each and from which the final product is constructed. Strassen's algorithm is not the asymptotically fastest algorithm, but it was the first one to break the cubic runtime.

Another important result is the Coppersmith–Winograd algorithm [CW90] from 1990 which is the base for all modern algorithms that achieve even lower matrix multiplication exponent than the original algorithm. Its running time is $O(n^{2.376})$. The algorithm is also a divide-and-conquer algorithm, but it uses a different identity and a laser method from [Str87] to achieve such running time.

Currently the fastest matrix multiplication algorithm is from January 2024 by Williams, Xu, Xu, and Zhou [Wil+24] that achieves $O(n^{2.371552})$ time. Their results is based on the refined laser method.

For the case of rectangular matrix multiplication, we consider computing the product of an $n \times \lceil n^k \rceil$ matrix with a $\lceil n^k \rceil \times n$ matrix for some parameter k. In an unexpected result, Coppersmith [Cop82] showed in 1982 that multiplication of an $n \times \lceil n^{0.172} \rceil$ matrix with a $\lceil n^{0.172} \rceil \times n$ matrix can be done in $O(n^{2+\epsilon})$ time for any $\epsilon > 0$. Then in January 2024, the same paper that improved square matrix multiplication also improved rectangular multiplication achieving $O(n^{2+\epsilon})$ runtime for matrices of sizes $n \times \lceil n^{0.321334} \rceil$ and $\lceil n^{0.321334} \rceil \times n$ [Wil+24].

For sparse n-by-n matrices, each having m non-zero elements, Yuster and Zwick [YZ05] designed an algorithm that computes their product in $O(m^{0.7}n^{1.2} + n^{2+o(1)})$ time. The algorithm partitions the matrices into dense and sparse parts and then for the dense part applies fast rectangular matrix multiplication and for the sparse part naive sparse matrix multiplication.

3.2.2 Matrix multiplication in parallel and distributed models

In the case of parallel matrix multiplication, Ballard and others [Bal+12] in 2012 parallelized Strassen's algorithm to obtain close to optimal parallel parallel execution. The computational cost of their approach is $\frac{n^{2.807}}{P}$, where

P is the number of processors. The bandwidth cost (word count) is

$$\max\left\{\frac{n^{2.807}}{PM^{0.403}}, \frac{n^2}{P^{0.725}}\right\},$$

where M is the memory size of each processor. And latency (message count) is

$$\min\left\{\frac{n^{2.807}}{P^{1.403}}\log P, \log P\right\}.$$

The costs are along the critical path of execution, which is the sequence of program activities that take longest time to execute.

For the case of distributed algorithms, Censor-Hillel et al. [Cen+15] study the case of matrix multiplication in the congested clique model. Congested clique differs from the low bandwidth model in that instead of each node sending only one message per round, one node can send one message to each node. In such model, they obtain a matrix multiplication algorithm that runs in $O(n^{0.158})$ rounds. Their method decomposes a matrix into blocks and uses a fast centralized matrix multiplication algorithm in a way such that each computer does only one block multiplication.

3.3 Applications

Matrix multiplication is used as a subroutine for solving various other computational problems. The most straightforward one is triangle counting in bounded degree graphs. In the supported model we need to provide a bounded degree super-graph whose any subgraph we are allowed to analyze. In terms of matrix multiplication, both matrices $\hat{A}$ and $\hat{B}$ are the adjacency matrix of the super-graph $\hat{G}$. Let $e = (i, j)$ be an arbitrary edge of a subgraph G we are interested in. Then in the product X of A and B (where A and B are the adjacency matrix of G) at positions indicated by the edge (that is X_{ij}) is the number of triangles in the graph that contain edge e.

Examples of other problems solvable using matrix multiplication are k-cycle detection, computation of girth, diameter and solving all-pairs shortest paths [Cen+15; Le 16].

