Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Aleksandr Tereshchenko

Automated classification of distributed graph problems

Master's Thesis
Espoo, April 21, 2021

Supervisor: Professor Jukka Suomela
Advisor: Professor Jukka Suomela

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author: Aleksandr Tereshchenko

Title:
Automated classification of distributed graph problems

Date:	April 21, 2021	**Pages:**	76
Major:	Computer Science	**Code:**	SCI3042
Supervisor:	Professor Jukka Suomela		
Advisor:	Professor Jukka Suomela		

The field of distributed computing and distributed algorithms is a well-established and quickly developing area of theoretical computer science. Similar to the study of traditional centralized algorithms, many researchers are particularly interested in understanding the complexity of distributed problems. In particular, over the last decade, the research community has had a series of breakthroughs in understanding the complexity landscape of locally checkable labeling problems (LCLs), which is one of the most significant and well-studied problem families in distributed computing.

As more and more individual LCL problems have been understood, people started to investigate whether it was possible to automate the process. This led to a discovery of a whole range of so-called meta-algorithms. These are centralized algorithms that take a distributed LCL problem as their input and return information about its complexity as an output. Furthermore, many of the meta-algorithms turned out to be practical and were subsequently implemented as computer programs that can tell something useful about the complexities of LCL problems.

The problem, however, is that these meta-algorithms are not currently used by the research community. This causes the scholars to solve the same problems that have been solved before. The reason for the failure to adopt the meta-algorithms is the fact that their implementations use different formalisms, which makes it inconvenient to use them in everyday work. The goal of the thesis is to resolve this problem and develop a solution that would unify the numerous meta-algorithms making it possible to benefit from them all while using only a single tool.

In this work, I have developed a unified system that encapsulates most of the existing meta-algorithms, providing a unified interface to its users. I also implemented a Web interface to make the tool even more accessible for the community. Furthermore, I critically analyze the solution and propose ideas for its further improvement.

Keywords: distributed algorithms, distributed computing, locally checkable labeling problems, round elimination, graph algorithms

Language: English

Acknowledgements

I wish to thank my thesis advisor and supervisor Jukka Suomela for the invaluable feedback and advice he provided during the process of writing the thesis. Besides, I would like to thank Sebastian Brandt, Juho Hirvonen, Darya Melnyk, Dennis Olivetti, and Jan Studený for offering their ideas for improvement, constructive feedback, and words of advice and encouragement. I also wish to acknowledge CSC – IT Center for Science, Finland, for computational resources.

Espoo, April 21, 2021

Aleksandr Tereshchenko

Contents

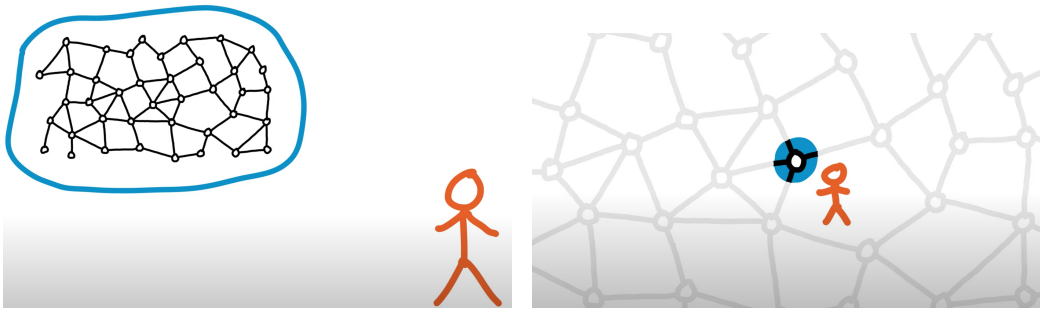
Chapter 1

Introduction

Distributed computing is used in a wide variety of areas such as telecommunication, data processing, process control in real-time systems, etc [39]. Similar to how traditional algorithms are studied with a classical centralized computational entity in mind, distributed algorithms are developed to be used in distributed systems that are becoming more and more ubiquitous in the field of technology [2]. Such distributed systems consist of many computational units each performing local computation and communicating with other such units close to it. Figure 1.1 demonstrates the difference between the centralized and distributed perspectives.

Unlike traditional centralized algorithms, in distributed computing, there is no centralized entity that would see the entirety of the system's input or output. Instead, input is shared among the computational units, each knowing only a part of it. Also, once a distributed algorithm halts, each unit is only aware of its own output piece. This, among other things, implies that it is now non-trivial to ensure that the executed algorithm has succeeded and the outputs of the computational units are indeed valid when considered together.

The field of distributed algorithms has several decades of research history and has grown significantly since its inception in the 1980s. One specific area of the field that has received a lot of attention in recent years from the scientific community is the study of *locally verifiable* problems. These are the kinds of problems, where, roughly speaking, once the execution of an algorithm has ended, each computational unit can collect information only from close-by computational units, and it will be sufficient to ensure that the executed algorithm has succeeded. Problems of this kind are particularly interesting because they have obvious relevance in cases where the size of the network is extremely large, and thus each computational unit cannot afford to check every other unit in the system.



(a) Perspective in classical computing (b) Perspective in distributed computing

Figure 1.1: The difference in perspectives in classical and distributed computing[1]. In the classical setting, a single all-seeing entity is given the whole graph, it solves a problem on the whole graph, and outputs the entire solution. In the distributed setting, there is no all-seeing entity. Instead, each node in the graph is performing some local computation, based on which each node produces only a part of the solution. The outputs of all the nodes are then combined to obtain the entire solution.

As the field has been studied, more and more knowledge about such problems has been accumulated by the research community. In particular, as is common in the field of algorithm design, much of the research has been focused on the complexity of problems in distributed computing. Apart from individual complexity results, the research community has invented several *meta-algorithms*. These are centralized algorithms that automatically determine the computational complexity of a provided problem.

1.1 Problem statement

The individual complexity results and the meta-algorithms that determine the complexity of a specified problem are scattered across tens of published papers. Each of the papers uses a different problem representation and formalism. This causes researchers to solve the same problems again and again since no centralized place exists that would accumulate all of the above-mentioned results.

I will attempt to rectify the problem by designing and implementing a system that would allow researchers to quickly access complexity-related results for locally verifiable problems, provided such problems have been already

[1]The image was copied from the video [33] that was a part of Distributed Algorithms 2020 course at Aalto University with the permission of the authors.

studied previously. I expect that such a solution would be of high value to the research community and would save significant time resources, allowing the researchers to concentrate on discovering new knowledge about the field instead of spending time on solving problems that have already been solved numerous times before.

1.2 Structure of the thesis

Chapter 2 provides a theoretical background necessary for understanding the contents of the following chapters. The chapter opens with a brief but informative overview of graph-theoretic concepts. Then, the chapter provides basic theoretical background related to distributed computing. Moreover, the chapter explains notions of locally checkable labeling problems, decidability, as well as covers recent developments in the area of automated classification in distributed algorithms.

Chapter 3 describes meta-algorithms that are used in my final solution. The meta-algorithms take a description of a locally verifiable problem as an input and as an output produce some information about the problem's complexity.

Chapter 4 states my research goals and defines the scope of the thesis project. Chapter 5 provides an overview of an implementation of my solution. Chapter 6 evaluates the outcomes of the implementation against the earlier stated research goals. It also points out possible directions for future development. Finally, Chapter 7 summarizes the entirety of the work.

Chapter 2

Theoretical background

In this chapter, I introduce the necessary theoretical background. Unless specified otherwise, the material presented in this chapter is based on the textbook on distributed algorithms by Hirvonen and Suomela [32].

2.1 Graphs

This section will outline the necessary theoretical background on graphs. Most of the notation related to the graph theoretic concepts in the current section and the rest of the thesis will follow that of the recent introductory textbook on distributed algorithms [32] unless specified otherwise.

A *graph* is a pair $G = (V, E)$, where V denotes the set of all *vertices* and E denotes a set of all *edges*. Each edge in E is represented as a set of 2 nodes i.e. the nodes the given edge is connecting e.g. $e = \{v, u\}$ such that $v \in V$ and $u \in V$.

When talking about the *size of a graph*, or *cardinality* of the graph, I refer to the number of nodes in the graph. Or in other words, the size of a graph G is always $|V|$.

Among other things, edges can be categorised into *directed* and *undirected*. The latter ones simply connect a pair of nodes in a graph, while the latter ones also contain extra information about which of the two connected nodes is a "source" and which one is a "destination". Informally, such a directed edge can be visualised as an arrow that starts in a "source" node and ends in a "destination" node. Similarly, we talk about *undirected graphs* – that is, graphs where all edges are undirected, and *directed graphs* – such graphs where all edges are directed. Formally, while an undirected edge e is defined as a set of two nodes $e = \{v, u\}$, a directed edge e is defined as a pair of two nodes $e = (v, u)$. Recall that in the case of a pair, the order of

its elements does matter, hence, the pair allows us to encode the direction of the edge. Unless mentioned otherwise, we assume that an edge is directed from the first element of the pair to the second. Also note that in this thesis, unless specified explicitly, all graphs are assumed to be undirected.

If two nodes are connected by an edge, we call such nodes *neighbors*. We also say that such nodes are *adjacent* to each other. Moreover, in an undirected graph if there are two edges e_1 and e_2 such that $e_1 \neq e_2$ and $e_1 \cap e_2 \neq \emptyset$ such edges are said to be *adjacent* to each other. Similarly, we can talk about adjacent edges in a directed graph. In that case, two edges e_1 and e_2 are adjacent if $e_1 \neq e_2$ and either first or second element of the pair e_1 contains the same element as either first or second element of the pair e_2.

Besides, we often talk about edges and nodes being *incident* to each other. In an undirected graph, an edge e is incident to a node v if and only if $v \in e$. In this case, we also say that the node v is incident to the edge e. Similarly, in a directed graph, an edge e is incident to a node v if and only if v is the first or the second element of the pair e. Again, in the case described in the previous sentence, the node v is said to be incident to the edge e.

We also often talk about *ports*. A port can be thought of as an end of an edge e associated with a node v so that e is an incident edge of v. Therefore, each edge $e = \{v, u\}$ has two ports associated with it. One port belongs to node v, and another port belongs to node u. From this, it follows that each node has as many ports as it has incident edges.

A *simple graph* is an undirected graph where no two nodes are connected by more than one edge and no edge starts and ends at the same node. In other words, there are no multiple edges or self-loops in a simple graph.

A *degree* of a node $v \in V$ in a graph $G = (V, E)$ is the number of all edges incident to v. That is, it is the number of edges such that for an edge e, $v \in e$ if e is an undirected edge, or v is the first or second element of pair e if e is a directed edge. Moreover, for directed graphs, we often talk about *indegrees* and *outdegrees*. Indegree of a node v is the number of edges incident to v directed towards the node, while outdegree of a node v is the number of edges incident to v directed outwards from the node. Finally, we are often interested in a maximum degree of a graph G, that is, the maximum value of degrees of all nodes belonging to the graph G. I denote such maximum degree as Δ. In this work, we only consider graphs whose nodes' degrees are bounded by a constant. That is, the degree of each node is necessarily finite and does not depend on the size of the graph.

A *walk* in an undirected graph $G = (V, E)$ is a sequence w of a form $w = (v_0, e_1, v_1, e_2, ..., e_l, v_l)$, where $v_i \in V$ and $e_i \in E$, and $e_i = v_{i-1}, v_i$ for all i. A walk from some node v to some other node u is then a walk w such that its first element i.e. $v_0 = v$ and its last element i.e. $v_l = u$. Having defined

a walk, we can talk about a *connected graph*. A connected graph, when talking about undirected graphs, is a graph $G = (V, E)$ in which for any pair of nodes v and u such that $v \neq u$ there is a walk from v to u. In this work, all the graphs should be assumed to be connected unless specified otherwise.

An *isomorphism* between two graphs G_1 and G_2 is a function f such that f is a bijection, and f maps a vertex of the graph G_1 to a vertex of the graph G_2, and an edge between some nodes v and u exists in the graph G_1 if and only if an edge between nodes $f(v)$ and $f(u)$ exists in the graph G_2. From this, it is rather easy to see that if an isomorphism exists from G_1 to G_2, then also an isomorphism exists from G_2 to G_1. If there exists an isomorphism between some two graphs, we say that such graphs are isomorphic (to each other).

A *radius-x neighborhood* of a node v in a graph $G = (V, E)$ is a set of all such nodes $u \in V$ that there exists a walk between v and u and the shortest walk from v to u is at most x. Note that it is possible that $v = u$, in which case the shortest walk between v and u is 0.

A diameter of a graph (denoted as $\text{diam}(G)$) is the length of the longest walk in a list W, where W is a list of the shortest walks, i.e. the one between each pair of nodes v and u in a graph $G = (V, E)$ such that $v \in V$ and $u \in V$. If a graph G is not connected and, therefore, there is a pair of nodes v and u that does not have a walk between them, we say that a diameter of such a graph G is infinity. That is, $\text{diam}(G) = \infty$.

2.1.1 Several important graph families

This subsection will introduce some of the graph families that will be necessary for understanding the content of the thesis. First, I will define paths and cycles. Then, I will briefly introduce trees and explain the difference between rooted and unrooted trees.

A *path* graph is a sequence of nodes that are joined together by edges and have the following properties:

1. The whole graph is connected. That is, there is a walk from any node of the path to any other node of the path.
2. The graph consists of at least two nodes.
3. Exactly two nodes have a degree 1, and all the other nodes have a degree 2.

A *cycle* is a connected graph where each node has a degree 2. If a graph G is a cycle or by removing several nodes and/or edges from the graph G we

can obtain a cycle graph, then we say that G contains a cycle. On the other hand, if G is not a cycle, and it is not possible to obtain a cycle from G just by removing some nodes and edges from it, we say that such a graph does not contain a cycle, or that it is acyclic.

A *tree* is an undirected acyclic connected graph. A *rooted tree* has a single *root vertex*, and thus some nodes have a *parent* node (some because e.g. a root node never has a parent) and one or more *children* nodes. For a node v, a node u is its parent if and only if v and u are neighbor nodes, and the shortest walk from the root to u is shorter by exactly 1 compared to the shortest walk from the root to v. Node v is a child of node u if and only if u is a parent of node v. Finally, in the context of rooted trees, a node l that has no children is referred to as a *leaf* node. On the other hand, an *unrooted tree* has no single root, and therefore the notion of parent or child is not defined. Instead, we talk about neighbors of some node v. However, we still use the notion of leaves to denote nodes of degree 1.

2.1.2 Biregular trees

A *bipartite* graph $G = (V, E)$ has its vertices partitioned into two subset $U \subseteq V$ and $W \subseteq V$ so that the partitioning forms a proper 2-coloring of the node V. In other words, in a bipartite graph $G = (V, E)$, if a node $v \in U$ then all its neighbors belong to W, and if a node $v \in W$ then all its neighbors belong to U.

A *regular* graph is a graph where each vertex has the same degree. A *biregular* graph is a graph where each vertex has one of the two allowed degrees and only them. In this thesis, when we talk about *(β, δ)-biregular* graph G, we always assume that $G = (V, E)$ is bipartite so that V is partitioned into two subsets U and W, nodes of these subsets together form a proper 2-coloring, all nodes in U are of degree β, and all nodes in W are of degree δ. In particular, when I talk about *(β, δ)-biregular trees*, I mean a bipartite biregular tree where all nodes except for the leaves follow the rules of (β, δ)-biregular graphs described above, but leaves themselves are allowed to have degree 1 (otherwise they would not be called leaves).

It is important to notice the equivalence of (δ, 2)-biregular trees and δ-regular trees. Indeed, given a δ-regular tree, we can "turn" each edge $e = \{v, u\}$ into a node v_e such that the node is connected to node v and u and only those nodes. On the other hand, given a (δ, 2)-biregular tree, we can remove every node v of degree 2 and, given that nodes u and w used to be neighbors of v before its removal from the graph, connect nodes v and w with an edge.

2.2 Distributed computing

Now that I have introduced some of the key graph theoretic concepts, I will next outline some of the foundations of distributed computing. We will start with a short historical note about the field of distributed computing. Then, I will explain some of the aspects of the model of computation we are concerned with. Finally, I will give some formal definitions of the model of computation that will be our major concern throughout the rest of the thesis.

