# Automatically Finding Non-constant Lower Bounds for Locally Checkable Labeling Problems in the LOCAL Model

Nikos Heikkilä

**School of Science**

Thesis submitted for examination for the degree of Master of Science in Technology.

Helsinki, May 30, 2022

**Supervisor**

Prof. Jukka Suomela

**Advisor**

Dr. Chetan Gupta

**Aalto University**
School of Science

Copyright © 2022 Nikos Heikkilä

**Aalto University**
**School of Science**

**Aalto University, P.O. BOX 11000, 00076 AALTO**
**www.aalto.fi**
**Abstract of the master's thesis**

**Author** Nikos Heikkilä

**Title** Automatically Finding Non-constant Lower Bounds for Locally Checkable Labeling Problems in the LOCAL Model

**Degree programme** Master's Programme in Computer, Communication and Information Sciences

**Major** Computer Science **Code of major** SCI3042

**Supervisor** Prof. Jukka Suomela

**Advisor** Dr. Chetan Gupta

**Date** May 30, 2022 **Number of pages** 59 **Language** English

**Abstract**

Distributed computing is any kind of computing that is performed on a spatially distributed system. It is often considered when high amounts of computation power is required. Distributed computing is used to solve large-scale problems that would otherwise be impractical to solve in a centralized system.

In this thesis, I study the theoretical foundations of distributed computing. The models of distributed computing I am interested in are the port-numbering (PN) model and the LOCAL model. In these models, each node of a computer network executes a common algorithm synchronously and can exchange messages with their neighboring nodes in each communication round. Between these communication rounds, the nodes can perform computation in an instant. The complexity of the algorithm is measured as the number of communication rounds it takes until each node of a network terminates. Locally checkable labeling (LCL) problems are a family of graph problems where a global solution can be verified locally by the individual nodes.

In the research of the foundations of distributed computing there is a recent trend to automate finding upper and lower bounds for LCL problems in the LOCAL model. This thesis contributes to the field by automating the process of finding non-constant lower bounds for LCL problems in the LOCAL model.

I present a new algorithm that can detect if an LCL problem does not have a solution in finite connected $(\Delta, \delta)$-biregular multigraphs. Then I show that if the problem does not have a solution in the graph family, it is also unsolvable in the PN model. I also prove that if an LCL problem is unsolvable in the PN model, then it cannot be solved in constant time in the LOCAL model. Thus, the algorithm can automatically prove that an LCL problem is unsolvable in constant time in the LOCAL model. In order to automatically find new lower bounds for LCL problems in the LOCAL model in practice, I present an implementation of the algorithm. With the implementation, I find new lower bounds for nine LCL problems and as a consequence, one of the problems is now classified with a tight bound of $\Theta(\log^* n)$.

**Keywords** Locally checkable labeling problems, LOCAL model, Port-numbering model, Distributed algorithms, Distributed computing

**Aalto University**
**School of Science**

**Aalto-yliopisto, PL 11000, 00076 AALTO**
**www.aalto.fi**
**Diplomityön tiivistelmä**

| | |
|---|---|
| **Tekijä** Nikos Heikkilä | |
| **Työn nimi** Ei-vakioaikaisten alarajojen etsiminen automaattisesti paikallisesti tarkastettaville merkitsemisongelmille LOCAL-mallissa | |
| **Koulutusohjelma** Master's Programme in Computer, Communication and Information Sciences | |
| **Pääaine** Computer Science | **Pääaineen koodi** SCI3042 |
| **Työn valvoja** Prof. Jukka Suomela | |
| **Työn ohjaaja** Dr. Chetan Gupta | |
| **Päivämäärä** May 30, 2022 **Sivumäärä** 59 | **Kieli** Englanti |

**Tiivistelmä**

Hajautettu laskenta on mitä tahansa laskentaa, johon osallistuu eri paikoissa sijaitsevia tietokoneita. Sitä käytetään usein tilanteissa, joissa vaaditaan suurta laskentatehoa. Hajautettua laskentaa hyödyntämällä voidaan ratkaista ison mittakaavan ongelmia, jotka olisivat epäkäytännöllisiä ratkaista keskitetyssä järjestelmässä.

Tässä työssä tutkitaan hajautetun laskennan teoreettista perustaa. Olen kiinnostunut hajautetun laskennan porttinumerointimallista eli PN-mallista sekä LOCAL-mallista. Näissä malleissa jokainen tietokoneverkon solmu suorittaa samaa algoritmia synkronisesti ja vaihtaa jokaisella viestintäkierroksella viestejä viereisten solmujen kanssa. Näiden viestintäkierroksien välillä solmut voivat laskea äärettömän nopeasti. Algoritmin nopeus on se kierrosten määrä, jonka jälkeen jokainen verkon solmu viimeistään lopettaa algoritmin suorittamisen. Paikallisesti tarkastettavat merkitsemisongelmat (LCL-ongelmat) ovat verkko-ongelmaperhe, jossa jokainen yksittäinen solmu voi paikallisesti tarkistaa globaalin ratkaisun.

Hajautetun laskennan teoreettisen perustan tutkimuksissa on viime aikoina näkynyt kehityssuunta, jossa automatisoidaan LCL-ongelmien ylä- ja alarajojen löytäminen LOCAL-mallissa. Tämän työ edistää kyseistä tutkimusalaa automatisoimalla LCL-ongelmien ei-vakioaikaisten alarajojen löytämisen LOCAL-mallissa.

Esittelen uuden algoritmin, joka tunnistaa, jos annetulle LCL-ongelmalle ei ole ratkaisua äärellisissä yhtenäisissä $(\Delta, \delta)$-säännöllisissä kaksijakoisissa moniverkoissa. Tämän jälkeen näytän, että jos tällä ongelmalla ei ole ratkaisua kyseisessä verkkoperheessä, niin se ei ole ratkeava PN-mallissa. Näytän myös, että jos annettu LCL-ongelma ei ole ratkeava PN-mallissa, niin sitä ei voi myöskään ratkaista vakioajassa LOCAL-mallissa. Tästä seuraa, että esittelemäni algoritmi voi todistaa automaattisesti, että LCL-ongelma ei ole vakioajassa ratkeava. Esittelen myös toteutuksen algoritmille, jotta voimme käytännössä löytää automaattisesti uusia alarajoja LCL-ongelmille LOCAL-mallissa. Löydän tällä toteutuksella uusia alarajoja yhdeksälle LCL-ongelmalle ja tämän seurauksena yhdelle näistä ongelmista tunnetaan sen laskennallinen vaativuus tarkasti: ongelman ratkaiseminen vaatii $\Theta(\log^* n)$ viestintäkierrosta.

**Avainsanat** Paikallisesti tarkastettavat merkitsemisongelmat, LOCAL-malli, porttinumerointimalli, hajautetut algoritmit, hajautettu laskenta

# Preface

I would like to thank my supervisor Jukka Suomela for the idea for this thesis, for the opportunity to write this thesis, and to work as a research assistant in the research group. I am grateful for the enormous amount of advice, comments, and support I have received from Jukka. I would also like to thank my advisor Chetan Gupta for giving me invaluable advice and comments on my thesis. Finally, I would like to thank my family and friends for supporting me during my academic journey.

This work was supported in part by the Academy of Finland, Grant 333837.

Helsinki, May 30, 2022 Nikos Heikkilä

# Contents

**Abstract** 3

**Abstract (in Finnish)** 4

**Preface** 5

**Contents** 6

**1 Introduction** 8
1.1 Objective . . . . . . . . . 8
1.2 Organization of the Thesis . . . . . . . . . 8

**2 Background** 10
2.1 Graphs . . . . . . . . . 10
2.2 Distributed Computing . . . . . . . . . 14
2.3 Port-Numbering Model . . . . . . . . . 15
2.4 Formalized Port-Numbering Model . . . . . . . . . 15
2.4.1 Underlying Graph . . . . . . . . . 17
2.4.2 Distributed Graph Problems . . . . . . . . . 17
2.4.3 Distributed Algorithms in the PN Model . . . . . . . . . 19
2.4.4 Execution of PN Algorithm . . . . . . . . . 19
2.4.5 Solving Graph Problems in PN . . . . . . . . . 20
2.5 Covering Map . . . . . . . . . 21
2.6 LOCAL Model . . . . . . . . . 23
2.7 Locally Checkable Labeling Problems . . . . . . . . . 24
2.8 LCL Problems in Biregular Graphs . . . . . . . . . 25

**3 Research Questions** 28

**4 Prior Work** 29
4.1 Automation of Proving . . . . . . . . . 29
4.2 Automation of Proving in the LOCAL Model . . . . . . . . . 30

**5 Unsolvability in the PN Model** 32
5.1 From Multiple Connection Networks to Simple Networks . . . . . . . . . 34
5.2 From Finite to Infinite . . . . . . . . . 36
5.3 From PN to LOCAL . . . . . . . . . 38

**6 Implementation** 40
6.1 Generating Multigraphs . . . . . . . . . 40
6.2 Boolean Satisfiability Problem . . . . . . . . . 44
6.3 Our SAT Encoding . . . . . . . . . 44
6.4 Generating LCL Problems . . . . . . . . . 47
6.5 Optimizations . . . . . . . . . 49

**7 Results** **50**
7.1 Improving Lower Bounds of LCL Problems . . . . . . . . . . . . . . . 50
7.2 Finding New Lower Bounds For Classes of LCL Problems . . . . . . . 53

**8 Conclusion** **55**
8.1 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55

**References** **56**

# 1 Introduction

Large problems often require large computational capacity. Distributed computing is *any kind* of computing that is performed on a spatially distributed system [6]. Distributed computing is often considered when high amounts of computation power is required. It is used to solve large-scale problems that would otherwise be impractical to solve in a centralized system.

In this thesis, we study the theoretical foundations of distributed computing. Commonly, the research in the field focuses on distributed algorithms and complexity classes of distributed graph problems. A distributed algorithm is a program that is executed in a distributed system. Different computation models are used as an abstraction of these distributed systems. The models we are particularly interested in are the port-numbering (PN) model [4] and the LOCAL model [5]. In these models, computation is done in synchronous communication rounds that consist of message passing. A communication round is the unit of time used in measuring a complexity of an algorithm in these models.

Our main focus is on a certain class of distributed graph problems called locally checkable labeling (LCL) problems. LCL problems are a family of graph problems where a global solution can be verified locally by the individual nodes. For example, in graph coloring, each node can check that their neighbors do not share the color of the node.

To understand the exact complexities of LCL problems in these models, we focus on finding lower bounds for them. Finding new lower bounds for some LCL problems can potentially help us to figure out a general rule on why these LCL problems have their specific lower bound. One way of proving a lower bound is to construct a problem instance which requires some amount of communication rounds to be solved. Finding a lower bound requires some kind of proof of existence, and proving it manually can be tiresome, so instead we want to automate the process.

## 1.1 Objective

In this thesis, we prove that an unsolvability of an LCL problem in the PN model implies that the problem is also not solvable in constant time in the LOCAL model. We will also implement a tool that can automatically find a proof of unsolvability of an LCL problem in the PN model. Using these two, we can derive lower bounds in the LOCAL model. We especially want this tool to be usable in practice, therefore it needs to be optimized to compute results reasonably fast. To conclude, our objective is to find automatically new lower bounds for LCL problems in the LOCAL model.

## 1.2 Organization of the Thesis

We organize the thesis as follows. In Section 2, we give a succinct theoretical background to the topic of this work. After the theoretical background, we present our research questions in more detail in Section 3. In Section 4, we overview the prior work related to the problems and models discussed in this thesis. We introduce our

algorithm and also prove our main theorem (Theorem 5.9) in Section 5. In Section 6, we discuss our implementation of the algorithm. The results from this thesis are presented in Section 7, and finally we conclude the thesis in Section 8.

# 2 Background

In this section, we introduce relevant terminology and concepts that are required for Sections 3–7. In Section 2.1, we introduce all the necessary definitions and notations related to graph theory. In Section 2.2, we discuss the essential concepts in the theory of distributed computing. We formally define distributed computing and introduce the main model of distributed computing, *message passing model*, which is a building block for many of the researched models. In Sections 2.3 and 2.4, we introduce the port-numbering model, and in Section 2.6 we introduce the LOCAL model. These are widely used models in the field of distributed computing, and they are strongly related to this work. In Section 2.5, we introduce a topological concept called *covering map* that reveals a limitation of port-numbering model. We are focusing on a certain family of graph problems called *locally checkable labeling* problems that we discuss in Sections 2.7 and 2.8.

## 2.1 Graphs

In the real world, there are many objects that are somehow related to each other. Such things can often be visualized by a diagram that consists of points, and lines that connect a pair of points. A graph is a mathematical concept that abstracts the relations of these objects. In the literature, the points are called vertices and the lines are called edges [2].

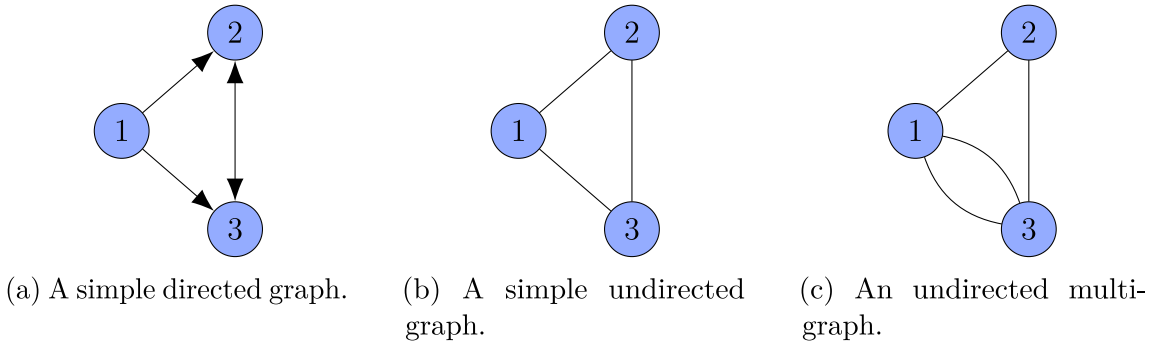
**Definition 2.1.** A graph is a tuple

$$G = (V, E),$$

where $V$ is the set of vertices and $E$ is the set of edges. An edge $e \in E$ is a pair (2-tuple or 2-element ordered list) $e = (v, w), v, w \in V$, where vertices $v$ and $w$ are the endpoints of the edge $e$.

For example, we have a graph $G = (\{1, 2, 3\}, \{(1, 2), (1, 3), (2, 3), (3, 2)\})$, and visualized it looks like the graph in Figure 1a.

When an edge is defined as a pair, the order of the vertices in the edge matters, that is, for all $v, w \in V, v \neq w$, we have $(v, w) \neq (w, v)$. We call such a graph as a *directed graph* or with the shortened variation *digraph*. In an edge $(v, w)$, the first vertex $v$ points to the vertex $w$. Usually the edge is visualized as an arrow pointing from $v$ to $w$. One example of a directed graph is a flow graph, in which the edges represent flows from a vertex to another vertex, as seen in the Figure 1a.



(a) A simple directed graph. (b) A simple undirected graph. (c) An undirected multigraph.

Figure 1: Examples of different graphs

*Undirected graph* is a graph in which the order of the vertices in an edge does not matter. An edge of an undirected graph is defined as an unordered pair $\{v, w\} \in E$, therefore the following holds:

$$\forall v, w \in V \colon \{v, w\} = \{w, v\}. \tag{1}$$

For example $E = \{\{1, 2\}, \{1, 2\}, \{3, 2\}, \{2, 3\}, \{1, 3\}\} = \{\{1, 2\}, \{1, 3\}, \{2, 3\}\}$. Visualization of this graph $G = (\{1, 2, 3\}, E)$ can be seen in the Figure 1b. For the purpose of this work, we need only undirected graphs.

The definitions of graphs shown earlier do not restrict an edge to start and end in itself ($e = \{v, v\}$). This kind of edge is called a *loop*. The definitions however restricts *multiple edges* (identical parallel edges). In order to allow multiple edges, the set of edges has to be defined as a *multiset*. A graph that is allowed to contain multiple edges is called a *multigraph*. Depending on the author or context, multigraphs either allow or disallow loops.

A graph that disallows loops and multiple edges is called a *simple graph*. Simple graphs can either be directed or undirected, and it should be explicitly mentioned when defining graphs, unless the context implies it. As we do not need directedness of edges, let us assume that further expressions of graphs are always undirected in this work.

Suppose there is an edge $e = \{v, w\}$. We say that vertex $v$ is *incident* to $e$ and $e$ is incident to $v$. We also say that vertex $v$ is *adjacent* to $w$ and vice versa. Similarly, when two edges share a vertex, we say that the edges are *adjacent*. We can also say that $w$ is a *neighbor* of $v$ and vice versa.

In a graph, the degree of a vertex is the number of edges it is incident to. We will use the notation $\deg_G(v)$ to denote the degree of vertex $v \in V$ in graph $G = (V, E)$. In multigraphs with loops, a loop counts as 2 to a degree.

A graph that is *bipartite* has exactly two disjoint sets of vertices (we call them part $A$ and part $B$), and every edge of the graph has one endpoint in vertex of $A$ and another in $B$:

$$V = A \cup B, \tag{2}$$

$$A \cap B = \emptyset, \tag{3}$$

$$E = \{\{a, b\} \mid a \in A, b \in B\}. \tag{4}$$

For bipartite graphs, the sum of degrees on $A$ and $B$ are equal:

$$\sum_{a \in A} \deg_G(a) = \sum_{b \in B} \deg_G(b). \tag{5}$$

An example of a bipartite graph can be seen in Figure 2.

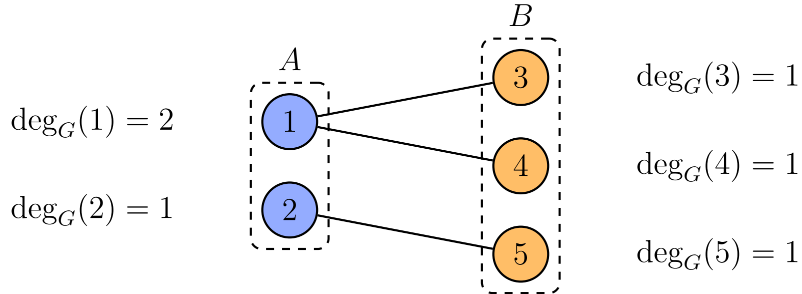



Figure 2: A simple disconnected bipartite graph.

If all vertices in a graph have the same degree, then the graph is called *regular*. For example, if every vertex in a bipartite graph shares a degree, then we call it a regular bipartite graph. When every node $a$ of part $A$ share a degree and every node $b$ of part $B$ share a degree, we call it *a biregular graph*. In fact, we use a notation $(\Delta, \delta)$-biregular, where $\Delta$ and $\delta$ denote the degrees of the nodes inside parts $A$ and $B$ respectively. We can see an example of a (3,2)-biregular in Figure 3. The bipartite graph in Figure 2 is not biregular because the nodes in part $A$ do not share a degree i.e. for all nodes $v, w \in A$ the condition $\deg_G(v) = \deg_G(w)$ does not hold.

