Distributed Algorithms 2020

Juho Hirvonen and Jukka Suomela
Aalto University, Finland

https://jukkasuomela.fi/da2020/
November 9, 2025

Contents

Foreword **viii**

About the Course viii
Acknowledgments ix
Website ix
License ix

Part I Informal Introduction

1 Warm-Up **2**

1.1 Running Example: Coloring Paths 2
1.2 Challenges of Distributed Algorithms 3
1.3 Coloring with Unique Identifiers 4
1.4 Faster Coloring with Unique Identifiers 7
1.4.1 Algorithm Overview 8
1.4.2 Algorithm for One Step 8
1.4.3 An Example 9
1.4.4 Correctness 11
1.4.5 Iteration 11
1.5 Coloring with Randomized Algorithms 12
1.5.1 Algorithm 12
1.5.2 Analysis 12
1.5.3 With High Probability 13
1.6 Summary 13
1.7 Quiz 14
1.8 Exercises 14
1.9 Bibliographic Notes 16

1.10 Appendix: Mathematical Preliminaries 16
1.10.1 Power Tower 16
1.10.2 Iterated Logarithm 17

Part II Graphs

2 Graph-Theoretic Foundations 19

2.1 Terminology 19
2.1.1 Adjacency 19
2.1.2 Subgraphs 20
2.1.3 Walks 20
2.1.4 Connectivity and Distances 22
2.1.5 Isomorphism 24
2.2 Packing and Covering 24
2.3 Labelings and Partitions 27
2.4 Factors and Factorizations 29
2.5 Approximations 31
2.6 Directed Graphs and Orientations 31
2.7 Quiz 32
2.8 Exercises 33
2.9 Bibliographic Notes 35

Part III Models of Computing

3 PN Model: Port Numbering 37

3.1 Introduction 37
3.2 Port-Numbered Network 39
3.2.1 Terminology 40
3.2.2 Underlying Graph 40
3.2.3 Encoding Input and Output 41
3.2.4 Distributed Graph Problems 42

3.3 Distributed Algorithms in the PN model 43
3.3.1 State Machine 43
3.3.2 Execution 44
3.3.3 Solving Graph Problems 45
3.4 Example: Coloring Paths 46
3.5 Example: Bipartite Maximal Matching 48
3.5.1 Algorithm 48
3.5.2 Analysis 48
3.6 Example: Vertex Covers 52
3.6.1 Virtual 2-Colored Network 52
3.6.2 Simulation of the Virtual Network 54
3.6.3 Algorithm 54
3.6.4 Analysis 55
3.7 Quiz 58
3.8 Exercises 58
3.9 Bibliographic Notes 60

4 LOCAL Model: Unique Identifiers 61

4.1 Definitions 61
4.2 Gathering Everything 63
4.3 Solving Everything 65
4.4 Focus on Computational Complexity 66
4.5 Greedy Color Reduction 67
4.5.1 Algorithm 67
4.5.2 Analysis 69
4.5.3 Remarks 70
4.6 Efficient $(\Delta+1)$-coloring 70
4.7 Additive-Group Coloring 71
4.7.1 Algorithm 71
4.7.2 Correctness 72
4.7.3 Running Time 73

4.8 Fast $O(\Delta^2)$-coloring . . . 75
4.8.1 Cover-Free Set Families . . . 75
4.8.2 Constructing Cover-Free Set Families . . . 76
4.8.3 Efficient Color Reduction . . . 78
4.8.4 Iterated Color Reduction . . . 79
4.8.5 Final Color Reduction Step . . . 80
4.9 Putting Things Together . . . 82
4.10 Quiz . . . 83
4.11 Exercises . . . 83
4.12 Bibliographic Notes . . . 86
4.13 Appendix: Finite Fields . . . 87

5 CONGEST Model: Bandwidth Limitations 89

5.1 Definitions . . . 89
5.2 Examples . . . 90
5.3 All-Pairs Shortest Path Problem . . . 91
5.4 Single-Source Shortest Paths . . . 91
5.5 Breadth-First Search Tree . . . 93
5.6 Leader Election . . . 95
5.7 All-Pairs Shortest Paths . . . 98
5.8 Quiz . . . 100
5.9 Exercises . . . 101
5.10 Bibliographic Notes . . . 102

6 Randomized Algorithms 103

6.1 Definitions . . . 103
6.2 Probabilistic Analysis . . . 104
6.3 With High Probability . . . 105
6.4 Randomized Coloring in Bounded-Degree Graphs . . . 106
6.4.1 Algorithm Idea . . . 106
6.4.2 Algorithm . . . 107
6.4.3 Analysis . . . 109

6.5 Quiz . . . 111
6.6 Exercises . . . 111
6.7 Bibliographic Notes . . . 113

Part IV Proving Impossibility Results

7 Covering Maps **115**

7.1 Definition . . . 115
7.2 Covers and Executions . . . 118
7.3 Examples . . . 119
7.4 Quiz . . . 124
7.5 Exercises . . . 124
7.6 Bibliographic Notes . . . 128

8 Local Neighborhoods **129**

8.1 Definitions . . . 129
8.2 Local Neighborhoods and Executions . . . 129
8.3 Example: 2-Coloring Paths . . . 131
8.4 Quiz . . . 133
8.5 Exercises . . . 134
8.6 Bibliographic Notes . . . 136

9 Round Elimination **137**

9.1 Bipartite Model and Biregular Trees . . . 137
9.1.1 Bipartite Locally Verifiable Problem . . . 138
9.1.2 Examples . . . 139
9.2 Introducing Round Elimination . . . 142
9.2.1 Impossibility Using Iterated Round Elimination . . . 143
9.2.2 Output Problems . . . 143
9.2.3 Example: Weak 3-labeling . . . 144
9.2.4 Complexity of Output Problems . . . 145
9.2.5 Example: Complexity of Weak 3-labeling . . . 147

9.2.6 Example: Iterated Round Elimination 148
9.3 Quiz 150
9.4 Exercises 150
9.5 Bibliographic Notes 152

10 Sinkless Orientation **153**

10.1 Sinkless Orientation on Paths 153
10.1.1 Hardness of Sinkless Orientation 154
10.1.2 Solving Sinkless Orientation on Paths 155
10.2 Sinkless Orientation on Trees 156
10.2.1 Solving Sinkless Orientation on Trees 157
10.2.2 Roadmap: Next Steps 159
10.3 Maximal Output Problems 159
10.4 Hardness of Sinkless Orientation on Trees 161
10.4.1 First Step 161
10.4.2 Equivalent Formulation 162
10.4.3 Fixed Points in Round Elimination 163
10.4.4 Sinkless Orientation Gives a Fixed Point 164
10.5 Quiz 167
10.6 Exercises 167
10.7 Bibliographic Notes 169

11 Hardness of Coloring **170**

11.1 Coloring and Round Elimination 170
11.1.1 Encoding Coloring 171
11.1.2 Output Problem of Coloring 172
11.1.3 Simplification 173
11.1.4 Generalizing Round Elimination for Coloring . . 175
11.1.5 Sequence of Output Problems 176
11.2 Round Elimination with Inputs 177
11.2.1 Randomized Round Elimination Step 178

11.3 Iterated Randomized Round Elimination 180
11.3.1 Proof of Lemma 11.1 183
11.3.2 Proof of Lemma 11.2 185
11.4 Quiz 187
11.5 Exercises 187
11.6 Bibliographic Notes 188

Part V Conclusions

12 Conclusions **190**

12.1 What Have We Learned? 190
12.2 What Else Exists? 191
12.2.1 Distance vs. Bandwidth vs. Local Memory 191
12.2.2 Asynchronous and Fault-Tolerant Algorithms . . 193
12.2.3 Other Directions 194
12.2.4 Research in Distributed Algorithms 195
12.3 Quiz 196
12.4 Exercises 197
12.5 Bibliographic Notes 198

Hints **199**

Bibliography **206**

Foreword

This book is an introduction to the theory of distributed algorithms, with a focus on distributed graph algorithms (network algorithms). The topics covered include:

- **Models of computing:** precisely what is a distributed algorithm, and what do we mean when we say that a distributed algorithm solves a certain computational problem?
- **Algorithm design and analysis:** which computational problems can be solved with distributed algorithms, which problems can be solved *efficiently*, and how to do so?
- **Computability and computational complexity:** which computational problems *cannot* be solved at all with distributed algorithms, which problems cannot be solved *efficiently*, why is this the case, and how to prove it?

No prior knowledge of distributed systems is required. Basic knowledge of discrete mathematics and graph theory is assumed, as well as familiarity with the basic concepts from undergraduate-level courses on models of computation, computational complexity, and algorithms and data structures.

About the Course

This textbook was written to support the lecture course *CS-E4510 Distributed Algorithms* at Aalto University. The course is worth 5 ECTS credits. There are 12 weeks of lectures. Each week we cover one chapter of this book, and our students are expected to complete the quiz and at least three of the exercises from the chapter.

Acknowledgments

Many thanks to Jaakko Alasaari, Alkida Balliu, Sebastian Brandt, Arthur Carels, Jacques Charnay, Faith Ellen, Massimo Equi, Aelitta Ezugbaya, Mika Göös, Jakob Greistorfer, Jana Hauer, Nikos Heikkilä, Joel Kaasinen, Samu Kallio, Mirco Kroon, Siiri Kuoppala, Teemu Kuusisto, Dang Lam, Tuomo Lempiäinen, Christoph Lenzen, Darya Melnyk, Abdulmelik Mohammed, Christopher Purcell, Mikaël Rabie, Joel Rybicki, Joona Savela, Stefan Schmid, Paul Schulte, Roelant Stegmann, Aleksandr Tereshchenko, Verónica Toro Betancur, Przemysław Uznański, and Jussi Väisänen for feedback, discussions, comments, and for helping us with the arrangements for this course. This work was supported in part by the Academy of Finland, grant 252018.

Website

For updates and additional material, see

https://jukkasuomela.fi/da2020/

License

This work is licensed under the *Creative Commons Attribution 4.0 International Public License*. To view a copy of this license, visit

https://creativecommons.org/licenses/by/4.0/

Part I
Informal Introduction

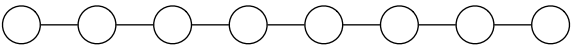
Chapter 1

Warm-Up

We will start this course with an informal introduction to distributed algorithms. We will formalize the model of computing later but for now the intuitive idea of computers that can exchange messages with each other is sufficient.

1.1 Running Example: Coloring Paths

Imagine that we have n computers (or *nodes* as they are usually called) that are connected to each other with communication channels so that the network topology is a *path*:



The computers can exchange messages with their neighbors. All computers run the *same* algorithm—this is the *distributed algorithm* that we will design. The algorithm will decide what messages a computer sends in each step, how it processes the messages that it receives, when it stops, and what it outputs when it stops.

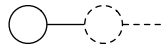
In this example, the task is to find a proper *coloring* of the path with 3 colors. That is, each node has to output one of the colors, 1, 2, or 3, so that neighbors have different colors—here is an example of a proper solution:



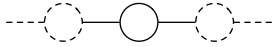
1.2 Challenges of Distributed Algorithms

With a bird's-eye view of the entire network, coloring a path looks like a very simple task: just start from one endpoint and assign colors 1 and 2 alternately. However, in a real-world computer network we usually do not have all-powerful entities that know everything about the network and can directly tell each computer what to do.

Indeed, when we start a networked computer, it is typically only aware of itself and the communication channels that it can use. In our simple example, the endpoints of the path know that they have one neighbor:

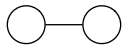


All other nodes along the path just know that they have two neighbors:



For example, the second node along the path looks no different from the third node, yet somehow they have to produce *different* outputs.

Obviously, the nodes have to exchange *messages* with each other in order to figure out a proper solution. Yet this turns out to be surprisingly difficult even in the case of just $n = 2$ nodes:



If we have two *identical* computers connected to each other with a single communication link, both computers are started simultaneously, and both of them run the same deterministic algorithm, how could they ever end up in *different* states?

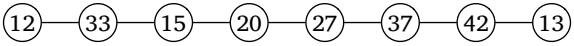
The answer is that it is not possible, without some additional assumptions. In practice, we could try to rely on some real-world imperfections (e.g., the computers are seldom perfectly synchronized), but in the theory of distributed algorithms we often assume that there is some *explicit* way to *break symmetry* between otherwise identical computers. In this chapter, we will have a brief look at two common assumptions:

- each computer has a unique name,
- each computer has a source of random bits.

In subsequent chapters we will then formalize these models, and develop a theory that will help us understand precisely what kind of tasks can be solved in each case, and how fast.

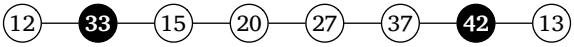
1.3 Coloring with Unique Identifiers

There are plenty of examples of real-world networks with globally unique identifiers: public IPv4 and IPv6 addresses are globally unique identifiers of Internet hosts, devices connected to an Ethernet network have globally unique MAC addresses, mobile phones have their IMEI numbers, etc. The common theme is that the identifiers are globally unique, and the numbers can be interpreted as natural numbers:

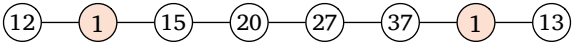


With the help of unique identifiers, it is now easy to design an algorithm that colors a path. Indeed, the unique identifiers already form a coloring with a large number of colors! All that we need to do is to reduce the number of colors to 3.

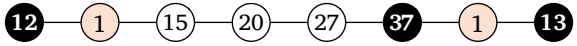
We can use the following simple strategy. In each step, a node is active if it is a "local maximum", i.e., its current color is larger than the current colors of its neighbors:



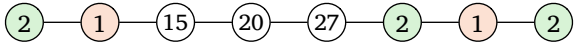
The active nodes will then pick a new color from the color palette $\{1, 2, 3\}$, so that it does not conflict with the current colors of their neighbors. This is always possible, as each node in a path has at most 2 neighbors, and we have 3 colors in our color palette:



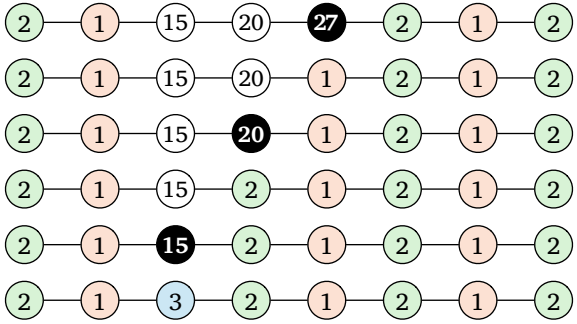
Then we simply repeat the same procedure until all nodes have small colors. First find the local maxima:



And then recolor the local maxima with colors $\{1, 2, 3\}$:



Continuing this way we will eventually have a path that is properly colored with colors $\{1, 2, 3\}$:



Note that we may indeed be forced to use all three colors.

So far we have sketched an algorithm idea, but we still have to show that we can actually implement this idea as a distributed algorithm. Remember that there is no central control; nobody has a bird's-eye view of the entire network. Each node is an independent computer, and all computers are running the *same* algorithm. What would the algorithm look like?

Let us fix some notation. Each node maintains a variable c that contains its current color. Initially, c is equal to the unique identifier of the node. Then computation proceeds as shown in Table 1.1.

This shows a typical structure of a distributed algorithm: an infinite send–receive–compute loop. A computer is seen as a state machine; here c is the variable that holds the current state of the computer. In this

Repeat forever:

- Send message c to all neighbors.
- Receive messages from all neighbors.
  Let M be the set of messages received.
- If $c \notin \{1,2,3\}$ and $c > \max M$:
  Let $c \leftarrow \min(\{1,2,3\} \setminus M)$.

Table 1.1: A simple 3-coloring algorithm for paths.

algorithm, we have three *stopping states*: $c = 1$, $c = 2$, and $c = 3$. It is easy to verify that the algorithm is indeed correct in the following sense:

(a) In any path graph, for any assignment of unique identifiers, all computers will eventually reach a stopping state.

(b) Once a computer reaches a stopping state, it never changes its state.

The second property is very important: each computer has to know when it is safe to announce its output and stop.

Our algorithm may look a bit strange in the sense that computers that have "stopped" are still sending messages. However, it is fairly straightforward to rewrite the algorithm so that you could actually turn off computers that have stopped. The basic idea is that nodes that are going to switch to a stopping state first inform their neighbors about this. Each node will memorize which of its neighbors have already stopped and what were their final colors. Implementing this idea is left as Exercise 1.2, and you will later see that this can be done for *any* distributed algorithm. Hence, without loss of generality, we can play by the following simple rules:

- The nodes are state machines that repeatedly send messages to their neighbors, receive messages from their neighbors, and up-

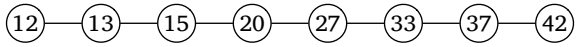
date their state—all nodes perform these steps synchronously in parallel.

- Some of the states are stopping states, and once a node reaches a stopping state, it no longer changes its state.
- Eventually all nodes have to reach stopping states, and these states must form a correct solution to the problem that we want to solve.

Note that here a "state machine" does not necessarily refer to a *finite*-state machine. We can perfectly well have a state machine with infinitely many states. Indeed, in the example of Table 1.1 the set of possible states was the set of all positive integers.

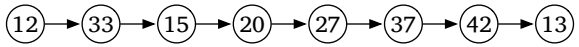
1.4 Faster Coloring with Unique Identifiers

So far we have seen that with the help of unique identifiers, it is *possible* to find a 3-coloring of a path. However, the algorithm that we designed is not particularly efficient in the worst case. To see this, consider a path in which the unique identifiers happen to be assigned in an increasing order:



In such a graph, in each round there is only one node that is active. In total, it will take $\Theta(n)$ rounds until all nodes have stopped.

However, it is possible to color paths *much* faster. The algorithm is easier to explain if we have a *directed* path:



That is, we have a consistent orientation in the path so that each node has at most one "predecessor" and at most one "successor". The orientations are just additional information that we will use in algorithm design—nodes can always exchange information along each edge in either direction. Once we have presented the algorithm for directed paths, we will then generalize it to undirected paths in Exercise 1.3.

1.4.1 Algorithm Overview

For the sake of concreteness, let us assume that the nodes are labeled with 128-bit unique identifiers—for example, IPv6 addresses. In most real-world networks 2^{128} identifiers is certainly more than enough, but the same idea can be easily generalized to arbitrarily large identifiers if needed.

Again, we will interpret the unique identifiers as colors; hence our starting point is a path that is properly colored with 2^{128} colors. In the next section, we will present a fast color reduction algorithm for directed paths that reduces the number of colors from 2^x to $2x$ in one round, for any positive integer x. Hence in one step we can reduce the number of colors from 2^{128} to $2 \cdot 128 = 256$. In just four iterations we can reduce the number of colors from 2^{128} to 6, as follows:

$$\begin{aligned} 2^{128} &\to 2 \cdot 128 = 2^8, \\ 2^8 &\to 2 \cdot 8 = 2^4, \\ 2^4 &\to 2 \cdot 4 = 2^3, \\ 2^3 &\to 2 \cdot 3 = 6. \end{aligned}$$

Once we have found a 6-coloring, we can then apply the algorithm of Table 1.1 to reduce the number of colors from 6 to 3. It is easy to see that this will take at most 3 rounds. Overall, we have an algorithm that reduces the number of colors from 2^{128} to 3 in only 7 rounds—no matter how many nodes we have in the path. Compare this with the simple 3-coloring algorithm, which may take millions of rounds for paths with millions of nodes.

1.4.2 Algorithm for One Step

Let us now show how to reduce the number of colors from 2^x to $2x$ in one round; this will be achieved by doing some bit manipulations. First, each node sends its current color to its predecessor. After this step, each node u knows two values:

- $c_0(u)$, the current color of the node,
- $c_1(u)$, the current color of its successor.

If a node does not have any successor, it just proceeds *as if* it had a successor of some color different from $c_0(u)$.

We can interpret both $c_0(u)$ and $c_1(u)$ as x-bit binary strings that represent integers from range 0 to $2^x - 1$. We know that the current color of node u is different from the current color of its successor, i.e., $c_0(u) \neq c_1(u)$. Hence in the two binary strings $c_0(u)$ and $c_1(u)$ there is at least one bit that differs. Define:

- $i(u) \in \{0, 1, \ldots, x-1\}$ is the ***index*** of the first bit that differs between $c_0(u)$ and $c_1(u)$,
- $b(u) \in \{0, 1\}$ is the ***value*** of bit number $i(u)$ in $c_0(u)$.

Finally, node u chooses

$$c(u) = 2i(u) + b(u)$$

as its new color.

1.4.3 An Example

Let $x = 8$, i.e., nodes are colored with 8-bit numbers. Assume that we have a node u of color 123, and u has a successor v of color 47; see Table 1.2 for an illustration. In binary, we have

$$c_0(u) = 01111011_2,$$
$$c_1(u) = 00101111_2.$$

Counting from the least significant bit, node u can see that:

- bit number 0 is the same in both $c_0(u)$ and $c_1(u)$,
- bit number 1 is the same in both $c_0(u)$ and $c_1(u)$,
- bit number 2 is different in $c_0(u)$ and $c_1(u)$.

Hence we will set

$$i(u) = 2, \quad b(u) = 0, \quad c(u) = 2 \cdot 2 + 0 = 4.$$

node u	input $c_0(u)$	$c_1(u)$	$i(u)$	$b(u)$	output $c(u)$
...	...	...	...	...	...
	01111011_2	00101111_2	2	0	4
	00101111_2	01101011_2	2	1	5
	01101011_2	...	...	...	...
...	...				
...	...	...	...	...	...
	01111011_2	00101111_2	2	0	4
	00101111_2	01101111_2	6	0	12
	01101111_2	...	...	...	...
...	...				

Table 1.2: Fast color reduction algorithm for directed paths: reducing the number of colors from 2^x to $2x$, for $x = 8$. There are two interesting cases: either $i(u)$ is the same for two neighbors (first example), or they are different (second example). In the first case, the values $b(u)$ will differ, and in the second case, the values $i(u)$ will differ. In both cases, the final colors $c(u)$ will be different.

That is, node u picks 4 as its new color. If all other nodes run the same algorithm, this will be a valid choice—as we will argue next, both the predecessor and the successor of u will pick a color that is different from 4.

1.4.4 Correctness

Clearly, the value $c(u)$ is in the range $\{0, 1, \ldots, 2x-1\}$. However, it is not entirely obvious that these values actually produce a proper $2x$-coloring of the path. To see this, consider a pair of nodes u and v so that v is the successor of u. By definition, $c_1(u) = c_0(v)$. We need to show that $c(u) \neq c(v)$. There are two cases—see Table 1.2 for an example:

(a) $i(u) = i(v) = i$: We know that $b(u)$ is bit number i of $c_0(u)$, and $b(v)$ is bit number i of $c_1(u)$. By the definition of $i(u)$, we also know that these bits differ. Hence $b(u) \neq b(v)$ and $c(u) \neq c(v)$.

(b) $i(u) \neq i(v)$: No matter how we choose $b(u) \in \{0, 1\}$ and $b(v) \in \{0, 1\}$, we have $c(u) \neq c(v)$.

We have argued that $c(u) \neq c(v)$ for any pair of two adjacent nodes u and v, and the value of $c(u)$ is an integer between 0 and $2x-1$ for each node u. Hence the algorithm finds a proper $2x$-coloring in one round.

1.4.5 Iteration

The algorithm that we presented in this section can reduce the number of colors from 2^x to $2x$ in one round; put otherwise, we can reduce the number of colors from x to $O(\log x)$ in one round.

If we iterate the algorithm, we can reduce the number of colors from x to 6 in $O(\log^* x)$ rounds (please refer to Section 1.10 for the definition of the $\log^*$ function if you are not familiar with it).

Once we have reduced the number of colors to 6, we can use the simple color reduction algorithm from Section 1.3 to reduce the number of colors from 6 to 3 in 3 rounds. The details of the analysis are left as Exercises 1.5 and 1.6.

1.5 Coloring with Randomized Algorithms

So far we have used unique identifiers to break symmetry. Another possibility is to use randomness. Here is a simple randomized distributed algorithm that finds a proper 3-coloring of a path: nodes try to pick colors from the palette $\{1, 2, 3\}$ uniformly at random, and they stop once they succeed in picking a color that is different from the colors of their neighbors.

1.5.1 Algorithm

Let us formalize the *simple randomized 3-coloring algorithm* that we sketched above. Each node u has a flag $s(u) \in \{0, 1\}$ indicating whether it has stopped, and a variable $c(u) \in \{1, 2, 3\}$ that stores its current color. If $s(u) = 1$, a node has stopped and its output is $c(u)$.

In each step, each node u with $s(u) = 0$ picks a new color $c(u) \in \{1, 2, 3\}$ uniformly at random. Then each node sends its current color $c(u)$ to its neighbors. If $c(u)$ is different from the colors of its neighbors, u will set $s(u) = 1$ and stop; otherwise it tries again in the next round.

1.5.2 Analysis

It is easy to see that in each step, a node u will stop with probability at least $1/3$: after all, no matter what its neighbors do, there is at least one choice for $c(u) \in \{1, 2, 3\}$ that does not conflict with its neighbors.

Fix a positive constant C. Consider what happens if we run the algorithm for

$$k = (C + 1) \log_{3/2} n$$

steps, where n is the number of nodes in the network. Now the probability that a given node u has not stopped after k steps is at most

$$(1 - 1/3)^k = \frac{1}{n^{C+1}}.$$

By the union bound, the probability that there is a node that has not stopped is at most $1/n^C$. Hence with probability at least $1 - 1/n^C$, all nodes have stopped after k steps.

1.5.3 With High Probability

Let us summarize what we have achieved: for any given constant C, there is an algorithm that runs for $k = O(\log n)$ rounds and produces a proper 3-coloring of a path with probability $1 - 1/n^C$. We say that the algorithm runs in time $O(\log n)$ *with high probability*—here the phrase "high probability" means that we can choose any constant C and the algorithm will succeed at least with a probability of $1 - 1/n^C$. Note that even for a moderate value of C, say, $C = 10$, the success probability approaches 1 very rapidly as n increases.

1.6 Summary

In this chapter we have seen three different distributed algorithms for 3-coloring paths:

- A simple 3-coloring algorithm, Section 1.3: A deterministic algorithm for paths with unique identifiers. Runs in $O(n)$ rounds, where n is the number of nodes.
- A fast 3-coloring algorithm, Section 1.4: A deterministic algorithm for *directed* paths with unique identifiers. Runs in $O(\log^* x)$ rounds, where x is the largest identifier.
- A simple randomized 3-coloring algorithm, Section 1.5: A randomized algorithm for paths without unique identifiers. Runs in $O(\log n)$ rounds with high probability.

We will explore and analyze these algorithms and their variants in more depth in the exercises.

1.7 Quiz

Construct a directed path of 3 nodes that is labeled with unique identifiers (of any size) such that the following holds: After two iterations of the fast color reduction algorithm from Section 1.4.2, the color of the first node is 7.

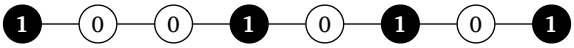
It is enough to just list the three unique identifiers (in decimal); there is no need to explain anything else.

1.8 Exercises

Exercise 1.1 (maximal independent sets)**.** A *maximal independent set* is a set of nodes I that satisfies the following properties:

- for each node $v \in I$, none of its neighbors are in I,
- for each node $v \notin I$, at least one of its neighbors is in I.

Here is an example—the nodes labeled with a "1" form a maximal independent set:



Your task is to design a distributed algorithm that finds a maximal independent set in any path graph, for each of the following settings:

(a) a deterministic algorithm for paths with arbitrarily large unique identifiers,

(b) a fast deterministic algorithm for *directed* paths with 128-bit unique identifiers,

(c) a randomized algorithm that does not need unique identifiers.

In part (a), use the techniques presented in Section 1.3, in part (b), use the techniques presented in Section 1.4, and in part (c), use the techniques presented in Section 1.5.

Exercise 1.2 (stopped nodes)**.** Rewrite the greedy algorithm of Table 1.1 so that stopped nodes do not need to send messages. Be precise: explain your algorithm in detail so that you could easily implement it.

Exercise 1.3 (undirected paths)**.** The fast 3-coloring algorithm from Section 1.4 finds a 3-coloring very fast in any directed path. Design an algorithm that is almost as fast and works in any path, even if the edges are not directed. You can assume that the range of identifiers is known.

▷ *hint A*

Exercise 1.4 (randomized and fast)**.** The simple randomized 3-coloring algorithm finds a 3-coloring in time $O(\log n)$ with high probability, and it does not need any unique identifiers. Can you design a randomized algorithm that finds a 3-coloring in time $o(\log n)$ with high probability? You can assume that n is known.

▷ *hint B*

Exercise 1.5 (asymptotic analysis)**.** Analyze the fast 3-coloring algorithm from Section 1.4:

(a) Assume that we are given a coloring with x colors; the colors are numbers from $\{1, 2, \ldots, x\}$. Show that we can find a 3-coloring in time $O(\log^* x)$.

(b) Assume that we are given unique identifiers that are polynomial in n, that is, there is a constant $c = O(1)$ such that the unique identifiers are a subset of $\{1, 2, \ldots, n^c\}$. Show that we can find a 3-coloring in time $O(\log^* n)$.

⋆ **Exercise 1.6** (tight analysis)**.** Analyze the fast 3-coloring algorithm from Section 1.4: Assume that we are given a coloring with x colors, for any integer $x \geq 6$; the colors are numbers from $\{1, 2, \ldots, x\}$. Show that we can find a 6-coloring in time $\log^*(x)$, and therefore a 3-coloring in time $\log^*(x) + 3$.

▷ *hint C*

⋆ **Exercise 1.7** (oblivious algorithms)**.** The simple 3-coloring algorithm works correctly even if we do not know how many nodes there are in

the network, or what is the range of unique identifiers—we say that the algorithm is *oblivious*. Adapt the fast 3-coloring algorithm from Section 1.4 so that it is also oblivious.

▷ *hint D*

1.9 Bibliographic Notes

The fast 3-coloring algorithm (Section 1.4) was originally presented by Cole and Vishkin [15] and further refined by Goldberg et al. [22]; in the literature, it is commonly known as the "Cole–Vishkin algorithm". Exercise 1.7 was inspired by Korman et al. [27].

1.10 Appendix: Mathematical Preliminaries

In the analysis of distributed algorithms, we will encounter power towers and iterated logarithms.

1.10.1 Power Tower

We write power towers with the notation

$$^{i}2 = 2^{2^{\cdot^{\cdot^{2}}}},$$

where there are i twos in the tower. Power towers grow very fast; for example,

$$\begin{aligned}
^{1}2 &= 2,\\
^{2}2 &= 4,\\
^{3}2 &= 16,\\
^{4}2 &= 65536,\\
^{5}2 &= 2^{65536} > 10^{19728}.
\end{aligned}$$

1.10.2 Iterated Logarithm

The iterated logarithm of x, in notation $\log^* x$ or $\log^*(x)$, is defined recursively as follows:

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

In essence, this is the inverse of the power tower function. For all positive integers i, we have

$$\log^*({}^i 2) = i.$$

As power towers grow very fast, iterated logarithms grow very slowly; for example,

$$\begin{aligned} \log^* 2 &= 1, & \log^* 16 &= 3, & \log^* 10^{10} &= 5, \\ \log^* 3 &= 2, & \log^* 17 &= 4, & \log^* 10^{100} &= 5, \\ \log^* 4 &= 2, & \log^* 65536 &= 4, & \log^* 10^{1000} &= 5, \\ \log^* 5 &= 3, & \log^* 65537 &= 5, & \log^* 10^{10000} &= 5, \ldots \end{aligned}$$

Part II
Graphs

Chapter 2

Graph-Theoretic Foundations

The study of distributed algorithms is closely related to graphs: we will interpret a computer network as a graph, and we will study computational problems related to this graph. In this section we will give a summary of the graph-theoretic concepts that we will use.

2.1 Terminology

A *simple undirected graph* is a pair $G = (V, E)$, where V is the set of *nodes* (*vertices*) and E is the set of *edges*. Each edge $e \in E$ is a 2-subset of nodes, that is, $e = \{u, v\}$ where $u \in V$, $v \in V$, and $u \neq v$. Unless otherwise mentioned, we assume that V is a non-empty finite set; it follows that E is a finite set. Usually, we will draw graphs using circles and lines—each circle represents a node, and a line that connects two nodes represents an edge.

2.1.1 Adjacency

If $e = \{u, v\} \in E$, we say that node u is *adjacent* to v, nodes u and v are *neighbors*, node u is *incident* to e, and edge e is also *incident* to u. If $e_1, e_2 \in E$, $e_1 \neq e_2$, and $e_1 \cap e_2 \neq \emptyset$ (i.e., e_1 and e_2 are distinct edges that share an endpoint), we say that e_1 is *adjacent* to e_2.

The *degree* of a node $v \in V$ in graph G is

$$\deg_G(v) = \left|\left\{ u \in V : \{u, v\} \in E \right\}\right|.$$

That is, v has $\deg_G(v)$ neighbors; it is adjacent to $\deg_G(v)$ nodes and incident to $\deg_G(v)$ edges. A node $v \in V$ is *isolated* if $\deg_G(v) = 0$. Graph G is *k-regular* if $\deg_G(v) = k$ for each $v \in V$.

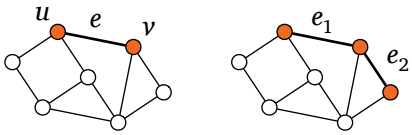


Figure 2.1: Node u is adjacent to node v. Nodes u and v are incident to edge e. Edge e_1 is adjacent to edge e_2.

2.1.2 Subgraphs

Let $G = (V, E)$ and $H = (V_2, E_2)$ be two graphs. If $V_2 \subseteq V$ and $E_2 \subseteq E$, we say that H is a *subgraph* of G. If $V_2 = V$, we say that H is a *spanning subgraph* of G.

If $V_2 \subseteq V$ and $E_2 = \{\{u, v\} \in E : u \in V_2,\ v \in V_2\}$, we say that $H = (V_2, E_2)$ is an *induced subgraph*; more specifically, H is the subgraph of G induced by the set of nodes V_2.

If $E_2 \subseteq E$ and $V_2 = \bigcup E_2$, we say that H is an *edge-induced subgraph*; more specifically, H is the subgraph of G induced by the set of edges E_2.

2.1.3 Walks

A *walk* of length ℓ from node v_0 to node v_ℓ is an alternating sequence

$$w = (v_0, e_1, v_1, e_2, v_2, \ldots, e_\ell, v_\ell)$$

where $v_i \in V$, $e_i \in E$, and $e_i = \{v_{i-1}, v_i\}$ for all i; see Figure 2.2. The walk is *empty* if $\ell = 0$. We say that walk w *visits* the nodes $v_0, v_1, \ldots, v_\ell$, and it *traverses* the edges $e_1, e_2, \ldots, e_\ell$. In general, a walk may visit the same node more than once and it may traverse the same edge more than once. A *non-backtracking walk* does not traverse the same edge twice consecutively, that is, $e_{i-1} \neq e_i$ for all i. A *path* is a walk that visits each node at most once, that is, $v_i \neq v_j$ for all $0 \leq i < j \leq \ell$. A walk is *closed* if $v_0 = v_\ell$. A *cycle* is a non-empty closed walk with $v_i \neq v_j$ and $e_i \neq e_j$ for all $1 \leq i < j \leq \ell$; see Figure 2.3. Note that the length of a cycle is at least 3.

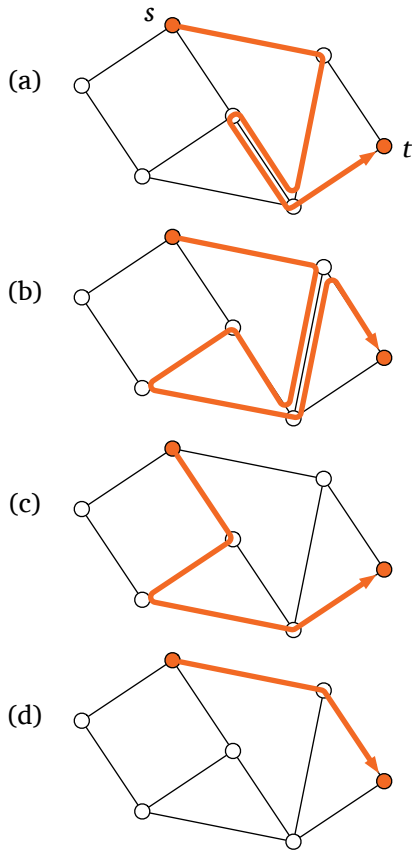


Figure 2.2: (a) A walk of length 5 from s to t. (b) A non-backtracking walk. (c) A path of length 4. (d) A path of length 2; this is a shortest path and hence $\text{dist}_G(s,t) = 2$.

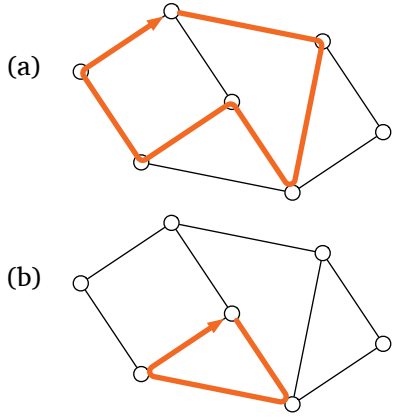


Figure 2.3: (a) A cycle of length 6. (b) A cycle of length 3; this is a shortest cycle and hence the girth of the graph is 3.

2.1.4 Connectivity and Distances

For each graph $G = (V, E)$, we can define a relation $\rightsquigarrow$ on V as follows: $u \rightsquigarrow v$ if there is a walk from u to v. Clearly $\rightsquigarrow$ is an equivalence relation. Let $C \subseteq V$ be an equivalence class; the subgraph induced by C is called a *connected component* of G.

If u and v are in the same connected component, there is at least one *shortest path* from u to v, that is, a path from u to v of the smallest possible length. Let ℓ be the length of a shortest path from u to v; we define that the *distance* between u and v in G is $\text{dist}_G(u, v) = \ell$. If u and v are not in the same connected component, we define $\text{dist}_G(u, v) = \infty$. Note that $\text{dist}_G(u, u) = 0$ for any node u.

For each node v and for a non-negative integer r, we define the *radius-r neighborhood* of v as follows (see Figure 2.4):

$$\text{ball}_G(v, r) = \{\, u \in V : \text{dist}_G(u, v) \leq r \,\}.$$

A graph is *connected* if it consists of one connected component. The *diameter* of graph G, in notation $\text{diam}(G)$, is the length of a longest

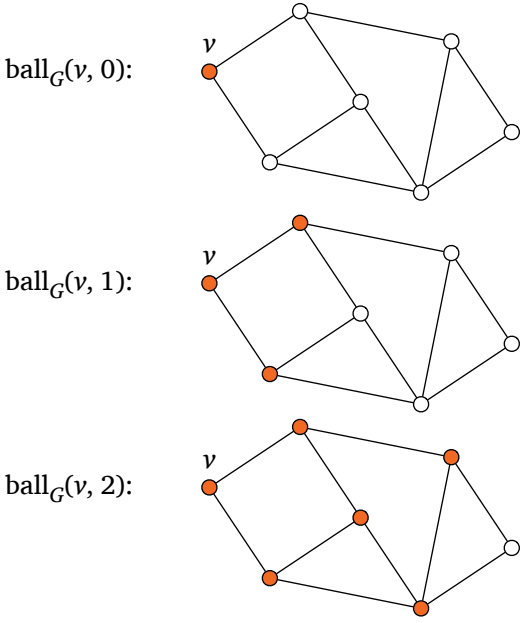


Figure 2.4: Neighborhoods.

shortest path, that is, the maximum of $\text{dist}_G(u, v)$ over all $u, v \in V$; we have $\text{diam}(G) = \infty$ if the graph is not connected.

The *girth* of graph G is the length of a shortest cycle in G. If the graph does not have any cycles, we define that the girth is ∞; in that case we say that G is *acyclic*.

A *tree* is a connected, acyclic graph. If $T = (V, E)$ is a tree and $u, v \in V$, then there exists precisely one path from u to v. An acyclic graph is also known as a *forest*—in a forest each connected component is a tree. A *pseudotree* has at most one cycle, and in a *pseudoforest* each connected component is a pseudotree.

A *path graph* is a graph that consists of one path, and a *cycle graph* is a graph that consists of one cycle. Put otherwise, a path graph is a tree in which all nodes have degree at most 2, and a cycle graph is a 2-regular pseudotree. Note that any graph of maximum degree 2 consists of disjoint paths and cycles, and any 2-regular graph consists of disjoint cycles.

2.1.5 Isomorphism

An *isomorphism* from graph $G_1 = (V_1, E_1)$ to graph $G_2 = (V_2, E_2)$ is a bijection $f : V_1 \to V_2$ that preserves adjacency: $\{u, v\} \in E_1$ if and only if $\{f(u), f(v)\} \in E_2$. If an isomorphism from G_1 to G_2 exists, we say that G_1 and G_2 are isomorphic.

If G_1 and G_2 are isomorphic, they have the same structure; informally, G_2 can be constructed by renaming the nodes of G_1 and vice versa.

2.2 Packing and Covering

A subset of nodes $X \subseteq V$ is

(a) an *independent set* if each edge has at most one endpoint in X, that is, $|e \cap X| \leq 1$ for all $e \in E$,

(b) a *vertex cover* if each edge has at least one endpoint in X, that is, $e \cap X \neq \emptyset$ for all $e \in E$,

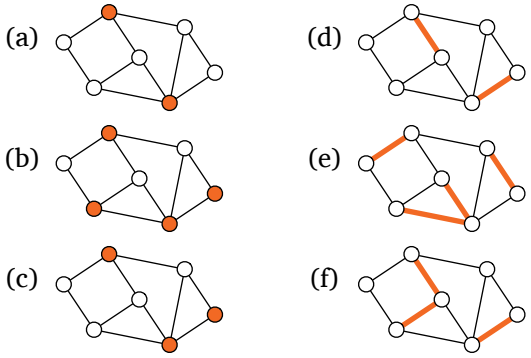


Figure 2.5: Packing and covering problems; see Section 2.2.

(c) a *dominating set* if each node $v \notin X$ has at least one neighbor in X, that is, $\text{ball}_G(v,1) \cap X \neq \varnothing$ for all $v \in V$.

A subset of edges $X \subseteq E$ is

(d) a *matching* if each node has at most one incident edge in X, that is, $\{t,u\} \in X$ and $\{t,v\} \in X$ implies $u = v$,

(e) an *edge cover* if each node has at least one incident edge in X, that is, $\bigcup X = V$,

(f) an *edge dominating set* if each edge $e \notin X$ has at least one neighbor in X, that is, $e \cap \left(\bigcup X\right) \neq \varnothing$ for all $e \in E$.

See Figure 2.5 for illustrations.

Independent sets and matchings are examples of *packing problems*—intuitively, we have to "pack" elements into set X while avoiding conflicts. Packing problems are *maximization problems*. Typically, it is trivial to find a feasible solution (for example, an empty set), but it is more challenging to find a large solution.