3.4 Warm-up – $O(d^2)$ time

To reiterate the task, we have two uniformly sparse matrices A and B and an indicator matrix $\hat{X}$ (that indicates which product elements we are interested in) and the task is to produce a product X of A and B at positions specified by $\hat{X}$. A computer i is holding row i of each of the input matrices and is supposed to

output row i of the product matrix. Now let us show how to compute such matrix multiplication in $O(d^2)$ rounds.

Note that each computer already has all the needed values from the matrix A to compute the product, it is just missing the values from the matrix B. Hence what we do is to simply send to each computer its missing values from the matrix B. In what follows we show that each computer will send and receive only d^2 values and the commutation can be parallelized to be finished in $O(d^2)$ rounds. Observe that each resulting entry in matrix X is computed as the sum of at most d products. This is because both A and B are uniformly sparse. And because of uniformly sparse $\hat{X}$, each row of X has at most d results. Therefore each computer needs to receive at most d^2 messages to compute all products needed for computing its part of the output. For the input values, each input value is needed at most d times for the matrix multiplication because the matrices are uniformly sparse. In addition to that, each row has at most d elements. Therefore, each computer needs to send at most d^2 messages. To show that all the messages can be exchanged in $O(d^2)$ rounds, we can build a bipartite graph of senders and receivers. As argued previously, such graph has maximum degree d^2 on the side of senders and also maximum degree d^2 on the side of receivers. Therefore such graph admits $O(d^2)$-edge coloring. And if we use this coloring as a schedule for sending the messages, we can send the values in parallel in $O(d^2)$ time.

3.5 Matrix multiplication as counting triangles

There are multiple ways how to view matrix multiplication. One of them is as counting triangles.

Consider a tripartite structure with sets I, J, K each of size n. These will be nodes of graph $G_{\hat{T}}$ which we will call *triangle graph*. Edges of the graph are formed by the indicator matrix $\hat{A}$ between I and J (there is an edge between node $i \in I$ and $j \in J$ if $\hat{A}_{ij} \neq 0$). Same goes for the indicator matrix $\hat{B}$ forming edges between nodes in J and K and for $\hat{X}$ forming edges between nodes in I and K. Example of a triangle graph can be seen in Figure 3.1.

We can observe that all relevant individual multiplications correspond to triangles in $G_{\hat{T}}$.

We can also interpret this as computing the matrix product using $3n$ computers, with a computer at each node. Computer at node $i \in I$ initially knows a_{ij} for all j, computer at node $j \in J$ knows b_{ik} for all k and computer at node $i \in I$ needs to compute x_{ik} for all k. Initially we only have n computers, but

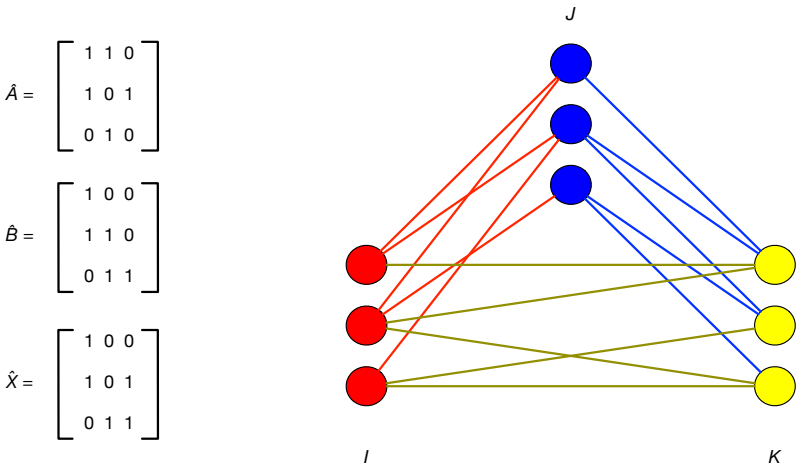


Figure 3.1. An example of indicator matrices $\hat{A}, \hat{B}$ and $\hat{X}$, and the resulting triangle graph $G_{\hat{T}}$.

we can task each physical computer to simulate 3 virtual computers.