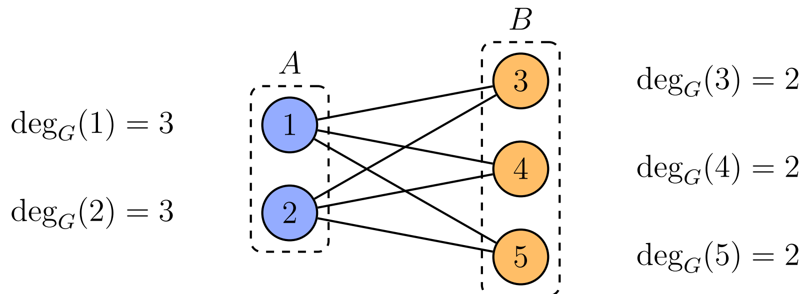



Figure 3: A simple (3,2)-biregular graph.

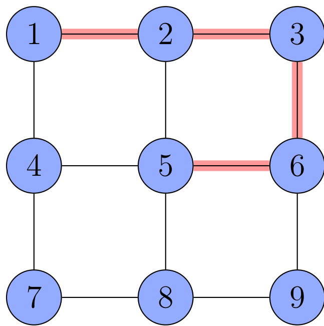
A graph is considered as *connected* if from every node one can traverse through the edges to all other nodes i.e. every pair of nodes need to also be connected. For instance, the graph in Figure 2 has two isolated vertex subsets $\{2, 5\}$ and $\{1, 3, 4\}$, therefore the graph is *disconnected*. On the other hand, the graphs in Figures 1b, 1c and 3 are connected. Let us not worry about the connectivity of the directed graph on Figure 1a as directed graphs are not relevant to this work other than in this section.

A *distance* between two nodes $u, v \in V$ is the length of the shortest path between $u$ and $v$ (see Figure 4b). We use the notation $\mathrm{dist}(u, v)$ for the distance between $u$

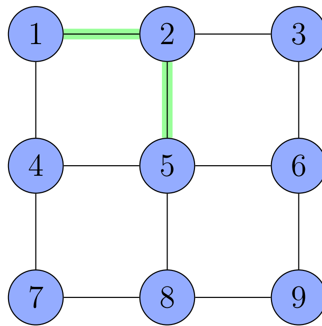
and $v$. The maximum of all shortest paths in a graph $G$ is called the *diameter* of $G$. A *subgraph* $H$ of a graph $G = (V, E)$ is a graph formed from a subset $V'$ of vertices of $V$ and a subset $E'$ of edges of $E$ such that each endpoint of $e' \in E'$ must be in $V'$. An $r$-radius *ball* of node $u$ of graph $G$ is a subgraph $H = (V', E')$ of $G$ containing:

- every vertex $v \in V$ such that the distance from $u$ to $v$ is at most $r$,
- every edge $(u', v') \in E$ such that $u'$ and $v'$ are also in $V'$.

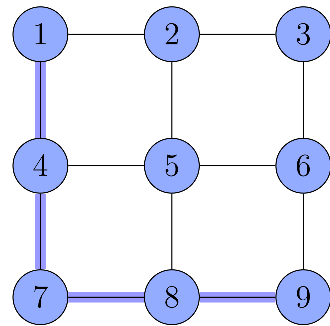
We use the notation $B_G(u, r)$ to denote the $r$-radius ball of node $u$ of graph $G$. For an example of a ball, see the Figure 5.




(a) A path between nodes 1 and 5 in a graph $G$. The length of the path is 4.

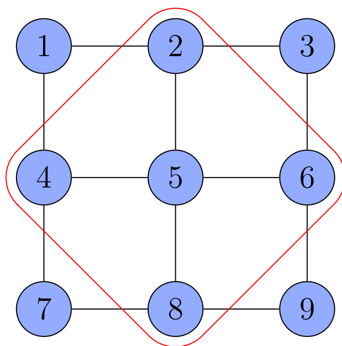



(b) The shortest path between nodes 1 and 5 in a graph $G$. The length of the path is 2, thus $\text{dist}(u, v) = 2$.

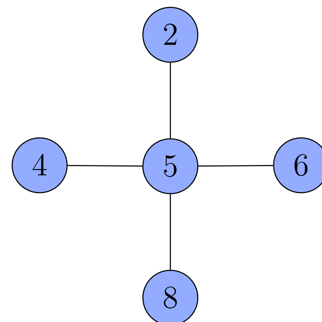



(c) A path between nodes 1 and 9 illustrating the diameter of the graph $G$. The length of the path is 4, thus $\text{diameter}(G) = 4$.

Figure 4: Examples of different kind of paths illustrated by highlighting the edges that form the path.




(a) The graph $G$ from Figure 4. Inside the red area is each node with distance at most 1 to node 5.



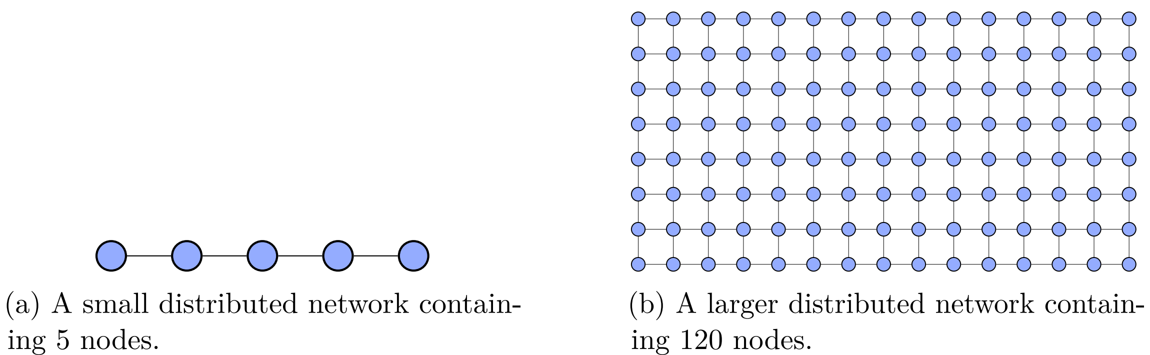

(b) A 1-radius ball $B_G(5, 1)$ is a subgraph of $G$.

Figure 5: Example of a ball.

## 2.2 Distributed Computing

Executing a computer program in several identical or different computers is called distributed computing [6]. It is similar to running a computer program that contains multiple concurrent tasks, in a single computer, but in distributed computing there are higher level tasks that are distributed to multiple different computers.

Computers are called nodes, and they are connected to each other with communication channels. These communication channels carry data from a node to another node. Together, nodes and communication channels form a network. A common way to visualize these networks is by drawing a graph in which the nodes represent computing nodes and edges represent the communication channels. [24]



(a) A small distributed network containing 5 nodes.

(b) A larger distributed network containing 120 nodes.

Figure 6: Examples of distributed networks.

According to Lamport [6], in the area of distributed computing, the term *model* denotes a view or abstract representation of a distributed system. There are multiple different computation models used in distributed computing [6]. The most important category of distributed computation models is *process models* [6]. In process models, the work or activities are represented as concurrently executed processes that execute their instructions sequentially [6]. A standard way to distinguish different process models from each other is to categorize them by the method they use to communicate with each other (*interprocess communication*) [6].

Message passing models are one form of a process model [6]. In the model, processes communicate by adding a message to a message queue, and the recipient process moves the message out (dequeues) from the message queue [6].

Different message passing models are widely used in the research field of distributed computing. The models can vary in different details, such as in the size of the message queues [6], in the size of the messages [10] and on how the nodes get identified [5], if they are identified at all [8]. We will dive deeper into the most relevant message passing models for this work in Sections 2.3 and 2.6.

An algorithm that is executed in a distributed fashion in a distributed network, is called as a distributed algorithm. Each node in a network is started simultaneously and will always execute the same algorithm. Initially the nodes are on the same state and there can be finitely or infinitely many states. Initially the nodes are aware of only themselves and of the connections to their neighbors. One might question that if every node starts with the same state, would not they also end up in the same

state? Well yes, this happens inevitably in a case where the nodes were not given any additional symmetry breaking inputs and if every node sees the same amount of neighbors [24].

In practice, every computer node on a network has a UUID (Universally unique identifier) that can be used to break the symmetry, and nodes are usually given some input data that they process, and nodes can always randomize data as they are never completely synchronized together, so this is not necessarily a problem. In theory, we explicitly have to assume that there exists these kinds of symmetry breaking elements. Distributed algorithms are deterministic in this work unless otherwise mentioned. In other research there might be randomizing involved.

## 2.3 Port-Numbering Model

This section is based on the textbook [24] unless otherwise mentioned.

Port-numbering model (PN model) is a rather weak model of computation that inherits from the message passing model. In the model, nodes do not have identification. A distributed algorithm that executes in a port-numbering model is called as a PN algorithm.

Communication channels start and end from communication ports. Each node has communication ports numbered from 1 to $d$, where $d$ is the degree of the node. The ports are numbered in an arbitrary order.

All nodes are considered identical; initially they are in the same internal state. Every node starts the execution simultaneously following the same PN algorithm $A$. The execution of $A$ is done synchronously in parallel. A communication round consists of following synchronous steps:

1. send a message to each port,
2. wait until all messages have been sent,
3. receive a message from each port,
4. update internal state.

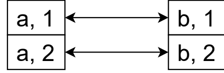
After each communication round, a node can optionally stop execution and announce its local output. All nodes are required to eventually stop. When all nodes have stopped, the algorithm is considered as stopped. The running time of the algorithm $A$ is the total communication rounds that took place.

## 2.4 Formalized Port-Numbering Model

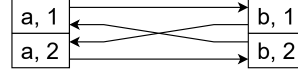
In Section 2.3 we briefly and informally introduced the PN model. Now in this section and in the following subsections we give a more formal definition to the PN model. As in the previous section, this section is based on the textbook [24] unless otherwise mentioned.

PN network is a 3-element tuple $N = (V, P, p)$, where $V$ and $P$ are the sets of vertices and ports respectively, and $p\colon P \to P$ is a function that maps a port to

another port, forming a communication channel. A port, an element of $P$, is a pair $(v, i)$ where $v \in V$ and $i \in \{1, 2, \dots\}$. Additionally, we assume that $p$ is an involution, that is, for all ports $x \in P$ we have $p(p(x)) = x$, i.e. each edge of the "underlying graph" is undirected. See Figures 7a and 7b for examples of valid and invalid PN networks.




(a) A PN network of two nodes, $a$ and $b$. Both $a$ and $b$ have a degree of 2, therefore 2 ports. Ports are $(a, 1), (a, 2), (b, 1)$ and $(b, 2)$. The connections are $p((a, 1)) = (b, 1)$, $p((a, 2)) = (b, 2)$, $p((b, 1)) = (a, 1)$ and $p((b, 2)) = (a, 2)$.




(b) An invalid PN network as port mapping function $p$ is not an involution: $p(p((a, 1))) = p(p((a, 2))) \neq (a, 1)$.

Figure 7: Examples of a valid and an invalid PN network.

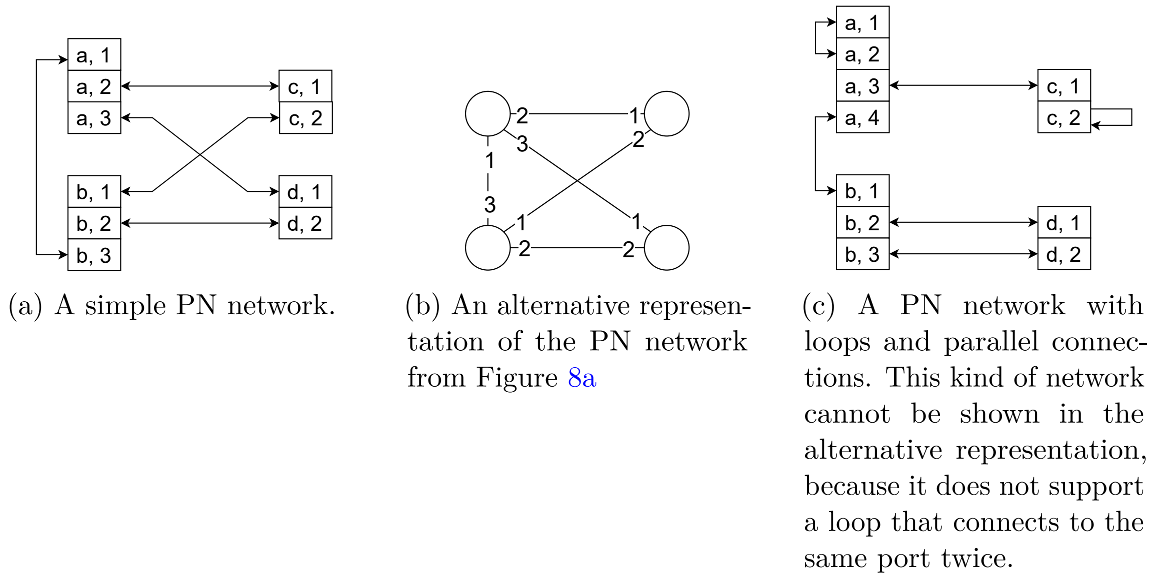
The degree $\deg_N(v)$ of a node $v \in V$ is equal to the number of ports of $v$. We assume that the port numbers are consecutive positive integers starting from 1, i.e. the ports of a node $v$ are $\{(v, i) \mid i \in \{1, 2, \dots, \deg_N(v)\}\}$. Note that the highest port number of $v$ is also the degree of $v$.

When we say *port number $i$ in node $v$*, we refer to the port $(v, i)$. When we say *port $(v, i)$ is connected to port $(w, j)$*, we refer to $p((v, i)) = (w, j)$.

A loop is a connection where port $(v, i)$ is connected to port $(v, j)$. For example there are two loops, $p((a, 1)) = (a, 2)$ and $p((c, 2)) = (c, 2)$, in Figure 8c.

There are multiple connections between two distinct nodes $v$ and $w$ if $p((v, i_1)) = (w, j_1)$, $p((v, i_2)) = (w, j_2)$, $i_1 \neq j_1$ and $i_2 \neq j_2$. For example there are connections $p((b, 2)) = (d, 1)$ and $p((b, 3)) = (d, 2)$ in Figure 8c.

If a PN network has neither loops nor multiple connections, it is called a *simple* PN network. For example the network is simple in Figure 8a.




(a) A simple PN network.

(b) An alternative representation of the PN network from Figure 8a

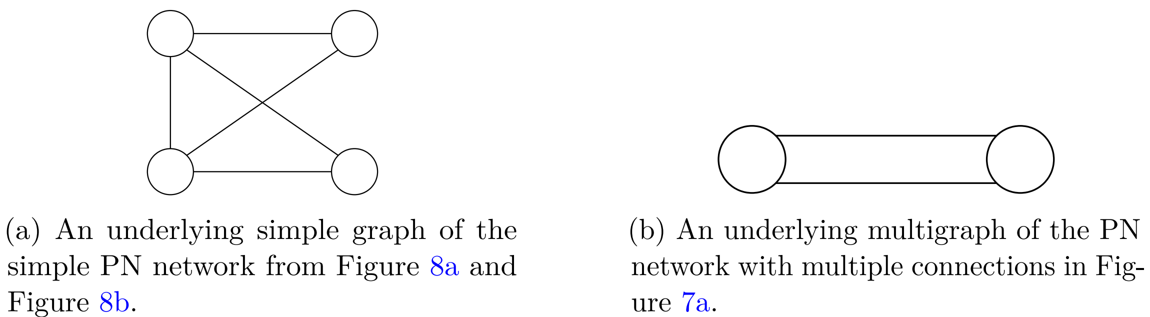
(c) A PN network with loops and parallel connections. This kind of network cannot be shown in the alternative representation, because it does not support a loop that connects to the same port twice.

Figure 8: Examples of simple and non-simple PN networks.

### 2.4.1 Underlying Graph

Each simple PN network $N = (V, P, p)$ has an *underlying graph* $G = (V, E)$. An edge $\{v, w\}$ is part of the underlying network $G$ if and only if $v$ is connected to $w$, that is, $E = \{\{v, w\} \mid p((v, i)) = (w, j)\}$. A network with multiple connections uses the same definition to determine an underlying network, but in that case $E$ has to be a multiset instead of a set, in order to allow multiple same edges.

In case an underlying graph of a network is connected, we say that *the network is connected.* Throughout this work, we assume that the introduced networks are connected.



(a) An underlying simple graph of the simple PN network from Figure 8a and Figure 8b.

(b) An underlying multigraph of the PN network with multiple connections in Figure 7a.

Figure 9: Examples of underlying graphs.

### 2.4.2 Distributed Graph Problems

Researchers are often interested in different distributed graph problems. These problems always have some kind of set of solution. As it turns out, many of these problems can be encoded using the node labelings.

Node labelings are a way to associate information with each node $v \in V$. It is defined as a function $f: V \to Y$, where $Y$ is an arbitrary set of labels.

We can use the concept of node labelings to represent subsets $X_1, X_2, \ldots, X_n \in V$ for some $n \in \mathbb{Z}_{>0}$ with node labeling $g: V \to \{1, 2, \ldots, n\}$ where $g(v) = i$ indicates that $v \in X_i$ and $v \notin X_j, j \neq i$. Note that the subsets $X_1, X_2, \ldots, X_n$ are disjoint, and $X_1 \cup X_2 \cup \cdots \cup X_n = V$.

Let $\Pi$ be a distributed graph problem. The value of $\Pi(N)$ is the set of solutions, where $N = (V, P, p)$ is a simple PN network. A solution $f \in \Pi(N)$ is a node labeling $f: V \to Y$, where $Y$ is some set of local outputs specific to the problem in question.

For example:

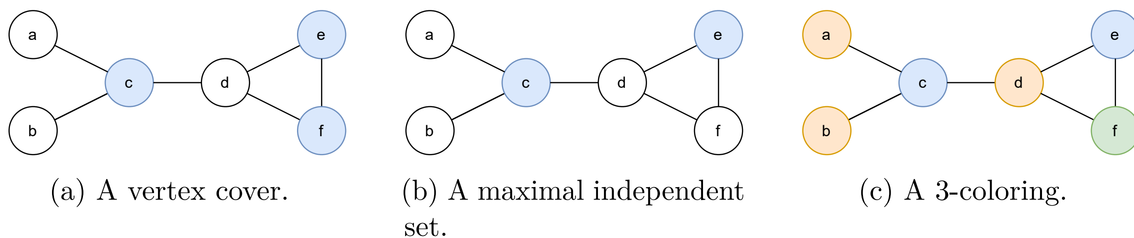
**Vertex cover** of a graph $G$ is a subset $V' \subseteq V$ of nodes which together *cover* all edges of the graph, that is, at least one endpoint from each edge must be in the subset $V'$. A node labeling $f$ is a solution $f \in \Pi(N)$ if $f$ encodes a vertex cover of the underlying graph $G$ of $N$.

**Independent set** of a graph $G$ is a subset $V' \subseteq V$ of nodes where no two nodes are adjacent. That is, for each $u, v, \in V'$ such that $u \neq v$, an edge $(u, v)$ is not in $E$. A node labeling $f$ is a solution $f \in \Pi(N)$ if $f$ encodes an independent set of nodes of the underlying graph $G$ of $N$.