Vertex covers, edge covers, dominating sets, and edge dominating sets are examples of *covering problems*—intuitively, we have to find a set X that "covers" the relevant parts of the graph. Covering problems are

minimization problems. Typically, it is trivial to find a feasible solution if it exists (for example, the set of all nodes or all edges), but it is more challenging to find a small solution.

The following terms are commonly used in the context of maximization problems; it is important not to confuse them:

(a) ***maximal***: a maximal solution is not a proper subset of another feasible solution,

(b) ***maximum***: a maximum solution is a solution of the largest possible cardinality.

Similarly, in the context of minimization problems, analogous terms are used:

(a) ***minimal***: a minimal solution is not a proper superset of another feasible solution,

(b) ***minimum***: a minimum solution is a solution of the smallest possible cardinality.

Using this convention, we can define the terms *maximal independent set*, *maximum independent set*, *maximal matching*, *maximum matching*, *minimal vertex cover*, *minimum vertex cover*, etc.

For example, Figure 2.5a shows a maximal independent set: it is not possible to greedily extend the set by adding another element. However, it is not a maximum independent set: there exists an independent set of size 3. Figure 2.5d shows a matching, but it is not a maximal matching, and therefore it is not a maximum matching either.

Typically, maximal and minimal solutions are easy to find—you can apply a greedy algorithm. However, maximum and minimum solutions can be very difficult to find—many of these problems are NP-hard optimization problems.

A *minimum maximal matching* is precisely what the name suggests: it is a maximal matching of the smallest possible cardinality. We can define a *minimum maximal independent set*, etc., in an analogous manner.

2.3 Labelings and Partitions

We will often encounter functions of the form

$$f : V \to \{1, 2, \ldots, k\}.$$

There are two interpretations that are often helpful:

(i) Function f assigns a *label* $f(v)$ to each node $v \in V$. Depending on the context, the labels can be interpreted as colors, time slots, etc.

(ii) Function f is a *partition* of V. More specifically, f defines a partition $V = V_1 \cup V_2 \cup \cdots \cup V_k$ where $V_i = f^{-1}(i) = \{ v \in V : f(v) = i \}$.

Similarly, we can study a function of the form

$$f : E \to \{1, 2, \ldots, k\}$$

and interpret it either as a labeling of edges or as a partition of E.

Many graph problems are related to such functions. We say that a function $f : V \to \{1, 2, \ldots, k\}$ is

(a) a *proper vertex coloring* if $f^{-1}(i)$ is an independent set for each i,

(b) a *weak coloring* if each non-isolated node u has a neighbor v with $f(u) \neq f(v)$,

(c) a *domatic partition* if $f^{-1}(i)$ is a dominating set for each i.

A function $f : E \to \{1, 2, \ldots, k\}$ is

(d) a *proper edge coloring* if $f^{-1}(i)$ is a matching for each i,

(e) an *edge domatic partition* if $f^{-1}(i)$ is an edge dominating set for each i.

See Figure 2.6 for illustrations.

Usually, the term *coloring* refers to a proper vertex coloring, and the term *edge coloring* refers to a proper edge coloring. The value of k is

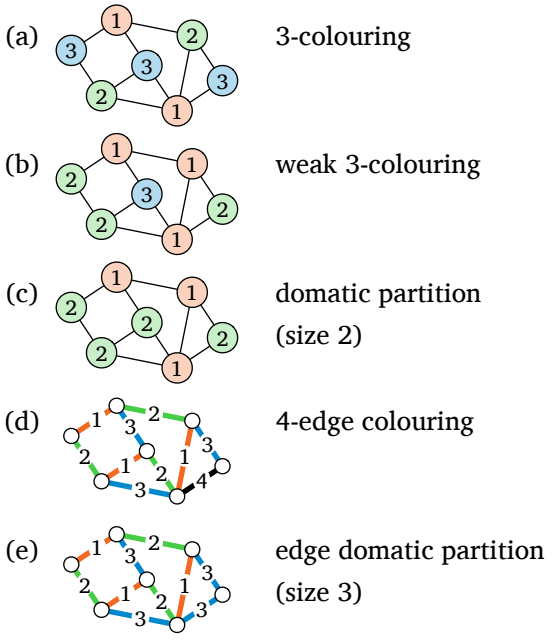


Figure 2.6: Partition problems; see Section 2.3.

the *size* of the coloring or the *number of colors*. We will use the term *k-coloring* to refer to a proper vertex coloring with k colors; the term *k-edge coloring* is defined in an analogous manner.

A graph that admits a 2-coloring is a *bipartite graph*. Equivalently, a bipartite graph is a graph that does not have an odd cycle.

Graph coloring is typically interpreted as a minimization problem. It is easy to find a proper vertex coloring or a proper edge coloring if we can use arbitrarily many colors; however, it is difficult to find an *optimal* coloring that uses the smallest possible number of colors.

On the other hand, domatic partitions are a maximization problem. It is trivial to find a domatic partition of size 1; however, it is difficult to find an *optimal* domatic partition with the largest possible number of disjoint dominating sets.

2.4 Factors and Factorizations

Let $G = (V, E)$ be a graph, let $X \subseteq E$ be a set of edges, and let $H = (U, X)$ be the subgraph of G induced by X. We say that X is a *d-factor* of G if $U = V$ and $\deg_H(v) = d$ for each $v \in V$.

Equivalently, X is a d-factor if X induces a spanning d-regular subgraph of G. Put otherwise, X is a d-factor if each node $v \in V$ is incident to exactly d edges of X.

A function $f : E \to \{1, 2, \ldots, k\}$ is a *d-factorization* of G if $f^{-1}(i)$ is a d-factor for each i. See Figure 2.7 for examples.

We make the following observations:

(a) A 1-factor is a maximum matching. If a 1-factor exists, a maximum matching is a 1-factor.

(b) A 1-factorization is an edge coloring.

(c) The subgraph induced by a 2-factor consists of disjoint cycles.

A 1-factor is also known as a *perfect matching*.

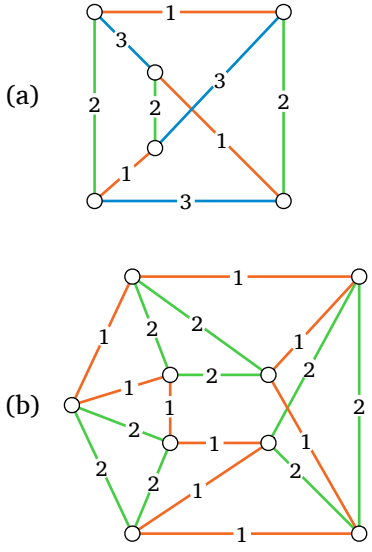


Figure 2.7: (a) A 1-factorization of a 3-regular graph. (b) A 2-factorization of a 4-regular graph.

2.5 Approximations

So far we have encountered a number of maximization problems and minimization problems. More formally, the definition of a maximization problem consists of two parts: a set of *feasible solutions* $\mathcal{S}$ and an *objective function* $g : \mathcal{S} \to \mathbb{R}$. In a maximization problem, the goal is to find a feasible solution $X \in \mathcal{S}$ that maximizes $g(X)$. A minimization problem is analogous: the goal is to find a feasible solution $X \in \mathcal{S}$ that minimizes $g(X)$.

For example, the problem of finding a maximum matching for a graph G is of this form. The set of feasible solutions $\mathcal{S}$ consists of all matchings in G, and we simply define $g(M) = |M|$ for each matching $M \in \mathcal{S}$.

As another example, the problem of finding an optimal coloring is a minimization problem. The set of feasible solutions $\mathcal{S}$ consists of all proper vertex colorings, and $g(f)$ is the number of colors in $f \in \mathcal{S}$.

Often, it is infeasible or impossible to find an optimal solution; hence we resort to approximations. Given a maximization problem $(\mathcal{S}, g)$, we say that a solution X is an *α-approximation* if $X \in \mathcal{S}$, and we have $\alpha g(X) \geq g(Y)$ for all $Y \in \mathcal{S}$. That is, X is a feasible solution, and the size of X is within factor α of the optimum.

Similarly, if $(\mathcal{S}, g)$ is a minimization problem, we say that a solution X is an α-approximation if $X \in \mathcal{S}$, and we have $g(X) \leq \alpha g(Y)$ for all $Y \in \mathcal{S}$. That is, X is a feasible solution, and the size of X is within factor α of the optimum.

Note that we follow the convention that the approximation ratio α is always at least 1, both in the case of minimization problems and maximization problems. Other conventions are also used in the literature.

2.6 Directed Graphs and Orientations

Unless otherwise mentioned, all graphs that we encounter are undirected. However, we will occasionally need to refer to so-called orientations,

and hence we need to introduce some terminology related to directed graphs.

A *directed graph* is a pair $G = (V, E)$, where V is the set of nodes and E is the set of *directed edges*. Each edge $e \in E$ is a pair of nodes, that is, $e = (u, v)$ where $u, v \in V$. Put otherwise, $E \subseteq V \times V$.

Intuitively, an edge (u, v) is an "arrow" that points from node u to node v; it is an *outgoing edge* for u and an *incoming edge* for v. The *outdegree* of a node $v \in V$, in notation $\text{outdegree}_G(v)$, is the number of outgoing edges, and the *indegree* of the node, $\text{indegree}_G(v)$, is the number of incoming edges.

Now let $G = (V, E)$ be a graph and let $H = (V, E')$ be a directed graph with the same set of nodes. We say that H is an *orientation* of G if the following holds:

(a) For each $\{u, v\} \in E$ we have either $(u, v) \in E'$ or $(v, u) \in E'$, but not both.

(b) For each $(u, v) \in E'$ we have $\{u, v\} \in E$.

Put otherwise, in an orientation of G we have simply chosen an arbitrary direction for each undirected edge of G. It follows that

$$\text{indegree}_H(v) + \text{outdegree}_H(v) = \deg_G(v)$$

for all $v \in V$.

2.7 Quiz

Construct a simple undirected graph $G = (V, E)$ with the following property: If you take any set X that is a maximal independent set of G, then X is not a minimum dominating set of G.

Present the graph in the set formalism by listing the sets of nodes and edges. For example, a cycle on three nodes can be encoded as $V = \{1, 2, 3\}$ and $E = \{\{1, 2\}, \{2, 3\}, \{3, 1\}\}$.

2.8 Exercises

Exercise 2.1 (independence and vertex covers)**.** Let $I \subseteq V$ and define $C = V \setminus I$. Show that

(a) if I is an independent set then C is a vertex cover and vice versa,

(b) if I is a maximal independent set then C is a minimal vertex cover and vice versa,

(c) if I is a maximum independent set then C is a minimum vertex cover and vice versa,

(d) it is possible that C is a 2-approximation of minimum vertex cover but I is not a 2-approximation of maximum independent set,

(e) it is possible that I is a 2-approximation of maximum independent set but C is not a 2-approximation of minimum vertex cover.

Exercise 2.2 (matchings)**.** Show that

(a) any maximal matching is a 2-approximation of a maximum matching,

(b) any maximal matching is a 2-approximation of a minimum maximal matching,

(c) a maximal independent set is not necessarily a 2-approximation of maximum independent set,

(d) a maximal independent set is not necessarily a 2-approximation of minimum maximal independent set.

Exercise 2.3 (matchings and vertex covers)**.** Let M be a maximal matching, and let $C = \bigcup M$, i.e., C consists of all endpoints of matched edges. Show that

(a) C is a 2-approximation of a minimum vertex cover,

(b) C is not necessarily a 1.999-approximation of a minimum vertex cover.

Would you be able to improve the approximation ratio if M was a minimum maximal matching?

Exercise 2.4 (independence and domination)**.** Show that

(a) a maximal independent set is a minimal dominating set,

(b) a minimal dominating set is not necessarily a maximal independent set,

(c) a minimum maximal independent set is not necessarily a minimum dominating set.

Exercise 2.5 (graph colorings and partitions)**.** Show that

(a) a weak 2-coloring always exists,

(b) a domatic partition of size 2 does not necessarily exist,

(c) if a domatic partition of size 2 exists, then a weak 2-coloring is a domatic partition of size 2,

(d) a weak 2-coloring is not necessarily a domatic partition of size 2.

Show that there are 2-regular graphs with the following properties:

(e) any 3-coloring is a domatic partition of size 3,

(f) no 3-coloring is a domatic partition of size 3.

Assume that G is a graph of maximum degree Δ; show that

(g) there exists a $(\Delta + 1)$-coloring,

(h) a Δ-coloring does not necessarily exist.

Exercise 2.6 (isomorphism)**.** Construct non-empty 3-regular connected graphs G and H such that G and H have the same number of nodes and G and H are *not* isomorphic. Just giving a construction is not sufficient —you have to *prove* that G and H are not isomorphic.

⋆ **Exercise 2.7** (matchings and edge domination)**.** Show that

(a) a maximal matching is a minimal edge dominating set,

(b) a minimal edge dominating set is not necessarily a maximal matching,

(c) a minimum maximal matching is a minimum edge dominating set,

(d) any maximal matching is a 2-approximation of a minimum edge dominating set.

▷ *hint E*

⋆ **Exercise 2.8** (Petersen 1891)**.** Show that any $2d$-regular graph $G = (V, E)$ has an orientation $H = (V, E')$ such that

$$\text{indegree}_H(v) = \text{outdegree}_H(v) = d$$

for all $v \in V$. Show that any $2d$-regular graph has a 2-factorization.

Your proof cannot use any "well-known graph theoretic facts"; it has to be entirely self-contained. For example, you cannot assume properties about Eulerian cycles without proving them first.

2.9 Bibliographic Notes

The connection between maximal matchings and approximations of vertex covers (Exercise 2.3) is commonly attributed to Gavril and Yannakakis—see, e.g., Papadimitriou and Steiglitz [35]. The connection between minimum maximal matchings and minimum edge dominating sets (Exercise 2.7) is due to Allan and Laskar [2] and Yannakakis and Gavril [43]. Exercise 2.8 is a 120-year-old result due to Petersen [37]. The definition of a weak coloring is from Naor and Stockmeyer [33].

Diestel's book [16] is a good source for graph-theoretic background, and Vazirani's book [41] provides further information on approximation algorithms.

Part III
Models of Computing

Chapter 3

PN Model: Port Numbering

Now that we have introduced the essential graph-theoretic concepts, we are ready to define what a "distributed algorithm" is. In this chapter, we will study one variant of the theme: deterministic distributed algorithms in the "port-numbering model". We will use the abbreviation PN for the port-numbering model, and we will also use the term "PN-algorithm" to refer to deterministic distributed algorithms in the port-numbering model. For now, everything will be deterministic—randomized algorithms will be discussed in later chapters.

3.1 Introduction

The basic idea of the PN model is best explained through an example. Suppose that I claim the following:

- A is a deterministic distributed algorithm that finds a 2-approximation of a minimum vertex cover in the port-numbering model.

Or, in brief:

- A is a PN-algorithm for finding a 2-approximation of a minimum vertex cover.

Informally, this entails the following:

(a) We can take any simple undirected graph $G = (V, E)$.

(b) We can then put together a computer network N with the same structure as G. A node $v \in V$ corresponds to a computer in N, and an edge $\{u, v\} \in E$ corresponds to a communication link between the computers u and v.

(c) Communication takes place through communication ports. A node of degree d corresponds to a computer with d ports that are labeled with numbers $1, 2, \ldots, d$ in an arbitrary order.

(d) Each computer runs a copy of the same deterministic algorithm A. All nodes are identical; initially they know only their own degree (i.e., the number of communication ports).

(e) All computers are started simultaneously, and they follow algorithm A synchronously in parallel. In each synchronous communication round, all computers in parallel

   (1) send a message to each of their ports,

   (2) wait while the messages are propagated along the communication channels,

   (3) receive a message from each of their ports, and

   (4) update their own state.

(f) After each round, a computer can stop and announce its *local output*: in this case the local output is either 0 or 1.

(g) We require that all nodes eventually stop—the *running time* of the algorithm is the number of communication rounds it takes until all nodes have stopped.

(h) We require that

$$C = \{ v \in V : \text{computer } v \text{ produced output } 1 \}$$

is a feasible vertex cover for graph G, and its size is at most 2 times the size of a minimum vertex cover.

Sections 3.2 and 3.3 will formalize this idea.

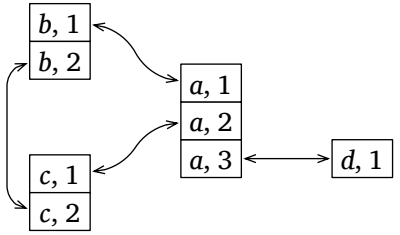


Figure 3.1: A port-numbered network $N = (V, P, p)$. There are four nodes, $V = \{a, b, c, d\}$; the degree of node a is 3, the degrees of nodes b and c are 2, and the degree of node d is 1. The connection function p is illustrated with arrows—for example, $p(a, 3) = (d, 1)$ and conversely $p(d, 1) = (a, 3)$. This network is simple.

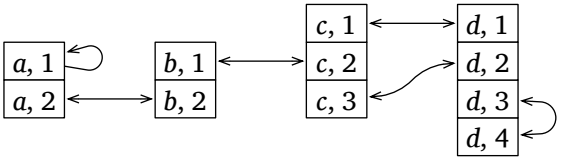


Figure 3.2: A port-numbered network $N = (V, P, p)$. There is a loop at node a, as $p(a, 1) = (a, 1)$, and another loop at node d, as $p(d, 3) = (d, 4)$. There are also multiple connections between c and d. Hence the network is not simple.

3.2 Port-Numbered Network

A *port-numbered network* is a triple $N = (V, P, p)$, where V is the set of *nodes*, P is the set of *ports*, and $p : P \to P$ is a function that specifies the *connections* between the ports. We make the following assumptions:

(a) Each port is a pair (v, i) where $v \in V$ and $i \in \{1, 2, \dots\}$.

(b) The connection function p is an involution, that is, for any port $x \in P$ we have $p(p(x)) = x$.

See Figures 3.1 and 3.2 for illustrations.

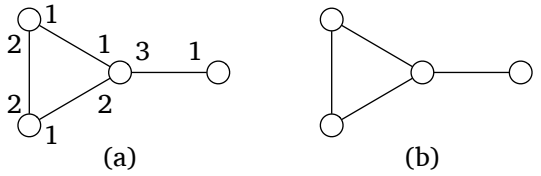


Figure 3.3: (a) An alternative drawing of the simple port-numbered network N from Figure 3.1. (b) The underlying graph G of N.

3.2.1 Terminology

If $(v,i) \in P$, we say that (v,i) is the port number i in node v. The *degree* $\deg_N(v)$ of a node $v \in V$ is the number of ports in v, that is, $\deg_N(v) = |\{\, i \in \mathbb{N} : (v,i) \in P \,\}|$.

Unless otherwise mentioned, we assume that the port numbers are *consecutive*: for each $v \in V$ there are ports $(v,1),(v,2),\ldots,(v,\deg_N(v))$ in P.

We use the shorthand notation $p(v,i)$ for $p((v,i))$. If $p(u,i) = (v,j)$, we say that port (u,i) is *connected* to port (v,j); we also say that port (u,i) is connected to node v, and that node u is connected to node v.

If $p(v,i) = (v,j)$ for some j, we say that there is a *loop* at v—note that we may have $i = j$ or $i \neq j$. If $p(u,i_1) = (v,j_1)$ and $p(u,i_2) = (v,j_2)$ for some $u \neq v$, $i_1 \neq i_2$, and $j_1 \neq j_2$, we say that there are *multiple connections* between u and v. A port-numbered network $N = (V,P,p)$ is *simple* if there are no loops or multiple connections.

3.2.2 Underlying Graph

For a simple port-numbered network $N = (V,P,p)$ we define the *underlying graph* $G = (V,E)$ as follows: $\{u,v\} \in E$ if and only if u is connected to v in network N. Observe that $\deg_G(v) = \deg_N(v)$ for all $v \in V$. See Figure 3.3 for an illustration.

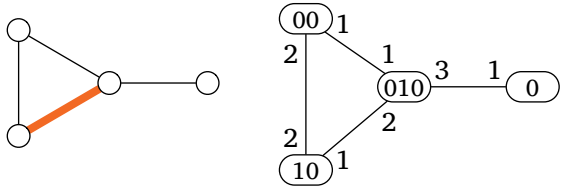


Figure 3.4: (a) A graph $G = (V, E)$ and a matching $M \subseteq E$. (b) A port-numbered network N; graph G is the underlying graph of N. The node labeling $f : V \to \{0,1\}^*$ is an encoding of matching M.

3.2.3 Encoding Input and Output

In a distributed system, nodes are the active elements: they can read input and produce output. Hence we will heavily rely on *node labelings*: we can directly associate information with each node $v \in V$.

Assume that $N = (V, P, p)$ is a simple port-numbered network, and $G = (V, E)$ is the underlying graph of N. We show that a node labeling $f : V \to Y$ can be used to represent the following graph-theoretic structures; see Figure 3.4 for an illustration.

Node labeling $g : V \to X$**.** Trivial: we can choose $Y = X$ and $f = g$.

Subset of nodes $X \subseteq V$**.** We can interpret a subset of nodes as a node labeling $g : V \to \{0,1\}$, where g is the indicator function of set X. That is, $g(v) = 1$ iff $v \in X$.

Edge labeling $g : E \to X$**.** For each node v, its label $f(v)$ encodes the values $g(e)$ for all edges e incident to v, in the order of increasing port numbers. More precisely, if v is a node of degree d, its label is a vector $f(v) \in X^d$. If $(v, j) \in P$ and $p(v, j) = (u, i)$, then element j of vector $f(v)$ is $g(\{u, v\})$.

Subset of edges $X \subseteq E$**.** We can interpret a subset of edges as an edge labeling $g : E \to \{0,1\}$.

Orientation $H = (V, E')$**.** For each node v, its label $f(v)$ indicates which of the edges incident to v are outgoing edges, in the order of increasing port numbers.

It is trivial to compose the labelings. For example, we can easily construct a node labeling that encodes both a subset of nodes and a subset of edges.

3.2.4 Distributed Graph Problems

A *distributed graph problem* Π associates a set of solutions $\Pi(N)$ with each simple port-numbered network $N = (V, P, p)$. A *solution* $f \in \Pi(N)$ is a node labeling $f : V \to Y$ for some set Y of *local outputs*.

Using the encodings of Section 3.2.3, we can interpret all of the following as distributed graph problems: independent sets, vertex covers, dominating sets, matchings, edge covers, edge dominating sets, colorings, edge colorings, domatic partitions, edge domatic partitions, factors, factorizations, orientations, and any combinations of these.

To make the idea more clear, we will give some more detailed examples.

(a) *Vertex cover*: $f \in \Pi(N)$ if f encodes a vertex cover of the underlying graph of N.

(b) *Minimal vertex cover*: $f \in \Pi(N)$ if f encodes a minimal vertex cover of the underlying graph of N.

(c) *Minimum vertex cover*: $f \in \Pi(N)$ if f encodes a minimum vertex cover of the underlying graph of N.

(d) 2*-approximation of minimum vertex cover*: $f \in \Pi(N)$ if f encodes a vertex cover C of the underlying graph of N; moreover, the size of C is at most two times the size of a minimum vertex cover.

(e) *Orientation*: $f \in \Pi(N)$ if f encodes an orientation of the underlying graph of N.

(f) 2-*coloring*: $f \in \Pi(N)$ if f encodes a 2-coloring of the underlying graph of N. Note that we will have $\Pi(N) = \varnothing$ if the underlying graph of N is not bipartite.

3.3 Distributed Algorithms in the Port-Numbering Model

We will now give a formal definition of a distributed algorithm in the port-numbering model. In essence, a distributed algorithm is a state machine (not necessarily a finite-state machine). To run the algorithm on a certain port-numbered network, we put a copy of the same state machine at each node of the network.

The formal definition of a distributed algorithm plays a similar role as the definition of a Turing machine in the study of non-distributed algorithms. A formally rigorous foundation is necessary to study questions such as computability and computational complexity. However, we do not usually present algorithms as Turing machines, and the same is the case here. Once we become more familiar with distributed algorithms, we will use higher-level pseudocode to define algorithms and omit the tedious details of translating the high-level description into a state machine.

3.3.1 State Machine

A distributed algorithm A is a state machine that consists of the following components:

(i) Input_A is the set of *local inputs*,

(ii) States_A is the set of states,

(iii) $\text{Output}_A \subseteq \text{States}_A$ is the set of stopping states (*local outputs*),

(iv) Msg_A is the set of possible messages.

Moreover, for each possible degree $d \in \mathbb{N}$ we have the following functions:

(v) $\text{init}_{A,d} \colon \text{Input}_A \to \text{States}_A$ initializes the state machine,

(vi) $\text{send}_{A,d} \colon \text{States}_A \to \text{Msg}_A^d$ constructs outgoing messages,

(vii) $\text{receive}_{A,d} \colon \text{States}_A \times \text{Msg}_A^d \to \text{States}_A$ processes incoming messages.

We require that $\text{receive}_{A,d}(x, y) = x$ whenever $x \in \text{Output}_A$. The idea is that a node that has already stopped and printed its local output no longer changes its state.

3.3.2 Execution

Let A be a distributed algorithm, let $N = (V, P, p)$ be a port-numbered network, and let $f : V \to \text{Input}_A$ be a labeling of the nodes. A *state vector* is a function $x \colon V \to \text{States}_A$. The *execution* of A on (N, f) is a sequence of state vectors $x_0, x_1, \ldots$ defined recursively as follows.

The initial state vector x_0 is defined by

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

where $u \in V$ and $d = \deg_N(u)$.

Now assume that we have defined state vector x_{t-1}. Define $m_t : P \to \text{Msg}_A$ as follows. Assume that $(u, i) \in P$, $(v, j) = p(u, i)$, and $\deg_N(v) = \ell$. Let $m_t(u, i)$ be component j of the vector $\text{send}_{A,\ell}(x_{t-1}(v))$.

Intuitively, $m_t(u, i)$ is the message received by node u from port number i on round t. Equivalently, it is the message sent by node v to port number j on round t—recall that ports (u, i) and (v, j) are connected.

For each node $u \in V$ with $d = \deg_N(u)$, we define the message vector

$$m_t(u) = \big(m_t(u, 1), m_t(u, 2), \ldots, m_t(u, d)\big).$$

Finally, we define the new state vector x_t by

$$x_t(u) = \text{receive}_{A,d}\big(x_{t-1}(u), m_t(u)\big).$$

We say that algorithm A *stops in time* T if $x_T(u) \in \text{Output}_A$ for each $u \in V$. We say that A *stops* if A stops in time T for some finite T. If A stops in time T, we say that $g = x_T$ is the *output* of A, and $x_T(u)$ is the *local output* of node u.

3.3.3 Solving Graph Problems

Now we will define precisely what it means if we say that a distributed algorithm A solves a certain graph problem.

Let $\mathscr{F}$ be a family of simple undirected graphs. Let Π and Π' be distributed graph problems (see Section 3.2.4). We say that *distributed algorithm A solves problem Π on graph family $\mathscr{F}$ given Π'* if the following holds: assuming that

(a) $N = (V, P, p)$ is a simple port-numbered network,
(b) the underlying graph of N is in $\mathscr{F}$, and
(c) the input f is in $\Pi'(N)$,

the execution of algorithm A on (N, f) stops and produces an output $g \in \Pi(N)$. If A stops in time $T(|V|)$ for some function $T : \mathbb{N} \to \mathbb{N}$, we say that A solves the problem *in time* T.

Obviously, A has to be compatible with the encodings of Π and Π'. That is, each $f \in \Pi'(N)$ has to be a function of the form $f : V \to \text{Input}_A$, and each $g \in \Pi(N)$ has to be a function of the form $g : V \to \text{Output}_A$.

Problem Π' is often omitted. If A does not need the input f, we simply say that *A solves problem Π on graph family $\mathscr{F}$*. More precisely, in this case we provide a trivial input $f(v) = 0$ for each $v \in V$.

In practice, we will often specify $\mathscr{F}$, Π, Π', and T implicitly. Here are some examples of common parlance:

(a) *Algorithm A finds a maximum matching in any path graph*: here $\mathscr{F}$ consists of all path graphs; Π' is omitted; and Π is the problem of finding a maximum matching.

(b) *Algorithm A finds a maximal independent set in k-colored graphs in time k*: here $\mathscr{F}$ consists of all graphs that admit a k-coloring;

Π' is the problem of finding a k-coloring; Π is the problem of finding a maximal independent set; and T is the constant function $T : n \mapsto k$.

3.4 Example: Coloring Paths

Recall the fast 3-coloring algorithm for paths from Section 1.3. We will now present the algorithm in a formally precise manner as a state machine. Let us start with the problem definition:

- $\mathscr{F}$ is the family of path graphs.
- Π is the problem of coloring graphs with 3 colors.
- Π' is the problem of coloring graphs with any number of colors.

We will present algorithm A that solves problem Π on graph family $\mathscr{F}$ given Π'. Note that in Section 1.3 we assumed that we have unique identifiers, but it is sufficient to assume that we have some graph coloring, i.e., a solution to problem Π'.

The set of local inputs is determined by what we assume as input:

$$\text{Input}_A = \mathbb{Z}^+.$$

The set of stopping states is determined by the problem that we are trying to solve:

$$\text{Output}_A = \{1, 2, 3\}.$$

In our algorithm, each node only needs to store one positive integer (the current color):

$$\text{States}_A = \mathbb{Z}^+.$$

Messages are also integers:

$$\text{Msg}_A = \mathbb{Z}^+.$$

Initialization is trivial: the initial state of a node is its color. Hence for all d we have

$$\text{init}_{A,d}(x) = x.$$

In each step, each node sends its current color to each of its neighbors. As we assume that all nodes have degree at most 2, we only need to define $\text{send}_{A,d}$ for $d \leq 2$:

$$\begin{aligned}\text{send}_{A,0}(x) &= ().\\ \text{send}_{A,1}(x) &= (x).\\ \text{send}_{A,2}(x) &= (x,x).\end{aligned}$$

The nontrivial part of the algorithm is hidden in the receive function. To define it, we will use the following auxiliary function that returns the smallest positive number not in X:

$$g(X) = \min(\mathbb{Z}^+ \setminus X).$$

Again, we only need to define $\text{receive}_{A,d}$ for degrees $d \leq 2$:

$$\text{receive}_{A,0}(x,()) = \begin{cases} g(\varnothing) & \text{if } x \notin \{1,2,3\},\\ x & \text{otherwise.}\end{cases}$$

$$\text{receive}_{A,1}(x,(y)) = \begin{cases} g(\{y\}) & \text{if } x \notin \{1,2,3\}\\ & \text{and } x > y,\\ x & \text{otherwise.}\end{cases}$$

$$\text{receive}_{A,2}(x,(y,z)) = \begin{cases} g(\{y,z\}) & \text{if } x \notin \{1,2,3\}\\ & \text{and } x > y,\ x > z,\\ x & \text{otherwise.}\end{cases}$$

This algorithm does precisely the same thing as the algorithm that was described in pseudocode in Table 1.1. It can be verified that this algorithm indeed solves problem Π on graph family $\mathscr{F}$ given Π', in the sense that we defined in Section 3.3.3.

We will not usually present distributed algorithms in the low-level state-machine formalism. Typically we are happy with a higher-level presentation (e.g., in pseudocode), but it is important to understand that any distributed algorithm can be always translated into the state machine formalism.

In the next two sections we will give some non-trivial examples of PN-algorithms. We will give informal descriptions of the algorithms; in the exercises we will see how to translate these algorithms into the state machine formalism.

3.5 Example: Maximal Matching in Two-Colored Graphs

In this section we present a distributed *bipartite maximal matching* algorithm: it finds a maximal matching in 2-colored graphs. That is, $\mathscr{F}$ is the family of bipartite graphs, we are given a 2-coloring $f : V \to \{1, 2\}$, and the algorithm will output an encoding of a maximal matching $M \subseteq E$.

3.5.1 Algorithm

In what follows, we say that a node $v \in V$ is *white* if $f(v) = 1$, and it is *black* if $f(v) = 2$. During the execution of the algorithm, each node is in one of the states

$$\{\mathsf{UR}, \mathsf{MR}(i), \mathsf{US}, \mathsf{MS}(i)\},$$

which stand for "unmatched and running", "matched and running", "unmatched and stopped", and "matched and stopped", respectively. As the names suggest, US and $\mathsf{MS}(i)$ are stopping states. If the state of a node v is $\mathsf{MS}(i)$ then v is matched with the neighbor that is connected to port i.

Initially, all nodes are in state UR. Each black node v maintains variables $M(v)$ and $X(v)$, which are initialized

$$M(v) \leftarrow \varnothing, \quad X(v) \leftarrow \{1, 2, \ldots, \deg(v)\}.$$

The algorithm is presented in Table 3.1; see Figure 3.5 for an illustration.

3.5.2 Analysis

The following invariant is useful in order to analyze the algorithm.

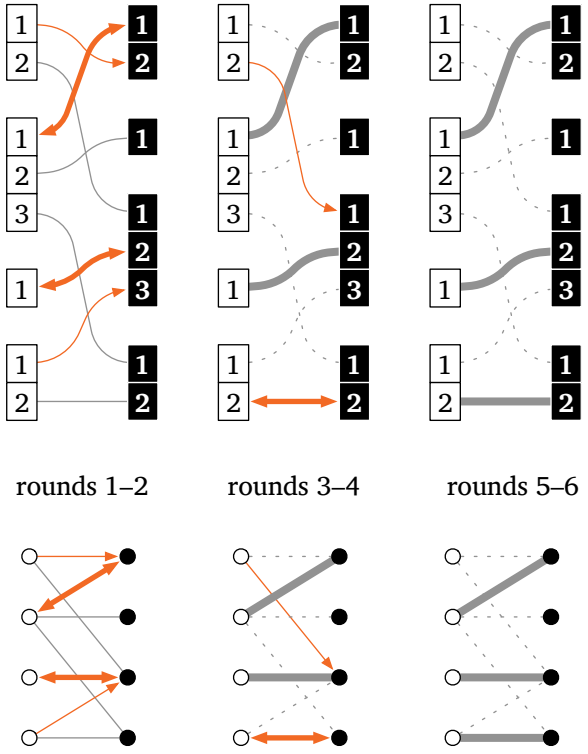


Figure 3.5: The bipartite maximal matching algorithm; the illustration shows the algorithm both from the perspective of the port-numbered network N and from the perspective of the underlying graph G. Arrows pointing right are proposals, and arrows pointing left are acceptances. Wide gray edges have been added to matching M.

Round $2k-1$*, white nodes:*

- State UR, $k \leq \deg_N(v)$: Send '*proposal*' to port (v,k).
- State UR, $k > \deg_N(v)$: Switch to state US.
- State MR(i): Send '*matched*' to all ports.
  Switch to state MS(i).

Round $2k-1$*, black nodes:*

- State UR: Read incoming messages.
  If we receive '*matched*' from port i, remove i from $X(v)$.
  If we receive '*proposal*' from port i, add i to $M(v)$.

Round $2k$*, black nodes:*

- State UR, $M(v) \neq \varnothing$: Let $i = \min M(v)$.
  Send '*accept*' to port (v,i). Switch to state MS(i).
- State UR, $X(v) = \varnothing$: Switch to state US.

Round $2k$*, white nodes:*

- State UR: Process incoming messages.
  If we receive '*accept*' from port i, switch to state MR(i).

Table 3.1: The bipartite maximal matching algorithm; here $k = 1, 2, \ldots$.

Lemma 3.1. *Assume that u is a white node, v is a black node, and $(u,i) = p(v,j)$. Then at least one of the following holds:*

(a) element j is removed from $X(v)$ before round $2i$,
(b) at least one element is added to $M(v)$ before round $2i$.

Proof. Assume that we still have $M(v) = \varnothing$ and $j \in X(v)$ after round $2i-2$. This implies that v is still in state UR, and u has not sent '*matched*' to v. In particular, u is in state UR or MR(i) after round $2i-2$. In the former case, u sends '*proposal*' to v on round $2i-1$, and j is added to $M(v)$ on round $2i-1$. In the latter case, u sends '*matched*' to v on round $2i-1$, and j is removed from $X(v)$ on round $2i-1$. □

Now it is easy to verify that the algorithm actually makes some progress and eventually halts.

Lemma 3.2. *The bipartite maximal matching algorithm stops in time $2\Delta+1$, where Δ is the maximum degree of N.*

Proof. A white node of degree d stops before or during round $2d+1 \leq 2\Delta+1$.

Now let us consider a black node v. Assume that we still have $j \in X(v)$ on round 2Δ. Let $(u,i) = p(v,j)$; note that $i \leq \Delta$. By Lemma 3.1, at least one element has been added to $M(v)$ before round 2Δ. In particular, v stops before or during round 2Δ. □

Moreover, the output is correct.

Lemma 3.3. *The bipartite maximal matching algorithm finds a maximal matching in any two-colored graph.*

Proof. Let us first verify that the output correctly encodes a matching. In particular, assume that u is a white node, v is a black node, and $p(u,i) = (v,j)$. We have to prove that u stops in state MS(i) if and only if v stops in state MS(j). If u stops in state MS(i), it has received an '*accept*' from v, and v stops in state MS(j). Conversely, if v stops in state MS(j), it has received a '*proposal*' from u and it sends an '*accept*' to u, after which u stops in state MS(i).

Let us then verify that M is indeed maximal. If this was not the case, there would be an unmatched white node u that is connected to an unmatched black node v. However, Lemma 3.1 implies that at least one of them becomes matched before or during round 2Δ. □

3.6 Example: Vertex Covers

We will now give a distributed *minimum vertex cover* 3*-approximation* algorithm; we will use the bipartite maximal matching algorithm from the previous section as a building block.

So far we have seen algorithms that assume something about the input (e.g., we are given a proper coloring of the network). The algorithm that we will see in this section makes no such assumptions. We can run the minimum vertex cover 3-approximation algorithm in any port-numbered network, without any additional input. In particular, we do not need any kind of coloring, unique identifiers, or randomness.

3.6.1 Virtual 2-Colored Network

Let $N = (V, P, p)$ be a port-numbered network. We will construct another port-numbered network $N' = (V', P', p')$ as follows; see Figure 3.6 for an illustration. First, we double the number of nodes—for each node $v \in V$ we have two nodes v_1 and v_2 in V':

$$V' = \{ v_1, v_2 : v \in V \},$$
$$P' = \{ (v_1, i),\ (v_2, i) : (v, i) \in P \}.$$

Then we define the connections. If $p(u, i) = (v, j)$, we set

$$p'(u_1, i) = (v_2, j),$$
$$p'(u_2, i) = (v_1, j).$$

With these definitions we have constructed a network N' such that the underlying graph $G' = (V', E')$ is bipartite. We can define a 2-coloring

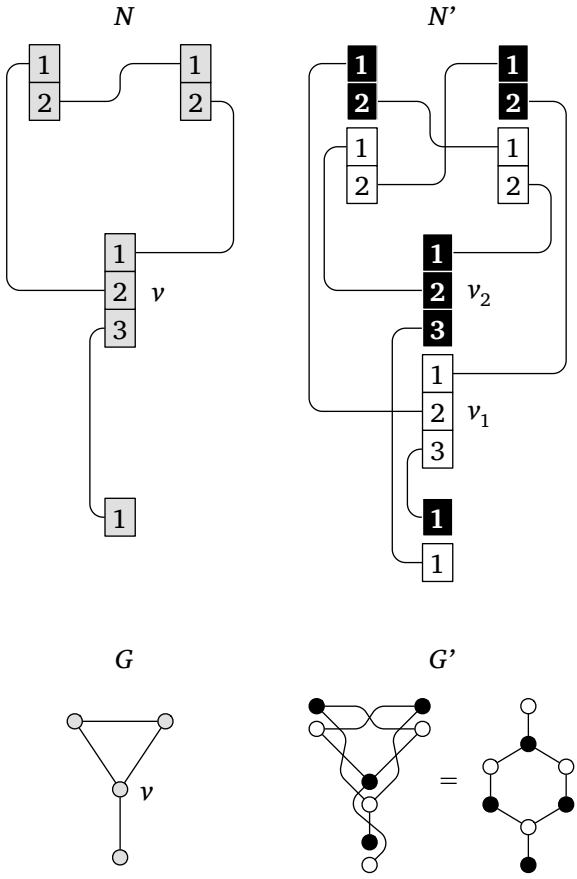


Figure 3.6: Construction of the virtual network N' in the minimum vertex cover 3-approximation algorithm.

$f' : V' \to \{1, 2\}$ as follows:

$$f'(v_1) = 1 \text{ and } f'(v_2) = 2 \text{ for each } v \in V.$$

Nodes of color 1 are called *white* and nodes of color 2 are called *black*.

3.6.2 Simulation of the Virtual Network

Now N is our physical communication network, and N' is merely a mathematical construction. However, the key observation is that we can use the physical network N to efficiently *simulate* the execution of any distributed algorithm A on (N', f'). Each physical node $v \in V$ simulates nodes v_1 and v_2 in N':

(a) If v_1 sends a message m_1 to port (v_1, i) and v_2 sends a message m_2 to port (v_2, i) in the simulation, then v sends the pair (m_1, m_2) to port (v, i) in the physical network.

(b) If v receives a pair (m_1, m_2) from port (v, i) in the physical network, then v_1 receives message m_2 from port (v_1, i) in the simulation, and v_2 receives message m_1 from port (v_2, i) in the simulation.

Note that we have here reversed the messages: what came from a white node is received by a black node and vice versa.

In particular, we can take the bipartite maximal matching algorithm of Section 3.5 and use the network N to simulate it on (N', f'). Note that network N is not necessarily bipartite and we do not have any coloring of N; hence we would not be able to apply the bipartite maximal matching algorithm on N.

3.6.3 Algorithm

Now we are ready to present the minimum vertex cover 3-approximation algorithm:

(a) Simulate the bipartite maximal matching algorithm in the virtual network N'. Each node v waits until both of its copies, v_1 and v_2, have stopped.

(b) Node v outputs 1 if at least one of its copies v_1 or v_2 becomes matched.

3.6.4 Analysis

Clearly the minimum vertex cover 3-approximation algorithm stops, as the bipartite maximal matching algorithm stops. Moreover, the running time is $2\Delta + 1$ rounds, where Δ is the maximum degree of N.

Let us now prove that the output is correct. To this end, let $G = (V, E)$ be the underlying graph of N, and let $G' = (V', E')$ be the underlying graph of N'. The bipartite maximal matching algorithm outputs a maximal matching $M' \subseteq E'$ for G'. Define the edge set $M \subseteq E$ as follows:

$$M = \left\{ \{u, v\} \in E : \{u_1, v_2\} \in M' \text{ or } \{u_2, v_1\} \in M' \right\}. \tag{3.1}$$

See Figure 3.7 for an illustration. Furthermore, let $C' \subseteq V'$ be the set of nodes that are incident to an edge of M' in G', and let $C \subseteq V$ be the set of nodes that are incident to an edge of M in G; equivalently, C is the set of nodes that output 1. We make the following observations.

(a) Each node of C' is incident to precisely one edge of M'.
(b) Each node of C is incident to one or two edges of M.
(c) Each edge of E' is incident to at least one node of C'.
(d) Each edge of E is incident to at least one node of C.

We are now ready to prove the main result of this section.