2.2.1 General background

The field of distributed computing studies computation in distributed systems, which have become ubiquitous in the modern world, being especially prevalent in the area of technology [2]. A distributed system consists of a number of relatively independent computing modules, which usually need to cooperate in order to fulfill a computational task the system has been given. Usually, each computing module in such a system only has a part of the whole input and is required to produce only a part of the whole output. This is in contrast with a centralized system, in which there exists an all-knowing entity taking all of the input, performing all of the computation, and producing the whole of a result as its output. Due to its modularised and parallel nature, distributed computing has many applications in communication, computation, the Internet, but also in biology and sociology [64].

The field of distributed computing appeared already in the late 1980s, with several prominent papers exploring potentialities of computation performed by multiple interconnected processing units [20, 37, 45]. The first major step in the field happened in 1987 when Linial formalized some of the principles of one variant of a distributed system. This model of computation is currently known as Linial's or the *LOCAL model* [37].

2.2.2 Model of computation

Here, I will describe the model of computation that we are going to assume throughout the rest of the thesis. Note that there are several different models, which are also widely studied in the research community.

First, as it was already stated, unlike in the case of centralized computation, we are concerned with multiple interconnected computing entities that together form a graph. Each such entity is referred to as a vertex or a node in a graph. Connections between the vertices are referred to as edges. The entities only can transfer information along the edges and in no other way between each other. Unless specified otherwise, the graphs are assumed to be

simple graphs, and all edges are assumed to be undirected. Moreover, communication along the edges can simultaneously happen in both directions.

Apart from sending information to each other, nodes can also do local computation based on the local information each node possesses. One thing to note that differs significantly from some of the centralized models of computation is that each vertex in a graph has arbitrarily huge, but finite, storage and computation resources. Informally, this means that anything that can be computed in a centralized setting in a finite amount of time can also be computed locally by a node in an arbitrarily small amount of time. It is also worth noticing that in the model of computation described here, every node of a graph executes the same algorithm. Nevertheless, the algorithm might lead to different commands being executed by different nodes if, for example, nodes' unique identifiers or initial local inputs are different. Besides, the behavior might also differ if two nodes have somewhat different neighborhoods around themselves and therefore will receive potentially different information during their communication rounds. Finally, at the beginning of execution, each node knows only its own input (including its own unique identifier) and its own degree i.e. the number of its neighbors.

Furthermore, computation in a graph happens in synchronous rounds. That implies, for example, that round $x + 1$ is not started by any of the nodes before all of the nodes have completed round x. Each round is divided into three stages:

1. sending some information to some (or all) of its neighbors,
2. receiving information from some (or all) of its neighbors,
3. performing some local computation based on the information that has been stored by a node locally before and the information received during the current round from its neighbors,
4. updating its local state i.e. replacing and adding data in its own local storage.

Each of these stages is also executed completely synchronously by all nodes in a graph, meaning that e.g. no node starts processing any of its data, before all other nodes have received data from their neighbors. The computation of a graph is said to be finished when all of the nodes have outputted their final states and halted. More formally, this means that there are a set of node states that are considered as *halting states*. Whenever a node switches its internal state to one of the halting states, it does not perform any of the actions starting from that point in time, or – to be even more formal

– it does not send any messages to its neighbors, ignores all of the messages sent to it, and does not update its internal state in all of the subsequent rounds. In addition to the requirement that the local computation of each node in each round has to be finite and needs to stop after a finite amount of time passed, all nodes are required to eventually transition into one of the halting states. That is, a valid distributed algorithm – in our model of computing – cannot continue for an infinite number of rounds. Finally, the complexity of a distributed algorithm is measured as the number of such synchronous rounds of computation before the algorithm ends i.e. before all nodes transition into one of the halting states. Notice that different from a centralized model of computation, algorithms complexity (or in other words its running time) is not affected by the amount of local computation on a single node.

Another important thing that has to be mentioned when describing the mode of computation is that all of the operations within each node as well as inter-vertex communication are error-free. In other words, all nodes can be assumed to always act in a fault-free manner, all sent messages can be assumed to never be lost or undelivered. Therefore, we can always assume that messages sent and received by nodes are all in accordance with the actual algorithm that is being executed by the nodes and not a result of an accidental fault or an intentional adversary effort.

2.2.3 Distinctive qualities of the LOCAL model

As was already mentioned previously, our model of computation is known as Linial's or the LOCAL model [37]. Therefore, to complete the description, I will describe in more detail two distinctive qualities of the LOCAL model from other models of distributed computing: namely unique identifiers and arbitrarily large bandwidth.

Each node in the LOCAL model is provided with a unique identifier as part of the initial input. The identifiers are usually assumed to be integer numbers between 1 and $|V|^c$, where $|V|$ is the number of nodes in the graph and c is some constant. Unless specified, we assume that such a constant c is not known by the nodes at the beginning of algorithm execution. Thus, unique identifiers are guaranteed to be positive integer numbers bounded by a polynomial in the number of nodes, but it is not known – by the nodes at the start of algorithm execution – what is the largest unique identifier in the graph. Moreover, the identifiers of the node do not necessarily form a continuous range of integers. That is, if an integer x is used as a unique identifier for some node v, it is possible that an integer $x + 1$ is not used as an identifier in the graph. This implies that at the beginning, nodes do not

know what integers have been used for unique identifiers and what have not been, with the exception of only one integer – their own identifier.

Another characteristic of the LOCAL model, which also was already mentioned before, is the fact that nodes can send (and receive) an arbitrarily large (but finite) amount of bytes of information over a single edge during one round. This fact, combined with the existence of unique identifiers, renders the model as a rather strong one. In particular, this implies that a node can send all of its information – no matter how large it is – to all its neighbors in just one round.

This, in turn, implies a rather curious property. Imagine that every node sends all of its information during the first round and, having received some data from its neighbors, saves all this information locally. Consider some node v in the middle of a graph $G = (V, E)$. Since all nodes send all their data, after the first round, node v will have all the data that all its neighbors had initially, plus its own initial information. Notice that each of v's neighbors now has all the information of their neighbors. Thus, if we combine all the data in the possession of v and v's neighbors after the first round, it is easy to observe that they together have all the information of v's radius-2 neighborhood. But this means that after the second round, when all v's neighbors have sent all their information to v, the node v alone possesses all the data that its radius-2 neighborhood had at the beginning of the algorithm execution. Similarly, after the 3rd round, v will have all the radius-3 neighborhood's data, and so on. In general, after round x, node v will have all information that was initially available in its radius-x neighborhood. Therefore, when considering the LOCAL model, time and space are in a certain sense equivalent. In other words, the number of rounds needed to solve a certain problem is always equal to the distance (or radius) to which a node needs to see to solve a certain problem. One technicality to note here is that because all nodes have unique identifiers, and these identifiers are sent together with the rest of the data, receiving nodes can differentiate what nodes the received data belongs to, and consequently, reconstruct the structure of the neighborhood in the graph.

As a consequence of the above, we can observe that after $\mathrm{diam}(G)$ rounds, node v will have all the information there is in the graph G. This implies that after $\mathrm{diam}(G)$ rounds, in the LOCAL model, we can solve anything that can be solved in a centralized setting. That is, because at the end of $\mathrm{diam}(G)$'th round, each node in the graph has collected all the information there is in the graph and thus can just run the computation locally. And because local computation in the LOCAL model is virtually free, anything that could be computed in a centralized setting will be computed locally by each node individually. Then, in the next round, each node can output its

part of the solution.

2.2.4 Formalizing the LOCAL model

In this subsection, I will formalize the LOCAL model that I informally described above. This will, first of all, help to clarify the aspects of the model that are still left ambiguous after reading the previous subsections. Besides, it will allow us to introduce a randomized model below once the basic deterministic one is unambiguously defined.

An algorithm being executed by each node consists of three functions. One for local state initialization, which is executed only once, after a node v has received its initial local inputs $u(v)$ but before the first round:

$$\text{init}_\text{A}(u(v), d)$$

, where A is a given distributed algorithm and d is the degree of the node v. The function returns an initial local state of the node v.

The second function takes as an input an internal state $x(v)$ of a node v and returns a tuple of size d, where d is a degree of node v. The tuple contains messages that are to be sent to d neighbors of the node v during the current round.

$$\text{send}_\text{A}(x(v), d)$$

The third function takes as an input a local internal state $x(v)$ of a node v, and a tuple $m(v)$ of size d that contains messages received from d neighbors of the node v.

$$\text{receive}_\text{A}(x(v), m(v), d)$$

The function returns a new local internal state of the node v, which becomes a starting internal state $x(v)$ in the next round.

2.2.5 Randomized algorithms

This subsection will introduce the randomized distributed model of computation and highlight some of the differences between randomized and deterministic models. In this section, the focus will be specifically on the randomized LOCAL model as opposed to some other distributed algorithms models.

In the randomized LOCAL model, there are two major differences from the deterministic LOCAL model. First, the function $\text{init}_\text{A}(u(v), d)$ becomes randomised. This means that the initial state $x_0(v) = \text{init}_\text{A}(u(v), d)$ of a node v is chosen from a discrete probability distribution of all possible initial states. Second, the function $\text{receive}_\text{A}(x(v), m(v), d)$ becomes randomised.

This means that, after all messages have been sent and received for round i, a new local internal state $x_i(v) = \text{receive}_\text{A}(x(v), m(v), d)$ for a node v is selected from a discrete probability distribution. Everything else in the model remains exactly as it is in the deterministic case, that is, no changes are made to the $\text{send}_\text{A}(x(v), d)$ function.

Since probabilities are involved when in the randomized setting, it might not be obvious what it means to "solve" a problem. The major problem is that since states of the nodes are chosen from a discrete probability distribution, there is always a probability that the output of the nodes will not be valid. Therefore, there is often a probability that the problem has not been solved after a certain number of rounds.

But before I define what it means to solve a problem in the randomized setting, it is worth noting that there exist two kinds of randomized algorithms: "Monte Carlo" and "Las Vegas". *Monte Carlo* algorithms always stop after a specified $f(n)$ number of rounds, where n is the number of nodes in the graph, but it is not guaranteed that the output will be a valid one. Monte Carlo algorithms only guarantee their execution time and succeed only with probability p. *Las Vegas* algorithms, on the other hand, always produce correct output, once all the nodes in a graph have stopped, but the algorithm will stop after a specified $f(n)$ number of rounds only with some probability p. Thus, in some way, these types of randomized algorithms are complements of each other: one guarantees the correctness of the output but has a chance of going over some specified execution time, while another has a guarantee on the execution time but has some probability of producing incorrect output. In the rest of the thesis, unless explicitly specified otherwise, we can assume that the randomized algorithms are Monte Carlo algorithms.

Finally, I now can define what it means to solve a problem in a randomized context. When saying that a certain randomized algorithm "solves" a certain problem in $T(n)$ rounds, unless specified otherwise, I mean that the algorithm stops after $T(n)$ computational rounds, and the nodes output (or to be more precise the nodes have transitioned to one of the output states) a correct solution "with high probability" (often denoted as "w.h.p."). Formally, if an algorithm succeeds with high probability, it means that the algorithm succeeds with the probability of at least $1 - 1/n^c$, where n is a number of nodes in a graph and c is some constant such that $c > 0$. Moreover, c is a constant that can be freely chosen when running the algorithm. To give a simple example, an algorithm's running time may linearly depend on such a constant c, then before an algorithm is executed, one may freely choose the constant so that the larger the constant the longer is the running time, but also the lower the chance that the produced output will be incorrect. Notice that c does not need to affect running time, but it often does since the larger

it is the lower is the probability of the algorithm's failure, and decreasing such probability usually comes at the cost of something else – often running time. Thus, when saying that something succeeds "with high probability" I do not mean it vaguely but rather refer to a very precise mathematical definition given above.

2.3 Iterated logarithm

This section will introduce the iterated logarithm function. The function is one of the most common complexity results of distributed algorithms and will appear often in the rest of the document.

The iterated logarithm function, denoted as $\log^*(x)$ is defined as follows:

$$\log^*(x) = \begin{cases} 0 & \text{if } x \leq 1, \\ 1 + \log^*(\log_2 x) & \text{otherwise.} \end{cases}$$

The function is notable for the fact that it grows extremely slow. For example,

$$\begin{aligned} &\log^*(1) = 0, \\ &\log^*(2) = 1, \\ &\log^*(4) = 2, \\ &\log^*(16) = 3, \\ &\log^*(65536) = 4, \\ &\log^*(2^{65536}) = 5. \end{aligned}$$

2.4 Locally checkable labeling problems

This section will introduce locally checkable labeling problems (from now on referred to as LCLs), which are the main focus of this thesis. The section will start with a short historical note and then provide a formal definition of LCL problems.

As the field of distributed algorithms started to develop, it became clear that some classes of problems are of particular interest to the theoretical research community. One class of such problems has been first introduced in 1993 by Moni Naor and Larry Stockmeyer under the name of locally checkable labeling (LCL) problems [46].

In short, an LCL problem is a distributed problem that satisfies the following criteria [46, 59]:

- The maximum degree of a graph is a finite number that does not depend on n.
- There is a finite number of input labels, and the number is not dependent on n.
- There is a finite number of output labels, and the number does not depend on n.
- The correctness of a solution can be checked locally by each individual node. That is, after a solution is produced and all the nodes in the graph have stopped, each node can just check a radius-$O(1)$ neighborhood around itself. The solution is valid if and only if each of the individual neighborhoods for all the nodes is valid.

In the list above, n is the number of nodes in the graph. Finally, as is easy to see from the definition of LCL problems above, there is always a finite representation of any LCL problem. Indeed, one can simply list all valid local neighborhoods, list all valid input labels, list all valid output labels, and specify the maximum degree of a graph. Since all of the beforementioned are of size $O(1)$, it is always possible to have a finite representation of an LCL. In practice, however, in many cases, it is more practical to come up with a more concise representation. We will return to the topic of representing LCL problems later in this thesis.

The study of LCL problems has been one of the major research directions during the last 6–7 years, with numerous papers published in major distributed computing conferences [6–8, 13, 17, 18, 24, 52]. Currently, the study of LCL problems has reached the stage of maturity with, for example, the complexity landscape of LCL problems being almost entirely understood [16, 59].

2.5 Major LCL problems

This section will introduce some of the common LCL problems widely studied in the distributed algorithms research community. Note that all of the problems also exist in the context of centralized computing.

Vertex coloring is perhaps one of the most well-known problems on graphs. Informally, the goal is to color a graph in such a way that no two adjacent nodes are colored with the same color. It is also easy to see that the problem is an LCL problem because it is sufficient and necessary for each node to check its radius-1 neighborhood. If each node's radius-1 neighborhood is a

valid one i.e. none of the neighbors of node v have the same color as v itself, then and only then the output is a valid vertex coloring. An example of a solved instance of the vertex coloring problem is presented in Figure 2.1a.

Another well-known and widely studied problem on graphs is **edge coloring**. Informally, each edge needs to be colored in such a way that no two adjacent edges are of the same color. Coloring edges in a graph can still be simulated with the setting in which only nodes output the labels. Each node will output a tuple of the size equal to the node's degree. Each element of a tuple represents a label that the node outputs on a corresponding incident edge. The problem is also an LCL problem because after a final output is produced, each node v will need to check its radius-1 neighborhood to make sure two things are in order:

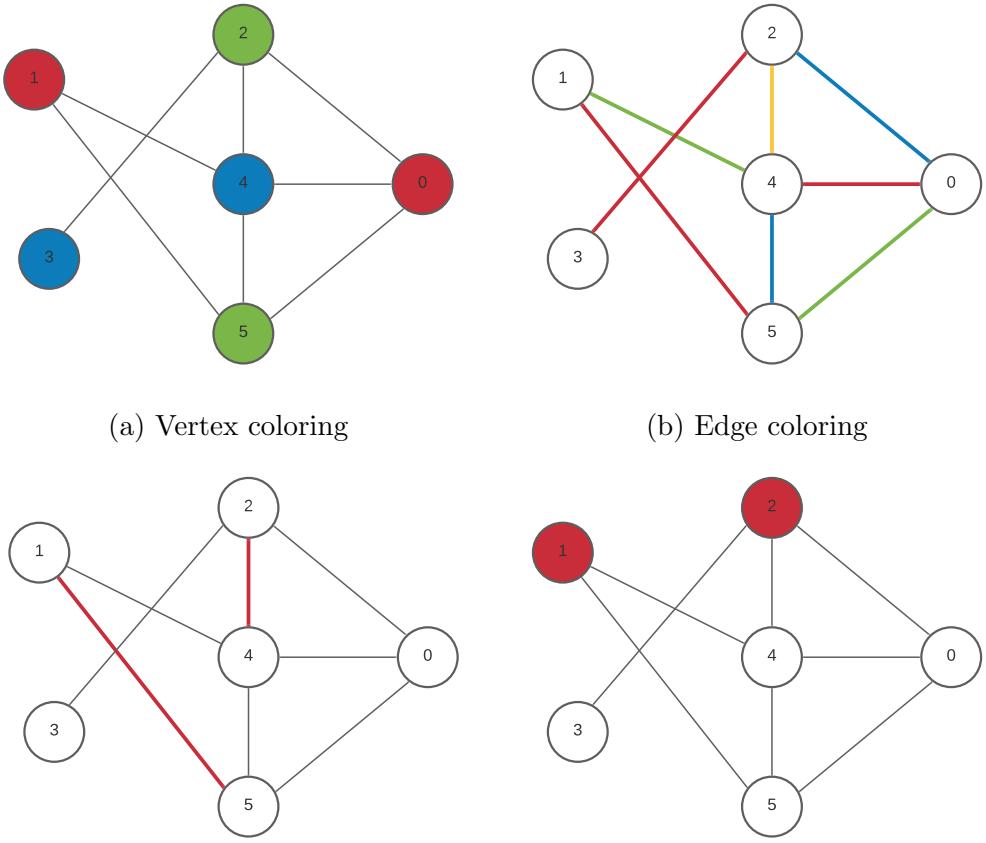
1. All edges incident to v have to be of a different color. That is, no two elements of node v's output tuple can be the same.

