Distributed Algorithms

Jukka Suomela

Aalto University, Finland

2 January 2021

https://jukkasuomela.fi/da/

An extended and revised version of this book is now available online:

https://jukkasuomela.fi/da2020/

Contents

Part II Graphs

Part III Models of Computing

Foreword

This book is an introduction to the theory of distributed 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 it?
- **Computability and computational complexity:** which computational problems *cannot* be solved at all with distributed algorithms, which problems cannot be solved efficiently, and why is this the case?

No prior knowledge of distributed systems is needed. A basic knowledge of discrete mathematics and graph theory is assumed, as well as familiarity with the basic concepts from undergraduate-level courses on models on 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 ETCS credits. There are 12 weeks of lectures: each week there is one 2-hour lecture and one 2-hour exercise session. Each week we will cover one chapter of this book. In each chapter there are at least 5 exercises, and the students are expected to solve at least 3 of them.

Prior versions of this textbook have been used in the lecture course of *Deterministic Distributed Algorithms*, which I lectured at the University of Helsinki in 2010, 2012, and 2014.

Acknowledgements

Many thanks to Jaakko Alasaari, Alkida Balliu, Mika Göös, Juho Hirvonen, Joel Kaasinen, Mirco Kroon, Teemu Kuusisto, Dang Lam, Tuomo Lempiäinen, Christoph Lenzen, Abdulmelik Mohammed, Christopher Purcell, Joel Rybicki, Stefan Schmid, Roelant Stegmann, Przemysław Uznański, and Jussi Väisänen for feedback, discussions, comments, and for helping me with the arrangements of this course. This work was supported in part by the Academy of Finland, Grant 252018. For updates and additional material, see

https://jukkasuomela.fi/da/

License

Mathematical Preliminaries

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

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,

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

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,

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

Part I

Informal Introduction

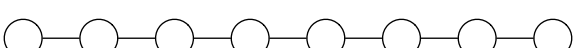
Chapter 1

Warm-Up — Positive Results

We will start this book with an informal introduction to distributed algorithms. We will formalise the model of computing later, starting with some graph-theoretic preliminaries in Chapter 3, and then followed by the definitions of three models of distributed computing in Chapters 4–6. However, in the first two chapters the intuitive idea of computers that can exchange messages with each others is sufficient.

1.1 Running Example: Colouring Paths

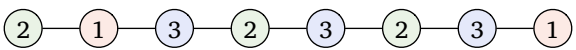
Imagine that we have n computers (or *nodes* as they are usually called) that are connected to each other with communication channels so that the network topology is a *path*:



The computers can exchange messages with their neighbours. 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.

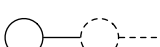
In this example, the task is to find a proper *colouring* of the path with 3 colours. That is, each node has to output one of the colours, 1, 2, or 3, so that neighbours have different colours — here is an example of a proper solution:



1.2 Challenges of Distributed Algorithm

With a bird's-eye view of the entire network, colouring a path looks like a very simple task: just start from one endpoint and assign colours 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.

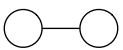
Indeed, when we start a networked computer, it is typically only aware of itself and the communication channels that it can use. In our simple example, the endpoints of the path know that they have one neighbour:



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

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?

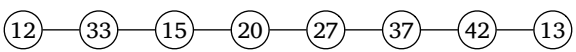
The answer is that it is not possible, without some additional assumptions. In practice, we could try to rely on some real-world imperfections (e.g., the computers are seldom perfectly synchronised), but in the theory of distributed algorithms we often assume that there is some *explicit* way to *break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

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

In subsequent chapters we will then formalise 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 (the model with unique names will be discussed in detail in Chapter 5, and randomised algorithms will be discussed in Chapter 7).

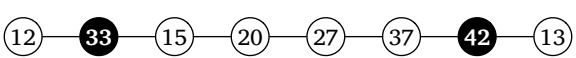
1.3 Colouring with Unique Identifiers

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:

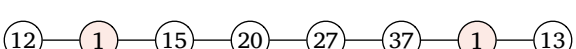


With the help of unique identifiers, it is now easy to design an algorithm that colours a path. Indeed, the unique identifiers already form a colouring with a large number of colours! All that we need to do is to reduce the number of colours to 3.

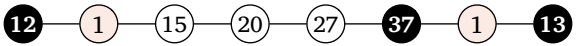
We can use the following simple strategy. In each step, a node is active if it is a "local maximum", i.e., its current colour is larger than the current colours of its neighbours:



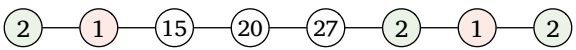
The active nodes will then pick a new colour from the colour palette $\{1, 2, 3\}$, so that it does not conflict with the current colours of their neighbours. This is always possible, as each node in a path has at most 2 neighbours, and we have 3 colours in our colour palette:



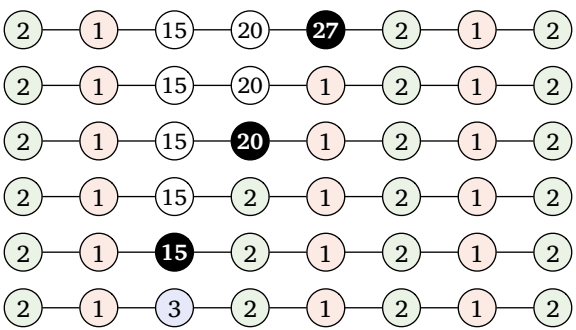
Then we simply repeat the same procedure until all nodes have small colours. First find the local maxima:



And then recolour the local maxima with colours from $\{1, 2, 3\}$:



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



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

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?

Repeat forever:

- Send message c to all neighbours.
- Receive messages from all neighbours.
  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: Algorithm P3C: 3-colouring a path.

Let us fix some notation. Each node maintains a variable c that contains its current colour. Initially, c is equal to the unique identifier of the node. Then computation proceeds as shown in Table 1.1; we will call this algorithm P3C.

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 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.

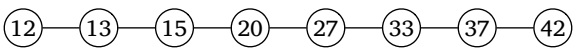
Our algorithm may look a bit strange in the sense that computers that have "stopped" are still sending messages. However, it is fairly straightforward to rewrite the algorithm so that you could actually turn off computers that have stopped. The basic idea is that nodes that are going to switch to a stopping state first inform their neighbours about this. Each node will memorise which of its neighbours have already stopped and what where their final colours. Implementing this idea is left as Exercise 1.2, and you will later see in Chapter 4 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 neighbours, receive messages from their neighbours, and update 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, its 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.

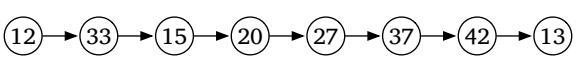
1.4 Faster Colouring with Unique Identifiers

So far we have seen that with the help of unique identifiers, it is *possible* to find a 3-colouring 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 colour 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 generalise 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 labelled 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 generalised to arbitrarily large identifiers if needed.

Again, we will interpret the unique identifiers as colours; hence our starting point is a path that is properly coloured with 2^{128} colours. In the next section, we will present algorithm called P3CBit that reduces the number of colours from 2^x to $2x$ in one round, for any positive integer x. Hence in one step we can reduce the number of colours from 2^{128} to $2 \cdot 128 = 256$. In just four iterations we can reduce the number of colours 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-colouring, we can then apply the algorithm of Table 1.1 to reduce the number of colours 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 colours from 2^{128} to 3 in only 7 rounds — no matter how many nodes we have in the path. Compare this with algorithm P3C, which may take millions of rounds for paths with millions of nodes.

1.4.2 Algorithm for One Step

Let us now show how algorithm P3CBit reduces the number of colours from 2^x to $2x$ in one round; as the name suggests, we will be doing some bit manipulations here. First, each node

sends its current colour to its predecessor. After this step, each node u knows two values:

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

If a node does not have any successor, it just proceeds *as if* it had a successor of some colour 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 colour of node u is different from the current colour 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 colour.

1.4.3 An Example

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

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

node u	input $c_0(u)$	$c_1(u)$	$i(u)$	$b(u)$	output $c(u)$
⋯	⋯	⋯	⋯	⋯	⋯
↓					
○	$01111\mathbf{0}11_2$	$00101\mathbf{1}11_2$	2	0	4
↓					
○	$00101\mathbf{1}11_2$	$0110\mathbf{1}011_2$	2	1	5
↓					
○	01101011_2	⋯	⋯	⋯	⋯
↓					
⋯	⋯				
⋯	⋯	⋯	⋯	⋯	⋯
↓					
○	$01111\mathbf{0}11_2$	$00101\mathbf{1}11_2$	2	0	4
↓					
○	$0\mathbf{0}101111_2$	$0\mathbf{1}101111_2$	6	0	12
↓					
○	01101111_2	⋯	⋯	⋯	⋯
↓					
⋯	⋯				

Table 1.2: Algorithm P3CBit: reducing the number of colours from 2^x to $2x$, for $x = 8$. There are two interesting cases: either $i(u)$ is the same for two neighbours (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 colours $c(u)$ will be different.

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

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

That is, node picks 4 as its new colour. 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 colour 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 $2x$-colouring 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 $2x$-colouring in one round.

1.4.5 Iteration

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

If we iterate the algorithm, we can reduce the number of colours from x to 6 in $O(\log^* x)$ rounds, after which we can use algorithm P3C from Section 1.3 to reduce the number of colours from 6 to 3 in 3 rounds — the details of the analysis are left as Exercises 1.5 and 1.6.

1.5 Colouring with Randomised Algorithms

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

1.5.1 Algorithm

Let us formalise the algorithm that we sketched above; we will call this algorithm P3CRand. 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 colour. 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 colour $c(u) \in \{1, 2, 3\}$ uniformly at random. Then each node sends its current colour $c(u)$ to its neighbours. If $c(u)$ is different from the colours of its neighbours, 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 neighbours do, there is at least one choice for $c(u) \in \{1, 2, 3\}$ that does not conflict with its neighbours.

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 summarise 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-colouring 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-colouring paths:

- Algorithm P3C, Section 1.3: A deterministic algorithm for paths with unique unique identifiers. Runs in $O(n)$ rounds, where n is the number of nodes.
- Algorithm P3CBit, Section 1.4: A deterministic algorithm for *directed* paths with unique unique identifiers. Runs in $O(\log^* x)$ rounds, where x is the largest identifier.

- Algorithm P3CRand, Section 1.5: A randomised algorithm for paths without unique identifiers. Runs in $O(\log n)$ rounds with high probability.

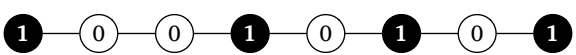
We will explore and analyse these algorithms and their variants in more depth in the exercises.

1.7 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 neighbours are in I,
- for each node $v \notin I$, at least one of its neighbours is in I.

Here is an example — the nodes labelled 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 randomised 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)**.** Algorithm P3CBit finds a 3-colouring 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.

▷ *hint A*

Exercise 1.4 (randomised and fast)**.** Algorithm P3CRand finds a 3-colouring in time $O(\log n)$ with high probability, and it does not need any unique identifiers. Can you design a randomised algorithm that finds a 3-colouring in time $o(\log n)$ with high probability? You can assume that n is known.

▷ *hint B*

Exercise 1.5 (asymptotic analysis)**.** Analyse algorithm P3CBit:

(a) Assume that we are given a colouring with x colours. Show that we can find a 3-colouring 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-colouring in time $O(\log^* n)$.

⋆ **Exercise 1.6** (tight analysis)**.** Analyse algorithm P3CBit: Assume that we are given a colouring with x colours, for any integer $x \geq 6$. Show that we can find a 6-colouring in time $\log^*(x)$, and therefore a 3-colouring in time $\log^*(x) + 3$.

▷ *hint C*

⋆ **Exercise 1.7** (oblivious algorithms)**.** Algorithm P3C 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 algorithm P3CBit so that it is also oblivious.

▷ *hint D*

1.8 Bibliographic Notes

Algorithm P3CBit was originally presented by Cole and Vishkin [6] and further refined by Goldberg et al. [10]; in the literature, it is commonly known as the "Cole–Vishkin algorithm". Exercise 1.7 was inspired by Korman et al. [13].

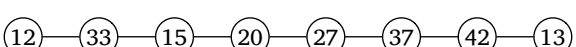
Chapter 2

Warm-Up — Negative Results

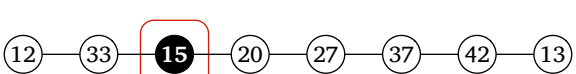
The defining property of fast distributed algorithms is *locality*: if we run a distributed algorithm for t time steps, the nodes can only be aware of the information that is available within distance at most t from them. In this chapter we will see why this is the case, and what consequences it has.

2.1 Locality

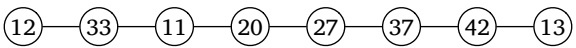
Locality is easiest to understand through an example. Consider the following network, familiar from the previous section:



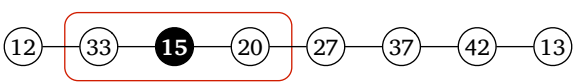
Let us focus on the node number 15. Initially, there is only one node in the network that is aware of the existence of such a node — the node itself. Let us highlight the set of nodes that are aware of node 15 at time $\boldsymbol{t = 0}$:



All other nodes are completely unaware of the existence of node number 15. For example, for all that they know, we might equally well have the following instance, in which we do not have any node with identifier 15:

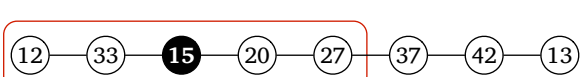


Now let us consider what happens at time $t = 1$, after one communication round. In this round, all nodes can exchange messages with their neighbours, simultaneously in parallel. Nodes can send anything that they know to their neighbours. In particular, node 15 can inform its neighbours about its existence, so after one round, its neighbours 33 and 20 may also be aware of it:

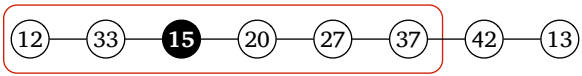


However, the crucial observation is that only these three nodes can be aware of the existence of node 15. For example, consider node 27. Before the first round, this node and its neighbours were unaware of node 15; hence during the first round node 27 could not have learned anything about node 15 from any of its neighbours.

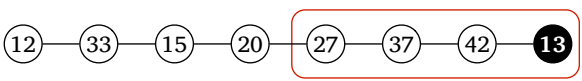
By a similar reasoning, at time $t = 2$, after two communication rounds, the set of nodes that may be aware of node 15 consists of precisely those nodes that are within distance $t = 2$ from it:



And at time $t = 3$ this information may have propagated up to distance $t = 3$, but not any further:



Of course the same reasoning holds for any node, and for any information related to the node. For example, here is a picture that shows the nodes that are within distance 3 from node 13:



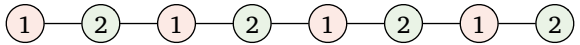
At time $t = 3$, precisely these nodes can be aware of the existence of node 13, and precisely these nodes can know that node 13 is a node of degree 1, i.e., it has got only one neighbour.

Naturally, if a node stops after time t, whatever output it produces can only depend on what it knows, and as we have seen, a node can only know information that is available at distance t. This is the crux of locality in distributed computing:

- time and distance are interchangeable,
- in a *fast* algorithm, nodes have to make decisions based on the information that is available *near* them.

2.2 Locality and 2-Colouring

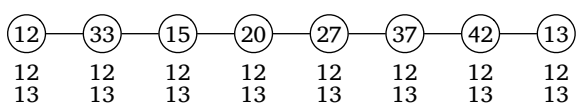
Recall from Chapter 1 that there are very fast algorithms for 3-colouring paths. However, there is no need to settle for 3 colours — a path can be always coloured with 2 colours:



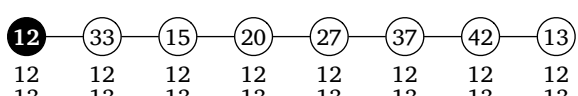
2.2.1 Algorithm for 2-Colouring Paths

With some thought, we can also come up with a distributed algorithm that finds a 2-colouring of a path with n nodes in time $O(n)$. An algorithm that works along these lines should do the trick; let us call this algorithm P2C:

- First, the endpoints of the path (i.e., nodes of degree 1) send their identifiers to their neighbours. Other nodes forward this information until all nodes along the path learn the identifiers of the endpoints. This takes $n-1$ communication rounds.

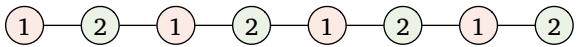


- Now the endpoints know each other's identifiers. We elect the endpoint with the smaller identifier as the *leader*.



- Finally, the leader colours itself with colour 1, sends its colour to its neighbour, and stops. The neighbour responds by picking colour 2, etc.; after $n-1$ rounds, we

have coloured all nodes with alternating colours 1 and 2, and all nodes have stopped.



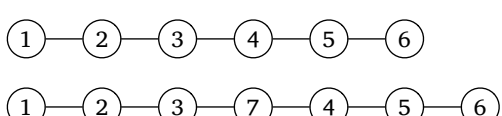
However, in comparison with algorithm P3CBit from Section 1.4, this is very slow. Hence we can ask the following question: is it really necessary to spend $\Omega(n)$ rounds in order to find a 2-colouring of a path?

2.2.2 Lower Bound for 2-Colouring

To reach a contradiction, suppose that there is a deterministic algorithm A that runs in time $o(n)$. In particular, there is 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, numbered $1, 2, \ldots, 2k$, and path H contains $2k+1$ nodes, numbered

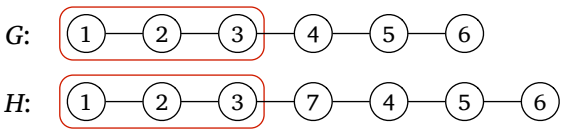
$$1, 2, \ldots, k, 2k+1, k+1, k+2, \ldots, 2k.$$

Here is an example for $k = 3$:

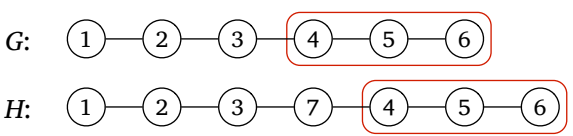


By assumption, the running time t is at most $k-1$ rounds in both cases. In particular, node number 1 is only aware of the first

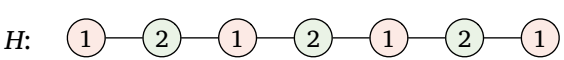
k nodes along the path, and it must produce its output based on what it sees. As what it sees is the same in G and H, we conclude that node 1 picks the same colour in both instances:



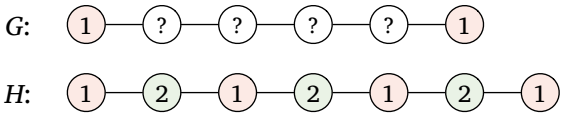
By a similar reasoning, node $2k$ (i.e., the last node of the path) has the same neighbourhood up to distance t, and therefore it also has to produces the same output in both cases:



However, now we reach a contradiction. In path H, in any proper 2-colouring nodes 1 and $2k$ have the same colour — for example, both of them are of colour 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-colouring of G.



We conclude that algorithm A fails to find a proper 2-colouring in at least one of these instances.

In summary, we have shown that there is no deterministic algorithm that finds a 2-colouring in time $o(n)$, even if the algorithm can use unique identifiers. On the other hand, there is a deterministic algorithm that solves the problem in time $O(n)$; we conclude that the distributed computational complexity of 2-colouring paths is precisely $\Theta(n)$.

While we have focused on deterministic algorithms here, we can use similar ideas to prove an analogous result for randomised algorithms, too — this is left as an exercise.

2.3 Locality and 3-Colouring

In the previous section we saw that 2-colouring paths with distributed algorithms takes $\Theta(n)$ rounds. In Chapter 1 we saw that 3-colouring is possible much faster. Let us now study precisely how much faster it is.

For the sake of concreteness, we will consider the following case:

- we have a *directed* path with n nodes, so that each node has at most one successor and at most one predecessor,
- the unique identifiers are a permutation of $\{1, 2, \ldots, n\}$.

In this case algorithm P3CBit finds a 3-colouring in time $O(\log^* n)$. We will now show that this is optimal: any algorithm A that solves this problem requires $\Omega(\log^* n)$ rounds.

2.3.1 Proof Overview

Fix any positive integer n. We will prove the claim as follows.

(a) We define the following concept: "k-ary c-colouring function".

(b) We show that if A is a distributed algorithm that finds a 3-colouring in time T, then there exists a k-ary 3-colouring function for $k = 2T + 1$.