**Maximal independent set** of a graph $G$ is an independent subset $V' \subseteq V$ of nodes such that there is no additional vertex $w \in V \backslash V'$ that can be included in $V'$ such that it stays as an independent set. A node labeling $f$ is a solution $f \in \Pi(N)$ if $f$ encodes a maximal independent set of nodes of the underlying graph $G$ of $N$.

**$k$-coloring** of a graph $G$ is partition of $V$ with $k$ subsets $X_1, X_2, \ldots, X_k$ where each subset is an independent set. A node labeling $f$ is a solution $f \in \Pi(N)$ if $f$ encodes a $k$-coloring of the underlying graph $G$ of $N$.

In the first three problems it seems like we are looking for only one subset of vertices $V' \subseteq V$. Indeed, but we also have the subset of vertices $V \backslash V'$, thus $n = 2$. Therefore, we can define $X_1 = V'$ and $X_2 = V \backslash V'$. In the problem $k$-coloring, we are looking for $k$ subsets of vertices $X_1, X_2, \ldots, X_k$, therefore $n = k$.




(a) A vertex cover. (b) A maximal independent set. (c) A 3-coloring.

Figure 10: Example solutions to different graph problems. In Figures 10a and 10b, the blue nodes form a node labeling that denotes a solution. There are two additional labels in Figure 10c, yellow and green, in order there to be 3 labels for 3 colors.

In Figure 10 we can see two examples of graph problem encodings. The node labeling in Figure 10a is a vertex cover. However, the node labeling in Figure 10b is not a vertex cover, because there is an edge $\{d, f\}$ and neither of the nodes are in the solution set $\{c, e\}$. The node labeling in Figure 10b is a maximal independent set. On the other hand, the node labeling in Figure 10a is not an independent set, because there is an edge $\{e, f\}$ and both $e$ and $f$ are in the solution set.

### 2.4.3 Distributed Algorithms in the PN Model

A *distributed algorithm* $A$ can be described as a state machine. The algorithm $A$ can possibly have infinitely many states or only finitely many states. The components of $A$ are:

1. $\mathrm{Input}_A$ is the set of local inputs,
2. $\mathrm{States}_A$ is the set of states,
3. $\mathrm{Output}_A \subseteq \mathrm{States}_A$ is the set of states that are considered as output states,
4. $\mathrm{Msg}_A$ is the set of messages.

The algorithm always consists of 3 functions, each defined for each degree $d \in \mathbb{N}$. The first function,

$$\mathrm{init}_{A,d} \colon \mathrm{Input}_A \to \mathrm{States}_A,$$

initializes the state of the calling node using the given input data. Each node calls the function $\mathrm{init}_{A,d}$ as its first function call.

After initialization, each node constructs messages for each of their neighbor by calling the second function:

$$\mathrm{send}_{A,d} \colon \mathrm{States}_A \to \mathrm{Msg}_A^d\,.$$

A node gives its current local state as an input to the function and the returning value is a $d$-element tuple of messages.

The third function allows nodes to receive messages from each of their neighbors. It is defined as

$$\mathrm{receive}_{A,d} \colon \mathrm{States}_A \times \mathrm{Msg}_A^d \to \mathrm{States}_A\,.$$

The function is given a node's current state and a $d$-element tuple of received messages. In return, it gives a state value that represents the node's new state. Additionally, whenever the current state is an output state $x \in \mathrm{Output}$, we require that the function $\mathrm{receive}_{A,d}$ returns the exact same state $x$.

### 2.4.4 Execution of PN Algorithm

As formerly mentioned, a distributed algorithm $A$ can be described as a state machine. Correspondingly the *execution* of $A$ can be described as the history of states of each node of a PN network $N = (V, P, p)$. Let $f \colon V \to \mathrm{Input}_A$ be a node labeling. A *state vector* is a function $x \colon V \to \mathrm{States}_A$. The execution of $A$ on $(N, f)$ is a sequence

of state vectors $x_0, x_1, \ldots$ where the $x_0$ is the initial state vector. When we use the notation $x_t, t \in \mathbb{Z}_{\geq 0}$, we refer to the state vector at time $t$. Similarly, we use $x_t(u), u \in V$ when we refer to the state of the node $u$ at time $t$.

We define

$$x_0(u) = \mathrm{init}_{A,d}(f(u))$$

where $u \in V$ and $d = \deg_N(u)$. The state vectors $x_1, x_2, \ldots$ are defined recursively in the following paragraphs.

We assume that we have defined $x_{t-1}$, and we show how we define $x_t$ using the assumption of $x_{t-1}$. Let function $m_t \colon P \to \mathrm{Msg}_A$ map a port to a message at time $t$. Let port $(v, j) \in P$, port $(u, i) = p((v, j))$, and degree $\deg_N(v) = d'$. Let $m_t((u, i))$ be $j$-th element of the vector $\mathrm{send}_{A,d'}(x_{t-1}(v))$. Here the element $m_t((u, i))$ is a message sent by node $v$ through communication port $(v, j)$. It is also the message received by node $u$ from port $(u, i)$.

We define the message vector

$$r_t(u) = (m_t((u, 1)), m_t((u, 2)), \ldots, m_t((u, \deg_N(u))))$$

for each $u \in V$. The message vector contains every message node $u$ receives at time $t$.

Now we give the final definition for $x_t(u)$:

$$x_t(u) = \mathrm{receive}_{A,d}(x_{t-1}(u), r_t(u)).$$

Next we define when the execution is considered as stopped, using the definition of execution. Algorithm $A$ *stops in time* $T$, if $x_T(u) \in \mathrm{Output}_A$ for each $u \in V$, that is, each node $u$ is in output state at time $T$. Algorithm $A$ *stops*, if $A$ stops in some finite time $T$.

Now that we know when $A$ stops, we define what is considered as the output of $A$. Assuming that $A$ stops in time $T$, we define the *output* of $A$ as $g = x_T$. With the same assumption, we define the *local output* of node $u$ as $x_t(u)$.

### 2.4.5 Solving Graph Problems in PN

As this thesis focuses strongly on graph problems, it is necessary to define what it means for a distributed algorithm to solve a graph problem.

A family of graphs is a set of graphs with same kind of properties. For example a set of all simple undirected graphs is a family of graphs.

Let $\mathcal{F}$ be a family of simple undirected graphs. Let us assume that $N = (V, P, p)$ is a simple PN network, the underlying graph $G$ of $N$ is in the family $\mathcal{F}$ and finally the input $f$ to the algorithm $A$ is in $\Pi'(N)$. We say that a distributed algorithm $A$ solves a problem $\Pi$ on a graph family $\mathcal{F}$ given problem $\Pi'$, if the execution of $A$ on $(N, f)$ stops and yields an output $g \in \Pi(N)$. We say that $A$ solves the problem in time $T$, if $A$ stops in time $T(|V|)$ where $T \colon \mathbb{N} \to \mathbb{N}$.

The input problem $\Pi'$ is often omitted. Then we say that a distributed algorithm $A$ solves a problem $\Pi$ on a graph family $\mathcal{F}$.

Often when we discuss some algorithms solving some problems, we state $\mathcal{F}, \Pi, \Pi'$ and $T$ implicitly. For example *algorithm $A$ finds a vertex cover in any bipartite graph* implies that $\mathcal{F}$ is bipartite graphs, $\Pi$ is the problem of finding a vertex cover and problem $\Pi'$ is omitted.

## 2.5 Covering Map

There exists a limitation to what problems can be solved in a deterministic PN model. The limitation comes from an underlying graph of a PN network being symmetric [5, 7].

In this section, we discuss a topological concept that can help us to formalize symmetries in a PN network. The concept is called *covering map*[1]. Later in this work, we use the concept in proving Lemmas 5.5 and 5.6.

First we want to show a definition of *covering* for graphs from the paper [4]:

> A graph $H$ is a *covering* of a graph $G$ if there is a way to label the nodes of $H$ with the names of nodes in $G$ in such a way that if a node $x$ of $H$ is labelled "v" then the labels of the neighbors of $x$ are precisely the neighbors of $v$ in G.

In this definition, covering map is the function that maps the nodes from $H$ to $G$. If $H$ is a covering of a connected graph $G$, then an intuition is that $H$ consists of multiple copies of $G$ "glued" together in a clever way.

In order for us to use covering maps with PN networks, we need to additionally consider the port numbers of nodes so that they are also preserver by the covering map.

We define now the covering map for PN networks similarly it is defined in the textbook [24].

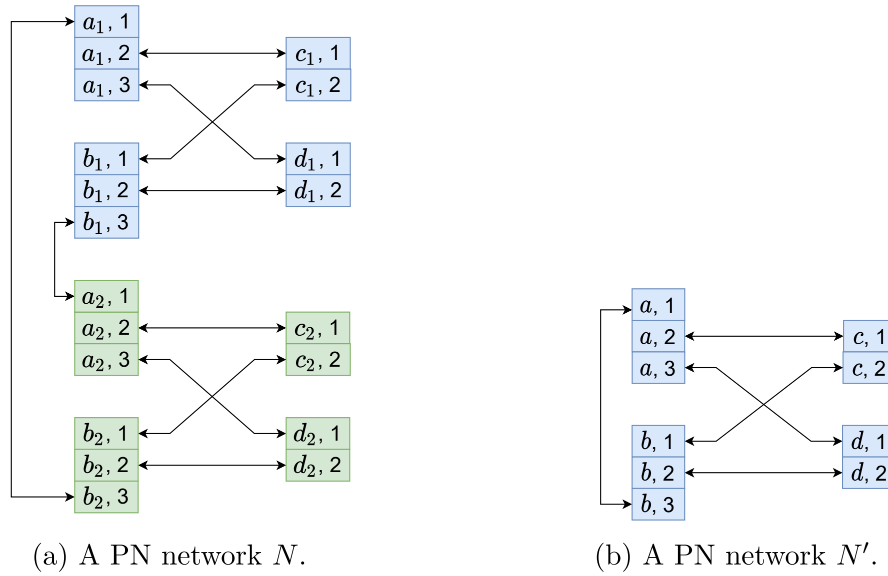
**Definition 2.2.** Let $N = (V, P, p)$ and $N' = (V', P', p')$ be PN networks and let $\phi\colon V \to V'$. The function $\phi$ is a covering map from $N$ to $N'$ if all the following hold:

1. $\phi$ is a surjection i.e. for every $v' \in V'$ there exists at least one $v \in V$ such that $\phi(v) = v'$.

2. $\phi$ preserves the degrees of nodes i.e. $\deg_N(v) = \deg_{N'}(\phi(v))$, for all $v \in V$.

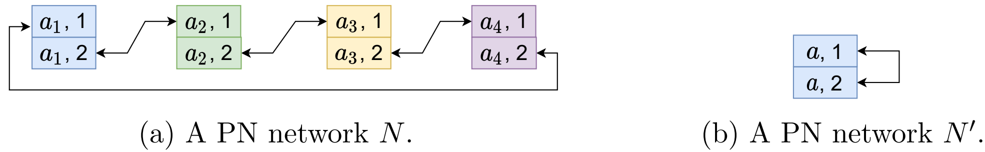
3. $\phi$ preserves port numbers and connections i.e. if $p((u, i)) = (v, j)$ then $p'((\phi(u), i)) = (\phi(v), j)$.

[1]Note that this is not related to the vertex cover problem even though they share the word "cover".




(a) A PN network $N$. (b) A PN network $N'$.

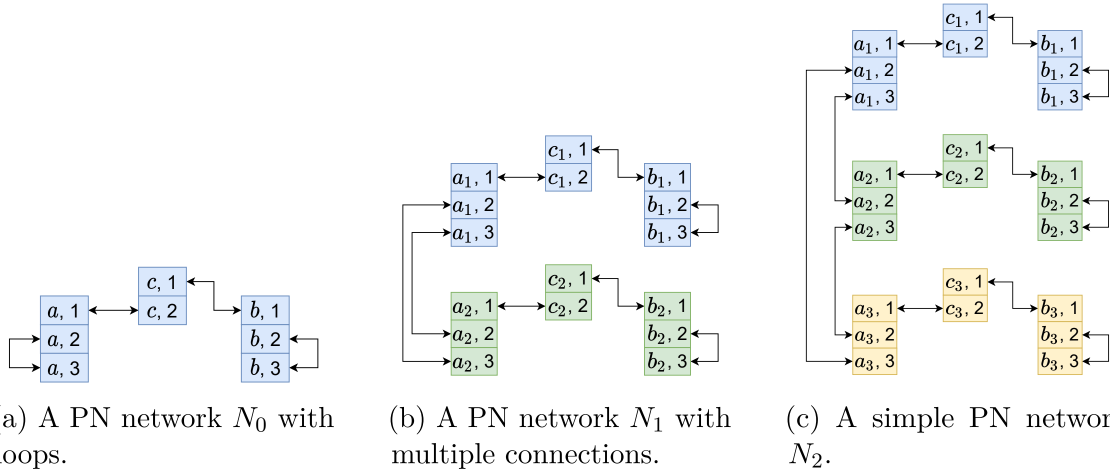
Figure 11: There is a covering map $\phi$ from $N$ to $N'$ that maps each $x_i$ to $x$ for all $x \in \{a, b, c, d\}$ and for all $i \in \{1, 2\}$.




(a) A PN network $N$. (b) A PN network $N'$.

Figure 12: There is a covering map $\phi$ from $N$ to $N'$ that maps each $a_i$ to $a$ for all $i \in \{1, 2, 3, 4\}$.

There are two examples of covering maps on Figure 11 and 12. In these examples we can see that the sizes of covering networks are integer multiplications of the sizes of original networks. This actually applies to every covering map, i.e. if we have a covering map $\phi\colon V \to V'$, then $|V| = k|V'|$ for some $k \in \mathbb{N}_{>0}$ [3]. We often call covering networks as *lifts*. A $k$*-lift* is a lift where the covering network $N$ is $k$ times the size of $N'$.




(a) A PN network $N_0$ with loops.

(b) A PN network $N_1$ with multiple connections.

(c) A simple PN network $N_2$.

Figure 13: There is a covering map $\phi_0$ from $N_1$ to $N_0$ that maps each $x_i$ to $x$ for all $x \in \{a, b, c\}$ and for all $i \in \{1, 2\}$. Similarly, there is a covering map $\phi_1$ from $N_2$ to $N_0$ that maps each $x_i$ to $x$ for all $x \in \{a, b, c\}$ and for all $i \in \{1, 2, 3\}$.

In the Figure 13 we can see that the network $N_1$ is a 2-lift of the network $N_0$ and the network $N_2$ is a 3-lift of the network $N_0$. The network $N_0$ has a loop $p(p((a, 2))) = p((a, 3)) = (a, 2)$. The networks $N_1$ and $N_2$ do not have loops, but instead they connect to other symmetrical equivalent parts of their networks (denoted by subscripts $1, 2, 3$), i.e. in the network $N_1$ we have connections $p(p((a_2, 2))) = p((a_1, 3)) = (a_2, 2)$ and $p(p((a_2, 3))) = p((a_1, 2)) = (a_2, 3)$. This symmetricity implies that each node ends up in an identical state with its symmetrical equivalent after each communication round, assuming the input problem $\Pi'$ does not break the symmetry. For example, when node $a_2$ from network $N_1$ uses its port 2, it cannot determine whether it is communicating with node $a_1$ or with itself, as $a_1$ and $a_2$ are always in identical states.

## 2.6 LOCAL Model

The LOCAL model is another message passing model that is used in the field of distributed computation. It inherits from the PN model, hence most of the definition is already done in Section 2.4. The difference is that the graph problem $\Pi'$, given as an input, is always predetermined. A solution to problem $\Pi'$ is a node labeling $\mathrm{id}\colon V \to \{1, 2, \ldots, |V|\}$, where each node has a unique label [5]. Sometimes the node labeling is defined $\mathrm{id}\colon V \to \{1, 2, \ldots, |V|^c\}$, where $c$ is a positive constant greater than 1 [24], but in this work we assume $c = 1$. If a distributed algorithm $A$ solves a problem $\Pi$ on a graph family $\mathcal{F}$ given $\Pi'$, we say that $A$ solves $\Pi$ on graph family $\mathcal{F}$ in the LOCAL model [24].

Earlier in Section 2.2 we discussed the limitation in a deterministic PN model that comes from symmetry. In the LOCAL model, an algorithm can utilize these identifiers to break the symmetry and avoid the limitation of the PN model.

A well-known property of the LOCAL model is the ability to compute *every* function of a graph $G$ in time $\mathcal{O}(\mathrm{diameter}(G))$. This amount of time is enough for

any node in $G$ to gather complete information of both the graph and the unique labels from function id, because a message can travel from a node to any other node at most in diameter($G$) time. With the complete information of everything, each node can compute the whole solution and output its own part of the solution. Because of this, researchers are usually interested in complexity classes below the $\mathcal{O}(\text{diameter}(G))$.

## 2.7 Locally Checkable Labeling Problems

In this section we define *locally checkable labeling (LCL)* problems generally. This section is intended as an introduction to Section 2.8 where we introduce an alternative definition of LCL problems that we will be using in this work. Briefly, LCL problems are a family of graph problems where a global solution (node labeling) can be verified locally by the individual nodes.

Now we define LCL problems using the formalism from the paper [9], and we also use the formalism from a more recent paper [31]. An LCL problem $\Pi$ consists of:

- a positive integer $r$, which is called the *radius* of $\Pi$,
- a finite set of *input labels* $\Sigma_{\text{in}}$,
- a finite set of *output labels* $\Sigma_{\text{out}}$ and
- a finite set of *locally consistent labelings* $C$, where:
  - the elements are pairs $(H = (V^H, E^H), s)$,
  - $H$ is a graph, node $s$ is in $V^H$, and the distance from the node $s$ to any other node in $V^H$ is at most $r$,
  - every pair $(v, e) \in (V^H \times E^H)$ is labeled with a pair from $\Sigma_{\text{in}} \times \Sigma_{\text{out}}$.

Given a graph $G = (V, E)$ and a node labeling $f : V \to \Sigma_{\text{in}} \times \Sigma_{\text{out}}$, the node labeling $f$ is a solution to an LCL problem $\Pi$ if for every node $v \in V$, the $r$-radius ball $B_G(v, r)$ is isomorphic to some labeled graph in $C$ [31].

Several common graph problems are in the LCL family. For example the problems introduced in Section 2.4.2 are LCL problems. We now define these problems using the LCL notation. The input is unnecessary for each of these problems, therefore we set the input labels $\Sigma_{\text{in}} = \{1\}$ for each problem. Each problem has radius 1. Listing the locally consistent labelings of each problem would result in substantially large sets of graphs, hence we give only a brief description of the elements.

**$k$-coloring.** The set of output labels $\Sigma_{\text{out}}$ contains all $k$ colors, e.g. $1, 2, \ldots, k$. The set $C$ contains all the possible 1-ball graphs of a given graph family with all possible vertex coloring combinations with $k$-colors.

**Maximal independent set.** The set of output labels $\Sigma_{\text{out}}$ contains labels 0 and 1. The set $C$ contains all the possible 1-ball graphs of a given graph family with all possible combinations of output labels such that if and only if the node

$s$ (the center node) is 1, then all adjacent nodes are labelled with 0. Label 0 denotes that the vertex is not in the maximal independent set, and label 1 denotes that it is in the set.