Lemma 3.4. *Set C is a 3-approximation of a minimum vertex cover of G.*

Proof. First, observation (d) above already shows that C is a vertex cover of G.

To analyze the approximation ratio, let $C^* \subseteq V$ be a vertex cover of G. By definition each edge of E is incident to at least one node of C^*; in particular, each edge of M is incident to a node of C^*. Therefore $C^* \cap C$ is a vertex cover of the subgraph $H = (C, M)$.

By observation (b) above, graph H has a maximum degree of at most 2. Set C consists of all nodes in H. We will then argue that any

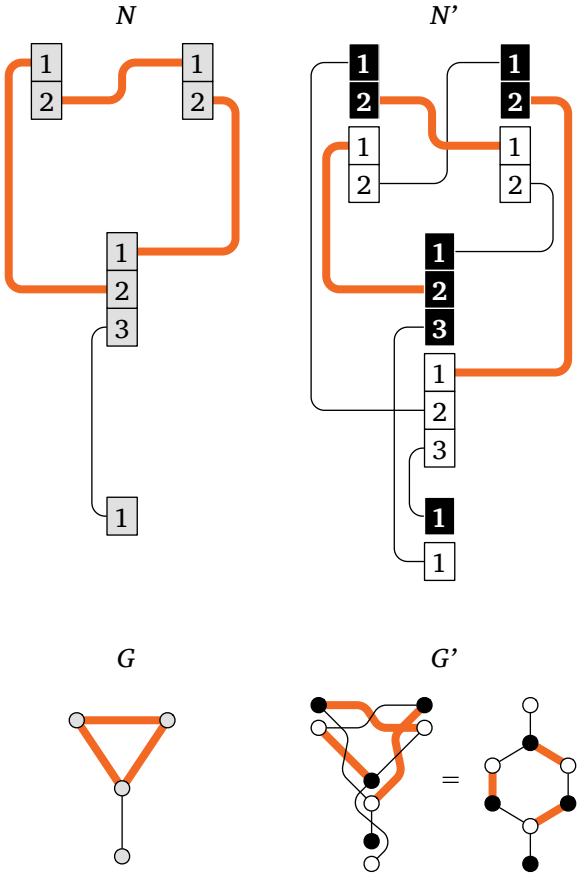


Figure 3.7: Set $M \subseteq E$ (left) and matching $M' \subseteq E'$ (right).

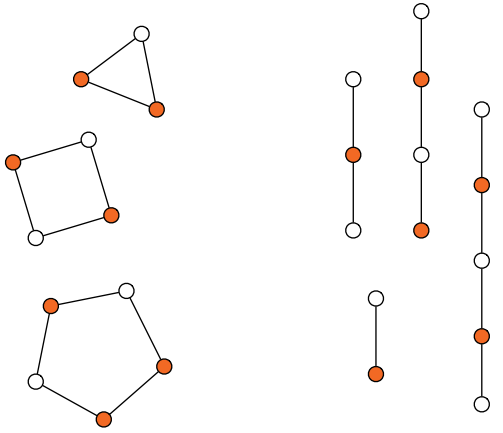


Figure 3.8: (a) In a cycle with n nodes, any vertex cover contains at least $n/2$ nodes. (b) In a path with n nodes, any vertex cover contains at least $n/3$ nodes.

vertex cover C^* contains at least a fraction $1/3$ of the nodes in H; see Figure 3.8 for an example. Then it follows that C is at most 3 times as large as a minimum vertex cover.

To this end, let $H_i = (C_i, M_i)$, $i = 1, 2, \ldots, k$, be the connected components of H; each component is either a path or a cycle. Now $C_i^* = C^* \cap C_i$ is a vertex cover of H_i.

A node of C_i^* is incident to at most two edges of M_i. Therefore

$$|C_i^*| \geq |M_i|/2.$$

If H_i is a cycle, we have $|C_i| = |M_i|$ and

$$|C_i^*| \geq |C_i|/2.$$

If H_i is a path, we have $|M_i| = |C_i| - 1$. If $|C_i| \geq 3$, it follows that

$$|C_i^*| \geq |C_i|/3.$$

The only remaining case is a path with two nodes, in which case trivially $|C_i^*| \geq |C_i|/2$.

In conclusion, we have $|C_i^*| \geq |C_i|/3$ for each component H_i. It follows that

$$|C^*| \geq |C^* \cap C| = \sum_{i=1}^{k} |C_i^*| \geq \sum_{i=1}^{k} |C_i|/3 = |C|/3. \qquad \square$$

In summary, the minimum vertex cover algorithm finds a 3-approximation of a minimum vertex cover in any graph G. Moreover, if the maximum degree of G is small, the algorithm is fast: we only need $O(\Delta)$ rounds in a network of maximum degree Δ.

3.7 Quiz

Construct a simple port-numbered network $N = (V, P, p)$ and its underlying graph $G = (V, E)$ that has *as few nodes as possible* and that satisfies the following properties:

- We have $E \neq \emptyset$.
- The set $M = \big\{\{u, v\} \in E : p(u, 1) = (v, 2)\big\}$ is a perfect matching in graph G.

Please answer by listing all elements of sets V, E, and P, and by listing all values of p. For example, you might specify a network with two nodes as follows: $V = \{1, 2\}$, $E = \{\{1, 2\}\}$, $P = \{(1, 1), (2, 1)\}$, $p(1, 1) = (2, 1)$, and $p(2, 1) = (1, 1)$.

3.8 Exercises

Exercise 3.1 (formalizing bipartite maximal matching)**.** Present the bipartite maximal matching algorithm from Section 3.5 in a formally precise manner, using the definitions of Section 3.3. Try to make Msg_A as small as possible.

Exercise 3.2 (formalizing vertex cover approximation)**.** Present the minimum vertex cover 3-approximation algorithm from Section 3.6 in a formally precise manner, using the definitions of Section 3.3. Try to make both Msg_A and States_A as small as possible.

▷ *hint F*

Exercise 3.3 (stopped nodes)**.** In the formalism of this chapter, a node that stops will repeatedly send messages to its neighbors. Show that this detail is irrelevant, and we can always rewrite algorithms so that such messages are ignored. Put otherwise, a node that stops can also stop sending messages.

More precisely, assume that A is a distributed algorithm that solves problem Π on family $\mathscr{F}$ given Π' in time T. Show that there is another algorithm A' such that (i) A' solves problem Π on family $\mathscr{F}$ given Π' in time $T + O(1)$, and (ii) in A' the state transitions never depend on the messages that are sent by nodes that have stopped.

Exercise 3.4 (more than two colors)**.** Design a distributed algorithm that finds a maximal matching in k-colored graphs. You can assume that k is a known constant.

Exercise 3.5 (analysis of vertex cover approximation)**.** Is the analysis of the minimum vertex cover 3-approximation algorithm tight? That is, is it possible to construct a network N such that the algorithm outputs a vertex cover that is exactly 3 times as large as the minimum vertex cover of the underlying graph of N?

⋆ **Exercise 3.6** (implementation)**.** Using your favorite programming language, implement a simulator that lets you play with distributed algorithms in the port-numbering model. Implement the algorithms for bipartite maximal matching and minimum vertex cover 3-approximation and try them out in the simulator.

⋆ **Exercise 3.7** (composition)**.** Assume that algorithm A_1 solves problem Π_1 on family $\mathscr{F}$ given Π_0 in time T_1, and algorithm A_2 solves problem Π_2 on family $\mathscr{F}$ given Π_1 in time T_2.

Is it always possible to design an algorithm A that solves problem Π_2 on family $\mathscr{F}$ given Π_0 in time $O(T_1 + T_2)$?

▷ *hint G*

3.9 Bibliographic Notes

The concept of a port numbering is from Angluin's [3] work. The bipartite maximal matching algorithm is due to Hańćkowiak et al. [23], and the minimum vertex cover 3-approximation algorithm is from a paper with Polishchuk [38].

Chapter 4

LOCAL Model: Unique Identifiers

In the previous chapter, we studied deterministic distributed algorithms in port-numbered networks. In this chapter we will study a stronger model: *networks with unique identifiers*—see Figure 4.1. Following the standard terminology of the field, we will use the term "LOCAL model" to refer to networks with unique identifiers.

4.1 Definitions

Throughout this chapter, fix a constant $c > 1$. An assignment of *unique identifiers* for a port-numbered network $N = (V, P, p)$ is an injection

$$\mathrm{id} \colon V \to \{1, 2, \ldots, |V|^c\}.$$

That is, each node $v \in V$ is labeled with a unique integer, and the labels are assumed to be relatively small.

Formally, unique identifiers can be interpreted as a graph problem Π', where each solution $\mathrm{id} \in \Pi'(N)$ is an assignment of unique identifiers for network N. If a distributed algorithm A solves a problem Π on a family $\mathscr{F}$ given Π', we say that *A solves Π on $\mathscr{F}$ given unique identifiers*, or equivalently, *A solves Π on $\mathscr{F}$ in the* LOCAL *model*.

For the sake of convenience, when we discuss networks with unique identifiers, we will identify a node with its unique identifier, i.e., $v = \mathrm{id}(v)$ for all $v \in V$.

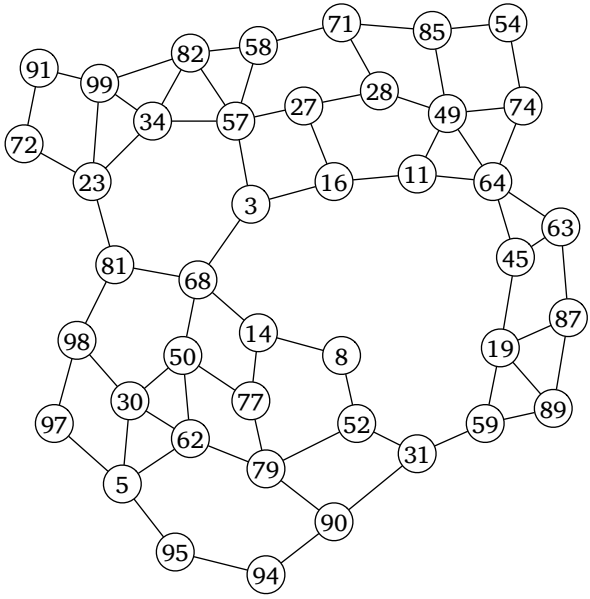


Figure 4.1: A network with unique identifiers.

4.2 Gathering Everything

In the LOCAL model, if the underlying graph $G = (V, E)$ is connected, all nodes can learn everything about G in time $O(\mathrm{diam}(G))$. In this section, we will present a gathering algorithm that accomplishes this.

In the gathering algorithm, each node $v \in V$ will construct sets $V(v, r)$ and $E(v, r)$, where $r = 1, 2, \ldots$. For all $v \in V$ and $r \geq 1$, these sets will satisfy

$$V(v, r) = \mathrm{ball}_G(v, r), \tag{4.1}$$

$$E(v, r) = \big\{\{s, t\} \in E : s \in \mathrm{ball}_G(v, r),\ t \in \mathrm{ball}_G(v, r-1)\big\}. \tag{4.2}$$

Now define the graph

$$G(v, r) = (V(v, r), E(v, r)). \tag{4.3}$$

See Figure 4.2 for an illustration.

The following properties are straightforward corollaries of (4.1)–(4.3).

(a) Graph $G(v, r)$ is a subgraph of $G(v, r + 1)$, which is a subgraph of G.

(b) If G is a connected graph, and $r \geq \mathrm{diam}(G) + 1$, we have $G(v, r) = G$.

(c) If G_v is the connected component of G that contains v, and $r \geq \mathrm{diam}(G_v) + 1$, we have $G(v, r) = G_v$.

(d) For a sufficiently large r, we have $G(v, r) = G(v, r + 1)$.

(e) If $G(v, r) = G(v, r + 1)$, we will also have $G(v, r + 1) = G(v, r + 2)$.

(f) Graph $G(v, r)$ for $r > 1$ can be constructed recursively as follows:

$$V(v, r) = \bigcup_{u \in V(v,1)} V(u, r - 1), \tag{4.4}$$

$$E(v, r) = \bigcup_{u \in V(v,1)} E(u, r - 1). \tag{4.5}$$

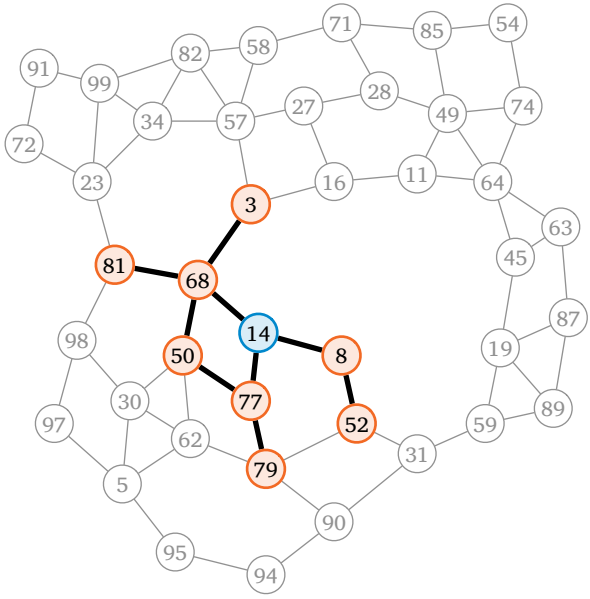


Figure 4.2: Subgraph $G(v, r)$ defined in (4.3), for $v = 14$ and $r = 2$.

The gathering algorithm maintains the following invariant: after round $r \geq 1$, each node $v \in V$ has constructed graph $G(v, r)$. The execution of the algorithm proceeds as follows:

(a) In round 1, each node $u \in V$ sends its identity u to each of its ports. Hence after round 1, each node $v \in V$ knows its own identity and the identities of its neighbors. Put otherwise, v knows precisely $G(v, 1)$.

(b) In round $r > 1$, each node $u \in V$ sends $G(u, r-1)$ to each of its ports. Hence after round r, each node $v \in V$ knows $G(u, r-1)$ for all $u \in V(v, 1)$. Now v can reconstruct $G(v, r)$ using (4.4) and (4.5).

(c) A node $v \in V$ can stop once it detects that the graph $G(v, r)$ no longer changes.

It is easy to extend the gathering algorithm so that we can discover not only the underlying graph $G = (V, E)$ but also the original port-numbered network $N = (V, P, p)$.

4.3 Solving Everything

Let $\mathscr{F}$ be a family of connected graphs, and let Π be a distributed graph problem. Assume that there is a deterministic *centralized* (non-distributed) algorithm A' that solves Π on $\mathscr{F}$. For example, A' can be a simple brute-force algorithm—we are not interested in the running time of algorithm A'.

Now there is a simple distributed algorithm A that solves Π on $\mathscr{F}$ in the LOCAL model. Let $N = (V, P, p)$ be a port-numbered network with the underlying graph $G \in \mathscr{F}$. Algorithm A proceeds as follows.

(a) All nodes discover N using the gathering algorithm from Section 4.2.

(b) All nodes use the centralized algorithm A' to find a solution $f \in \Pi(N)$. From the perspective of algorithm A, this is merely a state transition; it is a local step that requires no communication at all, and hence takes 0 communication rounds.

(c) Finally, each node $v \in V$ switches to state $f(v)$ and stops.

Clearly, the running time of the algorithm is $O(\mathrm{diam}(G))$.

It is essential that all nodes have the same canonical representation of network N (for example, V, P, and p are represented as lists that are ordered lexicographically by node identifiers and port numbers), and that all nodes use the same deterministic algorithm A' to solve Π. This way we are guaranteed that all nodes have locally computed the *same* solution f, and hence the outputs $f(v)$ are globally consistent.

4.4 Focus on Computational Complexity

So far we have learned the key difference between PN and LOCAL models: while there are plenty of graph problems that cannot be solved at all in the PN model, we know that all computable graph problems can be easily solved in the LOCAL model.

Hence our focus shifts from computability to computational complexity. While it is trivial to determine if a problem can be solved in the LOCAL model, we would like to know which problems can be solved quickly. In particular, we would like to learn which problems can be solved in time that is much smaller than $\mathrm{diam}(G)$. It turns out that graph coloring is an example of such a problem.

In the rest of this chapter, we will design an efficient distributed algorithm that finds a graph coloring in the LOCAL model. The algorithm will find a proper vertex coloring with $\Delta + 1$ colors in $O(\Delta + \log^* n)$ communication rounds, for any graph with $n = |V|$ nodes and maximum degree Δ. We will start with a simple greedy algorithm that we will later use as a subroutine.

4.5 Greedy Color Reduction

Let $x \in \mathbb{N}$. We present a greedy color reduction algorithm that reduces the number of colors from x to

$$y = \max\{x - 1, \Delta + 1\},$$

where Δ is the maximum degree of the graph. That is, given a proper vertex coloring with x colors, the algorithm outputs a proper vertex coloring with y colors. The running time of the algorithm is one communication round.

4.5.1 Algorithm

The algorithm proceeds as follows; here f is the x-coloring that we are given as input and g is the y-coloring that we produce as output. See Figure 4.3 for an illustration.

(a) In the first communication round, each node $v \in V$ sends its color $f(v)$ to each of its neighbors.

(b) Now each node $v \in V$ knows the set

$$C(v) = \{i : \text{there is a neighbor } u \text{ of } v \text{ with } f(u) = i\}.$$

We say that a node is *active* if $f(v) > \max C(v)$; otherwise it is *passive*. That is, the colors of the active nodes are local maxima. Let

$$\bar{C}(v) = \{1, 2, \dots\} \setminus C(v)$$

be the set of *free colors* in the neighborhood of v.

(c) A node $v \in V$ outputs

$$g(v) = \begin{cases} f(v) & \text{if } v \text{ is passive,} \\ \min \bar{C}(v) & \text{if } v \text{ is active.} \end{cases}$$

Informally, a node whose color is a local maximum re-colors itself with the first available free color.

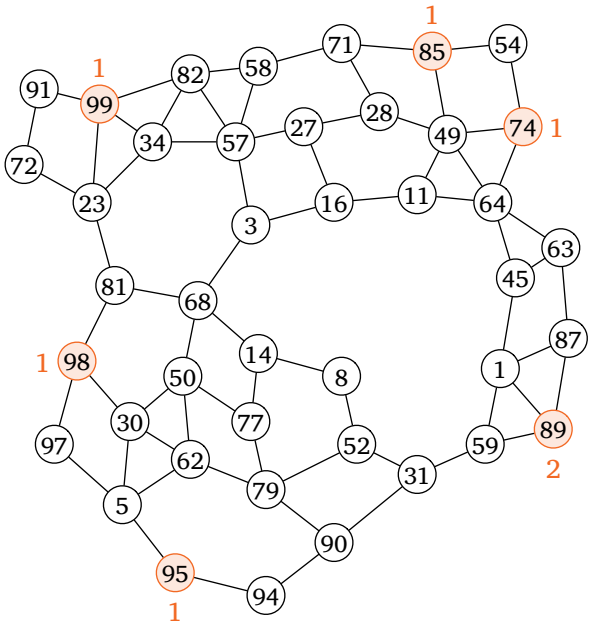


Figure 4.3: Greedy color reduction. The active nodes have been highlighted. Note that in the original coloring f, the largest color was 99, while in the new coloring, the largest color is strictly smaller than 99—we have successfully reduced the number of colors in the graph.

4.5.2 Analysis

Lemma 4.1. *The greedy color reduction algorithm reduces the number of colors from x to*

$$y = \max\{x-1, \Delta+1\},$$

where Δ is the maximum degree of the graph.

Proof. Let us first prove that $g(v) \in \{1, 2, \ldots, y\}$ for all $v \in V$. As f is a proper coloring, we cannot have $f(v) = \max C(v)$. Hence there are only two possibilities.

(a) $f(v) < \max C(v)$. Now v is passive, and it is adjacent to a node u such that $f(v) < f(u)$. We have

$$g(v) = f(v) \le f(u) - 1 \le x - 1 \le y.$$

(b) $f(v) > \max C(v)$. Now v is active, and we have

$$g(v) = \min \bar{C}(v).$$

There is at least one value $i \in \{1, 2, \ldots, |C(v)| + 1\}$ with $i \notin C(v)$; hence

$$\min \bar{C}(v) \le |C(v)| + 1 \le \deg_G(v) + 1 \le \Delta + 1 \le y.$$

Next we will show that g is a proper vertex coloring of G. Let $\{u, v\} \in E$. If both u and v are passive, we have

$$g(u) = f(u) \neq f(v) = g(v).$$

Otherwise, w.l.o.g., assume that u is active. Then we must have $f(u) > f(v)$. It follows that $f(u) \in C(v)$ and $f(v) \le \max C(v)$; therefore v is passive. Now $g(u) \notin C(u)$ while $g(v) = f(v) \in C(u)$; we have $g(u) \neq g(v)$. □

The key observation is that the set of active nodes forms an independent set. Therefore all active nodes can pick their new colors simultaneously in parallel, without any risk of choosing colors that might conflict with each other.

4.5.3 Remarks

The greedy color reduction algorithm does not need to know the number of colors x or the maximum degree Δ; we only used them in the analysis. We can take any graph, blindly apply greedy color reduction, and we are guaranteed to reduce the number of colors by one—provided that the number of colors was larger than $\Delta + 1$. In particular, we can apply the greedy color reduction repeatedly until we get stuck, at which point we have a $(\Delta + 1)$-coloring of G—we will formalize and generalize this idea in Exercise 4.3.

4.6 Efficient $(\Delta + 1)$-coloring

In the remaining sections we will describe two coloring algorithms that, together with the greedy algorithm from the previous section, can be used to $(\Delta + 1)$-color graphs of maximum degree Δ.

On a high level, the $(\Delta + 1)$-coloring algorithm is composed of the following subroutines:

(a) Algorithm from Section 4.8: Using unique identifiers as input, compute an $O(\Delta^2)$-coloring x in $O(\log^* n)$ rounds.

(b) Algorithm from Section 4.7: Given x as input, compute an $O(\Delta)$-coloring y in $O(\Delta)$ rounds.

(c) Algorithm from Section 4.5: Given y as input, compute a $(\Delta + 1)$-coloring z in $O(\Delta)$ rounds.

We have already seen the greedy algorithm that we will use in the final step; we will proceed in the reverse order and present next the algorithm that turns an $O(\Delta^2)$-coloring into an $O(\Delta)$-coloring. In what follows, we will assume that the nodes are given the values of Δ and n as input; these assumptions will simplify the algorithms significantly.

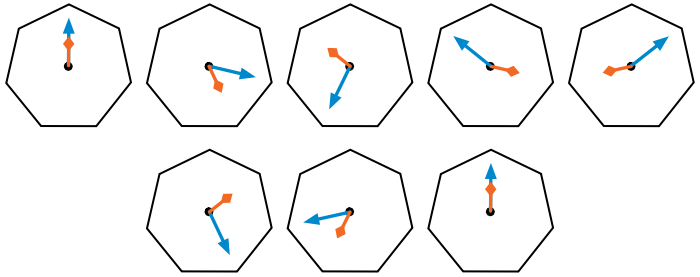


Figure 4.4: Two clocks for $q = 7$. The blue hand moves 2 steps per time unit, and the orange hand 3 steps. Hands moving at different speeds meet again after q moves, but not before.

4.7 Additive-Group Coloring

Consider two clocks with q steps, for any prime q; see Figure 4.4. The first clock moves its hand a steps in each time unit, and the second clock moves its hand $b \neq a$ steps in each time unit. Starting from the same position, when are the two hands in the same position again?

It is a fundamental property of *finite fields* that they are in the same position again after exactly q steps. We recap definitions and facts about finite fields in Section 4.13.

Building on this observation, we construct an algorithm where each node behaves like a clock with one hand, turning its hand with some constant speed. We will use the input coloring to ensure that *clocks with the same starting position turn their hands at different speeds*. Then we will simply wait until a clock is in a position not shared by any of the neighbors, and this position becomes the final color of the node. If we do not have too many neighbors, each node will eventually find such a position, leading to a proper coloring.

4.7.1 Algorithm

Let q be a prime number with $q > 2\Delta$. We assume that we are given a coloring with q^2 colors, and we will show how to construct a coloring

with q colors in $O(q)$ rounds. Put otherwise, we can turn a coloring with $O(\Delta^2)$ colors into a coloring with $O(\Delta)$ colors in $O(\Delta)$ rounds, as long as we choose our prime number q in a suitable manner.

If we have an input coloring with q^2 colors, we can represent the color of node v as a pair $f(v) = \langle f_1(v), f_2(v) \rangle$ where $f_1(v)$ and $f_2(v)$ are integers between 0 and $q-1$.

Using the clock analogy, v can be seen as a clock with the hand at position $f_2(v)$, turning at speed $f_1(v)$. In the algorithm we will stop clocks by setting $f_1(v) = 0$ whenever this is possible in a conflict-free manner. When all clocks have stopped, all nodes have colors of the form $\langle 0, f_2(v) \rangle$ where $f_2(v)$ is between 0 and $q-1$, and hence we obtain a proper coloring with q colors.

We say that two colors $\langle a_1, a_2 \rangle$ and $\langle b_1, b_2 \rangle$ are in *conflict* if $a_2 = b_2$. The algorithm repeatedly performs the following steps:

- Each node sends its current colors to each neighbor.
- For each node v, if $f(v)$ is in conflict with any neighbor, set

$$f(v) \leftarrow \big\langle f_1(v),\, (f_1(v) + f_2(v)) \bmod q \big\rangle.$$

  Otherwise, set

$$f(v) \leftarrow \big\langle 0,\, f_2(v) \big\rangle.$$

In essence, we stop non-conflicting clocks and keep moving all other clocks at a constant rate. We say that a node v is *stopped* when $f_1(v) = 0$; otherwise it is *running*; note a stopped node will not change its color any more.

We show that after $O(q)$ iterations of this loop, all nodes will be stopped, and they form a proper coloring—assuming we started with a proper coloring.

4.7.2 Correctness

First, we show that in each iteration a proper coloring remains proper. In what follows, we use f to denote the coloring before one iteration and

g to denote the coloring after the iteration. Consider a fixed node v and an arbitrary neighbor u. We show by a case analysis that $f(v) \neq f(u)$ implies $g(v) \neq g(u)$.

(1) Assume that v is stopped after this round; then $g(v) = \langle 0, f_2(v) \rangle$.

   (a) If $f_1(u) = 0$, then u has stopped and $g(u) = f(u)$. By assumption $f(v) \neq f(u)$ and therefore $g(v) \neq g(u)$.

   (b) If $f_1(u) \neq 0$ and $f(u)$ is not in conflict with its neighbors, then $g(u) = \langle 0, f_2(u) \rangle$. As there are no conflicts with v, we must have $f_2(v) \neq f_2(u)$, and therefore $g(v) \neq g(u)$.

   (c) Otherwise $f_1(u) \neq 0$ and $f(u)$ is in conflict with a neighboring color. Then $g_1(u) = f_1(u) \neq 0 = g_1(v)$, and therefore $g(v) \neq g(u)$.

(2) Otherwise we have $g(v) = \langle f_1(v), (f_1(v) + f_2(v)) \bmod q \rangle$, where $f_1(v) \neq 0$.

   (a) If u has stopped, then $g_1(u) = 0$, and therefore $g(v) \neq g(u)$.

   (b) Otherwise u is running. Then

$$g(u) = \langle f_1(u), (f_1(u) + f_2(u)) \bmod q \rangle.$$

   If $f_1(v) \neq f_1(u)$, we will have $g_1(v) \neq g_1(u)$ and therefore $g(v) \neq g(u)$. Otherwise $f_1(v) = f_1(u)$ but then by assumption we must have $f_2(v) \neq f_2(u)$, which implies $g_2(v) \neq g_2(u)$ and therefore $g(v) \neq g(u)$.

4.7.3 Running Time

Next we analyze the running time. Assume that we start with a proper coloring f. We want to show that after a sufficient number of iterations of the additive-group algorithm, each node must have had an iteration in which its color did not conflict with color of its neighbors, and hence got an opportunity to stop.

Let f^0 denote the initial coloring before the first iteration and let f^i denote the coloring after iteration $i = 1, 2, \ldots$. The following lemma shows that two running nodes do not conflict too often during the execution.

Lemma 4.2. *Consider t consecutive iterations of the additive-group coloring algorithm, for $t \leq q$. Let u and v be adjacent nodes such that both of them are still running before iteration t. Then there is at most one iteration $i = 0, 1, \ldots, t-1$ with a conflict $f_2^i(u) = f_2^i(v)$.*

Proof. Assume that for some i we have $f^i(u) = \langle a, b \rangle$ and $f^i(v) = \langle a', b \rangle$ with $a \neq a'$. In the subsequent iterations $j = i+1, i+2, \ldots$, we have

$$f_2^j(u) - f_2^j(v) \equiv (a - a')(j - i) \mod q.$$

Assume that for some j we have another conflict $f_2^j(u) = f_2^j(v)$, implying that $(a-a')(j-i) \equiv 0 \mod q$. If a prime divides a product xy of two integers x and y, then it also divides x or y (Euclid's Lemma). But $a - a'$ cannot be a multiple of q, since $a \neq a'$ and $0 \leq a, a' < q$, and $j - i$ cannot be a multiple of q, either, since $0 \leq i < j < q$. □

We also need to show that a node is not in conflict with a stopped node too often.

Lemma 4.3. *Consider t consecutive iterations of the additive-group coloring algorithm, for $t \leq q$. Let u and v be adjacent nodes such that u is still running before iteration t but v was stopped before iteration 1. Then there is at most one iteration $i = 0, 1, \ldots, t-1$ with a conflict $f_2^i(u) = f_2^i(v)$.*

Proof. The same argument as in the proof of Lemma 4.2 works, this time with $a' = 0$. □

It remains to show that, based on Lemmas 4.2 and 4.3, the algorithm finishes fast.

Consider a sequence of consecutive $q > 2\Delta$ iterations of the additive-group coloring algorithm starting with any initial coloring f. Consider an arbitrary node u that does not stop during any of these rounds. Let v

be a neighbor of u. No matter if and when v stops, the color of v will conflict with color of u at most twice during the q rounds:

- Consider the rounds (if any) in which v is running. There are at most q such rounds. By Lemma 4.2, u conflicts with v at most once during these rounds.
- Consider the remaining rounds (if any) in which v is stopped. There are at most q such rounds. By Lemma 4.3, u conflicts with v at most once during these rounds.

So for each neighbor v of u, there are at most 2 rounds among q rounds such that the color of v conflicts with the color of u. As there are at most Δ neighbors, there are at most 2Δ rounds among q rounds such that the color of some neighbor of u conflicts with the current color of u. But $q > 2\Delta$, so there has to be at least one round after which none of the neighbors are in conflict with u—and hence there will be an opportunity for u to stop.

4.8 Fast $O(\Delta^2)$-coloring

The additive-group coloring algorithm assumes that we start with an $O(\Delta^2)$-coloring of the network. In this section we present an algorithm that computes an $O(\Delta^2)$-coloring in $O(\log^* n)$ communication rounds.

The algorithm proceeds in two phases. In the first phase, the coloring given by the unique identifiers is iteratively reduced to an $O(\Delta^2 \log^2 \Delta)$-coloring. In the second phase, a final color reduction step yields an $O(\Delta^2)$ coloring.

Both phases are based on the same combinatorial construction, called a cover-free set family. We begin by describing the construction for the first phase.

4.8.1 Cover-Free Set Families

The coloring algorithm is based on the existence of non-covering families of sets. Intuitively, these are families of sets such that any two sets do

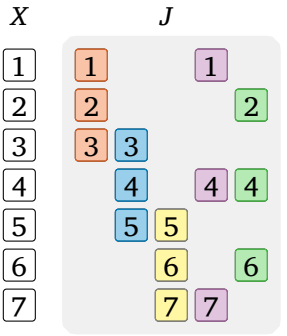


Figure 4.5: A 2-cover-free set family J of 5 subsets of a base set X on 7 elements. No two sets cover a third distinct set.

not have a large overlap: then no small collection of sets contains all elements in another set. Therefore, if each node is assigned such a set, it can find an element that is not in the sets of its neighbors, and pick that element as its new color.

A family J of n subsets of $\{1,\ldots,m\}$ is *k-cover-free* if for every $S \in J$ and every collection of k sets $S_1,\ldots,S_k \in J$ distinct from S we have that

$$S \not\subseteq \bigcup_{i=1}^{k} S_i.$$

See Figure 4.5 for an example.

4.8.2 Constructing Cover-Free Set Families

Cover-free set families can be constructed using *polynomials over finite fields*. The example of finite fields we are interested in is GF(q) for a prime q, which is simply modular arithmetic of integers modulo q. We consider polynomials over such a field. A brief recap is given in Section 4.13.

A basic result about polynomials states that two distinct polynomials evaluate to the same value at a bounded number of points

Lemma 4.4. *Let f, g be two distinct polynomials of degree d over a finite field* GF(q)*, for some prime q. Then $f(x) = g(x)$ holds for at most d elements $x \in \mathrm{GF}(q)$.*

Proof. See Section 4.13. □

Now fix a prime q. Our base set will be $X = \mathrm{GF}(q) \times \mathrm{GF}(q)$. Thus we have that $|X| = m = q^2$.

For a positive natural number d, consider $\mathrm{Poly}(d, q)$, the set of polynomials of degree d over GF(q). For each polynomial $g \in \mathrm{Poly}(d, q)$, fix the set

$$S_g = \big\{(a, g(a)) \;\big|\; a \in \mathrm{GF}(q)\big\}$$

that is associated with this polynomial. Note that each S_g contains exactly q elements: one for each element of GF(q). Then we can define the family

$$J = J_{d,q} = \big\{S_g \;\big|\; g \in \mathrm{Poly}(d, q)\big\}.$$

Consider any two distinct polynomials f and g in $\mathrm{Poly}(d, q)$: by Lemma 4.4 there are at most d elements a such that $f(a) = g(a)$. Therefore $|S_f \cap S_g| \leq d$, and J is a $\lfloor q/d \rfloor$-cover-free set family.

Any polynomial is uniquely defined by its coefficients. Therefore the set $J_{d,q}$ has size q^{d+1}, as it consists of a set of pairs for each polynomial of degree d.

By choosing parameters q and d we can construct a Δ-cover free family that can be used to color efficiently.

Lemma 4.5. *For all integers x, Δ such that $x > \Delta \geq 2$, there exists a Δ-cover-free family J of x subsets from a base set of $m \leq 4(\Delta+1)^2 \log^2 x$ elements.*

Proof. We begin by choosing a prime q such that

$$\big\lfloor (\Delta+1)\log x \big\rfloor \leq q \leq 2 \cdot \big\lfloor (\Delta+1)\log x \big\rfloor.$$

By the Bertrand–Chebyshev theorem such a prime must always exist. Set $d = \lfloor \log x \rfloor$. By the previous observation, the family $J_{d,q}$, for the above

parameter settings, is a $\lfloor q/d \rfloor$-cover-free family, where

$$\lfloor q/d \rfloor \geq \left\lfloor \frac{\lfloor (\Delta+1)\log x \rfloor}{\lfloor \log x \rfloor} \right\rfloor \geq \left\lfloor \frac{(\Delta+1)\log x - 1}{\log x} \right\rfloor \geq \Delta.$$

There are at least

$$q^{d+1} \geq (\Delta \log x)^{\log x} > x$$

sets in $J_{d,q}$, so we can choose x of them. The base set has

$$q^2 \leq 4(\Delta+1)^2 \log^2 x$$

elements. □

4.8.3 Efficient Color Reduction

Using Δ-cover-free sets we can construct an algorithm that reduces the number of colors from x to $y \leq 4(\Delta+1)^2 \log^2 x$ in one communication round, as long as $x > \Delta$.

Let f denote the input x-coloring and g the output y-coloring. Assume that J is a Δ-cover-free family of x sets on a base set of y elements, as in Lemma 4.5, that is ordered as $S_1, S_2, \ldots, S_x$. The algorithm functions as follows.

(a) Each node $v \in V$ sends its current color $f(v)$ to each of its neighbors.

(b) Each node receives the colors $f(u)$ of its neighbors $u \in N(v)$. Then it constructs the set $S_{f(v)}$, and the sets $S_{f(u)}$ for all $u \in N(v)$. Since $f(v) \neq f(u)$ for all $u \in N(v)$, and J is a Δ-cover-free family, we have that

$$S_{f(v)} \nsubseteq \bigcup_{u \in N(v)} S_{f(u)}.$$

In particular, there exists at least one $c \in S_{f(v)} \setminus \cup_{u \in N(v)} S_{f(u)}$. Node v sets $g(v) = c$ for the smallest such c.

Now assume that f is a proper coloring, that is, $f(v) \neq f(u)$ for all neighbors v and u. This implies that for each node v, each of its neighbors u selects a set that is different from $S_{f(v)}$; overall, the neighbors will select at most Δ distinct sets. Since J is a Δ-cover-free family, each node v can find an element $c \in S_{f(v)}$ that is not in the sets of its neighbors. Therefore setting $g(v) = c$ forms a proper coloring. Finally, since the sets $S \in J$ are subsets of $\{1, \ldots, y\}$, for $y \leq 4(\Delta+1)^2(\log x)^2$, we have that g is a y-coloring.

4.8.4 Iterated Color Reduction

By a repeated application of the color reduction algorithm it is possible to reduce the number of colors down to $O(\Delta^2 \log^2 \Delta)$. Assuming we start with an input x-coloring, this will take $O(\log^* x)$ rounds.

We will now show that $O(\log^* x)$ iterations of the color reduction algorithm will reduce the number of colors from x to $O(\Delta^2 \log^2 \Delta)$. We assume that in the beginning, both x and Δ are known. Therefore after each iteration, all nodes know the total number of colors.

Assume that $x > 4(\Delta+1)^2 \log^2 \Delta$. Repeated iterations of the color reduction algorithm reduce the number of colors as follows:

$$\begin{aligned} x_0 \mapsto x_1 &\leq 4(\Delta+1)^2 \log^2 x, \\ x_1 \mapsto x_2 &\leq 4(\Delta+1)^2 \log^2(4(\Delta+1)^2 \log^2 x) \\ &= 4(\Delta+1)^2 \big(\log 4 + 2\log(\Delta+1) + 2\log\log x\big)^2. \end{aligned}$$

If $\log\log x \geq \log 4 + 2\log(\Delta+1)$, we have that

$$\begin{aligned} x_2 &\leq 4(\Delta+1)^2 (3\log\log x)^2 \\ &= \big(6(\Delta+1)\log\log x\big)^2. \end{aligned}$$

In the next step, we reduce colors as follows:

$$\begin{aligned} x_2 \mapsto x_3 &\leq 4(\Delta+1)^2 \log^2\big(36(\Delta+1)^2(\log\log x)^2\big) \\ &= 4(\Delta+1)^2 \big(\log 36 + 2\log(\Delta+1) + 2\log\log\log x\big)^2. \end{aligned}$$

If $\log\log\log x \geq \log 36 + 2\log(\Delta+1)$, we have that

$$\begin{aligned} x_3 &\leq 4(\Delta+1)^2(3\log\log\log x)^2 \\ &= \big(6(\Delta+1)\log\log\log x\big)^2. \end{aligned}$$

Now we can see the pattern: as long as

$$\log^{(i)} x \geq \log 36 + 2\log(\Delta+1),$$

where $\log^{(i)} x$ is the i times iterated logarithm of x, we reduce colors from $(6(\Delta+1)\log^{(i-1)} x)^2$ to $(6(\Delta+1)\log^{(i)} x)^2$ in the ith step.

Once $\log^{(i)} x \geq \log 36 + 2\log(\Delta+1)$ no longer holds, we have a coloring with at most

$$c_\Delta = 4(\Delta+1)^2\big(3(\log 36 + 2\log(\Delta+1))\big)^2$$

colors. We can numerically verify that for all $\Delta \geq 2$, we have that

$$4(\Delta+1)^2\big(3(\log 36 + 2\log(\Delta+1))\big)^2 \leq (11(\Delta+1))^3.$$

We will use this observation in the next step.

It remains to calculate how many color reduction steps are required. By definition, after $T = \log^* x$ iterations we have that $\log^{(T)} x \leq 1$. Thus, after at most $\log^* x$ iterations of the color reduction algorithm we have a coloring with at most c_Δ colors.

4.8.5 Final Color Reduction Step

In the last step, we will reduce the coloring to an $O(\Delta^2)$-coloring. We will use another construction of Δ-cover-free families based on polynomials.

Lemma 4.6. *For all Δ, there exists a Δ-cover-free family J of x subsets from a base set of $m \leq (22(\Delta+1))^2$ elements for $x \leq (11(\Delta+1))^3$.*

This immediately gives us the following color reduction algorithm.

Corollary 4.7. *There is a distributed algorithm that, given a* $(11(\Delta+1))^3$*-coloring as an input, in one round computes a* $(22(\Delta+1))^2$*-coloring.*

Proof of Lemma 4.6. Our base set will be X with $|X| = m = q^2$, for a prime q. Again it is useful to see $X = \mathrm{GF}(q) \times \mathrm{GF}(q)$ as pairs of elements from the finite field over q elements.

Now consider polynomials $\mathrm{Poly}(2,q)$ of degree 2 over $\mathrm{GF}(q)$. For each such polynomial $g \in \mathrm{Poly}(2,q)$, let

$$S_g = \big\{(a, g(a)) \;\big|\; a \in \mathrm{GF}(q)\big\}$$

be the pairs defined by the valuations of the polynomial g at each element of $\mathrm{GF}(q)$. We have that $|S_g| = q$ for all g.

Now we can construct the family

$$J = J_{2,q} = \big\{S_g \;\big|\; g \in \mathrm{Poly}(2,q)\big\}$$

as the collection of point sets defined by all polynomials of degree 2. We have that $|J| = q^3$ since a polynomial is uniquely determined by its coefficients.

By Lemma 4.4, we have that $|S_f \cap S_g| \leq 2$ for any distinct polynomials $f, g \in P(2,q)$. Therefore covering any set S_g requires at least $\lceil q/2 \rceil$ other sets (distinct from S_g) from J.

We are now ready to prove that J is a Δ-cover-free family for suitable parameter settings. Since each set S_g contains q elements, and the intersection between the sets of distinct polynomials is at most 2, we want to find q such that $2\Delta \leq q-1$ and q^3 is large enough. Using the Bertrand–Chebyshev theorem we know that there exists a prime q such that

$$11(\Delta+1) \leq q \leq 22(\Delta+1).$$

Any value q from this range is large enough. The base set X has size

$$m = q^2 \leq (22(\Delta+1))^2.$$

The family J has size

$$|J| \geq (11(\Delta+1))^3.$$

Finally, since we choose $q \geq 2\Delta+1$, we have that no collection of Δ sets $\mathcal{S} = \{S_1, S_2, \ldots, S_\Delta\} \subseteq J$ can cover a set $S \notin \mathcal{S}$. □

4.9 Putting Things Together

It remains to show how to use the three algorithms we have seen so far together.

Theorem 4.8. *Assume that we know parameters Δ and n, and some polynomial bound n^c on the size of the unique identifiers. Graphs on n vertices with maximum degree Δ can be $(\Delta+1)$-colored in $O(\Delta+\log^* n)$ rounds in the* LOCAL *model.*

Proof. We begin with the unique identifiers, and treat them as an initial coloring with n^c colors.