2. For each edge incident to v, a color that node v colored some incident edge e must be the same as the color that node u colored the edge e with. Here $e = \{v, u\}$. In other words, because each edge's coloring depends on the output of two nodes, such coloring has to be consistent for each edge i.e. the produced colors need to be the same on both ends of each edge.

An example of a solved instance of the edge coloring problem is presented in Figure 2.1b.

An **independent set** is a set I of vertices in a graph $G = (V, E)$ such that $I \subseteq V$. The only condition, however, is that no pair of vertices in the set I can be adjacent to each other. A **maximal independent set problem** is then a problem in which in addition to finding such independent set I, there is an additional maximality restriction. The maximality restriction forbids cases where a node v is not in the set I but also does not have any adjacent nodes in the set I. In other words, in a **maximal independent set** all nodes either belong to the set I or they *cannot* join it because one of their neighbors has already joined. It is trivial to see that the problem is an LCL as checking local neighborhoods will suffice to determine if the solution is valid or not. For an example of a solved instance of the maximal independent set problem, see Figure 2.1c.

Finally, a **maximal matching problem** is a problem in which nodes need to produce a matching such that each node is either matched with one of its neighbors or it cannot be matched because all of its neighbors are matched. That is, no node can be left unmatched while at least one of its neighbors is unmatched. It is trivial to see that the problem is an LCL since



(a) Vertex coloring

(b) Edge coloring

(c) Maximal independent set

(d) Maximal matching

Figure 2.1: Some of the common LCL problems

each node can check its radius-1 neighborhood to check whether it is matched with a neighbor or all neighbors are matched with someone else. And if the condition of the previous sentence is not satisfied, the output is invalid. Refer to Figure 2.1d, for a solved instance of the maximal matching problem.

2.6 Decidability and undecidability

Before moving forward, it is important to remind about a notion of decidability in the context of theoretical computer science. It is assumed that the reader already has some basic knowledge of complexity theory, computability, Turing machines, automata, and languages. The content of this section follows the definitions and formalisms as defined in the book by Sipser [57] unless specified otherwise.

First of all, recall that a deterministic Turing machine is called *decider* if it halts on all inputs. Similarly, a nondeterministic Turing machine is called a decider, if it ends up in a halting state in all of its branches. We then call a language *decidable* if and only if there is some Turing machine that decides it. That is, a language L is decidable if and only if there exists some Turing machine M such that given an input x, where $x \in L$, it always halts.

A decision problem is an algorithmic problem that has only two valid solutions to it: "yes" and "no". We assume that a decision problem has an infinite set of inputs. That is, a decision problem defines an infinite but possibly restricted set of valid inputs, and for each input, only one of the two outputs is the correct output. An algorithm is said to solve a problem P if and only if, given any input from a set of valid inputs, it can always produce the correct output. Recall also that we can represent any decision problem P as a formal language L_P, where a string $x \in L_P$ if and only if, given that x is an input to the problem P, the correct output is "yes". To clarify, for each problem P, there is a language L_P such that if P's output on some input x is "yes", then $x \in L_P$, otherwise $x \notin L_P$. From now on, I will call such language L_P that "represents" a decision problem P_L as *corresponding language* of problem P.

Then, a decision problem P is *decidable*, if and only if there is a decidable corresponding language L_P. On the other hand, a decision problem is *undecidable*, if and only if, there is no decidable corresponding language L_P. Note that a problem is undecidable if and only if it is not decidable.

To prove that a problem P is decidable, it is sufficient to show an algorithm (formally a Turing machine) A, such that A correctly solves the problem for all valid inputs. That is, such an algorithm A must always halt and must produce the output "yes" if and only if it is the correct output of

the problem P given the input. Thus, proving decidability requires finding such an algorithm, and then proving that A always halts and that A is indeed correct. To prove that a problem is undecidable is however less trivial. One of the common techniques is to show that the problem can be reduced to the Halting problem [40, 62].

2.7 Automated synthesis and classification

Automated classification, broadly speaking, is a problem where objects of a certain type, which possibly share some properties in common, are required to be classified into several, often in-advance known categories automatically, that is, without or with minimal human supervision. Due to the recent successes in the field of machine learning, numerous automated classification problems have been solved, or at least for many of the problems, significant progress has been made. In particular, the field of automated classification of image, sound, and text data based on deep neural networks – one of the popular machine learning techniques that have had multiple breakthroughs in recent years – had some significant results in the areas of medicine, biology, agriculture, and many others [15, 21, 25, 34, 56, 65].

However, automated classification is also possible beyond the physical media such as images, sounds, etc. For example, it is often useful to classify abstract problems, e.g. mathematical or computer science problems. Indeed, Fulton et al. [26] have shown an approach for automated classification of Sturm-Liouville problems [66].

automated synthesis is a problem of automatically synthesizing or finding a solution to a certain problem given a set of constraints and/or objectives. For example, automated synthesis is fairly common in the field of electrical engineering, and is commonly applied when designing analog and digital circuits [14, 44]. Furthermore, with the recent rise in popularity of quantum computing, the need has arisen for the design of quantum circuits. Similar to the classical cases, the synthesis of quantum circuits has become a popular research direction with numerous publications on the matter [35, 41, 43].

In general, algorithm synthesis in the context of theoretical computer science is undecidable. However, there are some research directions in the field of distributed computing and distributed algorithms, where some forms of automated synthesis and/or automated problem classifications have led to successful developments [3, 10, 13, 18, 22, 23, 31, 36, 55]. Moreover, the notions of automated synthesis and automated classification are often interconnected in the context of distributed algorithms. To demonstrate this, I take an example of a common problem in a distributed computing: given a

problem Π, we need to automatically determine its complexity. Assigning a complexity class to the problem Π can be viewed as a classification problem here. On the other hand, proofs of the type that demonstrate that Π belongs to a certain complexity class are often constructive in nature. That is, to show that a problem can be solved in the number of rounds x, we often need to develop an algorithm that solves the problem in the stated number of rounds. But developing such an algorithm can potentially be automated, meaning that *automated algorithm synthesis* can be used. Thus, in the context of distributed computing, automated algorithm synthesis and automated problem classification are often two perspectives on the same thing: determining whether a problem Π can be solved in x rounds or is fundamentally more difficult.

2.8 Recent developments in the automated classification of LCL problems

At this point that the theoretical background has been given, it is useful to give a background to much more recent developments in the distributed algorithms research. As the branch of research of studying LCL problems had been reaching maturity, many focused on the idea of automated classification of LCL problems. That is, the question of interest is whether we can create meta-algorithms that would take the description of an LCL problem as its input and produce the problem's round complexity as an output. Figure 2.2 conceptually depicts an example of a meta-algorithm. This being the core topic of the thesis, it is thus important to describe all the prior work done in this direction. In the rest of the section, I will introduce some of the negative as well as positive results related to the automated classification of LCL problems.

The questions of whether a certain family of LCL problems can be automatically classified are often referred to as decidability questions. Roughly speaking, a problem is said to be decidable if there exists an algorithm that, given the problem, will output its complexity. Such decidability questions are often studied in relation to the whole family of graph problems and not to individual instances of problems.

First of all, it is important to notice that it has been proven that in general LCL problems are undecidable. In fact, already in the cases when the graph family is a grid, LCL problems cannot be automatically classified [13, 46].

Nevertheless, many interesting LCL problems are still decidable. For example, it was known for several years now that LCLs on paths and cycles

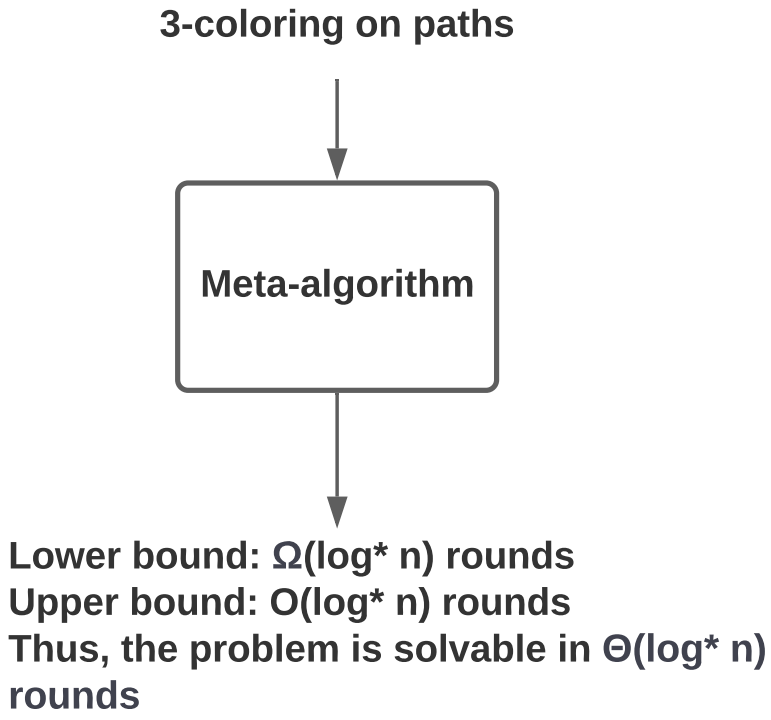


Figure 2.2: Conceptual representation of a meta-algorithm

are decidable [3, 13, 46]. Besides, certain types of LCL problems are also decidable on trees [18].

But the fact that a problem is decidable in theory does not always imply that it is practical to use an algorithm for automated classification of such a problem. For instance, it is known that even in the case of paths and cycles, if a node labeling is given as part of the input, decidability becomes PSPACE hard [3].

However, not everything looks that gloomy, and many interesting and sufficiently broad families of LCL problems are both decidable, and it is feasible to automatically classify them in practice. Indeed a lot of work has been done attempting to derive practical algorithms that determine the complexity of a given LCL problem, especially in trees.

Balliu et al. [4] has shown that a complete classification of binary labeling

problems (problems where a node's output is allowed to be only one of the two possible labels) at least in a deterministic setting is possible and moreover can be done in $O(1)$ time since computational complexity of such problems can be simply looked up in a table in a constant time. In addition to this, the paper also shows that it is decidable to classify at least *some* of the randomized binary labeling LCL problems [4].

Building on top of the work outlined in the paper, Rocher [50] has developed a meta-algorithm that classifies *almost* all ternary labeling problems on trees in the deterministic setting. Furthermore, he also implemented the meta-algorithm as a computer program written in Python programming language [49]. Although the implementation focuses mainly on the case of trees with degree 3, the techniques described in the manuscript are applicable to ternary problems on trees of higher degrees as well.

In addition to this, LCL problems on trees and cycles are fully decidable in polynomial time as was shown by Chang et al. [19]. The algorithm for classifying the problems can be used in practice and has been implemented in Python programming language by Aalto's Distributed Algorithms research group [60].

Furthermore, a technique called round elimination was introduced in the context of LCL problems by Brandt et al. [11]. It is a series of mechanical steps that, if followed, in many cases transform an LCL problem Π_0 into another LCL problem Π_1. An interesting property is that, given certain assumptions and the fact that Π_0 is solvable in constant time e.g. in T rounds, Π_1 is guaranteed to be solvable in exactly $T-1$ rounds. This single property has numerous implications. For example, if we apply the technique k times and in the end obtain an LCL problem Π_k such that it is zero-round solvable, we know that the original problem Π_0 is exactly k-round solvable. Moreover, round elimination is not only possible in theory but is also very much feasible in practice. Indeed, Olivetti [47] has implemented a computer program that, given an LCL Π_0 solvable in T rounds, it produces another LCL Π_1 solvable in $T-1$ rounds. There even exists a web interface for the program, and one can apply the technique and obtain its results within milliseconds. I will describe the technique and its implications in much more detail in the next section.

Finally, very recently, Balliu et al. [5] demonstrated that LCL problems on rooted trees are completely decidable. In particular, an LCL problem on rooted trees can be completely classified in both deterministic and randomized settings. The only exception to this completeness is deciding whether a given problem has round complexity of $\Theta(n)$ or it is unsolvable. In addition to the theoretical results in the manuscript, the authors also provide a freely available open-source software that can classify an LCL with a relatively

small number of labels in a matter of milliseconds [58].

Chapter 3

Meta-algorithms used in the solution

This chapter will describe in detail all the meta-algorithms that are used in the final solution as subroutines. Chapter 5 will then describe how the final solution was built on top of the meta-algorithms described in this chapter.

3.1 Round elimination

This section explains in detail the round elimination (RE) technique introduced in Chapter 2, as well as the implementation of the round elimination technique as a computer program written in Rust [11, 47]. Besides, I will demonstrate that all LCL problems that we're interested in can be represented in a formalism of round elimination, which implies its wide applicability.

As already mentioned in the previous chapter, round elimination is a technique that, given an LCL problem Π_0 as an input, produces another LCL problem Π_1 which can be solved exactly one round faster. For round elimination to work, the input must comply with the following two constraints: Π_0 has to be a problem on a (δ, β)-biregular graph, and the number of rounds in which Π_0 can be solved should not be "too large". Indeed, the last constraint is a rather curious one since we rarely know the round complexity of an input problem when using round elimination. However, we can nevertheless use this rather inconvenient constraint to our advantage, which will be demonstrated below.

Furthermore, when applying round elimination, we talk about *active* and *passive* nodes. Since (δ, β)-bipartition of an input problem is already given (see our assumption above), nodes of degree δ are assumed to be active and

nodes of degree β are assumed to be passive. For a problem to be a valid input to round elimination, it has to be reformulated as a problem on a (δ, β)-biregular graph where only one partition of nodes produces some output while the other partition does not produce any output but instead checks that their radius-1 neighborhood's outputs comply to previously specified restrictions.

In order to demonstrate the technique, we will have as our running examples two canonical problems: weak 3-labeling and sinkless orientation.

Weak 3-labeling in this context is a problem on δ-regular trees where each node labels its incident edges in such a way that no node v in the graph has all its incident edges labeled with the same label. Besides, if two nodes v and u are incident to the same edge $e = \{v, u\}$, such edge e has to be labeled with the same label from both "sides". In other words, two neighboring nodes cannot output different labels on the same edge. It is easy to see that, although the initial problem is specified for a regular tree, we can obtain an equivalent problem for a $(\delta, 2)$-biregular tree by replacing edges with nodes as described in Section 2.1.2. Further, if we assume that all nodes of degree δ are active and all nodes of degree 2 are passive and that nodes of degree δ cannot have their incident edges labeled all with the same label, and nodes of degree 2 must have both of their incident edges labeled with the same label, we obtain an LCL problem equivalent to weak 3-labeling but in a formalism suitable for round elimination. To formalize, we can define the problem as follows:

$$\begin{aligned}
&\Sigma = \{A, B, C\},\\
&A = \{\{A, B, C\}, \{A, A, B \text{ or } C\}, \{B, B, A \text{ or } C\}, \{C, C, A \text{ or } B\}\},\\
&P = \{\{A, A\}, \{B, B\}, \{C, C\}\},
\end{aligned}$$

where Σ is an allowed alphabet, A is a set of *configurations* allowed for active nodes, and P is a set of configurations allowed for passive nodes. Here, we assumed that $\delta = 3$. Notice that each configuration is a set of labels that are allowed to be outputted on ports incident to any active/passive node v. Notice also that the order of the labels in a single configuration does not matter. Finally, in this particular problem and all problems in the context of round elimination, we usually do not care about leaf nodes. That is, leaf nodes are unconstrained and are fine with any configuration. Next, I will describe another problem using the same formalism.