**Minimal vertex cover.** The set of output labels $\Sigma_{\text{out}}$ contains labels 0 and 1. The set $C$ contains all the possible 1-ball graphs of a given graph family with all possible combinations of output labels such that if and only if the node $s$ (the center node) is 0, then all adjacent nodes are labelled with 1. Label 0 denotes that the vertex is not in the minimal vertex cover, and label 1 denotes that it is in the set. [2]

It is clear that defining the whole set $C$ explicitly in this formalism would be a substantial amount of work, at least when radius $r$ is high. Therefore, we depend on another formalism of LCL problems that provides a more compact representation of the problems.

## 2.8 LCL Problems in Biregular Graphs

In this section, we will introduce an alternative formalism of LCL problems used in this work, that originates from paper [20], and recently it has been seen in the papers including [25] and [28]. We will be referring to these papers in the following definitions.

In the formalism, LCL problems are generally defined for *infinite* $\Delta$*-regular* $\delta$*-uniform hypertrees* [25]. Hypertrees are hypergraphs with tree structure. Hypergraph is a generalization of graphs. The concept of hypergraphs might first seem difficult to understand. It simply means that an edge is called as a hyperedge, and it can connect *any* number of nodes. To be $\delta$-uniform means that the hyperedges connect to exactly $\delta$ nodes. Note that if we fix $\delta = 2$, then the problems are defined for infinite $\Delta$-regular trees [25].

An LCL problem $\Pi$ is a tuple $(\Sigma, A, P)$, where $\Sigma$ is a finite set of labels, and $A$ and $P$ are finite sets containing all allowed *label configurations* of nodes and hyperedges, respectively. A label configuration is a multiset containing labels from the set $\Sigma$. The label configurations in $A$ have a length of $\Delta$ and the label configurations in $P$ have a length of $\delta$ [28].

The task of each node is to label each of its incident hyperedge with a label from $\Sigma$ [28]. Each node will label $\Delta$ incident hyperedges, thus each hyperedge will be labeled with $\delta$ labels. The labels given by a node to its incident hyperedges form a label configuration. Similarly, the labels given to a hyperedge form a label configuration. Label configurations of a node have a length of $\Delta$ and label configurations of a hyperedge have a length of $\delta$. We require from a solution that:

1. every label configuration of a node is contained in $A$, and
2. every label configuration of a hyperedge is contained in $P$.

[2] This problem is a complement of the maximal independent set.

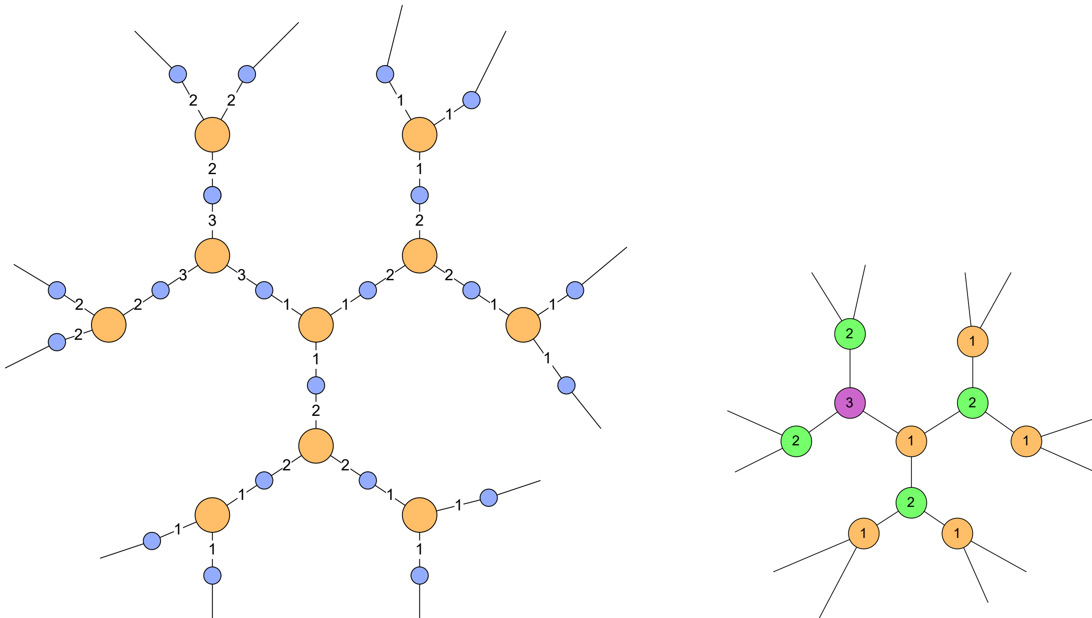
Alternatively to $\Delta$-regular $\delta$-uniform hypertrees, we can think of the hyperedges as separate nodes [25]. This way the problems are defined for infinite $(\Delta, \delta)$-biregular trees. We say that the original nodes of the graph are called *active nodes*, and the newly introduced nodes, formerly hyperedges, are called *passive nodes*. In this setting the passive nodes do not output anything. Only the active nodes output their labels.

We showed earlier examples of LCL problems in Section 2.7. Now we show some of the problems in the alternative formalism.

The sets $\Sigma$, $A$ and $P$ of $k$-coloring in infinite (3,2)-biregular graphs are defined as:

$$\begin{aligned}
\Sigma &= \{1, 2, \ldots, k\}, \\
A &= \{\{1,1,1\}, \{2,2,2\}, \ldots, \{k,k,k\}\}, \\
P &= \{\{a, b\} \mid a, b, \in \Sigma \text{ and } a \neq b\}.
\end{aligned}$$

An active node has a color between 1 and $k$. All label configurations of active nodes correspond to a single color. The label configurations of passive nodes ensure that the neighbors of a passive node do not share a color. For an example, see Figure 14.



(a) An infinite $(3, 2)$-biregular tree. Orange nodes are active nodes, and smaller blue nodes are passive nodes. The labels seen in the graph are part of a solution to 3-coloring problem.

(b) A visualization of the coloring in graph 14a. The graphs are identical, but the passive nodes are seen as 2-uniform hyperedges, or just as edges.

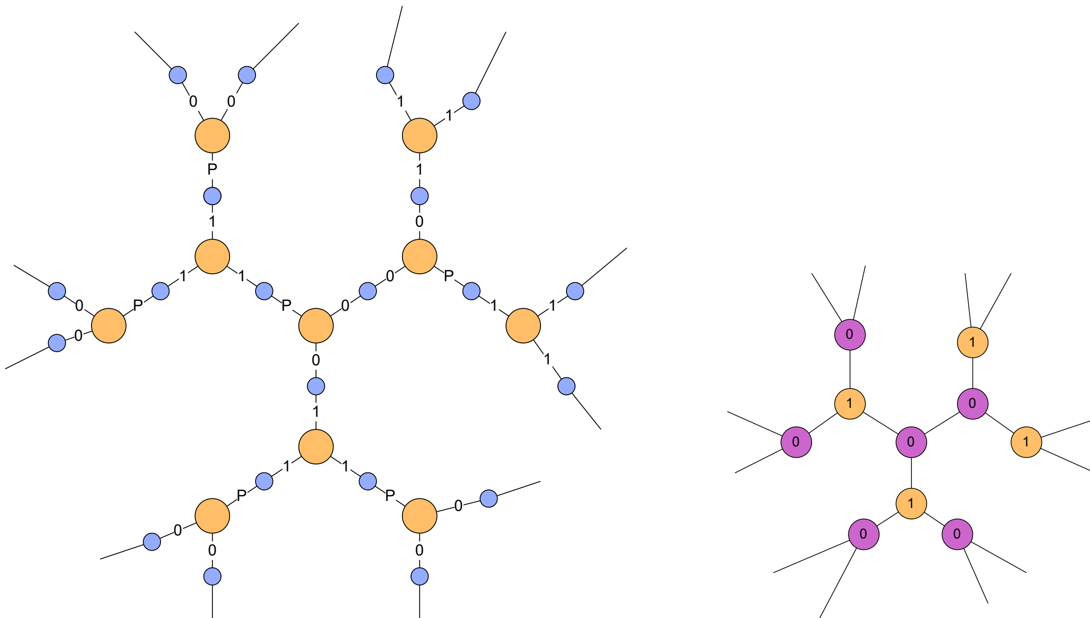
Figure 14: 3-coloring in infinite $(3, 2)$-biregular tree.

The sets $\Sigma$, $A$ and $P$ of maximal independent set in infinite (3,2)-biregular graphs

are defined [24] as :

$$
\begin{aligned}
\Sigma &= \{\mathrm{I}, \mathrm{O}, \mathrm{P}\}, \\
A &= \{\{\mathrm{I}, \mathrm{I}, \mathrm{I}\}, \{\mathrm{P}, \mathrm{O}, \mathrm{O}\}\}, \\
P &= \{\{\mathrm{I}, \mathrm{P}\}, \{\mathrm{I}, \mathrm{O}\}, \{\mathrm{O}, \mathrm{O}\}\}.
\end{aligned}
$$

An active node with label configuration $\{\mathrm{I}, \mathrm{I}, \mathrm{I}\}$ is considered to be in the maximal independent set and label configuration $\{\mathrm{P}, \mathrm{O}, \mathrm{O}\}$ means that the node is not in the set. The label configurations of passive nodes ensure that the output of active nodes is an independent set, and the label P ensures that the set is maximal [24]. For an example, see Figure 15.



(a) An infinite $(3, 2)$-biregular tree. Orange nodes are active nodes, and smaller blue nodes are passive nodes. The labels seen in the graph are part of a solution to maximal independent set problem.

(b) Same graph as in 15a, but the passive nodes are seen as 2-uniform hyperedges, or just as edges, and the nodes are now colored.

Figure 15: Maximal independent set in infinite $(3, 2)$-biregular tree.

One might wonder why are we interested in infinite $(\Delta, \delta)$-biregular trees and not finite trees? The reason for it is that irregularities in these trees, e.g. a leaf node, potentially makes the problems easier to solve [22], and we do not want that.

# 3 Research Questions

There are LCL problems that cannot be solved in the PN model. More specifically, there exist LCL problems such that no PN algorithm can solve it in every PN network. We are interested in detecting the unsolvability of an LCL problem automatically.

**Research question 1.** Can we automatically detect the unsolvability of an LCL problem in the PN model?

We answer this question in Section 5, where we present our algorithm that can automatically detect the unsolvability of an LCL problem in the PN model.

We also explore whether showing unsolvability in the PN model has some interesting implication in the LOCAL model.

**Research question 2.** Can we prove that an unsolvability of an LCL problem in the PN model implies that the problem is also not solvable in constant time in the LOCAL model?

As our answer to Research question 2, we prove Theorem 5.9 in Section 5.3. The proof relies on multiple lemmas that we introduce in Sections 5.1, 5.2, and 5.3. With the theorem, all results from the algorithm can be derived to non-constant lower bounds in the LOCAL model.

We have implemented the algorithm in Section 5 that detects the unsolvability of an LCL problem in the PN model, and we introduce details of the implementation in Section 6. There also exists a database of upper and lower bounds of LCL problems [35], and these bounds have been computed using various tools called *classifiers*. Our implementation is one kind of classifier, and we want to know if it can find any new lower bounds i.e. lower bounds that are better than the current known bounds for these problems.

**Research question 3.** Can we automate the detection of a new lower bound for an LCL problem in the LOCAL model?

The algorithm given in Section 5 can be run with a single problem, but ideally we want an algorithm that works for large classes of problems. Thus, we want our implementation to be reasonable fast in order it to have use in practice.

**Research question 4.** Can we make the implementation fast enough to classify large classes of problems?

As an answer to Research questions 3 and 4, we present the results from our implementation in Section 7. The results will include new lower bounds for some LCL problems and execution times from classifying large classes of problems.

# 4 Prior Work

In Section 4.1, we generally discuss the *automation of finding* information about the best possible algorithm for a problem, and we briefly discuss what has been previously done on this topic in general. In Section 4.2 we narrow our focus on the subject to the field of distributed computing. In addition, we also discuss relevant automated tools that have been developed prior to this work and the complexity classes that are relevant in the LOCAL model.

## 4.1 Automation of Proving

Given any computational problem, we want to find the best possible (i.e. optimal) algorithm for the problem. We like to learn something new about the computability of the problem in terms of computational complexity, in order to narrow the search for the best algorithm. There are usually two things that we would want to discover about a problem.

- An efficient algorithm that solves the problem. The existence of such algorithm would immediately give us an upper bound on the complexity of the problem.
- An efficient algorithm does not exist at all. Then we want to obtain a proof of nonexistence. This would give us a lower bound on the complexity of the problem.

Results, showing existence, an upper bound, or possibility, are called *positive* results. Similarly, the results, showing nonexistence, a lower bound, or impossibility, are called *negative* results. In our work, the objective is to find negative results, but we would also like to briefly discuss finding positive results as it is the other side of discovering new information about the computability of computational problems.

Traditionally, positive and negative results for computational problems are found using the pen-and-paper method. For example, Linial [5] showed in 1987 that there is a deterministic $\mathcal{O}(\log^* n)$ algorithm for $\mathcal{O}(\Delta^2)$-coloring. In a more recent paper from 2010, Barenboim and Elkin [11] showed that there is a deterministic *polylogarithmic* time algorithm for $\Delta^{1+o(1)}$-coloring, which is substantially better in terms of the number of colors, in comparison to the former one [5] from Linial.

A more recent approach to finding positive or negative results is the *automation* of the whole process. One could automate the finding of positive or negative results by utilizing massive computational power of modern computers. The case of negative results is exactly one of our work's objectives, and we will discuss the automation of finding these results in the distributed computing in Section 4.2. But first, let us discuss the automation of finding positive results.

The automation of finding positive results involves creating a tool that takes a specification of desired behavior of an algorithm as an input, and outputs an algorithm that matches the specification. This method is called *algorithm synthesis* [17]. For example, Dramnesc and Jebelean [15] manage to automatically synthesize various sorting algorithms including selection sort, insertion sort, quick sort, merge sort and

a novel sorting algorithm they call *unbalanced merge sort*. In another work, Gulwani et al. [12] present a tool that can efficiently synthesize highly nontrivial 10–20 line loop-free bit vector programs.

Propositional satisfiability solvers (SAT solvers) have been used in various synthesis problems to find positive results. SAT solvers have been used especially in synthesis of circuits, where one goal is to find optimal Boolean circuits in terms of size. For example, Järvisalo et al. [13] present a method to encode an ensemble computation problem as a propositional formula. The encoding is then fed to a SAT solver to obtain either a working circuit design for the ensemble computation problem or a proof of nonexistence. We use SAT solvers in our implementation to find negative results, and we discuss more about them in Sections 6.2 and 6.3.

## 4.2 Automation of Proving in the LOCAL Model

A tool that automates the finding of positive or negative results in distributed computing is called *classifier*. A classifier can automatically determine a lower bound or an upper bound for an LCL problem. Currently, we are aware of only small number of classifiers in the LOCAL model, and we list them below in Table 1.

Generally, there are infinitely many complexity classes possible because there are infinitely many functions. However, in the LOCAL model, many of them turns out to be redundant. *Gap theorems* show that there are ranges of complexities where there exists no optimal algorithms for LCL problems. For example, Balliu et al. [19] showed that the deterministic complexity of an optimal algorithm in the LOCAL model is either at most $\mathcal{O}(1)$ or requires at least $\Omega(\log^* n)$ rounds. In another work, Chang et al. [16] showed that there is a gap between $\mathcal{O}(\log^* n)$ and $\Omega(\log n)$. Currently, the complexity classes we care about, in the deterministic LOCAL model, are the following:

$$\mathcal{O}(1), \Theta(\log^* n), \Theta(\log n), \text{ and } n^{\Theta(1)}.$$

| Classifier | Complete | Labels | Paths | Cycles | Trees | |
|---|---|---|---|---|---|---|
| | | | | | Rooted | Unrooted |
| poly-classifier [39, 37] | Yes[1] | Any | No | No | Yes[2] | Yes[2] |
| Rooted tree classifier [34, 32] | Yes | Any | No | No | Yes[2] | No |
| Automata-theoretic lens classifier [29, 33] | Yes | Any | Yes | Yes | No[3] | No |
| Round Eliminator [25, 21, 20] | No | Any[4] | Yes | No | No | Yes |
| Tree-classifications (dataset) [30] | No | 2, 3 | No | No | No | Yes |
| TLP Classifier [27, 26] | No | 3 | Yes | No | No | Yes |

[1] Complete only in rooted trees.
[2] Only in binary trees. In theory, it applies to all regular trees.
[3] The paper [33] shows that it applies to some LCL problems in rooted trees.
[4] Up to $\frac{64}{\Delta}$.

Table 1: A list of all classifiers we are aware of prior this work.

In Table 1, we list for each classifier along with the graph families they target to, the amount of labels they support and the completeness. We define a classifier

to be *complete* if it can output a tight complexity class, from the previous list of 4 main complexities, for any input problem in the problem class it targets to. There are multiple reasons for a classifier to be *incomplete*, including:

- it only outputs either upper or lower bound,
- it outputs “no result” i.e. it does not know the complexity class,
- it does not terminate.

Our classifier works only with unrooted trees and is incomplete in the sense that it gives only lower bounds, and only for some LCL problems. In the table, we list the number of labels each classifier supports, and by ‘Any’ we mean that a classifier supports reasonably high quantity of labels.

# 5 Unsolvability in the PN Model

In this section, we will introduce the basic idea of an algorithm that finds a proof that an LCL problem is impossible to solve in the PN model. If the algorithm detects the problem is unsolvable, it outputs a multigraph as a result. In Sections 5.1 and 5.2, we show that this multigraph is a proof that the problem is unsolvable in the PN model. In Section 5.3, we prove that if an LCL problem is unsolvable in the PN model, then the problem is not solvable in constant time in the LOCAL model.

We first formally define the meaning of solvability of an LCL problem in the PN model.

**Definition 5.1.** An LCL problem $\Pi$ is solvable in PN model, if and only if there exists a PN algorithm $A$ that finds a solution for $\Pi$ in all PN networks.

We can define an alternative version of the Definition 5.1 using contraposition.

**Corollary 5.2.** *An LCL problem $\Pi$ is not solvable in PN model, if and only if there does not exist a PN algorithm $A$ that finds a solution for $\Pi$ in all PN networks.*

We have illustrated two examples of unsolvable LCL problems in Tables 2 and 3, where the values Y (Yes) and N (No) answer to the question: *can the algorithm in this row solve the problem in the network of this column?* Value of Y/N indicates that the value does not matter for the purpose of the example. In both examples, there are only a finite number of networks and algorithms for the sake of simplicity. In Table 2, the problem $\Pi$ is unsolvable, because there is at least one network for each algorithm, where the algorithm fails to solve the problem. In Table 3, we show an example that is a special case of Corollary 5.2. The problem $\Pi'$ is also unsolvable, but this time no algorithm can solve it in network $N'_{n'}$.

| $\Pi$ | $N_1$ | $N_2$ | $N_3$ | $\cdots$ | $N_n$ |
|---|---|---|---|---|---|
| $A_1$ | N | Y/N | Y/N | $\cdots$ | Y/N |
| $A_2$ | Y/N | N | Y/N | $\cdots$ | Y/N |
| $A_3$ | Y/N | Y/N | N | $\cdots$ | Y/N |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\ddots$ | $\vdots$ |
| $A_m$ | Y/N | Y/N | Y/N | $\cdots$ | N |

Table 2: Unsolvable LCL problem $\Pi$. Each algorithm fails to solve $\Pi$ in at least some network, denoted by “N” (No).

| $\Pi'$ | $N'_1$ | $N'_2$ | $N'_3$ | $\cdots$ | $N'_{n'}$ |
|---|---|---|---|---|---|
| $A'_1$ | Y/N | Y/N | Y/N | $\cdots$ | N |
| $A'_2$ | Y/N | Y/N | Y/N | $\cdots$ | N |
| $A'_3$ | Y/N | Y/N | Y/N | $\cdots$ | N |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\ddots$ | $\vdots$ |
| $A'_{m'}$ | Y/N | Y/N | Y/N | $\cdots$ | N |

Table 3: Unsolvable LCL problem $\Pi'$. Each algorithm fails to solve $\Pi'$ at least in network $N'_{n'}$ (the last column containing only values “N”).