(c) We show that $k + 1 \geq \log^* n$ for any k-ary 3-colouring function.

Now it follows that

$$2T + 2 \geq \log^*(n),$$

or put otherwise,

$$T \geq \frac{1}{2} \log^*(n) - 1.$$

2.3.2 Colouring Functions

Let k and c be positive integers. We say that a function f is a k-ary c-colouring function if

$$f(x_1, x_2, \ldots, x_k) \in \{1, 2, \ldots, c\} \quad \text{for all } 1 \le x_1 < x_2 < \ldots < x_k \le n, \tag{2.1}$$

$$f(x_1, x_2, \ldots, x_k) \neq f(x_2, x_3, \ldots, x_{k+1}) \quad \text{for all } 1 \le x_1 < x_2 < \ldots < x_{k+1} \le n. \tag{2.2}$$

For example, here is a 2-ary 3-colouring function for $n = 5$:

$$\begin{aligned} f(1,2) = 1, \quad & f(1,3) = 2, \quad f(1,4) = 2, \quad f(1,5) = 2, \\ & f(2,3) = 2, \quad f(2,4) = 2, \quad f(2,5) = 2, \\ & \qquad\quad f(3,4) = 1, \quad f(3,5) = 1, \\ & \qquad\qquad\quad f(4,5) = 3. \end{aligned}$$

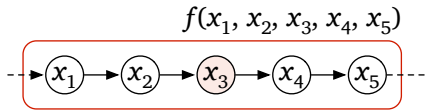
You can verify that this is indeed a colouring function; property (2.1) clearly holds, and property (2.2) can be verified by considering all cases:

$$\begin{aligned} & f(1,2) \neq f(2,3), \quad f(1,2) \neq f(2,4), \quad f(1,2) \neq f(2,5), \\ & f(1,3) \neq f(3,4), \quad f(1,3) \neq f(3,5), \quad f(1,4) \neq f(4,5), \\ & f(2,3) \neq f(3,4), \quad f(2,3) \neq f(3,5), \quad f(2,4) \neq f(4,5), \\ & f(3,4) \neq f(4,5). \end{aligned}$$

2.3.3 From Algorithms to Colouring Functions

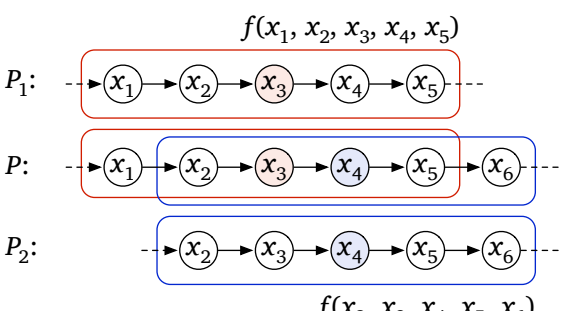
Now consider a distributed algorithm A that finds a 3-colouring in time T. Let $k = 2T + 1$. We will show how to use A to construct a k-ary 3-colouring function f.

To this end, let $1 \le x_1 < x_2 < \ldots < x_k \le n$. Construct a path in which we have k consecutive nodes with unique identifiers $x_1, x_2, \ldots, x_k$, in this order — here is an illustration for $T = 2$:



Then apply algorithm A to find a colouring of this path, and define that $f(x_1, x_2, \ldots, x_k)$ is the output of node x_{T+1}. Note that the output of x_{T+1} only depends on the identifiers $x_1, x_2, \ldots, x_k$, so this is well-defined: we will get the same output, regardless of how we choose the unique identifiers of the remaining $n - k$ nodes.

Now we need to argue that f is indeed a colouring function. Property (2.1) clearly holds. To verify property (2.2), let $1 \le x_1 < x_2 < \ldots < x_{k+1} \le n$. Consider a path P in which the identifiers are given in an increasing order — here is an illustration for $T = 2$:



We have defined function f so that

- $f(x_1, x_2, \ldots, x_k) =$ the output of node x_{T+1} in path P,
- $f(x_2, x_3, \ldots, x_{k+1}) =$ the output of node x_{T+2} in path P.

Here it is crucial that the output of a node only depends on its radius-T neighbourhood. Algorithm A finds a proper colouring of any path; therefore the output of x_{T+1} has to be different from the output of x_{T+2}. We conclude that

$$f(x_1, x_2, \ldots, x_k) \neq f(x_2, x_3, \ldots, x_{k+1}).$$

Function f is indeed a k-ary 3-colouring function.

2.3.4 Observations

We have seen that colouring functions are closely related to algorithms that colour paths. Before we continue, let us make the following observations:

- Given a distributed algorithm that finds a 3-colouring of a path in time T, we can construct a k-ary 3-colouring function for $k = 2T + 1$.
- The converse is not necessarily true. A colouring function only needs to colour properly path segments that have unique identifiers given in an increasing order, while an algorithm has to handle all kinds of paths (as well as all corner cases, such as nodes near the endpoints of a path).
- A distributed algorithm implies a k-ary colouring function for an odd k.

- Colouring functions are well-defined also for even values of k.

While we are interested in algorithms, it turns out that colouring functions are easier to analyse. It is sufficient to show that colouring functions for very small values of k do not exist — then it follows that algorithms for very small values of T do not exist, either.

2.3.5 Simple Base Case

We will now show that k-ary 3-colouring functions do not exist if k is too small. We start with a trivial lemma that shows that with $k = 1$ we cannot do much.

Lemma 2.1. *If f is a 1-ary c-colouring function, then we must have $c \geq n$.*

Proof. Note that a 1-ary c-colouring function is a mapping

$$f : \{1, 2, \ldots, n\} \to \{1, 2, \ldots, c\}.$$

If $c < n$, there are collisions: we can find some x_1 and x_2 with $x_1 < x_2$ and $f(x_1) = f(x_2)$, which contradicts property (2.2). □

2.3.6 Recursive Step

The key element of the proof is the following lemma. Informally, given any colouring function f, we can always construct another colouring function g that is "faster" (smaller number of arguments) but "worse" (larger number of colours).

Lemma 2.2. *If f is a k-ary c-colouring function, we can construct a $(k-1)$-ary 2^c-colouring function g.*

Proof. First, let h be a bijection from the subsets of $\{1,2,\ldots,c\}$ to the integers $\{1,2,\ldots,2^c\}$. For example,

$$h(\varnothing)=1,\ h(\{1\})=2,\ h(\{2\})=3,\ h(\{1,2\})=4,\ \ldots$$

Second, define function g' as follows:

$$g'(x_1,x_2,\ldots,x_{k-1})=\big\{f(x_1,x_2,\ldots,x_{k-1},x_k)\ :\ x_k>x_{k-1}\big\}.$$

Finally, let g be $h\circ g'$, that is,

$$g(x_1,x_2,\ldots,x_{k-1})=h(g'(x_1,x_2,\ldots,x_{k-1})).$$

We claim that this function g is indeed a $(k-1)$-ary 2^c-colouring function. Clearly it takes $k-1$ arguments and it satisfies property (2.1). The interesting part is property (2.2). Let $1\le x_1<x_2<\ldots<x_k\le n$. By way of contradiction, suppose that

$$g(x_1,x_2,\ldots,x_{k-1})=g(x_2,x_3,\ldots,x_k).$$

As h is a bijection, this implies

$$g'(x_1,x_2,\ldots,x_{k-1})=g'(x_2,x_3,\ldots,x_k). \tag{2.3}$$

Let $\alpha=f(x_1,x_2,\ldots,x_k)$. From the definition of g' we have

$$\alpha\in g'(x_1,x_2,\ldots,x_{k-1}).$$

By assumption (2.3), this implies

$$\alpha \in g'(x_2, x_3, \ldots, x_k).$$

But then we must have some $x_k < x_{k+1} \leq n$ such that

$$\alpha = f(x_2, x_3, \ldots, x_{k+1}).$$

However, we also had

$$\alpha = f(x_1, x_2, \ldots, x_k).$$

That is, f cannot be a colouring function. □

2.3.7 Completing the Proof

Assume that f_1 is a k-ary 3-colouring function. Certainly it is also a k-ary 4-colouring function, and $4 = {}^2 2$ (recall that we use the notation ${}^i 2$ for power towers). We can now apply Lemma 2.2 iteratively to obtain

- a $(k-1)$-ary ${}^3 2$-colouring function A_2,
- a $(k-2)$-ary ${}^4 2$-colouring function A_3, ...
- a 1-ary ${}^{k+1} 2$-colouring function A_k.

By Lemma 2.1, we must have ${}^{k+1} 2 \geq n$, which implies $k + 1 \geq \log^* n$.

This completes the proof. Recall that if A is a distributed algorithm that finds a 3-colouring of any path in time T, then there exists a k-ary 3-colouring function for $k = 2T + 1$. We have now shown that

$$k + 1 = 2T + 2 \geq \log^* n.$$

2.4 Exercises

Exercise 2.1 (counting)**.** Consider the following problem: counting the number of nodes in a path. That is, we are given a path with some unknown number of nodes. All nodes have to stop and output n, the number of nodes in the path.

(a) Design a deterministic distributed algorithm that solves the counting problem in time $O(n)$. You can assume that the nodes have unique identifiers.

(b) Prove that it is not possible to solve this problem in time $o(n)$.

Exercise 2.2 (known n)**.** In Section 2.2 we saw that 2-colouring a path with n nodes takes $\Omega(n)$ rounds. Show that the claim holds even if n is known. That is, all nodes are initially aware of their own identifier and of the exact number of nodes in the path.

Exercise 2.3 (randomised algorithms)**.** Show that there is no randomised distributed algorithm that finds a 2-colouring in time $o(n)$ with probability at least 0.9.

Exercise 2.4 (maximal independent sets)**.** Recall the definition of a maximal independent set from Exercise 1.1. Prove that it is not possible to find a maximal independent set with a deterministic algorithm in time $o(\log^* n)$. Show that this holds even if we have unique identifiers from set $\{1, 2, \ldots, n\}$.

Exercise 2.5 (large independent sets)**.** An independent set is a set of nodes I such that for each node $v \in I$, none of its

neighbours are in I. Consider a path with n nodes. Assume that we have unique identifiers that are bounded by some polynomial of n, that is, there is a constant c such that the unique identifiers are from $\{1, 2, \ldots, n^c\}$.

(a) Show that it is trivial to find some independent set in $O(1)$ time with a deterministic distributed algorithm.

(b) Show that there exists an independent set with at least $n/2$ nodes.

(c) Show that finding an independent set with at least $n/2$ nodes takes $\Theta(n)$ rounds.

(d) Design a deterministic distributed algorithm that finds an independent set with at least $n/10$ nodes in time $O(\log^* n)$, with the help of unique identifiers. You can assume that the identifiers are bounded by a polynomial in n.

(e) Design a randomised distributed algorithm that finds an independent set so that the *expected* number of nodes in the output is at least $n/10$ and the running time of the algorithm is $O(1)$.

⋆ **Exercise 2.6** (tight bounds)**.** Consider the following case: we have a directed path with n nodes, and the unique identifiers are a permutation of $\{1, 2, \ldots, n\}$. We have seen that 3-colouring the path with a deterministic distributed algorithm takes at least

$$\frac{1}{2}\log^*(n) - 1$$

rounds. On the other hand, the analysis of Exercise 1.6 shows that colouring is possible in

$$\log^*(n) + O(1)$$

rounds. Close the factor-2 gap between the bounds, and design a distributed algorithm that finds a 3-colouring in

$$\frac{1}{2}\log^*(n) + O(1)$$

rounds.

▹ *hint E*

2.5 Bibliographic Notes

The negative result of Section 2.3 is due to Linial [15]; our presentation follows a more streamlined version of the proof [14].

Part II

Graphs

Chapter 3

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.

3.1 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.

3.1.1 Adjacency

If $e = \{u, v\} \in E$, we say that node u is *adjacent* to v, nodes u and v are *neighbours*, node u is *incident* to e, and edge e is also

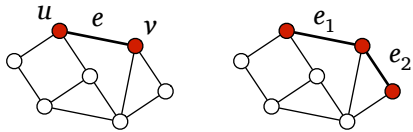


Figure 3.1: Node u is adjacent to node v. Nodes u and v are incident to edge e. Edge e_1 is adjacent to edge e_2.

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 $\deg_G(v)$ neighbours; 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 $v \in V$.

3.1.2 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.

3.1.3 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 3.2. 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 3.3. Note that the length of a cycle is at least 3.

3.1.4 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

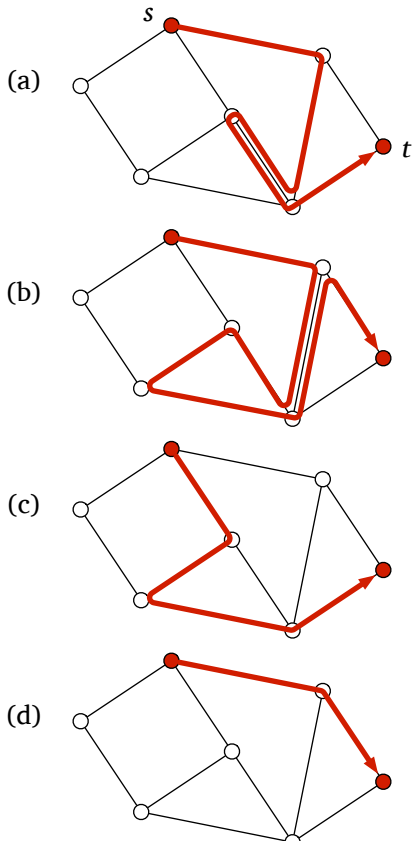


Figure 3.2: (a) A walk of length 5 from s to t. (b) A non-backtracking walk. (c) A path of length 4. (d) A path of length 2; this is a shortest path and hence $\text{dist}_G(s,t) = 2$.

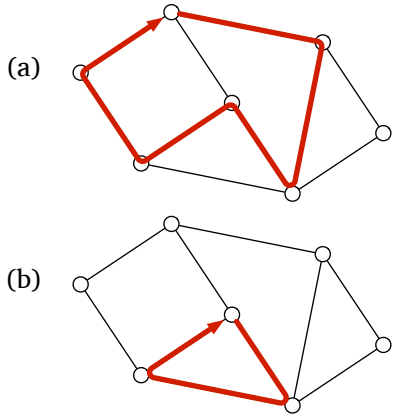


Figure 3.3: (a) A cycle of length 6. (b) A cycle of length 3; this is a shortest cycle and hence the girth of the graph is 3.

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 neighbourhood* of v as follows (see Figure 3.4):

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

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 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*.

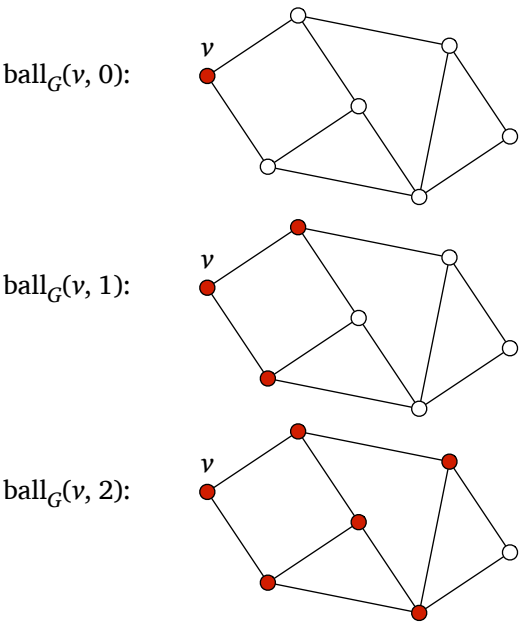


Figure 3.4: Neighbourhoods.

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.

3.1.5 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.

3.2 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 $e \in E$,

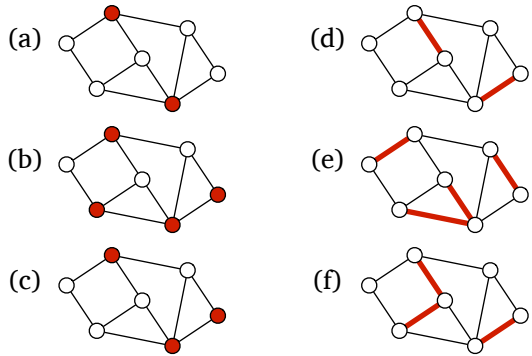


Figure 3.5: Packing and covering problems; see Section 3.2.

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

(c) a *dominating set* if each node $v \notin X$ has at least one neighbour in X, that is, $\text{ball}_G(v,1) \cap X \neq \emptyset$ 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 neighbour in X, that is, $e \cap \left(\bigcup X\right) \neq \emptyset$ for all $e \in E$.

See Figure 3.5 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 *maximisation 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 *minimisation 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 maximisation 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 minimisation 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 3.5a 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 3.5d 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 optimisation 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.

3.3 Labellings 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 colours, 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 labelling 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 colouring* if $f^{-1}(i)$ is an independent set for each i,

(b) a *weak colouring* if each non-isolated node u has a neighbour 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 colouring* 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 3.6 for illustrations.

Usually, the term *colouring* refers to a proper vertex colouring, and the term *edge colouring* refers to a proper edge colouring. The value of k is the *size* of the colouring or the *number of colours*. We will use the term *k-colouring* to refer

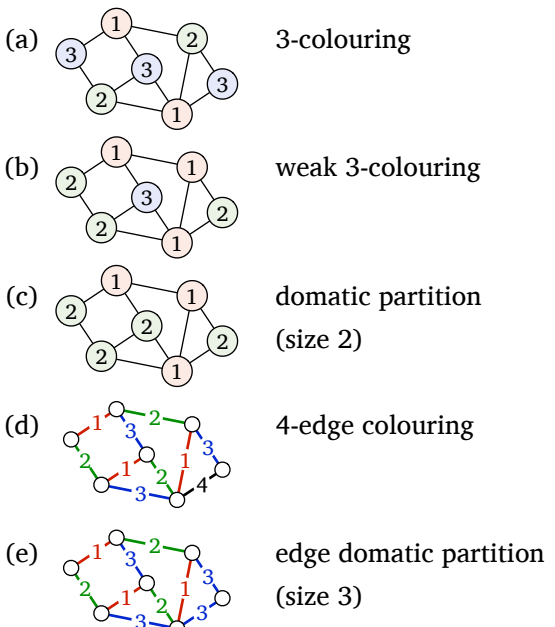


Figure 3.6: Partition problems; see Section 3.3.

to a proper vertex colouring with k colours; the term *k-edge colouring* is defined in an analogous manner.

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

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

On the other hand, domatic partitions are a maximisation 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.

3.4 Factors and Factorisations

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-factorisation* of G if $f^{-1}(i)$ is a d-factor for each i. See Figure 3.7 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.

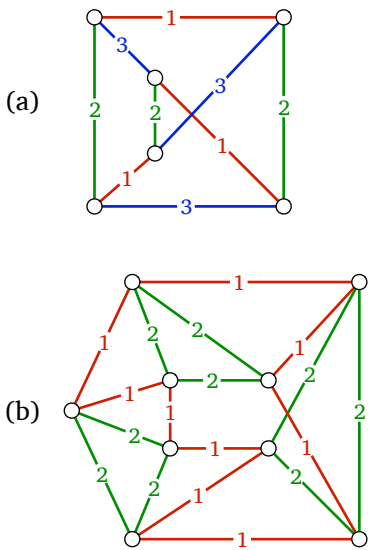


Figure 3.7: (a) A 1-factorisation of a 3-regular graph. (b) A 2-factorisation of a 4-regular graph.

(b) A 1-factorisation is an edge colouring.

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

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

3.5 Approximations

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

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 colouring is a minimisation problem. The set of feasible solutions $\mathcal{S}$ consists of all proper vertex colourings, and $g(f)$ is the number of colours in $f \in \mathcal{S}$.

Often, it is infeasible or impossible to find an optimal solution; hence we resort to approximations. Given a maximisation 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 minimisation 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 minimisation problems and maximisation problems. Other conventions are also used in the literature.

3.6 Directed Graphs and Orientations

Unless otherwise mentioned, all graphs in this book 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 $v \in V$.

3.7 Exercises

Exercise 3.1 (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 3.2 (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 3.3 (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 3.4 (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 3.5 (graph colourings and partitions)**.** Show that

(a) a weak 2-colouring 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-colouring is a domatic partition of size 2,

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

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

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

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

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

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

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

Exercise 3.6 (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 3.7** (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 F*

⋆ **Exercise 3.8** (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-factorisation.