Sinkless orientation (SO) is a problem where each non-leaf node has an outdegree of at least 1. We will consider the problem in the context of a δ-regular tree. Notice that we can interpret δ-regular tree as a (δ, δ)-biregular tree with both active and passive nodes being of degree δ. Recall also that since the tree is biregular, we necessarily assume bipartition of the nodes. In

other words, we assume that 2-coloring is already given for the tree. Then, we can define the problem formally as follows:

$$\Sigma = \{I, O\},$$
$$A = \{\{O, I \text{ or } O, I \text{ or } O\}\},$$
$$P = \{\{I, I \text{ or } O, I \text{ or } O\}\}.$$

Here we again assumed that $\delta = 3$. Label I here represents an incoming edge and label O represents an outgoing edge. Note that because the labels are produced only by active nodes, an outgoing edge O for an active node is an incoming edge for a passive node. And vise versa: an incoming edge I for an active node is an outgoing edge for a passive node. Thus, the sets of configurations A and P can be interpreted so that each node – no matter active or passive – has to have at least one outgoing edge, and the rest of the edges can be anything.

Now that I have defined our two example problems, I will apply the round elimination technique on each one, describe intermediate steps, and analyze the results. Notice that the low-level technicalities of round elimination are omitted in the current thesis for brevity. For in-depth technical details, a reader is encouraged to refer to the original paper by Brandt [11] or to the recently published textbook on distributed algorithms by Hirvonen et al. [32].

Let us denote the initial weak 3-labeling problem as Π_0. After applying round elimination, we get as its output another LCL problem Π_1. The formal description of the problem is as follows:

$$\Sigma = \{A, B, C\},$$
$$A = \{\{A, A\}, \{B, B\}, \{C, C\}\},$$
$$P = \{\{A, B, C\}, \{A, A, B \text{ or } C\}, \{B, B, A \text{ or } C\}, \{C, C, A \text{ or } B\}\}.$$

The problem Π_1 is not zero round solvable since after zero rounds each node only knows its own degree and unique identifier. Thus, no matter what is the mapping from unique identifiers to the output is, we can construct such a graph, where there exists a passive node whose all three neighbors will output the same label. But set P does not contain any configurations consisting of the same three labels, therefore such output would be invalid. So we just have another problem that looks a lot like our previous problem Π_0 (essentially sets A and P have simply been swapped). Let us run round elimination once again, now using Π_1 as its input. We obtain Π_2, which looks as follows:

$$\Sigma = \{A, B, C, D, E, F, G\},$$
$$A = \{\{C, E, F\}, \{C, D, G\}, \{B, E, G\}, \{A, F, G\}\},$$
$$P = \{\{ACEG, ACEG\}, \{BCFG, BCFG\}, \{DEFG, DEFG\}\},$$

where we use $X_1X_2...X_k$ notation to mean the same as X_1 or X_2 or ... or X_k. Now this problem is zero round solvable. Indeed, if all active nodes output $\{C, E, F\}$ on their incident edges in an arbitrary order, passive nodes will be satisfied (that is to say that all passive nodes will find a configuration from the set P) in all cases.

Now since we know that Π_2 is 0-round solvable, using the justification already described above, we can conclude that our original problem Π_0 is exactly 2-round solvable. Thus, weak 3-labeling on $(3, 2)$-biregular graphs is solvable in $O(1)$ rounds. Observe that we have just used the round elimination technique to *prove* that an LCL problem can be solved in a constant number of rounds.

Next, we will analyze SO in a similar manner. Let us again denote SO as Π_0 and apply round elimination to it. We obtain the following output problem:

$$\begin{aligned}\Sigma &= \{A, B\},\\ A &= \{\{A, B, B\}\},\\ P &= \{\{B, AB, AB\}\}.\end{aligned}$$

Applying round elimination once again, we get

$$\begin{aligned}\Sigma &= \{A, B\},\\ A &= \{\{A, B, B\}\},\\ P &= \{\{B, AB, AB\}\}.\end{aligned}$$

It turns out that $\Pi_2 = \Pi_1$. But we also know that the round complexity of Π_2 has to be smaller by 1 round than that of Π_1. Although this sounds like a contradiction, it turns out that the key here is to pay attention to the initial assumptions that I mentioned at the beginning of the section. Namely, the number of rounds T of the original problem cannot be "too large". Hence, in the case of SO, the number of rounds necessary to solve it is "too large". Moreover, it has been shown that in cases like this, when, after applying round elimination repeatedly, we encounter a problem that we have already seen before, the initial problem Π_0, in most of the cases, is solvable in $\Omega(\log n)$ rounds in deterministic setting and in $\Omega(\log \log n)$ rounds if randomness is allowed. Thus, we can use round elimination not only to prove upper bounds but also lower bounds for LCL problems on biregular graphs.

Although I have only demonstrated the technique for two problems, the same idea following for most of the other LCLs on biregular graphs. In general, RE can be used in an automatic manner to prove constant upper or logarithmic and polylogarithmic lower bounds for a wide range of LCL

problems. Furthermore, the open-source implementation of the technique written in Rust programming languages and publicly available as a command-line tool and via a web interface will hugely assist in using RE as part of my software solution for automated classification of LCL on trees.

Finally, it is worth demonstrating that the formalism used in round elimination (and in the Round Eliminator implementation) is universal.

Theorem 3.1.1. *Any LCL problem P on undirected trees, where leaves, nodes close to leaves, and nodes of irregular degrees are not constrained, can be represented in the formalism of round elimination.*

Proof. If r is the checkability radius of the LCL problem P, one can define a new problem that is the same as the old one except that each node has to output its entire radius-r output (potentially including the topology if the graph in question is not a regular graph). It is also possible to give this whole output on each port around the node together with the information about the location of the port in the radius-r ball (in other words, where this port is in the radius-r ball). Therefore, the fact that in the Round Eliminator we encode configurations on half-edges is not a problem. There are only two things that are special about the Round Eliminator representation. The first thing is that the complexity of the LCL problem may change by r additively (but since r is a constant, this does not matter for us since we are only concerned in the current work with asymptotics, and in addition, we treat $O(0)$ and $O(1)$ as the same complexity class – namely $O(1)$). The second thing is that we assume that in the original LCL definition, correctness does not depend on the existence and various properties of cycles in the graph. For instance, the LCL problem "am I contained in a triangle" cannot be translated to the Round Eliminator's representation. But again, in the current work, we are only concerned with LCLs on trees. Therefore, no matter what an LCL problem is, as long as it is a problem on undirected trees, where leaves, nodes close to leaves, and nodes of irregular degrees are not constrained, we can represent such problem in the formalism of round elimination. □

3.2 Isomorphic LCL problems

This section will introduce a notion of *problem isomorphism* in the context of LCL problems. I will also give an example of two isomorphic LCL problems.

When dealing with LCL problems, it is often useful to recognize whether certain problems are isomorphic to each other. The notion of LCL problem isomorphism is in many ways similar to the notion of graph isomorphism. I

will define problem isomorphism for the representation used in round elimination. Because we can represent any LCL problem in the formalism of round elimination (as was shown in Theorem 3.1.1), the isomorphism definition is useful for all LCL problems too. An *isomorphism* between two problems P_1 and P_2 is a function f such that f is a bijection, and f maps an allowed output label of the problem P_1 to an allowed output label of the problem P_2, and the underlying graph of problem P_1 is the same as the underlying graph of problem P_2, and the number of allowed configurations for active and passive nodes in P_1 is the same as the number of allowed configurations for active and passive nodes respectively in P_2, and if we apply the function f to the labels used in the configurations of active and passive nodes in P_1, we will receive all those and only those configurations that are allowed for active and passive nodes respectively in P_2. Similar to graph isomorphism, if an isomorphism exists from P_1 to P_2, then also an isomorphism exists from P_2 to P_1. If there exists an isomorphism between some two problems, we say that such problems are isomorphic (to each other).

Consider a problem P_1 on 3-regular trees, where leaves can produce any output, and non-leaves can output only one of the following two sets of labels on their ports: $\{A, B, B\}$ and $\{A, A, A\}$. Now consider another problem P_2 on 3-regular trees, where we also do not care about leaf nodes, and non-leaf nodes can output one of the following two sets of labels on their ports: $\{A, A, B\}$ and $\{B, B, B\}$. Here, we will ignore the actual meaning of the problem. What is important is to notice that these two problems are isomorphic to each other. Indeed, a function f that maps label A of P_1 to B of P_2 and label B of P_1 to A of P_2 is an isomorphism. Among other things, this implies that the problems can in some way be viewed as two name variations of the same single LCL problem. This means, for example, that if we know the round complexity of P_1, we also know the round complexity of P_2.

3.3 Automata-theoretic lens classifier

In this section, I will describe a technique that can be used to automatically classify LCL problems on paths, cycles, and in some cases, on rooted trees. Moreover, such classification can be done in polynomial time in the size of the description of the problem being classified [19].

On a high level, the classification algorithm first represents an input LCL problem as a directed graph and then, by analyzing the graph and its properties, decides which complexity class the given LCL problem belongs to. First, however, similar to the case of round elimination, it is important to explain a representation that the technique requires the input LCL problem

to be in. We will initially concentrate on the case of LCLs on paths and cycles and only afterward cover the extension to rooted trees.

The representation is often referred to as *node-edge-checkable* formalism. In it, each node outputs one label on each of its *ports* associated with its incident edges. Since only paths and cycles are allowed as graph families for input LCLs, most of the nodes have exactly two incident edges (except for endpoints nodes in paths, but these will be explained separately below). Therefore, a node constraint is a pair of labels, and node constraints is a set of such pairs. Each pair essentially means that a node can output these two labels on its incident edges. Edge constraints are, similarly, a set of pairs of allowed labels from the perspective of an edge. For example, consider an edge $e = \{u, v\}$. If u outputs label A on its port associated with edge e and v outputs label B on its port associated with the edge e, then the constraint (AB) (or (BA)) must be included in the set of edge constraints. Otherwise, such output would be invalid. For the case of paths, where two nodes have only one incident edge, it is also necessary to specify start-constraints and end-constraints, each consisting of a set of labels that two of the endpoint nodes are allowed to output on their only ports. To make the representation of the input LCLs clearer, here is an example of vertex 3-coloring in cycles (which can also be found in the original paper) [19]:

$$C_{\text{node}} = \{11, 22, 33\},$$
$$C_{\text{edge}} = \{12, 21, 13, 31, 23, 32\},$$

where C_{node} is a set of node-constraints and C_{edge} is a set of edge-constraints.

Set $\{11, 22, 33\}$ here allows each node to be colored in one of the three colors (each node outputs its color to both ports), and each edge is allowed to connect two nodes of any color as long as these colors are not the same (note that there is no e.g. 11 or 22 in the set C_{edge}). To give another example, maximal matching (MM) in cycles would be encoded in the node-edge-checkable formalism as follows:

$$C_{\text{node}} = \{00, 1M, M1\},$$
$$C_{\text{edge}} = \{01, 10, 11, MM\}.$$

It turns out that in the case of paths and cycles, all LCLs belong to one of the four complexity classes: $O(1)$, $\Theta(\log * n)$, $\Theta(n)$ and unsolvable. Notice that the complexity class of $\Theta(\log n)$ disappears in both deterministic and randomized settings. The technique described in the paper then takes the description of a problem as four sets: C_{node}, C_{edge}, C_{start} and C_{end}, and following a polynomial-time algorithm classifies the problem into one of the four

complexity classes. The classification technique works for all LCL problems on trees and cycles.

Moreover, some problems on rooted trees can be classified as well. For the technique to work, the problem has to be describable in a so-called *edge-checkable* formalism. In other words, we are allowed to only specify constraints for the edges of the tree i.e. C_{edge} set, nodes can output any label on the ports, but any single node v is allowed to only output the same label on all its ports. Also, we are not concerned in the current formalism with the output of leaves and the root – only nodes in the middle of the graph are constrained. Although such a formalism imposes certain restrictions on what problem families can be used, there are still a lot of interesting problems that can be classified this way. One restriction to keep in mind is that in the edge-checkable formalism it is not possible to place restrictions on the number of nodes with a certain output label among children of some node v. For example, it is not possible to represent problems of type "a node with label X can have at most one child with label Y", because if $(X, Y) \in C_{\text{edge}}$, then a node labeled with X can have an arbitrary number of children labeled with Y. If, on the other hand, $(X, Y) \notin C_{\text{edge}}$ and a node v is labeled with X, then none of its children can be labeled with Y.

3.4 Classification of binary labeling problems

This section is based on the results obtained by Balliu et al. [4]. The paper demonstrates that the case of binary labeling problems on (bi)regular unrooted regular trees is fully decidable in the deterministic LOCAL model. The paper also shows that in many cases, the randomized complexity of an LCL problem can also be decided following the described technique, although the complete decidability in the randomized setting still remains an open research question.

As in the previous cases, we will start the section by explaining the representation of an LCL that is compatible with the described methods. The representation assumes a (δ, σ)-biregular unrooted tree. Such biregularity assumes that a proper 2-coloring of the graph is initially known. As was already shown previously, biregularity is not necessary. It is sufficient that an underlying graph of an input problem is simply a regular unrooted tree. Given that, it is then easy to transform the δ-regular tree to a $(\delta, 2)$-biregular tree by replacing each edge $e = \{u, v\}$ with a new node that is connected to nodes u and v via new edges and not connected to any other nodes. Finally, in the given setting we do not care about nodes with smaller degrees e.g. leaves. This is because any irregularities in a graph can only make it easier

for nodes around to solve the problem, therefore, it was decided to leave any nodes with degrees other than δ and σ unconstrained [4].

Each problem is represented as a 4-tuple $\Pi = (\delta, \sigma, W, B)$, where δ and σ are degrees of *regular* nodes, W is a set of so-called *white constraints* and B is a set of *black constraints*. To explain what exactly the sets W and B represent, it is important to recall that this setting is concerned with *binary* labeling only, which means that each node can only output two labels on its edges. I will refer to such two labels as "zero" and "one" labels (denoted as 0 and 1). The problem is assumed to follow the edge-labeling formalism, in which each node outputs a label on each of its incident nodes so that any two adjacent nodes v and u can only output the same label on their common incident edge $e = \{u, v\}$. In addition to this restriction, sets W and B restrict the choice of outputs as well. A *white* node's output is valid if and only if the sum of labels on all its incident edges (that is, summing output labels on its incident edges as zeros and ones) is such a number s_W that $s_W \in W$. Similarly, the output of a *black* node is valid if and only if the sum of all labels on its incident edges sums to some number s_B such that $s_B \in B$.

As an example, we will consider the already familiar *sinkless orientation* problem. The problem can be represented as $\Pi = (\delta, 2, \{0, 1, 2, \ldots, \delta - 1\}, \{1\})$, where δ is any integer > 2. In this case, the degree of white nodes is δ, and the degree of black nodes is 2. Thus, white nodes here are the "actual" nodes of an underlying graph, while black nodes represent edges in the original graph. Furthermore, an output label 1 represents an edge that is directed from a white node towards a black node, while label 0 represents an edge directed from a black node towards a white node. That is why W contains all integers from 1 to $\delta - 1$ but not δ. This highlights the fact that each white node must have an outdegree of at least 1, or in other words, that edges of a white node cannot all be incoming. From the set B we can see that each black node must have exactly one incoming and one outgoing edge. This is because each edge (which is represented by a black node) in the original graph must be properly oriented, or in other words, it must have exactly one head and one tail.

I will not go into detail to describe the reasons how the decidability methods described in the paper work. The details are very technical and require a huge theoretical background to cover beforehand. It is worth noting, however, that given a description of a problem Π, it is possible to classify it in $O(1)$ time since the authors of the paper provide a simple table where one can easily look up what complexity class a certain LCL problem belongs to. Thus, given a description of a binary labeling LCL on (bi)regular unrooted tree, it is trivial to determine its complexity class (albeit only in the deterministic setting) for both computers and human beings.

3.5 Classification of ternary labeling problems

This section will describe an unpublished work by Rocher that builds on top of the results introduced in the previous section [49, 50]. The report shows that a majority of ternary labeling LCL problems on unrooted (bi)regular trees are decidable in the deterministic LOCAL model. Although not all such problems can be classified using the techniques, many interesting ternary labeling problems on trees are indeed covered by the presented methods. Similar to the previous sections, inputs are not allowed in the current setup.