Now we can rephrase and generalize our warm-up exercise in the triangle view.

Lemma 3.5.1. *Let $G_{\hat{T}}$ be a triangle graph. If the number of triangles that is incident to each vertex is bounded by t, then we can process all triangles in $O(t)$ time.*

Proof. For processing all triangles we want for each triangle $\{i, j, k\} \in G_{\hat{T}}$ that the computer j sends the value b_{jk} to computer i. This is enough as the computer j has the value of a_{ij} and is responsible for outputting x_{ik}. In this case we know that each node is incident to t triangles, so the computer needs to receive only t messages with values. The same holds for senders as they are also incident to only t triangles.

So when building the bipartite graph of senders and receivers, the graph has maximum degree t. Therefore it admits an $O(t)$-edge coloring. And if we use it again as a scheduler, we can send the values in parallel in $O(t)$ time. □

For the following parts we assume that we have reduced the overall number of triangles to $O(d^{2-\epsilon}n)$ for some value of ϵ. So on average, each vertex is incident to $O(d^{2-\epsilon})$ triangles.

But what if the triangles are not uniformly distributed across the instance so we cannot meaningfully use Lemma 3.5.1? In that case, we will create a virtual instance and assign computers to virtual nodes such that the virtual instance does not have places with as high triangle concentration as before.

At the end of this section, we want to show the following theorem.

Theorem 3.5.2. *Let $G_{\hat{T}}$ be a triangle graph with $O(d^{2-\epsilon}n)$ triangles and at most d edges from one vertex to the same partition. We can process all triangles in $O(d^{2-\epsilon/2})$ rounds.*

Firstly, we need the following lemma.

Lemma 3.5.3. *Let $G_{\hat{T}}$ be a triangle graph with $O(d^{2-\epsilon}n)$ triangles and at most d edges from one vertex to the same partition. There exists a coloring of triangles with $d^{\epsilon/2}/3$ colors such that for any color c, each node $i \in V$ touches at most $9d^{2-\epsilon/2}$ triangles of color c.*

Proof. To show that, we would first greedily color the triangles. This means that we process the triangles one by one and the triangle receives the lowest color that is available after removing all colors of its neighbors. How many colors do we need, i.e., how many neighbors at maximum has one triangle? Each vertex of a triangle can be incident to at most $d^2 - 1$ other triangles, as we have a tripartite graph that has at most d edges from one vertex to the same partition. Hence a triangle can be incident to at most $3d^2 - 3$ triangles and we can greedily color the triangles with $3d^2 - 2$ colors. As a next step, we group the colors to $d^{\epsilon/2}/3$ buckets so each bucket contains at most

$$\frac{3d^2 - 2}{d^{\epsilon/2}/3} \leq 9d^{2-\epsilon/2}$$

original colors. Finally, we give each bucket a color and recolor triangles by the bucket color. This coloring satisfies the property that for any color c, each node touches at most $9d^{2-\epsilon/2}$ triangles of color c. □

Now we need some definitions. By a *bad node* we describe a node that is included in at least $d^{2-\epsilon/2}$ triangles. And by a *bad triangle* we describe a triangle that has a bad node. Observe that we can have at most

$$\frac{d^{2-e}n}{d^{2-\epsilon/2}/3} = 3d^{\epsilon/2}n$$

bad nodes as each triangle contributes to 3 vertices.

With the definitions in mind, we can construct the virtual instance as follows.

Let χ be a coloring from Lemma 3.5.3 and C its color space.