3.8 Bibliographic Notes

The connection between maximal matchings and approximations of vertex covers (Exercise 3.3) is commonly attributed to Gavril and Yannakakis — see, e.g., Papadimitriou and Steiglitz [19]. The connection between minimum maximal matchings and minimum edge dominating sets (Exercise 3.7) is due to Allan and Laskar [1] and Yannakakis and Gavril [28]. Exercise 3.8 is a 120-year-old result due to Petersen [21]. The definition of a weak colouring is from Naor and Stockmeyer [16].

Diestel's book [8] is a good source for graph-theoretic background, and Vazirani's book [26] provides further information on approximation algorithms.

Part III

Models of Computing

Chapter 4

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 — randomised algorithms will be discussed in Chapter 7.

4.1 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 labelled 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 4.2 and 4.3 will formalise this idea.

4.2 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 $p \colon P \to P$ 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 4.1 and 4.2 for illustrations.

4.2.1 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, $\deg_N(v) = |\{\, i \in \mathbb{N} : (v, i) \in P \,\}|$.

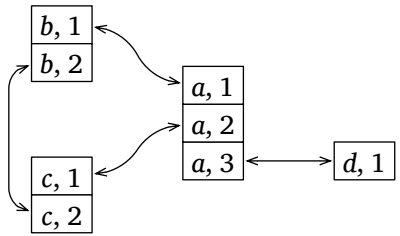


Figure 4.1: A port-numbered network $N = (V, P, p)$. There are four nodes, $V = \{a, b, c, d\}$; the degree of node a is 3, the degrees of nodes b and c are 2, and the degree of node d is 1. The connection function p is illustrated with arrows — for example, $p(a, 3) = (d, 1)$ and conversely $p(d, 1) = (a, 3)$. This network is simple.

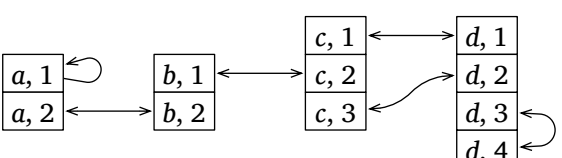


Figure 4.2: A port-numbered network $N = (V, P, p)$. There is a loop at node a, as $p(a, 1) = (a, 1)$, and another loop at node d, as $p(d, 3) = (d, 4)$. There are also multiple connections between c and d. Hence the network is not simple.

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.

4.2.2 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 4.3 for an illustration.

4.2.3 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 labellings*: 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

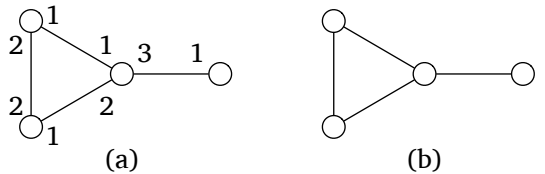


Figure 4.3: (a) An alternative drawing of the simple port-numbered network N from Figure 4.1. (b) The underlying graph G of N.

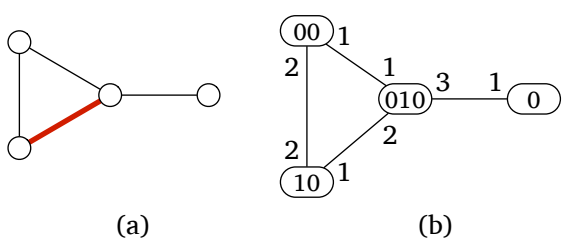


Figure 4.4: (a) A graph $G = (V, E)$ and a matching $M \subseteq E$. (b) A port-numbered network N; graph G is the underlying graph of N. The node labelling $f : V \to \{0,1\}^*$ is an encoding of matching M.

that a node labelling $f : V \to Y$ can be used to represent the following graph-theoretic structures; see Figure 4.4 for an illustration.

Node labelling $g : V \to X$**.** 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 labelling $g : V \to \{0, 1\}$, where g is the indicator function of the set X. That is, $g(v) = 1$ iff $v \in X$.

Edge labelling $g : E \to X$**.** 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 labelling $g : E \to \{0, 1\}$.

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 labellings. For example, we can easily construct a node labelling that encodes both a subset of nodes and a subset of edges.

4.2.4 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 labelling $f : V \to Y$ for some set Y of *local outputs*.

Using the encodings of Section 4.2.3, we can interpret all of the following as distributed graph problems: independent sets, vertex covers, dominating sets, matchings, edge covers, edge dominating sets, colourings, edge colourings, domatic partitions, edge domatic partitions, factors, factorisations, 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-*colouring*: $f \in \Pi(N)$ if f encodes a 2-colouring of the underlying graph of N. Note that we will have $\Pi(N) = \varnothing$ if the underlying graph of N is not bipartite.

4.3 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.

4.3.1 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) $\text{init}_{A,d} \colon \text{Input}_A \to \text{States}_A$ initialises 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.

4.3.2 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 labelling 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 $m_t : P \to \mathrm{Msg}_A$ 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 $\mathrm{send}_{A,\ell}(x_{t-1}(v))$.

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

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

We say that algorithm A *stops in time* T if $x_T(u) \in \mathrm{Output}_A$ 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.

4.3.3 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 4.2.4). 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-coloured graphs in time k*: here $\mathscr{F}$ consists of all graphs that admit a k-colouring; Π' is the problem of finding a k-colouring; Π is the problem of finding a maximal independent set; and T is the constant function $T : n \mapsto k$.

4.4 Example: Colouring Paths

Recall the algorithm P3C for 3-colouring 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 colouring graphs with 3 colours.
- Π' is the problem of colouring graphs with any number of colours.

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 colouring, 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 colour):

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

Messages are also integers:

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

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

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

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

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

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 $\mathrm{receive}_{A,d}$ for degrees $d \leq 2$:

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

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

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

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 4.3.3.

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.

4.5 Example: Maximal Matching in Two-Coloured Graphs

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

4.5.1 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 neighbour 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 initialised

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

The algorithm is presented in Table 4.1; see Figure 4.5 for an illustration.

4.5.2 Analysis

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

Lemma 4.1. *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 $M(v) = \emptyset$ 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 $\mathsf{MR}(i)$ after round $2i - 2$. In the former case, u sends '*proposal*' to v

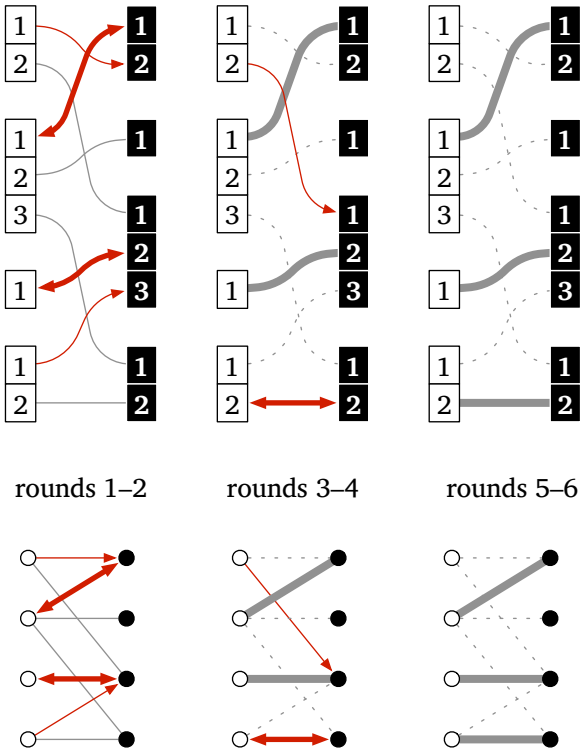


Figure 4.5: Algorithm BMM; the illustration shows the algorithm both from the perspective of the port-numbered network N and from the perspective of the underlying graph G. Arrows pointing right are proposals, and arrows pointing left are acceptances. Wide grey edges have been added to matching M.

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, $M(v) \neq \emptyset$: Let $i = \min M(v)$.
  Send '*accept*' to port (v,i). Switch to state MS(i).
- State UR, $X(v) = \emptyset$: 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 4.1: Algorithm BMM; here $k = 1, 2, \ldots$.

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 4.2. *Algorithm* BMM *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 4.1, 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 4.3. *Algorithm* BMM *finds a maximal matching in any two-coloured 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 $\mathsf{MS}(i)$ if and only if v stops in state $\mathsf{MS}(j)$. If u stops in state $\mathsf{MS}(i)$, it has received an '*accept*' from v, and v stops in state $\mathsf{MS}(j)$. Conversely, if v stops in state $\mathsf{MS}(j)$, it has received a '*proposal*' from u and it sends an '*accept*' to u, after which u stops in state $\mathsf{MS}(i)$.

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 4.1 implies that at least one of them becomes matched before or during round 2Δ. □

4.6 Example: Vertex Covers

We will now give a distributed algorithm VC3 that finds a 3-approximation of a minimum vertex cover; we will use algorithm BMM 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 colouring of the network). The algorithm that we will see in this section makes no such assumptions. We can run algorithm VC3 in any port-numbered network, without any additional input. In particular, we do not need any kind of colouring, unique identifiers, or randomness.

4.6.1 Virtual 2-Coloured 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 4.6 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':

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

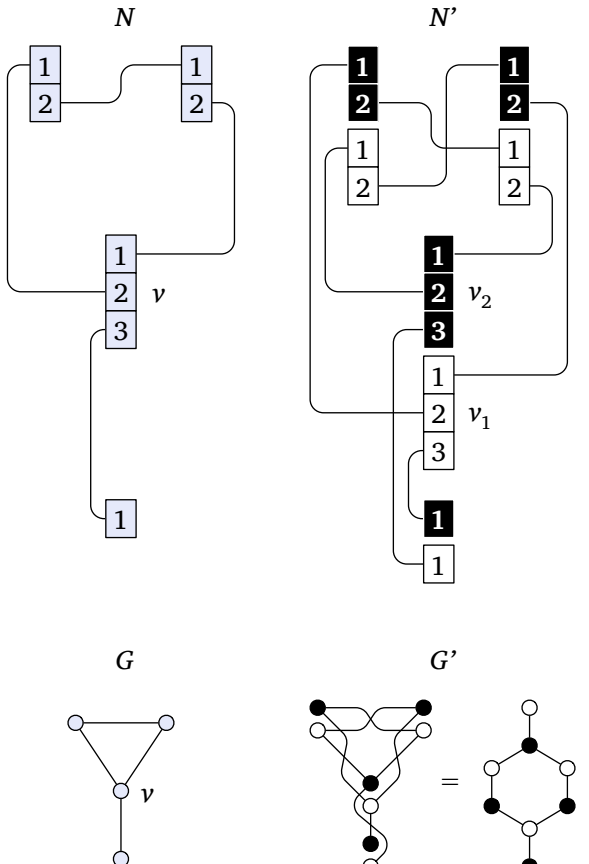


Figure 4.6: Construction of the virtual network N' in algorithm VC3.

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

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

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

4.6.2 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 algorithm BMM of Section 4.5 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 colouring of N; hence we would not be able to apply algorithm BMM on N.

4.6.3 Algorithm

Now we are ready to present algorithm VC3 that finds a vertex cover:

(a) Simulate algorithm BMM 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.

4.6.4 Analysis

Clearly algorithm VC3 stops, as algorithm BMM 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'. Algorithm BMM outputs a maximal

matching $M' \subseteq E'$ for G'. Define the edge set $M \subseteq E$ as follows:

$$M = \big\{ \{u, v\} \in E : \{u_1, v_2\} \in M' \text{ or } \{u_2, v_1\} \in M' \big\}. \qquad (4.1)$$

See Figure 4.7 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 4.4. *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 analyse 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 vertex cover C^* contains at least a fraction $1/3$ of the nodes in H; see Figure 4.8 for an example. Then it follows that C is at most 3 times as large as a minimum vertex cover.

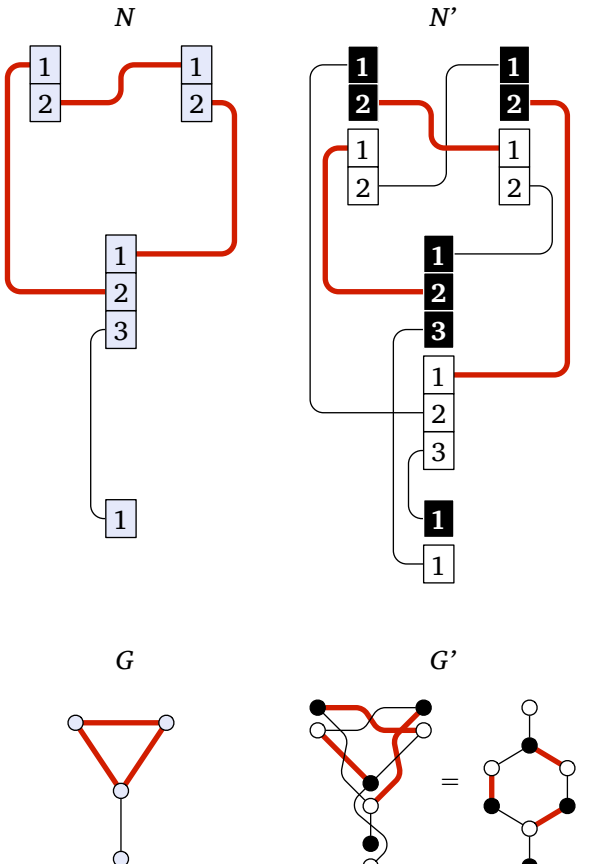


Figure 4.7: Set $M \subseteq E$ (left) and matching $M' \subseteq E'$ (right).

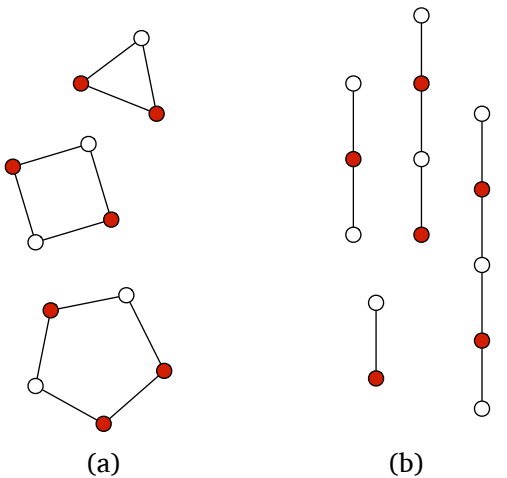


Figure 4.8: (a) In a cycle with n nodes, any vertex cover contains at least $n/2$ nodes. (b) In a path with n nodes, any vertex cover contains at least $n/3$ nodes.

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, VC3 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.

4.7 Exercises

Exercise 4.1 (formalising BMM)**.** Present algorithm BMM from Section 4.5 in a formally precise manner, using the definitions of Sections 4.2 and 4.3. Try to make Msg_A as small as possible.

Exercise 4.2 (formalising VC3)**.** Present algorithm VC3 from Section 4.6 in a formally precise manner, using the definitions of Sections 4.2 and 4.3. Try to make both Msg_A and States_A as small as possible.

▷ *hint G*

Exercise 4.3 (stopped nodes)**.** In the formalism of this chapter, a node that stops will repeatedly send messages to its neighbours. Show that this detail is irrelevant, and we can always re-write 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 4.4 (more than two colours)**.** Design a distributed algorithm that finds a maximal matching in k-coloured graphs. You can assume that k is a known constant.

Exercise 4.5 (analysis of VC3)**.** Is the analysis of VC3 tight? That is, is it possible to construct a network N such that VC3 outputs a vertex cover that is exactly 3 times as large as the minimum vertex cover of the underlying graph of N?

⋆ **Exercise 4.6** (implementation)**.** Using your favourite programming language, implement a simulator that lets you play with distributed algorithms in the port-numbering model. Implement BMM and VC3 and try them out in the simulator.

⋆ **Exercise 4.7** (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 H*

4.8 Bibliographic Notes

The concept of a port numbering is from Angluin's [2] work. Algorithm BMM is due to Hańćkowiak et al. [11], and algorithm VC3 is from a paper with Polishchuk [22].

Chapter 5

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 5.1. Following the standard terminology of the field, we will use the term "LOCAL model" to refer to networks with unique identifiers.

5.1 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 labelled with a unique integer, and the labels are assumed to be relatively small. We will use the shorthand notation $\chi = |V|^c$.

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*

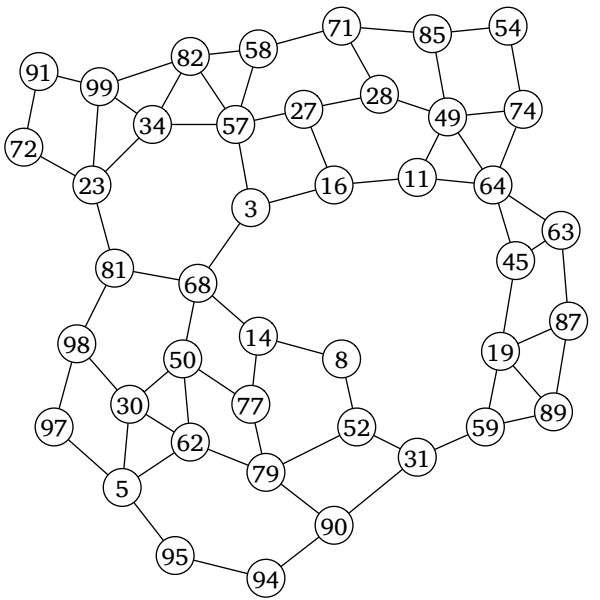


Figure 5.1: A network with unique identifiers.

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 $v \in V$.

5.2 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 algorithm Gather that accomplishes this.

In algorithm Gather, 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

$$V(v, r) = \mathrm{ball}_G(v, r), \tag{5.1}$$

$$E(v, r) = \big\{\{s, t\} : s \in \mathrm{ball}_G(v, r),\ t \in \mathrm{ball}_G(v, r-1)\big\}. \tag{5.2}$$

Now define the graph

$$G(v, r) = (V(v, r), E(v, r)). \tag{5.3}$$

See Figure 5.2 for an illustration.

The following properties are straightforward corollaries of (5.1)–(5.3).

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

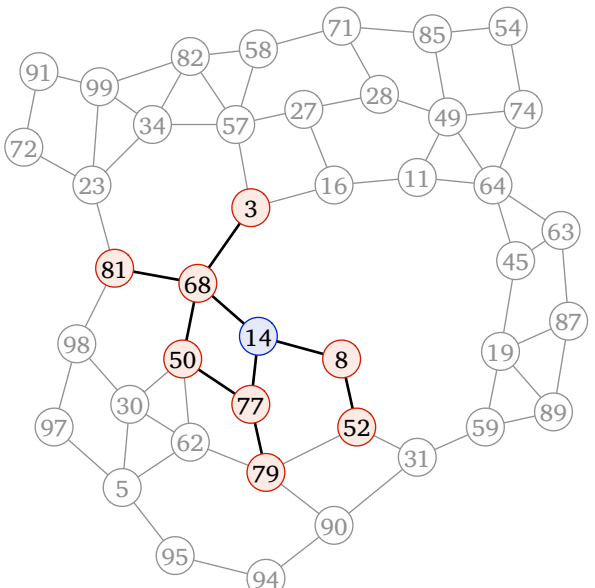


Figure 5.2: Subgraph $G(v, r)$ defined in (5.3), for $v = 14$ and $r = 2$.

(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:

$$V(v, r) = \bigcup_{u \in V(v,1)} V(u, r - 1), \tag{5.4}$$

$$E(v, r) = \bigcup_{u \in V(v,1)} E(u, r - 1). \tag{5.5}$$

Algorithm Gather maintains the following invariant: after round $r \geq 1$, each node $v \in V$ has constructed graph $G(v, r)$. The execution of Gather 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 neighbours. 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 (5.4) and (5.5).

(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 Gather so that we can discover not only the underlying graph $G = (V, E)$ but also the original port-numbered network $N = (V, P, p)$.

5.3 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 *centralised* (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 algorithm Gather from Section 5.2.

(b) All nodes use the centralised 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.

5.4 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 (recall the discussion in Section 1.2), 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 colouring is an example of such a problem.

In the rest of this chapter, we will design an efficient distributed algorithm that finds a graph colouring in the LOCAL model. The algorithm will find a proper vertex colouring with $\Delta + 1$ colours in $O(\Delta^2 + \log^* |V|)$ communication round, for any graph of maximum degree Δ. We will first present some simpler algorithms that will be used as subroutines — many of

these are generalisations and variants of algorithms P3C and P3CBit that we saw in Chapter 1.