The representation of a problem resembles that of the previous section. A ternary labeling problem is represented as $\Pi = (\delta, \sigma, W, B)$ where δ and σ are degrees in the underlying (δ, σ)-biregular tree, while W and B are constraint sets for white and black nodes respectively. Each constraint in the sets W or B is a 3-tuple of the form (x_0, x_1, x_2) where x_0 represents how many incident (to a given node) edges can be labeled with label "0", x_1 shows how many edges can be labeled with "1", and x_2 – with "2". Similar to the case of binary labeling in unrooted regular trees, we assume that only these three labels are allowed to be outputted by each node. Also, each node outputs one of the labels of each of its incident edges in such a way that each edge is labeled with the same label by each of its incident nodes. Furthermore, we again are only concerned with nodes of degree δ or σ. Nodes of any lower degree only make the problem easier.

We will consider vertex 3-coloring as an example problem to demonstrate the formalism. In this problem, each node picks one of the three labels such that none of its neighbors pick the same label. In the case of a 3-regular tree (which is the same as $(3, 2)$-biregular tree, as we can always turn edges into nodes of degree 2), the problem looks as follows:

$$\begin{aligned} \Pi &= (3, 2, W, B), \\ \mathrm{W} &= \{(3, 0, 0), (0, 3, 0), (0, 0, 3)\}, \\ \mathrm{B} &= \{(1, 1, 0), (1, 0, 1), (0, 1, 1)\}. \end{aligned}$$

Here, W signifies the fact that each node has to be colored with exactly one label. The node then outputs the single picked color on all of its incident edges. Then, black nodes that represent edges in the original underlying graph make sure (hence the constraint set B) that no two adjacent nodes pick the same color.

Nevertheless, it is also important to discuss the limitations of the work. First of all, the work concentrates mainly on the case of $(3, 2)$-biregular trees.

The author claims that the methods described can just as well be applied to cases of higher degrees, but it is unclear how practical it would be both in terms of the time it would require to classify a single problem and in terms of how many problems of higher degrees will be tightly classified. Secondly, even in the case when $\delta = 3$ and $\sigma = 2$, 106 non-isomorphic problems have not been classified tightly, or in other words, there are 106 problems with non-matching lower and upper bounds. Finally, the developed methods are not well-suited for situations when we would need to classify only one problem. Instead, the algorithm first classifies all non-isomorphic problems on (δ, σ)-biregular tree, and afterward allows the user to query one problem at a time from the list of already classified problems. Among other things, it means that to classify a single problem on $(4, 2)$-biregular trees, in the current implementation, it is necessary to first classify all non-isomorphic problems on $(4, 2)$-biregular trees, which might take a considerable amount of time. Finally, the write-up presents classification techniques only for the deterministic setting, which further limits the kinds of LCLs that can be classified using the techniques.

3.6 Classification of problems on rooted trees

This section will introduce a recent work by Balliu et al. [5] and how it can be used to automatically classify LCL problems on rooted trees. In particular, I will concentrate on the practical implementation of the ideas presented in the manuscript. The computer program implementing some of the ideas presented in the publication was written in Python programming language by Studený and Tereshchenko [58]. The program is limited to the case of binary trees only, although the theorems presented in the manuscript generalize to rooted trees of any bounded degree.

The manuscript demonstrates that for LCL problems on rooted trees, randomness does not help at all. This means that if a problem Π has the round complexity of $\Theta(X)$ in the deterministic setting, Π also has the round complexity of $\Theta(X)$ in the randomized setting. The converse also holds, i.e. a randomized complexity of $\Theta(X)$ for some problem implies deterministic complexity of $\Theta(X)$ for the same problem. The aforementioned holds for any LCL problem where the graph family is a rooted tree. Moreover, the manuscript also shows that any such LCL will necessarily fall into one of the four complexity classes (if it is solvable at all). The complexity classes are $O(1)$, $\Theta(\log * n)$, $\Theta(\log n)$ and $\Omega(n)$. In particular, the complexity class of $\Theta(\log \log n)$ is non-existent for LCL problems of this graph family.

In turn, the developed software provides a simple way to ask for the com-

plexity of any LCL problem on binary trees. As an output, the program returns one of the above-listed complexity classes. The program takes as input a number of allowed *configurations*, each in the following format: (child's output label)(node's own output label)(another child's output label). For example {112 121} means that if a node x outputs 1 as its output label, its two children must have two different output labels: 1 and 2. If a node x outputs 2 however, then both of its children must output 1. All the other combinations of output labels of any given node x and its children – referred from now on simply as configurations – are forbidden. All other output labels except for 1 and 2 are also forbidden. Finally, it is worth mentioning that, similar to previous meta-algorithms, we do not care about node's outputs if a node is a leaf, a root, or has only one child. In other words, if a node x has less than 2 children or is a root, it does not have any restrictions on its own output labels. However, a leaf's parent y might have restrictions on possible output labels of its children (and the leaf is a child of y). In this case, the restrictions still apply to the leaf node.

As an example of an interesting problem, we will consider *3-coloring in binary trees*. In the described paradigm, the problem can be represented as follows:

$$\{212,\ 213,\ 313,\ 121,\ 123,\ 323,\ 131,\ 132,\ 232\}.$$

It is easy to see why the set of configurations represent the 3-coloring problem. Indeed, a node x can be of any of the three colors. However, if a node x has color c, none of its children is allowed to have c as its output color. On the other hand, children can be of two different colors or the same color as long as this color is not c.

As a second example, consider *maximal independent set in binary trees*. Its representation in the described formalism would be as follows:

$$\{a1a,\ a1b,\ b1b,\ bab,\ bb1,\ 1b1\}.$$

Here, label 1 indicates belonging to the independent set I, label b indicates the fact that the node has at least one child which belongs to I, and a indicates that the node's parent belongs to I. Notice that nodes with label 1 are not allowed to have children with label 1 – this ensures independence of the independent set I. Besides, nodes with labels a or b must have a parent or at least one child, respectively, that belongs to I. This ensures maximality of the independent set. Therefore, each node is either a part of the independent set, has a child that belongs to I or a parent that belongs to I. All this together makes sure that the set I is independent and maximal.

Thus, despite its simplicity, the representation language is very powerful and can represent almost all LCL problems in rooted trees, with the exceptions of problems that in one way or another emphasize nodes near leaves and/or roots [5]. Furthermore, the practicality of the package – especially the fact that most of the problems interesting for the purpose of the thesis can be classified in a matter of several milliseconds – makes it a perfect candidate to be used as one of the meta-algorithms in my solution.

Chapter 4

Methodology

This chapter will describe in detail the research goals of the thesis project. Besides, I will also outline the scope of the thesis work.

4.1 Research goals

As has been shown in the previous chapters, a lot is known about the decidability of LCL problems on trees. In particular, plenty of research has been done on the topic of automated classification of LCL problems. However, the results of the research are scattered across numerous papers. Moreover, all of the meta-algorithms described before use different representations for the LCL problems that they accept as an input. The issue gets even more complicated by the fact that the existing practical implementations of the meta-algorithms are written in different programming languages, accept input in different forms, use different internal representations of a problem and produce output in different formats.

Due to this, researchers would have to download and install multiple software packages on their machines and learn the formalisms for representing LCL problems for each of the meta-algorithm representations. Most importantly, since each of the meta-algorithms' implementations is capable of classifying only a relatively narrow subfamily of LCL problems, the researchers would have to constantly keep in mind the information about what families of problems can be classified by what meta-algorithm. All this makes it difficult and impractical for members of the research community to use the meta-algorithms in their everyday work.

Therefore, the goal of this thesis is a software tool that would store most – if not all – of so far existing knowledge on the complexity of LCL problems on trees. The tool would, in a sense, encapsulate all of the capabilities of the

existing meta-algorithms. The tool would also allow for queries regarding a specific problem or a group of problems. As an example, we can imagine the following two types of queries:

- Is the complexity of a problem X already known (based on the existing meta-algorithms and/or accumulated individual complexity results)? If so, what is the complexity in both deterministic and randomized settings?
- Return all problems on *binary rooted trees* that have complexity of $\Theta(\log^* n)$.

In addition to this, the tool's output would include references to the sources (published papers, certain theorems, ad-hoc implemented scripts, etc.) that the returned results are based on.

All this would make it more accessible for members of the research community to use the powers of the existing meta-algorithms. This, we believe, will be highly useful for the distributed algorithms researchers, as it would allow them to get access to the existing information about the complexity of LCL problems on trees in a matter of seconds.

4.2 Scope

When selecting the scope of the thesis, it is necessary to decide on the limits in terms of the properties of the software. I present the limiting properties as a list of questions.

1. What graph families should my tool be able to work with?
2. What meta-algorithms are to be used as part of the solution?
3. Are datasets of individual complexity results to be used in addition to the meta-algorithms? If so, which of the datasets should be used?
4. How general should the query be, and what parameters the query should include?

To answer these questions, I utilize the existing literature on the subject, which was partly presented in Chapters 2 and 3. As was already explained earlier, problems on general graphs are not decidable. At the same time, all of the results listed in Chapter 3 can be applied to problems on trees. Thus, I have decided to limit ourselves to trees as an answer to the first question in the list.

At the moment of writing the thesis, the number of meta-algorithms is quite small. Indeed, all of the meta-algorithms that the author is aware of were listed and explained in Chapter 3. Therefore, there are only five meta-algorithms to use: round elimination technique, the automata-theoretic lens classifier, classifier for binary labeling problems on unrooted trees, classifier for ternary labeling problems on unrooted trees, and classifier for problems on rooted trees. Moreover, for each of these classifiers, there exists a practical implementation of the meta-algorithm, even if in a limited form (e.g. an implementation of the classifier for problems on rooted trees, currently only supports problems on *binary* rooted trees). Thus, I have decided to use all of the five listed meta-algorithms as part of the solution.

There are only two LCL problem datasets the author is aware of. One of them is the dataset of binary and ternary labeling problems on binary rooted trees, which is a part of the Python package for classifying binary rooted trees [61]. The other one is the dataset of binary and ternary labeling problems on unrooted $(2, 2)$-biregular and $(3, 2)$-biregular trees, which is a part of the TLP classifier Python package [50]. Since the latter dataset is already incorporated in the TLP classifier, the former dataset is the only one that I eventually decided to use in addition to the aforementioned meta-algorithms.

Finally, I will attempt to answer the last question. This is perhaps the most difficult one from the list, as the research needs of different people involved in the distributed algorithms research might differ a lot, and thus the parameters of the query that might prove to be useful will vary from case to case too. It was thus decided to first, based on some assumptions and general knowledge of the needs of the research community, to come up with some form of the query – even if somewhat arbitrary – and release an early version of the software as soon as possible to allow people on the research community to use the tool. This has enabled us to collect feedback about the implementation in general and about the query functionality in particular, and to reiterate the structure of the query based on the feedback received. As the result, it was decided that the query functionality should be able to perform filtering based on the following properties of a problem:

- Randomized complexity.
- Deterministic complexity.
- The graph family of problems (e.g. whether the problem is on rooted or unrooted trees, whether the underlying graph is a tree or a cycle, degree of the graph if a graph is regular, etc.).

- The number of labels allowed in the solutions to the problems.
- Restrictions on what outputs can nodes produce (e.g. the produced outputs must always include 2-coloring of a graph, or all passive nodes in a graph must output the same label on all incident edges, etc.).
- The possibility to query for only the *smallest* problem that satisfies other query parameters. Notice that the *smallest* term here is defined rather ambiguously and is open for an interpretation while implementing. Roughly speaking, it means a problem with the smallest number of valid outputs.
- The possibility to query for only the *largest* problem. Here again, the *largest* term is a rather ambiguous one and means a problem with the largest number of valid outputs.
- The possibility to query for the smallest not-yet-classified problem.

Chapter 5

Implementation

This chapter will describe the implementation of the tool. In particular, I will outline the high-level architecture of the software, discuss the internal representation of LCL problems as well as the representation of the query object. Moreover, I will discuss how the batch classification has been implemented. Finally, at the end of the chapter, I will cover several miscellaneous items such as the problem normalization algorithm, integration of the Round Eliminator tool into my software, and some of the properties of the Web interface, which has also been implemented as a part of the thesis project.

5.1 The high-level architecture of the tool

This section will outline the high-level architecture of the tool and will describe in further detail some of its most important components. Despite the chapter's name, I am not going to describe low-level implementation detail in this section, but instead, try to convey a technical perspective on the tool as a whole. I will also discuss and critically analyze some of the architectural and technological decisions that have been made during the implementation phase.

On the high level, the software consists of three parts: front-end application, back-end application, and a database. The database contains arguably the most important component of this project: the definitions of LCL problems as well as their known upper and lower bounds. The back-end part contains the core logic for classifying LCL problems. Besides, it acts as a middle layer between the database and the front-end part. That is, it is responsible for receiving data from one of the components (e.g. database), transforming it to the format understandable by the other part (e.g. the front end), and finally sending it to that part. Finally, the front-end part's main

responsibility is to provide an understandable and user-friendly user interface through which one could classify individual LCL problems as well as query for the whole families of problems that satisfy specified criteria. Furthermore, the client-side part of the application is responsible for displaying the results returned by the back-end part in a concise and understandable manner. For the depiction of the high-level architecture of the implementation, refer to Figure 5.1.

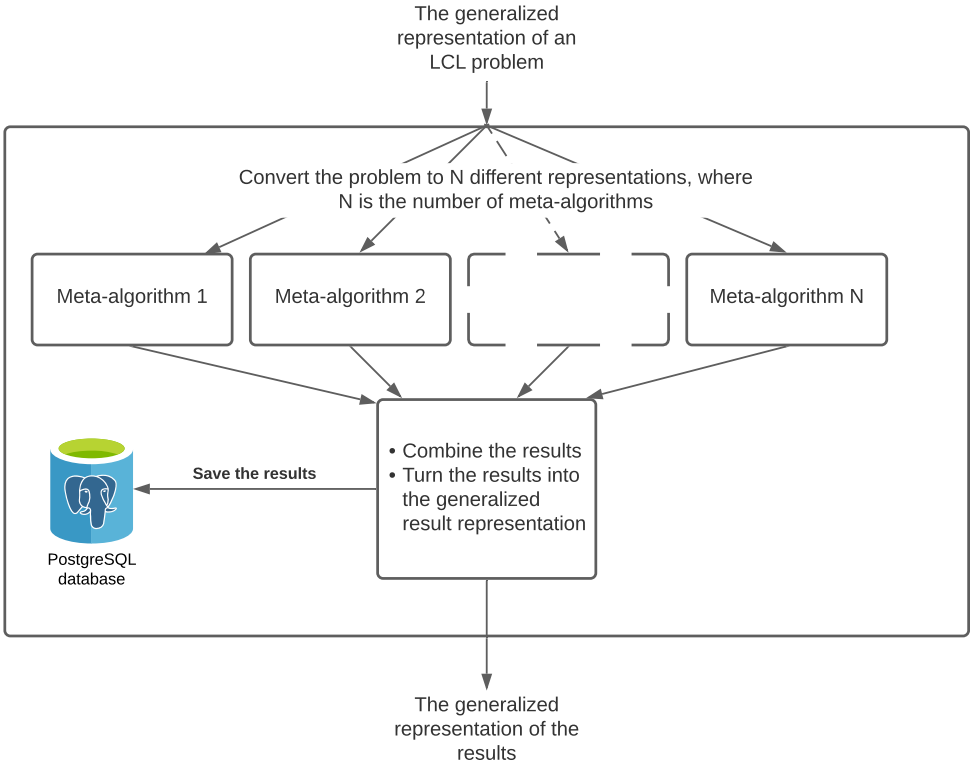


Figure 5.1: The high-level architecture of the implementation

The database contains only three tables: one for storing LCL problems together with their upper and lower bounds, one for keeping track of what families of problems have been generated and classified via batch classification mechanisms, and one storing meta-data of the meta-algorithms used in classifying the LCLs. The front end is quite simple from a technical perspective too. Some of the details about the client-side of the application will be discussed more in Section 5.5. The back end, however, contains most of the

complexity of the tool. Therefore, in this section, I will concentrate mainly on the server-side.