(a) In the first phase we run the efficient color reduction algorithm from Section 4.8.3 for $T_1 = \log^*(n^c) = O(\log^* n)$ rounds to produce a coloring y_1 with at most $(11(\Delta+1))^3$ colors.

(b) In the second phase, after T_1 rounds have passed, each vertex can apply the final color reduction step from Section 4.8.5 to compute a coloring y_2. This reduces colors from $(11(\Delta+1))^3$ to $(22(\Delta+1))^2$.

(c) After T_1+1 rounds, we have computed an $O(\Delta^2)$-coloring y_2. Now each vertex runs the additive-group coloring algorithm from Section 4.7, applying it with y_2 as input. For a parameter $q \leq 2\sqrt{(22(\Delta+1))^2} = 44\Delta+44$, this algorithm runs for $T_2 = q$ steps and computes a q-coloring y_3.

(d) In the last phase, after T_1+1+T_2 rounds, we apply the greedy color reduction algorithm from Section 4.5 iteratively $T_3 = 43\Delta+43$ times. Each iteration requires one round and reduces the maximum color by one.

After a total of

$$\begin{aligned} T_1+1+T_2+T_3 &\leq \log^*(n^c)+87\Delta+88 \\ &= O(\Delta+\log^* n) \end{aligned}$$

rounds, we have computed a $(\Delta+1)$-coloring. □

4.10 Quiz

Consider the algorithm from Section 4.7.1 in the following setting:

- The network is a complete graph with $n = 4$ nodes; hence the maximum degree is $\Delta = 3$, and we can choose $q = 7 > 2\Delta$.
- We are given a coloring with $q^2 = 49$ colors; we can represent the possible input colors as pairs $(0,0), (0,1), \ldots, (6,6)$.

Give an example of an input coloring such that we need to do exactly 6 iterations of the algorithm until all nodes have reached their final colors, i.e., colors of the form $(0, x)$.

Please give the answer by listing the four original color pairs of the nodes in any order; for example, if we asked for a coloring in which you need exactly 3 iterations, this would be a correct answer: $(2,3)$, $(3,2)$, $(3,6)$, $(4,6)$.

4.11 Exercises

Exercise 4.1 (applications)**.** Let Δ be a known constant, and let $\mathscr{F}$ be the family of graphs of maximum degree at most Δ. Design fast distributed algorithms that solve the following problems on $\mathscr{F}$ in the LOCAL model.

(a) Maximal independent set.

(b) Maximal matching.

(c) Edge coloring with $O(\Delta)$ colors.

You can assume that all nodes get the value of n (the number of nodes) as input; also the parameter c in the identifier space is a known constant, so all nodes know the range of unique identifiers.

Exercise 4.2 (vertex cover)**.** Let $\mathscr{F}$ consist of cycle graphs. Design a fast distributed algorithm that finds a 1.1-approximation of a minimum vertex cover on $\mathscr{F}$ in the LOCAL model.

▷ *hint H*

Exercise 4.3 (iterated greedy)**.** Design a color reduction algorithm A with the following properties: given any graph $G = (V, E)$ and any proper vertex coloring f, algorithm A outputs a proper vertex coloring g such that for each node $v \in V$ we have $g(v) \leq \deg_G(v) + 1$.

Let Δ be the maximum degree of G, let $n = |V|$ be the number of nodes in G, and let x be the number of colors in coloring f. The running time of A should be at most

$$\min\{n, x\} + O(1).$$

Note that the algorithm does not know n, x, or Δ. Also note that we may have either $x \leq n$ or $x \geq n$.

▷ *hint I*

Exercise 4.4 (distance-2 coloring)**.** Let $G = (V, E)$ be a graph. A *distance-2 coloring with k colors* is a function $f : V \to \{1, 2, \ldots, k\}$ with the following property:

$$\text{dist}_G(u, v) \leq 2 \text{ implies } f(u) \neq f(v) \text{ for all nodes } u \neq v.$$

Let Δ be a known constant, and let $\mathscr{F}$ be the family of graphs of maximum degree at most Δ. Design a fast distributed algorithm that finds a distance-2 coloring with $O(\Delta^2)$ colors for any graph $G \in \mathscr{F}$ in the LOCAL model.

You can assume that all nodes get the value of n (the number of nodes) as input; also the parameter c in the identifier space is a known constant, so all nodes know the range of unique identifiers.

▷ *hint J*

⋆ **Exercise 4.5** (numeral systems)**.** The fast color reduction algorithm from Section 1.4 is based on the idea of identifying a digit that differs in the *binary* encodings of the colors. Generalize the idea: design an analogous algorithm that finds a digit that differs in the base-k encodings of the colors, for an arbitrary k, and analyze the running time of the algorithm (cf. Exercise 1.6). Is the special case of $k = 2$ the best possible choice?

⋆ **Exercise 4.6** (from bits to sets)**.** The fast color reduction algorithm from Section 1.4 can reduce the number of colors from 2^x to $2x$ in one round in any directed pseudoforest, for any positive integer x. For example, we can reduce the number of colors as follows:

$$2^{128} \to 256 \to 16 \to 8 \to 6.$$

One of the problems is that an iterated application of the algorithm slows down and eventually "gets stuck" at $x = 3$, i.e., at six colors.

In this exercise we will design a faster color reduction algorithm that reduces the number of colors from

$$h(x) = \binom{2x}{x}$$

to $2x$ in one round, for any positive integer x. For example, we can reduce the number of colors as follows:

$$184756 \to 20 \to 6 \to 4.$$

Here

$$\begin{aligned} 184756 &= h(10), \\ 2 \cdot 10 = 20 &= h(3), \\ 2 \cdot 3 = 6 &= h(2). \end{aligned}$$

In particular, the algorithm does not get stuck at six colors; we can use the same algorithm to reduce the number of colors to four. Moreover, at least in this case the algorithm seems to be much more efficient—it can reduce the number of colors from 184756 to 6 in two rounds, while the prior algorithm requires three rounds to achieve the same reduction.

The basic structure of the new algorithm follows the fast color reduction algorithm—in particular, we use one communication round to compute the values $s(v)$ for all nodes $v \in V$. However, the technique for choosing the new color is different: as the name suggests, we will not interpret colors as bit strings but as *sets*.

To this end, let $H(x)$ consist of all subsets

$$X \subseteq \{1, 2, \ldots, 2x\}$$

with $|X| = x$. There are precisely $h(x)$ such subsets, and hence we can find a bijection

$$L : \{1, 2, \ldots, h(x)\} \to H(x).$$

We have $f(v) \neq s(v)$. Hence $L(f(v)) \neq L(s(v))$. As both $L(f(v))$ and $L(s(v))$ are subsets of size x, it follows that

$$L(f(v)) \setminus L(s(v)) \neq \varnothing.$$

We choose the new color $g(v)$ of a node $v \in V$ as follows:

$$g(v) = \min\big(L(f(v)) \setminus L(s(v))\big).$$

Prove that this algorithm works correctly. In particular, show that $g : V \to \{1, 2, \ldots, 2x\}$ is a proper graph coloring of the directed pseudoforest G.

Analyze the running time of the new algorithm and compare it with the old algorithm. Is the new algorithm always faster? Can you prove a general result analogous to the claim of Exercise 1.6?

⋆ **Exercise 4.7** (dominating set approximation)**.** Let Δ be a known constant, and let $\mathscr{F}$ be the family of graphs of maximum degree at most Δ. Design an algorithm that finds an $O(\log \Delta)$-approximation of a minimum dominating set on $\mathscr{F}$ in the LOCAL model.

▷ *hint K*

4.12 Bibliographic Notes

The model of computing is from Linial's [30] seminal paper, and the name LOCAL is from Peleg's [36] book. The additive-group coloring algorithm is due to Barenboim et al. [8]. The effective color reduction algorithm is from Linial [30], and the construction of cover-free families

from Barenboim and Elkin [7]. The algorithm of Exercise 4.7 is from Friedman and Kogan [19]. The Bertrand–Chebyshev theorem was first proven by Chebyshev [14]. The proof of Lemma 4.4 follows the proofs of Abraham [1].

4.13 Appendix: Finite Fields

For our purposes, finite field of size q can be seen as the set $\{0, \ldots, q-1\}$ equipped with modular arithmetic, for any prime q. Fields support addition, subtraction, multiplication, and division with the usual rules. We denote the finite field with q elements (also known as a Galois field) by $\mathrm{GF}(q)$.

Our proofs will use the following two properties of finite fields.

(a) Each element a of the field has a unique multiplicative inverse element, denoted by a^{-1}, such that $a \cdot a^{-1} = 1$.

(b) The product ab of two elements is zero if and only if $a = 0$ or $b = 0$.

We can define polynomials over $\mathrm{GF}(q)$. A polynomial $f[X]$ of degree d can be represented as

$$f_0 + f_1 X + f_2 X^2 + \cdots + f_d X^d,$$

where the coefficients f_i are elements of $\mathrm{GF}(q)$. A polynomial is non-trivial if there exists some $f_i \neq 0$. An element $a \in \mathrm{GF}(q)$ is a zero of a polynomial f if $f(a) = 0$.

Proof of Lemma 4.4. We will prove the lemma by proving a related statement: any non-trivial polynomial of degree d has at most d zeros. Since $f(x) - g(x)$ is a polynomial of degree at most d, Lemma 4.4 follows.

The proof is by induction on d. Let $f[X] = f_0 + f_1 X$ denote an arbitrary polynomial of degree 1 over some finite field of size q. Since each element a of a field has a unique inverse a^{-1}, there is a unique zero of $f[X]$: $X = -(f_0)(f_1)^{-1}$.

Now assume that $d \geq 2$ and that the claim holds for smaller degrees. If polynomial f has no zeros, the claim holds. Therefore assume that f has at least one zero $a \in \mathrm{GF}(q)$. We will show that there exists a polynomial g of degree $d-1$ such that $f = (X-a)g$. By the induction hypothesis g has at most $d-1$ zeros, $X-a$ has one zero, and we know that the product equals zero if and only if either $X-a=0$ or $g[X]=0$.

We show that g exists by induction. If $d = 1$, we can select $a = -(f_0)(f_1)^{-1}$ and $g = f_1$ to get $f[X] = (X + (f_0)(f_1)^{-1})f_1$.

For $d \geq 2$, we again make the induction assumption. Define

$$f' = f - f_d X^{d-1}(X-a),$$

where f_d is the dth coefficient of f. This polynomial has degree less than d, since the terms of degree d cancel out. We also have that $f'(a) = 0$ since $f(a) = 0$ by assumption. By induction hypothesis there exists a g' such that $f' = (X-a)g'$ and degree of g' is at most $d-2$. By substituting $f' = (X-a)g'$ we get

$$f = (X-a)g' + (X-a)f_d X^{d-1} = (X-a)(g' + f_d X^{d-1}).$$

Therefore $f = (X-a)g$ for the polynomial $g = g' + f_d X^{d-1}$, a polynomial of degree at most $d-1$. □

Chapter 5

CONGEST Model: Bandwidth Limitations

In the previous chapter, we learned about the LOCAL model. We saw that with the help of unique identifiers, it is possible to gather the full information on a connected input graph in $O(\operatorname{diam}(G))$ rounds. To achieve this, we heavily abused the fact that we can send arbitrarily large messages. In this chapter we will see what can be done if we are only allowed to send small messages. With this restriction, we arrive at a model that is commonly known as the "CONGEST model".

5.1 Definitions

Let A be a distributed algorithm that solves a problem Π on a graph family $\mathscr{F}$ in the LOCAL model. Assume that Msg_A is a countable set; without loss of generality, we can then assume that

$$\mathrm{Msg}_A = \mathbb{N},$$

that is, the messages are encoded as natural numbers. Now we say that A solves problem Π on graph family $\mathscr{F}$ in the CONGEST model if the following holds for some constant C: for any graph $G = (V, E) \in \mathscr{F}$, algorithm A only sends messages from the set $\{0, 1, \ldots, |V|^C\}$.

Put otherwise, we have the following *bandwidth restriction*: in each communication round, over each edge, we only send $O(\log n)$-bit messages, where n is the total number of nodes.

5.2 Examples

Assume that we have an algorithm A that is designed for the LOCAL model. Moreover, assume that during the execution of A on a graph $G = (V, E)$, in each communication round, we only need to send the following pieces of information over each edge:

- $O(1)$ node identifiers,
- $O(1)$ edges, encoded as a pair of node identifiers,
- $O(1)$ counters that take values from 0 to $\operatorname{diam}(G)$,
- $O(1)$ counters that take values from 0 to $|V|$,
- $O(1)$ counters that take values from 0 to $|E|$.

Now it is easy to see that we can encode all of this as a binary string with $O(\log n)$ bits. Hence A is not just an algorithm for the LOCAL model, but it is also an algorithm for the CONGEST model.

Many algorithms that we have encountered in this book so far are of the above form, and hence they are also CONGEST algorithms (see Exercise 5.1). However, there is a notable exception: the algorithm for gathering the entire network from Section 4.2. In this algorithm, we need to send messages of size up to $\Theta(n^2)$ bits:

- To encode the set of nodes, we may need up to $\Theta(n \log n)$ bits (a list of n identifiers, each of which is $\Theta(\log n)$ bits long).
- To encode the set of edges, we may need up to $\Theta(n^2)$ bits (the adjacency matrix).

While algorithms with a running time of $O(\operatorname{diam}(G))$ or $O(n)$ are trivial in the LOCAL model, this is no longer the case in the CONGEST model. Indeed, there are graph problems that *cannot* be solved in time $O(n)$ in the CONGEST model (see Exercise 5.6).

In this chapter, we will learn techniques that can be used to design efficient algorithms in the CONGEST model. We will use the all-pairs shortest path problem as the running example.

5.3 All-Pairs Shortest Path Problem

Throughout this chapter, we will assume that the input graph $G = (V, E)$ is connected, and as usual, we have $n = |V|$. In the *all-pairs shortest path* problem (APSP in brief), the goal is to find the distances between all pairs of nodes. More precisely, the local output of node $v \in V$ is

$$f(v) = \big\{(u, d) : u \in V,\ d = \mathrm{dist}_G(v, u)\big\}.$$

That is, v has to know the identities of all other nodes, as well as the shortest-path distance between itself and all other nodes.

Note that to represent the local output of a single node we need $\Theta(n \log n)$ bits, and just to transmit this information over a single edge we would need $\Theta(n)$ communication rounds. Indeed, we can prove that any algorithm that solves the APSP problem in the CONGEST model takes $\Omega(n)$ rounds—see Exercise 5.7.

In this chapter, we will present an optimal distributed algorithm for the APSP problem: it solves the problem in $O(n)$ rounds in the CONGEST model.

5.4 Single-Source Shortest Paths

As a warm-up, we will start with a much simpler problem. Assume that we have elected a leader $s \in V$, that is, there is precisely one node s with input 1 and all other nodes have input 0. We will design an algorithm such that each node $v \in V$ outputs

$$f(v) = \mathrm{dist}_G(s, v),$$

i.e., its shortest-path distance to leader s.

The algorithm proceeds as follows. In the first round, the leader will send message '*wave*' to all neighbors, switch to state 0, and stop. In round i, each node v proceeds as follows: if v has not stopped, and if it receives message '*wave*' from some ports, it will send message '*wave*'

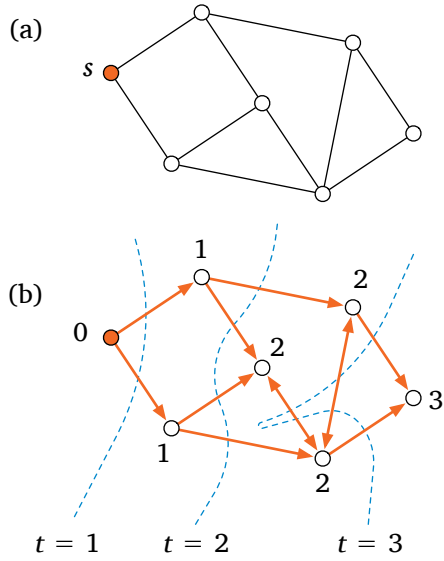


Figure 5.1: (a) Graph G and leader s. (b) Execution of algorithm Wave on graph G. The arrows denote '*wave*' messages, and the dotted lines indicate the communication round during which these messages were sent.

to all other ports, switch to state i, and stop; otherwise it does nothing. See Figure 5.1.

The analysis of the algorithm is simple. By induction, all nodes at distance i from s will receive message '*wave*' from at least one port in round i, and they will hence output the correct value i. The running time of the algorithm is $O(\mathrm{diam}(G))$ rounds in the CONGEST model.

5.5 Breadth-First Search Tree

Algorithm Wave finds the shortest-path distances from a single source s. Now we will do something slightly more demanding: calculate not just the distances but also the shortest paths.

More precisely, our goal is to construct a *breadth-first search tree* (BFS tree) T rooted at s. This is a spanning subgraph $T = (V, E')$ of G such that T is a tree, and for each node $v \in V$, the shortest path from s to v in tree T is also a shortest path from s to v in graph G. We will also label each node $v \in V$ with a *distance label* $d(v)$, so that for each node $v \in V$ we have

$$d(v) = \mathrm{dist}_T(s, v) = \mathrm{dist}_G(s, v).$$

See Figure 5.2 for an illustration. We will interpret T as a directed graph, so that each edge is of form (u, v), where $d(u) > d(v)$, that is, the edges point towards the root s.

There is a simple centralized algorithm that constructs the BFS tree and distance labels: breadth-first search. We start with an empty tree and unlabeled nodes. First we label the leader s with $d(s) = 0$. Then in step $i = 0, 1, \ldots$, we visit each node u with distance label $d(u) = i$, and check each neighbor v of u. If we have not labeled v yet, we will label it with $d(v) = i + 1$, and add the edge (v, u) to the BFS tree. This way all nodes that are at distance i from s in G will be labeled with the distance label i, and they will also be at distance i from s in T.

We can implement the same idea as a distributed algorithm in the CONGEST model. We will call this algorithm BFS. In the algorithm, each node v maintains the following variables:

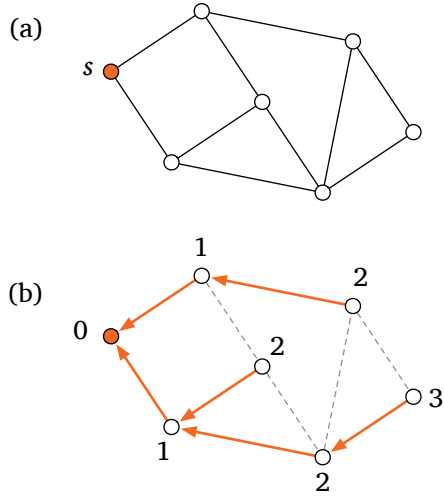


Figure 5.2: (a) Graph G and leader s. (b) BFS tree T (arrows) and distance labels $d(v)$ (numbers).

- $d(v)$: distance to the root.
- $p(v)$: pointer to the parent of node v in tree T (port number).
- $C(v)$: the set of children of node v in tree T (port numbers).
- $a(v)$: acknowledgment—set to 1 when the subtree rooted at v has been constructed.

Here $a(v) = 1$ denotes a stopping state. When the algorithm stops, variables $d(v)$ will be distance labels, tree T is encoded in variables $p(v)$ and $C(v)$, and all nodes will have $a(v) = 1$.

Initially, we set $d(v) \leftarrow \perp$, $p(v) \leftarrow \perp$, $C(v) \leftarrow \perp$, and $a(v) \leftarrow 0$ for each node v, except for the root which has $d(s) = 0$. We will grow tree T from s by iterating the following steps:

- Each node v with $d(v) \neq \perp$ and $C(v) = \perp$ will send a *proposal* with value $d(v)$ to all neighbors.

- If a node u with $d(u) = \bot$ receives some proposals with value j, it will *accept* one of them and *reject* all other proposals. It will set $p(u)$ to point to the node whose proposal it accepted, and it will set $d(u) \leftarrow j + 1$.

- Each node v that sent some proposals will set $C(v)$ to be the set of neighbors that accepted proposals.

This way T will grow towards the leaf nodes. Once we reach a leaf node, we will send acknowledgments back towards the root:

- Each node v with $a(v) = 1$ and $p(v) \neq \bot$ will send an *acknowledgment* to port $p(v)$.

- Each node v with $a(v) = 0$ and $C(v) \neq \bot$ will set $a(v) \leftarrow 1$ when it has received acknowledgments from each port of $C(v)$. In particular, if a node has $C(v) = \emptyset$, it can set $a(v) \leftarrow 1$ without waiting for any acknowledgments.

It is straightforward to verify that the algorithm works correctly and constructs a BFS tree in $O(\mathrm{diam}(G))$ rounds in the CONGEST model.

Note that the acknowledgments would not be strictly necessary in order to construct the tree. However, they will be very helpful in the next section when we use algorithm BFS as a subroutine.

5.6 Leader Election

Algorithm BFS constructs a BFS tree rooted at a single leader, assuming that we have already elected a leader. Now we will show how to elect a leader. Surprisingly, we can use algorithm BFS to do it!

We will design an algorithm Leader that finds the node with the smallest identifier; this node will be the leader. The basic idea is very simple:

(a) We modify algorithm BFS so that we can run multiple copies of it in parallel, with different root nodes. We augment the messages

with the identity of the root node, and each node keeps track of the variables d, p, C, and a separately for each possible root.

(b) Then we pretend that all nodes are leaders and start running BFS. In essence, we will run n copies of BFS in parallel, and hence we will construct n BFS trees, one rooted at each node. We will denote by BFS_v the BFS process rooted at node $v \in V$, and we will write T_v for the output of this process.

However, there are two problems: First, it is not yet obvious how all this would help with leader election. Second, we cannot implement this idea directly in the CONGEST model—nodes would need to send up to n distinct messages per communication round, one per each BFS process, and there is not enough bandwidth for all those messages.

Fortunately, we can solve both of these issues very easily; see Figure 5.3:

(c) Each node will only send messages related to the tree that has the *smallest identifier as the root*. More precisely, for each node v, let $U(v) \subseteq V$ denote the set of nodes u such that v has received messages related to process BFS_u, and let $\ell(v) = \min U(v)$ be the smallest of these nodes. Then v will ignore messages related to process BFS_u for all $u \neq \ell(v)$, and it will only send messages related to process $\mathsf{BFS}_{\ell(v)}$.

We make the following observations:

- In each round, each node will only send messages related to at most one BFS process. Hence we have solved the second problem —this algorithm can be implemented in the CONGEST model.
- Let $s = \min V$ be the node with the smallest identifier. When messages related to BFS_s reach a node v, it will set $\ell(v) = s$ and never change it again. Hence all nodes will follow process BFS_s from start to end, and thanks to the acknowledgments, node s will eventually know that we have successfully constructed a BFS tree T_s rooted at it.

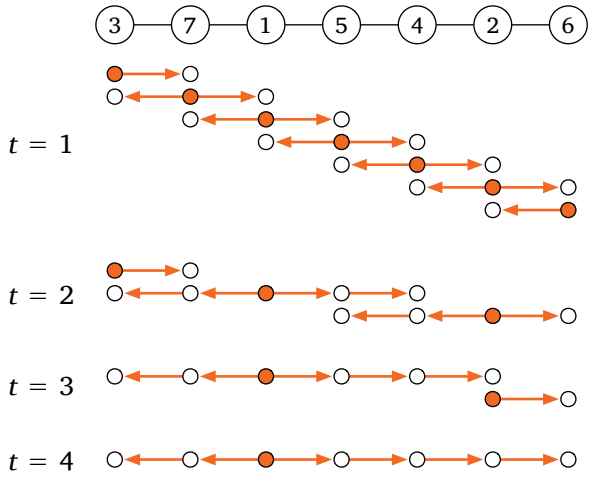


Figure 5.3: Leader election. Each node v will launch a process BFS_v that attempts to construct a BFS tree T_v rooted at v. Other nodes will happily follow BFS_v if v is the smallest leader they have seen so far; otherwise they will start to ignore messages related to BFS_v. Eventually, precisely one of the processes will complete successfully, while all other process will get stuck at some point. In this example, node 1 will be the leader, as it has the smallest identifier. Process BFS_2 will never succeed, as node 1 (as well as all other nodes that are aware of node 1) will ignore all messages related to BFS_2. Node 1 is the only root that will receive acknowledgments from every child.

- Let $u \neq \min V$ be any other node. Now there is at least one node, s, that will ignore all messages related to process BFS_u. Hence BFS_u will never finish; node u will never receive the acknowledgments related to tree T_u from all neighbors.

That is, we now have an algorithm with the following properties: after $O(\mathrm{diam}(G))$ rounds, there is precisely one node s that knows that it is the unique node $s = \min V$. To finish the leader election process, node s will inform all other nodes that leader election is over; node s will output 1 and all other nodes will output 0 and stop.

5.7 All-Pairs Shortest Paths

Now we are ready to design algorithm APSP that solves the all-pairs shortest path problem (APSP) in time $O(n)$.

We already know how to find the shortest-path distances from a single source; this is efficiently solved with algorithm Wave. Just like we did with the BFS algorithm, we can also augment Wave with the root identifier and hence have a separate process Wave_v for each possible root $v \in V$. If we could somehow run all these processes in parallel, then each node would receive a wave from every other node, and hence each node would learn the distance to every other node, which is precisely what we need to do in the APSP problem. However, it is not obvious how to achieve a good performance in the CONGEST model:

- If we try to run all Wave_v processes simultaneously in parallel, we may need to send messages related to several waves simultaneously over a single edge, and there is not enough bandwidth to do that.
- If we try to run all Wave_v processes sequentially, it will take a lot of time: the running time would be $O(n \, \mathrm{diam}(G))$ instead of $O(n)$.

The solution is to *pipeline* the Wave_v processes so that we can have many of them running simultaneously in parallel, without congestion. In essence, we want to have multiple wavefronts active simultaneously so that they never collide with each other.

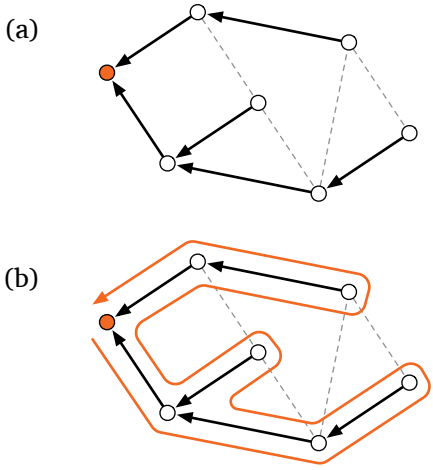


Figure 5.4: (a) BFS tree T_s rooted at s. (b) A depth-first traversal w_s of T_s.

To achieve this, we start with the leader election and the construction of a BFS tree rooted at the leader; let s be the leader, and let T_s be the BFS tree. Then we do a *depth-first traversal* of T_s. This is a walk w_s in T_s that starts at s, ends at s, and traverses each edge precisely twice; see Figure 5.4.

More concretely, we move a *token* along walk w_s. We move the token *slowly*: we always spend 2 communication rounds before we move the token to an adjacent node. Whenever the token reaches a new node v that we have not encountered previously during the walk, we launch process Wave_v. This is sufficient to avoid all congestion!

The key observation here is that the token moves slower than the waves. The waves move at speed 1 edge per round (along the edges of G), while the token moves at speed 0.5 edges per round (along the edges of T_s, which is a subgraph of G). This guarantees that two waves never collide. To see this, consider two waves Wave_u and Wave_v, so that Wave_u was launched before Wave_v. Let $d = \mathrm{dist}_G(u, v)$. Then it will take at least $2d$ rounds to move the token from u to v, but only d rounds for Wave_u to reach node v. Hence Wave_u was already past v before we

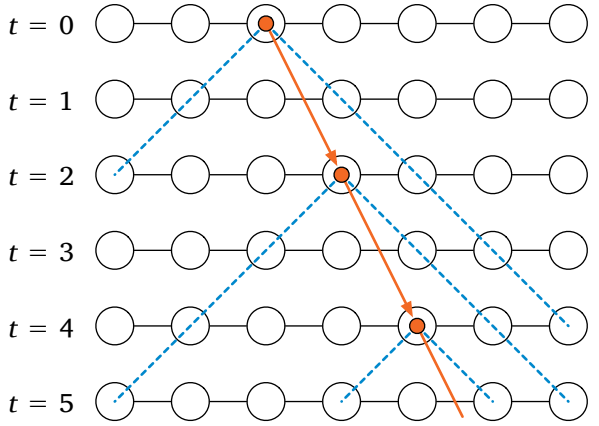


Figure 5.5: Algorithm APSP: the token walks along the BFS tree at speed 0.5 (thick arrows), while each Wave_v moves along the original graph at speed 1 (dashed lines). The waves are strictly nested: if Wave_v was triggered after Wave_u, it will never catch up with Wave_u.

triggered Wave_v, and Wave_v will never catch up with Wave_u as both of them travel at the same speed. See Figure 5.5 for an illustration.

Hence we have an algorithm APSP that is able to trigger all Wave_v processes in $O(n)$ time, without collisions, and each of them completes $O(\mathrm{diam}(G))$ rounds after it was launched. Overall, it takes $O(n)$ rounds for all nodes to learn distances to all other nodes. Finally, the leader can inform everyone else when it is safe to stop and announce the local outputs (e.g., with the help of another wave).

5.8 Quiz

Consider the algorithm in Section 5.7 in a tree. Assume your tree T has 6 nodes, numbered from 1 to 6, and you have already elected node 1 as the leader. You have also already constructed the BFS tree rooted at the leader, which in this case is the original tree T. You would like to know how long it takes in the worst case to run the rest of the algorithms, i.e.,

let the token walk in tree T and let the waves propagate throughout the tree. Your task is to construct a worst-case example of a tree T so that the process takes as long as possible (i.e., you maximize the time until the final node learns about its distance to some other node).

5.9 Exercises

Exercise 5.1 (prior algorithms)**.** In Chapters 3 and 4 we have seen examples of algorithms that were designed for the PN and LOCAL models. Many of these algorithms use only small messages—they can be used directly in the CONGEST model. Give at least four concrete examples of such algorithms, and prove that they indeed use only small messages.

Exercise 5.2 (edge counting)**.** The *edge counting* problem is defined as follows: each node has to output the value $|E|$, i.e., it has to indicate how many edges there are in the graph.

Assume that the input graph is connected. Design an algorithm that solves the edge counting problem in the CONGEST model in time $O(\mathrm{diam}(G))$.

Exercise 5.3 (detecting bipartite graphs)**.** Assume that the input graph is connected. Design an algorithm that solves the following problem in the CONGEST model in time $O(\mathrm{diam}(G))$:

- If the input graph is bipartite, all nodes output 1.
- Otherwise all nodes output 0.

Exercise 5.4 (detecting complete graphs)**.** We say that a graph $G = (V, E)$ is *complete* if for all nodes $u, v \in V$, $u \neq v$, there is an edge $\{u, v\} \in E$.

Assume that the input graph is connected. Design an algorithm that solves the following problem in the CONGEST model in time $O(1)$:

- If the input graph is a complete graph, all nodes output 1.
- Otherwise all nodes output 0.

Exercise 5.5 (gathering)**.** Assume that the input graph is connected. In Section 4.2 we saw how to gather full information on the input graph in time $O(\mathrm{diam}(G))$ in the LOCAL model. Design an algorithm that solves the problem in time $O(|E|)$ in the CONGEST model.

⋆ **Exercise 5.6** (gathering lower bounds)**.** Assume that the input graph is connected. Prove that there is no algorithm that gathers full information on the input graph in time $O(|V|)$ in the CONGEST model.

▷ *hint L*

⋆ **Exercise 5.7** (APSP lower bounds)**.** Assume that the input graph is connected. Prove that there is no algorithm that solves the APSP problem in $o(|V|) + O(\mathrm{diam}(G))$ rounds in the CONGEST model. Put otherwise, show that even if the diameter of the graph is small, you will still need $\Omega(|V|)$ rounds to solve the APSP problem.

5.10 Bibliographic Notes

The name CONGEST is from Peleg's [36] book. Algorithm APSP is due to Holzer and Wattenhofer [24]—surprisingly, it was published only as recently as in 2012.

Chapter 6

Randomized Algorithms

All models of computing that we have studied so far were based on the formalism that we introduced in Chapter 3: a distributed algorithm A is a state machine whose state transitions are determined by functions $\text{init}_{A,d}$, $\text{send}_{A,d}$, and $\text{receive}_{A,d}$. Everything has been fully deterministic: for a given network and a given input, the algorithm will always produce the same output. In this chapter, we will extend the model so that we can study randomized distributed algorithms.

6.1 Definitions

Let us first define a *randomized distributed algorithms in the* PN *model* or, in brief, a *randomized* PN *algorithm*. We extend the definitions of Section 3.3 so that the state transitions are chosen randomly according to some probability distribution that may depend on the current state and incoming messages.

More formally, the values of the functions init and receive are discrete probability distributions over States_A. The initial state of a node u is a random variable $x_0(u)$ chosen from a discrete probability distribution

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

that may depend on the local input $f(u)$. The state at time t is a random variable $x_t(u)$ chosen from a discrete probability distribution

$$\text{receive}_{A,d}\big(x_{t-1}(u), m_t(u)\big)$$

that may depend on the previous state $x_{t-1}(u)$ and on the incoming messages $m_t(u)$. All other parts of the model are as before. In particular, function $\text{send}_{A,d}$ is deterministic.

Above we have defined randomized PN algorithms. We can now extend the definitions in a natural manner to define randomized algorithms in the LOCAL model (add unique identifiers) and randomized algorithms in the CONGEST model (add unique identifiers and limit the size of the messages).

6.2 Probabilistic Analysis

In randomized algorithms, performance guarantees are typically probabilistic. For example, we may claim that algorithm *A stops in time T with probability p*.

Note that all probabilities here are over the random choices in the state transitions. We do not assume that our network or the local inputs are chosen randomly; we still require that the algorithm performs well with worst-case inputs. For example, if we claim that algorithm A solves problem Π on graph family $\mathscr{F}$ in time $T(n)$ with probability p, then we can take *any* graph $G \in \mathscr{F}$ and *any* port-numbered network N with G as its underlying graph, and we guarantee that with probability at least p the execution of A in N stops in time $T(n)$ and produces a correct output $g \in \Pi(G)$; as usual, n is the number of nodes in the network.

We may occasionally want to emphasize the distinction between "Monte Carlo" and "Las Vegas" type algorithms:

- Monte Carlo: Algorithm A always stops in time $T(n)$; the output is a correct solution to problem Π with probability p.
- Las Vegas: Algorithm A stops in time $T(n)$ with probability p; when it stops, the output is always a correct solution to problem Π.

However, Monte Carlo algorithms are not as useful in the field of distributed computing as they are in the context of classical centralized algorithms. In centralized algorithms, we can usually take a Monte Carlo algorithm and just run it repeatedly until it produces a feasible solution; hence we can turn a Monte Carlo algorithm into a Las Vegas algorithm. This is not necessarily the case with distributed algorithms: verifying

the output of an algorithm may require global information on the entire output, and gathering such information may take a long time. In this chapter, we will mainly focus on Las Vegas algorithms, i.e., algorithms that are always correct but may occasionally be slow, but in the exercises we will also encounter Monte Carlo algorithms.

6.3 With High Probability

We will use the word *failure* to refer to the event that the algorithm did not meet its guarantees—in the case of a Las Vegas algorithm, it did not stop in time $T(n)$, and in the case of Monte Carlo algorithms, it did not produce a correct output. The word *success* refers to the opposite case.

Usually we want to show that the probability of a failure is negligible. In computer science, we are usually interested in asymptotic analysis, and hence in the context of randomized algorithms, it is convenient if we can show that the success probability approaches 1 when n increases. Even better, we would like to let the user of the algorithm choose how quickly the success probability approaches 1.

This idea is captured in the phrase "*with high probability*" (commonly abbreviated *w.h.p.*). Please note that this phrase is not a vague subjective statement but it carries a precise mathematical meaning: it refers to the success probability of $1 - 1/n^c$, where we can choose any constant $c > 0$. (Unfortunately, different sources use slightly different definitions; for example, it may also refer to the success probability of $1 - O(1)/n^c$ for any constant $c > 0$.)

In our context, we say that algorithm A solves problem Π on graph family $\mathscr{F}$ in time $O(T(n))$ *with high probability* if the following holds:

- I can choose any constant $c > 0$. Algorithm A may depend on this constant.
- Then if I run A in any network N that has its underlying graph in $\mathscr{F}$, the algorithm will stop in time $O(T(n))$ with probability at least $1 - 1/n^c$, and the output is a feasible solution to problem Π.

Note that the $O(\cdot)$ notation in the running time is used to hide the dependence on c. This is a crucial point. For example, it would not make much sense to say that the running time is at most $\log n$ with probability $1-1/n^c$ for any constant $c > 0$. However, it is perfectly reasonable to say that the running time is, e.g., at most $c \log n$ or $2^c \log n$ or simply $O(\log n)$ with probability $1-1/n^c$ for any constant $c > 0$.

6.4 Randomized Coloring in Bounded-Degree Graphs

In Chapter 4 we presented a *deterministic* algorithm that finds a $(\Delta+1)$-coloring in a graph of maximum degree Δ. In this section, we will design a *randomized* algorithm that solves the same problem. The running times are different:

- the deterministic algorithm runs in $O(\Delta + \log^* n)$ rounds.
- the randomized algorithm runs in $O(\log n)$ rounds with high probability.

Hence for large values of Δ, the randomized algorithm can be much faster.

6.4.1 Algorithm Idea

A running time of $O(\log n)$ is very typical for a randomized distributed algorithm. Often randomized algorithms follow the strategy that in each step each node picks a value randomly from some probability distribution. If the value conflicts with the values of the neighbors, the node will try again next time; otherwise the node outputs the current value and stops. If we can prove that each node stops in each round with a constant probability, we can prove that after $\Theta(\log n)$ all nodes have stopped w.h.p. This is precisely what we saw in the analysis of the randomized path-coloring algorithm in Section 1.5.

However, adapting the same strategy to graphs of maximum degree Δ requires some thought. If each node just repeatedly tries to pick a

random color from $\{1, 2, \ldots, \Delta+1\}$, the success probability may be fairly low for large values of Δ.

Therefore we will adopt a strategy in which nodes are slightly less aggressive. Nodes will first randomly choose whether they are *active* or *passive* in this round; each node is passive with probability $1/2$. Only active nodes will try to pick a random color among those colors that are not yet used by their neighbors.

Informally, the reason why this works well is the following. Assume that we have a node v with d neighbors that have not yet stopped. Then there are at least $d+1$ colors among which v can choose whenever it is active. If all of the d neighbors were also active and if they happened to pick distinct colors, we would have only a

$$\frac{1}{d+1}$$

chance of picking a color that is not used by any of the neighbors. However, in our algorithm on average only $d/2$ neighbors are active. If we have at most $d/2$ active neighbors, we will succeed in picking a free color with probability at least

$$\frac{d+1-d/2}{d+1} > \frac{1}{2},$$

regardless of what the active neighbors do.

6.4.2 Algorithm

Let us now formalize the algorithm. For each node u, let

$$C(u) = \{1, 2, \ldots, \deg_G(u)+1\}$$

be the *color palette* of the node; node u will output one of the colors of $C(u)$.

In the algorithm, node u maintains the following variables:

- State $s(u) \in \{0, 1\}$

- Color $c(u) \in \{\bot\} \cup C(u)$.

Initially, $s(u) \leftarrow 1$ and $c(u) \leftarrow \bot$. When $s(u) = 1$ and $c(u) \neq \bot$, node u stops and outputs color $c(u)$.

In each round, node u always sends $c(u)$ to each port. The incoming messages are processed as follows, depending on the current state of the node:

- $s(u) = 1$ and $c(u) \neq \bot$:
  - This is a stopping state; ignore incoming messages.
- $s(u) = 1$ and $c(u) = \bot$:
  - Let $M(u)$ be the set of messages received.
  - Let $F(u) = C(u) \setminus M(u)$ be the set of *free colors*.
  - With probability $1/2$, set $c(u) \leftarrow \bot$; otherwise choose a $c(u) \in F(u)$ uniformly at random.
  - Set $s(u) \leftarrow 0$.
- $s(u) = 0$:
  - Let $M(u)$ be the set of messages received.
  - If $c(u) \in M(u)$, set $c(u) \leftarrow \bot$.
  - Set $s(u) \leftarrow 1$.

Informally, the algorithm proceeds as follows. For each node u, its state $s(u)$ alternates between 1 and 0:

- When $s(u) = 1$, the node either decides to be *passive* and sets $c(u) = \bot$, or it decides to be *active* and picks a random color $c(u) \in F(u)$. Here $F(u)$ is the set of colors that are not yet used by any of the neighbors that are stopped.
- When $s(u) = 0$, the node *verifies* its choice. If the current color $c(u)$ conflicts with one of the neighbors, we go back to the initial state $s(u) \leftarrow 1$ and $c(u) \leftarrow \bot$. However, if we were lucky and managed to pick a color that does not conflict with any of our neighbors, we keep the current value of $c(u)$ and switch to the stopping state.

6.4.3 Analysis

It is easy to see that if the algorithm stops, then the output is a proper $(\Delta+1)$-coloring of the underlying graph. Let us now analyze how long it takes for the nodes to stop.

In the analysis, we will write $s_t(u)$ and $c_t(u)$ for values of variables $s(u)$ and $c(u)$ after round $t = 0, 1, \ldots$, and $M_t(u)$ and $F_t(u)$ for the values of $M(u)$ and $F(u)$ during round $t = 1, 2, \ldots$. We also write

$$K_t(u) = \{v \in V : \{u, v\} \in E,\ s_{t-1}(v) = 1, c_{t-1}(v) = \bot\}$$

for the set of *competitors* of node u during round $t = 1, 3, 5, \ldots$; these are the neighbors of u that have not yet stopped.

First, let us prove that with probability at least $1/4$, a running node succeeds in picking a color that does not conflict with any of its neighbors.

Lemma 6.1. *Fix a node $u \in V$ and time $t = 1, 3, 5, \ldots$. Assume that $s_{t-1}(u) = 1$ and $c_{t-1}(u) = \bot$, i.e., u has not stopped before round t. Then with probability at least $1/4$, we have $s_{t+1}(u) = 1$ and $c_{t+1}(u) \neq \bot$, i.e., u will stop after round $t+1$.*

Proof. Let $f = |F_t(u)|$ be the number of free colors during round t, and let $k = |K_t(u)|$ be the number of competitors during round t. Note that $f \geq k + 1$, as the size of the palette is one larger than the number of neighbors.