Each non-bad node will be copied to the virtual instance as is and each bad node will create $|C|$ nodes, one for each color. Then given a non-bad triangle, it will be copied onto its nodes. A bad triangle with a color c will have its bad nodes placed on nodes with color c and non-bad nodes will stay the same. Values along the edges will be always copied (for matrix A, B) and each resulting edge (matrix X) can be obtained as the sum of all edges that

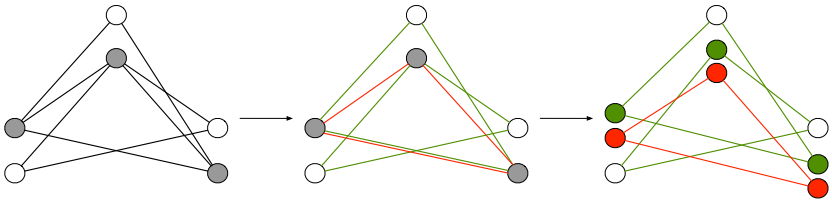


Figure 3.2. An example of coloring the triangles in a triangle graph and creating a virtual instance. Bad nodes in this artificial example are depicted in gray. Those will be the nodes that will be split into the green and red nodes.

start and end at the same vertices irrespective of colors. An example how the individual phases of coloring and creating virtual instance look like on a graph, see Figure 3.2.

Now let us move onto the simulation. In this part we show how the computers simulate the virtual instance so that we can use it for solving the problem. As we operate in a supported model, we can first use a centralized algorithm to compute the triangle coloring and create the virtual instance. Each computer will be responsible for two nodes. The computer initially holding a non-bad node will hold it and one colored bad node. The computer initially holding a bad node will hold two colored bad nodes. Note that by such distribution we cover all virtual nodes as we have in addition to non-bad nodes at most $3n/d^{\epsilon/2}$ original bad node and each creates only $d^{\epsilon/2}/3$ colored nodes.

Now we need to describe how we route the elements of the input matrix to the virtual nodes and how to gather back the elements of the output matrix. For that, we need to prove this supportive routing lemma.

Lemma 3.5.4. *Consider a node v from the set V and $U \subseteq V$ consisting of k nodes. In a low-bandwidth model, the communication tasks outlined below can be computed within $O(d + \log k)$ rounds, utilizing only communication between nodes in the set $\{v\} \cup U$.*

- *Broadcasting values $a_1, a_2, \ldots, a_d$ initially known only to node v to all nodes in U.*
- *Gathering sums of values $s_1, s_2, \ldots, s_d$ where $s_i = \sum_{u \in U} a_{i,u}$ to node v where values $a_{i,u}$ are initially spread across nodes in U.*

Proof. As a first step, we arbitrarily pick a binary tree rooted at v which spans all nodes in U.

For the first tasks, let us consider first the case with only one value to be broadcasted. The algorithms works as follows, starting from node v each node that just received a spends two rounds to communicate this value to both of its children. This process will end after $O(\log k)$ rounds as the depth of the

tree is $\lceil \log k \rceil$. For multiple values, the root node will send a new value every second round. That way we can keep the rest of the algorithm the same and such task will take $O(d + \log k)$ rounds as the last value is delayed by $2d$.

For the second task, lets again consider first the case of only one sum of values to be gathered. Starting at leaf nodes, in odd (resp. even) rounds a left (resp. right) child sends its computed value to the parent. After every two rounds, each node computes sum of its received values and its corresponding value that will be used to send it higher up. Also here, it will take $O(\log k)$ rounds to propagate the sum to the root. For multiple values, we will use the same idea as for the first task, i.e. delaying the subsequent values. The runtime will again be $O(d + \log k)$. □

So, the simulation proceeds in three steps.

1. Each bad node sends their input to all computers responsible for the colored counterparts. This takes $O(d^{\epsilon/2} + \log d)$ rounds by the first part of Lemma 3.5.4.
2. Computers simulate algorithm from Lemma 3.5.1 on the virtual instance. This runs in $O(d^{2-\epsilon/2})$ rounds.
3. Each bad node gathers back the output from colored counterparts. This takes $O(d^{\epsilon/2} + \log d)$ rounds by the second part of Lemma 3.5.4.