Not every unsolvable LCL problem necessarily fall within the case shown in Table 3. Nevertheless, we are interested in it, because it seems more feasible to find a single network where all algorithms fail, in comparison to finding a network for each algorithm separately. The special case is written as follows.

**Lemma 5.3.** *An LCL problem $\Pi$ is not solvable in PN model, if there exists a PN network $N$ such that no PN algorithm $A$ can solve the problem $\Pi$ in network $N$.*

*Proof.* Let $\Pi$ be an LCL problem. Let $N$ be a PN network such that no PN algorithm $A$ can solve $\Pi$ in $N$. Therefore, no PN algorithm $A$ can find a solution to $\Pi$ in all PN networks. According to Corollary 5.2, the problem $\Pi$ is unsolvable. □

From Lemma 5.3 we can say that, to show that a problem $\Pi$ is unsolvable in the PN model, it is sufficient to find a counterexample—a PN network $N$ in which the problem $\Pi$ cannot be solved. We also show that a problem is not solvable in a PN network if it is not solvable in the underlying graph represented by the network.

**Lemma 5.4.** *If a problem $\Pi$ is not solvable in graph $G$, then $\Pi$ is also not solvable in any PN network $N$ that has $G$ as its underlying graph.*

*Proof.* Let us assume that a problem $\Pi$ is not solvable in graph $G$, and $\Pi$ is solvable in some PN network $N$ that has $G$ as its underlying graph. This means that network $N$ has a valid labeling, thus its underlying graph $G$ has also a valid labeling. This is a contradiction, therefore the original implication must be true. □

With the fact from Lemma 5.4, we create an algorithm that automatically tries to find a counterexample—a graph for which the given LCL problem is not solvable. The algorithm is written as follows.

**Algorithm 1** Counterexample graph finder

**Require:** $1 \leq n_{low} \leq n_{high}$

1: **function** FIND($n_{low}, n_{high}, \Pi$) ▷ Graph bounds $n_{low}$ and $n_{high}$, LCL problem $\Pi$
2: $\quad\Delta \leftarrow$ ACTIVEDEGREE($\Pi$)
3: $\quad\delta \leftarrow$ PASSIVEDEGREE($\Pi$)
4: $\quad$**for** $n \leftarrow n_{low}, n_{high}$ **do** ▷ Iterate graph sizes from lowest to highest
5: $\quad\quad G_n \leftarrow$ GENERATEGRAPHS($n, \Delta, \delta$)
6: $\quad\quad$**for each** $G \in G_n$ **do**
7: $\quad\quad\quad$**if not** SOLUTIONEXISTS($\Pi, G$) **then**
8: $\quad\quad\quad\quad$**return** $G$ ▷ Return counterexample.
9: $\quad\quad\quad$**end if**
10: $\quad\quad$**end for**
11: $\quad$**end for**
12: $\quad$**return** ▷ No counterexample found. Return nothing.
13: **end function**

The Algorithm 1 is designed to find the smallest graph for which an LCL problem $\Pi$ is not solvable. It starts from graphs with $n_{low}$ vertices and goes up to graphs with $n_{high}$ vertices. We define $\Delta$ and $\delta$ to be the active and passive degree of the problem $\Pi$ respectively (rows 2 and 3). The variable for the current vertex count is called $n$ (row 4). For each $n$, we generate all possible $(\Delta, \delta)$-biregular multigraphs with GENERATEGRAPHS($n, \Delta, \delta$), and save the multigraphs into variable $G_n$ (row 5). Now, for each multigraph $G \in G_n$ (row 6) we check if the given problem $\Pi$ has a solution using the function SOLUTIONEXISTS($\Pi, G$) (row 7). If a solution does not exist, we return the multigraph as a counterexample (row 8). In the case there are

no counterexamples in multigraphs between $n_{low}$ and $n_{high}$ vertices, the algorithm returns nothing (row 12).

Up to this moment, we have not discussed how the function

$$\text{GENERATEGRAPHS}(n, \Delta, \delta)$$

from row 5 actually generates the multigraphs, nor have we discussed how the function

$$\text{SOLUTIONEXISTS}(\Pi, G)$$

from row 7 determines the existence of a solution. These functions are implementation specific and in this section we assume that they exist as black boxes. We discuss our implementations of these functions later in Section 6. However, we can discuss why the algorithm generates multigraphs instead of simple graphs. The algorithm generates multigraphs, because they are less restrictive than simple graphs. Thus, we generate more graphs when multiple edges are allowed, which implies more possibilities to find a counterexample.

As we said before, we use the LCL formalism from Section 2.8. The formalism uses infinite $(\Delta, \delta)$-biregular trees, but the algorithm generates finite $(\Delta, \delta)$-biregular multigraphs. In Section 5.1, we show if we can find $(\Delta, \delta)$-biregular network with multiple connections, where the problem is unsolvable, then it implies that it is also unsolvable in some simple $(\Delta, \delta)$-biregular network. This can be derived to work with graphs with Lemma 5.4. In Section 5.2, we show that if an LCL problem is unsolvable in finite connected $(\Delta, \delta)$-biregular graph $G$ with cycles, then it is also unsolvable in some infinite $(\Delta, \delta)$-biregular tree $G'$. Finite $(\Delta, \delta)$-biregular graphs are always with cycles, when both $\Delta$ and $\delta$ are greater than 1, and this is in practice always the case, because we are not interested in biregular graphs with $\Delta = 1$ or $\delta = 1$ as they are trivial.

## 5.1 From Multiple Connection Networks to Simple Networks

In this section we show that if an LCL problem is not solvable in PN networks with multiple connections, then the problem is also not solvable in simple PN networks.

**Lemma 5.5.** *If an LCL problem $\Pi$ is not solvable in a PN network $N$, then it is also not solvable in any PN network $N'$ that is a lift of $N$.*

*Proof.* Since problem $\Pi$ has no solution in some PN network $N$, any algorithm $A$ will produce an invalid solution to $\Pi$ i.e. a local constraint is violated at least at some node $v$. Let $\phi : V' \to V$ be a covering map from $N' = (V', P', p')$ to $N = (V, P, p)$. Theorem 7.1 from the textbook [24] shows that the nodes of $N'$ will have exactly the same state as their counterparts at $N$ for each time unit $t = 0, 1, \ldots$ . Hence, if we run algorithm $A$ on both networks $N$ and $N'$, the local constraint violation at some node $v \in N$ also appears in all nodes $v' \in V'$ such that $\phi(v') = v$ i.e. the local constraint violations also appear in the covering network $N'$. $\square$

**Lemma 5.6.** *If there is a PN network $N_2$ with at most $k$ connections between any two nodes, then there exists a $k$-lift $N_1$ of $N_2$ such that $N_1$ is a simple PN network.*

*Proof.* Let $N_2 = (V_2, P_2, p_2)$, and $\mathrm{mul}(u, v)$ be the number of connections between any nodes $u, v \in V_2$. Thus, we can say that $k = \max(\{m(u, v) \mid u, v \in V_2\})$. Let $\mathrm{M} : \mathbb{N}_{>0} \to \{1, ..., k\}$ be a function defined as $\mathrm{M}(x + hk) = x$ for all $x = 1, ..., k$ and $h \in \mathbb{N}$. For example $\mathrm{M}(1 + hk) = 1$ and $\mathrm{M}(k + hk) = k$.

Let $N_1 = (V_1, P_1, p_1)$ be another network such that:

- For each $v \in V_2$, there are $k$ copies in $V_1$, namely $v_1, v_2, ..., v_k \in V_1$. Thus, the sizes $|V_1|$ and $k|V_2|$ are equal.
- For each port $(v, i) \in P_2$, we have ports $(v_x, i) \in P_1$ where $x = 1, 2, ..., k$.
- For each multiple connection $p_2((v, i_a)) = (u, j_a)$ where $a = 1, 2, ..., \mathrm{mul}(u, v)$, we have $p_1((v_x, i_a)) = (u_{\mathrm{M}(x+a-1)}, j_a)$. Here $i_a$ is $a$-th port number of $v$ that connects to $u$, and $j_a$ is $a$-th port number of $u$ that connects to $v$. Note that if $\mathrm{mul}(u, v) = 1$, then $p_1((v_x, i_1)) = (u_{\mathrm{M}(x)}, j_1) = (u_x, j_1)$.

Now we show that there is a covering map $\phi : V_1 \to V_2$. Let $\phi(v_x) = v \in V_2$ for each $v_x \in V_1$ where $x = 1, 2, ..., k$. We will show that $\phi$ is a covering map using the Definition 2.2:

- By the definition of $\phi$, it is surjective.
- For each connection in $N_2$, we have $k$ similar connections in $N_1$, therefore degrees of each node are preserved.
- For each connection $p_1((v_x, i_a)) = (u_{\mathrm{M}(x+a-1)}, j_a)$ we have

  $$\begin{aligned} &p_2((\phi(v_x), i_a)) = (\phi(u_{\mathrm{M}(x+a-1)}), j_a) \\ \Leftrightarrow\ &p_2((v, i_a)) = (u, j_a). \end{aligned}$$

  Both $(v, i_a)$ and $(u, j_a)$ are in $P_2$ and $p_2((v, i_a)) = (u, j_a)$, therefore for each connection, the port numbers and connections are preserved.

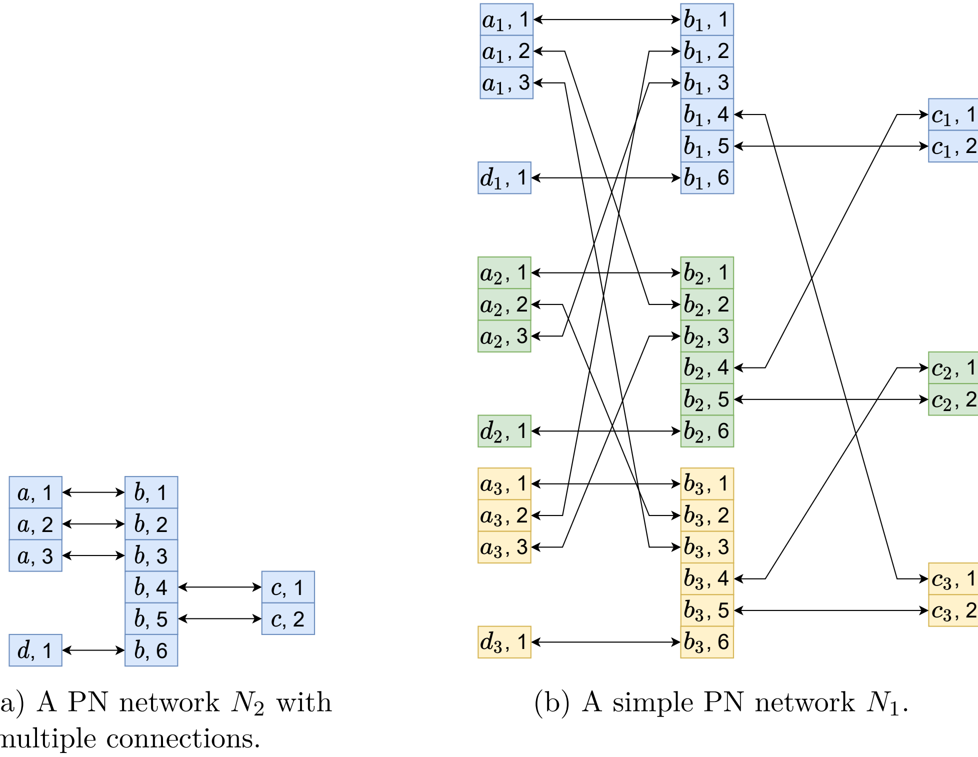
From above, we can conclude that $\phi$ is a covering map from $V_1$ to $V_2$ therefore $N_1$ is a $k$-lift of $N_2$.

We need to show that $N_1$ is a simple PN network i.e. it does not have multiple connections. Note that for each connection $p_1((v_x, i_a)) = (u_{\mathrm{M}(x+a-1)}, j_a)$ where $a = 1, 2, ..., \mathrm{mul}(\phi(v_x), \phi(u_{\mathrm{M}(x+a-1)}))$, nodes $u_{\mathrm{M}(x+a-1)}$ are distinct for all $a$ because $1 \le a \le \mathrm{mul}(\phi(v_x), \phi(u_{\mathrm{M}(x+a-1)})) \le k$ and there are exactly $k$ many "u" nodes in $V_1$, namely nodes $u_1, u_2, ..., u_k \in V_1$. Thus, for each node $v_x$, all connections from $v_x$ are mapped to distinct nodes, therefore $N_1$ is simple.

The network $N_2$ is connected by the assumption (Section 2.4.1) but it might not be clear that $N_1$ is connected, thus we show next that this is the case.

Let us look at all connections in $p_1$ and fix $a = 1$. Then $p_1((v_x, i_a)) = (u_{\mathrm{M}(x+a-1)}, j_a) = p_1((v_x, i_1)) = (u_x, j_1)$. This shows that we can traverse from any node $w_x \in P$ to any other node $w'_x \in P$ only using nodes with subscript $x$.

There are at least some nodes $u, v \in P_2$ such that $\mathrm{mul}(u, v) = k$, therefore we can traverse from any $u_x \in P_1$ to any $v_y \in P_1$ for any $x, y \in \{1, 2, ..., k\}$. Thus, $N_1$ is connected. $\square$




(a) A PN network $N_2$ with multiple connections.

(b) A simple PN network $N_1$.

Figure 16: The network $N_1$ is a 3-lift of the network $N_2$.

**Lemma 5.7.** *If an LCL problem $\Pi$ is not solvable in PN network $N_2$ with multiple connections, then it is also not solvable in simple PN network $N_1$ that is a lift of $N_2$.*

*Proof.* Let us assume that an LCL problem $\Pi$ is not solvable in PN network $N_2$ with multiple connections. Lemma 5.6 shows that there exists a $k$-lift $N_1$ of $N_2$ such that $N_1$ is a simple PN network. Lemma 5.5 shows that the problem $\Pi$ is also not solvable in network $N_1$. Therefore, we deduce that the problem $\Pi$ is not solvable in simple PN network $N_1$ that is a lift of $N_2$. □

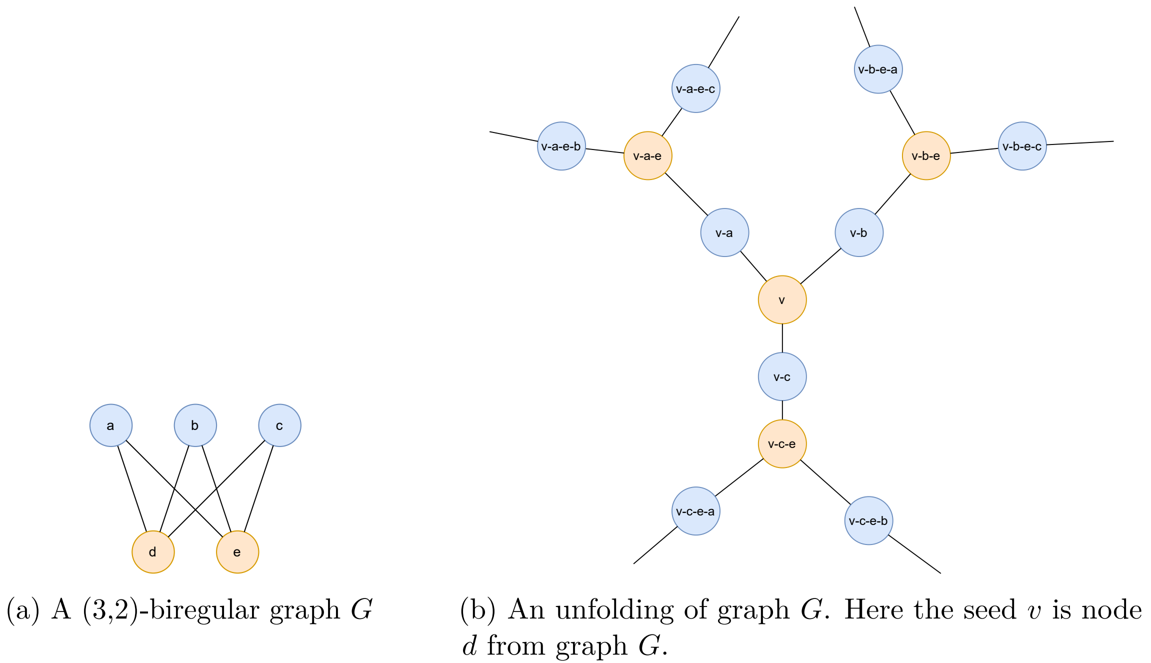
## 5.2 From Finite to Infinite

Our algorithm outputs a finite connected $(\Delta, \delta)$-biregular multigraph with cycles, but in the LCL formalism given in Section 2.8, graphs are infinite $(\Delta, \delta)$-biregular trees. Lemma 5.7 proves that unsolvability in PN network with multiple connections imply unsolvability in simple networks, but we also need to show how unsolvability in finite networks with cycles imply unsolvability in infinite tree networks. We will prove this using graphs, and then from Lemma 5.4, it immediately applies also to networks.

**Lemma 5.8.** *If an LCL problem $\Pi$ is unsolvable in finite connected $(\Delta, \delta)$-biregular graph $G$ with cycles, then it is also unsolvable in some infinite $(\Delta, \delta)$-biregular tree $G'$.*

*Proof.* We construct an infinite $(\Delta, \delta)$-biregular tree $G' = (V', E')$ by *unfolding* graph $G$. Let us fix a node $v \in V$. A non-backtracking walk is a walk that does not immediately visit a previous node at any point. Each node in $G'$ represents a non-backtracking walk in $G$ starting at node $v$. There is an edge between two nodes $u'$ and $v'$ in $G'$, if the walk presented by $u'$ can be obtained by the walk represented by $v'$ and one additional edge in $G$. Graph $G$ has cycles, therefore there are infinite walks of infinite length. The walks are non-backtracking, therefore the constructed graph $G'$ must be $(\Delta, \delta)$-biregular. The graph $G'$ is a covering graph of $G$, because for each node $v' \in G'$, the 1-radius neighborhood of $v'$ is preserved. Using Lemma 5.5, we can conclude that the problem $\Pi$ must also be unsolvable in $G'$. □

Now from Lemmas 5.4, 5.7, and 5.8, we can say that if Algorithm 1 outputs a graph, then the problem is unsolvable, and the output graph is a proof of it.