5.5 Algorithm BDGreedy: Colour Reduction in Bounded-Degree Graphs

Let $x \in \mathbb{N}$. We present an algorithm called BDGreedy that reduces the number of colours from x to

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

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

5.5.1 Algorithm

The algorithm proceeds as follows; here f is the x-colouring that we are given as input and g is the y-colouring that we produce as output. See Figure 5.3 for an illustration.

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

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

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

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

local maxima. Let

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

be the set of *free colours* in the neighbourhood 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 colour is a local maximum re-colours itself with the first available free colour.

5.5.2 Analysis

Lemma 5.1. *Algorithm* BDGreedy *reduces the number of colours from x to*

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

where Δ is the maximum degree of the graph.

Proof. Let us first prove that $g(v) \in \{1, 2, \dots, y\}$ for all $v \in V$. As f is a proper colouring, 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

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

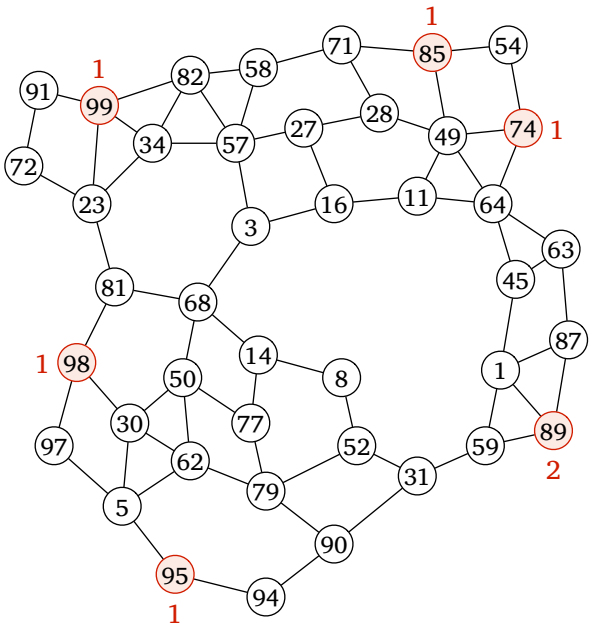


Figure 5.3: Greedy colour reduction. The active nodes have been highlighted. Note that in the original colouring f, the largest colour was 99, while in the new colouring, the largest colour is strictly smaller than 99 — we have successfully reduced the number of colours in the graph.

(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

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

Next we will show that g is a proper vertex colouring 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 $f(v) \leq \max C(v)$; 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 colours simultaneously in parallel, without any risk of choosing colours that might conflict with each other.

5.5.3 Remarks

Algorithm BDGreedy does not need to know the number of colours x or the maximum degree Δ; we only used them in the analysis. We can take any graph, blindly apply algorithm BDGreedy, and we are guaranteed to reduce the number of colours by one — provided that the number of colours was larger

than $\Delta + 1$. In particular, we can apply algorithm BDGreedy repeatedly until we get stuck, at which point we have a $(\Delta+1)$-colouring of G — we will formalise and generalise this idea in Exercise 5.3.

5.6 Directed Pseudoforests

We will next study graph colouring in so-called directed pseudoforests. As we will see later, algorithms that colour directed pseudoforests can be used as subroutines in algorithms that colour bounded-degree graphs.

A *directed pseudoforest* is a directed graph $G = (V, E)$ such that each node $v \in V$ has $\text{outdegree}_G(v) \leq 1$; see Figure 5.4 for an example.

Compare the definition of a directed pseudoforest with the definition of a *pseudoforest* in Section 3.1.4. We make the following observations:

(a) Let H be an undirected graph, and let G be an orientation of H. If G is a directed pseudoforest, then H is a pseudoforest.

(b) Let H be a pseudoforest. There exists an orientation G of H such that G is a directed pseudoforest.

(c) An orientation of a pseudoforest is not necessarily a directed pseudoforest.

If $(u, v) \in E$, we say that v is a *successor* of u and u is a *predecessor* of v. By definition, in a directed pseudoforest each node has at most one successor.

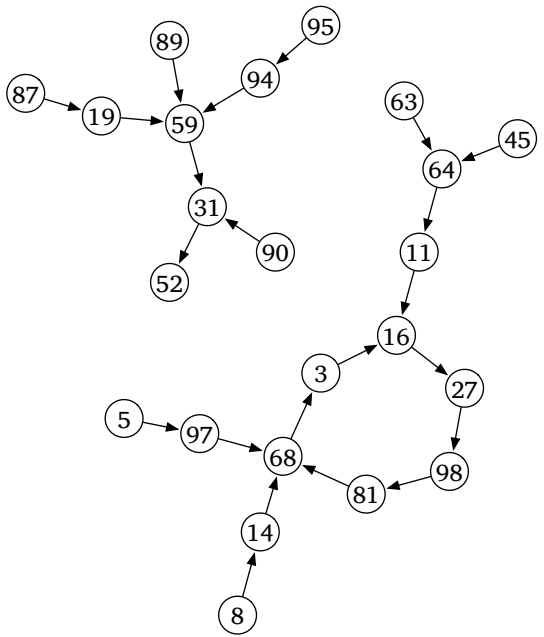


Figure 5.4: A directed pseudoforest with a colouring f.

5.7 Algorithm DPGreedy: Colour Reduction in Directed Pseudoforests

Let $G = (V, E)$ be a directed pseudoforest, and let f be a proper vertex colouring of G with x colours, for some $x \geq 4$. We design a distributed algorithm DPGreedy that reduces the number of colours from x to $x - 1$ in two communication rounds.

Note the key difference between algorithms BDGreedy and DPGreedy: algorithm BDGreedy gets stuck at $\Delta + 1$ colours, while DPGreedy can be used to reduce the number of colours down to 3.

5.7.1 Overview

The high-level structure of algorithm DPGreedy is as follows:

(a) We are given an x-colouring f (Figure 5.4).

(b) In one communication round, given f we construct another x-colouring s, which has the property that each node is adjacent to at most two different colour classes (Figure 5.5).

(c) In one communication round, given s we construct an $(x-1)$-colouring g using algorithm BDGreedy (Figure 5.6).

5.7.2 Algorithm

First, each $v \in V$ computes $s(v)$ as follows; see Figure 5.5:

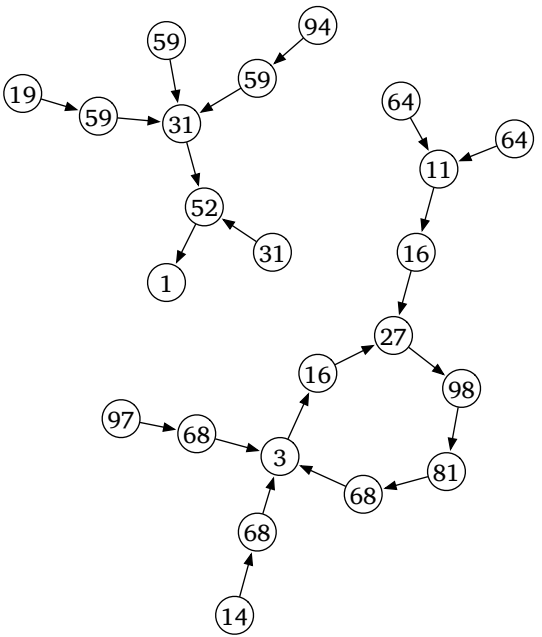


Figure 5.5: A directed pseudoforest with colouring s; compare with Figure 5.4. In colouring s, all predecessors of a node have the same colour; hence each node is adjacent to nodes of only two different colours.

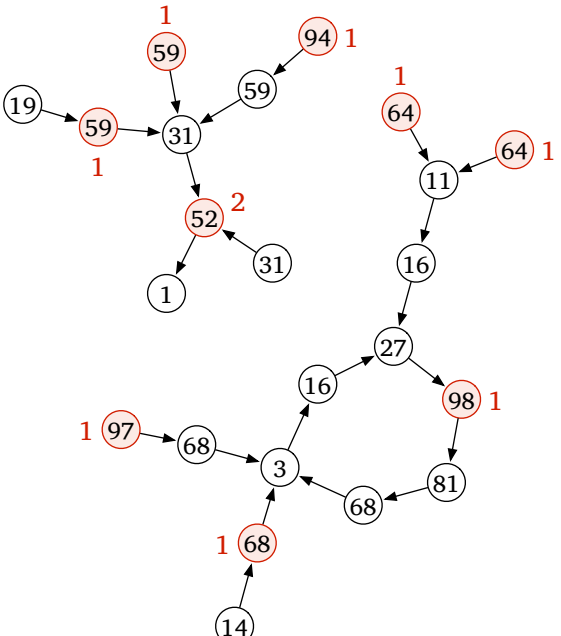


Figure 5.6: Algorithm BDGreedy applied to a directed pseudo-forest with colouring s. The active nodes are highlighted.

(a) If $\text{outdegree}_G(v) = 1$, let u be the successor of v, and let $s(v) = f(u)$.

(b) Otherwise, if $f(v) > 1$, let $s(v) = 1$.

(c) Otherwise $s(v) = 2$.

Then we apply BDGreedy from Section 5.5 to labelling s to construct another labelling g.

5.7.3 Analysis

We will first prove that the values $s(v)$ form a proper x-colouring of G. Moreover, we will show that each node is adjacent to only two different colours in colouring s.

Lemma 5.2. *Function s is an x-colouring of G.*

Proof. By construction, we have $s(v) \in \{1, 2, \ldots, x\}$. Now let $(u, v) \in E$. We need to show that $s(u) \neq s(v)$. To see this, observe that v is a successor of u. Hence

$$s(u) = f(v) \neq s(v).$$ □

Lemma 5.3. *Define*

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

We have $|C(v)| \leq 2$ for each node $v \in V$.

Proof. For each predecessor u of v, we have $s(u) = f(v)$. That is, all predecessors of v have the same colour. Hence $C(v)$ consists of at most two different values: the common colour of the predecessors of v (if any), and the colour of the successor of v (if any). □

Now let us consider what happens when we apply algorithm BDGreedy to construct labelling g. Each active node v will choose a colour

$$g(v) = \min \bar{C}(v) \in \{1, 2, 3\},$$

while each passive node v will output its old colour $g(v) = s(v)$. In particular, if the number of colours in f was $x \geq 4$, then the number of colours in g is at most $x - 1$.

We conclude that we have designed algorithm DPGreedy that reduces the number of colours from $x \geq 4$ to $x - 1$ in directed pseudoforests in 2 communication rounds. In particular, we can reduce the number of colours from any number $x \geq 3$ to 3 in $2(x - 3)$ rounds.

5.8 Algorithm DPBit: Fast Colour Reduction in Directed Pseudoforests

So far we have only seen algorithms that reduce the number of colours by one in each iteration. In this section we will present an algorithm that is *much* faster. We present algorithm DPBit that reduces the number of colours from 2^x to $2x$ in one communication round, in any directed pseudoforest. We will assume that $x \geq 1$ is a known constant. In essence, this is the same algorithm as P3CBit from Section 1.4 — fast colour reduction in directed pseudoforests is almost as easy as fast colour reduction in directed paths.

5.8.1 Algorithm

We assume that we are given a proper vertex colouring

$$f : V \to \{1, 2, \ldots, 2^x\}$$

of a directed pseudoforest $G = (V, E)$. We will use the values $s(v)$ defined in Section 5.7 — recall that $f(v) \neq s(v)$ for each node v, and if u is the successor of v, we have $s(v) = f(u)$.

The key idea is that each node compares the *binary encodings* of the values $s(v)$ and $f(v)$. More precisely, if $j \in \{1, 2, \ldots, 2^x\}$ is a colour, let us use $\langle j \rangle$ to denote the binary encoding of $j - 1$; this is always a binary string of length x. For example, if $x = 3$, we have

$$\langle 1 \rangle = 000, \quad \langle 2 \rangle = 001, \quad \ldots, \quad \langle 8 \rangle = 111.$$

If $i \in \{0, 1, \ldots, x - 1\}$, we use the notation $\langle j \rangle_i$ to refer to bit i of the binary string $\langle j \rangle$, counting from the lowest-order bit. For example, $\langle 2 \rangle_0 = 1$ and $\langle 2 \rangle_1 = 0$.

In algorithm DPBit, each node first finds out the values $s(v)$ and $f(v)$ — this takes only one communication round — and then compares the binary strings $\langle s(v) \rangle$ and $\langle f(v) \rangle$. As $s(v) \neq f(v)$, there is at least one bit in these strings that differs. Let

$$i(v) = \min\{i : \langle f(v) \rangle_i \neq \langle s(v) \rangle_i\}$$

be the *index* of the first bit that differs, and let

$$b(v) = \langle f(v) \rangle_{i(v)}$$

be the *value* of the bit that differs. Note that $0 \leq i(v) \leq x-1$ and $0 \leq b(v) \leq 1$. We encode the pair $\big(i(v), b(v)\big)$ as a colour

$$g(v) = 2i(v) + b(v) + 1.$$

Algorithm DPBit outputs the value $g(v)$.

5.8.2 Analysis

The key observation is that the pairs $\big(i(v), b(v)\big)$ form a proper colouring of G.

Lemma 5.4. *Let $(u,v) \in E$. We have $i(u) \neq i(v)$ or $b(u) \neq b(v)$.*

Proof. If $i(u) \neq i(v)$, the claim is trivial. Otherwise $i(u) = i(v)$. As v is the successor of u, we have $s(u) = f(v)$. Hence

$$b(v) = \langle f(v) \rangle_{i(v)} = \langle s(u) \rangle_{i(u)},$$

and by the definition of $i(u)$,

$$b(u) = \langle f(u) \rangle_{i(u)} \neq \langle s(u) \rangle_{i(u)}.$$

In summary, $b(u) \neq b(v)$. □

Note that if we have $g(u) = g(v)$ for two nodes u and v, this implies $b(u) = b(v)$ and $i(u) = i(v)$. Hence Lemma 5.4 implies that g is a proper vertex colouring of G. Moreover, we have $1 \leq g(v) \leq 2x$, and hence g is a $2x$-colouring of G.

In summary, we have designed algorithm DPBit that reduces the number of colours from 2^x to $2x$ in one communication round — given a 2^x-colouring f, the algorithm outputs a $2x$-colouring g.

5.9 Algorithm DP3C: Fast 3-Colouring in Directed Pseudoforests

Assume that we know $|V|$. We will design algorithm DP3C that finds a 3-colouring in $O(\log^* |V|)$ rounds in any directed pseudoforest. The algorithm proceeds as follows:

(a) Use the unique identifiers to construct a colouring with χ colours.

(b) Repeat algorithm DPBit for $\log^* \chi$ times to reduce the number of colours from χ to 6.

(c) Repeat algorithm DPGreedy for 3 times to reduce the number of colours from 6 to 3.

Here phase (a) takes 0 communication rounds, phase (b) takes $\log^* \chi$ communication rounds, and phase (c) takes 6 communication rounds. The analysis is, in essence, identical to what we already did in Exercise 1.5.

5.10 Algorithm BDColour: Fast Colouring in Bounded-Degree Graphs

Now we will turn our attention to bounded-degree graphs. Assume that we know $|V|$ and Δ. We will now design a distributed algorithm BDColour that finds a $(\Delta + 1)$-colouring in any graph of maximum degree at most Δ in $O(\Delta^2 + \log^* |V|)$ rounds.

5.10.1 Preliminaries

For each node v and each port number i, node v sends the pair (v, i) to port i. This way a node u learns the following information about each node v that is adjacent to u: what is the unique identifier of v, which port of u is connected to v, and which port of v is connected to u. This step requires one communication round.

5.10.2 Orientation

We construct an orientation $G' = (V, E')$ of G as follows: we have $(u, v) \in E'$ if and only if $\{u, v\} \in E$ and $u < v$. That is, we use the unique identifiers to orient the edges; see Figure 5.7. Each node only needs to know the orientation of its incident edges. This step requires zero communication rounds.

5.10.3 Partition in Pseudoforests

For each $i = 1, 2, \ldots, \Delta$, we construct a subgraph $G_i = (V, E_i)$ of G' as follows: we have $(u, v) \in E_i$ if and only if $(u, v) \in E'$ and v is connected to port number i of u in N. See Figure 5.8.

Observe that the sets $E_1, E_2, \ldots, E_\Delta$ form a partition of E': for each directed edge $e \in E'$ there is precisely one i such that $e \in E_i$. Also note that for each node $u \in V$ and for each index i there is at most one neighbour v such that $(u, v) \in E_i$. It follows that the outdegree of any node v in $G_i = (V, E_i)$ is at most one, and therefore G_i is a *directed pseudoforest*.

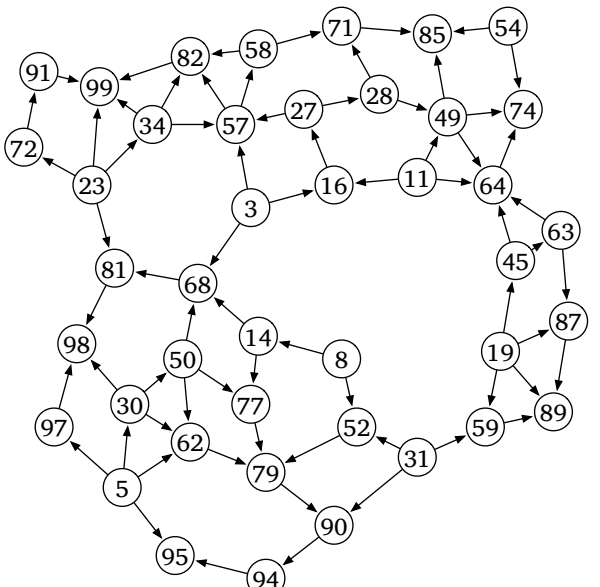


Figure 5.7: Orientation G' derived from the unique identifiers.

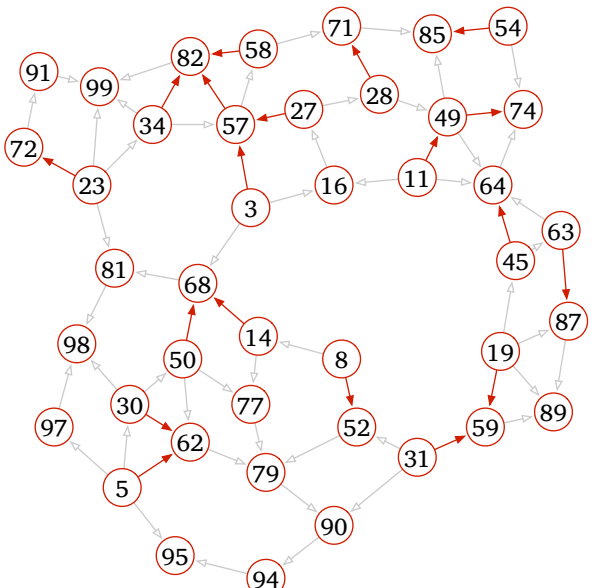


Figure 5.8: Subgraph G_i of G'. Each node has outdegree at most one.

Each node only needs to know which of its incident edges are in which subset E_i. This step requires zero communication rounds.

5.10.4 Parallel Colouring of Pseudoforests

For each i, we use algorithm DP3C to construct a 3-colouring g_i of G_i. Each node $v \in V$ needs to know the value $g_i(v)$ for each i. This step takes only $O(\log^* |V|)$ rounds: we can simulate the execution of A in parallel for all subgraphs G_i. In the simulation, each node has Δ different roles, one for each subgraph G_i.

5.10.5 Merging Colourings

For each $j = 0, 1, \ldots, \Delta$, define

$$E'_j = \bigcup_{i=1}^{j} E_i$$

and $G'_j = (V, E'_j)$. Note that G'_0 is a graph without any edges, each G'_j is a subgraph of G', and $G'_\Delta = G'$.

We will construct a sequence of colourings $g'_0, g'_1, \ldots, g'_\Delta$ such that g'_j is a $(\Delta+1)$-colouring of the subgraph G'_j. Then it follows that we can output $g = g'_\Delta$, which is a $(\Delta+1)$-colouring of G' and hence also a $(\Delta+1)$-colouring of the original graph G.

Our construction is recursive. The base case of $j = 0$ is trivial: we can choose $g'_0(v) = 1$ for all $v \in V$, and this is certainly a proper $(\Delta+1)$-colouring of G'_0.