Let us first study the case that u is active. As we have f free colors, for any given color $x \in \mathbb{N}$ we have

$$\Pr\left[c_t(u) = x \;\middle|\; c_t(u) \neq \bot\right] \leq 1/f.$$

In particular, this holds for any color $x = c_t(v)$ chosen by any active competitor $v \in K_t(u)$:

$$\Pr\left[c_t(u) = c_t(v) \;\middle|\; c_t(u) \neq \bot,\ c_t(v) \neq \bot\right] \leq 1/f.$$

That is, we conflict with an active competitor with probability at most $1/f$. Naturally, we cannot conflict with a passive competitor:

$$\Pr\left[c_t(u) = c_t(v) \;\middle|\; c_t(u) \neq \bot,\ c_t(v) = \bot\right] = 0.$$

As a competitor is active with probability

$$\Pr\big[c_t(v) \neq \perp\big] = 1/2,$$

and the random variables $c_t(u)$ and $c_t(v)$ are independent, the probability that we conflict with a given competitor $v \in K_t(u)$ is

$$\Pr\big[c_t(u) = c_t(v) \;\big|\; c_t(u) \neq \perp\big] \leq \frac{1}{2f}.$$

By the union bound, the probability that we conflict with some competitor is

$$\Pr\big[c_t(u) = c_t(v) \text{ for some } v \in K_t(u) \;\big|\; c_t(u) \neq \perp\big] \leq \frac{k}{2f},$$

which is less than $1/2$ for all $k \geq 0$ and all $f \geq k+1$. Put otherwise, node u will avoid conflicts with probability

$$\Pr\big[c_t(u) \neq c_t(v) \text{ for all } v \in K_t(u) \;\big|\; c_t(u) \neq \perp\big] > \frac{1}{2}.$$

So far we have studied the conditional probabilities assuming that u is active. This happens with probability

$$\Pr\big[c_t(u) \neq \perp\big] = 1/2.$$

Therefore node u will stop after round $t+1$ with probability

$$\begin{aligned}&\Pr\big[c_{t+1}(u) \neq \perp\big] = \\ &\Pr\big[c_t(u) \neq \perp \text{ and } c_t(u) \neq c_t(v) \text{ for all } v \in K_t(u)\big] > 1/4.\end{aligned}$$ □

Now we can continue with the same argument as what we used in Section 1.5 to analyze the running time. Fix a constant $c > 0$. Define

$$T(n) = 2(c+1)\log_{4/3} n.$$

We will prove that the algorithm stops in $T(n)$ rounds. First, let us consider an individual node. Note the exponent $c+1$ instead of c in the statement of the lemma; this will be helpful later.

Lemma 6.2. *Fix a node $u \in V$. The probability that u has not stopped after $T(n)$ rounds is at most $1/n^{c+1}$.*

Proof. By Lemma 6.1, if node u has not stopped after round $2i$, it will stop after round $2i+2$ with probability at least $1/4$. Hence the probability that it has not stopped after $T(n)$ rounds is at most

$$(3/4)^{T(n)/2} = \frac{1}{(4/3)^{(c+1)\log_{4/3} n}} = \frac{1}{n^{c+1}}.$$ □

Now we are ready to analyze the time until all nodes stop.

Theorem 6.3. *The probability that all nodes have stopped after $T(n)$ rounds is at least $1 - 1/n^c$.*

Proof. Follows from Lemma 6.2 by the union bound. □

Note that $T(n) = O(\log n)$ for any constant c. Hence we conclude that the algorithm stops in $O(\log n)$ rounds with high probability, and when it stops, it outputs a vertex coloring with $\Delta + 1$ colors.

6.5 Quiz

Consider a cycle with 10 nodes, and label the nodes with a random permutation of the numbers $1, 2, \ldots, 10$ (uniformly at random). A node is a *local maximum* if its label is larger than the labels of its two neighbors. Let X be the number of local maxima. What is the expected value of X?

6.6 Exercises

Exercise 6.1 (larger palette)**.** Assume that we have a graph without any isolated nodes. We will design a graph-coloring algorithm A that is a bit easier to understand and analyze than the algorithm of Section 6.4. In algorithm A, each node u proceeds as follows until it stops:

- Node u picks a color $c(u)$ from $\{1, 2, \ldots, 2d\}$ uniformly at random; here d is the degree of node u.

- Node u compares its value $c(u)$ with the values of all neighbors. If $c(u)$ is different from the values of its neighbors, u outputs $c(u)$ and stops.

Present this algorithm in a formally precise manner, using the state-machine formalism. Analyze the algorithm, and prove that it finds a 2Δ-coloring in time $O(\log n)$ with high probability.

Exercise 6.2 (unique identifiers)**.** Design a randomized PN algorithm A that solves the following problem in $O(1)$ rounds:

- As input, all nodes get value $|V|$.
- Algorithm outputs a labeling $f : V \to \{1, 2, \ldots, \chi\}$ for some $\chi = |V|^{O(1)}$.
- With high probability, $f(u) \neq f(v)$ for all nodes $u \neq v$.

Analyze your algorithm and prove that it indeed solves the problem correctly.

In essence, algorithm A demonstrates that we can use randomness to construct unique identifiers, assuming that we have some information on the size of the network. Hence we can take any algorithm B designed for the LOCAL model, and combine it with algorithm A to obtain a PN algorithm B' that solves the same problem as B (with high probability).

▷ *hint M*

Exercise 6.3 (large independent sets)**.** Design a randomized PN algorithm A with the following guarantee: in any graph $G = (V, E)$ of maximum degree Δ, algorithm A outputs an independent set I such that the *expected* size of the I is $|V|/O(\Delta)$. The running time of the algorithm should be $O(1)$. You can assume that all nodes know Δ.

▷ *hint N*

Exercise 6.4 (max cut problem)**.** Let $G = (V, E)$ be a graph. A *cut* is a function $f : V \to \{0, 1\}$. An edge $\{u, v\} \in E$ is a *cut edge* in f if $f(u) \neq f(v)$. The *size* of cut f is the number of cut edges, and a *maximum cut* is a cut of the largest possible size.

(a) Prove: If $G = (V, E)$ is a bipartite graph, then a maximum cut has $|E|$ edges.

(b) Prove: If $G = (V, E)$ has a cut with $|E|$ edges, then G is bipartite.

(c) Prove: For any $\alpha > 1/2$, there exists a graph $G = (V, E)$ in which the maximum cut has fewer than $\alpha|E|$ edges.

▷ *hint O*

Exercise 6.5 (max cut algorithm)**.** Design a randomized PN algorithm A with the following guarantee: in any graph $G = (V, E)$, algorithm A outputs a cut f such that the *expected* size of cut f is at least $|E|/2$. The running time of the algorithm should be $O(1)$.

Note that the analysis of algorithm A also implies that for any graph there *exists* a cut with at least $|E|/2$.

▷ *hint P*

Exercise 6.6 (maximal independent sets)**.** Design a randomized PN algorithm that finds a maximal independent set in time $O(\Delta + \log n)$ with high probability.

▷ *hint Q*

⋆ **Exercise 6.7** (maximal independent sets quickly)**.** Design a randomized distributed algorithm that finds a maximal independent set in time $O(\log n)$ with high probability.

▷ *hint R*

6.7 Bibliographic Notes

Algorithm of Section 6.4 and the algorithm of Exercise 6.1 are from Barenboim and Elkin's book [7, Section 10.1].

Part IV

Proving Impossibility Results

Chapter 7

Covering Maps

Chapters 3–6 have focused on positive results; now we will turn our attention to techniques that can be used to prove negative results. We will start with so-called covering maps—we will use covering maps to prove that many problems cannot be solved at all with deterministic PN-algorithms.

7.1 Definition

A covering map is a topological concept that finds applications in many areas of mathematics, including graph theory. We will focus on one special case: covering maps between port-numbered networks.

Let $N = (V, P, p)$ and $N' = (V', P', p')$ be port-numbered networks, and let $\phi : V \to V'$. We say that ϕ is a *covering map from N to N'* if the following holds:

(a) ϕ is a surjection: $\phi(V) = V'$.

(b) ϕ preserves degrees: $\deg_N(v) = \deg_{N'}(\phi(v))$ for all $v \in V$.

(c) ϕ preserves connections and port numbers: $p(u, i) = (v, j)$ implies $p'(\phi(u), i) = (\phi(v), j)$.

See Figures 7.1–7.3 for examples.

We can also consider labeled networks, for example, networks with local inputs. Let $f : V \to X$ and $f' : V' \to X$. We say that ϕ is a covering map from (N, f) to (N', f') if ϕ is a covering map from N to N' and the following holds:

(d) ϕ preserves labels: $f(v) = f'(\phi(v))$ for all $v \in V$.

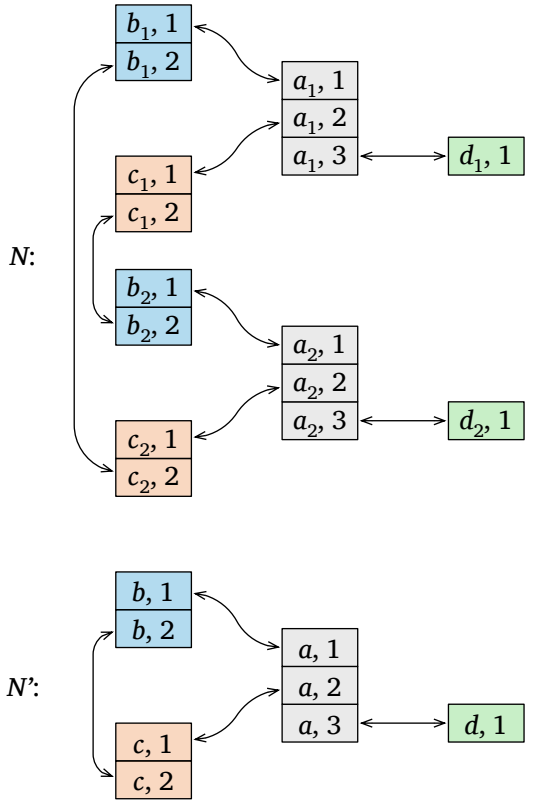


Figure 7.1: There is a covering map ϕ from N to N' that maps $a_i \mapsto a$, $b_i \mapsto b$, $c_i \mapsto c$, and $d_i \mapsto d$ for each $i \in \{1, 2\}$.

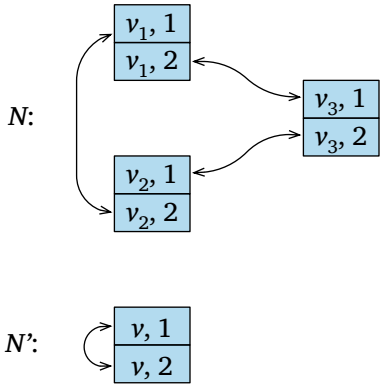


Figure 7.2: There is a covering map ϕ from N to N' that maps $v_i \mapsto v$ for each $i \in \{1,2,3\}$. Here N is a simple port-numbered network but N' is not.

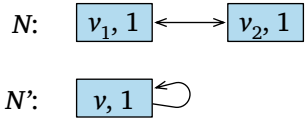


Figure 7.3: There is a covering map ϕ from N to N' that maps $v_i \mapsto v$ for each $i \in \{1,2\}$. Again, N is a simple port-numbered network but N' is not.

7.2 Covers and Executions

Now we will study covering maps from the perspective of deterministic PN-algorithms. The basic idea is that a covering map ϕ from N to N' fools any PN-algorithm A: a node v in N is indistinguishable from the node $\phi(v)$ in N'.

Without further ado, we state the main result and prove it—many applications and examples will follow.

Theorem 7.1. *Assume that*

(a) *A is a deterministic* PN*-algorithm with $X = \mathrm{Input}_A$,*

(b) *$N = (V, P, p)$ and $N' = (V', P', p')$ are port-numbered networks,*

(c) *$f : V \to X$ and $f' : V' \to X$ are arbitrary functions, and*

(d) *$\phi : V \to V'$ is a covering map from (N, f) to (N', f').*

Let

(e) *$x_0, x_1, \ldots$ be the execution of A on (N, f), and*

(f) *$x'_0, x'_1, \ldots$ be the execution of A on (N', f').*

Then for each $t = 0, 1, \ldots$ and each $v \in V$ we have $x_t(v) = x'_t(\phi(v))$.

Proof. We will use the notation of Section 3.3.2; the symbols with a prime refer to the execution of A on (N', f'). In particular, $m'_t(u', i)$ is the message received by $u' \in V'$ from port i in round t in the execution of A on (N', f'), and $m'_t(u')$ is the vector of messages received by u'.

The proof is by induction on t. To prove the base case $t = 0$, let $v \in V$, $d = \deg_N(v)$, and $v' = \phi(v)$; we have

$$x'_0(v') = \mathrm{init}_{A,d}(f'(v')) = \mathrm{init}_{A,d}(f(v)) = x_0(v).$$

For the inductive step, let $(u, i) \in P$, $(v, j) = p(u, i)$, $d = \deg_N(u)$, $\ell = \deg_N(v)$, $u' = \phi(u)$, and $v' = \phi(v)$. Let us first consider the messages sent by v and v'; by the inductive assumption, these are equal:

$$\mathrm{send}_{A,\ell}(x'_{t-1}(v')) = \mathrm{send}_{A,\ell}(x_{t-1}(v)).$$

A covering map ϕ preserves connections and port numbers: $(u,i) = p(v,j)$ implies $(u',i) = p'(v',j)$. Hence $m_t(u,i)$ is component j of $\mathrm{send}_{A,\ell}(x_{t-1}(v))$, and $m'_t(u',i)$ is component j of $\mathrm{send}_{A,\ell}(x'_{t-1}(v'))$. It follows that $m_t(u,i) = m'_t(u',i)$ and $m_t(u) = m'_t(u')$. Therefore

$$\begin{aligned} x'_t(u') &= \mathrm{receive}_{A,d}\big(x'_{t-1}(u'), m'_t(u')\big) \\ &= \mathrm{receive}_{A,d}\big(x_{t-1}(u), m_t(u)\big) = x_t(u). \end{aligned}$$ □

In particular, if the execution of A on (N,f) stops in time T, the execution of A on (N',f') stops in time T as well, and vice versa. Moreover, ϕ preserves the local outputs: $x_T(v) = x'_T(\phi(v))$ for all $v \in V$.

7.3 Examples

We will give representative examples of negative results that we can easily derive from Theorem 7.1. First, we will observe that a deterministic PN-algorithm cannot break symmetry in a cycle—unless we provide some symmetry-breaking information in local inputs.

Lemma 7.2. *Let $G = (V,E)$ be a cycle graph, let A be a deterministic* PN*-algorithm, and let f be a constant function $f : V \to \{0\}$. Then there is a simple port-numbered network $N = (V,P,p)$ such that*

(a) the underlying graph of N is G, and

(b) if A stops on (N,f), the output is a constant function $g : V \to \{c\}$ for some c.

Proof. Label the nodes $V = \{v_1, v_2, \ldots, v_n\}$ along the cycle so that the edges are

$$E = \big\{ \{v_1,v_2\},\ \{v_2,v_3\},\ \ldots,\ \{v_{n-1},v_n\},\ \{v_n,v_1\} \big\}.$$

Choose the port numbering p as follows:

$$\begin{aligned} p : (v_1,1) &\mapsto (v_2,2),\ (v_2,1) \mapsto (v_3,2),\ \ldots, \\ (v_{n-1},1) &\mapsto (v_n,2),\ (v_n,1) \mapsto (v_1,2). \end{aligned}$$

See Figure 7.2 for an illustration in the case $n = 3$.

Define another port-numbered network $N' = (V', P', p')$ with $V' = \{v\}$, $P' = \{(v,1),(v,2)\}$, and $p(v,1) = (v,2)$. Let $f' : V' \to \{0\}$. Define a function $\phi : V \to V'$ by setting $\phi(v_i) = v$ for each i.

Now we can verify that ϕ is a covering map from (N, f) to (N', f'). Assume that A stops on (N, f) and produces an output g. By Theorem 7.1, A also stops on (N', f') and produces an output g'. Let $c = g'(v)$. Now

$$g(v_i) = g'(\phi(v_i)) = g'(v) = c$$

for all i. □

In the above proof, we never assumed that the execution of A on N' makes any sense—after all, N' is not even a simple port-numbered network, and there is no underlying graph. Algorithm A was never designed to be applied to such a strange network with only one node. Nevertheless, the execution of A on N' is formally well-defined, and Theorem 7.1 holds. We do not really care what A outputs on N', but the existence of a covering map can be used to prove that the output of A on N has certain properties. It may be best to interpret the execution of A on N' as a thought experiment, not as something that we would actually try to do in practice.

Lemma 7.2 has many immediate corollaries.

Corollary 7.3. *Let $\mathcal{F}$ be the family of cycle graphs. Then there is no deterministic* PN*-algorithm that solves any of the following problems on $\mathcal{F}$:*

(a) maximal independent set,
(b) 1.999-approximation of a minimum vertex cover,
(c) 2.999-approximation of a minimum dominating set,
(d) maximal matching,
(e) vertex coloring,
(f) weak coloring,
(g) edge coloring.

Proof. In each of these cases, there is a graph $G \in \mathscr{F}$ such that a constant function is not a feasible solution in the network N that we constructed in Lemma 7.2.

For example, consider the case of dominating sets; other cases are similar. Assume that $G = (V, E)$ is a cycle with $3k$ nodes. Then a minimum dominating set consists of k nodes—it is sufficient to take every third node. Hence a 2.999-approximation of a minimum dominating set consists of at most $2.999k < 3k$ nodes. A solution $D = V$ violates the approximation guarantee, as D has too many nodes, while $D = \varnothing$ is not a dominating set. Hence if A outputs a constant function, it cannot produce a 2.999-approximation of a minimum dominating set. □

Lemma 7.4. *There is no deterministic* PN*-algorithm that finds a weak coloring for every* 3*-regular graph.*

Proof. Again, we are going to apply the standard technique: pick a suitable 3-regular graph G, find a port-numbered network N that has G as its underlying graph, find a smaller network N' such that we have a covering map ϕ from N to N', and apply Theorem 7.1.

However, it is not immediately obvious which 3-regular graph would be appropriate; hence we try the simplest possible case first. Let $G = (V, E)$ be the *complete graph* on four nodes: $V = \{s, t, u, v\}$, and we have an edge between any pair of nodes; see Figure 7.4. The graph is certainly 3-regular: each node is adjacent to the other three nodes.

Now it is easy to verify that the edges of G can be partitioned into a 2-factor X and a 1-factor Y. The 2-factor consists of a cycle and a 1-factor consists of disjoint edges. We can use the factors to guide the selection of port numbers in N.

In the cycle induced by X, we can choose symmetric port numbers using the same idea as what we had in the proof of Lemma 7.2; one end of each edge is connected to port 1 while the other end is connected to port 2. For the edges of the 1-factor Y, we can assign port number 3 at each end. We have constructed the port-numbered network N that is illustrated in Figure 7.4.

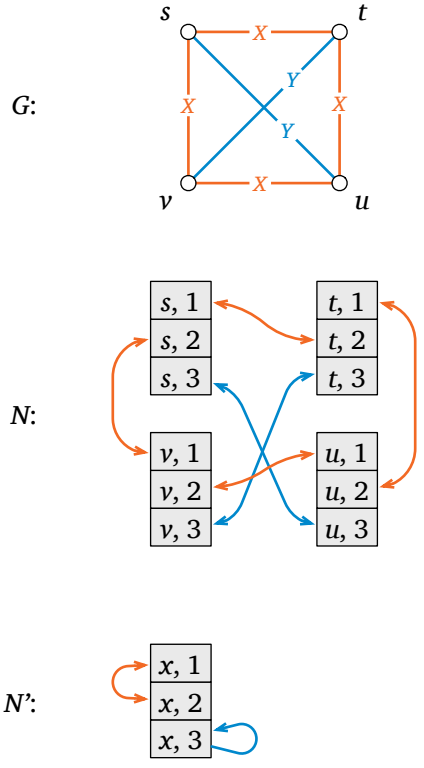


Figure 7.4: Graph G is the complete graph on four nodes. The edges of G can be partitioned into a 2-factor X and a 1-factor Y. Network N has G as its underlying graph, and there is a covering map ϕ from N to N'

Now we can verify that there is a covering map ϕ from N to N', where N' is the network with one node illustrated in Figure 7.4. Therefore in any algorithm A, if we do not have any local inputs, all nodes of N will produce the same output. However, a constant output is not a weak coloring of G. □

In the above proof, we could have also partitioned the edges of G into three 1-factors, and we could have used the 1-factorization to guide the selection of port numbers. However, the above technique is more general: there are 3-regular graphs that do not admit a 1-factorization but that can be partitioned into a 1-factor and a 2-factor.

So far we have used only one covering map in our proofs; the following lemma gives an example of the use of more than one covering map.

Lemma 7.5. *Let $\mathcal{F} = \{G_3, G_4\}$, where G_3 is the cycle graph with 3 nodes, and G_4 is the cycle graph with 4 nodes. There is no deterministic* PN*-algorithm that solves the following problem Π on $\mathcal{F}$: in $\Pi(G_3)$ all nodes output 3 and in $\Pi(G_4)$ all nodes output 4.*

Proof. We again apply the construction of Lemma 7.2; for each $i \in \{3, 4\}$, let N_i be the symmetric port-numbered network that has G_i as the underlying graph.

Now it would be convenient if we could construct a covering map from N_4 to N_3; however, this is not possible (see the exercises). Therefore we proceed as follows. Construct a one-node network N' as in the proof of Lemma 7.2, construct the covering map ϕ_3 from N_3 to N', and construct the covering map ϕ_4 from N_4 to N'; see Figure 7.5. The local inputs are assumed to be all zeros.

Let A be a PN-algorithm, and let c be the output of the only node of N'. If we apply Theorem 7.1 to ϕ_3, we conclude that all nodes of N_3 output c; if A solves Π on G_3, we must have $c = 3$. However, if we apply Theorem 7.1 to ϕ_4, we learn that all nodes of N_4 also output $c = 3$, and hence A cannot solve Π on $\mathcal{F}$. □

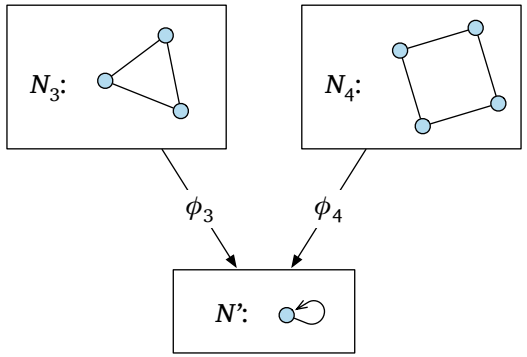


Figure 7.5: The structure of the proof of Lemma 7.5.

We have learned that a deterministic PN-algorithm cannot determine the length of a cycle. In particular, a deterministic PN-algorithm cannot determine if the underlying graph is bipartite.

7.4 Quiz

Let $G = (V, E)$ be a graph. A set $X \subseteq V$ is a *k-tuple dominating set* if for every $v \in V$ we have $|\mathrm{ball}_G(v, 1) \cap X| \geq k$. Consider the problem of finding a minimum 2-tuple dominating set in *cycles*. What is the best (i.e. smallest) approximation ratio we can achieve in the PN model?

7.5 Exercises

We use the following definition in the exercises. A graph G is *homogeneous* if there are port-numbered networks N and N' and a covering map ϕ from N to N' such that N is simple, the underlying graph of N is G, and N' has only one node. For example, Lemma 7.2 shows that all cycle graphs are homogeneous.

Exercise 7.1 (finding port numbers)**.** Consider the graph G and network N' illustrated in Figure 7.6. Find a simple port-numbered network N such that N has G as the underlying graph and there is a covering map from N to N'.

Exercise 7.2 (homogeneity)**.** Assume that G is homogeneous and it contains a node of degree at least two. Give several examples of graph problems that cannot be solved with any deterministic PN-algorithm in any family of graphs that contains G.

Exercise 7.3 (regular and homogeneous)**.** Show that the following graphs are homogeneous:

(a) graph G illustrated in Figure 7.7,

(b) graph G illustrated in Figure 7.6.

▷ *hint S*

Exercise 7.4 (complete graphs)**.** Recall that we say that a graph $G = (V, E)$ is *complete* if for all nodes $u, v \in V$, $u \neq v$, there is an edge $\{u, v\} \in E$. Show that

(a) any $2k$-regular graph is homogeneous,

(b) any complete graph with $2k$ nodes has a 1-factorization,

(c) any complete graph is homogeneous.

Exercise 7.5 (dominating sets)**.** Let $\Delta \in \{2, 3, \dots\}$, let $\epsilon > 0$, and let $\mathcal{F}$ consist of all graphs of maximum degree at most Δ. Show that it is possible to find a $(\Delta + 1)$-approximation of a minimum dominating set in constant time in family $\mathcal{F}$ with a deterministic PN-algorithm. Show that it is not possible to find a $(\Delta + 1 - \epsilon)$-approximation with a deterministic PN-algorithm.

▷ *hint T*

Exercise 7.6 (covers with covers)**.** What is the connection between covering maps and the vertex cover 3-approximation algorithm in Section 3.6?

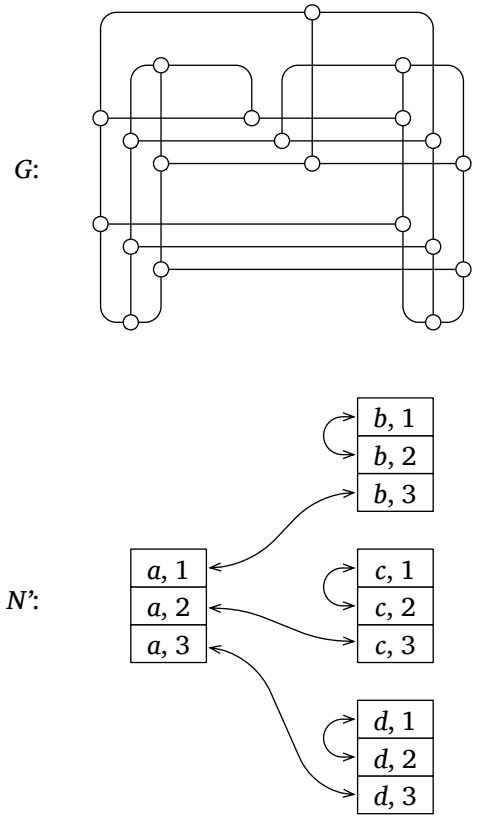


Figure 7.6: Graph G and network N' for Exercises 7.1 and 7.3b.

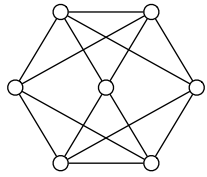


Figure 7.7: Graph G for Exercise 7.3a.

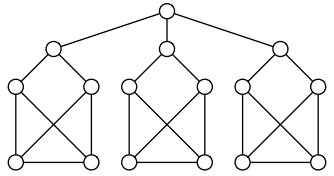


Figure 7.8: Graph G for Exercise 7.7.

⋆ **Exercise 7.7** (3-regular and not homogeneous)**.** Consider the graph G illustrated in Figure 7.8.

(a) Show that G is not homogeneous.

(b) Present a deterministic PN-algorithm A with the following property: if N is a simple port-numbered network that has G as the underlying graph, and we execute A on N, then A stops and produces an output where at least one node outputs 0 and at least one node outputs 1.

(c) Find a simple port-numbered network N that has G as the underlying graph, a port-numbered network N', and a covering map ϕ from N to N' such that N' has the smallest possible number of nodes.

▷ *hint U*

⋆ **Exercise 7.8** (covers and connectivity)**.** Assume that $N = (V, P, p)$ and $N' = (V', P', p')$ are simple port-numbered networks such that there is a covering map ϕ from N to N'. Let G be the underlying graph of network N, and let G' be the underlying graph of network N'.

(a) Is it possible that G is connected and G' is not connected?

(b) Is it possible that G is not connected and G' is connected?

⋆ **Exercise 7.9** (k-fold covers)**.** Let $N = (V, P, p)$ and $N' = (V', P', p')$ be simple port-numbered networks such that the underlying graphs of N and N' are connected, and assume that $\phi : V \to V'$ is a covering map

from N to N'. Prove that there exists a positive integer k such that the following holds: $|V| = k|V'|$ and for each node $v' \in V'$ we have $|\phi^{-1}(v')| = k$. Show that the claim does not necessarily hold if the underlying graphs are not connected.

7.6 Bibliographic Notes

The use of covering maps in the context of distributed algorithm was introduced by Angluin [3]. The general idea of Exercise 7.7 can be traced back to Yamashita and Kameda [42], while the specific construction in Figure 7.8 is from Bondy and Murty's textbook [9, Figure 5.10]. Parts of exercises 7.1, 7.3, 7.4, and 7.5 are inspired by our work [4, 39].

Chapter 8

Local Neighborhoods

Covering maps can be used to argue that a given problem cannot be solved *at all* with deterministic PN algorithms. Now we will study the concept of *locality*, which can be used to argue that a given problem cannot be solved *fast*, in any model of distributed computing.

8.1 Definitions

Let $N = (V, P, p)$ and $N' = (V', P', p')$ be simple port-numbered networks, with the underlying graphs $G = (V, E)$ and $G' = (V', E')$. Fix the local inputs $f : V \to Y$ and $f' : V' \to Y$, a pair of nodes $v \in V$ and $v' \in V'$, and a radius $r \in \mathbb{N}$. Define the radius-r neighborhoods

$$U = \mathrm{ball}_G(v, r), \quad U' = \mathrm{ball}_{G'}(v', r).$$

We say that (N, f, v) and (N', f', v') have *isomorphic radius-r neighborhoods* if there is a bijection $\psi : U \to U'$ with $\psi(v) = v'$ such that

(a) ψ preserves degrees: $\deg_N(v) = \deg_{N'}(\psi(v))$ for all $v \in U$.

(b) ψ preserves connections and port numbers: $p(u, i) = (v, j)$ if and only if $p'(\psi(u), i) = (\psi(v), j)$ for all $u, v \in U$.

(c) ψ preserves local inputs: $f(v) = f'(\psi(v))$ for all $v \in U$.

The function ψ is called an *r-neighborhood isomorphism from (N, f, v) to (N', f', v')*. See Figure 8.1 for an example.

8.2 Local Neighborhoods and Executions

Theorem 8.1. *Assume that*

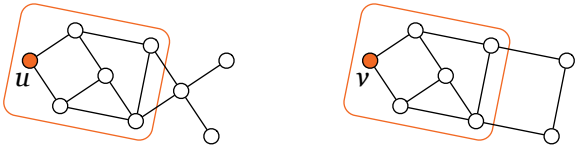


Figure 8.1: Nodes u and v have isomorphic radius-2 neighborhoods, provided that we choose the port numbers appropriately. Therefore in any algorithm A the state of u equals the state of v at time $t = 0, 1, 2$. However, at time $t = 3, 4, \ldots$ this does not necessarily hold.

(a) *A is a deterministic* PN *algorithm with $X = \mathrm{Input}_A$,*

(b) *$N = (V, P, p)$ and $N' = (V', P', p')$ are simple port-numbered networks,*

(c) *$f : V \to X$ and $f' : V' \to X$ are arbitrary functions,*

(d) *$v \in V$ and $v' \in V'$,*

(e) *(N, f, v) and (N', f', v') have isomorphic radius-r neighborhoods.*

Let

(f) *$x_0, x_1, \ldots$ be the execution of A on (N, f), and*

(g) *$x'_0, x'_1, \ldots$ be the execution of A on (N', f').*

Then for each $t = 0, 1, \ldots, r$ we have $x_t(v) = x'_t(v')$.

Proof. Let G and G' be the underlying graphs of N and N', respectively. We will prove the following stronger claim by induction: for each $t = 0, 1, \ldots, r$, we have $x_t(u) = x'_t(\psi(u))$ for all $u \in \mathrm{ball}_G(v, r - t)$.

To prove the base case $t = 0$, let $u \in \mathrm{ball}_G(v, r)$, $d = \deg_N(u)$, and $u' = \psi(u)$; we have

$$x'_0(u') = \mathrm{init}_{A,d}(f'(u')) = \mathrm{init}_{A,d}(f(u)) = x_0(u).$$

For the inductive step, assume that $t \geq 1$ and

$$u \in \mathrm{ball}_G(v, r - t).$$

Let $u' = \psi(u)$. By inductive assumption, we have

$$x'_{t-1}(u') = x_{t-1}(u).$$

Now consider a port $(u, i) \in P$. Let $(s, j) = p(u, i)$. We have $\{s, u\} \in E$, and therefore

$$\mathrm{dist}_G(s, v) \leq \mathrm{dist}_G(s, u) + \mathrm{dist}_G(u, v) \leq 1 + r - t.$$

Define $s' = \psi(s)$. By inductive assumption we have

$$x'_{t-1}(s') = x_{t-1}(s).$$

The neighborhood isomorphism ψ preserves the port numbers: $(s', j) = p'(u', i)$. Hence all of the following are equal:

(a) the message sent by s to port j on round t,
(b) the message sent by s' to port j on round t,
(c) the message received by u from port i on round t,
(d) the message received by u' from port i on round t.

As the same holds for any port of u, we conclude that

$$x'_t(u') = x_t(u). \qquad \square$$

To apply Theorem 8.1 in the LOCAL model, we need to include unique identifiers in the local inputs f and f'.

8.3 Example: 2-Coloring Paths

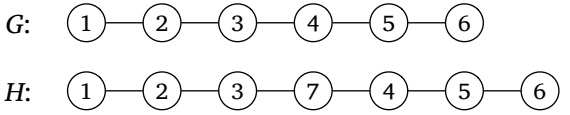
We know from Chapter 1 that one can find a proper 3-coloring of a path very fast, in $O(\log^* n)$ rounds. Now we will show that 2-coloring is much harder; it requires linear time.

To reach a contradiction, suppose that there is a deterministic distributed algorithm A that finds a proper 2-coloring of any path graph in $o(n)$ rounds in the LOCAL model. Then there has to be a number n_0

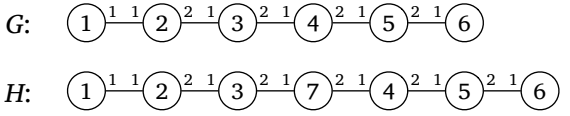
such that for any number of nodes $n \geq n_0$, the running time of algorithm A is at most $(n-3)/2$. Pick some integer $k \geq n_0/2$, and consider two paths: path G contains $2k$ nodes, with unique identifiers $1, 2, \ldots, 2k$, and path H contains $2k+1$ nodes, with unique identifiers

$$1, 2, \ldots, k, 2k+1, k+1, k+2, \ldots, 2k.$$

Here is an example for $k = 3$:



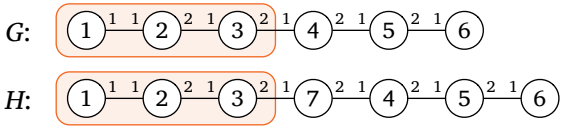
We assign the port numbers so that for all degree-2 nodes port number 1 points towards node 1:



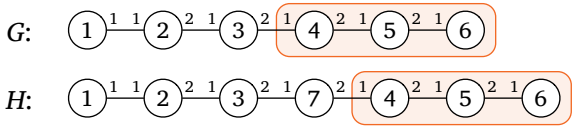
By assumption, the running time of A is at most

$$(n-3)/2 \leq (2k+1-3)/2 = k-1$$

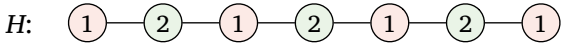
rounds in both cases. Since node 1 has the same radius-$(k-1)$ neighborhood in G and H, algorithm A will produce the same output for node 1 in both networks:



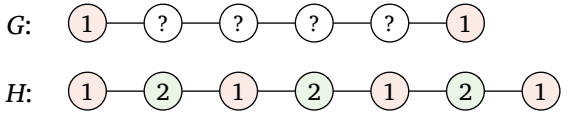
By a similar reasoning, node $2k$ (i.e., the last node of the path) also has to produce the same output in both cases:



However, now we reach a contradiction. In path H, in any proper 2-coloring nodes 1 and $2k$ have the same color—for example, both of them are of color 1, as shown in the following picture:



If algorithm A works correctly, it follows that nodes 1 and $2k$ must produce the same output in path H. However, then it follows that nodes 1 and $2k$ produces the same output also in G, too, but this cannot happen in any proper 2-coloring of G.



We conclude that algorithm A fails to find a proper 2-coloring in at least one of these instances.

8.4 Quiz

Let N a simple port-numbered network with 1000 nodes, such that the underlying graph of N is a cycle. Form another network N' by adding one edge to N. Let A be a LOCAL-model algorithm that runs in 100 rounds. Let f be the output of A in network N, and let f' be the output of A in network N'. At most how many nodes there can be such that their output differs in f and f'?

8.5 Exercises

Exercise 8.1 (edge coloring)**.** In this exercise, the graph family $\mathscr{F}$ consists of *path graphs*.

(a) Show that it is possible to find a 2-edge coloring in time $O(n)$ with deterministic PN-algorithms.

(b) Show that it is not possible to find a 2-edge coloring in time $o(n)$ with deterministic PN-algorithms.

(c) Show that it is not possible to find a 2-edge coloring in time $o(n)$ with deterministic LOCAL-algorithms.

Exercise 8.2 (maximal matching)**.** In this exercise, the graph family $\mathscr{F}$ consists of *path graphs*.

(a) Show that it is possible to find a maximal matching in time $O(n)$ with deterministic PN-algorithms.

(b) Show that it is not possible to find a maximal matching in time $o(n)$ with deterministic PN-algorithms.

(c) Show that it is possible to find a maximal matching in time $o(n)$ with deterministic LOCAL-algorithms.

Exercise 8.3 (optimization)**.** In this exercise, the graph family $\mathscr{F}$ consists of *path graphs*. Can we solve the following problems with deterministic PN-algorithms? If yes, how fast? Can we solve them any faster in the LOCAL model?

(a) Minimum vertex cover.

(b) Minimum dominating set.

(c) Minimum edge dominating set.

Exercise 8.4 (approximation)**.** In this exercise, the graph family $\mathscr{F}$ consists of *path graphs*. Can we solve the following problems with deterministic PN-algorithms? If yes, how fast? Can we solve them any faster in the LOCAL model?

(a) 2-approximation of a minimum vertex cover?

(b) 2-approximation of a minimum dominating set?

Exercise 8.5 (auxiliary information)**.** In this exercise, the graph family $\mathscr{F}$ consists of *path graphs*, and we are given a 4-coloring as input. We consider deterministic PN-algorithms.

(a) Show that it is possible to find a 3-coloring in time 1.

(b) Show that it is not possible to find a 3-coloring in time 0.

(c) Show that it is possible to find a 2-coloring in time $O(n)$.

(d) Show that it is not possible to find a 2-coloring in time $o(n)$.

⋆ **Exercise 8.6** (orientations)**.** In this exercise, the graph family $\mathscr{F}$ consists of *cycle graphs*, and we are given some *orientation* as input. The task is to find a *consistent orientation*, i.e., an orientation such that both the indegree and the outdegree of each node is 1.

(a) Show that this problem cannot be solved with any deterministic PN-algorithm.

(b) Show that this problem cannot be solved with any deterministic LOCAL-algorithm in time $o(n)$.

(c) Show that this problem can be solved with a deterministic PN-algorithm if we give n as input to all nodes. How fast? Prove tight upper and lower bounds on the running time.

⋆ **Exercise 8.7** (local indistinguishability)**.** Consider the graphs G_1 and G_2 illustrated in Figure 8.2. Assume that A is a deterministic PN-algorithm with running time 2. Show that A cannot distinguish between nodes v_1 and v_2. That is, there are simple port-numbered networks N_1 and N_2 such that N_i has G_i as the underlying graph, and the output of v_1 in N_1 equals the output of v_2 in N_2.

▷ *hint V*

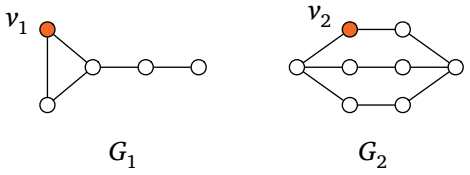


Figure 8.2: Graphs for Exercise 8.7.

8.6 Bibliographic Notes

Local neighborhoods were used to prove negative results in the context of distributed computing by, e.g., Linial [30].

Chapter 9

Round Elimination

In this chapter we introduce the basic idea of a proof technique, called *round elimination*. Round elimination is based on the following idea. Assume that there exists a distributed algorithm S_0 with complexity T solving a problem Π_0. Then there exists a distributed algorithm S_1 with complexity $T-1$ for solving another problem Π_1. That is, if we can solve problem Π_0 in T communication rounds, then we can solve a related problem Π_1 exactly one round faster—we can "eliminate one round". If this operation is repeated T times, we end up with some algorithm S_T with round complexity 0 for some problem Π_T. If Π_T is not a *trivial* problem, that is, cannot be solved in 0 rounds, we have reached a contradiction: therefore the assumption that Π_0 can be solved in T rounds has to be wrong. This is a very useful approach, as it is much easier to reason about 0-round algorithms than about algorithms in general.

9.1 Bipartite Model and Biregular Trees

When dealing with round elimination, we will consider a model that is a variant of the PN model from Chapter 3. We will restrict our attention to specific families of graphs (see Figure 9.1):

(a) **Bipartite.** The set of nodes V is partitioned into two sets: the *active* nodes V_A and the *passive* nodes V_P. The partitioning forms a proper 2-coloring of the graph, i.e., each edge connects an active node with a passive node. The role of a node—active or passive—is part of the local input.

(b) **Biregular trees.** We will assume that the input graphs are *biregular* trees: the graph is connected, there are no cycles, each node in

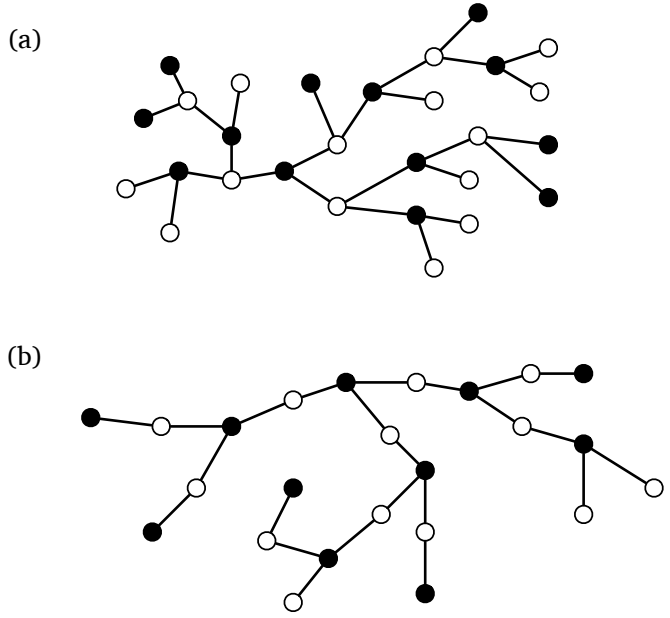


Figure 9.1: The bipartite model; black nodes are active and white nodes are passive. (a) A $(3,3)$-biregular tree. (b) A $(3,2)$-biregular tree.

V_A has degree d or 1, and each node in V_P has degree δ or 1. We say that such a tree is (d,δ)-biregular. See Figure 9.1 for an illustration.