(a) A (3,2)-biregular graph $G$

(b) An unfolding of graph $G$. Here the seed $v$ is node $d$ from graph $G$.

Figure 17: A graph $G$ and its unfolding.

## 5.3 From PN to LOCAL

In this section we prove the following theorem:

**Theorem 5.9.** *If an LCL problem is unsolvable in the PN model, then the problem is also not solvable in constant time in the LOCAL model.*

First, we introduce related lemmas, and at the end of this section, we prove Theorem 5.9. The lemmas form a chain of implications shown in Figure 18. In the figure, there are two models we have to briefly introduce. Both models are similar to the LOCAL and PN models in a sense that they are also message passing models. We define them briefly as follows.

The first model is called *order-invariant* (OI) [9]. The OI model can be thought as a LOCAL model with a restriction where each node can only compare the identifiers but not access their exact value. In the model, the output of an algorithm does not change even if identifiers of nodes are changed as long as their relative order is preserved.

The second model is called *port numbering + orientation* (PO) [8]. In the PO model, ports are numbered like in the PN model, and additionally the edges have an orientation [18].

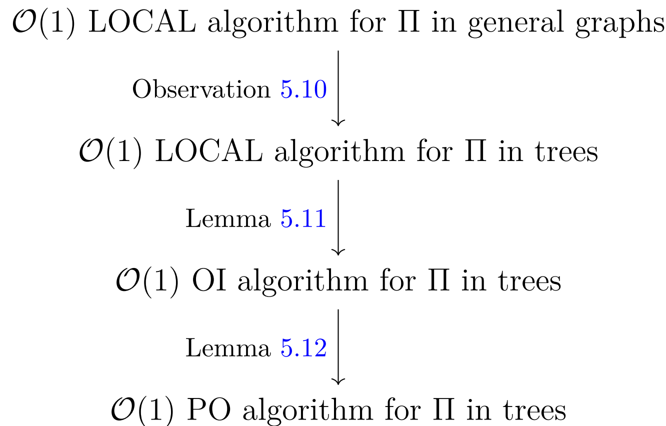



Figure 18: Here are all the implications we show in this section. Each implication is a step towards showing Theorem 5.9.

**Observation 5.10.** *If an LCL problem is solvable in constant time in the LOCAL model, then the problem is solvable in constant time in trees in the LOCAL model.*

*Proof.* This is trivial, because general graphs contain the graph family of trees. □

**Lemma 5.11.** *If an LCL problem is solvable in constant time in trees in the LOCAL model, then the problem is solvable in constant time in trees in the OI model.*

This has been proved for any fixed graph family in the work from Naor and Stockmeyer [9, theorem 3.3]. The proof applies Ramsey theory to show that there exists a finite set $Y$ of identifiers such that for all $X_1, X_2 \subseteq Y$ with $|X_1| = |X_2| = s$,

we have a constant time LOCAL algorithm giving identical results in any graph family whether we use identifiers $X_1$ or $X_2$. It shows that the constant time LOCAL algorithm applied to identifiers from set $Y$ behaves like an OI algorithm.

**Lemma 5.12.** *If an LCL problem is solvable in constant time in trees in the OI model, then the problem is solvable in constant time in trees in the PO model.*

Suppose there is a $t$-time PO algorithm that solves an LCL problem. The work from Göös et al. [18] shows that a PO algorithm can simulate an OI algorithm to solve the problem in $t$-time.

Now we prove Theorem 5.9:

*Proof of Theorem 5.9.* Let us have an LCL problem $\Pi$ that is unsolvable in the PN model. In the LCL formalism, we have active and passive nodes. Let $N = (V, P, p)$ be an infinite $(\Delta, \delta)$-biregular tree network. Let $u, v \in V$ be distinct active nodes such that both $u$ and $v$ are adjacent to passive node $w \in V$. We can define an orientation between $u$ and $v$ by looking at the ports of node $w$. Let the port that connects from $w$ to $u$ be $(w, i)$, and let the port that connects from $w$ to $v$ be $(w, j)$. Port numbers $i$ and $j$ are distinct, therefore we can say that if $i < j$, then $u < v$. Otherwise, we say that $v < u$. Now, if we interpret the passive nodes as $\delta - uniform$ hyperedges, we can say that we have an orientation between any two adjacent nodes. Thus, when we work in the LCL formalism, we are actually already working in the PO model. Thus, $\Pi$ is also unsolvable in the PO model.

Now, as shown in Figure 18, Observation 5.10, and Lemmas 5.11 and 5.12 form a chain of implications. This means that when there does not exist a constant time PO algorithm in trees for some LCL problem, there also does not exist a constant time LOCAL algorithm for the problem in general graphs. The problem $\Pi$ is unsolvable in the PN model and in the PO model. Therefore, problem $\Pi$ cannot be solved in constant time in the LOCAL model for general graphs. $\square$

Now we have proved Theorem 5.9. Therefore, the results from our algorithm now imply the problem is not solvable in constant time in the LOCAL model.

# 6 Implementation

In this section, we will explain all the modules that we use in the implementation of the Algorithm 1. The implementation itself is a computer program [38], that attempts to automatically find a proof of unsolvability for an LCL problem in the PN model. Although we have used Rust programming language [45] for the implementation, we keep the explanations independent of any programming language.

In Section 6.1, we discuss how we generate multigraphs, and describe the function $\textsc{GenerateGraphs}(n, \Delta, \delta)$. In Section 6.2, we discuss Boolean satisfiability problems (SAT), and in Section 6.3, we discuss how we use SAT to implement the function $\textsc{SolutionExists}(\Pi, G)$. As we are interested in classifying not just one LCL problem, but a class of problems, we need to know how to generate them. In Section 6.4, we discuss how we generate LCL problems. Lastly, we discuss some optimizations in our implementation in Section 6.5.

## 6.1 Generating Multigraphs

A naive way to generate $(\Delta, \delta)$-biregular multigraphs is to generate all possible multigraphs, and then filtering out graphs that are not in the graph family. Note that there are $2^{\frac{n(n-1)}{2}}$ different simple graphs of size $n$. The number of simple graphs grows exponentially fast, as shown in Table 4. Generating all graphs of size $n$ may take forever on modern computers even for small values of $n$.

| $n$ | Edges | Graphs |
|---:|---:|---:|
| 1 | 0 | 1 |
| 2 | 1 | 2 |
| 3 | 3 | 8 |
| 4 | 6 | 64 |
| 5 | 10 | 1024 |
| 6 | 15 | 32768 |
| 7 | 21 | 2097152 |
| 8 | 28 | 268435456 |
| 9 | 36 | 68719476736 |
| 10 | 45 | 35184372088832 |
| 11 | 55 | 36028797018963968 |
| $\vdots$ | $\vdots$ | $\vdots$ |

Table 4: A list of number of edges in complete graphs, and the number of simple graphs.

In this experiment we did not even consider multiple edges. With multiple edges without any limits, the number of multigraphs is infinite. However, this is not true when we consider only $(\Delta, \delta)$-biregular multigraphs of size $n$. Clearly the number of multiple edges is bounded by $\max(\Delta, \delta)$.

We call the parts of a biregular graph as $A$ and $B$ (see Section 2.1). The numbers of incident edges for nodes of parts $A$ and $B$ are $\Delta$ and $\delta$, respectively. There are only $n-1$ possibilities for pair $(|A|,|B|)$:

$$(1,n-1),(2,n-2),...,(n-2,2),(n-1,1).$$

We know from Equation 5 that the sums of degrees of each node in each part must be equal. Therefore, we need to consider only pairs $(n_A,n_B)$ such that the following hold

$$\Delta n_A = \delta n_B, \tag{6}$$

$$n_A + n_B = n. \tag{7}$$

If we assume that $\Delta \geq \delta$, then we can see from Equation 6 that $n_A \leq n_B$, therefore we only need to consider pairs

$$(1,n-1),(2,n-2),...,(\left\lfloor \frac{n}{2} \right\rfloor, n-\left\lfloor \frac{n}{2} \right\rfloor).$$

For example with $(3,2)$-biregular multigraphs and $n=10$, we can only have the pair $(4,6)$, as shown in Table 5. However, most of the time there are no valid pairs for chosen $n$, as we can see for $n=9$ in Table 6.

| $n_A$ | $n_B$ | $\Delta n_A$ | $\delta n_B$ |
|---|---|---|---|
| 1 | 9 | 3 | 18 |
| 2 | 8 | 6 | 16 |
| 3 | 7 | 9 | 14 |
| **4** | **6** | **12** | **12** |
| 5 | 5 | 15 | 10 |

Table 5: Possible pairs of node counts for part A and B in a $(3,2)$-biregular multigraph when $n=10$. The only valid pair with $\Delta n_A = \delta n_B$ is (4, 6), and it is bolded.

| $n_A$ | $n_B$ | $\Delta n_A$ | $\delta n_B$ |
|---|---|---|---|
| 1 | 8 | 3 | 16 |
| 2 | 7 | 6 | 14 |
| 3 | 6 | 9 | 12 |
| 4 | 5 | 12 | 10 |

Table 6: Possible pairs of node counts for part A and B in a $(3,2)$-biregular multigraph when $n=9$. There are no valid pairs.

Equations 6 and 7 are linear equations with two unknown variables $n_A$ and $n_B$, therefore there is at most one solution. Let us assume that the number of vertices in $n_A$ is $x$. From the equations we know that

$$n_B = n - x,$$
$$\Delta x = \delta(n-x),$$
$$\text{and therefore } n = \frac{x(\Delta+\delta)}{\delta}.$$

For some fixed $\Delta$ and $\delta$, $n$ may not always be an integer at all integer values of $x$. However, we can ensure integer solutions by assigning $x = y\frac{\delta}{\gcd(\delta,\Delta)}$, where $y$ is

any positive integer. Then we iterate through all $y$ up to some upper bound (see Table 7). We then compute all valid pairs of $(n_A, n_B)$ as shown in Table 8.

| $y$ | $x = y\frac{\delta}{\gcd(\delta,\Delta)}$ | $n = y\frac{(\Delta+\delta)}{\gcd(\delta,\Delta)}$ |
|---|---|---|
| 1 | 2 | 5 |
| 2 | 4 | 10 |
| 3 | 6 | 15 |
| 4 | 8 | 20 |
| $\vdots$ | $\vdots$ | $\vdots$ |

Table 7: All values of $y$, $x$, and $n$ for $(3, 2)$-biregular multigraph.

| $n$ | $n_A = x$ | $n_B = n - x$ |
|---|---|---|
| 5 | 2 | 3 |
| 10 | 4 | 6 |
| 15 | 6 | 9 |
| 20 | 8 | 12 |
| $\vdots$ | $\vdots$ | $\vdots$ |

Table 8: All valid values of $n$ and pairs of $(n_A, n_B)$ for $(3, 2)$-biregular multigraph.

Using the above observations, we can generate all possible graph sizes for $(\Delta, \delta)$-biregular multigraphs faster.

Before we discuss the function GenerateGraphs$(n, \Delta, \delta)$, we need to define isomorphism. Two graphs $G_1$ and $G_2$ are said to be isomorphic, if there is a bijection $f$ between vertices of $G_1$ and $G_2$ such that $f$ preserves the adjacency of vertices. In other words, they are similar if we ignore the labels on the vertices. Working with isomorphic graphs is redundant. Therefore, we are interested only in nonisomorphic graphs. There is a well known software package called *nauty and Traces* [14], that has tools for generating all nonisomorphic graphs of different graph families. The collection of tools is called *gtools*. In Table 9, we show the total number of simple graphs as well as the number of nonisomorphic simple graphs possible for different values of $n$. As we can see, there are significantly fewer nonisomorphic simple graphs than there are simple graphs. Thus, we want to generate only nonisomorphic graphs. For our implementation of GenerateGraphs$(n, \Delta, \delta)$, we use the tools *genbg* and *multig* from *gtools* as follows.

| $n$ | Simple graphs | Nonisomorphic simple graphs | Time (s) |
|---|---|---|---|
| 1 | 1 | 1 | 0.00 |
| 2 | 2 | 2 | 0.00 |
| 3 | 8 | 4 | 0.00 |
| 4 | 64 | 11 | 0.00 |
| 5 | 1024 | 34 | 0.00 |
| 6 | 32768 | 156 | 0.00 |
| 7 | 2097152 | 1044 | 0.00 |
| 8 | 268435456 | 12346 | 0.00 |
| 9 | 68719476736 | 274668 | 0.09 |
| 10 | 35184372088832 | 12005168 | 3.57 |
| 11 | 36028797018963968 | 1018997864 | 295.13 |
| $\vdots$ | $\vdots$ | $\vdots$ | |

Table 9: The numbers of simple graphs, and nonisomorphic simple graphs. There is also generation time for each nonisomorphic set of simple graphs. Nonisomorphic graphs were generated with the tool called *geng*. The tool was executed on a single thread of AMD Ryzen 9 3900X 12-Core Processor. For example, the command for generating all 10-sized connected simple graphs is `geng 10 > simple10.txt`.

The first tool, *genbg*, is used to generate small bipartite graphs that are nonisomorphic. The tool requires the sizes of parts $A$ and $B$ as an input, and we generate the sizes as we have shown earlier in this section. By default, the tool outputs the graphs to standard output stream in a format called *graph6*. A graph in *graph6* format is a one line of ASCII characters. We additionally specify two ranges of degrees so that we get bipartite graphs, where the degrees of nodes range from 1 to $\Delta$ and from 1 to $\delta$ in parts $A$ and $B$, respectively. We also use a flag `-c` to generate only connected graphs. The output from the graph is then fed into the second tool.

The second tool, *multig*, is used to generate small multigraphs with a given underlying graph. It replaces edges of a simple graph with multiple edges in all possible ways. All graphs outputted from *multig* are also nonisomorphic. The tool takes an exact range of edges we want for the graphs, and we already know that the number is exactly $\Delta n_A = \delta n_B$. We also give the tool a maximum edge multiplicity of $\Delta$ and upper bound of $\Delta$ on maximum degree, as we assume that $\Delta \geq \delta$. By feeding the bipartite graphs from *genbg* to *multig*, we will receive $(\Delta, \delta)$-biregular multigraphs. The output format from *multig* is a simple text format (different from *graph6*) that is quite trivial to parse, so we do not discuss more about it.

To conclude, our implementation of $\textsc{GenerateGraphs}(n, \Delta, \delta)$ first generates the pair $(n_A, n_B)$ with given $n$ and degrees $(\Delta, \delta)$. If the pair does not exist, the function returns an empty set. Otherwise, we continue to the second step where we generate the bipartite graphs using *genbg*, as described before. The last step is to feed the graphs from *genbg* to *multig* as described before. The result from *multig* is the set of all connected nonisomorphic $(\Delta, \delta)$-biregular multigraphs of size $n$.

In Table 10, we have listed the number of graphs generated in the second and

third steps, when $(\Delta, \delta) = (3, 2)$.

| $n$ | $n_A$ | $n_B$ | Simple bipartite graphs | Time (s) | (3,2)-biregular multigraphs | Time (s) |
|---|---|---|---|---|---|---|
| 5 | 2 | 3 | 4 | 0.00 | 4 | 0.00 |
| 10 | 4 | 6 | 24 | 0.00 | 42 | 0.00 |
| 15 | 6 | 9 | 272 | 0.01 | 658 | 0.00 |
| 20 | 8 | 12 | 4146 | 0.23 | 13902 | 0.04 |
| 25 | 10 | 15 | 79052 | 7.60 | 357219 | 1.37 |
| 30 | 12 | 18 | 1747977 | 224.70 | 10509351 | 47.28 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

Table 10: The counts of simple bipartite graphs and (3,2)-biregular multigraphs, when $(\Delta, \delta) = (3, 2)$. Both graph families are connected and nonisomorphic. We used the tools *genbg* and *multig*, as discussed above, and we feed the output from *genbg* to *multig*. We also show the time it takes to generate each set of graphs. The tools were executed on a single thread of AMD Ryzen 9 3900X 12-Core Processor.

## 6.2 Boolean Satisfiability Problem

Boolean satisfiability problem (SAT) is the problem of determining if there exists a truth value assignment to the variables of a given formula with respect to which the formula is true. A propositional formula is *satisfiable*, if there exists an assignment that results in true, otherwise it is *unsatisfiable*. For example, formula $a \wedge \neg b$ is satisfiable, because it is true if $a$ is true ($a = \top$) and $b$ is false ($b = \bot$), and formula $a \wedge \neg a$ is unsatisfiable, because it is always false.

Conjuctive normal form (CNF) is a standard form to describe propositional formulas. It is a conjuction (logical AND) of clauses, where each clause is a disjunction of variables (logical OR). That is, CNF is an AND of ORs, for example:

$$(a \vee d \vee \neg b) \wedge (c \vee a) \wedge (\neg a \vee \neg b).$$

The only allowed operators in CNF are AND ($\wedge$), OR ($\vee$) and NOT ($\neg$).

SAT was the first problem proved to be NP-complete [1]. A problem is said to be NP-complete if its solution is fast to verify, but there is no known way to solve the problem fast. However, there are tools called SAT solvers, that attempt to solve SAT as fast as possible [46]. These solvers usually accept a propositional formula in DIMACS CNF format [44]. It is a text format that represents a propositional formula in CNF form.

## 6.3 Our SAT Encoding

The function $\textsc{SolutionExists}(\Pi, G)$ checks if an LCL problem $\Pi$ is unsolvable in some $(\Delta, \delta)$-biregular multigraph $G$. To perform the checking, it is enough to show that there exists no valid labeling of $\Pi$ in $G$. In the implementation, this is done with the following steps:

1. encode the problem $\Pi$ and multigraph $G$ into a CNF formula $S$,
2. solve the problem $S$ using a fast SAT solver.

When we feed the CNF formula $S$ into a SAT solver, we get as a result either $\top$ (satisfiable) or $\bot$ (unsatisfiable). In case the result is $\bot$, the SAT solver found no possible labeling for the problem. Thus, we have found a counterexample, and we are done. Otherwise, we can continue searching using other multigraphs. The routine is illustrated in Figure 19.

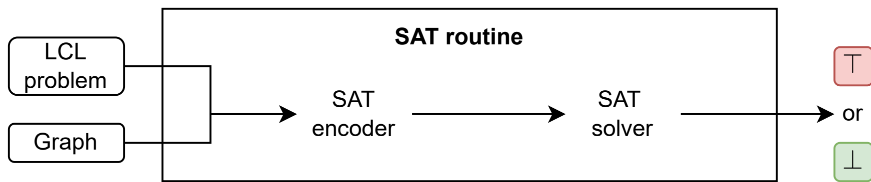



Figure 19: The SAT routine. When given a multigraph and an LCL problem, it checks if a valid labeling exists.

We repeat the routine for each multigraph, as shown in the Algorithm 1, and terminate early if the result is $\bot$. This is illustrated in Figure 20.