Now assume that we have already constructed a $(\Delta+1)$-colouring g'_{j-1} of G'_{j-1}. Recall that g_j is a 3-colouring of G_j; see

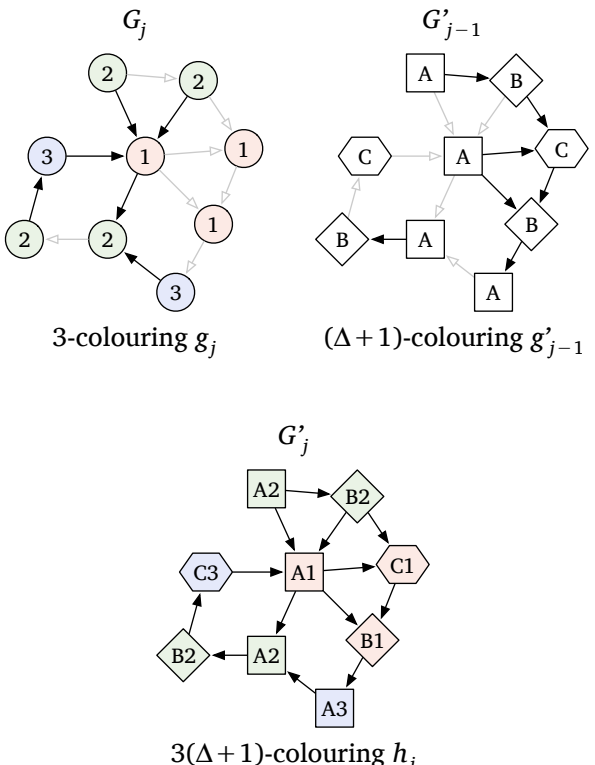


Figure 5.9: Merging a 3-colouring g_j of directed pseudotree G_j and a $(\Delta+1)$-colouring g'_{j-1} of subgraph G'_{j-1}. The end result is a proper $3(\Delta+1)$-colouring h_j of subgraph G'_j.

Figure 5.9. Define a function h_j as follows:

$$h_j(v) = (\Delta+1)(g_j(v)-1) + g'_{j-1}(v).$$

Observe that h_j is a proper $3(\Delta+1)$-colouring of G'_j. To see this, consider an edge $(u,v) \in E'_j$. If $(u,v) \in E_j$, we have $g_j(u) \neq g_j(v)$, which implies $h_j(u) \neq h_j(v)$. Otherwise $(u,v) \in E'_{j-1}$, and we have $g'_{j-1}(u) \neq g'_{j-1}(v)$, which implies $h_j(u) \neq h_j(v)$.

Now we use $2(\Delta+1)$ iterations of BDGreedy to reduce the number of colours from $3(\Delta+1)$ to $\Delta+1$. This way we can construct a proper $(\Delta+1)$-colouring g'_j of G'_j in time $O(\Delta)$.

After Δ phases, we have eventually constructed colouring $g = g'_\Delta$; the total running time is $O(\Delta^2)$, as each phase takes $O(\Delta)$ communication rounds.

5.10.6 Summary

In this section, we have seen how to find a proper vertex colouring with $\Delta+1$ colours in $O(\Delta^2 + \log^* x)$ communication round in the LOCAL model, for any graph of maximum degree Δ. In the exercises, we will see that efficient algorithms for vertex colouring can be used as subroutines to solve many other problems.

5.11 Exercises

Exercise 5.1 (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 colouring with $O(\Delta)$ colours.

▷ *hint I*

Exercise 5.2 (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 J*

Exercise 5.3 (iterated greedy)**.** Design a colour reduction algorithm A with the following properties: given any graph $G = (V, E)$ and any proper vertex colouring f, algorithm A outputs a proper vertex colouring 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 colours in colouring 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 K*

Exercise 5.4 (distance-2 colouring)**.** Let $G = (V, E)$ be a graph. A *distance-2 colouring with k colours* 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 colouring with $O(\Delta^2)$ colours for any graph $G \in \mathscr{F}$ in the LOCAL model.

▷ *hint L*

⋆ **Exercise 5.5** (numeral systems)**.** Algorithm DPBit is based on the idea of identifying a digit that differs in the *binary* encodings of the colours. Generalise the idea: design an analogous algorithm that finds a digit that differs in the base-k encodings of the colours, for an arbitrary k, and analyse the running time of the algorithm (cf. Exercise 1.6). Is the special case of $k = 2$ the best possible choice?

⋆ **Exercise 5.6** (from bits to sets)**.** Algorithm DPBit can reduce the number of colours 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 colours 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 colours.

In this exercise we will design a distributed algorithm DPSet that reduces the number of colours from

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

to $2x$ in one round, for any positive integer x. For example, we can reduce the number of colours 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, algorithm DPSet does not get stuck at six colours; we can use the same algorithm to reduce the number of colours to four. Moreover, at least in this case the algorithm seems to be much more efficient — algorithm DPSet can reduce the number of colours from 184756 to 6 in two rounds, while algorithm DPBit requires at three rounds to achieve the same reduction.

The basic structure of algorithm DPSet follows algorithm DPBit — 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 colour is different: as the name suggests, we will not interpret colours 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 colour $g(v)$ of a node $v \in V$ as follows:

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

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

Analyse the running time of DPSet and compare it with DPBit. Is DPSet always faster? Can you prove a general result analogous to the claim of Exercise 1.6?

⋆ **Exercise 5.7** (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 M*

5.12 Bibliographic Notes

The model of computing is from Linial's [15] seminal paper, and the name LOCAL is from Peleg's [20] book. Algorithm DPBit is based on the idea originally introduced by Cole and Vishkin [6] and further refined by Goldberg et al. [10]. The idea of algorithm DPSet is from Naor and Stockmeyer [16]. Algorithm BDColour is from Goldberg et al. [10] and Panconesi and Rizzi [18]. The algorithm of Exercise 5.7 is from Friedman and Kogan [9].

Chapter 6

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 $O(\mathrm{diam}(G))$ 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".

6.1 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

$G = (V, E) \in \mathcal{F}$, 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.

6.2 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 6.1). However, there is a notable exception: algorithm Gather from Section 5.2. 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 $O(\mathrm{diam}(G))$ 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 6.6).

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.

6.3 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 6.7.

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.

6.4 Algorithm Wave: 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 neighbours, 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*' to all other ports, switch to state i, and stop; otherwise it does nothing. See Figure 6.1.

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

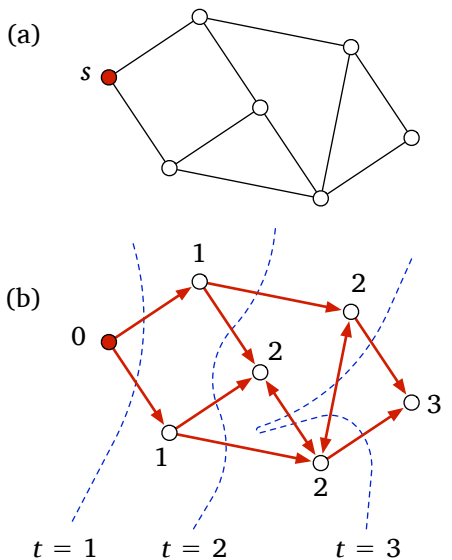


Figure 6.1: (a) Graph G and leader s. (b) Execution of algorithm Wave on graph G. The arrows denote '*wave*' messages, and the dotted lines indicate the communication round during which these messages were sent.

value i. The running time of the algorithm is $O(\mathrm{diam}(G))$ rounds in the CONGEST model.

6.5 Algorithm BFS: 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 6.2 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 centralised algorithm that constructs the BFS tree and distance labels: breadth-first search. We start with an empty tree and unlabelled 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 neighbour v of u. If we have not labelled v yet, we will label it with $d(v) = i + 1$, and add the edge (u, v) to the BFS tree. This way all nodes that

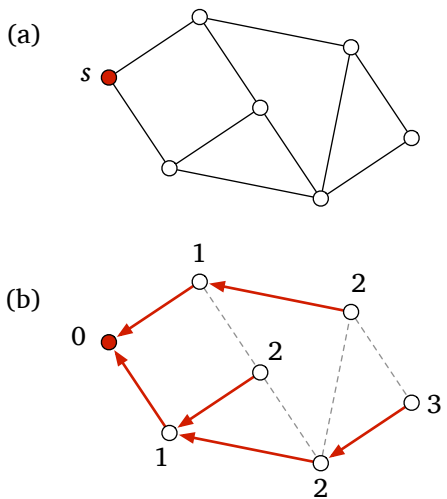


Figure 6.2: (a) Graph G and leader s. (b) BFS tree T (arrows) and distance labels $d(v)$ (numbers).

are at distance i from s in G will be labelled 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:

- $d(v)$: 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).
- $a(v)$: acknowledgement — 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 neighbours.
- If a node u with $d(u) = \perp$ 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 neighbours that accepted proposals.

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

- Each node v with $a(v) = 1$ and $p(v) \neq \perp$ will send an *acknowledgement* to port $p(v)$.
- Each node v with $a(v) = 0$ and $C(v) \neq \perp$ will set $a(v) \leftarrow 1$ when it has received acknowledgements from each port of $C(v)$. In particular, if a node has $C(v) = \varnothing$, it can set $a(v) \leftarrow 1$ without waiting for any acknowledgements.

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 acknowledgements 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.

6.6 Algorithm Leader: 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 6.3:

(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.

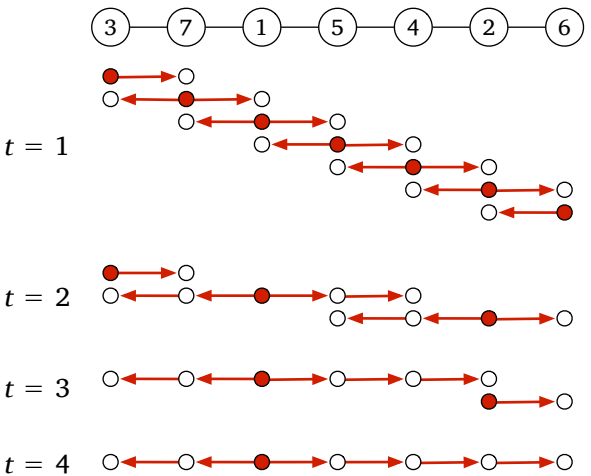


Figure 6.3: Leader election. Each node v will launch a process BFS_v that attempts to construct a BFS tree T_v rooted at v. Other nodes will happily follow BFS_v if v is the smallest leader they have seen so far; otherwise they will start to ignore messages related to BFS_v. Eventually, precisely one 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 acknowledgements from every child.

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 acknowledgements, node s will eventually know that we have successfully constructed a BFS tree T_s rooted at it.
- 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 acknowledgements related to tree T_u from all neighbours.

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.

6.7 Algorithm APSP: 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 $O(n \operatorname{diam}(G))$ 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.

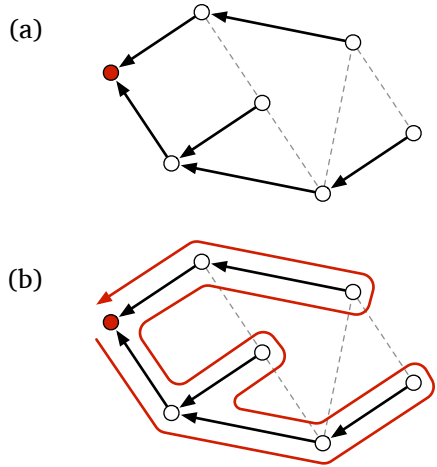


Figure 6.4: (a) BFS tree T_s rooted at s. (b) A depth-first traversal w_s of T_s.

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 6.4.

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 triggered Wave_v, and Wave_v will never catch up with Wave_u as both of them travel at the same speed. See Figure 6.5 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).

6.8 Exercises

Exercise 6.1 (prior algorithms)**.** In Chapters 4 and 5 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 three examples of such algorithms.

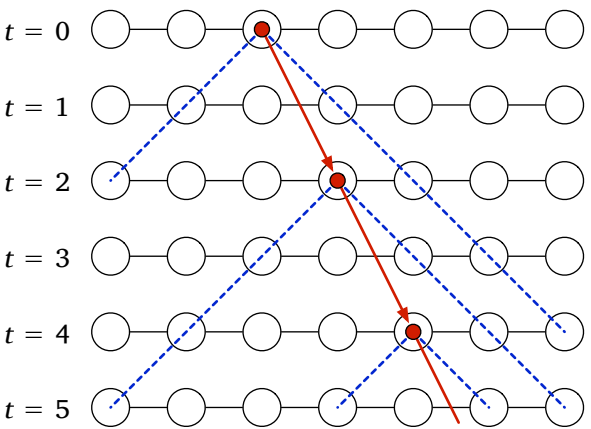


Figure 6.5: Algorithm APSP: the token walks along the BFS tree at speed 0.5 (thick arrows), while each Wave_v moves along the original graph at speed 1 (dashed lines). The waves are strictly nested: if Wave_v was triggered after Wave_u, it will never catch up with Wave_u.

Exercise 6.2 (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 6.3 (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 6.4 (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 6.5 (gathering)**.** Assume that the input graph is connected. In Section 5.2 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 6.6** (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 N*

⋆ **Exercise 6.7** (APSP lower bounds)**.** Assume that the input graph is connected. Prove that there is no algorithm that solves the APSP problem in time $o(|V|)$ in the CONGEST model.

6.9 Bibliographic Notes

The name CONGEST is from Peleg's [20] book. Algorithm APSP is due to Holzer and Wattenhofer [12] — surprisingly, it was published only as recently as in 2012.

Chapter 7

Randomised Algorithms

All models of computing that we have studied so far were based on the formalism that we introduced in Chapter 4: 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 randomised distributed algorithms.

7.1 Definitions

Let us first define a *randomised distributed algorithms in the* PN *model* or, in brief, a *randomised* PN *algorithm*. We extend the definitions of Section 4.3 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 initial state $f(u)$. 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 randomised PN algorithms. We can now extend the definitions in a natural manner to define randomised algorithms in the LOCAL model (add unique identifiers, see Chapter 5) and randomised algorithms in the CONGEST model (add unique identifiers and limit the size of the messages, see Chapter 6).

7.2 Probabilistic Analysis

In randomised 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 $\mathcal{F}$ in time $T(n)$ with probability p, then we can take *any* graph $G \in \mathcal{F}$ 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 emphasise 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 solutions to problem Π.

However, Monte Carlo algorithms are not as useful in the field of distributed computing as they were in the context of centralised algorithms. In centralised 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.

7.3 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 randomised 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 $c > 0$.

7.4 Algorithm BDRand: Randomised Colouring in Bounded-Degree Graphs

In Section 5.10 we presented a *deterministic* algorithm BDColour that finds a $(\Delta + 1)$-colouring in a graph of maximum degree Δ. In this section, we will design a *randomised* algorithm BDRand that solves the same problem. The running times are different:

- BDColour runs in $O(\Delta^2 + \log^* n)$ rounds.
- BDRand runs in $O(\log n)$ rounds with high probability.

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

7.4.1 Algorithm Idea

A running time of $O(\log n)$ is very typical for a randomised distributed algorithm. Often randomised 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 neighbours, 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 randomised path-colouring 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 colour 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. In algorithm BDRand, 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 colour among those colours that are not yet used by their neighbours.

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

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

chance of picking a colour that is not used by any of the neighbours. However, in algorithm BDRand on average only $d/2$ neighbours are active. If we have at most $d/2$ active neigh-

bours, we will succeed in picking a free colour with probability at least

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

regardless of what the active neighbours do.

7.4.2 Algorithm

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

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

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

In the algorithm, node u maintains the following variables:

- State $s(u) \in \{0, 1\}$
- Colour $c(u) \in \{\bot\} \cup C(u)$.

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

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 colours*.
- With probability $1/2$, set $c(u) \leftarrow \perp$; 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 $c(u) \leftarrow \perp$.
  - 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 $c(u) = \perp$, or it decides to be *active* and picks a random colour $c(u) \in F(u)$. Here $F(u)$ is the set of colours that are not yet used by any of the neighbours that are stopped.
- When $s(u) = 0$, the node *verifies* its choice. If the current colour $c(u)$ conflicts with one of the neighbours, we go back to the initial state $s(u) \leftarrow 1$ and $c(u) \leftarrow \perp$. However, if we were lucky and managed to pick a colour that does not conflict with any of our neighbours, we keep the current value of $c(u)$ and switch to the stopping state.

7.4.3 Analysis

It is easy to see that if the algorithm stops, then the output is a proper $(\Delta + 1)$-colouring of the underlying graph. Let us now analyse 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 neighbours 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 colour that does not conflict with any of its neighbours.

Lemma 7.1. *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 colours 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 neighbours.

Let us first study the case that u is active. As we have got f free colours, for any given colour $x \in \mathbb{N}$ we have

$$\Pr\big[c_t(u) = x \;\big|\; c_t(u) \neq \bot\big] \leq 1/f.$$

In particular, this holds for any colour $x = c_t(v)$ chosen by any active competitor $v \in K_t(u)$:

$$\Pr\big[c_t(u) = c_t(v) \;\big|\; c_t(u) \neq \bot,\ c_t(v) \neq \bot\big] \leq 1/f.$$

That is, we conflict with an active competitor with probability at most $1/f$. Naturally, we cannot conflict with a passive competitor:

$$\Pr\big[c_t(u) = c_t(v) \;\big|\; c_t(u) \neq \perp,\ c_t(v) = \perp\big] = 0.$$

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

$$\Pr\big[c_{t+1}(u) \neq \perp\big] =$$
$$\Pr\big[c_t(u) \neq \perp \text{ and } c_t(u) \neq c_t(v) \text{ for all } v \in K_t(u)\big] > 1/4. \quad \square$$

Now we can continue with the same argument as what we used in Section 1.5 to analyse the running time. Fix a constant $c > 0$. Define

$$T(n) = 2(c+1)\log_{4/3} n.$$

We will prove that algorithm BDRand 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 7.2. *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 7.1, 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

$$(3/4)^{T(n)/2} = \frac{1}{(4/3)^{(c+1)\log_{4/3} n}} = \frac{1}{n^{c+1}}. \qquad \square$$

Now we are ready to analyse the time until all nodes stop.

Theorem 7.3. *The probability that all nodes have stopped after $T(n)$ rounds is at least $1 - 1/n^c$.*

Proof. Follows from Lemma 7.2 by the union bound. $\square$

Note that $T(n) = O(\log n)$ for any constant c. Hence we conclude that algorithm BDRand stops in $O(\log n)$ rounds with high probability, and when it stops, it outputs a vertex colouring with $\Delta + 1$ colours.

7.5 Exercises

Exercise 7.1 (larger palette)**.** Assume that we have a graph without any isolated nodes. We will design a graph-colouring algorithm A that is a bit easier to understand and analyse than algorithm BDRand. In algorithm A, each node u proceeds as follows until it stops:

- Node u picks a colour $c(u)$ 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 neighbours. If $c(u)$ is different from the values of its neighbours, u outputs $c(u)$ and stops.

Present this algorithm in a formally precise manner, using the state-machine formalism. Analyse the algorithm, and prove that it finds a 2Δ-colouring in time $O(\log n)$ with high probability.

Exercise 7.2 (unique identifiers)**.** Design a randomised PN algorithm A that solves the following problem in $O(1)$ rounds:

- As input, all nodes get value $|V|$.
- Algorithm outputs a labelling $f : V \to \{1, 2, \ldots, \chi\}$ for some $\chi = |V|^{O(1)}$.
- With high probability, $f(u) \neq f(v)$ for all nodes $u \neq v$.

Analyse 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 O*

Exercise 7.3 (large independent sets)**.** Design a randomised 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 P*

Exercise 7.4 (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 Q*

Exercise 7.5 (max cut algorithm)**.** Design a randomised 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 R*

Exercise 7.6 (maximal independent sets)**.** Design a randomised PN algorithm that finds a maximal independent set in time $O(\Delta + \log n)$ with high probability.

▷ *hint S*

⋆ **Exercise 7.7** (maximal independent sets quickly)**.** Design a randomised distributed algorithm that finds a maximal independent set in time $O(\log n)$ with high probability.

▷ *hint T*

7.6 Bibliographic Notes

Algorithm BDRand and the algorithm of Exercise 7.1 are from Barenboim and Elkin's book [4, Section 10.1].

Part IV

Proving Impossibility Results

Chapter 8

Covering Maps

Chapters 4–7 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.

8.1 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 $p'(\phi(u), i) = (\phi(v), j)$.