Hence, the whole simulation runs in $O(d^{2-\epsilon/2})$ rounds which is exactly what is needed for the Theorem 3.5.2 to hold.

3.6 Complete algorithm

The previous section showed how to process triangles and hence compute the matrix multiplication when we have already reduced the number of triangles to $O(d^{2-\epsilon}n)$. But for the complete algorithm to work, we also need to solve the initial triangle reduction. We solve it by finding a non-overlapping set of dense clusters (each of size $d^{3-\delta}$ for a small delta) and then we use dense matrix multiplication from [Cen+15] to each cluster in parallel. We repeat these steps until we have sufficiently reduced the number of triangles so we can use the Theorem 3.5.2 to finish the triangle processing. For more details visit Sections 4 and 5 of Publication IV. By optimizing the parameters of the clustering algorithm we can obtain the final running time of $O(d^{1.907})$ rounds. In the case of semirings, where we have to use slower algorithm for the dense matrix multiplication we obtain running time of $O(d^{1.927})$ rounds.

3.7 Future direction

One natural avenue for further direction is to lower the exponent in the runtime. There is a information-theoretic lower bound of $O(d)$, but anything in between is an open question. One place where we could obtain improvements is in the processing of instance after the clustering phase. There we are having an average of $O(d^{2-\epsilon})$ triangles per vertex, yet we are spending $O(d^{2-\epsilon/2})$ rounds to process them.

Next direction is removal of the support. We use it extensively for routing, so it is expected that new routing schemes would be needed.

Lastly, we are considering only uniformly sparse matrices, so the open question is how much can we relax the constrains while still achieving subquadratic runtime.

4. Conclusion

We have seen two areas of distributed computing that share a common theme that is finding fast distributed algorithms in sparse graphs. In the first part, which focused on locally checkable labeling problems we have shown that designing a meta-algorithm that finds optimal distributed algorithm for solving a given problem on a given graph family is possible. It is not only possible but we can find the right complexity also efficiently for most of the cases and there is a practical implementation of all the algorithms. It is important to note that given the graph family (that is cycles, paths or rooted regular trees) we can classify any LCL problem as opposed to results finding an optimal algorithm for one specific problem.

In the second part we have focused on finding an efficient algorithm for sparse matrix multiplication when the communication bandwidth is low. By finding dense regions inside the sparse matrix and using fast dense multiplication we have managed to break the quadratic barrier for uniformly sparse matrix. By having such algorithm we opened possibilities for further improvement.

Results in both of the research areas have already led to follow-up work: The concept of certificates was extended to the online-LOCAL model [Akb+23] and the result of Yi-Jun Chang [Cha24] shows that there is a simple characterization of the minor-closed graph classes sharing the same deterministic complexity landscape as paths. For the matrix multiplication, a manuscript [Gup+24] shows an improved exponent for the sparse matrix multiplication and extends the results to more general notions of sparsity.

Bibliography

[Akb+23] Amirreza Akbari, Navid Eslami, Henrik Lievonen, Darya Melnyk, Joona Särkijärvi, and Jukka Suomela. "Locality in Online, Dynamic, Sequential, and Distributed Graph Algorithms". In: *50th International Colloquium on Automata, Languages, and Programming (ICALP 2023)*. Ed. by Kousha Etessami, Uriel Feige, and Gabriele Puppis. Vol. 261. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 10:1–10:20. ISBN: 978-3-95977-278-5. DOI: 10.4230/LIPIcs.ICALP.2023.10.

[Ang80] Dana Angluin. "Local and Global Properties in Networks of Processors (Extended Abstract)". In: *Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*. STOC '80. Los Angeles, California, USA: Association for Computing Machinery, 1980, pp. 82–93. ISBN: 0897910176. DOI: 10.1145/800141.804655.