5.1.1 Problem

The core element of the whole application and the back-end part, in particular, is the Problem class. For representing an LCL problem, I have decided to use a notation similar to that of Round Eliminator. My representation, however, also allows for graph types other than trees – namely cycles. Besides, it allows the underlying graph of a problem to be rooted (or directed in the case of cycles). Finally, it allows representing problems where leaves' and root's output labels are also constrained. The representation similar to that of a Round Eliminator has been chosen because of its generality. In particular, as I have shown in Theorem 3.1.1, any LCL problem on (β, δ)-biregular unrooted trees – provided that nodes of degrees other than β and δ are unconstrained and nodes near leaves are unconstrained – can be represented using the formalism used in the Round Eliminator.

Since we are only concerned with problems on regular trees and cycles this representation works well for us. For the cases of cycles, the only restriction is that degree of both passive and active configurations has to be two. Also, the case of directed (rooted) cycles (trees) can be handled using the *colon notation* that has been already previously used in a recent paper on automating the classification of LCLs in rooted trees [5]. Furthermore, the cases when *irregular* nodes, i.e. leaf nodes or root node, are restricted can also be handled by specifying those restrictions (separately for leaf nodes and a root node) as separate properties of the Problem object. Thus, my representation of LCL problems is general enough to cover all the LCL problems that are included in the scope of the project.

Having explained this, I can finally list all the properties that are encapsulated into the Problem object. The object contains the already mentioned active configuration, passive configurations, leaf constraints (which can be an empty set), root constraints (which can be an empty set too), as well as special flag variables indicating whether all active or all passive configurations are allowed for active and passive nodes respectively. If one of the two flag variables is set to true, it overrides whatever is contained in the variables holding the actual active and passive configs. Besides, each problem object contains information on the type of the underlying graph: path, cycle, or tree. Notice that although any path is always a tree, I chose to differentiate between those two for convenience. It is worth mentioning that a path graph type will be chosen as an internal representation whenever both active and passive configurations are of degree two and the user-selected graph type is

not cycle. In particular, if a user specifies a tree as a graph type and provides configurations of degree two, a tree will still be chosen as a graph type of the problem internally. Finally, after parsing the configs that are always provided in a string format, the tool automatically decides whether the given problem's underlying graph is rooted/directed or not.

It is also worth noting that as the first step of creating a Problem object, the specified parameters, including nodes' configurations and constraints are checked for validity. Thus, if, for example, a problem is misspecified so that some of the configurations are "directed" while others are not, an exception is thrown containing a message with a human-readable explanation for why the specification was invalid. After the validity of the specification is checked, a problem is normalized by executing the normalization algorithm described below in Section 5.2.

5.1.2 Classification

Another central component of the back end is a function responsible for classifying LCL problems. On the high level, the function takes as input an LCL problem and runs underlying meta-algorithms, some of which might be able to provide some useful information about the complexity of the problem. After that, the function combines the results returned by all the meta-algorithms. Based on the results returned, the function determines the tightest known lower and upper bounds of the LCL problem both in the deterministic and randomized LOCAL models. Finally, a Response object is constructed based on the bounds and is returned as the function's result. In the best case, tight complexity bounds are found for both deterministic and randomized settings. In the worst case, trivial bounds are returned: *unsolvable* for upper bounds and $\Omega(1)$ for lower bounds.

Each of the underlying meta-algorithms or datasets is wrapped in a function – I will refer to it as *subclassify* from now on – such that it provides a unified interface for the main *classify* function. In particular, subclassify takes as input an instance of the Problem class and returns an instance of the Response class. A Response object simply encapsulates upper and lower bounds for both deterministic and randomized settings. There is exactly one subclassify function for each meta-algorithm or dataset that I use. Inside each subclassify function, I first check if the provided problem can be classified at all by the underlying meta-algorithm. Often, we can exit early already at this stage if, for example, the meta-algorithm in question deals only with rooted trees and the provided input is a problem on unrooted trees (or e.g. a problem on cycles). Then, if we have not exited the function yet, we transform an instance of the Problem class to a problem representation used by

the meta-algorithm that the given subclassify function wraps around. The meta-algorithm is then executed, the returned result is transformed into an instance of the Response class and the Response object is returned as a result of the subclassify function. Once the subclassify function is called for each of the meta-algorithms, the results of the calls are combined, and a single Response object is returned from the classify function.

5.1.3 Query

The query class encapsulates some of the properties based on which the database is queried for problems. Only problems that satisfy the provided Query object will be returned. The properties encapsulated by a Query object are more or less directly mapped to an SQL query's WHERE clause. The Query class groups those properties into three parent items:

- problem properties,
- complexity bounds,
- include/exclude configuration.

Problem properties define a family of problems that are to be returned from the database. It includes the following fields:

- active degree,
- passive degree,
- alphabet size (an upper bound on it to be precise),
- whether active/passive configurations are necessarily monochromatic (meaning that only such configurations are allowed where the same label is outputted on all ports by a single node, e.g. $\{A,\ A,\ A\}$, or $\{B,\ B,\ B,\ B\}$, etc. Note that different nodes can still output different labels on their ports, as long as the labels are the same on the ports of a single node. For example, some node v can output A on all its ports, while some node u can output B on all its ports.),
- whether the underlying graph is directed/rooted or not,
- the type of the underlying graph, which is always one of the following three options: tree, cycle, path.

Complexity bounds limit the returned problems further by restricting their complexities. The *complexity bounds* object contains the following items:

- an upper bound in the randomized LOCAL setting (referred to as RUB),
- a lower bound in the randomized LOCAL setting (referred to as RLB),
- an upper bound in the deterministic LOCAL setting (referred to as DUB),
- a lower bound in the deterministic LOCAL setting (referred to as DLB).

The restriction of the fetched problems is done according to the following logic: only those problems are returned (not filtered out) whose known randomized upper bound is at most as high as RUB, whose known randomized lower bound is at least as high as RLB, whose known deterministic upper bound is at most as high as DUB and whose known deterministic lower bounds is at least as high as DLB. Notice that for a problem not to be filtered out, all four criteria must be satisfied.

Finally, *include/exclude configuration* has the following items encapsulated in it:

- Whether only the smallest problem has to be returned (in terms of the number of configurations).
- Whether only the largest problem has to be returned (in terms of the number of configurations).
- Whether only those problems must be returned whose randomized upper and lower bounds do not match.
- Whether only those problems must be returned whose deterministic upper and lower bounds do not match.
- Whether only those problems must be returned whose randomized both upper and lower bounds are trivial (i.e. unsolvable and $\Omega(1)$ respectively).
- Whether only those problems must be returned whose deterministic both upper and lower bounds are trivial (i.e. unsolvable and $\Omega(1)$ respectively).

- List of configurations L such that if a problem's configurations contain at least one configuration from L, the problem is excluded. Note that the item has no effect if left empty.
- List of configurations L such that if a problem's configurations contain all of the configurations from L, the problem is excluded. Note that the item has no effect if left empty.
- List of configurations L such that if a problem's configurations contain at least one configuration from L, the problem is included. Otherwise, a problem is excluded. Note that the item has no effect if left empty.
- List of configurations L such that if a problem's configurations contain all of the configurations from L, the problem is included. Otherwise, a problem is excluded. Note that the item has no effect if left empty.

5.2 Problem normalization

This section describes the above-mentioned problem normalization algorithm. The algorithm is used for determining which problems are equivalent to each other. This, in turn, allows us to reduce the amount of needed storage. Besides, this prevents situations when a tool needs to run classification for a problem A from the very beginning while an isomorphic problem B has already been classified by the tool previously and saved in the database. Thus, the normalization algorithm also allows us to reduce the usage of computational resources.

The algorithm is based on the problem normalization algorithm from the Round Eliminator [47]. While the original algorithm is written in Rust programming language, I reimplemented it in Python and changed some of its implementation details to better suit my purpose. However, the general idea of the algorithm stays the same.

The reason why I chose the normalization algorithm in my solution to be heavily based on the algorithm from the Round Eliminator [47] is primarily due to the fact that the Round Eliminator's problem representation is very similar to the problem representation used in my solution. Furthermore, the normalization algorithm used in the Round Eliminator is known to be correct and works sufficiently fast for the Round Eliminator to be useful for the research community. While it is perhaps true that better, more efficient normalization algorithms exist, I decided to avoid optimizing prematurely and, as the first iteration, selected a working, "good enough" algorithm. If and when the current implementation of normalization will start causing

performance problems, it should be relatively easy to switch to another more efficient normalization algorithm.

First, we determine how many labels are used in the description of the problem P i.e. its alphabet size. Then, we map each letter in the description to a letter from A to Z. For example, if integers were used for describing the problem, the integers will be mapped to capital letters of the English alphabet. It is important to point out already now that one limitation of the current implementation is the fact that a problem with more than 26 labels cannot be normalized correctly. Once we obtain the list of letters used, we calculate all the permutations of the list. For each permutation, we execute the following subroutine:

1. Based on the given permutation, create a mapping from each element of the problem's alphabet to each symbol in the permutation. That is, we map the i'th letter of the P's alphabet to the i'th symbol in the permutation for all i from 1 to the size of P's alphabet.

2. Based on the obtained mapping, do the following with the problem's active configurations, passive configurations, leaf constraints, and root constraints one at a time.

   (a) For each line of the configurations/constraints, rename the symbols according to the mapping constructed above.

   (b) If the problem P is specified on an unrooted/undirected graph, change the positions of all symbols on each line of the configurations/constraints according to the symbols' alphabetical order. If the underlying graph of P is directed/rooted, keep the currently first symbol as the first (this is because it is assumed that the first configuration symbol in configurations of LCLs on directed/rooted graphs always points towards a predecessor/parent node, so its position must be preserved), but change the positions of all the other symbols (again separately for each line) according to the symbols' alphabetical order.

   (c) If at this point, some configuration lines are duplicated, remove all the duplicates so that each configuration line is unique.

   (d) Sort the unique configuration lines in alphabetical order treating the lines as strings.

3. Return the newly constructed active configurations, passive configurations, leaf constraints, and root constraints as a tuple of four elements.

Each execution of the subroutine described above yields a tuple of size four. Thus, after executing the subroutine on each permutation of the labels, we eventually obtain a list of tuples of size four. Then we sort the list and take its first element. The four elements of this first tuple are then assigned to the P's active configurations, passive configurations, leaf constraints, and root constraints, respectively.

5.3 Batch classification and reclassification

This section describes how the batch classification and batch reclassification mechanisms have been implemented. Batch classification is important since we want the database to be pre-populated with a significant number of classified problems already before the first users start using it. In particular, we would need to classify huge but finite families of problems consisting of tens and hundreds of thousands of LCL problems. Moreover, once a family of problems is selected, the problems of the family need to be generated and turned into instances of Problem class before we can classify them. Batch reclassify functionality, on the other hand, is needed for cases when, for one reason or another, we would want to reclassify a certain family of LCL problems that is already stored in the database. For example, if I integrate a new meta-algorithm into my solution in the future, I would need to reclassify all problem families that are compatible with the newly integrated meta-algorithm. Or perhaps, a programming mistake has been made when integrating one of the meta-algorithms, and therefore, the problems need to be reclassified again.

One option for implementing batch reclassification mechanisms would be to simply delete all of the problems of the class in question from the database and execute the batch classification logic from the beginning including its first step of generating the problems of the problem class. However, this is suboptimal, since the generation of problems of a certain family usually takes significantly longer times than the subsequent classification step. For this reason, I decided to have separate functionality for batch reclassification.

The batch classification process starts with creating an object that specifies a family of problems to be generated, classified, and stored in the database. The specification includes properties like active and passive degrees, label count, type of the underlying graph, etc. The problems of the given family are then generated via the following algorithm:

1. Based on the specified label count, get a list of all symbols i.e. the alphabet of the problem family.

2. Based on the alphabet and the specified active/passive degrees, generate lists of all the possible active and passive one-line configurations.

3. Get a list of all possible combinations of active one-line configurations and a list of all possible combinations of passive one-line configurations. The combinations can be of any size. This can be accomplished via the *powerset* function. Now we have lists of all possible active and passive configurations.

4. Remove empty configurations.

5. Generate all possible tuples of size two where the first element is an active configuration and the second is a passive configuration. This corresponds to the Cartesian product between the previously generated lists of all active and passive configurations.

6. Remove tuples that have exact duplicates in the generated list of tuples, so that all tuples are unique.

7. Turn each tuple into a Problem object using elements of the tuple as active and passive configurations.

8. Normalize all the problems.

9. Now that all the problems are in a normalized representation, it is easy to detect whether any two problems are isomorphic to each other. Remove such problems from the list so that the list contains unique non-isomorphic problems only.

10. Return the list of the newly-generated problems.

Once the problems are generated, they are inserted into the database, even though they are not classified yet. This is done mainly because the database will automatically associate a unique identifier with each of the problems.

After that, the problems are classified. Notice that some of the meta-algorithms provide a way to classify lists of problems at the same time, which is often significantly faster than classifying one problem at a time. This is due to the fact that some meta-algorithms (e.g. TLP classifier and BRT classifier) are huge datasets of pre-classified problems. When classifying each problem individually, such meta-algorithms would need to read out the whole dataset from the disk, find one specific problem, and return the result. On the other hand, when classifying multiple problems at the same time, numerous

optimizations can be applied e.g. reading the dataset from the disk just once and search for all the requested problems at the same time. For this reason, when batch classifying families of problems, the program first calls such batch classification methods of meta-algorithm if such methods exist. Once this is done, a modification of the previously-described classify function is called for each individual Problem object. The difference is that if a meta-algorithm A's batch classify method has been already called, its subclassify function is not called anymore during the individual-problem classification which can significantly speed up the whole batch classification functionality. Finally, when all problems have been classified, the database is updated with the classification results.

The batch reclassification functionality resembles what has been described above with the exception that, once a family of problems has been specified, it is not generated but is instead queries from the database. After that, the fetched problems undergo the same batch classification process. Finally, the database is updated with the received classifications.

5.4 Integration of the Round Eliminator

This section describes how the Round Eliminator tool [47] has been integrated into my solution. Before the integration, I had considered several options:

- Using an API of a server that runs the back end of the Round Eliminator tool.
- Calling the Round Eliminator as a command-line tool from my Python application and then parsing the output, which is returned in a text format via stdout [1].
- Compiling the Round Eliminator's source code to a "Shared Object" file (.so extension) using CPython [63] bindings and then importing the compiled functions to my Python application as CPython dependency.

At the moment of writing the thesis, the Round Eliminator tool is deployed using WASM [51]. This virtually means that there is no back-end part running on a server, but instead, all of the logic is executed on the client-side of the application. Redeploying the tool with a standalone server and a separate client part did not seem like a feasible alternative since this would require a nontrivial amount of work from the tool's maintainers. Thus, the first option was not feasible.

On the other hand, the second option was feasible. Indeed, this method of using the Round Eliminator had already been used in the implementation of TLP Classifier [50]. However, this approach has several crucial disadvantages. Firstly, the overhead of calling a command-line command from a Python program is significant compared to just calling a Python function. Secondly, the output of the Round Eliminator would have to be parsed from the plain text format, which would have added additional complexity to the software. Finally, the approach complicates the distribution and deployment process of the software. Indeed, one would have to install the specific version of Rust programming language as well as download and build from source codes the specific version of the Round Eliminator required.

Instead, I have decided to go with the latter option. This alternative does not have any of the disadvantages listed in the previous paragraph while being just as feasible and easy to implement. Compiled in this manner, the whole of the Round Eliminator logic is contained in a single .so file that can be easily deployed to e.g. a production server. Besides, the speed improvement is significant: I was able to execute up to three rounds of the Round Eliminator in under 100 milliseconds. Finally, the functions exposed via the .so file, return python-native data structures so that no additional parsing is required on my side. This, in turn, has been achieved by adding CPython bindings to the Round Eliminator source code. The bindings wrap the relevant functions written in Rust, thus enabling us to pass Python data structures as parameters and receive the return values as python-interpretable values. The bindings were implemented using the rust-cpython library [27].

5.5 Web interface

This section will briefly discuss some implementation details of the client-side of the application. First, I start by explaining why the web interface has been implemented in the first place and why it is important. Then, I describe some of the technologies that were chosen as part of the implementation. Finally, I describe the reasoning behind and advantages of the forms' state being stored as a query part of the URL.