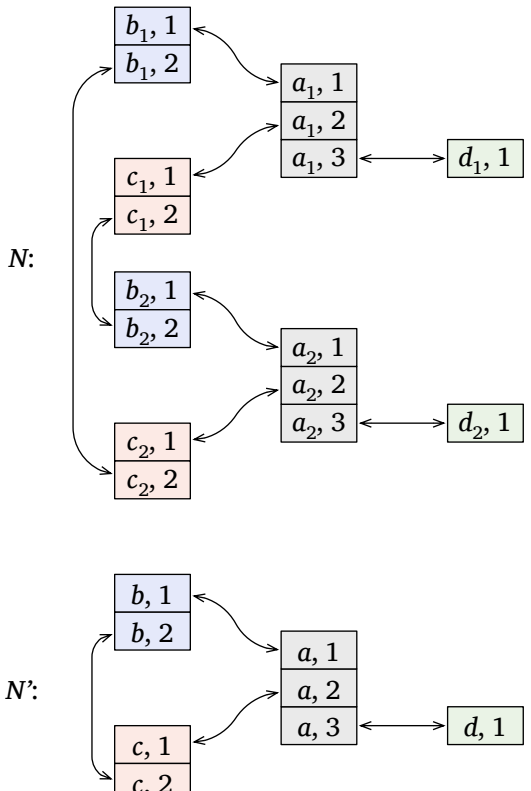


Figure 8.1: There is a covering map ϕ from N to N' that maps $a_i \mapsto a$, $b_i \mapsto b$, $c_i \mapsto c$, and $d_i \mapsto d$ for each $i \in \{1, 2\}$.

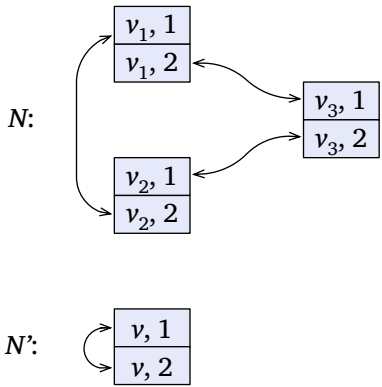


Figure 8.2: There is a covering map ϕ from N to N' that maps $v_i \mapsto v$ for each $i \in \{1, 2, 3\}$. Here N is a simple port-numbered network but N' is not.

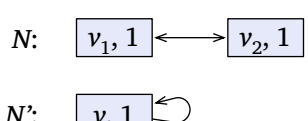


Figure 8.3: There is a covering map ϕ from N to N' that maps $v_i \mapsto v$ for each $i \in \{1, 2\}$. Again, N is a simple port-numbered network but N' is not.

See Figures 8.1–8.3 for examples.

We can also consider labelled 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 $v \in V$.

8.2 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 8.1. *Assume that*

(a) *A is a deterministic* PN*-algorithm with $X = \text{Input}_A$,*

(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 4.3.2; 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

$$x'_0(v') = \text{init}_{A,d}(f'(v')) = \text{init}_{A,d}(f(v)) = x_0(v).$$

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:

$$\text{send}_{A,\ell}(x'_{t-1}(v')) = \text{send}_{A,\ell}(x_{t-1}(v)).$$

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 $\text{send}_{A,\ell}(x_{t-1}(v))$, and $m'_t(u', i)$ is component j of $\text{send}_{A,\ell}(x'_{t-1}(v'))$. It follows that $m_t(u, i) = m'_t(u', i)$ and $m_t(u) = m'_t(u')$. Therefore

$$\begin{aligned} x'_t(u') &= \text{receive}_{A,d}\big(x'_{t-1}(u'), m'_t(u')\big) \\ &= \text{receive}_{A,d}\big(x_{t-1}(u), m_t(u)\big) = x_t(u). \end{aligned}$$

□

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 $v \in V$.

8.3 Examples

We will give representative examples of negative results that we can easily derive from Theorem 8.1. 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 8.2. *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:

$$p : (v_1, 1) \mapsto (v_2, 2),\ (v_2, 1) \mapsto (v_3, 2),\ \ldots,$$
$$(v_{n-1}, 1) \mapsto (v_n, 2),\ (v_n, 1) \mapsto (v_1, 2).$$

See Figure 8.2 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 8.1, 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 8.1 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 8.2 has many immediate corollaries.

Corollary 8.3. *Let $\mathscr{F}$ be the family of cycle graphs. Then there is no deterministic* PN*-algorithm that solves any of the following problems on $\mathscr{F}$:*

(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 colouring,*
(f) *weak colouring,*
(g) *edge colouring.*

Proof. In each of these cases, there is a graph $G \in \mathcal{F}$ such that a constant function is not a feasible solution in the network N that we constructed in Lemma 8.2.

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 8.4. *There is no deterministic* PN*-algorithm that finds a weak colouring for any* 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 8.1.

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 8.4. 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 8.2; 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 8.4.

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 8.4. 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 colouring 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-factorisation 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-factorisation 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.

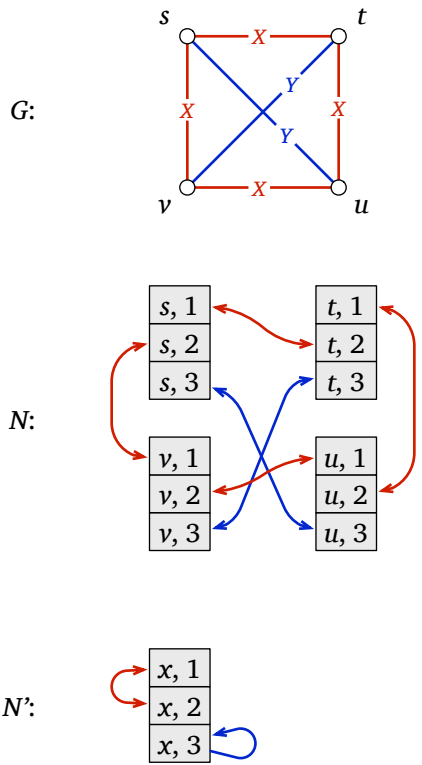


Figure 8.4: Graph G is the complete graph on four nodes. The edges of G can be partitioned into a 2-factor X and a 1-factor Y. Network N has G as its underlying graph, and there is a covering map ϕ from N to N'

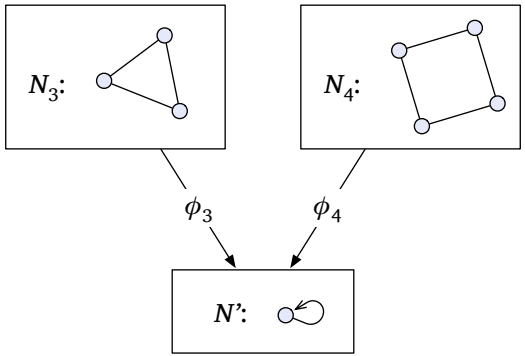


Figure 8.5: The structure of the proof of Lemma 8.5.

Lemma 8.5. *Let $\mathscr{F} = \{ G_3, G_4 \}$, 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 $\mathscr{F}$: 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 8.2; 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 8.2, 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 8.5. The local inputs are assumed to be all zeroes.

Let A be a PN-algorithm, and let c be the output of the only node of N'. If we apply Theorem 8.1 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 8.1 to ϕ_4, we learn that all nodes of N_4 also output $c = 3$, and hence A cannot solve Π on $\mathscr{F}$. □

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.

8.4 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 8.2 shows that all cycle graphs are homogeneous.

Exercise 8.1 (finding port numbers)**.** Consider the graph G and network N' illustrated in Figure 8.6. 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 8.2 (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.

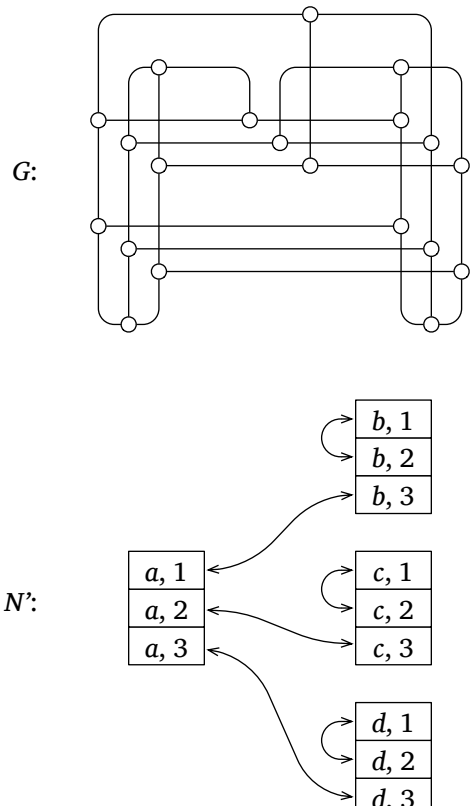


Figure 8.6: Graph G and network N' for Exercises 8.1 and 8.3b.

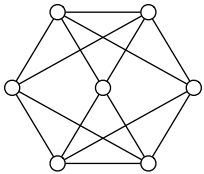


Figure 8.7: Graph G for Exercise 8.3a.

Exercise 8.3 (regular and homogeneous)**.** Show that the following graphs are homogeneous:

(a) graph G illustrated in Figure 8.7,

(b) graph G illustrated in Figure 8.6.

▷ *hint U*

Exercise 8.4 (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-factorisation,

(c) any complete graph is homogeneous.

Exercise 8.5 (dominating sets)**.** Let $\Delta \in \{2, 3, \dots\}$, let $\epsilon > 0$, and let $\mathscr{F}$ 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 $\mathscr{F}$ with a deterministic PN-algorithm. Show that it is not possible to find a $(\Delta + 1 - \epsilon)$-approximation with a deterministic PN-algorithm.

▷ *hint V*

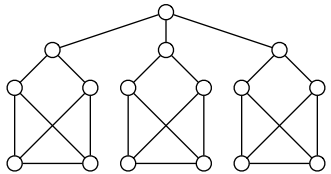


Figure 8.8: Graph G for Exercise 8.7.

Exercise 8.6 (covers with covers)**.** What is the connection between covering maps and algorithm VC3 of Section 4.6?

⋆ **Exercise 8.7** (3-regular and not homogeneous)**.** Consider the graph G illustrated in Figure 8.8.

(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 W*

⋆ **Exercise 8.8** (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 8.9** (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.

8.5 Bibliographic Notes

The use of covering maps in the context of distributed algorithm was introduced by Angluin [2]. The general idea of Exercise 8.7 can be traced back to Yamashita and Kameda [27], while the specific construction in Figure 8.8 is from Bondy and Murty's textbook [5, Figure 5.10]. Parts of exercises 8.1, 8.3, 8.4, and 8.5 are inspired by our work [3, 25].

Chapter 9

Local Neighbourhoods

Covering maps can be used to argue that a given problem cannot be solved at all with deterministic PN algorithms. Now we will revisit the concept of *locality* that we already studied in Chapter 2. Locality can be used to argue that a given problem cannot be solved fast, in any model of distributed computing.

9.1 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 neighbourhoods

$$U = \mathrm{ball}_G(v, r), \quad U' = \mathrm{ball}_{G'}(v', r).$$

We say that (N, f, v) and (N', f', v') have *isomorphic radius-r neighbourhoods* 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 $v \in U$.

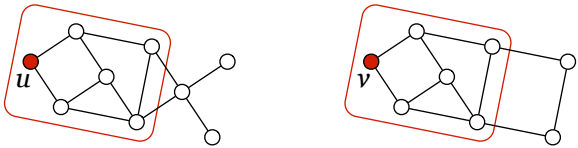


Figure 9.1: Nodes u and v have isomorphic radius-2 neighbourhoods, provided that we choose the port numbers appropriately. Therefore in any algorithm A the state of u equals the state of v at time $t = 0, 1, 2$. However, at time $t = 3, 4, \ldots$ this does not necessarily hold.

(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-neighbourhood isomorphism from* (N, f, v) *to* (N', f', v'). See Figure 9.1 for an example.

9.2 Local Neighbourhoods and Executions

Theorem 9.1. *Assume that*

(a) *A is a deterministic* PN *algorithm with $X = \text{Input}_A$,*

(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 neighbourhoods.*

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 neighbourhood 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 9.1 in the LOCAL model, we need to include unique identifiers in the local inputs f and f'.

9.3 Exercises

Exercise 9.1 (edge colouring)**.** In this exercise, the graph family $\mathcal{F}$ consists of *path graphs*.

(a) Show that it is possible to find a 2-edge colouring in time $O(n)$ with deterministic PN-algorithms.

(b) Show that it is not possible to find a 2-edge colouring in time $o(n)$ with deterministic PN-algorithms.

(c) Show that it is not possible to find a 2-edge colouring in time $o(n)$ with deterministic LOCAL-algorithms.

Exercise 9.2 (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 9.3 (optimisation)**.** 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 9.4 (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 9.5 (auxiliary information)**.** In this exercise, the graph family $\mathscr{F}$ consists of *path graphs*, and we are given a 4-colouring as input. We consider deterministic PN-algorithms.

(a) Show that it is possible to find a 3-colouring in time 1.

(b) Show that it is not possible to find a 3-colouring in time 0.

(c) Show that it is possible to find a 2-colouring in time $O(n)$.

(d) Show that it is not possible to find a 2-colouring in time $o(n)$.

⋆ **Exercise 9.6** (orientations)**.** In this exercise, the graph family $\mathcal{F}$ 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 9.7** (local indistinguishability)**.** Consider the graphs G_1 and G_2 illustrated in Figure 9.2. 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 X*

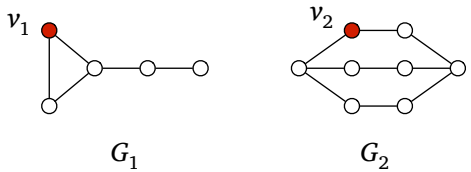


Figure 9.2: Graphs for Exercise 9.7.

9.4 Bibliographic Notes

Local neighbourhoods were used to prove negative results in the context of distributed computing by, e.g., Linial [15].

Chapter 10

Ramsey Theory

In this chapter we will prove Ramsey's theorem, which is a mathematical statement with numerous applications. This chapter is pure combinatorics; we will not discuss distributed algorithms at all. In Chapter 11 we will then see how to apply Ramsey's theorem to prove negative results related to distributed algorithms.

10.1 Monochromatic Subsets

Let Y be a finite set. We say that X is a *k-subset* of Y if $X \subseteq Y$ and $|X| = k$. We use the notation

$$Y^{(k)} = \{X \subseteq Y : |X| = k\}$$

for the collection of all k-subsets of Y.

A *c-labelling* of $Y^{(k)}$ is an arbitrary function

$$f : Y^{(k)} \to \{1, 2, \ldots, c\}.$$

Fix some Y, k, c, and f, where f is a c-labelling of $Y^{(k)}$. We say that

(a) $X \subseteq Y$ is *monochromatic in f* if $f(A) = f(B)$ for all $A, B \in X^{(k)}$,

$f : Y^{(2)} \to \{1,2,3\}$	
$\{1,2\} \mapsto 1$	$\{2,4\} \mapsto 1$
$\{1,3\} \mapsto 1$	$\{2,5\} \mapsto 2$
$\{1,4\} \mapsto 2$	$\{3,4\} \mapsto 3$
$\{1,5\} \mapsto 1$	$\{3,5\} \mapsto 3$
$\{2,3\} \mapsto 2$	$\{4,5\} \mapsto 3$

Figure 10.1: In this example, $Y = \{1,2,3,4,5\}$. Function f is a 3-labelling of $Y^{(2)}$. Set $\{1,2,3,5\}$ is almost monochromatic but not monochromatic in f. Set $\{3,4,5\}$ is both almost monochromatic and monochromatic in f.

(b) $X \subseteq Y$ is *almost monochromatic in* f if $f(A) = f(B)$ for all $A, B \in X^{(k)}$ with $\min(A) = \min(B)$.

See Figure 10.1 for examples. Monochromatic subsets are a central concept in Ramsey theory, while almost monochromatic subsets are a technical definition that we will use in the proof.

10.2 Ramsey Numbers

For all positive integers c, n, and k, we define the numbers $R_c(n;k)$ and $\bar{R}_c(n;k)$ as follows.

(a) $R_c(n;k)$ is the smallest natural number N such that the following holds: for any set Y with at least N elements, and for any c-labelling f of $Y^{(k)}$, there is an n-subset of Y that is monochromatic in f. If no such N exists, $R_c(n;k) = \infty$.

(b) $\bar{R}_c(n;k)$ is the smallest natural number N such that the following holds: for any set Y with at least N elements, and for any c-labelling f of $Y^{(k)}$, there is an n-subset of Y that is almost monochromatic in f. If no such N exists, $\bar{R}_c(n;k) = \infty$.

Numbers $R_c(n;k)$ are called *Ramsey numbers*, and Ramsey's theorem shows that they are always finite.

Theorem 10.1 (Ramsey's theorem)**.** *Numbers $R_c(n;k)$ are finite for all positive integers c, n, and k.*

We will prove Theorem 10.1 in Section 10.4; let us first have a look at an application.

10.3 An Application

In the case of $k = 2$, Ramsey's theorem can be used to derive various graph-theoretic results. As a simple application, we can use Ramsey's theorem to prove that sufficiently large graphs necessarily contain large cliques or large independent sets.

Let $G = (V, E)$ be a graph. Recall that an *independent set* is a subset $X \subseteq V$ such that $\{u, v\} \notin E$ for all $\{u, v\} \in X^{(2)}$. A complementary concept is a *clique*: it is a subset $X \subseteq V$ such that $\{u, v\} \in E$ for all $\{u, v\} \in X^{(2)}$.

Lemma 10.2. *For any natural number n there is a natural number N such that the following holds: if $G = (V, E)$ is a graph with at least N nodes, then G contains a clique with n nodes or an independent set with n nodes.*

Proof. Choose an integer $N \geq R_2(n;2)$; by Theorem 10.1, such an N exists.

Now if $G = (V, E)$ is any graph with at least N nodes, we can define a 2-labelling f of $V^{(2)}$ as follows:

$$f(\{u,v\}) = \begin{cases} 1 & \text{if } \{u,v\} \in E, \\ 2 & \text{if } \{u,v\} \notin E. \end{cases}$$

By the definition of Ramsey numbers, if $|V| \geq N$, there is an n-subset $X \subseteq V$ that is monochromatic in f. If $X \subseteq V$ is monochromatic, we have one of the following cases:

(a) we have $f(\{u,v\}) = 1$ for all $\{u,v\} \in X^{(2)}$; therefore X is a clique,

(b) we have $f(\{u,v\}) = 2$ for all $\{u,v\} \in X^{(2)}$; therefore X is an independent set. □

10.4 Proof

Let us now prove Theorem 10.1. Throughout this section, let c be fixed. We will show that $R_c(n;k)$ is finite for all n and k. The proof outline is as follows:

(a) Lemma 10.3: $R_c(n;1)$ is finite for all n.

(b) Corollary 10.7: if $R_c(n;k-1)$ is finite for all n, then $R_c(n;k)$ is finite for all n.

Here we will use the following auxiliary results:

(i) Lemma 10.5 — if $R_c(n;k-1)$ is finite for all n, then $\bar{R}_c(n;k)$ is finite for all n.

(ii) Lemma 10.6 — if $\bar{R}_c(n;k)$ is finite for all n, then $R_c(n;k)$ is finite for all n.

(c) Now by induction on k, it follows that $R_c(n;k)$ is finite for all n and k.

The base case of $k = 1$ is, in essence, equal to the familiar pigeonhole principle.

Lemma 10.3. *Ramsey number $R_c(n;1)$ is finite for all n.*

Proof. Let $N = c(n-1)+1$. We can use the pigeonhole principle to show that $R_c(n;1) \leq N$.

Let Y be a set with at least N elements, and let f be a c-labelling of $Y^{(1)}$. In essence, we have c boxes, labelled with $\{1,2,\ldots,c\}$, and function f places each element of Y into one of these boxes. As there are N elements, there is a box that contains at least

$$\lceil N/c \rceil = n$$

elements. These elements form a monochromatic subset. □

Let us now study the case of $k > 1$. We begin with a technical lemma.

Lemma 10.4. *Let n and k be integers, $n > k > 1$. If $M = \bar{R}_c(n-1;k)$ and $R_c(M;k-1)$ are finite, then $\bar{R}_c(n;k)$ is finite.*

Proof. Define

$$N = 1 + R_c(M;k-1).$$

We will prove that $\bar{R}_c(n;k) \le N$.