9.1.1 Bipartite Locally Verifiable Problem

We consider a specific family of problems, called *bipartite locally verifiable* problems. Such a problem is defined as a 3-tuple $\Pi = (\Sigma, \mathbf{A}, \mathbf{P})$, where:

- Σ is a finite alphabet.
- $\mathbf{A}$ and $\mathbf{P}$ are finite collections of multisets, where each multiset $A \in \mathbf{A}$ and $P \in \mathbf{P}$ consists of a finite number of elements from Σ. These are called the *active* and *passive configurations*.

Recall that multisets are sets that allow elements to be repeated. We use the notation $[x_1, x_2, \ldots, x_k]$ for a multiset that contains k elements; for example, $[1, 1, 2, 2, 2]$ is a multiset with two 1s and three 2s. Note that the order of elements does not matter, for example, $[1, 1, 2] = [1, 2, 1] = [2, 1, 1]$.

In problem Π, each active node $v \in V_A$ must label its incident $\deg(v)$ edges with elements of Σ such that the labels of the incident edges, considered as a multiset, form an element of $\mathbf{A}$. The order of the labels does not matter. The passive nodes do not have outputs. Instead, we require that for each passive node the labels of its incident edges, again considered as a multiset, form an element of $\mathbf{P}$. A labeling $\varphi : E \to \Sigma$ is a solution to Π if and only if the incident edges of all active and passive nodes are labeled according to some configuration.

In this chapter we will only consider labelings such that all nodes of degree 1 accept any configuration: these will not be explicitly mentioned in what follows. Since we only consider problems in (d, δ)-biregular trees, each active configuration will have d elements and each passive configuration δ elements.

9.1.2 Examples

To illustrate the definition of bipartite locally verifiable labelings, we consider some examples (see Figure 9.2).

Edge Coloring. A c-edge coloring is an assignment of labels from $\{1, 2, \ldots, c\}$ to the edges such that no node has two incident edges with the same label.

Consider the problem of 5-edge coloring $(3, 3)$-biregular trees. The alphabet Σ consists of the five edge colors $\{1, 2, 3, 4, 5\}$. The active configurations consist of all multisets of three elements $[x, y, z]$, such that all elements are distinct and come from Σ. The problem is symmetric, and the passive configurations consist of the same multisets:

$$\mathbf{A} = \mathbf{P} = \big\{ [1,2,3], [1,2,4], [1,2,5], [1,3,4], [1,3,5], \\ [1,4,5], [2,3,4], [2,3,5], [2,4,5], [3,4,5] \big\}.$$

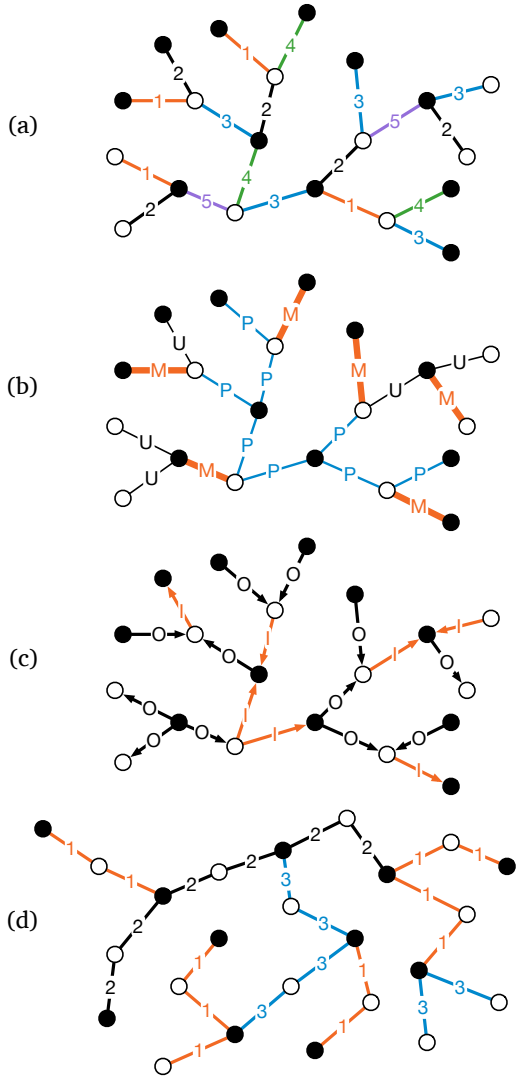


Figure 9.2: Bipartite locally verifiable labeling problems. (a) 5-edge coloring in a $(3,3)$-biregular tree. (b) Maximal matching in a $(3,3)$-biregular tree. (c) Sinkless orientation in a $(3,3)$-biregular tree. (d) Weak 3-labeling in a $(3,2)$-biregular tree.

Maximal Matching. A maximal matching M is a subset of the edges such that no two incident edges are in M and no edge can be added to M.

Consider maximal matching on $(3,3)$-biregular trees. To encode a matching, we could use just two labels: M for matched and U for unmatched. Such a labeling, however, has no way of guaranteeing maximality. We use a third label P, called a pointer:

$$\Sigma = \{\mathsf{M}, \mathsf{P}, \mathsf{U}\}.$$

The active nodes either output $[\mathsf{M}, \mathsf{U}, \mathsf{U}]$, denoting that the edge marked M is in the matching, or they output $[\mathsf{P}, \mathsf{P}, \mathsf{P}]$, denoting that they are unmatched, and thus all passive neighbors *must* be matched with another active node:

$$\mathbf{A} = \big\{ [\mathsf{M}, \mathsf{U}, \mathsf{U}], [\mathsf{P}, \mathsf{P}, \mathsf{P}] \big\}.$$

Passive nodes must verify that they are matched with at most one node, and that if they have an incident label P, then they also have an incident label M (to ensure maximality). Hence the passive configurations are

$$\mathbf{P} = \big\{ [\mathsf{M}, \mathsf{P}, \mathsf{P}], [\mathsf{M}, \mathsf{P}, \mathsf{U}], [\mathsf{M}, \mathsf{U}, \mathsf{U}], [\mathsf{U}, \mathsf{U}, \mathsf{U}] \big\}.$$

Sinkless Orientation. A *sinkless orientation* is an orientation of the edges such that each node has an edge oriented away from it. That is, no node is a *sink*. We will consider here sinkless orientation in (3,3)-biregular trees; leaf nodes can be sinks, but nodes of degree 3 must have at least one outgoing edge.

To encode sinkless orientation, each active node chooses an orientation of its incident edges: outgoing edges are labeled O and incoming edges I. Thus the alphabet is $\Sigma = \{\mathsf{O}, \mathsf{I}\}$. Each node must have an outgoing edge, so the active configurations are all multisets that contain at least one O:

$$\mathbf{A} = \big\{ [\mathsf{O}, x, y] \;\big|\; x, y \in \Sigma \big\}.$$

The passive configurations are similar, but the roles of the labels are reversed: an outgoing edge for an active node is an incoming edge for a

passive node. Therefore each passive node requires that at least one of its incident edges is labeled l, and the passive configurations are

$$\mathbf{P} = \big\{ [\mathsf{l}, x, y] \;\big|\; x, y \in \Sigma \big\}.$$

Weak Labeling. We will use the following problem as the example in the remainder of this chapter. Consider $(3,2)$-biregular trees. A *weak* 3*-labeling* is an assignment of labels from the set $\{1,2,3\}$ to the edges such that each active node has at least two incident edges labeled with different labels. Each passive node must have its incident edges labeled with the same label. The problem can be formalized as

$$\begin{aligned}
&\Sigma = \{1,2,3\},\\
&\mathbf{A} = \big\{ [1,1,2], [1,1,3], [1,2,2], [1,2,3], [1,3,3], [2,2,3], [2,3,3] \big\},\\
&\mathbf{P} = \big\{ [1,1], [2,2], [3,3] \big\}.
\end{aligned}$$

9.2 Introducing Round Elimination

Round elimination is based on the following basic idea. Assume that we can solve some bipartite locally verifiable problem Π_0 in T communication rounds on (d,δ)-biregular trees. Then there exists a bipartite locally verifiable problem Π_1, called the *output problem* of Π_0, that can be solved in $T-1$ rounds on (δ,d)-biregular trees. The output problem is uniquely defined, and we refer to the output problem of Π as $\mathrm{re}(\Pi)$. The definition of output problem will be given in Section 9.2.2.

A single round elimination step is formalized in the following lemma.

Lemma 9.1 (Round elimination lemma)**.** *Let* Π *be bipartite locally verifiable problem that can be solved in* T *rounds in* (d,δ)*-biregular trees. Then the output problem* $\mathrm{re}(\Pi)$ *of* Π *can be solved in* $T-1$ *rounds in* (δ,d)*-biregular trees.*

9.2.1 Impossibility Using Iterated Round Elimination

Lemma 9.1 can be iterated, applying it to the output problem of the previous step. This will yield a sequence of $T+1$ problems

$$\Pi_0 \to \Pi_1 \to \cdots \to \Pi_T,$$

where $\Pi_{i+1} = \mathrm{re}(\Pi_i)$ for each $i = 0, 1, \ldots, T-1$.

If we assume that there is a T-round algorithm for Π_0, then by an iterated application of Lemma 9.1, there is a $(T-1)$-round algorithm for Π_1, a $(T-2)$-round algorithm for Π_2, and so on. In particular, there is a 0-round algorithm for Π_T.

Algorithms that run in 0 rounds are much easier to reason about than algorithms in general. Since there is no communication, each active node must simply map its input, essentially its degree, to some output. In particular, we can try to show that there is no 0-round algorithm for Π_T. If this is the case, we have a contradiction with our original assumption: there is no T-round algorithm for Π_0.

We will now proceed to formally define output problems.

9.2.2 Output Problems

For each locally verifiable problem Π we will define a unique *output problem* $\mathrm{re}(\Pi)$.

Let $\Pi_0 = (\Sigma_0, \mathbf{A}_0, \mathbf{P}_0)$ be a bipartite locally verifiable problem on (d, δ)-biregular trees. We define the output problem $\Pi_1 = \mathrm{re}(\Pi_0) = (\Sigma_1, \mathbf{A}_1, \mathbf{P}_1)$ of Π_0 on (δ, d)-biregular trees as follows—note that we swapped the degrees of active vs. passive nodes here.

The alphabet Σ_1 consists of all possible non-empty subsets of Σ_0. The roles of the active and passive nodes are inverted, and new configurations are computed as follows.

(a) The active configurations $\mathbf{A}_1$ consist of all multisets

$$[X_1, X_2, \ldots, X_\delta], \text{ where } X_i \in \Sigma_1 \text{ for all } i = 1, \ldots, \delta,$$

such that for ***every*** choice of $x_1 \in X_1$, $x_2 \in X_2$, …, $x_\delta \in X_\delta$ we have $[x_1, x_2, \ldots, x_\delta] \in \mathbf{P}_0$, i.e., it is a passive configuration of Π_0.

(b) The passive configurations $\mathbf{P}_1$ consist of all multisets

$$[Y_1, Y_2, \ldots, Y_d], \text{ where } Y_i \in \Sigma_1 \text{ for all } i = 1, \ldots, d,$$

for which ***there exists*** a choice $y_1 \in Y_1$, $y_2 \in Y_2$, …, $y_d \in Y_d$ with $[y_1, y_2, \ldots, y_d] \in \mathbf{A}_0$, i.e., it is an active configuration of Π_0.

9.2.3 Example: Weak 3-labeling

To illustrate the definition, let us construct the output problem $\mathrm{re}(\Pi_0) = (\Sigma_1, \mathbf{A}_1, \mathbf{P}_1)$ of weak 3-labeling problem $\Pi_0 = (\Sigma_0, \mathbf{A}_0, \mathbf{P}_0)$. Recall that

$$\Sigma_0 = \{1, 2, 3\},$$
$$\mathbf{A}_0 = \big\{[1,1,2], [1,1,3], [1,2,2], [1,2,3], [1,3,3], [2,2,3], [2,3,3]\big\},$$
$$\mathbf{P}_0 = \big\{[1,1], [2,2], [3,3]\big\}.$$

The alphabet Σ_1 consists of all possible (non-empty) subsets of Σ_0:

$$\Sigma_1 = \big\{\{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\big\}.$$

The active configurations $\mathbf{A}_1$ are all multisets $[X, Y]$ with $X, Y \in \Sigma_1$ such that *all* choices of elements $x \in X$ and $y \in Y$ result in a multiset $[x, y] \in \mathbf{P}_0$. For example $X = \{1\}$ and $Y = \{1, 2\}$ is *not* a valid choice: we could choose $x = 1$ and $y = 2$ to construct $[1, 2] \notin \mathbf{P}_0$. In general, whenever $|X| > 1$ or $|Y| > 1$, we can find $x \in X$, $y \in Y$ with $x \neq y$, and then $[x, y] \notin \mathbf{P}_0$. Therefore the only possibilities are $|X| = |Y| = 1$, and then we must also have $X = Y$. We obtain

$$\mathbf{A}_1 = \big\{\big[\{1\}, \{1\}\big], \big[\{2\}, \{2\}\big], \big[\{3\}, \{3\}\big]\big\}.$$

But since the active configurations only allow singleton sets, we can restrict ourselves to them when listing the possible passive configurations; we obtain simply

$$\mathbf{P}_1 = \big\{\big[\{1\}, \{1\}, \{2\}\big], \big[\{1\}, \{1\}, \{3\}\big], \big[\{1\}, \{2\}, \{2\}\big], \big[\{1\}, \{2\}, \{3\}\big],$$
$$\big[\{1\}, \{3\}, \{3\}\big], \big[\{2\}, \{2\}, \{3\}\big], \big[\{2\}, \{3\}, \{3\}\big]\big\}.$$

9.2.4 Complexity of Output Problems

In this section we prove Lemma 9.1 that states that the output problem $\mathrm{re}(\Pi)$ of Π can be solved one round faster than Π.

The proof is by showing that we can truncate the execution of a T-round algorithm and output the set of *possible outputs*. As we will see, this is a solution to the output problem.

Proof of Lemma 9.1. Assume that we can solve some problem $\Pi_0 = (\Sigma_0, \mathbf{A}_0, \mathbf{P}_0)$ on (d, δ)-biregular trees in T rounds using some deterministic PN-algorithm S. We want to design an algorithm that works in (δ, d)-biregular trees and solves $\Pi_1 = \mathrm{re}(\Pi_0)$ in $T - 1$ rounds.

Note that we are considering the same family of networks, but we are only switching the sides that are marked as active and passive. We will call these Π_0-*active* and Π_1-*active* sides, respectively.

The algorithm for solving Π_1 works as follows. Let $N = (V, P, p)$ be any port-numbered network with a (δ, d)-biregular tree as the underlying graph. Each Π_1-active node u, in $T - 1$ rounds, gathers its full $(T-1)$-neighborhood $\mathrm{ball}_N(u, T-1)$. Now it considers all *possible* outputs of its Π_0-active neighbors under the algorithm S, and outputs these.

Formally, this is done as follows. When N is a port-numbered network, we use $\mathrm{ball}_N(u, r)$ to refer to the information within distance r from node u, including the local inputs of the nodes in this region, as well as the port numbers of the edges connecting nodes within this region. We say that a port-numbered network H is *compatible* with $\mathrm{ball}_N(u, r)$ if there is a node $v \in H$ such that $\mathrm{ball}_H(v, r)$ is isomorphic to $\mathrm{ball}_N(u, r)$.

For each neighbor v of u, node u constructs all possible fragments $\mathrm{ball}_H(v, T)$ such that H is compatible with $\mathrm{ball}_N(u, T-1)$ and has a (δ, d)-biregular tree as its underlying graph. Then u simulates the Π_0-algorithm S on $\mathrm{ball}_H(v, T)$. The algorithm outputs some label $x \in \Sigma_0$ on the edge $\{u, v\}$. Node u adds each such label x to set $S(u, v)$; finally node u will label edge $\{u, v\}$ with $S(u, v)$.

By construction, $S(u, v)$ is a nonempty set of labels from Σ_0, i.e., $S(u, v) \in \Sigma_1$. We now prove that the sets $S(u, v)$ form a solution to Π_1. We use the assumption that the underlying graph G is a tree. Let H be

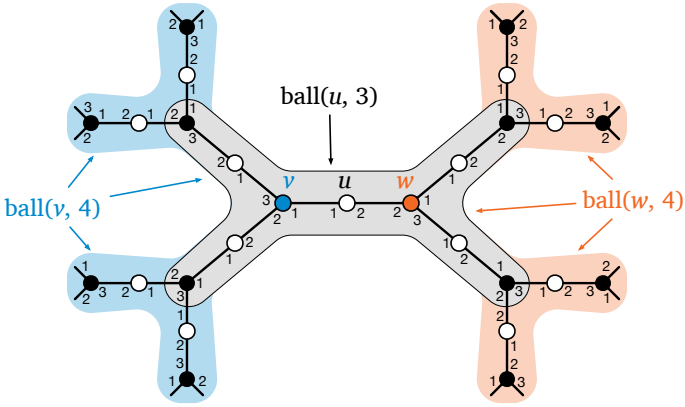


Figure 9.3: Illustration of the round elimination step. A fragment of a $(2,3)$-biregular tree. The 3-neighborhood of node u consists of the gray area. The 4-neighborhoods of nodes v and w consist of the blue and orange areas, respectively. Since the input is a tree, these intersect exactly in the 3-neighborhood of u.

any port-numbered network compatible with $\mathrm{ball}_N(u, T-1)$. Consider any two neighbors v and w of u: since there are no cycles, we have

$$\mathrm{ball}_H(v, T) \cap \mathrm{ball}_H(w, T) = \mathrm{ball}_H(u, T-1) = \mathrm{ball}_N(u, T-1).$$

In particular, once $\mathrm{ball}_H(u, T-1)$ is fixed, the outputs of v and w, respectively, depend on the structures of $\mathrm{ball}_H(v, T) \setminus \mathrm{ball}_H(u, T-1)$ and $\mathrm{ball}_H(w, T) \setminus \mathrm{ball}_H(u, T-1)$, which are completely distinct. See Figure 9.3 for an illustration. Therefore, if there exist $x \in S(u, v)$ and $y \in S(u, w)$, then there exists a port-numbered network H such that running S, node v outputs x on $\{v, u\}$ and node w outputs y on $\{w, u\}$. This further implies that since S is assumed to work correctly on all port-numbered networks, for any combination of $x_1 \in S(u, v_1)$, $x_2 \in S(u, v_2)$, $\ldots$, $x_\delta \in S(u, v_\delta)$, we must have that

$$[x_1, x_2, \ldots, x_\delta] \in \mathbf{P}_0.$$

This implies that

$$[S(u, v_1), S(u, v_2), \ldots, S(u, v_\delta)] \in \mathbf{A}_1.$$

It remains to show that for each Π_0-active node v, it holds that the sets $S(u_1, v)$, $S(u_2, v)$, $\ldots$, $S(u_d, v)$, where u_i are neighbors of v, form a configuration in $\mathbf{P}_1$. To see this, note that the Π_1-active nodes u_i simulate S on every port-numbered fragment, including the true neighborhood $\text{ball}_N(v, T)$ of v. This implies that the output of v on $\{v, u_i\}$ running S in network N is included in $S(u_i, v)$. Since S is assumed to be a correct algorithm, these true outputs $x_1 \in S(u_1, v)$, $x_2 \in S(u_2, v)$, $\ldots$, $x_d \in S(u_d, v)$ form a configuration

$$[x_1, x_2, \ldots, x_d] \in \mathbf{A}_0,$$

which implies that

$$[S(u_1, v), S(u_2, v), \ldots, S(u_d, v)] \in \mathbf{P}_1,$$

as required. □

9.2.5 Example: Complexity of Weak 3-labeling

Now we will apply the round elimination technique to show that the weak 3-labeling problem is not solvable in 1 round. To do this, we show that the output problem of weak 3-labeling is not solvable in 0 rounds.

Lemma 9.2. *Weak 3-labeling is not solvable in 1 round in the* PN*-model on* $(3, 2)$*-biregular trees.*

Proof. In Section 9.2.3 we saw the output problem of weak 3-labeling. We will now show that this problem is not solvable in 0 rounds on $(2, 3)$-biregular trees. By Lemma 9.1, weak 3-labeling is then not solvable in 1 round on $(3, 2)$-biregular trees. Let $\Pi_1 = (\Sigma_1, \mathbf{A}_1, \mathbf{P}_1)$ denote the output problem of weak 3-labeling.

In a 0-round algorithm an active node v sees only its own side (active or passive) and its own port numbers. Since

$$\mathbf{A}_1 = \big\{ [\{1\},\{1\}], [\{2\},\{2\}], [\{3\},\{3\}] \big\},$$

each active node v must output the same label $X \in \big\{\{1\},\{2\},\{3\}\big\}$ on both of its incident edges.

Since all active nodes look the same, they all label their incident edges with exactly one label X. Since $[X,X,X]$ is not in $\mathbf{P}_1$ for any $X \in \Sigma_1$, we have proven the claim. □

9.2.6 Example: Iterated Round Elimination

We will finish this chapter by applying round elimination *twice* to weak 3-labeling. We will see that the problem

$$\Pi_2 = \mathrm{re}(\Pi_1) = \mathrm{re}(\mathrm{re}(\Pi_0))$$

obtained this way *is* 0-round solvable.

Let us first construct Π_2. Note that this is again a problem on $(3,2)$-biregular trees. We first *simplify* notation slightly; the labels of Π_1 are sets and labels of Π_2 would be sets of sets, which gets awkward to write down. But the configurations in Π_1 only used *singleton* sets. Therefore we can *leave out* all non-singleton sets without changing the problem, and then we can *rename* each singleton set $\{x\}$ to x. After these simplifications, we have

$$\begin{aligned}
&\Sigma_1 = \{1,2,3\},\\
&\mathbf{A}_1 = \big\{ [1,1], [2,2], [3,3] \big\},\\
&\mathbf{P}_1 = \big\{ [1,1,2], [1,1,3], [1,2,2], [1,2,3], [1,3,3], [2,2,3], [2,3,3] \big\}.
\end{aligned}$$

Alphabet Σ_2 consists of all non-empty subsets of Σ_1, that is

$$\Sigma_2 = \big\{\{1\},\{2\},\{3\},\{1,2\},\{1,3\},\{2,3\},\{1,2,3\}\big\}.$$

The active configurations are all multisets $[X_1, X_2, X_3]$ where $X_1, X_2, X_3 \in \Sigma_2$ such that *any* way of choosing $x_1 \in X_1$, $x_2 \in X_2$, and $x_3 \in X_3$ is a configuration in $\mathbf{P}_1$. There are many cases to check, the following observation will help here (the proof is left as Exercise 9.4):

- $\mathbf{P}_1$ consists of all 3-element multisets over Σ_1 where at least one of the elements is not 1, at least one of the elements is not 2, and at least one of the elements is not 3.
- It follows that $\mathbf{A}_2$ consists of all 3-element multisets over Σ_2 where at least one of the elements does not contain 1, at least one of the elements does not contain 2, and at least one of the elements does not contain 3.

It follows that we can enumerate all possible configurations e.g. as follows (here $X, Y, Z \in \Sigma_2$):

$$\begin{aligned}\mathbf{A}_2 = &\left\{ [X,Y,Z] \,\middle|\, X \subseteq \{1,2\}, Y \subseteq \{1,3\}, Z \subseteq \{2,3\} \right\} \\ &\cup \left\{ [X,Y,Z] \,\middle|\, X \subseteq \{1\}, Y \subseteq \{2,3\}, Z \subseteq \{1,2,3\} \right\} \\ &\cup \left\{ [X,Y,Z] \,\middle|\, X \subseteq \{2\}, Y \subseteq \{1,3\}, Z \subseteq \{1,2,3\} \right\} \\ &\cup \left\{ [X,Y,Z] \,\middle|\, X \subseteq \{3\}, Y \subseteq \{1,2\}, Z \subseteq \{1,2,3\} \right\}.\end{aligned} \tag{9.1}$$

On the passive side, $\mathbf{P}_2$ consists of all multisets $[X, Y]$ where we can choose $x \in X$ and $y \in Y$ with $[x, y] \in \mathbf{A}_1$. But $[x, y] \in \mathbf{A}_1$ is equivalent to $x = y$, and hence $\mathbf{P}_2$ consists of all multisets $[X, Y]$ where we can choose some $x \in X$ and choose the same value $x \in Y$. Put otherwise,

$$\mathbf{P}_2 = \left\{ [X,Y] \,\middle|\, X \in \Sigma_2, Y \in \Sigma_2, X \cap Y \neq \varnothing \right\}.$$

Lemma 9.3. *Let Π_0 denote the weak 3-labeling problem. The problem $\Pi_2 = \mathrm{re}(\mathrm{re}(\Pi_0)) = (\Sigma_2, \mathbf{A}_2, \mathbf{P}_2)$ is solvable in 0 rounds.*

Proof. The active nodes always choose the configuration

$$[\{1,2\}, \{1,3\}, \{2,3\}] \in \mathbf{A}_2$$

and assign the sets in some way using the port numbers, e.g., the edge incident to port 1 is labeled with $\{2,3\}$, the edge incident to port 2 is labeled with $\{1,3\}$, and the edge incident to port 3 is labeled with $\{1,2\}$.

Since each pair of these sets has a non-empty intersection, no matter which sets are assigned to the incident edges of passive nodes, these form a valid passive configuration in $\mathbf{P}_2$. □

9.3 Quiz

Consider the following bipartite locally verifiable labeling problem $\Pi = (\Sigma, \mathbf{A}, \mathbf{P})$ on $(2,2)$-biregular trees:

$$\begin{aligned}
\Sigma &= \{1,2,3,4,5,6\},\\
\mathbf{A} &= \big\{[1,6],[2,5],[3,4]\big\}, \text{ and}\\
\mathbf{P} &= \big\{[x,y] \bigm| x \in \{3,5,6\}, y \in \{1,2,3,4,5,6\}\big\}\\
&\quad \cup \big\{[x,y] \bigm| x \in \{4,5,6\}, y \in \{2,3,4,5,6\}\big\}.
\end{aligned}$$

Give a 0-round algorithm for solving Π.

9.4 Exercises

Exercise 9.1 (encoding graph problems)**.** Even if a graph problem is defined for general (not bipartite) graphs, we can often represent it in the bipartite formalism. If we take a d-regular tree G and subdivide each edge, we arrive at a $(d,2)$-biregular tree H, where the active nodes represent the nodes of G and passive nodes represent the edges of G.

Use this idea to encode the following graph problems as bipartite locally verifiable labelings in $(d,2)$-biregular trees. Give a brief explanation of why your encoding is equivalent to the original problem. You can ignore the leaf nodes and their constraints; it is sufficient to specify constraints for the active nodes of degree d and passive nodes of degree 2.

(a) Vertex coloring with $d+1$ colors.

(b) Maximal matching.
(c) Dominating set.
(d) Minimal dominating set.

Exercise 9.2 (algorithms in the bipartite model)**.** The bipartite model can be used to run algorithms from the standard PN and LOCAL models. Using the idea of Exercise 9.1, we encode the *maximal independent set* problem in 3-regular trees as the following bipartite locally verifiable problem $\Pi = (\Sigma, \mathbf{A}, \mathbf{P})$ in $(3,2)$-biregular trees:

$$\begin{aligned}\Sigma &= \{\mathsf{I}, \mathsf{O}, \mathsf{P}\},\\ \mathbf{A} &= \big\{[\mathsf{I},\mathsf{I},\mathsf{I}], [\mathsf{P},\mathsf{O},\mathsf{O}]\big\},\\ \mathbf{P} &= \big\{[\mathsf{I},\mathsf{P}], [\mathsf{I},\mathsf{O}], [\mathsf{O},\mathsf{O}]\big\}.\end{aligned}$$

In $\mathbf{A}$, the first configuration corresponds to a node in the independent set X, and the second configuration to a node not in X. A node not in X points to a neighboring active node with the label P: the node pointed to has to be in X. The passive configurations ensure that two active nodes connected by a passive node are not both in X, and that the pointer P always points to a node in X.

Assume that the active nodes are given a 4-coloring c as input. That is, $c\colon V_A \to \{1,2,3,4\}$ satisfies $c(v) \neq c(u)$ whenever the active nodes $v, u \in V_A$ share a passive neighbor $w \in V_P$. The nodes also know whether they are active or passive, but the nodes do not have any other information.

Present a PN-algorithm in the state machine formalism for solving Π. Prove that your algorithm is correct. What is its running time? How does it compare to the complexity of solving maximal independent set in the PN model, given a 4-coloring?

Exercise 9.3 (Round Eliminator)**.** There is a computer program, called Round Eliminator, that implements the round elimination technique and that you can try out in a web browser:

https://github.com/olidennis/round-eliminator

Let Π_0 be the weak 3-labeling problem defined in Section 9.1.2. Use the Round Eliminator to find out what are $\Pi_1 = \mathrm{re}(\Pi_0)$ and $\Pi_2 = \mathrm{re}(\Pi_1)$. In your answer you need to show how to encode Π_0 in a format that is suitable for the Round Eliminator, what were the answers you got from the Round Eliminator, and how to turn the answers back into our usual mathematical formalism.

Exercise 9.4 (iterated round elimination)**.** Fill in the missing details in Section 9.2.6 to show that formula (9.1) is a correct definition of the active configurations for problem Π_2 (i.e., it contains all possible configurations and only them).

Exercise 9.5 (solving weak 3-labeling)**.** Present a 2-round deterministic PN-algorithm for solving weak 3-labeling in $(3,2)$-biregular trees.

Exercise 9.6 (sinkless orientation)**.** Consider the sinkless orientation problem, denoted by Π, on $(3,3)$-biregular trees from Section 9.1.2. Compute the output problems $\mathrm{re}(\Pi)$ and $\mathrm{re}(\mathrm{re}(\Pi))$; include a justification for your results.

9.5 Bibliographic Notes

Linial's [30] lower bound for vertex coloring in cycles already used a proof technique that is similar to round elimination. However, for a long time it was thought that this is an ad-hoc technique that is only applicable to this specific problem. This started to change in 2016, when the same idea found another very different application [11]. Round elimination as a general-purpose technique was defined and formalized by Brandt [10] in 2019, and implemented as a computer program by Olivetti [34].

Chapter 10

Sinkless Orientation

In this chapter we will study the complexity of *sinkless orientation*, a problem that was introduced in the previous chapter. This is a problem that is understood well: we will design algorithms and show that these are asymptotically optimal.

Recall that sinkless orientation is the problem of orienting the edges of the tree so that each internal node has at least one outgoing edge. We begin by studying sinkless orientation on paths (or $(2,2)$-biregular trees), and show that we can easily argue about local neighborhoods to prove a tight lower bound result. However, when we switch to $(3,3)$-biregular trees, we will need the round elimination technique to do the same.

10.1 Sinkless Orientation on Paths

We define *sinkless orientation on paths* to be the following bipartite locally verifiable problem $\Pi = (\Sigma, \mathbf{A}, \mathbf{P})$. The alphabet is $\Sigma = \{\mathsf{I}, \mathsf{O}\}$, with the interpretation that I indicates that the edge is oriented towards the active node ("incoming") and O indicates that the edge is oriented away from the active node ("outgoing"). Each active node must label at least one incident edge with O, and thus the active configurations are

$$\mathbf{A} = \big\{ [\mathsf{O}, \mathsf{I}], [\mathsf{O}, \mathsf{O}] \big\}.$$

Each passive node must have at least one incident edge labeled with I, and thus the passive configurations are

$$\mathbf{P} = \big\{ [\mathsf{I}, \mathsf{O}], [\mathsf{I}, \mathsf{I}] \big\}.$$

As previously, nodes of degree 1 are unconstrained; the edges incident to them can be labeled arbitrarily.

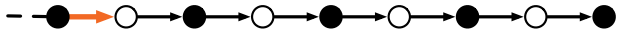


Figure 10.1: Propagation of a sinkless orientation on paths. Orienting a single edge (orange) forces the orientation of the path all the way to the other endpoint.

10.1.1 Hardness of Sinkless Orientation

We begin by showing that solving sinkless orientation requires $\Omega(n)$ rounds on $(2,2)$-biregular trees.

Lemma 10.1. *Solving sinkless orientation on $(2,2)$-biregular trees in the bipartite* PN*-model requires at least $n/4-1$ rounds, even if the nodes know the value n.*

Let us first see why the lemma is intuitively true. Consider a path, as illustrated in Figure 10.1. Each active node u must choose some label for its incident edges, and at least one of these labels must be O. Then its passive neighbor v over the edge labeled with O must have its other incident edge labeled I. This further implies that the other active neighbor w of v must label its other edge with O. The original output of u *propagates* through the path and the outputs of other nodes far away from u depend on the output of u.

Let us now formalize this intuition.

Proof of Lemma 10.1. Consider any algorithm A running in $T(n) = o(n)$ rounds. Then there exists n_0 such that for all $n \geq n_0$, we have that $T(n) \leq (n-5)/4$. Now fix such an integer n and let $T = T(n)$ denote the running time of the algorithm.

Consider an active node v in the middle of a path N on n nodes. Let $\mathrm{ball}_N(v,T)$ denote the T-neighborhood of v. Assume that $\mathrm{ball}_N(v,T)$ is consistently port-numbered from *left* to *right*, as illustrated in Figure 10.2. Node v must use the output label O on one of its incident edges; without loss of generality, assume that this is port 1. We can now construct a counterexample N' as follows. Take two copies of $\mathrm{ball}_N(v,T)$, denoted by $\mathrm{ball}_{N'}(v_1,T)$ and $\mathrm{ball}_{N'}(v_2,T)$. In particular, this includes the port-numbering in $\mathrm{ball}_N(v,T)$. Add one new node that is connected to the

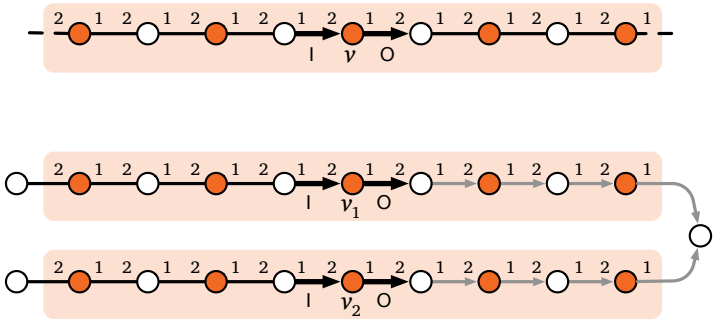


Figure 10.2: Sinkless orientation lower bound. Assume $T = 4$. Top: any algorithm must fix some output labeling with an outgoing edge for a fixed neighborhood $\text{ball}_N(v, 4)$. Bottom: Copying the same 4-neighborhood twice, and arranging the copies towards the same node creates an input N' where the two nodes orient towards the middle. There is no legal way to label the rest of the path.

right endpoint of both $\text{ball}_{N'}(v_1, T)$ and $\text{ball}_{N'}(v_2, T)$. Finally, add leaves at the other endpoints of $\text{ball}_{N'}(v_1, T)$ and $\text{ball}_{N'}(v_2, T)$; see Figure 10.2 for an illustration. We claim that the algorithm A fails on N'.

By definition, edges must be labeled alternatively O and I starting from both v_1 and v_2. Therefore we must eventually find an active node labeled $[\mathsf{I}, \mathsf{I}]$ or a passive node labeled $[\mathsf{O}, \mathsf{O}]$, an incorrect solution.

The total number of nodes in N' is $n = 2(2T + 1) + 3 = 4T + 5$, giving $T = (n-5)/4$. Thus solving sinkless orientation requires at least $T + 1 = (n-5)/4 + 1 \geq n/4 - 1$ rounds, as required. □

10.1.2 Solving Sinkless Orientation on Paths

The proof of Lemma 10.1 shows that it is impossible to solve sinkless orientation on paths in sublinear number of rounds. Now we will show a linear upper bound: it is possible to solve sinkless orientation once all nodes see an endpoint of the path.

Lemma 10.2. *Sinkless orientation can be solved in $\lfloor n/2 \rfloor$ rounds in the bipartite* PN*-model on $(2, 2)$-biregular trees.*

Proof. The idea of the algorithm is the following: initially send messages from the leaves towards the middle of the path. Orient edges against the incoming messages, i.e., toward the closest leaf. Once the messages reach the midpoint of the path, all edges have been correctly oriented away from the midpoint.

The algorithm works as follows. Initially all non-leaf nodes wait. The leaf nodes send a message to their neighbor and stop. If they are active, they output I on their incident edge. Whenever a node receives a message for the first time, in the next round it sends a message to the other port and stops. If it is an active node, it outputs O in the port from which it received the message, and I in the other port. That is, it orients its incident edges towards the closer leaf. If a node receives two messages in the same round, it is the midpoint of the path; it does not send any further messages. If it is an active node, it outputs O on both of its incident edges.

The algorithm runs in $\lfloor n/2 \rfloor$ rounds: on paths with an even number of nodes, all nodes have received a message in round $n/2-1$, and thus stop in the next round. On paths with an odd number of nodes, the middle node receives two messages in round $\lfloor n/2 \rfloor$ and stops.

It remains to show that our algorithm is correct. All leaf nodes are trivially labeled correctly. Any active non-leaf node always has an incident label O. Now consider a passive node u: there is an active v that sends u a message before stopping. This node will output I on $\{u, v\}$, and thus u is also correctly labeled. □

Theorem 10.3. *The complexity of sinkless orientation on paths is* $\Theta(n)$.

Proof. Follows from Lemmas 10.1 and 10.2. □

10.2 Sinkless Orientation on Trees

In Section 10.1 we saw that if we require that degree-2 nodes have at least one outgoing edge, we arrive at a problem that is hard already in the case of paths. The proof of hardness was a straightforward argument that used local neighborhoods.

However, what happens if we relax the problem slightly and allow any orientation around degree-2 nodes? The proof of hardness from Section 10.1.1 no longer works, but does the problem get easier to solve?

For concreteness, let us consider *trees of maximum degree* 3, that is, both active and passive nodes have degree at most 3; the case of higher degrees is very similar. We define the problem so that nodes of degree 1 and 2 are unconstrained, but nodes of degree 3 must have at least one outgoing edge. We can encode it as a bipartite locally verifiable problem $\Pi_0 = (\Sigma_0, \mathbf{A}_0, \mathbf{P}_0)$ as follows:

$$\Sigma_0 = \{\mathsf{O}, \mathsf{I}\},$$
$$\mathbf{A}_0 = \big\{[\mathsf{O}], [\mathsf{I}], [\mathsf{O},\mathsf{O}], [\mathsf{O},\mathsf{I}], [\mathsf{I},\mathsf{I}], [\mathsf{O},\mathsf{I},\mathsf{I}], [\mathsf{O},\mathsf{O},\mathsf{I}], [\mathsf{O},\mathsf{O},\mathsf{O}]\big\},$$
$$\mathbf{P}_0 = \big\{[\mathsf{O}], [\mathsf{I}], [\mathsf{O},\mathsf{O}], [\mathsf{O},\mathsf{I}], [\mathsf{I},\mathsf{I}], [\mathsf{I},\mathsf{O},\mathsf{O}], [\mathsf{I},\mathsf{I},\mathsf{O}], [\mathsf{I},\mathsf{I},\mathsf{I}]\big\}.$$

Here we have listed all possible configurations for nodes of degrees 1, 2, and 3.

10.2.1 Solving Sinkless Orientation on Trees

The algorithm for solving sinkless orientation on trees uses ideas similar to the algorithm on paths: each node u must determine the closest unconstrained node v, i.e., a node of degree 1 or 2, and the path from u to v is oriented away from u. This will make all nodes happy: each active node of degree 3 has an outgoing edge, and all other nodes are unconstrained.

Let us call nodes of degree 1 and 2 *special nodes*. We must be careful in how the nodes choose the special node: the algorithm would fail if two nodes want to orient the same edge in different directions.

The algorithm functions as follows. In the first round, only special nodes are sending messages, broadcasting to each port. Then the special nodes stop and, if they are active nodes, they output I on each edge. Nodes of degree 3 *wake up* in the first round in which they receive at least one message. In the next round they broadcast to each port from which they did not receive a message in the previous round. After sending

this message, the nodes stop. If they are active nodes, they orient their incident edges towards the smallest port from which they received a message: output O on that edge, and I on the other edges.

Correctness. Assume that the closest special nodes are at distance t from some node u. Assume that v is one of those nodes, and let $(v_1, v_2, \ldots, v_{t+1})$ denote the unique path from $v = v_1$ to $u = v_{t+1}$. We claim that in each round i, node v_i broadcasts to v_{i+1}. By assumption, v is also one of the closest special nodes to all v_i; otherwise there would be a closer special node to u as well. In particular, there will never be a broadcast from v_{i+1} to v_i, as then v_{i+1} would have a different closer special node. Therefore each v_i will broadcast to v_{i+1} in round i. This implies that in round t, node u will receive a broadcast from v_t.

All nodes that receive a broadcast become happy: Active nodes output O on one of the edges from which they received a broadcast, making them happy. They output I on the other edges, so each passive node is guaranteed that every edge from which it receives a broadcast has the label I.

Time Complexity. It remains to bound the round by which all nodes have received a broadcast. To do this, we observe that each node is at distance $O(\log n)$ from a special node.

Consider a fragment of a 3-regular tree centered around some node v, and assume that there are no special nodes near v. Then at distance 1 from v there are 3 nodes, at distance 2 there are 6 nodes, at distance 3 there are 12 nodes, and so on. In general, if we do not have any special nodes within distance i, then at distance i there are $3 \cdot 2^{i-1} > 2^i$ nodes in the tree. At distance $i = \log_2 n$, we would have more than n nodes. Thus, within distance $\log_2 n$, there has to be a special node. Since each node can stop once it has received a broadcast, the running time of the algorithm is $O(\log n)$.

10.2.2 Roadmap: Next Steps

We have seen that sinkless orientation in trees can be solved in $O(\log n)$ rounds. We would like to now prove a matching lower bound and show that sinkless orientation cannot be solved in $o(\log n)$ rounds. We will apply the round elimination technique from Chapter 9 to do this. However, we will need one further refinement to the round elimination technique that will make our life a lot easier: we can *ignore all non-maximal configurations*. We will explain this idea in detail in Section 10.3, and then we are ready to prove the hardness result in Section 10.4.

10.3 Maximal Output Problems

In Chapter 9 we saw how to use the round elimination technique to construct the *output problem* $\Pi' = \mathrm{re}(\Pi)$ for any given bipartite locally verifiable problem Π. We will now make an observation that allows us to *simplify* the description of output problems. We will change the definition of output problems to include this simplification.

Consider an output problem $\Pi' = (\Sigma, \mathbf{A}, \mathbf{P})$. Recall that Σ now consists of *sets* of labels. Assume that there are two configurations

$$X = [X_1, X_2, \ldots, X_d],$$
$$Y = [Y_1, Y_2, \ldots, Y_d],$$

in $\mathbf{A}$. We say that Y *contains* X if we have $X_i \subseteq Y_i$ for all i.