[Bal+12] Grey Ballard, James Demmel, Olga Holtz, Benjamin Lipshitz, and Oded Schwartz. "Communication-optimal parallel algorithm for strassen's matrix multiplication". In: *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*. 2012, pp. 193–204.

[Bal+19] Alkida Balliu, Sebastian Brandt, Yi-Jun Chang, Dennis Olivetti, Mikaël Rabie, and Jukka Suomela. "The distributed complexity of locally checkable problems on paths is decidable". In: *Proc. 38th ACM Symposium on Principles of Distributed Computing (PODC 2019)*. ACM Press, 2019, pp. 262–271. DOI: 10.1145/3293611.3331606. eprint: 1811.01672.

[Bal+21] Alkida Balliu, Sebastian Brandt, Dennis Olivetti, and Jukka Suomela. "Almost global problems in the LOCAL model". In: *Distributed*

Computing 34 (2021), pp. 259–281. DOI: 10.1007/s00446-020-00375-2.

[Bra+17] Sebastian Brandt, Juho Hirvonen, Janne H. Korhonen, Tuomo Lempiäinen, Patric R. J. Östergård, Christopher Purcell, Joel Rybicki, Jukka Suomela, and Przemysław Uznański. "LCL problems on grids". In: *Proc. 36th ACM Symposium on Principles of Distributed Computing (PODC 2017)*. ACM Press, 2017, pp. 101–110. DOI: 10.1145/3087801.3087833. eprint: 1702.05456.

[CDP21] Artur Czumaj, Peter Davies, and Merav Parter. "Simple, Deterministic, Constant-Round Coloring in Congested Clique and MPC". In: *SIAM Journal on Computing* 50.5 (2021), pp. 1603–1626. DOI: 10.1137/20M1366502.

[Cen+15] Keren Censor-Hillel, Petteri Kaski, Janne H Korhonen, Christoph Lenzen, Ami Paz, and Jukka Suomela. "Algebraic methods in the congested clique". In: *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*. 2015, pp. 143–152.

[Cha20] Yi-Jun Chang. "The Complexity Landscape of Distributed Locally Checkable Problems on Trees". In: *Proc. 34th International Symposium on Distributed Computing (DISC 2020)*. Vol. 179. LIPIcs. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, 18:1–18:17. DOI: 10.4230/LIPIcs.DISC.2020.18.

[Cha24] Yi-Jun Chang. "The Distributed Complexity of Locally Checkable Labeling Problems Beyond Paths and Trees". In: *15th Innovations in Theoretical Computer Science Conference (ITCS 2024)*. Ed. by Venkatesan Guruswami. Vol. 287. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 26:1–26:25. ISBN: 978-3-95977-309-6. DOI: 10.4230/LIPIcs.ITCS.2024.26.

[CKP19] Yi-Jun Chang, Tsvi Kopelowitz, and Seth Pettie. "An Exponential Separation between Randomized and Deterministic Complexity in the LOCAL Model". In: *SIAM Journal on Computing* 48.1 (2019), pp. 122–143. DOI: 10.1137/17M1117537.

[Cop82] Don Coppersmith. "Rapid multiplication of rectangular matrices". In: *SIAM Journal on Computing* 11.3 (1982), pp. 467–471.

[CP19] Yi-Jun Chang and Seth Pettie. "A Time Hierarchy Theorem for the LOCAL Model". In: *SIAM Journal on Computing* 48.1 (2019), pp. 33–69. DOI: 10.1137/17M1157957.

[CW90] Don Coppersmith and Shmuel Winograd. “Matrix multiplication via arithmetic progressions”. In: *Journal of Symbolic Computation* 9.3 (1990). Computational algebraic complexity editorial, pp. 251–280. ISSN: 0747-7171. DOI: 10.1016/S0747-7171(08)80013-2.