The reason why I decided to spend additional time and implement the web interface can be best understood when considering another tool that has recently been rising in popularity among the distributed algorithms community – the Round Eliminator [47]. The tool has proved to be a very useful utility when doing research connected to LCL problems on biregular trees. However, if it was not for the web interface that Olivetti has implemented, it is most likely that the popularity and frequency of use of the tool would have

been nowhere near the current levels. Indeed, almost all users of the tool use it via the web client. Otherwise, a user would need to install Rust programming language [53], build the tool locally using the tool called Cargo [54] and then run it using a command line. It is clear that the number of people willing to do that would be significantly smaller than the number of people who ended up using an easy-to-use web interface that requires no installation or configuration.

By analogy, we can assume with high confidence that significantly more users would be willing to use my tool via the web interface rather than downloading the source of the tool locally and installing a specific version of Python programming language [63]. Moreover, the richness of the web user interface allows us to convey complicated ideas related to distributed computing in a way that is easier to understand, at least for a knowledgeable audience. Thus, it is has been decided to spend some time resources – even though they were quite limited – on implementing a web user interface that is relatively easy to use and understand compared to its command-line-based analog.

The web interface consists of two forms. One form for classifying individual problems, and another one for issuing queries about groups of problems against the database. When using the first form, the application will first check if the requested problem can be found in a database. If not, it will proceed to run the classifiers (which corresponds with the classify function described above) and will eventually return the newly classified result. If the classification result is not trivial (which means that lower bounds are constant and upper bounds are unsolvable), it will also be stored in the database. The second form, once submitted, will be transformed into the previously described *Query* object. The query will then be executed against the database, the results will be returned to the client-side and rendered in a user-friendly format.

As a programming language for the front-end application, I decided to use TypeScript [42], which is a general-purpose programming language developed and maintained by Microsoft. It is closely related to JavaScript programming language, which has become the de facto standard programming language for the Web. As a framework for the client application, I decided to use Svelte [29]. Svelte is an open-source front-end framework written in TypeScript. Among other things, it allows a simplified approach for application state management and reduces initial page load times compared to other front-end frameworks [28].

For the styling of the front end, I used a minimalist CSS framework called Milligram [48]. It provides a good starting point when it comes to styling a modern web application. Besides, the total size of the framework is just 2

kilobytes when zipped [48]. This, simplicity to use, and my previous positive experience with the framework were the decisive factors that influenced my final choice.

As a final interesting piece of technology used, I describe properties and the reason for choosing to use svelte-virtual-list [30]. The library allows rendering only a part of the content instead of rendering the whole of the content on a web page. In the given case, this is particularly useful as it allows us to show tens of thousands of LCL problems (returned as part of the described above querying functionality) in a virtual list. The virtual list, implemented via the above-mentioned svelte-virtual-list library, renders only a small number (about 3–4) LCL problems at the same time. At the same time, it allows users to scroll through the list of problems with no noticeable delay. If not for the library, the client application would have to render tens of thousands of LCL problems all at once. This would result in the user interface freezing for a prolonged period of time (up to several minutes).

Finally, I will explain the reasoning behind and benefits of storing the state of the two forms as part of the URL's query component. When searching for interesting query results, or even just classifying different individual LCL problems, it is often required to demonstrate the results to somebody else. To allow for such sharing, I decided to encode the state of both forms in the URL. Thus, once e.g. the results of a query are obtained, it is possible to copy the link and simply send it to, for example, a colleague. When opening the link, both forms will be prefilled with exactly the same values as those of the sender. This will make sure that, in most cases, the sender and the receiver of the link will end up with the same results displayed. Besides, if a particularly interesting query has been discovered, it is possible to simply copy and store the link. Opening the link in the future, will prefill the forms with the same parameters and display the same relevant results.

Chapter 6

Evaluation and further enhancements

In this chapter, I will attempt to evaluate the success of the implementation based on the research goals and the scope that I have outlined in Chapter 4. Besides, I will also evaluate the performance of the implementation. As seen from the title, I have decided to combine the evaluation with discussion on opportunities for further development. Therefore, I will also propose several ideas for how my implementation can be improved and extended.

6.1 Decidability of a wide range of problem families on trees

The first goal among the research goals listed in Chapter 4 was to store (or to be able to compute dynamically) most (if not all) of the existing knowledge on the classification of LCL problems on trees. To evaluate how well I have done, we can refer to Table 6.1. The table shows which families of LCL problems on trees, paths, and cycles are decidable and which are not. It also gives further details about the decidability of the families. For instance, it shows whether all complexity classes are decidable for a particular problem family or only some.

The first three columns under the "Paths and cycles" grouping refer to LCL problems on paths and cycles where input is not allowed. The graphs can be directed or undirected, have an arbitrary (but finite) number of labels in the output, and have restrictions on nodes of degree 1 (in cases of paths). All of these cases are covered by the cycle-path classifier [60], which is based on the recently published paper [19]. The classifier has been integrated into the solution and, thus, I claim that the problems of this family

Setting		*Paths and cycles*				*Trees*				
	unlabeled input	✓	✓	✓		✓	✓	✓	✓	
	regular	✓	✓			✓	✓	✓	✓	
	directed or rooted	✓	✓					✓		
	binary output	✓					✓			
	homogeneous [9]					✓				
Complexity classes	$O(1)$	D+R	D+R	D+R	D+R	D+R	D+R	D+R	D+R	D+R
	$\cdots$	—	—	—	—	—	—	—	—	—
	$\Theta(\log \log^* n)$	—	—	—	—	—	—	—	?	?
	$\cdots$	—	—	—	—	—	—	—	?	?
	$\Theta(\log^* n)$	—	D+R	D+R	D+R	D+R	—	D+R	D+R	D+R
	$\cdots$	—	—	—	—	—	—	—	—	—
	$\Theta(\log \log n)$	—	—	—	—	R	R	—	R	R
	$\Theta(\log^\alpha \log n)$	—	—	—	—	—	—	—	—	—
	$\cdots$	—	—	—	—	—	—	—	—	—
	$\Theta(\log n)$	—	—	—	—	D+R	D+R	D+R	D+R	D+R
	$\cdots$	—	—	—	—	—	—	—	—	—
	$\Theta(n^{1/k})$	—	—	—	—	—	—	—	?	(n)
	$\cdots$	—	—	—	—	—	—	—	—	—
	$\Theta(n)$	D+R	D+R	D+R	D+R	—	D+R	D+R	D+R	D+R
Decidability	P	✓	✓	✓	—	?	(D)	?	?	—
	PSPACE	✓	✓	✓	?	?	(D)	?	?	—
	EXPTIME	✓	✓	✓	?	?	(D)	✓	?	?
	decidable	✓	✓	✓	✓	?	(D)	✓	(H)	(H)
References		[4, 13, 46]	[13, 46]	[19]	[3]	[9]	[4]	[5]	[16, 18]	[16, 18]

Legend
- ✓ = yes, ? = unknown, — = not possible, $\alpha > 1$, $k = 2, 3, \ldots$.
- D = class exists for deterministic algorithms, R = class exists for randomized algorithms.
- (n) = the current construction assumes the knowledge of n; unknown without this information.
- (D) = known only for deterministic complexities, unknown for randomized.
- (H) = known only for classes between $\Omega(\log n)$ and $O(n)$.

Table 6.1: An overview of the decidability of LCL problems on paths, cycles, and trees[1]. The decidability is given assuming P ≠ PSPACE ≠ EXPTIME. Apart from the references given in the table for each of the columns, the works that contributed to the discovery of the presented knowledge include [6–8, 12, 17, 20, 38, 45]

[1] The table was copied from the manuscript by Balliu et al. [5] with minor modifications with the permission of the authors.

are fully classifiable by the tool. The fourth column of the table refers to LCL problems on paths and cycles where input labeling is allowed. Since none of the used classifiers are capable of processing problems with inputs, this family of problems has not been covered by my implementation, although it has been shown that LCLs of this class are decidable (even though it is PSPACE-hard) [3].

As can be seen from the first column of the "Tree" grouping, it is unknown whether unlabeled regular homogeneous [9] undirected trees are classifiable if the number of output labels exceeds two. On the other hand, LCL problems on unlabeled regular undirected trees with binary label output are fully decidable, albeit only in the deterministic setting. In addition to this, for some problems of the problem family, complexity is decidable even in the randomized setting [4]. Since my solution does not directly include a classifier that would implement the classification of problems of the described family, the tool is not capable of classifying the problems. It is important to notice however that the TLP Classifier [50] uses the classifier as part of its implementation. Therefore, at least some problems of the described family can be classified. Specifically, only problems on (2, 2)-, (2, 3)-, and (3, 2)-biregular trees of the family (and only in deterministic setting) can be classified in my implementation. Therefore, a natural opportunity for extending the current functionality of the tool would be to integrate a classifier that would classify all binary labeling LCL problems on unlabeled regular undirected trees. Given that such implementation would essentially involve a simple table lookup (the authors of the original paper reduce the classification problem to a two-dimensional table lookup), the classifier could work efficiently even with trees of a relatively high degree.

The next column in the table is for LCL problems on unlabeled regular directed trees. It has been shown that the problems of this family are fully decidable with any finite number of output labels [5]. However, the only currently-existing implementation of the decidability algorithm works exclusively with binary rooted trees [58]. The meta-algorithm has been successfully integrated into the solution, and thus all LCL problems on binary rooted trees can be classified also in practice using the tool. Nevertheless, an implementation of the decidability algorithm that would work for trees of higher degrees would be of high interest to the research community. The original paper shows that such implementation for an arbitrary number of output labels is possible at least in principle. Besides, there is no evidence that such implementation would be impractical from the performance perspective. Therefore, a natural opportunity for extending the software would be to build such an implementation and consequently integrate it into the tool.

All other – more general – LCL problem families are currently only known to be decidable between complexity classes of $\Omega(\log n)$ and $O(n)$ [16]. There is currently no known implementation of the classification algorithm described in the referenced paper. Although the described meta-algorithm is at least EXPTIME-hard, there is also no evidence that the algorithm would not be practical at least for a limited subfamily of problems. Therefore, its implementation and integration into my solution would be a potential direction for the tool's future development.

Thus, while many of the listed graph families are already classifiable with the tool, more meta-algorithms can be developed and integrated. Moreover, the theoretical research of decidability of LCL problems on trees is currently developing at a high pace, which will likely bring even more positive theoretical decidability results in the future. It is then the task of the practitioners to catch up with the theory and make the practical decidability landscape as complete as possible.

6.2 Finding a problem's complexity and performing group queries

The two other goals in the list of the research goals were

1. to be able to find deterministic and randomized complexity of a particular problem P, provided that the problem belongs to one of the classes that the tool is capable of classifying,

2. to be able to query for groups of multiple problems based on a variety of parameters.

As can be seen from Chapter 5, both of these functionalities have been successfully implemented. Notice that I address performance issues in another section later in this chapter. Also, I have already evaluated the number of problem families the tool is capable of working with. In this section, we are discussing just the availability of the functionality itself, while not being concerned with other, although related, metrics.

As a final research goal in the list, we had a possibility to see which of the meta-algorithms has determined each of the lower and upper bounds for each of the pre-classified LCL problems being stored in the database. As described in the previous chapter, this is automatically done for all problems being classified via my solution. At the same time as a problem is classified, all of its classification results — deterministic upper bound, deterministic lower

bound, randomized upper bound, randomized lower bound – are associated with one of the meta-algorithms. This information is also persisted in the database at the same time a newly classified problem is being stored. Thus, we can consider this research goal as being fulfilled as well.

6.3 Performance

In this section, we will evaluate my implementation from a performance perspective. Performance is essential for the tool to be useful for the research community. We will evaluate the performance of the tool in the context of classifying a single LCL problem and querying the database for a group of problems matching user-defined specifications.

Notice that the purpose of this section is to evaluate how useable the tool is from the perspective of its users. Therefore, we are not concerned with the exact millisecond precision of the operations. Instead, I will present the performance results with precision down to a tenth of a second. Notice also that the times will vary depending on the underlying hardware, the fact whether other users are using the tool at the same time, etc. All the results presented below are performed on the author's local machine (MacBook Pro, 13-inch, 2018, Four Thunderbolt 3 ports). Refer to the technical specification on the Internet for the exact hardware specifications of the machine.

Finally, when presenting results for classifying a problem or performing a query, I measure the time it takes for the corresponding function to execute as defined in the back-end part of the implementation. Therefore, the time as experienced by the users of the Web interface might be somewhat longer since it would include also a network delay and executing front-end logic.

6.3.1 Classifying a problem

If a requested problem already exists in the database, finding out the problem's complexity consistently takes less than a second. On the other hand, if a problem has to be classified dynamically, classifying it will take less than 3.2 seconds (at least in cases where problems' degrees are less than 5 and the number of labels allowed is less than 5 as well). Below are some of the performance measurement results, based on which the above-mentioned conclusions have been drawn.

- Any 2-labeling problem on undirected paths is classified in *under 0.1 sec.*

- Any 3-labeling problem on directed (3, 2)-biregular trees is classified in *under 0.7 sec.*
- Any 3-labeling problem on undirected (3, 2)-biregular trees where passive configurations are monochromatic is classified in *under 3.2 sec.*

6.3.2 Querying a problem family

Below is a similar list of performance results for the querying functionality. The query that retrieves the largest problem class, in terms of the number of problems, takes under 20 seconds. Notice that this query finds and fetches several tens of thousands of problems. Queries that fetch only a couple of hundred problems are executed in under 1.5 seconds.

- All 2-labeling problems on undirected paths are fetched in *under 0.2 sec.*
- All 3-labeling problems on directed paths are fetched in *under 7.7 sec.*
- All 3-labeling problems on directed (3, 2)-biregular trees are fetched in *under 20 sec.*
- All 3-labeling problems on directed (3, 2)-biregular trees where passive configurations are monochromatic are fetched in *under 0.3 sec.*
- Any 3-labeling problem on undirected (3, 2)-biregular trees where passive configurations are monochromatic are fetched in *under 3.2 sec.*
- Any 3-labeling problem on undirected (3, 2)-biregular trees where active configurations are monochromatic are fetched in *under 1.1 sec.*

Based on all the obtained performance results, as well as based on the feedback that I have received from the members of the research community who already used the tool, it is evident that the performance is fast enough for the tool to be useful for the kinds of tasks it is intended for. On the other hand, waiting for more than 20 seconds might prove to be impractical or at the very least inconvenient. Therefore, improving the performance – especially the performance of the querying functionality – is a natural improvement direction that would most likely provide a lot of value for the users of the solution.

6.4 The number of pre-classified problems

At the moment of writing the thesis, the database contains 335 314 LCL problems. Out of them, 197 218 have at least one non-trivial bound. *Trivial bound* here refers to $\Omega(1)$ for lower bounds and *unsolvable* for upper bounds.

The following families of LCL groups have been pre-generated and batch classified using the batch classification functionality described in Chapter 5:

- binary labeling problems on both directed and undirected paths and cycles,
- ternary labeling problems on both directed and undirected paths,
- 4-labeling problems on undirected paths,
- binary labeling problems on both rooted and unrooted (3, 2)-biregular trees,
- ternary labeling problems on unrooted (3, 2)-biregular trees,
- ternary labeling problems on rooted (3, 2)-biregular trees where passive configurations are monochromatic,
- 4-labeling problems on rooted (3, 2)-biregular trees where active configurations are monochromatic,
- a part of 4-labeling problems on directed paths,
- a part of ternary labeling problems on rooted (3, 2)-biregular trees (with no constraints on active or passive configurations),
- a part of 4-labeling problems on rooted (3, 2)-biregular trees where passive configurations are monochromatic.

Notice that in all of the problem families listed above, leaf and root nodes (or equivalently nodes of degree one on paths) are unconstrained. Moreover, in addition to the groups of pre-classified problems, the database contains several individual problems that were added as a result of the application being used by several members of the distributed algorithms research community.

Thus, it is evident that the number of pre-classified problems is already at this stage sufficient for the tool to be useful to the research community. On the other hand, all of the problems currently stored in the database are such that the number of the allowed labels is limited to under 5, and the degrees of the nodes are limited to under 4. This is a huge limitation, and it would

be highly useful to investigate ways in which it can be mitigated. Therefore, populating the database with more problems of more diverse problem families is a natural opportunity for future development.

6.5 Miscellaneous

This section contains some of the possibilities for improving and extending the solution. The ideas collected here have been obtained as a result of the feedback received from members of the distributed algorithms research community who had already used the system as part of their research process.

Currently, it is only possible to specify an upper bound for the number of labels used in a problem when constructing a query. A common request was to also be able to specify a lower bound for the number of labels.

Another request was the ability to query only for those problems that are or lead to *periodic points* as a result of applying round elimination. We call a problem P a periodic point if we can obtain the same problem P by applying round elimination to it one or more times. The information about whether a problem is a periodic point or not is currently only available at the time of classification but is not stored in the database. Storing the data and allowing for querying based on it would be yet another possibility to expand the tool.