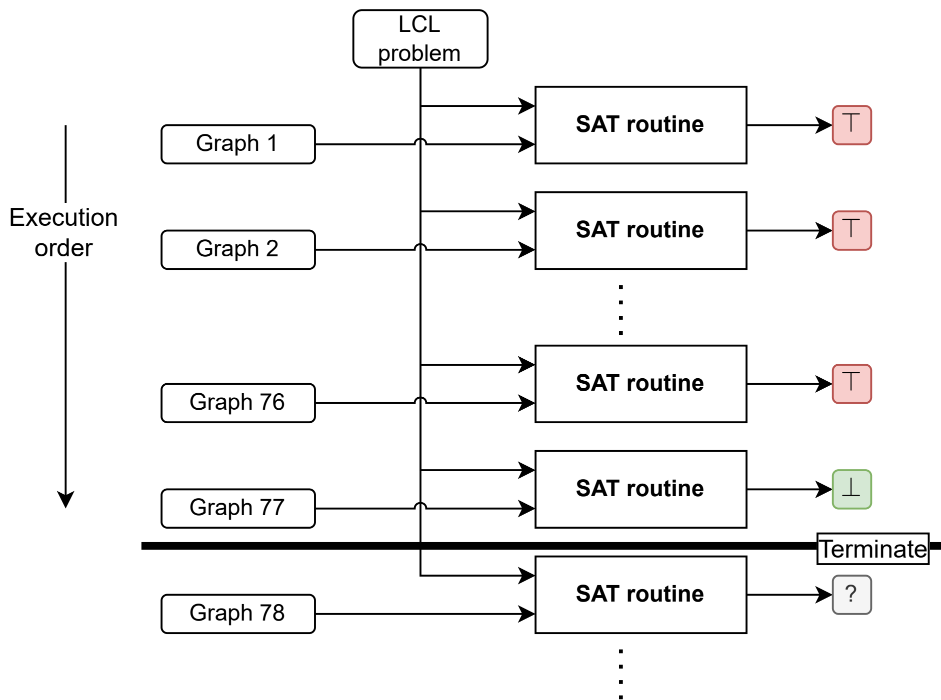



Figure 20: An example of an execution of multiple SAT routines. Execution terminates at the 77th multigraph, when the first unsatisfiable result is encountered.

Now we discuss how we encode an LCL problem $\Pi = (A, P, \Sigma)$ from class $(\Delta, \delta, \lambda)$, and a connected $(\Delta, \delta)$-biregular multigraph $G = (V, E)$, where $V$ is a union of two

disjoint sets $V_A$ and $V_P$, into a CNF formula. Each active node can have possibly any label configuration from $A$ (respectively passive nodes and $P$), and there are always as many ways to arrange the labels of a label configuration as there are permutations of the label configuration. Let $\mathrm{Perm}(A)$ and $\mathrm{Perm}(P)$ be the sets of all permutations of label configurations in sets $A$ and $P$, respectively.

We define variables $L_{A,u,v,l}$ for each $(u, v)$ incident on an active node $u$ and for each possible label $l \in \Sigma$. We also define variables $R_{A,u,p}$ for each permutation of label configurations in $A$ and each active node $u$. Similarly, we define variables $L_{P,u,v,l}$ and $R_{P,u,p}$.

We have 3 conditions for our encoding. Each condition is applied twice, once for active nodes and once for passive nodes.

1. Each node has to agree with their adjacent node on the labels of each edge between them.

   1a. For each $e = (u, v) \in E, u \in V_A$ and each label $(l_1, l_2) \in \Sigma^2, l_1 \neq l_2$, we require that at most one of the variables $L_{A,u,v,l_1}, L_{P,v,u,l_2}$ is true, i.e. clause $(\neg L_{A,u,v,l_1} \vee \neg L_{P,v,u,l_2})$.

   1b. For each $e = (u, v) \in E, u \in V_P$ and each label $(l_1, l_2) \in \Sigma^2, l_1 \neq l_2$, we require that at most one of the variables $L_{P,u,v,l_1}, L_{A,v,u,l_2}$ is true, i.e. clause $(\neg L_{P,u,v,l_1} \vee \neg L_{A,v,u,l_2})$

2. Each node can only be applied one permutation of label configurations.

   2a. For an active node $u \in V_A$ and each permutation $p \in \mathrm{Perm}(A)$, we require that only one $R_{A,u,p}$ is true, therefore we add clause $(R_{A,u,p_1} \vee R_{A,u,p_2} \vee \dots \vee R_{A,u,p_{|\mathrm{Perm}(A)|}})$ which ensures at least one permutation, and clauses $(\neg R_{A,u,p_i} \vee \neg R_{A,u,p_j})$ where $i, j \in \{1, ..., |\mathrm{Perm}(A)|\}, i < j$, which ensures at most one permutation. We repeat this for each active node $u \in V_A$.

   2b. For a passive node $u \in V_P$ and each permutation $p \in \mathrm{Perm}(P)$, we require that only one $R_{P,u,p}$ is true, therefore we add clause $(R_{P,u,p_1} \vee R_{P,u,p_2} \vee \dots \vee R_{P,u,p_{|\mathrm{Perm}(P)|}})$ which ensures at least one permutation, and clauses $(\neg R_{P,u,p_i} \vee \neg R_{P,u,p_j})$ where $i, j \in \{1, ..., |\mathrm{Perm}(P)|\}, i < j$, which ensures at most one permutation. We repeat this for each passive node $u \in V_P$.

3. A permutation of label configurations implies that a node labels its incident edges according to the permutation.

   3a. For an active node $u \in V_A$, each permutation $p \in \mathrm{Perm}(A)$, and each label $l_i \in p, 1 \leq i \leq \Delta$, we require that $R_{A,u,p}$ implies $L_{A,u,v_i,l_i}$, where $v_i$ is the $i$-th adjacent passive node of active node $u$. Therefore, we add clause $(\neg R_{A,u,p} \vee L_{A,u,v_i,l_i})$. We repeat this requirement for each active node $u \in V_A$.

   3b. For a passive node $u \in V_P$, each permutation $p \in \mathrm{Perm}(P)$, and each label $l_i \in p, 1 \leq i \leq \delta$, we require that $R_{P,u,p}$ implies $L_{P,u,v_i,l_i}$, where $v_i$ is the $i$-th adjacent active node of passive node $u$. Therefore, we add clause

$(\neg R_{P,u,p} \vee L_{P,u,v_i,l_i})$. We repeat this requirement for each passive node $u \in V_P$.

Based on the above conditions and clauses, we encode a CNF by forming a conjunction of clauses. Formatting the problem as DIMACS CNF can be done with a simple mapping from the variables to unique positive integers, or negative integers in case of a negated variable. It is quite trivial, therefore we do not discuss more about it. After obtaining the DIMACS CNF, we feed it into a SAT solver, as showed in Figure 19. In our implementation, we use the SAT solver called Kissat [23, 41], which is the winner in the main track of the SAT Competition 2020 [43].

## 6.4 Generating LCL Problems

We generate a class of LCL problems for a triple $(\Delta, \delta, \lambda)$, where $\Delta$ and $\delta$ are the sizes of label configurations in $A$ and $P$, respectively, and $\lambda$ is the size of $\Sigma$ i.e. the number of labels.

The set of all possible label configurations is a $k$-combination with repetitions, where $k$ is the size of a label configuration. Let $\mathrm{comb}(k, \Sigma)$ be the function that represents $k$-combination with repetitions. For example, if $\Delta$ is 2, $\Sigma = \{\mathrm{A}, \mathrm{B}\}$ and $\lambda$ is 2, then the set of all possible label configurations is

$$\mathrm{comb}(\Delta, \Sigma) = \{\{\mathrm{A}, \mathrm{A}\}, \{\mathrm{A}, \mathrm{B}\}, \{\mathrm{B}, \mathrm{B}\}\}.$$

The set of all possible sets of label configurations is a power set of the set of all possible label configurations, excluding an empty set. Let us define this power set without an empty sets as a function $\mathrm{pow}(d, S)$, where $d$ is the size of a label configuration, and $S$ is the set of all possible label configurations. For example, the power set of the above example excluding an empty set is

$$\begin{aligned}\mathrm{pow}(\Delta, \mathrm{comb}(\Delta, \Sigma)) = \{&\{\{\mathrm{A}, \mathrm{A}\}\},\\ &\{\{\mathrm{A}, \mathrm{B}\}\},\\ &\{\{\mathrm{B}, \mathrm{B}\}\},\\ &\{\{\mathrm{A}, \mathrm{A}\}, \{\mathrm{A}, \mathrm{B}\}\},\\ &\{\{\mathrm{A}, \mathrm{A}\}, \{\mathrm{B}, \mathrm{B}\}\},\\ &\{\{\mathrm{A}, \mathrm{B}\}, \{\mathrm{B}, \mathrm{B}\}\},\\ &\{\{\mathrm{A}, \mathrm{A}\}, \{\mathrm{A}, \mathrm{B}\}, \{\mathrm{B}, \mathrm{B}\}\}\}.\end{aligned}$$

The set of all LCL problems of class $(\Delta, \delta, \lambda)$ is defined as

$$\mathrm{comb}(\Delta, \Sigma) \times \mathrm{comb}(\delta, \Sigma),$$

where $\Sigma$ is any set of labels of length $\lambda$. The elements are pairs $(A, P)$, where $A$ is the set of all allowed label configurations for active nodes, and $P$ is the set of all allowed label configurations for passive nodes.

Now that we know how to generate a class of LCL problems, we show the number of problems in classes in Table 11. In the table, we also show the numbers of *purged*

and *normalized* problems for each class. These are the optimizations that reduce the size of problems we have to process through in our implementation, and we discuss them next.

| $\Delta$ | $\delta$ | $\lambda$ | Problems | Purged problems | Purged and normalized problems |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 2 | 9 | 3 | 2 |
| 2 | 1 | 2 | 21 | 7 | 5 |
| 2 | 2 | 2 | 49 | 27 | 18 |
| 3 | 2 | 2 | 105 | 67 | 38 |
| 4 | 2 | 2 | 217 | 147 | 84 |
| 4 | 3 | 2 | 465 | 379 | 200 |
| 4 | 4 | 2 | 961 | 843 | 446 |
| 6 | 6 | 2 | 16129 | 15627 | 7926 |
| 2 | 2 | 3 | 3969 | 2103 | 419 |
| 3 | 2 | 3 | 64449 | 44343 | 7735 |
| 3 | 3 | 3 | 1046529 | 962871 | 162299 |

Table 11: The number of LCL problems in different problem classes.

Purging is an operation, where an LCL problem is cut down from label configurations that are redundant, in a sense that they cannot possibly be in a valid configuration, no matter what. These redundant label configurations are found by forming a list of labels that appear in both sets $A$ and $P$. Then we remove each label configuration from both sets, that have some other labels than what was in the list.

For example, if $A = \{\{\mathrm{A},\mathrm{A}\},\{\mathrm{A},\mathrm{B}\},\{\mathrm{B},\mathrm{B}\}\}$ and $P = \{\{\mathrm{A},\mathrm{A}\},\{\mathrm{A},\mathrm{C}\},\{\mathrm{C},\mathrm{C}\}\}$, then the labels in $A$ are $\{\mathrm{A},\mathrm{B}\}$ and labels in $P$ are $\{\mathrm{A},\mathrm{C}\}$. The set of labels that appear in both $A$ and $P$, i.e. $\{\mathrm{A},\mathrm{B}\} \cap \{\mathrm{A},\mathrm{C}\}$ is $\{\mathrm{A}\}$. Thus, we remove all label configurations from both $A$ and $P$ that contain any other label than A. In the end, we get $A = \{\{\mathrm{A},\mathrm{A}\}\}$ and $P = \{\{\mathrm{A},\mathrm{A}\}\}$.

Normalizing an LCL problem is an operation of renaming the labels with a permutation such that the resulting problem is the minimum of all permutated problems. Here, we assume that each label in a label configuration is represented in lexicographic order, and each label configuration is also represented in lexicographic order in both sets $A$ and $P$. We also assume a common set of labels, the upper-case alphabet. The total order of two LCL problems is the lexicographic order. That is, the total order of problems $\Pi_1 = (A_1, P_1, \Sigma_1), \Pi_2 = (A_2, P_2, \Sigma_2)$ from problem class $(\Delta, \delta, \lambda)$ is $\Pi_1 < \Pi_2$ if and only if $A_1 < A_2$ or $(A_1 = A_2$ and $P_1 < P_2)$. The result of normalization is an LCL problem in its *canonical form*. After normalizing any two isomorphic LCL problems, their appearance is identical.

For example, the isomorphic problems

$$\Pi_1 = (\{\{\mathrm{B},\mathrm{B}\}\},\{\{\mathrm{A},\mathrm{C}\},\{\mathrm{C},\mathrm{C}\}\},\Sigma),$$

$$\Pi_2 = (\{\{\mathrm{D},\mathrm{D}\}\},\{\{\mathrm{A},\mathrm{A}\},\{\mathrm{E},\mathrm{A}\}\},\Sigma),$$

after normalization look like

$$\Pi'_1 = (\{\{\mathrm{A},\mathrm{A}\}\}, \{\{\mathrm{B},\mathrm{B}\}, \{\mathrm{B},\mathrm{C}\}\}, \Sigma),$$

$$\Pi'_2 = (\{\{\mathrm{A},\mathrm{A}\}\}, \{\{\mathrm{B},\mathrm{B}\}, \{\mathrm{B},\mathrm{C}\}\}, \Sigma),$$

(denoted by a prime).

In our implementation, we first generate all LCL problems. Then we use the purge operation for each LCL problem, and remove problems that have an empty set of label configurations. Finally, we normalize each problem, and remove duplicates. The amount of problems after purging and normalizing can be seen in Table 11.

## 6.5 Optimizations

In this section, we will discuss some optimizations used in the implementation that we have not yet mentioned in previous sections. The optimizations we discuss here briefly, utilizes either parallelization or caching.

When we generate graphs with *genbg*, we actually do it in parallel. If we give *genbg* the option `r/m`, it generates the `r`-th subset of `m` roughly equal sized subsets. For some fixed `m` we can generate each subset in parallel by giving each option `0/m`, `1/m`, . . . , `m-1/m` to different threads. This makes it roughly $t$-times faster, where $t$ is the number of threads used, assuming that there are $t$ available cores in the system.

When multigraphs are generated, we can optionally save them into a database. Then multigraphs can be fetched from the database when needed, nearly for free in terms of time. Similarly, LCL problems can also be saved into the database, and automatically fetched from there.

In our implementation, finding counterexamples for multiple LCL problems is done in parallel, utilizing all logical cores of the system. This can be done quite trivially, as different instances of finding counterexamples do not really depend on each other. They only depend on the multigraphs to generate SAT problems, which can be accessed safely with read-only access.

# 7 Results

In this section, we will discuss the results from our implementation. We answer to Research question 3 in Section 7.1 by finding new non-constant lower bounds for LCL problems that have already been classified with constant lower bounds prior this work. In Section 7.2, we answer to Research question 4 by finding new lower bounds for classes of LCL problems. We use the implementation [38, commit `c244e8e683`] in its current state as of the time of writing this thesis. When we say `bin`, we refer to the compiled binary of the implementation. Throughout this section, we assume that the results are obtained in a modern computer with AMD Ryzen 9 3900X 12-core 24-thread CPU, and 32 GB of RAM.

We do not use the caching feature of our implementation unless otherwise mentioned, as fetching already generated problem classes and multigraphs from the disk is substantially faster than generating them. We want to give the measurements without pre-generated data unless we target some specific problems, like in Section 7.1.

Earlier in Section 2.8, we used the following notation to present the sets $A$ and $P$ of maximal independent set.

$$A = \{\{\mathrm{I}, \mathrm{I}, \mathrm{I}\}, \{\mathrm{P}, \mathrm{O}, \mathrm{O}\}\},$$
$$P = \{\{\mathrm{I}, \mathrm{P}\}, \{\mathrm{I}, \mathrm{O}\}, \{\mathrm{O}, \mathrm{O}\}\}.$$

In the following sections, all LCL problems are presented in an alternative format, where each label in a label configuration are concatenated, and each label configuration is separated by a space. Maximal independent set would be encoded as $A =$ III POO and $P =$ IP IO OO. The problems are also normalized, as explained in Section 6.4. Therefore, the problem is actually encoded as $A =$ AAA BBC and $P =$ AB AC BB.

## 7.1 Improving Lower Bounds of LCL Problems

As we discussed in Section 3, we use a database [35, 42, 36] that consists of LCL problems and their current known lower and upper bounds. There are two LCL problem classes in the database that are for non-directed non-rooted trees—$(\Delta, \delta, \lambda) = (3, 2, 3)$ and $(\Delta, \delta, \lambda) = (3, 2, 2)$. All 38 problems of the class $(3, 2, 2)$ are already classified with tight deterministic bounds. This is not the case in class $(3, 2, 3)$, therefore we try to find new deterministic lower bounds for the problems using our implementation.

We use the latest database dump [40] of the database. In the class $(3, 2, 3)$, there are 7735 unique LCL-problems after purging and normalizing the problems, as we can see from Table 11. The database also agrees with the number of problems.

We are interested in the problems that have a constant ($\mathcal{O}(1)$) deterministic lower bound and a non-constant upper-bound, as these are the only problems for which we can possibly find a better lower bound of non-constant i.e. $\Omega(\log^* n)$. When we remove problems with non-constant deterministic lower bounds, we get 6538 problems with a constant deterministic lower bound. Out of these problems, we need to remove problems with constant upper bound, as these problems already have a

tight constant bound, i.e. $\Theta(1)$. After the removal, we have only 66 problems left. The problems can be fetched from the database to a file problems.txt using the command bin fetch_problems 3 2 3 <database_address> > problems.txt, assuming that there is a PostgreSQL database in the address <database_address> with the data from the dump.

We execute our implementation using the command

```
bin find <n_low> <n_high> --stats from_stdin < problems.txt
```

where $n_{\text{low}}$ and $n_{\text{high}}$ are substituted with the smallest and highest number of vertices we want for the graphs, respectively. The option --stats gives us the execution times from generating graphs, and the execution times from encoding and solving SAT problems. We run the command as follows in Table 12.

| | | | Time (s) | |
|---|---|---|---|---|
| $n_{\text{low}}$ | $n_{\text{high}}$ | # of new lower bounds | Generate graphs | SAT |
| 1 | 5 | 1 | 0.031 | 0.002 |
| 6 | 10 | 7 | 0.030 | 0.004 |
| 11 | 15 | 8 | 0.062 | 0.015 |
| 16 | 20 | 7 | 0.140 | 0.080 |
| 21 | 25 | 7 | 3.758 | 0.510 |
| 26 | 30 | 7 | 137.5 | 3.980 |

Table 12: Executing the command multiple times with different vertex ranges gives us new lower bounds. We can also see how long it takes to generate multigraphs, and also how long it takes to encode and solve SAT problems.