Let Y be a set with N elements; w.l.o.g., we can assume that $Y = \{1,2,\dots,N\}$. Let f be any c-labelling of $Y^{(k)}$. We need to show that there is an almost monochromatic n-subset $W \subseteq Y$.

To this end, let $Y_2 = \{2,3,\dots,N\}$, and define a c-labelling f_2 of $Y_2^{(k-1)}$ as follows; see Figure 10.2 for an illustration:

$$f_2(A) = f(\{1\} \cup A) \text{ for each } A \in Y_2^{(k-1)}.$$

Now f_2 is a c-labelling of $Y_2^{(k-1)}$, and Y_2 contains

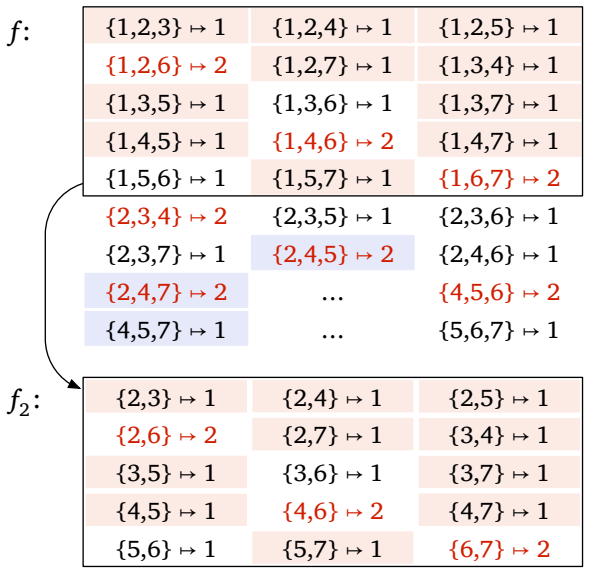
$$N-1 = R_c(M;k-1)$$

elements. Hence, by the definition of Ramsey numbers, there is an M-subset $X_2 \subseteq Y_2$ that is monochromatic in f_2.

Function f is a c-labelling of $Y^{(k)}$, and $X_2 \subseteq Y$. Hence by restriction f defines a c-labelling of $X_2^{(k)}$. Set X_2 contains $M = \bar{R}_c(n-1;k)$ elements. Therefore there is an $(n-1)$-subset $W_2 \subseteq X_2$ that is almost monochromatic in f.

To conclude the proof, let $W = \{1\} \cup W_2$. By construction, W contains n elements. Moreover, W is almost monochromatic in f. To see this, assume that $A, B \subseteq W$ are k-subsets such that $\min(A) = \min(B)$. We need to show that $f(A) = f(B)$. There are two cases:

(a) We have $\min(A) = \min(B) = 1$. Let $A_2 = A \setminus \{1\}$ and $B_2 = B \setminus \{1\}$. Now A_2 and B_2 are $(k-1)$-subsets of X_2. Set X_2 was monochromatic in f_2, and hence $f(A) = f_2(A_2) = f_2(B_2) = f(B)$.



$X_2 = \{2,3,4,5,7\}$, monochromatic in f_2

$W_2 = \{2,4,5,7\}$, almost monochromatic in f

$W = \{1,2,4,5,7\}$, almost monochromatic in f

Figure 10.2: The proof of Lemma 10.4, for the case of $c = 2$, $k = 3$, and $n = 5$, assuming completely fictional values $M = 5$ and $N = 7$.

(b) Otherwise $1 \notin A$ and $1 \notin B$. Now A and B are k-subsets of W_2. Set W_2 was almost monochromatic in f, and we have $\min(A) = \min(B)$, which implies $f(A) = f(B)$. □

Lemma 10.5. *Let $k > 1$ be an integer. If $R_c(n; k-1)$ is finite for all n, then $\bar{R}_c(n; k)$ is finite for all n.*

Proof. The proof is by induction on n.

The base case of $n \leq k$ is trivial: a set with n elements has at most one subset with k elements, and hence it is almost monochromatic and monochromatic.

Now let $n > k$. Inductively assume that $\bar{R}_c(n-1; k)$ is finite. Recall that in the statement of this lemma, we assumed that $R_c(M; k-1)$ is finite for any M; in particular, it is finite for $M = \bar{R}_c(n-1; k)$. Hence we can apply Lemma 10.4, which implies that $\bar{R}_c(n; k)$ is finite. □

Lemma 10.6. *Let $k > 1$ be an integer. If $\bar{R}_c(n; k)$ is finite for all n, then $R_c(n; k)$ is finite for all n.*

Proof. Let $M = R_c(n; 1)$. By Lemma 10.3, M is finite. By assumption, $\bar{R}_c(M; k)$ is also finite. We will show that

$$R_c(n; k) \leq \bar{R}_c(M; k).$$

Let Y be a set with $N = \bar{R}_c(M; k)$ elements, and let f be any c-labelling of $Y^{(k)}$. We need to show that there is a monochromatic n-subset $W \subseteq Y$.

By definition, there is an almost monochromatic M-subset $X \subseteq Y$. Hence we can define a c-labelling g of $X^{(1)}$ such that

$$g(\{\min(A)\}) = f(A)$$

f	g
$\{1,2\} \mapsto 1$ $\{1,3\} \mapsto 1$ $\{1,4\} \mapsto 1$	$\{1\} \mapsto 1$
$\{2,3\} \mapsto 3$ $\{2,4\} \mapsto 3$	$\{2\} \mapsto 3$
$\{3,4\} \mapsto 2$	$\{3\} \mapsto 2$
	$\{4\} \mapsto 1$

Figure 10.3: The proof of Lemma 10.6. In this example, $c = 3$, $k = 2$, and $X = \{1,2,3,4\}$ is almost monochromatic in f. We define a c-labelling g of $X^{(1)}$ such that $g(\{\min(A)\}) = f(A)$ for all $A \in X^{(2)}$. Note that the choice of $g(4)$ is arbitrary.

for each k-subset $A \subseteq X$; see Figure 10.3. As X is a subset with $M = R_c(n;1)$ elements, we can find an n-subset $W \subseteq X$ that is monochromatic in g.

Now we claim that W is also monochromatic in f. To see this, let A and B be k-subsets of W. Let $x = \min(A)$ and $y = \min(B)$. We have $x, y \in W$ and

$$f(A) = g(\{x\}) = g(\{y\}) = f(B). \qquad \square$$

Lemmas 10.5 and 10.6 have the following corollary.

Corollary 10.7. *Let $k > 1$ be an integer. If $R_c(n;k-1)$ is finite for all n, then $R_c(n;k)$ is finite for all n.*

Now Ramsey's theorem follows by induction on k: the base case is Lemma 10.3, and the inductive step is Corollary 10.7.

10.5 Exercises

Exercise 10.1. Prove that $R_c(n;1) = c \cdot (n-1) + 1$.

▷ *hint Y*

Exercise 10.2. Prove that $R_2(3;2) \geq 6$.

▷ *hint Z*

Exercise 10.3. Prove that $R_2(3;2) \leq 6$. Together with the previous exercise, this will show that $R_2(3;2) = 6$.

Exercise 10.4. Prove a non-trivial lower bound on $R_2(4;2)$. For example, show that $R_2(4;2) \geq 10$.

Exercise 10.5. Prove some concrete upper bound on $R_2(4;2)$. For example, show that $R_2(4;2) \leq 100$.

⋆ **Exercise 10.6.** Find the exact value of $R_2(4;2)$.

10.6 Bibliographic Notes

Ramsey's theorem dates back to 1930s [24]; our proof follows Nešetřil [17], and the notation is from Radziszowski [23].

Chapter 11

Applications of Ramsey's Theorem

In Section 2.3, we gave a proof of the following result: in the LOCAL model, it is not possible to find a 3-colouring of a directed cycle in $O(1)$ rounds with deterministic algorithms. In this chapter, we will give another proof of the same result, this time with the help of Ramsey's theorem (recall Chapter 10). In the exercises, we will see plenty of other applications of Ramsey's theorem in this context, some of which would be rather difficult to prove with the technique that we used in Section 2.3.

11.1 Claim

We will prove the following theorem.

Theorem 11.1. *Assume that A is a deterministic distributed algorithm for the* LOCAL *model. Assume that there is a constant $T \in \mathbb{N}$ such that A stops in time T in any directed cycle $G = (V, E)$, and outputs a labelling $g : V \to \{1, 2, 3\}$. Then there exists a directed cycle G and an assignment of unique identifiers such that if we execute A on G, the output of A is not a proper vertex colouring of G.*

11.2 Preliminaries

To prove Theorem 11.1, let $n = 2T+2$, $k = 2T+1$, and $c = 3$. By Ramsey's theorem, $R_c(n;k)$ is finite. Choose any $N \geq R_c(n;k)$.

We will construct a directed cycle $G = (V, E)$ with N nodes. In our construction, the set of nodes is $V = \{1, 2, \ldots, N\}$. This is also the set of unique identifiers in our cycle; recall that we follow the convention that the unique identifier of a node $v \in V$ is v.

With the set of nodes fixed, we proceed to define the set of edges. In essence, we only need to specify in which order the nodes are placed along the cycle.

11.3 Subsets and Cycles

For each subset $X \subseteq V$, we define a directed cycle $G_X = (V, E_X)$ as follows; see Figure 11.1. Let $\ell = |X|$. Label the nodes by $x_1, x_2, \ldots, x_N$ such that

$$\begin{aligned} X &= \{ x_1, x_2, \ldots, x_\ell \}, \\ V \setminus X &= \{ x_{\ell+1}, x_{\ell+2}, \ldots, x_N \}, \\ x_1 &< x_2 < \cdots < x_\ell, \\ x_{\ell+1} &< x_{\ell+2} < \cdots < x_N. \end{aligned}$$

Then choose the edges

$$E_X = \{ (x_i, x_{i+1}) : 1 \leq i < N \} \cup \{ (x_N, x_1) \}.$$

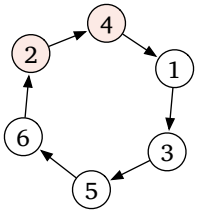


Figure 11.1: Construction of G_X. Here $N = 6$ and $X = \{2, 4\}$.

Informally, G_X is constructed as follows: first take all nodes of X, in the order of increasing identifiers, and then take all other nodes, again in the order of increasing identifiers.

11.4 Labelling

If $B \subseteq V$ is a k-subset, then we define that the *internal node* $i(B)$ is the median of the set B. Put otherwise, $i(B)$ is the unique node in B that is not among the T smallest nodes of B, nor among the T largest nodes of B.

We will use algorithm A to construct a c-labelling f of $V^{(k)}$ as follows. For each k-subsets $B \subseteq V$, we construct the cycle G_B, execute A on G_B, and define that $f(B)$ is the output of node $i(B)$ in G_B. See Figure 11.2 for an illustration.

11.5 Monochromatic Subsets

We have constructed a certain c-labelling f. As N is sufficiently large, there exists an n-subset $X \subseteq V$ that is monochromatic

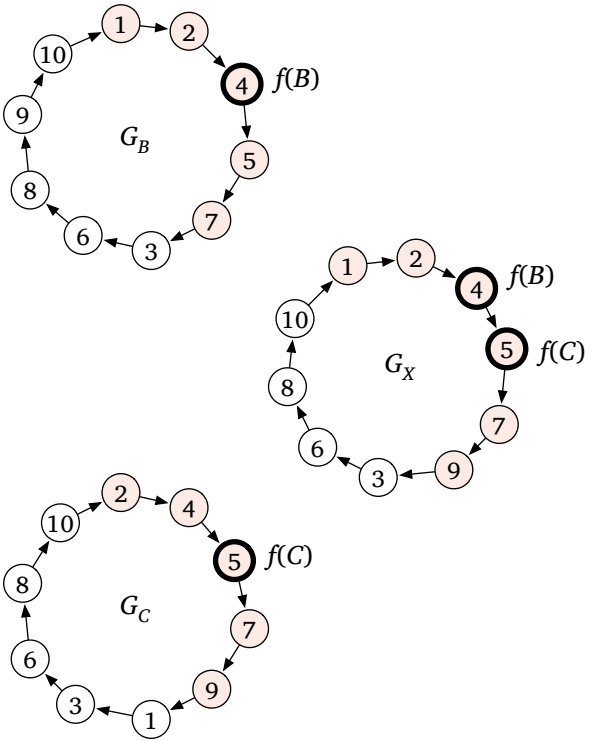


Figure 11.2: In this example, $N = 10$ and $T = 2$. Let $B = \{1, 2, 4, 5, 7\}$, $C = \{2, 4, 5, 7, 9\}$, and $X = \{1, 2, 4, 5, 7, 9\}$. The label $f(B)$ is defined as follows: we construct G_B, execute algorithm A, and take the output of the internal node $i(B) = 4$. Similarly, the label $f(C)$ is the output of node $i(C) = 5$ in G_C. As the local neighbourhoods are identical, the output of node 4 in G_X is also $f(B)$, and the output of node 5 in G_X is also $f(C)$. If X is monochromatic in f, we have $f(B) = f(C)$.

in f. Let us label the nodes of X by

$$X = \{x_0, x_1, \ldots, x_k\},$$

where $x_0 < x_1 < \cdots < x_k$. Let

$$B = \{x_0, x_1, \ldots, x_{k-1}\},$$
$$C = \{x_1, x_2, \ldots, x_k\}.$$

See Figure 11.2 for an illustration.

Sets B and C are k-subsets of X, and their internal nodes are $i(B) = x_T$ and $i(C) = x_{T+1}$. As X is monochromatic, we have $f(B) = f(C)$. Therefore we know that the output of x_T in G_B equals the output of x_{T+1} in G_C.

Moreover, node x_T has isomorphic radius-T neighbourhoods in G_B and G_X; in both graphs, the radius-T neighbourhood of node x_T is a directed path, along which we have the nodes $x_0, x_1, \ldots, x_{k-1}$ in this order. Hence by Theorem 9.1, the output of x_T in G_B equals the output of x_T in G_X.

A similar argument shows that the output of x_{T+1} in G_C equals the output of x_{T+1} in G_X. In summary, the output of x_T in G_X equals $f(B)$, which equals $f(C)$, which equals the output of x_{T+1} in G_X.

We have shown that in the directed cycle G_X, there are two adjacent nodes, x_T and x_{T+1}, that produce the same output. Hence A does not output a proper vertex colouring in G_X.

11.6 Exercises

Exercise 11.1 (Ramsey and cycles)**.** You are given a constant $\ell = O(1)$. Let A be any deterministic distributed algorithm in the LOCAL model such that:

- $\text{Output}_A = \{1, 2, \ldots, c\}$ for a constant $c = O(1)$,
- the running time of A is bounded by a constant $T = O(1)$.

Prove: There exists a cycle G of diameter larger than ℓ, and an assignment of unique identifiers in G such that for some node v, all nodes in the radius-ℓ neighbourhood of v output the same value.

Exercise 11.2 (impossibility in cycles)**.** Use the result of Exercise 11.1 to prove that there is no deterministic constant-time algorithm that solves any of the following problems in the LOCAL model in the family of cycle graphs:

(a) Vertex colouring with $O(1)$ colours.
(b) Edge colouring with $O(1)$ colours.
(c) Weak colouring with $O(1)$ colours.
(d) Maximal independent set.
(e) Maximal matching.
(f) Minimal dominating set.
(g) Minimal edge dominating set.

Exercise 11.3 (Ramsey and 4-regular graphs)**.** In Exercise 11.1, we considered cycles, i.e., connected 2-regular graphs. Generalise the result to connected 4-regular graphs.

▷ *hint AA*

Exercise 11.4 (impossibility in 4-regular graphs)**.** Using the result of Exercise 11.3, generalise the result of Exercise 11.2 to 4-regular graphs.

Exercise 11.5 (impossibility in 3-regular graphs)**.** Prove that there is no deterministic constant-time algorithm that finds a vertex colouring with $O(1)$ colours in the LOCAL model in the family of 3-regular graphs.

▷ *hint AB*

⋆ **Exercise 11.6** (positive results in 3-regular graphs)**.** Prove that there is a deterministic constant-time algorithm that finds a weak colouring with $O(1)$ colours in the LOCAL model in the family of 3-regular graphs.

⋆ **Exercise 11.7** (impossibility of approximations)**.** Prove that there is no deterministic constant-time algorithm that finds an $O(1)$-approximation of a maximum independent set in any cycle in the LOCAL model.

▷ *hint AC*

11.7 Bibliographic Notes

Ramsey's theorem has been used to prove lower bounds on distributed algorithms by, e.g., Naor and Stockmeyer [16] and Czygrinow et al. [7]. In particular, the idea of Exercise 11.7 is from Czygrinow et al. [7]. Exercise 11.6 is a special case of the classical result by Naor and Stockmeyer [16].

Part V

Conclusions

Chapter 12

Conclusions

We have reached the end of this book. 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 project in which we will analyse one graph problem — edge dominating sets — from the perspective of distributed algorithms, and apply many of the techniques that we have learned in this book.

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 parallelised 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*.

Learning Objectives. Let us now have a more detailed look of the detailed learning objectives. The following is based on the scope of the lecture course *Distributed Algorithms* that I lecture at Aalto University. This is a 12-week course, with one 2-hour lecture and one 2-hour exercise session per week, worth 5 ECTS credits, which roughly translates to 133 hours of work in total. Roughly speaking, normal exercises are something that the students should be able to solve, while the exercises marked with a star are material that goes beyond the learning objectives.

Objective 1: Models. As the title of this book suggests, by now you should know precisely what is a *distributed algorithm*. You can now define in a formally precise manner what is a distributed algorithm in each of the following models:

- deterministic PN-algorithms (Chapter 4),
- deterministic LOCAL-algorithms (Chapter 5),
- deterministic CONGEST-algorithms (Chapter 6),
- randomised PN-algorithms (Chapter 7),
- randomised LOCAL-algorithms (Chapter 7),
- randomised CONGEST-algorithms (Chapter 7).

Objective 2: Algorithms. You have now learned how to *design* a distributed algorithm in each of these models, how to *prove* that the algorithm is correct, and how to *analyse* the running time of the algorithm.

We have seen many concrete examples of distributed algorithms. At least, you should be familiar with the following key examples:

- Colouring paths in $O(\log^* n)$ rounds with deterministic algorithms in the LOCAL model (Section 1.4).
- Colouring graphs in $O(\log n)$ rounds w.h.p. with randomised algorithms in the LOCAL model (Section 7.4).
- Gathering everything in $O(\mathrm{diam}(G))$ rounds in the LOCAL model (Section 5.2).
- Maximal matching in bipartite graphs in $O(\Delta)$ rounds in the PN model (Section 4.5).

- All-pairs shortest paths in $O(n)$ rounds in the CONGEST model (Section 6.7).

These algorithm highlight the key aspect of the models of computing, they illustrate important techniques for algorithm design and analysis, and they also serve as key building blocks that can be used as subroutines in many other algorithms.

The use of other algorithms as subroutines is one of the key algorithm design techniques in distributed computing. More formally, we employ *reductions*: to solve a problem X, you first show that given a solution to another problem Y, you can easily solve problem X, too. Then it is sufficient to find an algorithm for solving problem Y; in many cases, you can simply reuse an existing algorithm.

Graph colouring is a prime example of the power of reductions: given an efficient distributed algorithm for graph colouring, we can also solve many other problems efficiently. A graph colouring helps with *symmetry breaking* and a graph colouring makes it easier to *coordinate* or *schedule* the activities of the nodes in a conflict-free manner — for example, we can proceed by colour classes, so that in step i nodes of colour i are active.

Note that in the PN model, it is impossible to find a graph colouring with a deterministic algorithm. Nevertheless, we can still use graph colourings in algorithm design even in this model: recall the vertex cover algorithm from Section 4.6, in which we were able to produce a 2-colouring out of thin air, and exploit it in algorithm design.

Objective 3: Lower Bounds. You can now also prove what *cannot be computed* with distributed algorithms, and *what cannot be computed* efficiently. There are two main arguments that we use:

- Covering maps can be used to show that many problems cannot be solved *at all* in the PN model (Chapter 8).
- Local neighbourhoods can be used to show that many problems cannot be solved *efficiently* in any of the models (Chapters 2 and 9).

For many problems these two basic tools — together with a bit of creativity in how to apply them — are all that is needed. However, there are some problems that need a more heavyweight machinery. The key example is, again, graph colouring.