If Y contains X, then configuration X is redundant; whenever an algorithm solving Π' would like to use the configuration X, it can equally well use Y instead:

- Active nodes are still happy if active nodes switch from X to Y: By assumption, Y is also a configuration in $\mathbf{A}$.
- Passive nodes are still happy if active nodes switch from X to Y: Assume that $Z = [Z_1, Z_2, \ldots, Z_\delta]$ is a passive configuration in $\mathbf{P}$. As this is a passive configuration of $\mathrm{re}(\Pi)$, it means that there is a

choice $z_i \in Z_i$ such that $[z_1, z_2, \ldots, z_\delta]$ is an active configuration in the original problem Π. But now if we replace each Z_i with a superset $Z'_i \supseteq Z_i$, then we can still make the same choice $z_i \in Z'_i$, and hence $Z' = [Z'_1, Z'_2, \ldots, Z'_\delta]$ also has to be a passive configuration in $\mathbf{P}$. Therefore replacing a label with its superset is always fine from the perspective of passive nodes, and in particular switching from X to Y is fine.

Therefore we can *omit* all active configurations that are contained in another active configuration and only include the *maximal* configurations, i.e., configurations that are not contained in any other configuration.

We get the following mechanical process for constructing the output problem $\mathrm{re}(\Pi) = (\Sigma, \mathbf{A}, \mathbf{P})$.

(a) Construct the output problem $\mathrm{re}(\Pi) = (\Sigma, \mathbf{A}, \mathbf{P})$ as described in Section 9.2.2.
(b) Remove all non-maximal active configurations from $\mathbf{A}$.
(c) Remove all unused elements from Σ.
(d) Remove all passive configurations containing labels not in Σ.

The resulting problem is exactly as hard to solve as the original problem:

- Since the simplified sets of configurations are subsets of the original sets of configurations, any solution to the simplified problem is a solution to the original problem, and thus the original problem is *at most as hard as* the simplified problem.

- By construction, any algorithm solving the original output problem can solve the simplified problem equally fast, by replacing some labels by their supersets as appropriate. Therefore the original problem is *at least as hard as* the simplified problem.

We will apply this simplification when we use the round elimination technique to analyze the sinkless orientation problem.

10.4 Hardness of Sinkless Orientation on Trees

We will now show that sinkless orientation requires $\Omega(\log n)$ rounds on $(3,3)$-biregular trees—and therefore also in trees of maximum degree 3, as $(3,3)$-biregular trees are a special case of such trees.

Let us first write down the sinkless orientation problem as a bipartite locally verifiable problem $\Pi_0 = (\Sigma_0, \mathbf{A}_0, \mathbf{P}_0)$ in $(3,3)$-biregular trees; as before, we will only keep track of the configurations for nodes of degree 3, as leaf nodes are unconstrained:

$$\begin{aligned}
\Sigma_0 &= \{\mathsf{O}, \mathsf{I}\},\\
\mathbf{A}_0 &= \left\{ [\mathsf{O}, x, y] \;\middle|\; x, y \in \Sigma \right\},\\
\mathbf{P}_0 &= \left\{ [\mathsf{I}, x, y] \;\middle|\; x, y \in \Sigma \right\}.
\end{aligned}$$

10.4.1 First Step

Lemma 10.4. *Let Π_0 be the sinkless orientation problem. Then the output problem is $\Pi_1 = \mathrm{re}(\Pi_0) = (\Sigma_1, \mathbf{A}_1, \mathbf{P}_1)$, where*

$$\begin{aligned}
\Sigma_1 &= \left\{ \{\mathsf{I}\}, \{\mathsf{O}, \mathsf{I}\} \right\},\\
\mathbf{A}_1 &= \left\{ \left[\{\mathsf{I}\}, \{\mathsf{O}, \mathsf{I}\}, \{\mathsf{O}, \mathsf{I}\} \right] \right\},\\
\mathbf{P}_1 &= \left\{ \left[\{\mathsf{O}, \mathsf{I}\}, x, y \right] \;\middle|\; x, y \in \Sigma_1 \right\}.
\end{aligned}$$

Proof. Let us follow the procedure from Section 10.3. First, we arrive at alphabet Σ_1 that contains all non-empty subsets of Σ_0:

$$\Sigma_1 = \left\{ \{\mathsf{O}\}, \{\mathsf{I}\}, \{\mathsf{O}, \mathsf{I}\} \right\}.$$

The active configurations $\mathbf{A}_1$ consist of all multisets $[X, Y, Z]$ such that no matter how we choose $x \in X$, $y \in Y$, and $z \in Z$, at least one element of the multiset $[x, y, z]$ is I. This happens exactly when at least one of

the labels X, Y, and Z is the singleton set $\{\mathsf{I}\}$. We get that

$$\begin{aligned}\mathbf{A}_1 &= \Big\{ \big[\{\mathsf{I}\}, X, Y\big] \,\Big|\, X, Y \subseteq \{\mathsf{O},\mathsf{I}\} \Big\} \\ &= \Big\{ \big[\{\mathsf{I}\}, \{\mathsf{I}\}, \{\mathsf{I}\}\big], \\ &\qquad \big[\{\mathsf{I}\}, \{\mathsf{I}\}, \{\mathsf{O}\}\big], \\ &\qquad \big[\{\mathsf{I}\}, \{\mathsf{I}\}, \{\mathsf{O},\mathsf{I}\}\big], \\ &\qquad \big[\{\mathsf{I}\}, \{\mathsf{O}\}, \{\mathsf{O}\}\big], \\ &\qquad \big[\{\mathsf{I}\}, \{\mathsf{O}\}, \{\mathsf{O},\mathsf{I}\}\big], \\ &\qquad \big[\{\mathsf{I}\}, \{\mathsf{O},\mathsf{I}\}, \{\mathsf{O},\mathsf{I}\}\big] \Big\}.\end{aligned}$$

Next we remove all non-maximal configurations. We note that all other active configurations are contained in the configuration

$$\big[\{\mathsf{I}\}, \{\mathsf{O},\mathsf{I}\}, \{\mathsf{O},\mathsf{I}\}\big].$$

This becomes the only active configuration:

$$\mathbf{A}_1 = \Big\{ \big[\{\mathsf{I}\}, \{\mathsf{O},\mathsf{I}\}, \{\mathsf{O},\mathsf{I}\}\big] \Big\}.$$

Since the label $\{\mathsf{O}\}$ is never used, we may remove it from the alphabet, too: we get that

$$\Sigma_1 = \big\{ \{\mathsf{I}\}, \{\mathsf{O},\mathsf{I}\} \big\}.$$

The passive configurations are all multisets such that at least one label contains O. Thus the simplified passive configurations are

$$\begin{aligned}\mathbf{P}_1 = \Big\{ &\big[\{\mathsf{O},\mathsf{I}\}, \{\mathsf{O},\mathsf{I}\}, \{\mathsf{O},\mathsf{I}\}\big], \\ &\big[\{\mathsf{O},\mathsf{I}\}, \{\mathsf{O},\mathsf{I}\}, \{\mathsf{I}\}\big], \\ &\big[\{\mathsf{O},\mathsf{I}\}, \{\mathsf{I}\}, \{\mathsf{I}\}\big] \Big\}.\end{aligned}$$

□

10.4.2 Equivalent Formulation

Now let us simplify the notation slightly. We say that a problem Π' is *equivalent* to another problem Π if a solution of Π' can be converted in

zero rounds to a solution of Π and vice versa. In particular, equivalent problems are exactly as hard to solve.

Lemma 10.5. *Let Π_0 be the sinkless orientation problem. Then the output problem* $\mathrm{re}(\Pi_0)$ *is equivalent to* $\Pi_1 = (\Sigma_1, \mathbf{A}_1, \mathbf{P}_1)$*, where*

$$\begin{aligned}\Sigma_1 &= \{\mathsf{A}, \mathsf{B}\},\\ \mathbf{A}_1 &= \big\{[\mathsf{A}, \mathsf{B}, \mathsf{B}]\big\},\\ \mathbf{P}_1 &= \big\{[\mathsf{B}, x, y] \;\big|\; x, y \in \Sigma_1\big\}.\end{aligned}$$

Proof. Rename the labels of $\mathrm{re}(\Pi_0)$ as follows to arrive at Π_1:

$$\begin{aligned}\mathsf{A} &= \{\mathsf{I}\},\\ \mathsf{B} &= \{\mathsf{O}, \mathsf{I}\}.\end{aligned}$$

□

In what follows, we will use Π_1 and $\mathrm{re}(\Pi_0)$ interchangeably, as they are equivalent.

10.4.3 Fixed Points in Round Elimination

As we will see soon, problem Π_1 is a *fixed point* in round elimination: when round elimination is applied to Π_1, the output problem is again Π_1 (or, more precisely, a problem equivalent to Π_1).

This means that if we assume that Π_1 can be solved in T rounds, then by applying round elimination T times we get a 0-round algorithm for Π_1. It can be shown that Π_1 is *not* 0-round solvable. This would seem to imply that Π_1, and thus sinkless orientation, are not solvable at all, which would contradict the existence of the $O(\log n)$-time algorithm presented in Section 10.2.1!

To resolve this apparent contradiction, we must take a closer look at the assumptions that we have made. The key step in round elimination happens when a node u simulates the *possible* outputs of its neighbors. The correctness of this step assumes that the possible T-neighborhoods of the neighbors are *independent* given the $(T-1)$-neighborhood of u. When T is so large in comparison with n that the T-neighborhoods of

the neighbors put together might already imply the existence of more than n nodes, this assumption no longer holds—see Figure 10.3 for an example.

For the remainder of this chapter we consider algorithms that know the value n, the number of nodes in the graph. This allows us to define a *standard form* for algorithms that run in $T = T(n)$ rounds, where $T(n)$ is some function of n: since n is known, each node can calculate $T(n)$, gather everything up to distance $T(n)$, and simulate any $T(n)$-time algorithm.

In (d, δ)-biregular trees, where $d > 2$, it can be shown that round elimination can be applied if the initial algorithm is assumed to have running time $T(n) = o(\log n)$: this guarantees the independence property in the simulation step. However, round elimination fails for some function $T(n) = \Theta(\log n)$; calculating this threshold is left as Exercise 10.7.

Any problem that can be solved in time $T(n)$ can be solved in time $T(n)$ with an algorithm in the standard form. If the problem is a fixed point and $T(n) = o(\log n)$, we can apply round elimination. We get the following lemma.

Lemma 10.6. *Assume that bipartite locally verifiable problem Π on (d, d)-biregular trees, for $d > 2$, is a fixed point. Then the deterministic complexity of Π in the bipartite* PN*-model is either 0 rounds or $\Omega(\log n)$ rounds, even if the number of nodes n is known.*

10.4.4 Sinkless Orientation Gives a Fixed Point

We will now show that the output problem $\Pi_1 = \mathrm{re}(\Pi_0)$ of the sinkless orientation problem Π_0 is a fixed point, that is, $\mathrm{re}(\Pi_1)$ is a problem equivalent to Π_1 itself. Since this problem cannot be solved in 0 rounds, it requires $\Omega(\log n)$ rounds. As sinkless orientation requires, by definition, one more round than its output problem, sinkless orientation also requires $\Omega(\log n)$ rounds.

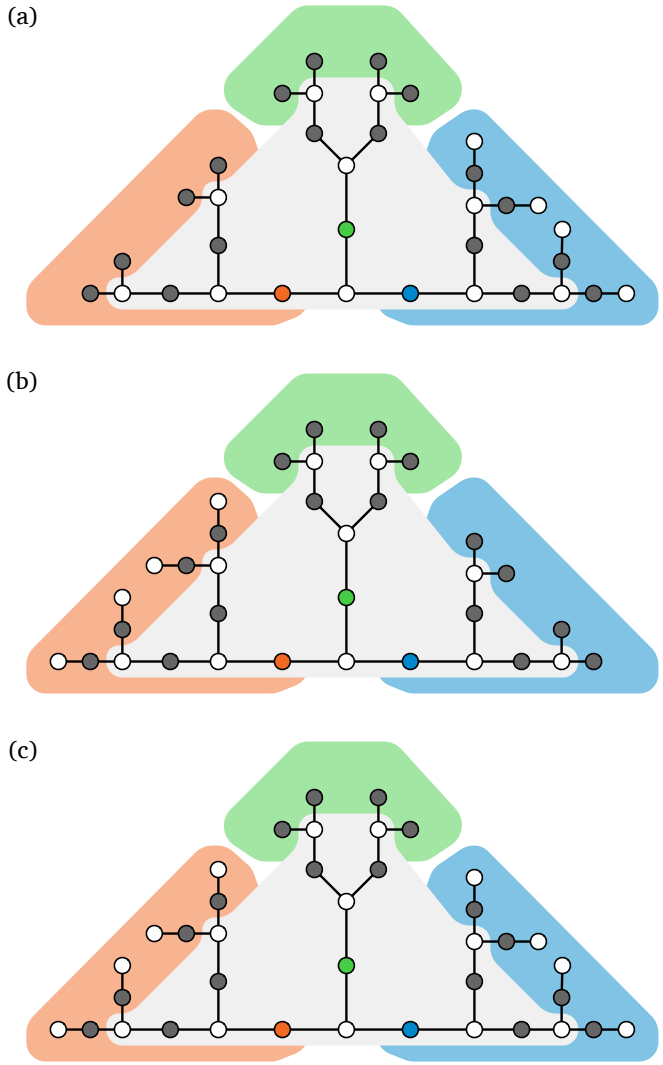


Figure 10.3: If we know that, e.g., $n = 35$, then orange, green, and blue extensions are no longer independent of each other: inputs (a) and (b) are possible but we cannot combine them arbitrarily to form e.g. input (c).

Lemma 10.7. *The output problem* $\Pi_1 = (\Sigma_1, \mathbf{A}_1, \mathbf{P}_1)$ *of sinkless orientation, given by*

$$\begin{aligned}\Sigma_1 &= \{\mathsf{A}, \mathsf{B}\},\\ \mathbf{A}_1 &= \big\{ [\mathsf{A}, \mathsf{B}, \mathsf{B}] \big\},\\ \mathbf{P}_1 &= \big\{ [\mathsf{B}, x, y] \;\big|\; x, y \in \Sigma_1 \big\},\end{aligned}$$

is a fixed point.

Proof. Let $\Pi_2 = \mathrm{re}(\Pi_1) = (\Sigma_2, \mathbf{A}_2, \mathbf{P}_2)$ denote the output problem of Π_1. Again, we have that

$$\Sigma_2 = \big\{ \{\mathsf{A}\}, \{\mathsf{B}\}, \{\mathsf{A}, \mathsf{B}\} \big\}.$$

The active configurations $\mathbf{A}_2$ are

$$\mathbf{A}_2 = \Big\{ \big[\{\mathsf{B}\}, x, y \big] \;\Big|\; x, y \subseteq \{\mathsf{A}, \mathsf{B}\} \Big\}.$$

That is, one set must be the singleton $\{\mathsf{B}\}$ to satisfy $\mathbf{P}_1$ for *all choices*, and the remaining sets are arbitrary.

Next we determine the maximal configurations. Again, there is a single active configuration that covers the other configurations:

$$\mathbf{A}_2 = \Big\{ \big[\{\mathsf{B}\}, \{\mathsf{A}, \mathsf{B}\}, \{\mathsf{A}, \mathsf{B}\} \big] \Big\}.$$

The alphabet is immediately simplified to

$$\Sigma_2 = \big\{ \{\mathsf{B}\}, \{\mathsf{A}, \mathsf{B}\} \big\},$$

as the label $\{\mathsf{A}\}$ is never used by any active configuration.

The passive configurations $\mathbf{P}_2$ are all multisets that contain the active configuration $[\mathsf{A}, \mathsf{B}, \mathsf{B}]$. Since A is now only contained in $\{\mathsf{A}, \mathsf{B}\}$, we get that

$$\begin{aligned}\mathbf{P}_2 = \Big\{ &\big[\{\mathsf{A}, \mathsf{B}\}, \{\mathsf{A}, \mathsf{B}\}, \{\mathsf{A}, \mathsf{B}\} \big],\\ &\big[\{\mathsf{A}, \mathsf{B}\}, \{\mathsf{A}, \mathsf{B}\}, \{\mathsf{B}\} \big],\\ &\big[\{\mathsf{A}, \mathsf{B}\}, \{\mathsf{B}\}, \{\mathsf{B}\} \big] \Big\}.\end{aligned}$$

Now we may do a simple renaming trick to see that Π_2 is equivalent to Π_1: rename $\{\mathsf{B}\} \to \mathsf{A}$ and $\{\mathsf{A}, \mathsf{B}\} \to \mathsf{B}$. Written this way, we have that Π_2 is equivalent to the following problem:

$$\begin{aligned}
\Sigma_2 &= \{\mathsf{A}, \mathsf{B}\}, \\
\mathbf{A}_2 &= \big\{ [\mathsf{A}, \mathsf{B}, \mathsf{B}] \big\}, \\
\mathbf{P}_2 &= \big\{ [\mathsf{B}, x, y] \;\big|\; x, y \in \Sigma_2 \big\},
\end{aligned}$$

which is exactly the same problem as Π_1. □

10.5 Quiz

Calculate the *number of different* 2*-round algorithms* in the PN model on $(3,3)$-biregular trees for bipartite locally verifiable problems with the binary alphabet $\Sigma = \{0, 1\}$.

Here two algorithms A_1 and A_2 are considered to be *different* if there is some port-numbered network N and some edge e such that the label edge e in the output of A_1 is different from the label of the same edge e in algorithm A_2. Note that A_1 and A_2 might solve the same problem, they just solved it differently. Please remember to take into account that in a $(3,3)$-biregular tree each node has degree 1 or 3.

Please give your answer in the scientific notation with two significant digits: your answer should be a number of the form $a \cdot 10^b$, where a is rounded to two decimal digits, we have $1 \le a < 10$, and b is a natural number.

10.6 Exercises

Exercise 10.1 (0-round solvability)**.** Prove that the following problems are not 0-round solvable.

(a) $(d+1)$-edge coloring in (d,d)-biregular trees (see Section 9.1.2).

(b) Maximal matching in (d,d)-biregular trees (see Section 9.1.2).

(c) Π_1, the output problem of sinkless orientation, in $(3,3)$-biregular trees (see Section 10.4.2).

▷ *hint W*

Exercise 10.2 (higher degrees)**.** Generalize the sinkless orientation problem from $(3,3)$-biregular trees to $(10,10)$-biregular trees. Apply round elimination and show that you get again a fixed point.

Exercise 10.3 (non-bipartite sinkless orientation)**.** Define non-bipartite sinkless orientation as the following problem $\Pi = (\Sigma, \mathbf{A}, \mathbf{P})$ on $(3,2)$-biregular trees:

$$\begin{aligned}\Sigma &= \{\mathsf{O}, \mathsf{I}\},\\ \mathbf{A} &= \big\{ [\mathsf{O}, x, y] \;\big|\; x, y \in \Sigma \big\},\\ \mathbf{P} &= \big\{ [\mathsf{I}, \mathsf{O}] \big\}.\end{aligned}$$

Prove that applying round elimination to Π leads to a period-2 point, that is, to a problem Π' such that $\Pi' = \mathrm{re}(\mathrm{re}(\Pi'))$.

Exercise 10.4 (matching lower bound)**.** Let us define *sloppy perfect matching* in trees as a matching such that all non-leaf nodes are matched. Encode this problem as a bipartite locally verifiable problem on $(3,3)$-biregular trees. Show that solving it requires $\Omega(\log n)$ rounds in the PN-model with deterministic algorithms.

Exercise 10.5 (matching upper bound)**.** Consider the sloppy perfect matching problem from Exercise 10.4. Design an algorithm for solving it with a deterministic PN-algorithm on $(3,3)$-biregular trees in $O(\log n)$ rounds.

▷ *hint X*

Exercise 10.6 (sinkless and sourceless)**.** Sinkless and sourceless orientation is the problem of orienting the edges of the graph so that each non-leaf node has at least one outgoing edge *and* at least one incoming edge. Encode the sinkless and sourceless orientation problem as a binary locally verifiable labeling problem on $(5,5)$-biregular trees. Design an $O(\log n)$-round algorithm for solving sinkless and sourceless orientation on $(5,5)$-biregular trees.

Exercise 10.7 (failure of round elimination)**.** In Section 10.4.3 we discussed that round elimination fails if in the simulation step the $T(n)$-neighborhoods of the neighbors are dependent from each other. This happens when there exist T-neighborhoods of the neighbors such that the resulting tree would have more than n nodes. Consider (d, d)-biregular trees. Calculate the bound for $F(n)$ such that round elimination fails for algorithms with running time $T(n) \geq F(n)$.

⋆ **Exercise 10.8.** Design an $O(\log n)$-round algorithm for solving sinkless and sourceless orientation on $(3, 3)$-biregular trees.

10.7 Bibliographic Notes

Brandt et al. [11] introduced the sinkless orientation problem and proved that it cannot be solved in $o(\log \log n)$ rounds with randomized algorithms, while Chang et al. [12] showed that it cannot be solved in $o(\log n)$ rounds with deterministic algorithms. Ghaffari and Su [21] gave matching upper bounds.

Exercises 10.4 and 10.5 are inspired by [6], and Exercises 10.6 and 10.8 are inspired by [20].

Chapter 11

Hardness of Coloring

This week we will apply round elimination to coloring. We will show that 3-coloring paths requires $\Omega(\log^* n)$ rounds. This matches the fast coloring algorithms that we saw in Chapter 1.

To prove this result, we will see how to apply round elimination to randomized algorithms. Previously round elimination was purely deterministic: a T-round deterministic algorithm would imply a $(T-1)$-round deterministic algorithm for the output problem. With randomized algorithms, round elimination affects the success probability of the algorithm: a T-round randomized algorithm implies a $(T-1)$-round randomized algorithm for the output problem with a worse success probability.

We will see how round elimination can be applied in the presence of inputs. These inputs can be, in addition to randomness, e.g. a coloring or an orientation of the edges. The critical property for round elimination is that there are no *long range dependencies* in the input.

11.1 Coloring and Round Elimination

We begin by applying round elimination to coloring on paths, or $(2,2)$-biregular trees. For technical reasons, we also encode a consistent orientation in the coloring. That is, in addition to computing a coloring, we require that the nodes also orient the path consistently from one endpoint to the other. This is a hard problem, as we saw in the previous chapter; therefore we will assume that the input is *already oriented*. We will show that 3-coloring a path requires $\Omega(\log^* n)$ rounds *even if the path is consistently oriented*.

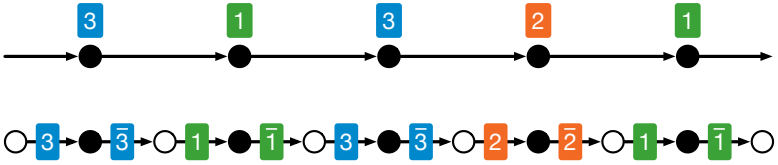


Figure 11.1: Encoding of 3-coloring in the bipartite formalism. On top, a 3-coloring of a path fragment. Below, the corresponding 3-coloring as a bipartite locally verifiable problem. The path is assumed to be consistently oriented, so each node has an incoming and an outgoing edge. They use the regular label on the incoming edge, and the barred label on the outgoing edge. Passive nodes verify that the colors differ and have different type.

11.1.1 Encoding Coloring

We will study the problem of 3-coloring the *active nodes* of a $(2,2)$-biregular tree. We will say that two active nodes are *adjacent* if they share a passive neighbor.

To encode the orientation, we use two versions of each color label: e.g. 1 and $\bar{1}$. We call these *regular* and *barred* labels, respectively. For 3-coloring, we have the following problem $\Pi_0 = (\Sigma_0, \mathbf{A}_0, \mathbf{P}_0)$:

$$\begin{aligned}
\Sigma_0 &= \{1, \bar{1}, 2, \bar{2}, 3, \bar{3}\},\\
\mathbf{A}_0 &= \big\{[1,\bar{1}], [2,\bar{2}], [3,\bar{3}]\big\},\\
\mathbf{P}_0 &= \big\{[1,\bar{2}], [1,\bar{3}], [2,\bar{1}], [2,\bar{3}], [3,\bar{1}], [3,\bar{2}]\big\}.
\end{aligned}$$

The encoding of 3-coloring is shown in Figure 11.1. The active configurations ensure that each node chooses a color and an orientation of its edges: we can think of the edges labeled with $\bar{1}$, $\bar{2}$ or $\bar{3}$ as outgoing edges, and the regular labels as incoming edges. The passive configurations ensure that adjacent active nodes are properly colored and that the passive node is properly oriented (has incident labels of different types).

We will also need to define coloring with more colors. We say that a label *matches* with its barred version: $1 \sim \bar{1}$, $2 \sim \bar{2}$ and so on. A label *does not match* with the other labels: e.g. $1 \not\sim \bar{2}$.

We define c-coloring as the following problem $\Pi = (\Sigma, \mathbf{A}, \mathbf{P})$:

$$\begin{aligned}
\Sigma &= \{1, \bar{1}, 2, \bar{2}, \ldots, c, \bar{c}\}, \\
\mathbf{A} &= \left\{ [x, \bar{x}] \;\middle|\; x \in \{1, 2, \ldots, c\} \right\}, \\
\mathbf{P} &= \left\{ [x, \bar{y}] \;\middle|\; x, y \in \{1, 2, \ldots, c\},\ x \nsim \bar{y} \right\}.
\end{aligned}$$

11.1.2 Output Problem of Coloring

We start by assuming that we have a fast algorithm that solves the 3-coloring problem Π_0. Let us now compute the output problem $\Pi_1 = \mathrm{re}(\Pi_0)$ of 3-coloring Π_0.

Let $\Pi_1 = (\Sigma_1, \mathbf{A}_1, \mathbf{P}_1)$ denote the output problem. For now, we will let Σ_1 consist of all non-empty subsets of Σ_0, and prune it later.

Recall that passive configurations $\mathbf{P}_0$ consist of all non-matching pairs of a regular and barred label. Therefore the active configurations in Π_1 consist of all pairs of sets such that

- one set consists of regular and one of barred labels, and
- there are no matching labels.

We get that

$$\mathbf{A}_1 = \left\{ [X, Y] \;\middle|\; X \subseteq \{1, 2, 3\}, Y \subseteq \{\bar{1}, \bar{2}, \bar{3}\}, \forall x \in X, y \in Y : x \nsim y \right\}.$$

Next we make the sets maximal: when neither the regular or the barred version of a label is contained in either set, we can add the corresponding variant to either set. Thus the maximal active configurations *split* the color set over their edges:

$$\begin{aligned}
\mathbf{A}_1 = \Big\{ &\big[\{1\}, \{\bar{2}, \bar{3}\}\big], \big[\{2\}, \{\bar{1}, \bar{3}\}\big], \big[\{3\}, \{\bar{1}, \bar{2}\}\big], \\
&\big[\{\bar{1}\}, \{2, 3\}\big], \big[\{\bar{2}\}, \{1, 3\}\big], \big[\{\bar{3}\}, \{1, 2\}\big] \Big\}
\end{aligned}$$

No label can be added to any of the configurations, and the above labels contain all active configurations.

We have the following alphabet:

$$\Sigma_1 = \big\{\{1\},\{2\},\{3\},\{1,2\},\{1,3\},\{2,3\},\\ \{\bar{1}\},\{\bar{2}\},\{\bar{3}\},\{\bar{1},\bar{2}\},\{\bar{1},\bar{3}\},\{\bar{2},\bar{3}\}\big\}.$$

Finally, the passive configurations consist of all pairs such that it is possible to pick matching regular and barred labels, forming a configuration in $\mathbf{A}_0$:

$$\mathbf{P}_1 = \big\{[X,Y] \;\big|\; (1 \in X, \bar{1} \in Y) \vee (2 \in X, \bar{2} \in Y) \vee (3 \in X, \bar{3} \in Y)\big\}.$$

11.1.3 Simplification

The output problem of 3-coloring looks much more complicated than the problem we started with. If we kept applying round elimination, it would become extremely difficult to understand the structure of the problem. Therefore we will *simplify* the problem: we will map it back to a coloring with a *larger number of colors*.

The intuition is the following. Assume that our original labels consist of some set of c colors, and the output problem has sets of these colors as labels. Then there are at most 2^c different sets. If adjacent nodes have always different sets, we can treat it as a coloring with 2^c colors by mapping the sets to the labels $1, 2, \ldots, 2^c$.

Now consider the output problem of 3-coloring, Π_1. We will treat the different sets of the regular labels as the color classes. Each of them is paired with a unique set of barred labels. Enumerating all options, we rename the labels as follows to match the alphabet of 6-coloring:

$$\begin{aligned} \{1\} &\mapsto 1, & \{\bar{2},\bar{3}\} &\mapsto \bar{1},\\ \{2\} &\mapsto 2, & \{\bar{1},\bar{3}\} &\mapsto \bar{2},\\ \{3\} &\mapsto 3, & \{\bar{1},\bar{2}\} &\mapsto \bar{3},\\ \{1,2\} &\mapsto 4, & \{\bar{3}\} &\mapsto \bar{4},\\ \{1,3\} &\mapsto 5, & \{\bar{2}\} &\mapsto \bar{5},\\ \{2,3\} &\mapsto 6, & \{\bar{1}\} &\mapsto \bar{6}. \end{aligned}$$

Now let us verify that this is indeed a 6-coloring. After renaming, the active configurations are

$$\begin{aligned}\mathbf{A}_1 = \big\{&[1,\bar{1}],\,[2,\bar{2}],\,[3,\bar{3}],\\ &[\bar{6},6],\,[\bar{5},5],\,[\bar{4},4]\big\}.\end{aligned}$$

By rearrangement we can see that these match exactly the definition of 6-coloring. The passive configurations, before renaming, were the pairs of sets, one consisting of the regular labels and the other of the barred labels, that contained a matching label. We get that

$$\begin{aligned}\mathbf{P}_1 = &\big\{[1,x] \;\big|\; x \in \{\bar{2},\bar{3},\bar{6}\}\big\}\\ &\cup \big\{[2,x] \;\big|\; x \in \{\bar{1},\bar{3},\bar{5}\}\big\}\\ &\cup \big\{[3,x] \;\big|\; x \in \{\bar{1},\bar{2},\bar{4}\}\big\}\\ &\cup \big\{[4,x] \;\big|\; x \in \{\bar{1},\bar{2},\bar{3},\bar{5},\bar{6}\}\big\}\\ &\cup \big\{[5,x] \;\big|\; x \in \{\bar{1},\bar{2},\bar{3},\bar{4},\bar{6}\}\big\}\\ &\cup \big\{[6,x] \;\big|\; x \in \{\bar{1},\bar{2},\bar{3},\bar{4},\bar{5}\}\big\}.\end{aligned}$$

We notice that these are a subset of the passive configurations of 6-coloring: colors 1, 2, and 3 cannot be paired with some of the non-matching colors. This means that Π_1 is *at least as hard* to solve as 6-coloring.

We may *relax* the output problem Π_1 and construct a new problem $\Pi_1' = (\Sigma_1', \mathbf{A}_1', \mathbf{P}_1')$ as follows:

$$\begin{aligned}\mathbf{A}_1' &= \mathbf{A}_1,\\ \mathbf{P}_1' = &\big\{[1,x] \;\big|\; x \in \{\bar{2},\bar{3},\bar{4},\bar{5},\bar{6}\}\big\}\\ &\cup \big\{[2,x] \;\big|\; x \in \{\bar{1},\bar{3},\bar{4},\bar{5},\bar{6}\}\big\}\\ &\cup \big\{[3,x] \;\big|\; x \in \{\bar{1},\bar{2},\bar{4},\bar{5},\bar{6}\}\big\}\\ &\cup \big\{[4,x] \;\big|\; x \in \{\bar{1},\bar{2},\bar{3},\bar{5},\bar{6}\}\big\}\\ &\cup \big\{[5,x] \;\big|\; x \in \{\bar{1},\bar{2},\bar{3},\bar{4},\bar{6}\}\big\}\\ &\cup \big\{[6,x] \;\big|\; x \in \{\bar{1},\bar{2},\bar{3},\bar{4},\bar{5}\}\big\}.\end{aligned}$$

Note that Π_1' is exactly the 6-coloring problem. As we have that $\mathbf{A}_1' = \mathbf{A}_1$ and $\mathbf{P}_1' \supseteq \mathbf{P}_1$, any solution to Π_1 is also a solution to Π_1'. We conclude that if we can solve problem Π_0 in T rounds, we can solve $\Pi_1 = \mathrm{re}(\Pi_0)$ *exactly* one round faster, and we can solve its relaxation Π_1' *at least* one round faster.

11.1.4 Generalizing Round Elimination for Coloring

Let us now see how to generalize the first round elimination step. In the first step, we saw that 6-coloring is at least as easy to solve as the output problem of 3-coloring.

Now consider applying round elimination to the c-coloring problem. Let $\mathrm{re}(\Pi_0) = \Pi_1 = (\Sigma_1, \mathbf{A}_1, \mathbf{P}_1)$ denote the output problem of c-coloring Π_0.

Again, the active configurations in $\mathbf{A}_1$ consist of all splits of the colors. For a set $X \subseteq \{1, 2, \ldots, c\}$, let $\bar{X}$ denote the *barred complement of* X:

$$\bar{X} = \{\bar{x} \mid x \in \{1, 2, \ldots, c\} \setminus X\}.$$

Then the active configurations are

$$\mathbf{A}_1 = \left\{[X, \bar{X}] \;\middle|\; \varnothing \neq X \subsetneq \{1, 2, \ldots, c\}\right\}.$$

The labels are all non-empty and non-full subsets of the regular and barred labels, respectively. The passive configurations in $\mathbf{P}_1$ consists of pairs of sets such that it is possible to pick matching regular and barred labels from them:

$$\mathbf{P}_1 = \left\{[X, Y] \;\middle|\; x \in X, \bar{x} \in Y : x \sim \bar{x}\right\}.$$

We do the exact same renaming trick as in the previous section. There are a total of $2^c - 2$ different sets on regular labels. We rename them in some order with the integers from 1 to $2^c - 2$. For each set X renamed to integer y, we rename the unique barred complement $\bar{X}$ to $\bar{y}$. The active configurations after renaming are

$$\mathbf{A}_1 = \left\{[x, \bar{x}] \;\middle|\; x \in \{1, 2, \ldots, 2^c - 2\}\right\}.$$

We note that passive configurations never include $[x, \bar{x}]$ for any x. This is because $\bar{x}$ represents the complement of x as a set: it is not possible to pick a matching element from x and $\bar{x}$. Therefore we may again *relax* the passive configurations to be the configurations for c-coloring:

$$\mathbf{P}_1 = \left\{ [x, \bar{y}] \;\middle|\; x, y \in \{1, 2, \ldots, 2^c - 2\},\ x \nsim \bar{y} \right\}.$$

The resulting problem is *at least as easy* as the output problem of c-coloring: if c-coloring can be solved in T rounds, then coloring with $2^c - 2$ colors can be solved in *at most* $T - 1$ rounds.

Now in what follows, it will be awkward to use the expression $2^c - 2$, so we will simply round it up to 2^c. Clearly, coloring with 2^c colors is at least as easy as coloring with $2^c - 2$ colors.

11.1.5 Sequence of Output Problems

We have shown that 2^c-coloring is *at least as easy* as the output problem of c-coloring. Now if we were to iteratively apply round elimination k times in the PN-model we would get the following sequence of problems:

$$\begin{aligned} \Pi_0 = 3\text{-coloring} &\to \Pi_1 = 2^3\text{-coloring} \\ &\to \Pi_2 = 2^{2^3}\text{-coloring} \\ &\to \Pi_3 = 2^{2^{2^3}}\text{-coloring} \\ &\cdots \\ &\to \Pi_k = C(k)\text{-coloring}, \end{aligned}$$

where

$$C(k) = \underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2^3}}}}}}_{k \text{ times 2 and one 3}}.$$

Now if we show that coloring with $C(k)$ colors cannot be solved in 0 rounds in the PN model with deterministic algorithms, it would imply that 3-coloring cannot be solved in k rounds in the PN model. This result, however, would not be very meaningful. As we have already

seen in Chapter 7, the vertex coloring problem cannot be solved at all in the PN-model! Therefore we must strengthen the round elimination technique itself to apply in the LOCAL model.

11.2 Round Elimination with Inputs

If we try to do round elimination in the LOCAL model, we run into a technical challenge. In round elimination, the nodes simulate the outputs of their neighbors. It is crucial that the inputs of the neighbors are independent: for each combination of possible outputs, there must exist a network in which the algorithm actually produces those outputs. This step no longer holds in the LOCAL model: the identifiers are globally unique, and therefore do not repeat. The inputs of the neighbors *are dependent*: if one of the regions contains, e.g., a node with identifier 1, then another region cannot contain identifier 1, and vice versa.

To overcome this difficulty, we will consider *randomized algorithms*. In Exercise 6.2 we saw that randomness can be used to generate unique identifiers. Therefore the PN-model, equipped with randomness, is at least as powerful as the LOCAL model: if a problem Π can be solved in time $T(n)$ in the LOCAL model, then we can generate unique identifiers with high probability and simulate the LOCAL-algorithm in $T(n)$ rounds. This clearly succeeds if the random identifiers were unique. Therefore any impossibility results we prove for randomized algorithms also hold for the LOCAL model.

For the remainder of the chapter, we assume that the nodes receive two types of inputs.

(a) Random inputs. Each node receives some arbitrary but finite number of uniform random bits as input. In addition, to simplify the proof, we will assume that the port numbers are assigned randomly. This latter assumption is made only for the purposes of this proof, we do not change the standard assumptions about the models.

(b) Consistent orientation. As we mentioned in the beginning, for technical reasons we consider a variant of coloring that includes an orientation. To solve this part of the problem easily, we include the required orientation as input.

It is crucial that we can apply round elimination in the presence of these inputs. First, consider the orientation. In the round elimination step, each node must simulate the outputs of its neighbors over all possible inputs. Now we add the promise that each node, in addition to its usual inputs, receives an orientation of its edges as input. In particular, they form a *consistent orientation* of the whole path from one end to the other. Clearly nodes can include this input in their simulation, as the orientation of the remaining edges is fixed after seeing the orientation of just one edge. Similarly, random inputs do not have any long-range dependencies. They do, however, affect the simulation step. We will discuss randomized round elimination in the next section.

11.2.1 Randomized Round Elimination Step

We will now introduce a variant of the Round Elimination Lemma that we proved in Chapter 9. Assume that we have a randomized algorithm that solves problem Π in $T(n)$ rounds with *local failure probability* q: that is, each active node chooses an active configuration of Π with probability *at least* $1-q$, and each passive node is labeled according to some passive configuration of Π with probability *at least* $1-q$. Then we want to show that there is an algorithm that solves the output problem $\mathrm{re}(\Pi)$ in $T(n)-1$ rounds with some local failure probability at most $f(q)$ for some function f.

The round elimination step works essentially as in the PN-model. Each $\mathrm{re}(\Pi)$-active node simulates the outputs of its Π-active nodes over all possible inputs, including the random bits. We make one modification to the model: we assume that the port numbers are assigned randomly instead of being assigned by an adversary. This modification is made to simplify the analysis that follows. It does not affect the power of the model: the nodes could use their local randomness to shuffle the

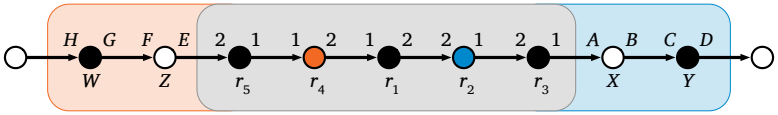


Figure 11.2: The randomized round elimination step. Assume an algorithm A for c-coloring with running time $T = 3$. The simulation functions as follows. The active node gathers its $(T-1)$-neighborhood, including the assignment of random port numbers and random bits r_1, r_2, r_3, r_4 and r_5. Then it simulates A on its right and left neighbor. On the right, over all possible assignments of ports A, B, C, D and random bit strings X, Y. On the left, over all possible assignments of ports E, F, G, H and random bit strings Z, W. For each edge, it outputs the set of labels that appear as outputs for at least fraction $t(q)$ of inputs.

ports and any algorithm designed for the worst-case port numbering also works, by definition, with a random port numbering. We will also only consider nodes that are internal nodes in the $(2,2)$-biregular tree: we assume that T-neighborhoods of the neighbors of re(Π)-active nodes do not contain the endpoints of the path.

It no longer makes sense to construct the set of *all possible* outputs. We can imagine an algorithm that tries to make round elimination hard: it always uses each possible output label with at least one specific random input labeling. These outputs would make no sense, but would cause the set of *possible* output labels to always be the full set of labels. Since the random inputs can be arbitrarily large (but finite!), the failure probability this would add to the algorithm would be small.

To avoid this issue, we will define a *threshold* $t(q)$: an output label $\sigma \in \Sigma$ is *frequent* if it appears as the output label with probability at least $t(q)$. More formally, for a fixed re(Π)-active node u, an assignment of random inputs to $\mathrm{ball}_N(u, T-1)$, and a neighbor v, label $\sigma \in \Sigma$ is frequent for the edge $\{u, v\}$ if the probability that v outputs σ on $\{u, v\}$, conditioned on fixing the random inputs in $\mathrm{ball}_N(u, T-1)$, is at least $t(q)$. We will fix the value of this threshold later.

The randomized simulation step is defined as follows. Each Π_1-active node u gathers its $(T-1)$-neighborhood. Then it computes for each edge

$\{u, v\}$ the set of frequent labels $S(u, v)$ and outputs that set on $\{u, v\}$. See Figure 11.2 for an illustration.

We will prove that randomized round elimination works for the special case of c-coloring $(2, 2)$-biregular trees. This can be generalized to cover all bipartite locally verifiable problems in (d, δ)-biregular trees, for any parameters d and δ.

Lemma 11.1 (Randomized Round Elimination Lemma)**.** *Assume that there is an algorithm that solves the c-coloring problem on $(2, 2)$-biregular trees in the randomized* PN*-model in $T(n)$ rounds with local failure probability at most q. Then there exists an algorithm that solves the 2^c-coloring problem in $T(n) - 1$ rounds with local failure probability at most $3cq^{1/3}$.*

Intuitively the lemma is true, as in most neighborhoods the true outputs must also appear frequently in the simulation. Similarly, combinations of non-configurations cannot be frequent too often, as this would imply that the original algorithms also fails often.

We prove the lemma in Section 11.3.1. Next we will see how to apply it to prove an impossibility result for 3-coloring.

11.3 Iterated Randomized Round Elimination

Given the randomized round elimination lemma, we will proceed as follows.

(1) Assume there is a randomized $(\log^* n - 4)$-round algorithm for solving 3-coloring on *consistently oriented* $(2, 2)$-biregular trees. Since randomized algorithms are assumed to succeed *with high probability*, the local failure probability q_0 must be at most $1/n^k$ for some constant k.

(2) Apply randomized round elimination $T(n) = \log^* n - 4$ times to get a 0-round randomized algorithm for $c_{T(n)}$-coloring with some local failure probability $q_{T(n)}$. For the chosen value of $T(n)$ show that we have $q_{T(n)} < 1/c_{T(n)}$.

(3) Prove that there are no 0-round algorithms for solving $c_{T(n)}$-coloring with local failure probability $q_{T(n)} < 1/c_{T(n)}$.

We must show that fast coloring algorithms imply 0-round coloring algorithms that do not fail with large enough probability. In Section 11.3.2 we will prove the following lemma.