The executions from Table 12 were all executed with an identical input problem set, therefore some problems can be as a result in multiple executions. When we consider only the smallest graph where a problem was found to be unsolvable, there are only 9 distinct problems left. The problems are listed in Table 13.

| $n$ | $A$ | $P$ | Lower bound | | Upper bound |
|---|---|---|---|---|---|
| | | | Old | New | |
| 5 | AAA AAB BBC BCC | AB CC | $\Omega(1)$ | $\Omega(\log^* n)$ | $\mathcal{O}(\log n)$ |
| 10 | AAA AAB ABC BCC | AC BB | $\Omega(1)$ | $\Omega(\log^* n)$ | $\mathcal{O}(\log n)$ |
| 10 | AAA AAB ABC BCC | AC BB CC | $\Omega(1)$ | $\Omega(\log^* n)$ | $\mathcal{O}(\log n)$ |
| 10 | AAA AAB ABC BCC CCC | AC BB | $\Omega(1)$ | $\Omega(\log^* n)$ | $\mathcal{O}(\log n)$ |
| 10 | AAA AAB ABC CCC | AC BB | $\Omega(1)$ | $\Omega(\log^* n)$ | $\mathcal{O}(\log n)$ |
| 10 | AAA AAB BCC | AC BB CC | $\Omega(1)$ | $\Omega(\log^* n)$ | $\mathcal{O}(\log n)$ |
| 10 | AAA ABC BBC | AB BB CC | $\Omega(1)$ | $\Omega(\log^* n)$ | $\mathcal{O}(\log n)$ |
| 10 | AAA BBC | AB BB CC | $\Omega(1)$ | $\Omega(\log^* n)$ | $\mathcal{O}(\log n)$ |
| 15 | AAA BBC | AB AC BB CC | $\Omega(1)$ | $\Omega(\log^* n)$ | $\mathcal{O}(\log^* n)$ |

Table 13: New lower bounds for LCL problems from the class $(3, 2, 3)$. Here, the column $n$ indicates the size of the smallest multigraph, in which the problem is unsolvable. The last column lists the current known upper bound from the database.

We make an observation that the last problem with a new lower bound in Table 13 has a matching upper bound. Therefore, this problem has now a tight bound of $\Theta(\log^* n)$. Also, we make an observation that it is a relaxation of the maximal independent set i.e. the set $P$ has an additional label configuration CC.

Next, in Table 14, we list all the multigraphs in which we found the problems to be unsolvable.

| $A$ | $P$ | $n$ | | | | | |
|---|---|---|---|---|---|---|---|
| | | 5 | 10 | 15 | 20 | 25 | 30 |
| AAA AAB BBC BCC | AB CC | 1 | | | | | |
| AAA AAB ABC BCC | AC BB | | 1 | $5, 9, 12$ | $X$ | $Y$ | $Z$ |
| AAA AAB ABC BCC | AC BB CC | | 1 | $5, 9, 12$ | $X$ | $Y$ | $Z$ |
| AAA AAB ABC BCC CCC | AC BB | | 1 | $5, 9, 12$ | $X$ | $Y$ | $Z$ |
| AAA AAB ABC CCC | AC BB | | 1 | $5, 9, 12$ | $X$ | $Y$ | $Z$ |
| AAA AAB BCC | AC BB CC | | 1 | $5, 9, 12$ | $X$ | $Y$ | $Z$ |
| AAA ABC BBC | AB BB CC | | 1 | $5, 9, 12$ | $X$ | $Y$ | $Z$ |
| AAA BBC | AB BB CC | | 1 | $5, 9, 12$ | $X$ | $Y$ | $Z$ |
| AAA BBC | AB AC BB CC | | | 9 | | | |

Table 14: List of multigraphs in which the problems are unsolvable. The multigraphs are listed as identifiers under each graph size $n$, and these identifiers are each specific to the size $n$. The identifiers under sizes 20, 25 and 30 are too large to be shown in this table, thus we use sets $X, Y, Z$. The sizes of these sets are $|X| = 12, |Y| = 66$ and $|Z| = 424$.

We visualize the multigraphs of sizes $n \in \{5, 10, 15\}$ from Table 14. These graphs can be seen in Figure 21.

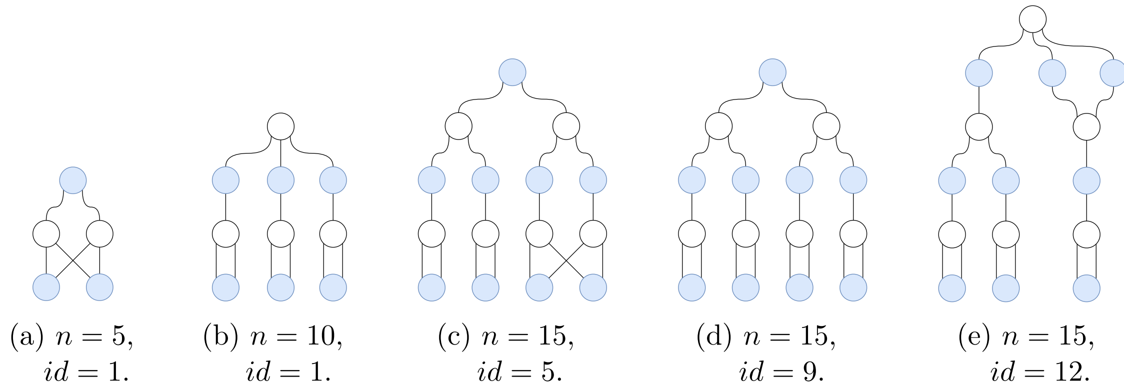



Figure 21: All multigraphs from Table 14, where $n \in \{5, 10, 15\}$.

## 7.2 Finding New Lower Bounds For Classes of LCL Problems

In this section, we try to find new lower bounds for various classes of LCL problems, with the focus on measuring performance. We give the execution times of each execution similarly we did in Table 12, except this time we exclude all problems with new lower bounds before executing the implementation with higher graph sizes. We also use only the exact graph sizes that are possible with each problem class, as explained in Section 6.1. For example, there are only multigraphs of sizes $5, 10, 15, 20, 25, 30, \ldots$, with the problem class $(3, 2, 3)$. In this section, we use a cache for problems, therefore only the first graph size for each example has the true generation time for problems, and all consequent entries in each table show the time it takes to fetch the problems from the cache.

Our first experiment with problem class $(7, 5, 2)$ is a failure, as seen in Table 15. We could not generate even the smallest multigraphs without the process being terminated. The class has 7770 unique problems. At first, it seemed the class would be a good starting point, as previously in Section 7.1, we ran the implementation with the class $(3, 2, 3)$, which has a similar number of problems (7735). It seems that the degrees $\Delta = 7$ and $\delta = 5$ along with graph size $n = 12$ results in too large set of graphs. Our implementation cannot handle such cases, so there is a lot of room for improvements in the part of graph generation.

| | | Time (s) | | |
|---|---|---|---|---|
| $n$ | # of new lower bounds | Generate problems | Generate graphs | SAT |
| 12 | ? | 0.068 | $> 359$ | ? |
| 24 | ? | ? | ? | ? |
| 36 | ? | ? | ? | ? |

Table 15: Failed experiment with the problem class $(7, 5, 2)$. The execution terminated early during generating graphs, probably because lack of memory.

Our second experiment is with problem class $(6, 4, 2)$. We kept the label count

same as in the first experiment, but we lower each degree by one. Thus, there are only 1616 problems in the class. The results can be seen in Table 16. This time we got new lower bounds at least for some problems.

| | | Time (s) | | |
|---|---|---|---|---|
| $n$ | # of new lower bounds | Generate problems | Generate graphs | SAT |
| 5 | 250 | 0.036 | 0.053 | 0.188 |
| 10 | 21 | 0.002 | 0.373 | 47.5 |
| 15 | ? | $< 1$ | $> 453$ | ? |

Table 16: New lower bounds for problems from the class $(6, 4, 2)$. The execution terminates at $n = 15$ during the generation of graphs, probably for the same reason it terminated in the first experiment.

In Research question 4, we ask if we can make the implementation fast enough to classify large classes of problems. The class $(7, 5, 2)$ with size of 7770 was definitely too large for our implementation, because of the high degrees. The class $(3, 2, 3)$ has roughly the same size (7735) and it could be classified using graphs up to size $n = 30$. On the other hand the class $(6, 4, 2)$ had only 1616 problems, but we were able to classify the problems using graphs up to size $n = 10$. It seems that the answer depends a lot on the definition of what is a large class of problems. In this thesis, we consider a problem class with over 1000 problems as large. With this definition, we get an answer of yes, with some classes.

The bottleneck that causes our implementation to terminate early is the graph generation. We make an observation that the large classes that can be classified fast are the classes with low number of degrees, because then there is only a low number of graphs, thus no running out of memory.

# 8 Conclusion

I present a new algorithm that can detect if an LCL problem does not have a solution in finite connected $(\Delta, \delta)$-biregular multigraphs. Then I show that if the problem does not have a solution in the graph family, it is also unsolvable in the PN model.

I also prove that if an LCL problem is not solvable in the PN model, it cannot be solved in constant time in the LOCAL model. Thus, the algorithm can automatically prove that an LCL problem is not solvable in constant time in the LOCAL model i.e. giving it a lower bound of $\Omega(\log^* n)$.

In order to automatically find new lower bounds in practice, I present an implementation of the algorithm. With the implementation, I find new lower bounds for 9 LCL problems and as a consequence, the problem

$$A = \text{AAA BBC},$$
$$P = \text{AB AC BB CC},$$

is now classified with a tight bound of $\Theta(\log^* n)$. The implementation uses the state-of-the-art graph generation and SAT solving software, in addition to parallelization, to be able to classify large LCL problem classes.

## 8.1 Future Work

Concerning the performance of the implementation, there is still room for improvements. The graph generation is a huge bottleneck and generating large number of graphs causes the implementation to terminate. If this can be solved, the implementation could classify problems using higher degree multigraphs. The implementation in its current state does not fully support high performance computing. With a support for high performance computing, we could potentially find even more interesting results than what we have found. We could also analyze more closely the problems and multigraphs in which the problems are unsolvable. This could reveal some general patterns or ideas for more classifiers.

# References


[1] Stephen A. Cook. "The Complexity of Theorem-Proving Procedures". In: *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*. Ed. by Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman. ACM, 1971, pp. 151–158. DOI: 10.1145/800157.805047.

[2] J. Adrian Bondy and Uppaluri S. R. Murty. *Graph Theory with Applications*. Macmillan Education UK, 1976. ISBN: 978-1-349-03521-2. DOI: 10.1007/978-1-349-03521-2.

[3] Jonathan L. Gross and Thomas W. Tucker. "Generating all graph coverings by permutation voltage assignments". In: *Discret. Math.* 18.3 (1977), pp. 273–283. DOI: 10.1016/0012-365X(77)90131-5.

[4] Dana Angluin. "Local and Global Properties in Networks of Processors (Extended Abstract)". In: *Proceedings of the 12th Annual ACM Symposium on Theory of Computing, April 28-30, 1980, Los Angeles, California, USA*. Ed. by Raymond E. Miller et al. ACM, 1980, pp. 82–93. DOI: 10.1145/800141.804655.

[5] Nathan Linial. "Distributive Graph Algorithms-Global Solutions from Local Data". In: *28th Annual Symposium on Foundations of Computer Science, Los Angeles, California, USA, 27-29 October 1987*. IEEE Computer Society, 1987, pp. 331–335. DOI: 10.1109/SFCS.1987.20.

[6] Leslie Lamport and Nancy A. Lynch. "Distributed Computing: Models and Methods". In: *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Ed. by Jan van Leeuwen. Elsevier and MIT Press, 1990, pp. 1157–1199. DOI: 10.1016/b978-0-444-88074-1.50023-8.

[7] Nathan Linial. "Locality in Distributed Graph Algorithms". In: *SIAM J. Comput.* 21.1 (1992), pp. 193–201. DOI: 10.1137/0221015.

[8] Alain J. Mayer, Moni Naor, and Larry J. Stockmeyer. "Local Computations on Static and Dynamic Graphs (Preliminary Version)". In: *Third Israel Symposium on Theory of Computing and Systems, ISTCS 1995, Tel Aviv, Israel, January 4-6, 1995, Proceedings*. IEEE Computer Society, 1995, pp. 268–278. DOI: 10.1109/ISTCS.1995.377023.

[9] Moni Naor and Larry J. Stockmeyer. "What Can be Computed Locally?" In: *SIAM J. Comput.* 24.6 (1995), pp. 1259–1277. DOI: 10.1137/S0097539793254571.

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

[11] Leonid Barenboim and Michael Elkin. "Deterministic distributed vertex coloring in polylogarithmic time". In: *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*. Ed. by Andréa W. Richa and Rachid Guerraoui. ACM, 2010, pp. 410–419. DOI: 10.1145/1835698.1835797.

[12] Sumit Gulwani et al. "Synthesis of loop-free programs". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. Ed. by Mary W. Hall and David A. Padua. ACM, 2011, pp. 62–73. DOI: 10.1145/1993498.1993506.

[13] Matti Järvisalo et al. "Finding Efficient Circuits for Ensemble Computation". In: *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*. Ed. by Alessandro Cimatti and Roberto Sebastiani. Vol. 7317. Lecture Notes in Computer Science. Springer, 2012, pp. 369–382. DOI: 10.1007/978-3-642-31612-8_28.

[14] Brendan D. McKay and Adolfo Piperno. "Practical graph isomorphism, II". In: *J. Symb. Comput.* 60 (2014), pp. 94–112. DOI: 10.1016/j.jsc.2013.09.003.

[15] Isabela Dramnesc and Tudor Jebelean. "Synthesis of list algorithms by mechanical proving". In: *J. Symb. Comput.* 69 (2015), pp. 61–92. DOI: 10.1016/j.jsc.2014.09.030.

[16] Yi-Jun Chang, Tsvi Kopelowitz, and Seth Pettie. "An Exponential Separation between Randomized and Deterministic Complexity in the LOCAL Model". In: *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*. Ed. by Irit Dinur. IEEE Computer Society, 2016, pp. 615–624. DOI: 10.1109/FOCS.2016.72.

[17] Joel Rybicki. "Counting, clocking, and colouring - Fault-tolerant distributed coordination". PhD thesis. Aalto University, Helsinki, Finland, 2016. URL: https://aaltodoc.aalto.fi/handle/123456789/23023.

[18] Mika Göös, Juho Hirvonen, and Jukka Suomela. "Linear-in-$\Delta$ lower bounds in the LOCAL model". In: *Distributed Comput.* 30.5 (2017), pp. 325–338. DOI: 10.1007/s00446-015-0245-8.

[19] Alkida Balliu et al. "Hardness of Minimal Symmetry Breaking in Distributed Computing". In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*. Ed. by Peter Robinson and Faith Ellen. ACM, 2019, pp. 369–378. DOI: 10.1145/3293611.3331605.

[20] Sebastian Brandt. "An Automatic Speedup Theorem for Distributed Problems". In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*. Ed. by Peter Robinson and Faith Ellen. ACM, 2019, pp. 379–388. DOI: 10.1145/3293611.3331611.

[21] Dennis Olivetti. *Round Eliminator: a tool for automatic speedup simulation*. https://github.com/olidennis/round-eliminator. 2019. (Visited on 04/21/2022).

[22] Alkida Balliu et al. "Classification of Distributed Binary Labeling Problems". In: *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*. Ed. by Hagit Attiya. Vol. 179. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 17:1–17:17. DOI: 10.4230/LIPIcs.DISC.2020.17.

[23] Armin Biere et al. "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020". In: *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*. Ed. by Tomas Balyo et al. Vol. B-2020-1. Department of Computer Science Report Series B. University of Helsinki, 2020, pp. 51–53.

[24] Juho Hirvonen and Jukka Suomela. *Distributed Algorithms 2020*. 2020. URL: https://jukkasuomela.fi/da2020/da2020.pdf (visited on 12/14/2021).

[25] Dennis Olivetti. "Brief Announcement: Round eliminator: a tool for automatic speedup simulation". In: *PODC '20: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, August 3-7, 2020*. Ed. by Yuval Emek and Christian Cachin. ACM, 2020, pp. 352–354. DOI: 10.1145/3382734.3405694.

[26] Tanguy Rocher. *Classification of distributed ternary labeling problems*. Bachelor's thesis. 2020. URL: https://github.com/trocher/tlpDoc (visited on 04/21/2022).

[27] Tanguy Rocher. *tplClassifier*. 2020. URL: https://github.com/trocher/tlpClassifier (visited on 04/21/2022).

[28] Jukka Suomela. "Using Round Elimination to Understand Locality". In: *SIGACT News* 51.3 (2020), pp. 63–81. DOI: 10.1145/3427361.3427374.

[29] Aleksandr Tereshchenko. *Cyclepath classifier*. https://github.com/AleksTeresh/cyclepath-classifier. 2020. (Visited on 04/22/2022).

[30] Aleksandr Tereshchenko. *Tree classifications*. https://github.com/AleksTeresh/tree-classifications. 2020. (Visited on 04/21/2022).

[31] Alkida Balliu et al. "Locally Checkable Labelings with Small Messages". In: *CoRR* abs/2105.05574 (2021). arXiv: 2105.05574. URL: https://arxiv.org/abs/2105.05574.

[32] Alkida Balliu et al. "Locally Checkable Problems in Rooted Trees". In: *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*. Ed. by Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen. ACM, 2021, pp. 263–272. DOI: 10.1145/3465084.3467934.

[33] Yi-Jun Chang, Jan Studený, and Jukka Suomela. "Distributed Graph Problems Through an Automata-Theoretic Lens". In: *Structural Information and Communication Complexity - 28th International Colloquium, SIROCCO 2021, Wrocław, Poland, June 28 - July 1, 2021, Proceedings*. Ed. by Tomasz Jurdzinski and Stefan Schmid. Vol. 12810. Lecture Notes in Computer Science. Springer, 2021, pp. 31–49. DOI: 10.1007/978-3-030-79527-6_3.

[34] Jan Studený and Aleksandr Tereshchenko. *rooted-tree-classifier*. https://github.com/jendas1/rooted-tree-classifier. 2021. (Visited on 04/21/2022).

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

[36] Aleksandr Tereshchenko. *LCL classifier*. https://github.com/AleksTeresh/lcl-classifier. 2021. (Visited on 05/24/2022).

[37] Alkida Balliu et al. "Efficient Classification of Local Problems in Regular Trees". In: *CoRR* abs/2202.08544 (2022). arXiv: 2202.08544. URL: https://arxiv.org/abs/2202.08544.

[38] Nikos Heikkilä. *Nonconstant LCL classifier*. https://github.com/Niketin/nonconstant-lcl-classifier. 2022. (Visited on 05/20/2022).

[39] Jan Studený and Jukka Suomela. *poly-classifier*. https://github.com/jendas1/poly-classifier. 2022. (Visited on 04/21/2022).

[40] *A database dump of LCL classifier*. https://zenodo.org/record/4671485/files/2021-04-08.sql?download=1. (Visited on 05/24/2022).

[41] Armin Biere and Mathias Fleury. *Kissat SAT Solver*. http://fmv.jku.at/kissat/. (Visited on 05/19/2022).

[42] *LCL classifier*. https://lcl-classifier.cs.aalto.fi/. (Visited on 05/24/2022).

[43] *SAT Competition 2020*. https://satcompetition.github.io/2020/. (Visited on 05/19/2022).

[44] *Satisfiability Suggested Format*. http://www.domagoj-babic.com/uploads/ResearchProjects/Spear/dimacs-cnf.pdf. (Visited on 05/18/2022).

[45] Rust Team. *Rust programming language*. https://www.rust-lang.org/. (Visited on 05/16/2022).

[46] *The International SAT Competition Web Page*. http://satcompetition.org/. (Visited on 05/18/2022).