[FG17] Manuela Fischer and Mohsen Ghaffari. “Sublogarithmic Distributed Algorithms for Lovász Local Lemma, and the Complexity Hierarchy”. In: *Proc. 31st International Symposium on Distributed Computing (DISC 2017)*. Vol. 91. LIPIcs. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017, 18:1–18:16. DOI: 10.4230/LIPIcs.DISC.2017.18.

[Gup+24] Chetan Gupta, Janne H. Korhonen, Jan Studený, Jukka Suomela, and Hossein Vahidi. *Low-Bandwidth Matrix Multiplication: Faster Algorithms and More General Forms of Sparsity*. 2024. arXiv: 2404.15559 [cs.DC].

[IS99] Balázs Imreh and Magnus Steinby. “Directable nondeterministic automata”. In: *Acta Cybernetica* 14.1 (1999), pp. 105–115. URL: https://cyber.bibl.u-szeged.hu/index.php/actcybern/article/view/3514.

[Le 16] François Le Gall. “Further algebraic algorithms in the congested clique model and applications to graph-theoretic problems”. In: *Distributed Computing: 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings 30*. 2016, pp. 57–70.

[Lin92] Nathan Linial. “Locality in distributed graph algorithms”. In: *SIAM Journal on Computing* 21.1 (1992), pp. 193–201. DOI: 10.1137/0221015.

[Lot+03] Zvi Lotker, Elan Pavlov, Boaz Patt-Shamir, and David Peleg. “MST construction in O(log log n) communication rounds”. In: *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA ’03. San Diego, California, USA: Association for Computing Machinery, 2003, pp. 94–100. ISBN: 1581136617. DOI: 10.1145/777412.777428.

[MR85] Gary L. Miller and John H. Reif. “Parallel tree contraction and its application”. In: *Proc. 26th Annual Symposium on Foundations of Computer Science (FOCS 1985)*. IEEE, 1985, pp. 478–489. DOI: 10.1109/SFCS.1985.43.

[NS95] Moni Naor and Larry J. Stockmeyer. “What Can be Computed Locally?” In: *SIAM Journal on Computing* 24.6 (1995), pp. 1259–1277. DOI: 10.1137/S0097539793254571.

[Pel00] David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, 2000. DOI: 10.1137/1.9780898719772.

[RG20] Václav Rozhoň and Mohsen Ghaffari. “Polylogarithmic-time deterministic network decomposition and distributed derandomization”. In: *Proc. 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC 2020)*. ACM, 2020, pp. 350–363. DOI: 10.1145/3357713.3384298.

[Str69] Volker Strassen. “Gaussian elimination is not optimal”. In: *Numerische mathematik* 13.4 (1969), pp. 354–356.

[Str87] Volker Strassen. “Relative bilinear complexity and matrix multiplication.” In: 1987.375-376 (1987), pp. 406–443. DOI: 10.1515/crll.1987.375-376.406.

[Ter21] Aleksandr Tereshchenko. “Automated classification of distributed graph problems”. English. Master’s thesis. Aalto University. School of Science, 2021, p. 76. URL: http://urn.fi/URN:NBN:fi:aalto-202105236941.

[Wil+24] Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. “New Bounds for Matrix Multiplication: from Alpha to Omega”. In: *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2024, pp. 3792–3835. DOI: 10.1137/1.9781611977912.134.

[YZ05] Raphael Yuster and Uri Zwick. “Fast sparse matrix multiplication”. In: *ACM Transactions On Algorithms (TALG)* 1.1 (2005), pp. 2–13.

Aalto-DT 219/2024

ISBN 978-952-64-2076-9 (printed)
ISBN 978-952-64-2077-6 (pdf)
ISSN 1799-4934 (printed)
ISSN 1799-4942 (pdf)

Aalto University
School of Science
Department of Computer Science
www.aalto.fi

BUSINESS + ECONOMY

ART + DESIGN + ARCHITECTURE

SCIENCE + TECHNOLOGY

CROSSOVER

DOCTORAL THESES