Lemma 11.2. *Assume that there is a* $(\log^* n - 4)$*-round 3-coloring algorithm in the randomized* PN*-model. Then there is a 0-round* c*-coloring algorithm with local failure probability* $q < 1/c$.

We must also show that any 0-round c-coloring algorithm fails locally with probability at least $1/c$.

Lemma 11.3. *Any 0-round* c*-coloring algorithm fails locally with probability at least* $1/c$.

Proof. Any 0-round c-coloring algorithm defines a probability distribution over the possible output colors. This distribution is the same for each active node inside a path: they are indistinguishable from each other in 0 rounds. The algorithm fails if two adjacent nodes select the same color.

Let $p_i = b_i + 1/c$ denote the probability that the algorithm outputs color i. The terms b_i denote the deviation of each probability p_i from the average: we must have that

$$\sum_{i=1}^{c} b_i = 0.$$

The local failure probability is at least

$$\begin{aligned}\sum_{i=1}^{c} p_i^2 &= \sum_{i=1}^{c} (b_i + 1/c)^2 \\ &= \sum_{i=1}^{c} \left(b_i^2 + 2b_i/c + 1/c^2\right) \\ &= 1/c + \left(\sum_{i=1}^{c} b_i^2\right) + 2/c \cdot \left(\sum_{i=1}^{c} b_i\right) \\ &= 1/c + \left(\sum_{i=1}^{c} b_i^2\right) + 0.\end{aligned}$$

This is clearly minimized by setting $b_i = 0$ for all i. Thus the local failure probability is minimized when $p_i = 1/c$ for all i, and we get that it is at least $1/c$. □

The bound on the failure probability of 0-round coloring algorithms combined with Lemma 11.2 shows that there is no 3-coloring algorithm that runs in at most $\log^* n - 4$ rounds in the randomized PN-model, even if we know n and the path is consistently oriented. Since the randomized PN-model can simulate the LOCAL model with high probability, this implies that there is no (randomized) 3-coloring algorithm in the LOCAL model that runs in at most $\log^* n - 4$ rounds.

Theorem 11.4. *3-coloring (in the bipartite formalism) cannot be solved in the* LOCAL *model in less than* $\log^* n - 4$ *rounds.*

Corollary 11.5. *3-coloring (in the usual sense) cannot be solved in the* LOCAL *model in less than* $\frac{1}{2}\log^* n - 2$ *rounds.*

Proof. Distances between nodes increase by a factor of 2 when we switch to the bipartite encoding (see Figure 11.1). □

This result is asymptotically optimal: already in Chapter 1 we saw that paths *can be colored* with 3 colors in time $O(\log^* n)$.

In the final two sections we give the proofs for Lemmas 11.1 and 11.2.

11.3.1 Proof of Lemma 11.1

In this section we prove Lemma 11.1. The proof consists of bounding the local failure probability of the simulation algorithm given in Section 11.2.1.

Proof of Lemma 11.1. Let $\Pi_0 = (\Sigma_0, \mathbf{A}_0, \mathbf{P}_0)$ denote the c-coloring problem for some c, and let $\Pi_1 = \mathrm{re}(\Pi_0) = (\Sigma_1, \mathbf{A}_1, \mathbf{P}_1)$ denote the output problem of c-coloring. Assume that there is a T-round randomized algorithm A for solving Π_0 with local failure probability q. Consider an arbitrary Π_1-active node u with some fixed $(T-1)$-neighborhood $\mathrm{ball}_N(u, T-1)$ (including the random inputs).

The passive configurations $\mathbf{P}_0$ consist of pairs $[x, \bar{y}]$ that do not match. The algorithm A fails if it outputs any x and $\bar{x}$, or two labels of the same type on the incident edges of a Π_0-passive node. We say that the $(T-1)$-neighborhood $\mathrm{ball}_N(u, T-1)$ is *lucky*, if the algorithm A fails in labeling the incident edges of u, given $\mathrm{ball}_N(u, T-1)$, with probability less than t^2. Here t is the probability threshold for frequent labels; we will choose the value of t later.

We want to prove that most random bit assignments must be lucky, and that in lucky neighborhoods the simulation succeeds with a good probability. We will ignore the other cases, and simply assume that in those cases the simulation can fail.

Consider any fixed $\mathrm{ball}_N(u, T+1)$ *without* the random bits and the random port numbering: since we consider nodes inside the path, the remaining structure is the same for all nodes. The randomness determines whether A succeeds around u. Let L denote the event that $\mathrm{ball}_N(u, T-1)$ is lucky. Since we know that A fails in any neighborhood with probability at most q, we can bound the probability of a neighborhood *not being lucky* as follows:

$$\begin{aligned} &\Pr[A \text{ fails at } u] \geq \Pr[A \text{ fails at } u \mid \text{not } L] \cdot \Pr[\text{not } L] \\ \implies &\Pr[\text{not } L] \leq \frac{\Pr[A \text{ fails at } u]}{\Pr[A \text{ fails at } u \mid \text{not } L]} < \frac{q}{t^2}. \end{aligned}$$

From now on we will assume that $\text{ball}_N(u, T-1)$ is lucky. Let v and w denote the passive neighbors of re(Π)-active node u. The simulation fails if and only if the sets $S(u,v)$ and $S(u,w)$ contain labels x and y such that $[x,y] \notin \mathbf{P}_0$ (all choices do not yield a configuration in $\mathbf{P}_0$). Since both of these labels are frequent (included in the output), each of them must appear with probability at least t given $\text{ball}_N(u, T-1)$. But since these labels are frequent, we have that the original algorithm fails, given $\text{ball}_N(u, T-1)$ with probability at least t^2, and the neighborhood cannot be lucky. We can deduce that the simulation *always succeeds in lucky neighborhoods*. Therefore we have that

$$\Pr[\text{simulation fails}] \leq \Pr[\text{not } L] < \frac{q}{t^2}.$$

Next we must determine the failure probability of the simulation around *passive nodes*. The re(Π)-passive nodes succeed when the *true output* of the algorithm is contained in the sets of frequent outputs. For a re(Π)-passive neighbor v, consider the event that its output on edge $\{u,v\}$ in the original algorithm is not contained in the set of frequent outputs $S(u,v)$ based on $\text{ball}_N(u, T-1)$. By definition, for each fixed $\text{ball}_N(u, T-1)$ each infrequent label is the true output with probability at most t. There are c colors, so by union bound one of these is the true output color with probability at most ct. There are two neighbors, so by another union bound the total failure probability for a passive node is at most $2ct$. Since the simulation can also fail when the original algorithm fails, we get for each passive node that

$$\Pr[\text{simulation fails}] \leq q + 2ct.$$

To minimize the maximum of the two failure probabilities, we can for example set $t = q^{1/3}$ and get that

$$\frac{q}{t^2} = q^{1/3} \leq q + 2cq^{1/3} \leq 3cq^{1/3}. \qquad \square$$

11.3.2 Proof of Lemma 11.2

It remains to show that a fast 3-coloring algorithm implies a 0-round c-coloring algorithm that fails locally with a probability less than $1/c$.

Proof of Lemma 11.2. Assume there is a T-round 3-coloring algorithm that succeeds with high probability. This implies that it has a local failure probability $q_0 \leq 1/n^k$ for some constant $k \geq 1$. Applying the round elimination lemma, this implies the following coloring algorithms and local failure probabilities q_i:

$$\begin{aligned}
&C(0) = 3 \text{ colors:} && q_0 \leq 1/n^k \leq 1/n,\\
&C(1) = 2^3 \text{ colors:} && q_1 \leq 3C(0) \cdot q_0^{1/3},\\
&C(2) = 2^{2^3} \text{ colors:} && q_2 \leq 3C(1) \cdot q_1^{1/3} = 3C(1) \cdot \left(3C(0) \cdot q_0^{1/3}\right)^{1/3}.
\end{aligned}$$

Generalizing, we see that after T iterations, the local failure probability is bounded by

$$q_T \leq \left(\prod_{i=0}^{T} 3^{1/3^i}\right)\left(\prod_{i=0}^{T} C(T-i)^{1/3^i}\right) \cdot n^{-1/3^T},$$

and the algorithm uses

$$C(T) = \underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2^3}}}}}}_{\substack{T \text{ times } 2\\ \text{and one } 3}} < \underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2^{2^2}}}}}}}_{T+2 \text{ times } 2} = {}^{T+2}2$$

colors. To finish the proof, we must show that for any $T(n) \leq \log^* n - 4$ and for a sufficiently large n we have that $q_T < 1/C(T)$, or, equivalently, $q_T \cdot C(T) < 1$ or

$$\log q_T + \log C(T) < 0; \tag{11.1}$$

here all logarithms are to the base 2. Evaluating the expression $\log q_T$, we get that

$$\begin{aligned}\log q_T &\leq \sum_{i=0}^{T} \frac{1}{3^i} \log 3 + \sum_{i=0}^{T} \frac{1}{3^i} \log C(T-i) - 3^{-T} \log n \\ &\leq \frac{3}{2} \log 3 + \frac{3}{2} \log C(T) - 3^{-T} \log n,\end{aligned}$$

since the sums are converging geometric sums. Therefore

$$\log q_T + \log C(T) \leq \frac{3}{2} \log 3 + \frac{5}{2} \log C(T) - 3^{-T} \log n. \tag{11.2}$$

Note that

$$n \leq {}^{\log^* n}2 < 2^n.$$

Therefore for $T = \log^* n - 4$ we have that

$$\log C(T) < \log {}^{T+2}2 = \log\log\log {}^{T+4}2 < \log\log n. \tag{11.3}$$

On the other hand, for a large enough n we have that

$$3^T < 3^{\log^* n} < 3^{\log_3 \log\log n} < \log\log n,$$

and therefore

$$3^{-T} \log n > \frac{\log n}{\log\log n}. \tag{11.4}$$

Now (11.3) and (11.4) imply that for a sufficiently large n, term $3^{-T} \log n$ will dominate the right hand side of (11.2), and we will eventually have

$$\log q_T + \log C(T) < 0,$$

which is exactly what we needed for Equation (11.1). □

11.4 Quiz

Construct the *smallest possible* (i.e., fewest nodes) properly 4-colored cycle C such that the following holds: if you take *any* deterministic 0-round PN-algorithm A and apply it to C, then the output of A is not a valid 3-coloring of C.

Please note that here we are working in the usual PN model, exactly as it was originally specified in Chapter 3, and we are doing graph coloring in the usual sense (we do not use the bipartite formalism here). Please give the answer by listing the n colors of the n-cycle C.

11.5 Exercises

Exercise 11.1 (randomized 2-coloring)**.** Prove that solving 2-coloring paths in the randomized PN-model requires $\Omega(n)$ rounds.

Exercise 11.2 (coloring grids)**.** Prove that 5-coloring 2-dimensional grids in the deterministic LOCAL model requires $\Omega(\log^* n)$ rounds.

A 2-dimensional grid $G = (V, E)$ consists of n^2 nodes $v_{i,j}$ for $i, j \in \{1, \ldots, n\}$ such that if $i < n$, add $\{v_{i,j}, v_{i+1,j}\}$ to E for each j, and if $j < n$ add $\{v_{i,j}, v_{i,j+1}\}$ to E for each i.

▷ *hint Y*

Exercise 11.3 (more colors)**.** Prove that cycles cannot be colored with $O(\log^* n)$ colors in $o(\log^* n)$ rounds in the deterministic LOCAL model.

▷ *hint Z*

Exercise 11.4 (lying about n)**.** Show that if a bipartite locally verifiable labeling problem can be solved in $o(\log n)$ rounds in the deterministic PN-model in (d, δ)-biregular trees, then it can be solved in $O(1)$ rounds.

▷ *hint AA*

Exercise 11.5 (hardness of sinkless orientation)**.** Let Π denote the sinkless orientation problem as defined in Chapter 10. Prove the following.

(a) Sinkless orientation requires $\Omega(\log \log n)$ rounds in the randomized PN-model.

(b) Sinkless orientation requires $\Omega(\log \log n)$ rounds in the deterministic and randomized LOCAL model.

▷ *hint AB*

⋆ **Exercise 11.6.** Show that sinkless orientation requires $\Omega(\log n)$ rounds in the deterministic LOCAL model.

▷ *hint AC*

11.6 Bibliographic Notes

Linial [30] showed that 3-coloring cycles with a deterministic algorithm is not possible in $o(\log^* n)$ rounds, and Naor [32] proved the same lower bound for randomized algorithms. Our presentation uses the ideas from these classic proofs, but in the modern round elimination formalism.

Part V
Conclusions

Chapter 12

Conclusions

We have reached the end of this course. In this chapter we will review what we have learned, and we will also have a brief look at what else is there in the field of distributed algorithms. The exercises of this chapter form a small research project related to the distributed complexity of locally verifiable problems.

12.1 What Have We Learned?

By now, you have learned a new mindset—an entirely new way to think about computation. You can reason about distributed systems, which requires you to take into account many challenges that we do not encounter in basic courses on algorithms and data structures:

- Dealing with *unknown systems*: you can design algorithms that work correctly in any computer network, no matter how the computers are connected together, no matter how we choose the port numbers, and no matter how we choose the unique identifiers.
- Dealing with *partial information*: you can solve graph problems in sublinear time, so that each node only sees a small part of the network, and nevertheless the nodes produce outputs that are globally consistent.
- Dealing with *parallelism*: you can design highly parallelized algorithms, in which several nodes take steps simultaneously.

These skills are in no way specific to distributed algorithms—they play a key role also in many other areas of modern computer science. For example, dealing with unknown systems is necessary if we want to

design *fault-tolerant* algorithms, dealing with partial information is the key element in e.g. *online algorithms* and *streaming algorithms*, and parallelism is the cornerstone of any algorithm that makes the most out of modern *multicore CPUs*, *GPUs*, and *computing clusters*.

12.2 What Else Exists?

Distributed computing is a vast topic and so far we have merely scratched the surface. This course has focused on what is often known as *distributed graph algorithms* or *network algorithms*, and we have only focused on the most basic models of distributed graph algorithms. There are many questions related to distributed computing that we have not addressed at all; here are a few examples.

12.2.1 Distance vs. Bandwidth vs. Local Memory

Often we would like to understand computation in two different kinds of distributed systems:

(a) *Geographically distributed networks*, e.g. the Internet. A key challenge is large distances and communication latency: some parts of the input are physically far away from you, so in a fast algorithm you have to act based on the information that is available in your local neighborhood.

(b) *Big data systems*, e.g. data processing in large data centers. Typically all input data is nearby (e.g. in the same local-area network). However, this does not make problems trivial to solve fast: individual computers have a limited *bandwidth* and limited amount of *local memory*.

The LOCAL model is well-suited for understanding computation in networks, but it does not make much sense in the study of big data systems: if all information is available within one hop, then in LOCAL model it would imply that everything can be solved in one communication round!

Congested Clique Model. Many other models have been developed to study big data systems. From our perspective, perhaps the easiest to understand is the *congested clique* model [17, 28, 31]. In brief, the model is defined as follows:

- We work in the CONGEST model, as defined in Chapter 5.
- We assume that the underlying graph G is the complete graph on n nodes, i.e., an n-clique. That is, every node is within one hop from everyone else.

Here it would not make much sense to study graph problems related to G itself, as the graph is fixed. However, here each node v gets some local input $f(v)$, and it has to produce some local output $g(v)$. The local inputs may encode e.g. some input graph $H \neq G$ (for example, $f(v)$ indicates which nodes are adjacent to v in H), but here it also makes perfect sense to study other computational problems that are not related to graphs. Consider, for example, the task of computing the matrix product $X = AB$ of two $n \times n$ matrices A and B. Here we may assume that initially each node knows one column of A and one column of B, and when the algorithm stops, each node has to hold one column of X.

Other Big Data Models. There are many other models of big data algorithms that have similar definitions—all input data is nearby, but communication bandwidth and/or local memory is bounded; examples include:

- *BSP model* (bulk-synchronous parallel) [40],
- *MPC model* (massively parallel computation) [25], and
- *k-machine model* [26].

Note that when we limit the amount of local memory, we also implicitly limit communication bandwidth (you can only send what you have in your memory). Conversely, if you have limited communication bandwidth and a fast algorithm, you do not have time to accumulate a large

amount of data in your local memory, even if it was unbounded. Hence all of these models and their variants often lead to similar algorithm design challenges.

12.2.2 Asynchronous and Fault-Tolerant Algorithms

So far in this course we have assumed that all nodes start at the same time, computation proceeds in a synchronous manner, and all nodes always work correctly.

Synchronization. If we do not have any failures, it turns out we can easily adapt all of the algorithms that we have covered in this course also to asynchronous settings. Here is an example of a very simple solution, known as the *α-synchronizer* [5]: all messages contain a piece of information indicating "this is my message for round i", and each node first waits until it has received all messages for round i from its neighbors before it processes the messages and switches to round $i+1$.

Crash Faults and Byzantine Faults. Synchronizers no longer work if nodes can fail. If we do not have any bounds on the relative speeds of the communication channels, it becomes impossible to distinguish between e.g. a node behind a very slow link and a node that has *crashed*.

Failures are challenging even if we work in a synchronous setting. In a synchronous setting it is easy to tell if a nodes has crashed, but if some nodes can misbehave in an arbitrary manner, many seemingly simple tasks become very difficult to solve. The term *Byzantine failure* is commonly used to refer to a node that may misbehave in an arbitrary manner, and in the quiz (Section 12.3) we will explore some challenges of solving the *consensus problem* in networks with Byzantine nodes.

Self-Stabilization. Another challenge is related to consistent initialization. In our model of computing, we have assumed that all nodes are initialized by using the $\text{init}_{A,d}$ function. However, it would be great to have algorithms that converge to a correct output even if the initial

states of the nodes may have been corrupted in an arbitrary manner. Such algorithms are called *self-stabilizing* algorithms [18].

If we have a deterministic T-time algorithms A designed in the LOCAL model, it can be turned into a self-stabilizing algorithm in a mechanical manner [29]: all nodes keep track of T possible states, indicating "what would be my state if now was round i", they send vectors of T messages, indicating "what would be my message for round i", and they repeatedly update their states according to the messages that they receive from their neighbors. However, if we tried to do something similar for a randomized algorithm, it would no longer converge to a fixed output—there are problems that are easy to solve with randomized Monte Carlo LOCAL algorithms, but difficult to solve with self-stabilizing algorithms.

12.2.3 Other Directions

Shared Memory. Our models of computing can be seen as a *message-passing system*: nodes send messages (data packets) to each other. A commonly studied alternative is a system with *shared memory*: each node has a shared register, and the nodes can communicate with each other by reading and writing the shared registers.

Physical Models. Our models of computing are a good match with systems in which computers are connected to each other by physical wires. If we connect the nodes by wireless links, the *physical properties of radio waves* (e.g., reflection, refraction, multipath propagation, attenuation, interference, and noise) give rise to new models and new algorithmic challenges. The *physical locations* of the nodes as well as the properties of the environment become relevant.

Robot Navigation. In our model, the nodes are active computational entities, and they cannot move around in the network—they can only send information around in the network. Another possibility is to study computation with autonomous agents ("robots") that can move around in the network. Typically, the nodes are passive entities (corresponding

to possible physical locations), and the robots can communicate with each other by e.g. leaving some tokens in the nodes.

Nondeterministic Algorithms. Just like we can study nondeterministic Turing machines, we can study nondeterministic distributed algorithms. In this setting, it is sufficient that there exists a *certificate* that can be verified efficiently in a distributed setting; we do not need to construct the certificate efficiently. Locally verifiable problems that we have studied in this course are examples of problems that are easy to solve with nondeterministic algorithms.

Complexity Measures. For us the main complexity measure has been the number of synchronous communication rounds. Many other possibilities exist: e.g., how many messages do we need to send in total?

Practical Aspects of Networking. This course has focused on the theory of distributed algorithms. There is of course also the practical side: We need physical computers to run our algorithms, and we need networking hardware to transmit information between computers. We need modulation techniques, communication protocols, and standardization to make things work together, and good software engineering practices, programming languages, and reusable libraries to keep the task of implementing algorithms manageable. In the real world, we will also need to worry about privacy and security. There is plenty of room for research in computer science, telecommunications engineering, and electrical engineering in all of these areas.

12.2.4 Research in Distributed Algorithms

There are two main conferences related to the theory of distributed computing:

- PODC, the ACM Symposium on Principles of Distributed Computing: https://www.podc.org/

- DISC, the International Symposium on Distributed Computing: http://www.disc-conference.org/

The proceedings of the recent editions of these conferences provide a good overview of the state-of-the-art of this research area.

12.3 Quiz

In the *binary consensus problem* the task is this: Each node gets 0 or 1 as input, and each node has to produce 0 or 1 as output. All outputs must be the same: you either have to produce all-0 or all-1 as output. Moreover, if the input is all-0, your output has to be all-0, and if the input is all-1, your output has to be all-1. For mixed inputs either output is fine.

Your network is a complete graph on 5 nodes; we work in the usual LOCAL model. You are using the following 1-round algorithm:

- Each node sends its input to all other nodes. This way each node knows all inputs.
- Each node applies the majority rule: if at least 3 of the 5 inputs are 1s, output 1, otherwise output 0.

This algorithm clearly solves consensus if all nodes correctly follow this algorithm. Now assume that node 5 is controlled by a *Byzantine adversary*, while nodes 1–4 correctly follow this algorithm. Show that now this algorithm *fails* to solve consensus among the correct nodes 1–4, i.e., there is some input so that the adversary can force nodes 1–4 to produce *inconsistent* outputs (at least one of them will output 0 and at least one of them will output 1).

Your answer should give the inputs of nodes 1–4 (4 bits), the messages sent by node 5 to nodes 1–4 (4 bits), and the outputs of nodes 1–4 (4 bits). An answer with these three bit strings is sufficient.

12.4 Exercises

In Exercises 12.1–12.4 we use the tools that we have learned in this course to study *locally verifiable problems* in cycles in the LOCAL model (both deterministic and randomized). For the purposes of these exercises, we use the following definitions:

A locally verifiable problem $\Pi = (\Sigma, \mathbf{C})$ consists of a finite alphabet Σ and a set $\mathbf{C}$ of allowed *configurations* (x, y, z): $x, y, z \in \Sigma$. An assignment $\varphi : V \to \Sigma$ is a *solution* to Π if and only if for each node u and its two neighbors v, w it holds that

$$\big(\varphi(v), \varphi(u), \varphi(w)\big) \in \mathbf{C} \quad \text{or} \quad \big(\varphi(w), \varphi(u), \varphi(v)\big) \in \mathbf{C}.$$

Put otherwise, if we look at any three consecutive labels x, y, z in the cycle, either (x, y, z) or (z, y, x) has to be an allowed configuration.

You can assume in Exercises 12.1–12.4 that the value of n is known (but please then make the same assumption consistently throughout the exercises, both for positive and negative results).

Exercise 12.1 (trivial and non-trivial problems)**.** We say that a locally verifiable problem $\Pi_0 = (\Sigma_0, \mathbf{C}_0)$ is *trivial*, if $(x, x, x) \in \mathbf{C}_0$ for some $x \in \Sigma_0$. We define that *weak c-coloring* is the problem $\Pi_1 = (\Sigma_1, \mathbf{C}_1)$ with

$$\Sigma_1 = \{1, 2, \ldots, c\},$$
$$\mathbf{C}_1 = \big\{(x_1, x_2, x_3) \;\big|\; x_1 \neq x_2 \text{ or } x_3 \neq x_2\big\}.$$

That is, each node must have at least one neighbor with a different color.

(a) Show that if a problem is trivial, then it can be solved in constant time.

(b) Show that if a problem is not trivial, then it is at least as hard as weak c-coloring for some c.

▹ *hint AD*

Exercise 12.2 (hardness of weak coloring)**.** Consider the weak c-coloring problem, as defined in Exercise 12.1.

(a) Show that weak 2-coloring can be solved in cycles in $O(\log^* n)$ rounds.

(b) Show that weak c-coloring, for any $c = O(1)$, requires $\Omega(\log^* n)$ rounds in cycles.

▷ *hint AE*

What does this imply about the possible complexities of locally verifiable problems in cycles?

Exercise 12.3 (randomized constant time)**.** Show that if a locally verifiable problem Π can be solved in constant time in cycles with a randomized LOCAL-algorithm, then it can be solved in constant time with a deterministic LOCAL-algorithm.

▷ *hint AF*

Exercise 12.4 (deterministic speed up)**.** Assume that there is a deterministic algorithm A for solving problem Π in cycles with a running time $T(n) = o(n)$. Show that there exists a deterministic algorithm A' for solving Π in $O(\log^* n)$ rounds.

What does this imply about the possible complexities of locally verifiable problems in cycles?

▷ *hint AG*

⋆⋆ **Exercise 12.5.** Prove or disprove: vertex coloring with $\Delta+1$ colors in graphs of maximum degree Δ can be solved in $O(\log \Delta + \log^* n)$ rounds in the LOCAL model.

▷ *hint AH*

12.5 Bibliographic Notes

The exercises in this chapter are inspired by Chang and Pettie [13].

Hints

A. Use the local maxima and minima to partition the path in subpaths so that within each subpath we have unique identifiers given in an increasing order. Use this ordering to orient each subpath. Then we can apply the fast color reduction algorithm in each subpath. Finally, combine the solutions.

B. Design a randomized algorithm that finds a coloring with a large number of colors quickly. Then apply the technique of the fast 3-coloring algorithm from Section 1.4 to reduce the number of colors to 3 quickly.

C. Consider the following cases separately:

$$\text{(i) } \log^* x \le 2, \quad \text{(ii) } \log^* x = 3, \quad \text{(iii) } \log^* x \ge 4.$$

In case (iii), prove that after $\log^*(x) - 3$ iterations, the number of colors is at most 64.

D. One possible strategy is this: Choose some threshold, e.g., $d = 10$. Focus on the nodes that have identifiers smaller than d, and find a proper 3-coloring in those parts, in time $O(\log^* d)$. Remove the nodes that are properly colored. Then increase threshold d, and repeat. Be careful with the way in which you increase d. Show that you can achieve a running time of $O(\log^* x)$, where x is the largest identifier, without knowing x in advance.

E. Assume that D is an edge dominating set; show that you can construct a maximal matching M with $|M| \le |D|$.

F. For the purposes of the minimum vertex cover algorithm, it is sufficient to know which nodes are matched in the bipartite maximal

matching algorithm—we do not need to know with whom they are matched.

G. This exercise is not trivial. If T_1 was a constant function $T_1(n) = c$, we could simply run A_1, and then start A_2 at time c, using the output of A_1 as the input of A_2. However, if T_1 is an arbitrary function of $|V|$, this strategy is not possible—we do not know in advance when A_1 will stop.

H. Solve small problem instances by brute force and focus on the case of long cycles. In a long cycle, use a graph coloring algorithm to find a 3-coloring, and then use the 3-coloring to construct a maximal independent set. Observe that a maximal independent set partitions the cycle into short fragments (with 2–3 nodes in each fragment).

   Apply the same approach recursively: interpret each fragment as a "supernode" and partition the cycle that is formed by the supernodes into short fragments, etc. Eventually, you have partitioned the original cycle into *long* fragments, with dozens of nodes in each fragment.

   Find an optimal vertex cover within each fragment. Make sure that the solution is feasible near the boundaries, and prove that you are able to achieve the required approximation ratio.

I. Adapt the basic idea of the greedy color reduction algorithm—find local maxima and choose appropriate colors for them—but pay attention to the stopping conditions and low-degree nodes. One possible strategy is this: a node becomes active if its current color is a local maximum among those neighbors that have not yet stopped; once a node becomes active, it selects an appropriate color and stops.

J. Given a graph $G \in \mathscr{F}$, construct a virtual graph $G^2 = (V, E')$ as follows: $\{u, v\} \in E'$ if $u \neq v$ and $\mathrm{dist}_G(u, v) \leq 2$. Prove that the

maximum degree of G^2 is $O(\Delta^2)$. Simulate a fast graph coloring algorithm on G^2.

K. First, design (or look up) a greedy *centralized* algorithm achieves an approximation ratio of $O(\log \Delta)$ on $\mathscr{F}$. The following idea will work: repeatedly pick a node that *dominates* as many new nodes as possible—here a node $v \in V$ is said to dominate all nodes in $\text{ball}_G(v, 1)$. For more details, see a textbook on approximation algorithms, e.g., Vazirani [41].

   Second, show that you can *simulate* the centralized greedy algorithm in a distributed setting. Use the algorithm of Exercise 4.4 to construct a distance-2 coloring. Prove that the following strategy is a faithful simulation of the centralized greedy algorithm:

   - For each possible value $i = \Delta + 1, \Delta, \ldots, 2, 1$:
     - For each color $j = 1, 2, \ldots, O(\Delta^2)$:
       - Pick all nodes $v \in V$ that are of color j and that dominate i new nodes.

   The key observation is that if $u, v \in V$ are two distinct nodes of the same color, then the set of nodes dominated by u and the set of nodes dominated by v are disjoint. Hence it does not matter whether the greedy algorithm picks u before v or v before u, provided that both of them are equally good from the perspective of the number of new nodes that they dominate. Indeed, we can equally well pick both u and v simultaneously in parallel.

L. To reach a contradiction, assume that A is an algorithm that solves the problem. For each n, let $\mathscr{F}(n)$ consists of all graphs with the following properties: there are n nodes with unique identifiers $1, 2, \ldots, n$, the graph is connected, and the degree of node 1 is 1. Then compare the following two quantities as a function of n:

   (a) $f(n) =$ how many different graphs there are in family $\mathscr{F}(n)$.

(b) $g(n)$ = how many different message sequences node number 1 may receive during the execution of algorithm A if we run it on any graph $G \in \mathscr{F}(n)$.

Argue that for a sufficiently large n, we will have $f(n) > g(n)$. Then there are at least two different graphs $G_1, G_2 \in \mathscr{F}(n)$ such that node 1 receives the same information when we run A on either of these graphs.

M. Pick the labels randomly from a sufficiently large set; this takes 0 communication rounds.

N. Each node u picks a random number $f(u)$. Nodes that are local maxima with respect to the labeling f will join I.

O. For the last part, consider a complete graph with a sufficiently large number of nodes.

P. Each node chooses an output 0 or 1 uniformly at random and stops; this takes 0 communication rounds. To analyze the algorithm, prove that each edge is a cut edge with probability $1/2$.

Q. Use the randomized coloring algorithm.

R. Look up "Luby's algorithm".

S. (a) Apply the result of Exercise 2.8. (b) Find a 1-factor.

T. For the lower bound, use the result of Exercise 7.4c.

U. Show that if a 3-regular graph is homogeneous, then it has a 1-factor. Show that G does not have any 1-factor.

V. Argue using both covering maps and local neighborhoods. For $i = 1, 2$, construct a network N_i' and a covering map ϕ_i from N_i' to N_i. Let $v_i' \in \phi_i^{-1}(v_i)$. Show that v_1' and v_2' have isomorphic radius-2 neighborhoods; hence v_1' and v_2' produce the same output. Then use the covering maps to argue that v_1 and v_2 also produce

the same outputs. In the construction of N_1', you will need to eliminate the 3-cycle; otherwise v_1' and v_2' cannot have isomorphic neighborhoods.

W. Show that a 0-round algorithm consists of choosing one active configuration and assigning its labels to the ports. Show that for any way of assigning the outputs to the ports, there exists a port numbering such that the incident edges of a passive node are not labeled according to any passive configuration.

X. Decompose the graph into *layers* $(V_0, V_1, \ldots, V_L)$, where nodes in layer i have distance i to the closest leaf. Then iteratively solve the problem, starting from layer V_L: match all nodes in layer V_L, V_{L-1}, and so on.

Y. Show that a fast 5-coloring algorithm could be simulated on any path to 5-color it. Then turn a 5-coloring into a 3-coloring yielding an algorithm that contradicts the 3-coloring lower bound.

Z. Show that an $O(\log^* n)$-coloring could be used to color cycles fast, which contradicts the lower bound for 3-coloring. Note that our lower bound is for paths: why does it also apply to cycles?

AA. Show that we can run an algorithm for graphs of size n_0, for some constant n_0, on all networks of size $n \geq n_0$, and get a correct solution. In particular, show that for any $T(n) = o(\log n)$ and a sufficiently large n_0, networks on n_0 nodes are locally indistinguishable from networks on n nodes, for $n \geq n_0$, in $O(T(n))$ rounds.

AB. (a) Show that if $\mathrm{re}(\Pi)$ can be solved by a randomized PN-algorithm in T rounds with local failure probability q, then it can be solved in $T-1$ rounds with local failure probability $\mathrm{poly}(q)$. Analyze the failure probability over T iterations for $T = o(\log \log n)$ and show that it is $\omega(1)$.

AC. One approach is the following. Prove that any *deterministic* $o(\log n)$-time algorithm for sinkless orientation implies an $O(\log^* n)$-time deterministic algorithm for sinkless orientation. To prove this speedup, "lie" to the algorithm about the size of the graph. Take an algorithm A for $(3,3)$-biregular trees on n_0 nodes, for a sufficiently large constant n_0. On any network N of size $n > n_0$, compute a coloring of N that *locally* looks like an assignment of unique identifiers in a network of size n_0. Then simulate A given this identifier assignment to find a sinkless orientation.

AD. Show that if a problem is not trivial, then each configuration must use a distinct adjacent label.

AE. Give an algorithm for turning a weak c-coloring into a 3-coloring in $O(c)$ rounds.

AF. Use an argument to boost the failure probability of a constant-time randomized algorithm. A constant-time algorithm cannot depend on the size of the input. Consider a network N such that a randomized algorithm succeeds with probability $p < 1$. Boost this by considering the same algorithm on network N' that consists of many copies of N.

AG. Use a similar argument as in Exercise 11.4. In this case we want a speedup simulation in the LOCAL model, so we also need to simulate the identifiers. Color the network so that the colors look locally like unique identifiers. Then simulate algorithm A using the colors instead of real identifiers.

Since A runs in time $o(n)$, we can find a *constant* n_0 such that $T(n_0) \ll n_0$. On any network with $n > n_0$ nodes, find a coloring with n_0-colors such that two nodes with the same color have distance at least $2T(n_0)+3$. Show that this can be done in $O(\log^* n)$ rounds. Then run A on N, using the coloring instead of unique identifiers. Show that this simulation can be done in constant time. Show that this simulation is correct in every 1-neighborhood.

AH. This is an open research question.

Bibliography

[1] Ittai Abraham. Decentralized thoughts: The marvels of polynomials over a field, 2020. URL: https://decentralizedthoughts.github.io/2020-07-17-the-marvels-of-polynomials-over-a-field/.

[2] Robert B. Allan and Renu Laskar. On domination and independent domination numbers of a graph. *Discrete Mathematics*, 23(2):73–76, 1978. doi:10.1016/0012-365X(78)90105-X.

[3] Dana Angluin. Local and global properties in networks of processors. In *Proc. 12th Annual ACM Symposium on Theory of Computing (STOC 1980)*, 1980. doi:10.1145/800141.804655.

[4] Matti Åstrand, Valentin Polishchuk, Joel Rybicki, Jukka Suomela, and Jara Uitto. Local algorithms in (weakly) coloured graphs, 2010. arXiv:1002.0125.

[5] Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, 1985. doi:10.1145/4221.4227.

[6] Alkida Balliu, Sebastian Brandt, Yuval Efron, Juho Hirvonen, Yannic Maus, Dennis Olivetti, and Jukka Suomela. Classification of distributed binary labeling problems. In *Proc. 34th International Symposium on Distributed Computing (DISC 2020)*, 2020. arXiv:1911.13294, doi:10.4230/LIPIcs.DISC.2020.17.

[7] Leonid Barenboim and Michael Elkin. *Distributed Graph Coloring: Fundamentals and Recent Developments*. Morgan & Claypool, 2013. doi:10.2200/S00520ED1V01Y201307DCT011.

[8] Leonid Barenboim, Michael Elkin, and Uri Goldenberg. Locally-iterative distributed $(\Delta+1)$-coloring below Szegedy-Vishwanathan

barrier, and applications to self-stabilization and to restricted-bandwidth models. In *Proc. 37th ACM Symposium on Principles of Distributed Computing (PODC 2018)*, 2018. arXiv:1712.00285, doi:10.1145/3212734.3212769.

[9] John A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. North-Holland, 1976.

[10] Sebastian Brandt. An automatic speedup theorem for distributed problems. In *Proc. 38th ACM Symposium on Principles of Distributed Computing (PODC 2019)*, 2019. arXiv:1902.09958, doi:10.1145/3293611.3331611.

[11] Sebastian Brandt, Orr Fischer, Juho Hirvonen, Barbara Keller, Tuomo Lempiäinen, Joel Rybicki, Jukka Suomela, and Jara Uitto. A lower bound for the distributed Lovász local lemma. In *Proc. 48th ACM Symposium on Theory of Computing (STOC 2016)*, 2016. arXiv:1511.00900, doi:10.1145/2897518.2897570.

[12] Yi-Jun Chang, Tsvi Kopelowitz, and Seth Pettie. An exponential separation between randomized and deterministic complexity in the LOCAL model. In *Proc. 57th IEEE Symposium on Foundations of Computer Science (FOCS 2016)*, pages 615–624. IEEE, 2016. arXiv:1602.08166, doi:10.1109/FOCS.2016.72.

[13] Yi-Jun Chang and Seth Pettie. A time hierarchy theorem for the local model. *SIAM Journal on Computing*, 48(1):33–69, 2019. arXiv:1704.06297, doi:10.1137/17M1157957.

[14] Pafnuty Chebyshev. Mémoire sur les nombres premiers. *Journal de mathématiques pures et appliquées*, 17(1):366–390, 1852.

[15] Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, 1986. doi:10.1016/S0019-9958(86)80023-7.

[16] Reinhard Diestel. *Graph Theory*. Springer, 3rd edition, 2005. URL: http://diestel-graph-theory.com/.

[17] Danny Dolev, Christoph Lenzen, and Shir Peled. "Tri, tri again": Finding triangles and small subgraphs in a distributed setting. In *Proc. 26th International Symposium on Distributed Computing (DISC 2012)*, 2012. arXiv:1201.6652, doi:10.1007/978-3-642-33651-5_14.

[18] Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.

[19] Roy Friedman and Alex Kogan. Deterministic dominating set construction in networks with bounded degree. In *Proc. 12th International Conference on Distributed Computing and Networking (ICDCN 2011)*, 2011. doi:10.1007/978-3-642-17679-1_6.

[20] Mohsen Ghaffari, Juho Hirvonen, Fabian Kuhn, Yannic Maus, Jukka Suomela, and Jara Uitto. Improved distributed degree splitting and edge coloring. *Distributed Computing*, 33:293–310, 2020. arXiv:1706.04746, doi:10.1007/s00446-018-00346-8.

[21] Mohsen Ghaffari and Hsin-Hao Su. Distributed degree splitting, edge coloring, and orientations. In *Proc. 28th ACM-SIAM Symposium on Discrete Algorithms (SODA 2017)*, 2017. arXiv:1608.03220, doi:10.1137/1.9781611974782.166.

[22] Andrew V. Goldberg, Serge A. Plotkin, and Gregory E. Shannon. Parallel symmetry-breaking in sparse graphs. *SIAM Journal on Discrete Mathematics*, 1(4):434–446, 1988. doi:10.1137/0401044.

[23] Michał Hańćkowiak, Michał Karoński, and Alessandro Panconesi. On the distributed complexity of computing maximal matchings. In *Proc. 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1998)*, 1998.

[24] Stephan Holzer and Roger Wattenhofer. Optimal distributed all pairs shortest paths and applications. In *Proc. 31st Annual ACM Symposium on Principles of Distributed Computing (PODC 2012)*, 2012. doi:10.1145/2332432.2332504.

[25] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for MapReduce. In *Proc. 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2010)*. 2010. doi:10.1137/1.9781611973075.76.

[26] Hartmut Klauck, Danupon Nanongkai, Gopal Pandurangan, and Peter Robinson. Distributed computation of large-scale graph problems. In *Proc. 26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2015)*, 2015. doi:10.1137/1.9781611973730.28.

[27] Amos Korman, Jean-Sébastien Sereni, and Laurent Viennot. Toward more localized local algorithms: removing assumptions concerning global knowledge. In *Proc. 30th Annual ACM Symposium on Principles of Distributed Computing (PODC 2011)*, 2011. doi:10.1145/1993806.1993814.

[28] Christoph Lenzen. Optimal deterministic routing and sorting on the congested clique. In *Proc. 32nd Annual ACM symposium on Principles of Distributed Computing (PODC 2013)*, 2013. arXiv:1207.1852, doi:10.1145/2484239.2501983.

[29] Christoph Lenzen, Jukka Suomela, and Roger Wattenhofer. Local algorithms: self-stabilization on speed. In *Proc. 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2009)*, 2009. doi:10.1007/978-3-642-05118-0_2.

[30] Nathan Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992. doi:10.1137/0221015.

[31] Zvi Lotker, Boaz Patt-Shamir, Elan Pavlov, and David Peleg. Minimum-weight spanning tree construction in $O(\log \log n)$ communication rounds. *SIAM Journal on Computing*, 35(1):120–131, 2005. doi:10.1137/S0097539704441848.

[32] Moni Naor. A lower bound on probabilistic algorithms for distributive ring coloring. *SIAM Journal on Discrete Mathematics*, 4(3):409–412, 1991. doi:10.1137/0404036.

[33] Moni Naor and Larry Stockmeyer. What can be computed locally? *SIAM Journal on Computing*, 24(6):1259–1277, 1995. doi:10.1137/S0097539793254571.

[34] Dennis Olivetti. Round Eliminator: a tool for automatic speedup simulation, 2020. URL: https://github.com/olidennis/round-eliminator.

[35] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications, Inc., 1998.

[36] David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. SIAM Monographs on Discrete Mathematics and Applications. Society for Industrial and Applied Mathematics, 2000.

[37] Julius Petersen. Die Theorie der regulären graphs. *Acta Mathematica*, 15(1):193–220, 1891. doi:10.1007/BF02392606.

[38] Valentin Polishchuk and Jukka Suomela. A simple local 3-approximation algorithm for vertex cover. *Information Processing Letters*, 109(12):642–645, 2009. arXiv:0810.2175, doi:10.1016/j.ipl.2009.02.017.

[39] Jukka Suomela. Distributed algorithms for edge dominating sets. In *Proc. 29th Annual ACM Symposium on Principles of Distributed Computing (PODC 2010)*, pages 365–374, 2010. doi:10.1145/1835698.1835783.

[40] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990. doi:10.1145/79173.79181.

[41] Vijay V. Vazirani. *Approximation Algorithms*. Springer, 2001.

[42] Masafumi Yamashita and Tsunehiko Kameda. Computing on anonymous networks: part I—characterizing the solvable cases. *IEEE*

Transactions on Parallel and Distributed Systems, 7(1):69–89, 1996. doi:10.1109/71.481599.

[43] Mihalis Yannakakis and Fanica Gavril. Edge dominating sets in graphs. *SIAM Journal on Applied Mathematics*, 38(3):364–372, 1980. doi:10.1137/0138030.