Similarly, it would be useful to query for problems based on their exact constant complexity (as opposed to asymptotics) if such information is available. Besides, displaying precise constant complexity would also be beneficial when classifying a single problem. Currently, precise constant complexity is sometimes available when classifying a problem with the Round Eliminator. Storing this information to the database, showing it to users, and incorporating it in the Query object would be another useful extension to the tool.

Finally, the query functionality is still currently rather limited. Moreover, it is not clear which features of it are going to be the most useful to the users and which will not be used often. Thus, investigating this by collecting feedback from the research community and then modifying the tool based on this feedback is yet another opportunity for improving the software.

Chapter 7

Conclusions

The study of distributed algorithms has become one of the central topics in theoretical computer science over the last twenty years. Many significant results in the area have been achieved, especially when it comes to topics related to computational complexity. Some of these results try to move the field into a direction where deciding the computational complexity of a distributed problem becomes increasingly automated. A long-term goal behind this is to show theoretically – and often practically via an implementation – that deciding the complexity of a certain class of problems can be done in a mechanical manner relying on systematic algorithms instead of ad-hoc proof techniques by humans.

In this thesis, I have shown that, although the work on decidability of LCL problems has seen a surge over the last several years in terms of research output, the results failed to adopt a unified framework or formalism. This prevents the research community from utilizing the results in their everyday work, even though practical implementations for many of the theoretical results have been made available. Therefore, as the goal of this project, I envisioned a solution that would unify the individual implementations behind a common interface. Moreover, to make the system more useful to the research community, I had decided to implement a Web interface for the tool.

Although I have not reached the rather ambitious initially formulated research goal of encapsulating "most – if not all – of so far existing knowledge on the complexity of LCL problems on trees", I did succeed in creating a useful tool that unifies most of the existing implementations that provide some possibility of computer-aided classification of LCL problems. Moreover, the tool has been designed and implemented in such a way that it can be easily extended with more meta-algorithms and other decidability implementations if such were to appear. Some of the details of the implementation are described in Chapter 5.

Finally, the groundwork done will allow for numerous extensions and improvements to the tool. I described some of the opportunities for future development in Chapter 6. Apart from integrating more meta-algorithms and problem datasets into the solution, a lot can be improved regarding what data is stored and displayed to the user (providing the information about the exact constant complexities of LCL problems is just one example of this). Besides, possibilities related to populating the database with more families of pre-classified problems are virtually limitless. Furthermore, getting feedback from the research community and developing the query functionality to become even more relevant is yet another way to continue the current project.

Bibliography

[1] Apple. Mac OS X Manual Page For stdio(3). Documentation Archive. `https://developer.apple.com/library/archive/documentation/System/Conceptual/ManPages_iPhoneOS/man3/stdio.3.html`. Accessed 12 April 2021.

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

[3] Balliu, A., Brandt, S., Chang, Y.-J., Olivetti, D., Rabie, M., and Suomela, J. The distributed complexity of locally checkable problems on paths is decidable. In *Proc. 38th ACM Symposium on Principles of Distributed Computing (PODC)* (2019), ACM Press, pp. 262–271.

[4] Balliu, A., Brandt, S., Efron, Y., Hirvonen, J., Maus, Y., Olivetti, D., and Suomela, J. Classification of distributed binary labeling problems. In *Proc. 34th International Symposium on Distributed Computing (DISC)* (2020), vol. 179 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp. 17:1–17:17.

[5] Balliu, A., Brandt, S., Olivetti, D., Studený, J., Suomela, J., and Tereshchenko, A. Locally checkable problems in rooted trees. arXiv submission, 18 February 2021. `https://arxiv.org/abs/2102.09277`. Accessed 14 April 2021.

[6] Balliu, A., Brandt, S., Olivetti, D., and Suomela, J. Almost global problems in the LOCAL model. In *Proc. 32nd International Symposium on Distributed Computing (DISC)* (2018), U. Schmid and J. Widder, Eds., vol. 121 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 9:1–9:16.

[7] Balliu, A., Brandt, S., Olivetti, D., and Suomela, J. How much does randomness help with locally checkable problems? In *Proc. 39th ACM Symposium on Principles of Distributed Computing (PODC)* (2020), ACM Press, pp. 299–308.

[8] Balliu, A., Hirvonen, J., Korhonen, J. H., Lempiäinen, T., Olivetti, D., and Suomela, J. New classes of distributed time complexity. In *Proc. 50th ACM Symposium on Theory of Computing (STOC)* (2018), ACM Press, pp. 1307–1318.

[9] Balliu, A., Hirvonen, J., Olivetti, D., and Suomela, J. Hardness of minimal symmetry breaking in distributed computing. In *Proc. 38th ACM Symposium on Principles of Distributed Computing (PODC)* (2019), ACM Press, pp. 369–378.

[10] Bloem, R., Braud-Santoni, N., and Jacobs, S. Synthesis of self-stabilising and Byzantine-resilient distributed systems. In *Computer Aided Verification (CAV). Lecture Notes in Computer Science (LNCS)* (2016), vol. 9779, Springer, pp. 157–176.

[11] Brandt, S. An automatic speedup theorem for distributed problems. In *Proc. 38th ACM Symposium on Principles of Distributed Computing (PODC)* (2019), ACM Press, pp. 379–388.

[12] Brandt, S., Fischer, O., Hirvonen, J., Keller, B., Lempiäinen, T., Rybicki, J., Suomela, J., and Uitto, J. A lower bound for the distributed Lovász local lemma. In *Proc. 48th ACM Symposium on Theory of Computing (STOC)* (2016), ACM Press, pp. 479–488.

[13] Brandt, S., Hirvonen, J., Korhonen, J. H., Lempiäinen, T., Östergård, P. R. J., Purcell, C., Rybicki, J., Suomela, J., and Uznański, P. LCL problems on grids. In *Proc. 36th ACM Symposium on Principles of Distributed Computing (PODC)* (2017), ACM Press, pp. 101–110.

[14] Campilho-Gomes, M., Tavares, R., and Goes, J. Automatic flat-level circuit generation with genetic algorithms. In *Technological Innovation for Life Improvement* (2020), L. M. Camarinha-Matos, N. Farhadi, F. Lopes, and H. Pereira, Eds., Springer, pp. 101–108.

[15] Capizzi, G., Sciuto, G. L., Napoli, C., Tramontana, E., and Wozniak, M. Automatic classification of fruit defects based on co-

occurrence matrix and neural networks. In *Proc. Federated Conference on Computer Science and Information Systems (FedCSIS)* (2015), IEEE, pp. 861–867.

[16] Chang, Y. The complexity landscape of distributed locally checkable problems on trees. In *Proc. 34th International Symposium on Distributed Computing (DISC)* (2020), vol. 179 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp. 18:1–18:17.

[17] Chang, Y., Kopelowitz, T., and Pettie, S. An exponential separation between randomized and deterministic complexity in the LOCAL model. *SIAM J. Comput. 48*, 1 (2019), 122–143.

[18] Chang, Y., and Pettie, S. A time hierarchy theorem for the LOCAL model. *SIAM J. Comput. 48*, 1 (2019), 33–69.

[19] Chang, Y.-J., Studený, J., and Suomela, J. Distributed graph problems through an automata-theoretic lens. In *Proc. 28th International Colloquium on Structural Information and Communication Complexity (SIROCCO)* (2021), Lecture Notes in Computer Science (LNCS), Springer.

[20] Cole, R., and Vishkin, U. Deterministic coin tossing with applications to optimal parallel list ranking. *Inf. Control. 70*, 1 (1986), 32–53.

[21] Colonna, J., Peet, T., Ferreira, C. A., Jorge, A. M., Gomes, E. F., and Gama, J. Automatic classification of anuran sounds using convolutional neural networks. In *Proc. 9th International C* Conference on Computer Science & Software Engineering* (2016), ACM Press, pp. 73–78.

[22] Dolev, D., Heljanko, K., Järvisalo, M., Korhonen, J. H., Lenzen, C., Rybicki, J., Suomela, J., and Wieringa, S. Synchronous counting and computational algorithm design. *Journal of Computer and System Sciences 82* (March 2016), 310–332.

[23] Fathiyeh, F., and Borzoo, B. SMT-based synthesis of distributed self-stabilizing systems. *ACM Transactions on Autonomous and Adaptive Systems 10* (October 2015), 1–26.

[24] Fischer, M., and Ghaffari, M. Sublogarithmic distributed algorithms for Lovász local lemma, and the complexity hierarchy. In *Proc. 31st International Symposium on Distributed Computing (DISC)* (2017),

vol. 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp. 18:1–18:16.

[25] Frate, F. D., Pacifici, F., Schiavon, G., and Solimini, C. Use of neural networks for automatic classification from high-resolution images. *IEEE Transactions on Geoscience and Remote Sensing 45* (April 2007), 800–809.

[26] Fulton, C. T., Pruess, S., and Xie, Y. The automatic classification of Sturm-Liouville problems. *J. Appl. Math. Comput 124* (2005), 149–186.

[27] Grunwald, D. rust-cpython. GitHub repository. Rust to CPython bindings. `https://github.com/dgrunwald/rust-cpython`. Accessed 2 April 2021.

[28] halfnelson. Will it scale? - Finding Svelte's inflection point. GitHub repository, 24 August 2020. `https://github.com/halfnelson/svelte-it-will-scale/blob/master/README.md`. Accessed 5 April 2021.

[29] Harris, R. Svelte. webpage. `https://svelte.dev`. Accessed 21 March 2021.

[30] Harris, R., Wright, J., and Keller, M. A virtual list component for Svelte apps. GitHub repository. `https://github.com/sveltejs/svelte-virtual-list`. Accessed 7 April 2021.

[31] Hirvonen, J., Rybicki, J., Schmid, S., and Suomela, J. Large cuts with local algorithms on triangle-free graphs. *Electronic Journal of Combinatorics 24* (October 2017), P4.21–P4.21.

[32] Hirvonen, J., and Suomela, J. Distributed Algorithms 2020. webpage, 28 December 2020. Free online textbook. `https://jukkasuomela.fi/da2020/`. Accessed 11 April 2021.

[33] Hirvonen, J., and Suomela, J. Distributed Algorithms 2020: lecture 1a · Introduction. YouTube video, 30 August 2020. `https://www.youtube.com/watch?v=0rc2hq2gcZo`. Accessed 15 April 2021.

[34] Ibrahim, A. K., Zhuang, H., Chérubin, L. M., Schärer-Umpierre, M. T., and Erdol, N. Automatic classification of grouper species by their sounds using deep neural networks. *The Journal of the Acoustical Society of America 144* (2018), EL196–EL202.

[35] Khan, M. H., Thapliyal, H., and Munoz-Coreas, E. Automatic synthesis of quaternary quantum circuits. *Journal of Supercomputing 73* (May 2017), 1733–1759.

[36] Klinkhamer, A. On the limits and practice of automatically designing self-stabilization. *Michigan Technological University. Dissertations, Master's Theses and Master's Reports* (2016). `https://digitalcommons.mtu.edu/etdr/90`. Accessed 24 March 2021.

[37] Linial, N. Distributive graph algorithms global solutions from local data. In *Proc. 28th Annual Symposium on Foundations of Computer Science (SFCD)* (1987), IEEE, pp. 331–335.

[38] Linial, N. Locality in distributed graph algorithms. *SIAM J. Comput. 21*, 1 (1992), 193–201.

[39] Lynch, N. *Distributed algorithms*. Elsevier, 1996.

[40] Margenstern, M. Frontier between decidability and undecidability: A survey. *Theoretical Computer Science 231* (January 2000), 217–251.

[41] Meuli, G., Soeken, M., and Micheli, G. D. Sat-based {CNOT, T} quantum circuit synthesis. In *Proc. 10th International Conference on Reversible Computation* (2018), J. Kari and I. Ulidowski, Eds., vol. 11106 of *Lecture Notes in Computer Science (LNCS)*, Springer, pp. 175–188.

[42] Microsoft. TypeScript: Typed JavaScript at Any Scale. Documentation. `https://www.typescriptlang.org/docs/`. Accessed 11 April 2021.

[43] Miller, D. M., Maslov, D., and Dueck, G. W. A transformation based algorithm for reversible logic synthesis. In *Proc. Design Automation Conference* (2003), IEEE, pp. 318–323.

[44] Mrazek, V., Hanif, M. A., Vasicek, Z., Sekanina, L., and Shafique, M. autoax: An automatic design space exploration and circuit building methodology utilizing libraries of approximate components. In *Proc. 56th Annual Design Automation Conference (DAC)* (2019), ACM Press, pp. 1–6.

[45] Naor, M. A lower bound on probabilistic algorithms for distributive ring coloring. *SIAM J. Discret. Math. 4*, 3 (1991), 409–412.

[46] Naor, M., and Stockmeyer, L. J. What can be computed locally? *SIAM J. Comput. 24*, 6 (1995), 1259–1277.

[47] Olivetti, D. Round Eliminator: a tool for automatic speedup simulation. GitHub repository, 2020. `https://github.com/olidennis/round-eliminator`. Accessed 13 March 2021.

[48] Patoilo, C. Milligram - A minimalist CSS framework. webpage. `https://milligram.io/`. Accessed 2 April 2021.

[49] Rocher, T. Classification of distributed ternary labeling problems. GitHub repository, 2020. `https://github.com/trocher/tlpDoc`. Accessed 14 April 2021.

[50] Rocher, T. tlpClassifier. GitHub repository, 2020. `https://github.com/trocher/tlpClassifier`. Accessed 10 April 2021.

[51] Rossberg, A. WebAssembly specification. webpage, April 2021. `https://webassembly.github.io/spec/core/`. Accessed 14 April 2021.

[52] Rozhoň, V., and Ghaffari, M. Polylogarithmic-time deterministic network decomposition and distributed derandomization. In *Proc. 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC)* (2020), ACM Press, pp. 350–363.

[53] Rust Team. Rust programming language. webpage. `https://www.rust-lang.org/`. Accessed 14 April 2021.

[54] Rust Team. The Cargo Book. Documentation. `https://doc.rust-lang.org/cargo/`. Accessed 14 April 2021.

[55] Rybicki, J., and Suomela, J. Exact bounds for distributed graph colouring. In *Proc. 22nd International Colloquium on Structural Information and Communication Complexity (SIROCCO)* (2015), vol. 9439, Springer, pp. 46–60.

[56] Sharma, H., Zerbe, N., Klempert, I., Hellwich, O., and Hufnagl, P. Deep convolutional neural networks for automatic classification of gastric carcinoma using whole slide images in digital histopathology. *Computerized Medical Imaging and Graphics 61* (November 2017), 2–13.

[57] Sipser, M. *Introduction to the Theory of Computation*, 3rd ed. Cengage Learning, 27 June 2012.

[58] Studený, J., and Tereshchenko, A. Rooted tree classifier. GitHub repository, 2021. `https://github.com/jendas1/rooted-tree-classifier`. Accessed 19 March 2021.

[59] Suomela, J. Landscape of locality. webpage, 23 June 2020. SWAT 2020 keynote talk. `https://jukkasuomela.fi/landscape-of-locality/`. Accessed 12 April 2021.

[60] Tereshchenko, A. Cyclepath classifier. GitHub repository, 2020. `https://github.com/AleksTeresh/cyclepath-classifier`. Accessed 14 April 2021.

[61] Tereshchenko, A. Tree classifications. GitHub repository, 2020. `https://github.com/AleksTeresh/tree-classifications`. Accessed 14 April 2021.

[62] Turing, A. M. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Mathematical Society vol. s2-42* (January 1937), 230–265.

[63] van Rossum, G., Peterson, B., Brandl, G., Drake, F., Stinner, V., Hettinger, R., and Storchaka, S. The Python programming language. GitHub repository.

[64] Wattenhofer, R. Principles of Distributed Computing. webpage, 2016. Course at ETH Zürich (FS 2016), Lecture material, Chapter 0 Introduction. `https://disco.ethz.ch/courses/fs16/podc/lecture/chapter0.pdf`. Accessed 14 April 2021.

[65] Winkler, J., and Vogelsang, A. Automatic classification of requirements based on convolutional neural networks. In *Proc. 24th International Requirements Engineering Conference Workshops (REW)* (2017), IEEE, pp. 39–45.

[66] Zettl, A. *Sturm-Liouville theory*. American Mathematical Soc., 2010.