While it is possible to, e.g., find a 3-colouring of a path in $O(\log^* n)$ rounds with deterministic LOCAL-algorithms, this is not possible in $O(1)$ rounds. We have now seen *two* different ways to prove this result; hopefully you are comfortable with at least one of the proofs:

(a) Section 2.3 gave a proof that is self-contained but a bit technical. This proof gives a tight result, showing that the problem cannot be solved in $o(\log^* n)$ rounds.

(b) Chapter 11 showed how to prove the claim using Ramsey's theorem. The proof itself is fairly simple, and it can be easily generalised to many other results. Unfortunately, it relies on Ramsey's theorem, which is a bit tedious to prove. Moreover, our proof is a bit sloppy; we only showed that

$O(1)$ rounds is not sufficient. To prove the stronger claim that $o(\log^* n)$ rounds is not sufficient requires more work if we want to use Ramsey's theorem.

Objective 4: Graph Theory. The theory of distributed algorithms often relies heavily on graph theory. We use graph theory to define the problems that we want to solve, we use graph theory to define the model of computing that we use, and we also use graph theory in algorithm design and analysis, as well as in lower bound proofs.

By now you should be familiar with the standard graph-theoretic terms that we introduced in Chapter 3, and you should be able to prove simple graph-theoretic results that e.g. show connections between different graph problems. This is often needed in reductions if we want to apply existing distributed algorithms in order to solve new problems.

A typical example of a graph-theoretic statement that you should be able to easily prove is the connection between maximal matchings and approximate vertex covers (Exercise 3.3). This immediately gives you a distributed algorithm that finds a 2-approximation of a minimum vertex cover, provided that you have a distributed algorithm that finds a maximal matching — and to find maximal matchings, you can once again resort to graph colourings. Such results have direct applications also outside the area of distributed algorithms.

We have also encountered two concepts that go beyond elementary graph theory. The first one is the concept of covering maps (Chapter 8); while we studied covering maps in the context of port-numbered networks, an analogous concept with

similar properties can be defined for e.g. undirected or directed graphs. The second one is Ramsey's theorem; we presented Ramsey's theorem in a very general form, but the special case of $k = 2$ has many direct graph-theoretic applications. While the proof of Ramsey's theorem is a bit tedious in the general case, you should be able to prove it without much difficulty e.g. for the special case of $c = 2$ and $k = 2$.

12.2 What Else Exists?

Distributed computing is a vast topic and so far we have merely scratched the surface. This book has focused on what is often known as *distributed graph 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.

Fault-tolerant Algorithms. In distributed systems, the nodes may fail and the communication links may be unreliable. We may e.g. want to tolerate *Byzantine failures*, in which a small number of nodes may be controlled by an adversary. Or we may want to design *self-stabilising algorithms*, in which the algorithm must work correctly, no matter what are the initial states of the nodes.

Asynchronous Algorithms. In asynchronous networks, there is no global clock that guarantees that all nodes take steps simultaneously in parallel. Communication links may have un-

predictable delays. This is not a major issue if we do not need to tolerate failures — we can apply efficient *synchronisers*. However, if the nodes may fail, it becomes impossible to distinguish between e.g. a node behind a very slow links and a node that has stopped responding.

Shared Memory. Our model 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. We have pretended that 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, 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.

Complexity Measures. For us the main complexity measure has been the number of synchronous communication rounds. Many other possibilities exist: e.g., how many bits of memory we need per node, and how many messages do we need to send in total?

High-Performance Computing. For the general public, distributed computing often refers to large-scale high-performance computing in a computer network. This includes scientific computing on grids and clusters, and volunteer computing projects. Here a key question is how to partition the computation and the data set efficiently among multiple computers; this is closely related to similar questions in traditional *parallel computing*.

Practical Aspects of Networking. This book 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 standardisation 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.

12.3 Research in Distributed Algorithms

There are two main conferences related to the theory of distributed computing:

- PODC, Symposium on Principles of Distributed Computing

  http://www.podc.org/

- DISC, 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.

12.4 Exercises

In the following exercises, we will study distributed approximation algorithms for the edge dominating set problem. We will first show that the problem is easy to approximate within factor 4 in general graphs. Then we will have a look at some special

cases, and derive tight upper and lower bounds for the approximation ratio. We use the abbreviation *MEDS* for a minimum edge dominating set. Unless otherwise mentioned, all exercises in this chapter are related to *deterministic* algorithms.

Exercise 12.1 (general case)**.** Design a PN-algorithm that finds a 4-approximation of MEDS.

▷ *hint AD*

Exercise 12.2 (2-regular, upper bounds)**.** Show that the following is possible in 2-regular graphs:

(a) finding a 3-approximation of MEDS in $O(1)$ time in the PN model

(b) finding a 2-approximation of MEDS in $O(\log^* n)$ time in the LOCAL model

(c) finding a 2-approximation of MEDS with a randomised algorithm in the PN model

⋆ **Exercise 12.3** (2-regular, lower bounds)**.** Show that the following is not possible in 2-regular graphs:

(a) finding a 2.999-approximation of MEDS in the PN model

(b) finding a 2.999-approximation of MEDS in $O(1)$ time in the LOCAL model

Exercise 12.4 (4-regular, upper bound)**.** Show that it is possible to find a 3.5-approximation of MEDS in 4-regular graphs in constant time in the PN model.

▷ *hint AE*

Exercise 12.5 (4-regular, lower bound)**.** Show that it is not possible to find a 3.499-approximation of MEDS in 4-regular graphs in the PN model.

▷ *hint AF*

Exercise 12.6 (3-regular, lower bound)**.** Show that it is not possible to find a 2.499-approximation of MEDS in 3-regular graphs in the PN model.

▷ *hint AG*

⋆ **Exercise 12.7** (3-regular, upper bound)**.** Show that it is possible to find a 2.5-approximation of MEDS in 3-regular graphs in constant time in the PN model.

▷ *hint AH*

12.5 Bibliographic Notes

Exercises 12.1–12.7 are inspired by our work [3, 25].

Index

Notation

$\lvert X \rvert$	the number of elements in set X
$f^{-1}(y)$	preimage of y, i.e., $f^{-1}(y) = \{ x : f(x) = y \}$
${}^{i}2$	power tower, ${}^{i}2 = 2^{2^{\cdot^{\cdot^{2}}}}$ with i twos
$\log^* n$	iterated logarithm, $\log^*({}^{i}2) = i$
$\deg_G(v)$	degree of node v in graph G
$\mathrm{dist}_G(u, v)$	distance between nodes u and v in G
$\mathrm{ball}_G(v, r)$	nodes that are within distance r from v in G
$\mathrm{diam}(G)$	diameter of graph G
$\mathbb{N}$	the set of natural numbers, $\{0, 1, 2, \dots\}$
$\mathbb{Z}^+$	the set of positive integers, $\{1, 2, \dots\}$
$\mathbb{R}$	the set of real numbers
$[a, b]$	set $\{x \in \mathbb{R} : a \le x \le b\}$
$Y^{(k)}$	the collection of all k-subsets of Y
$R_c(n; k)$	Ramsey numbers

Symbols

These conventions are usually followed in the choice of symbols.

α	approximation factor
χ	range of unique identifiers, $\chi = \lvert V\rvert^c$
ϕ	covering map, $\phi : V \to V'$
ψ	local isomorphism, $\psi :\ \mathrm{ball}_G(v,r) \to \mathrm{ball}_H(u,r)$
Δ	maximum degree; an upper bound of the maximum degree
Π	graph problem
$\mathscr{F}$	graph family
$\mathscr{S}$	set of feasible solutions
id	unique identifiers
A	distributed algorithm
C	vertex cover $C \subseteq V$, edge cover $C \subseteq E$
D	dominating set $D \subseteq V$, edge dominating set $D \subseteq E$
E	set of edges
G, H	graph, $G = (V, E)$
I	independent set $I \subseteq V$
M	matching $M \subseteq E$
N	port-numbered network, $N = (V, P, p)$
P	set of ports
T	running time (number of rounds)
T	tree
U	subset of nodes
V	set of nodes

c, C, d	natural number
e	edge, element of E
f, g, h	function
i, j, k, ℓ	natural number
m_t	message
m	number of edges, $m = \lvert E \rvert$
n	number of nodes, $n = \lvert V \rvert$
p	connection function, involution $p \colon P \to P$
r	natural number
s, t, u, v	node, element of V
t	time step (round), $t = 0, 1, \ldots, T$
w	walk
x_t	state

Models of Computing

PN	port-numbering model, Chapter 4.
LOCAL	networks with unique identifiers, Chapter 5.
CONGEST	bandwidth-limited networks, Chapter 6.

Algorithms

P3C	3-colouring a path. Runs in $O(n)$ rounds in the LOCAL model. Sections 1.3 and 4.4.

P3CBit	3-colouring a directed path. Runs in $O(\log^* n)$ rounds in the LOCAL model. Section 1.4.
P3CRand	3-colouring a path. Randomised algorithm, runs in $O(\log n)$ rounds with high probability. Section 1.5.
P2C	2-colouring a path. Runs in $O(n)$ rounds in the LOCAL model. Section 2.2.1.
BMM	Maximal matching in 2-coloured graphs. Runs in $O(\Delta)$ rounds in the PN model. Section 4.5.
VC3	3-approximation of a minimum vertex cover. Runs in $O(\Delta)$ rounds in the PN model. Section 4.6.
Gather	Gathering the full information on the communication network. Runs in $O(\mathrm{diam}(G))$ rounds in the LOCAL model. Section 5.2.
BDGreedy	Greedy colour reduction in bounded-degree graphs. Section 5.5.
DPGreedy	Greedy colour reduction in directed pseudoforests. Section 5.7.
DPBit	Fast colour reduction in directed pseudoforests. Section 5.8.
DPSet	Fast colour reduction in directed pseudoforests. Exercise 5.6.
DP3C	3-colouring a directed pseudoforest. Runs in $O(\log^* n)$ rounds in the LOCAL model. Section 5.9.

BDColour	$(\Delta+1)$-colouring graphs of maximum degree Δ. Runs in $O(\Delta^2+\log^* n)$ rounds in the LOCAL model. Section 5.10.
Wave	Single-source shortest paths. Runs in $O(\mathrm{diam}(G))$ rounds in the CONGEST model. Section 6.4.
BFS	Breadth-first search tree. Runs in $O(\mathrm{diam}(G))$ rounds in the CONGEST model. Section 6.5.
Leader	Leader election. Runs in $O(\mathrm{diam}(G))$ rounds in the CONGEST model. Section 6.6.
APSP	All-pairs shortest paths. Runs in $O(n)$ rounds in the CONGEST model. Section 6.7.
BDRand	$(\Delta+1)$-colouring graphs of maximum degree Δ. Randomised algorithm, runs in $O(\log n)$ rounds in the LOCAL model with high probability. Section 7.4.

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 colour reduction algorithm in each subpath. Finally, combine the solutions.

B. Design a randomised algorithm that finds a colouring with a large number of colours quickly. Then apply the technique of algorithm P3CBit to reduce the number of colours to 3 quickly.

C. Consider the following cases separately:

$$\text{(i) } \log^* x \leq 2, \quad \text{(ii) } \log^* x = 3, \quad \text{(iii) } \log^* x \geq 4.$$

In case (iii), prove that after $\log^*(x) - 3$ iterations, the number of colours 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-colouring in those parts, in time $O(\log^* d)$. Remove the nodes that are properly coloured. 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. Consider the following strategy: after each iteration, reverse the directions of the edges. Then consider two iterations of the algorithm, and observe that the new colour of a node v after *two* iterations only depends on the original colours within distance *one* from node v. Hence one communication step is enough to simulate two iterations of colour reduction.

F. Assume that D is an edge dominating set; show that you can construct a maximal matching M with $|M| \leq |D|$.

G. For the purposes of algorithm VC3, it is sufficient to know which nodes are matched in BMM — we do not need to know with whom they are matched.

H. 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.

I. You can either use algorithm BDColour as a subroutine, or you can modify the basic idea of BDColour slightly to solve these problems.

J. Solve small problem instances by brute force and focus on the case of long cycles. In a long cycle, use a graph colouring algorithm to find a 3-colouring, and then use the

3-colouring 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.

K. Adapt the basic idea of algorithm BDGreedy — find local maxima and choose appropriate colours 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 colour is a local maximum among those neighbours that have not yet stopped; once a node becomes active, it selects an appropriate colour and stops.

L. 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 colouring algorithm on G^2.

M. First, design (or look up) a greedy *centralised* 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 $\mathrm{ball}_G(v, 1)$. For more details, see a textbook on approximation algorithms, e.g., Vazirani [26].

Second, show that you can *simulate* the centralised greedy algorithm in a distributed setting. Use the algorithm of Exercise 5.4 to construct a distance-2 colouring. Prove that the following strategy is a faithful simulation of the centralised greedy algorithm:

- For each possible value $i = \Delta + 1, \Delta, \ldots, 2, 1$:
  - For each colour $j = 1, 2, \ldots, O(\Delta^2)$:
    - Pick all nodes $v \in V$ that are of colour j and that dominate i new nodes.

The key observation is that if $u, v \in V$ are two distinct nodes of the same colour, 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.

N. 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.

O. Pick the labels randomly from a sufficiently large set; this takes 0 communication rounds.

P. Each node u picks a random number $f(u)$. Nodes that are local maxima with respect to the labelling f will join I.

Q. For the last part, consider a complete graph with a sufficiently large number of nodes.

R. Each node chooses an output 0 or 1 uniformly at random and stops; this takes 0 communication rounds. To analyse the algorithm, prove that each edge is a cut edge with probability $1/2$.

S. Use algorithm BDRand.

T. Look up "Luby's algorithm".

U. (a) Apply the result of Exercise 3.8. (b) Find a 1-factor.

V. For the lower bound, use the result of Exercise 8.4c.

W. Show that if a 3-regular graph is homogeneous, then it has a 1-factor. Show that G does not have any 1-factor.

X. We need to combine the results of Theorems 8.1 and 9.1. 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 neighbourhoods; 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 neighbourhoods.

Y. The proof of Lemma 10.3 shows that

$$R_c(n;1) \leq c \cdot (n-1) + 1.$$

You need to show that

$$R_c(n;1) > c \cdot (n-1).$$

Z. Prove that $R_2(3;2) > 5$. That is, show that there is a 2-labelling of 2-subsets of a base set of size 5 such that there is no monocromatic subset of size 3.

AA. Consider 2-dimensional grids; to make it 4-regular, wrap around at borders.

AB. Consider a ladder graph that consists of two n-cycles connected by n rungs.

AC. You will need several applications of Ramsey's theorem. First, choose a (very large) space of unique identifiers. Then apply Ramsey's theorem to find a large monochromatic subset, remove the set, and repeat. This way you have partitioned *almost* all identifiers into monochromatic subsets. Each monochromatic subset is used to construct a fragment of the cycle.

AD. Use the idea of Section 4.6. Show that the edge set $M \subseteq E$ defined in (4.1) is a 4-approximation of MEDS. To this end, consider an optimal solution D^* and show that each edge of D^* is adjacent to at most 4 edges of M.

AE. Consider an algorithm that selects all edges that have port number 1 in at least one end. Derive an upper bound on the size of the solution and a lower bound on the size of an optimal solution, as a function of $|V|$.

AF. Use the construction of Exercise 8.3a.

AG. Use the construction of Exercise 8.1.

AH. Let $G = (V, E)$ be a 3-regular graph. We say that a set $D \subseteq E$ is *good* if it satisfies the following properties:

(a) D is an edge cover for G,

(b) the subgraph induced by D does not contain a path of length 3.

Put otherwise, D induces a spanning subgraph that consists of node-disjoint stars. Prove that

(a) any good set D is a 2.5-approximation of MEDS,

(b) there is a distributed algorithm that finds a good set D.

The distributed algorithm has to exploit the port numbers of the edges. One possible approach is this: First, use the port numbers to find nine matchings, $M_1, M_2, \ldots, M_9$, such that each node is incident to an edge in at least one of the sets M_i; do not worry if some edges are present in more than one matching. Then construct an edge cover D by greedily adding edges from the sets M_i; in step $i = 1, 2, \ldots, 9$ you can consider all edges of M_i in parallel. Finally, eliminate paths of length three by removing redundant edges in order to make D a good set; again, in step $i = 1, 2, \ldots, 9$ you can consider all edges of M_i in parallel.

Bibliography

[1] 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.

[2] Dana Angluin. Local and global properties in networks of processors. In *Proc. 12th Annual ACM Symposium on Theory of Computing (STOC 1980)*, pages 82–93. ACM Press, 1980. doi:10.1145/800141.804655.

[3] Matti Åstrand, Valentin Polishchuk, Joel Rybicki, Jukka Suomela, and Jara Uitto. Local algorithms in (weakly) coloured graphs, 2010. arXiv:1002.0125.

[4] Leonid Barenboim and Michael Elkin. *Distributed Graph Coloring: Fundamentals and Recent Developments*. Morgan & Claypool, 2013. doi:10.2200/S00520ED1V01Y201307DCT011.

[5] John A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. North-Holland, New York, 1976.

[6] 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.

[7] Andrzej Czygrinow, Michał Hańćkowiak, and Wojciech Wawrzyniak. Fast distributed approximations in planar graphs. In *Proc. 22nd International Symposium on Distributed Computing (DISC 2008)*, volume 5218 of *Lecture Notes in Computer Science*, pages 78–92. Springer, 2008. doi:10.1007/978-3-540-87779-0_6.

[8] Reinhard Diestel. *Graph Theory*. Springer, Berlin, 3rd edition, 2005. http://diestel-graph-theory.com/.

[9] 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)*, volume 6522 of *Lecture Notes in Computer Science*, pages 65–76. Springer, 2011. doi:10.1007/978-3-642-17679-1_6.

[10] 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.

[11] 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)*, pages 219–225. Society for Industrial and Applied Mathematics, 1998.

[12] 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), pages 355–364. ACM Press, 2012. doi:10.1145/2332432.2332504.

[13] 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)*, pages 49–58. ACM Press, 2011. doi:10.1145/1993806.1993814.

[14] Juhana Laurinharju and Jukka Suomela. Brief announcement: Linial's lower bound made easy. In *Proc. 33rd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 2014)*, pages 377–378. ACM Press, 2014. doi:10.1145/2611462.2611505. arXiv:1402.2552.

[15] Nathan Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992. doi:10.1137/0221015.

[16] Moni Naor and Larry Stockmeyer. What can be computed locally? *SIAM Journal on Computing*, 24(6):1259–1277, 1995. doi:10.1137/S0097539793254571.

[17] Jaroslav Nešetřil. Ramsey theory. In R. L. Graham, M. Grötschel, and L. Lovász, editors, *Handbook of Combinatorics*, volume 2, chapter 25. Elsevier, Amsterdam, 1995.

[18] Alessandro Panconesi and Romeo Rizzi. Some simple distributed algorithms for sparse networks. *Distributed Computing*, 14(2):97–100, 2001. doi:10.1007/PL00008932.

[19] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications, Inc., Mineola, 1998.

[20] David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. SIAM Monographs on Discrete Mathematics and Applications. Society for Industrial and Applied Mathematics, Philadelphia, 2000.

[21] Julius Petersen. Die Theorie der regulären graphs. *Acta Mathematica*, 15(1):193–220, 1891. doi:10.1007/BF02392606.

[22] Valentin Polishchuk and Jukka Suomela. A simple local 3-approximation algorithm for vertex cover. *Information Processing Letters*, 109(12):642–645, 2009. doi:10.1016/j.ipl.2009.02.017. arXiv:0810.2175.

[23] Stanisław P. Radziszowski. Small Ramsey numbers. *The Electronic Journal of Combinatorics*, page Dynamic Survey DS1, 2011. http://www.combinatorics.org/ojs/index.php/eljc/article/view/DS1.

[24] Frank P. Ramsey. On a problem of formal logic. *Proceedings of the London Mathematical Society*, 30:264–286, 1930. doi:10.1112/plms/s2-30.1.264.

[25] Jukka Suomela. Distributed algorithms for edge dominating sets. In *Proc. 29th Annual ACM Symposium on Principles of Distributed Computing (PODC 2010)*, pages 365–374. ACM Press, 2010. doi:10.1145/1835698.1835783.

[26] Vijay V. Vazirani. *Approximation Algorithms*. Springer, Berlin, 2001.

[27] 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.

[28] Mihalis Yannakakis and Fanica Gavril. Edge dominating sets in graphs. *SIAM Journal on Applied Mathematics*, 38(3):364–372, 1980. doi:10.1137/0138030.